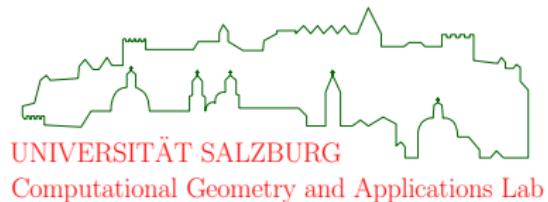


# Computational Geometry (WS 2013/14)

Martin Held

FB Computerwissenschaften  
Universität Salzburg  
A-5020 Salzburg, Austria  
[held@cosy.sbg.ac.at](mailto:held@cosy.sbg.ac.at)

January 24, 2014



# Personalia

**Instructor (VO+PS):** M. Held.

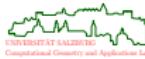
**Email:** `held@cosy.sbg.ac.at`.

**Base-URL:** `http://www.cosy.sbg.ac.at/~held`.

**Office:** Universität Salzburg, Computerwissenschaften, Rm. 1.20,  
Jakob-Haringer Str. 2, 5020 Salzburg-Itzling.

**Phone number (office):** (0662) 8044-6304.

**Phone number (secr.):** (0662) 8044-6328.



**URL of course (VO+PS):** Base-URL/*teaching/geom\_mod/geom\_mod.html*.

**Lecture times (VO):** Friday 9<sup>00</sup>–10<sup>40</sup>.

**Venue (VO):** T01, Computerwissenschaften, Jakob-Haringer Str. 2.

**Lecture times (PS):** Friday 8<sup>00</sup>–8<sup>50</sup>.

**Venue (PS):** T01, Computerwissenschaften, Jakob-Haringer Str. 2.

**URL of course (VO+PS):** Base-URL/*teaching/geom\_mod/geom\_mod.html*.

**Lecture times (VO):** Friday 9<sup>00</sup>–10<sup>40</sup>.

**Venue (VO):** T01, Computerwissenschaften, Jakob-Haringer Str. 2.

**Lecture times (PS):** Friday 8<sup>00</sup>–8<sup>50</sup>.

**Venue (PS):** T01, Computerwissenschaften, Jakob-Haringer Str. 2.

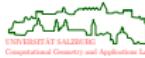
**Note** — PS is graded according to continuous-assessment mode!

## Electronic Slides and Online Material

In addition to these slides, you are encouraged to consult the WWW home-page of this lecture:

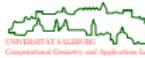
*[http://www.cosy.sbg.ac.at/~held/teaching/compgeo/comp\\_geo.html](http://www.cosy.sbg.ac.at/~held/teaching/compgeo/comp_geo.html).*

In particular, this WWW page contains links to online manuals, slides, and code.



## A Few Words of Warning

I hope that these slides will serve as a practice-minded introduction to various aspects of geometric modeling. I would like to warn you explicitly not to regard these slides as the sole source of information on the topics of my course. It may and will happen that I'll use the lecture for talking about subtle details that need not be covered in these slides! In particular, the slides won't contain all sample calculations, proofs of theorems, demonstrations of algorithms, or solutions to problems posed during my lecture. That is, by making these slides available to you I do not intend to encourage you to attend the lecture on an irregular basis.



## Acknowledgments

These slides are partially based on notes and slides transcribed by various students — most notably Elias Pschernig, Christian Spielberger, Werner Weiser and Franz Wilhelmstötter — for previous courses on “Algorithmische Geometrie”. Some figures were derived from figures originally prepared by students of my lecture “Wissenschaftliche Arbeitstechniken und Präsentation”, while others were taken from papers co-authored with my members of my research group, such as Stefan Huber, Willi Mann, Peter Palfrader. I would like to express my thankfulness to all of them for their help. This revision and extension was carried out by myself, and I am responsible for any errors.

I am also happy to acknowledge that we benefited from material published by colleagues on diverse topics that are partially covered in this lecture. While some of the material used for this lecture was originally presented in traditional-style publications (such as textbooks), some other material has its roots in non-standard publication outlets (such as online documentations, electronic course notes, or user manuals).

Salzburg, September 2013

Martin Held

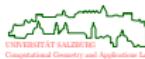


## Legal Fine Print and Disclaimer

To the best of our knowledge, these slides do not violate or infringe upon somebody else's copyrights. If copyrighted material appears in these slides then it was considered to be available in a non-profit manner and as an educational tool for teaching at an academic institution, within the limits of the "fair use" policy. For copyrighted material we strive to give references to the copyright holders (if known). Of course, any trademarks mentioned in these slides are properties of their respective owners.

Please note that these slides are copyrighted. The copyright holder(s) grant you the right to download and print it for your personal use. Any other use, including non-profit instructional use and re-distribution in electronic or printed form of significant portions of it, beyond the limits of "fair use", requires the explicit permission of the copyright holder(s). All rights reserved.

These slides are made available without warrant of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. In no event shall the copyright holder(s) and/or their respective employers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, arising out of or in connection with the use of information provided in these slides.

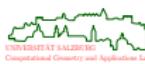


## Recommended Textbooks I

-  M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars.  
*Computational Geometry. Algorithms and Applications.*  
Springer-Verlag, 3rd rev. edition, March 2008. ISBN 978-3540779735.
-  R. Klein.  
*Algorithmische Geometrie.*  
Springer-Verlag, 2nd rev. edition, May 2005. ISBN 978-3540209560.
-  J. O'Rourke.  
*Computational Geometry in C.*  
Cambridge University Press, 2nd edition, 2000. ISBN 978-0521649766.
-  J.-D. Boissonnat and M. Yvinec.  
*Algorithmic Geometry.*  
Cambridge University Press, March 1998. ISBN 978-0521565295.
-  F.P. Preparata and M.I. Shamos.  
*Computational Geometry – An Introduction.*  
Springer-Verlag, 3rd edition, Aug 1993. ISBN 978-0387961316.

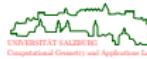
## Recommended Textbooks II

-  S.L. Devadoss and J. O'Rourke.  
*Discrete and Computational Geometry*.  
Princeton University Press, April 2011. ISBN 978-0691145532.
-  F. Aurenhammer, R. Klein and D.-T. Lee.  
*Voronoi Diagrams and Delaunay Triangulations*.  
World Scientific Publ., Aug 2013. ISBN 978-981-4447-63-8.



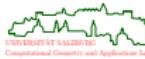
# Table of Content

- 1 Introduction
- 2 Geometric Searching
- 3 Convex Hulls
- 4 Voronoi Diagrams of Points
- 5 Generalized Voronoi Diagrams
- 6 Triangulations
- 7 Robustness Issues



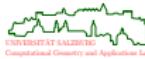
## Introduction

- Motivation
- History
- Asymptotic Notation



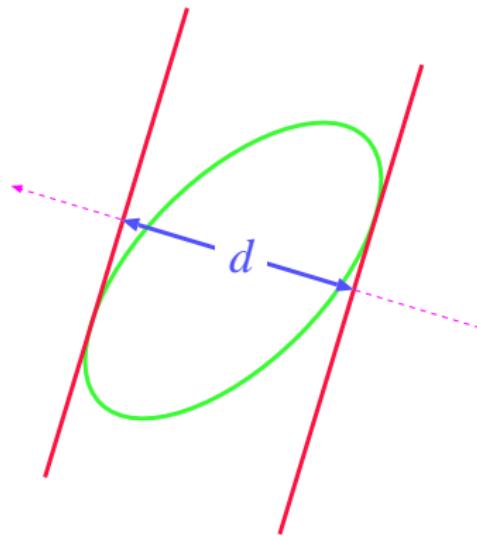
# Introduction

- Motivation
- History
- Asymptotic Notation



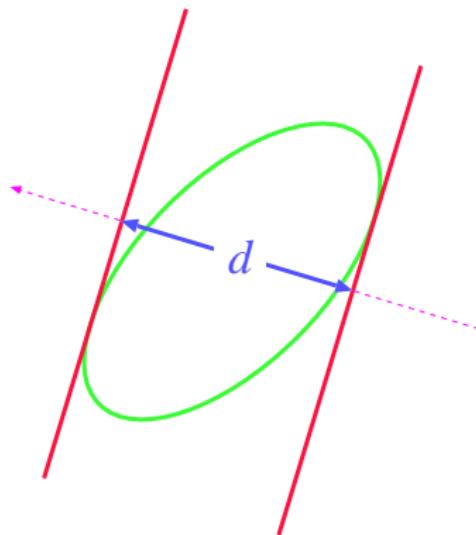
## Motivation: Roundness of a Convex Shape

- Define the width of a planar shape with respect to a direction vector as the minimum distance  $d$  of its two parallel lines of support normal to the direction vector such that the shape is enclosed.



## Motivation: Roundness of a Convex Shape

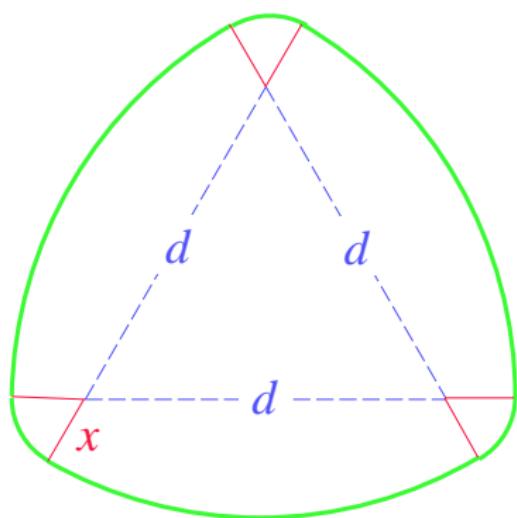
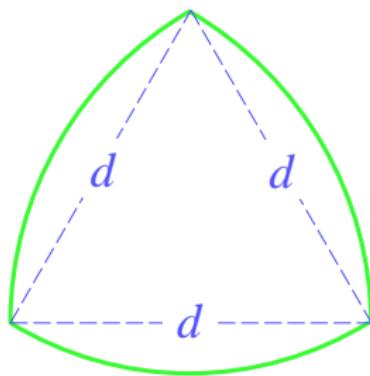
- Define the width of a planar shape with respect to a direction vector as the minimum distance  $d$  of its two parallel lines of support normal to the direction vector such that the shape is enclosed.



- Question: Can we conclude that the shape resembles a circle of diameter  $d$  if a sufficiently large number of caliper probes all yield  $d$  as width?

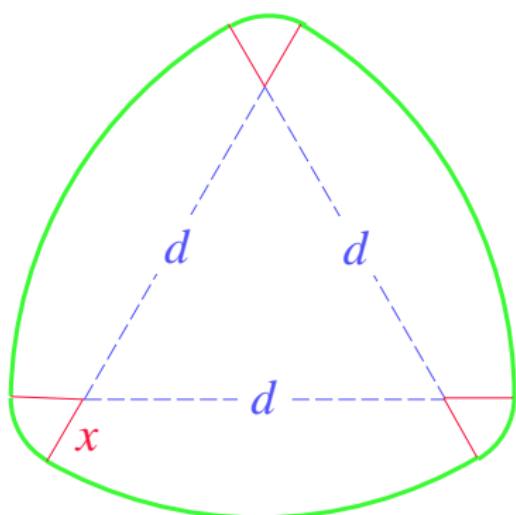
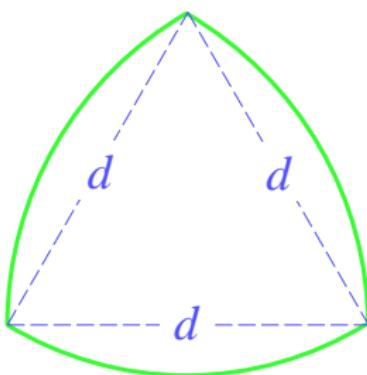
## Roundness of a Convex Shape

- Answer: No!! Even an infinite number of caliper probes all would yield a constant width  $d$  for a Reuleaux triangle!



## Roundness of a Convex Shape

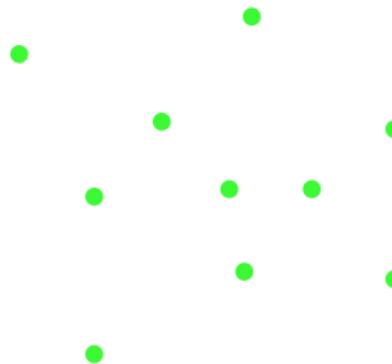
- Answer: No!! Even an infinite number of caliper probes all would yield a constant width  $d$  for a Reuleaux triangle!



- Apparently, three caliper probes were applied when checking parts of the Challenger's solid-fuel booster rockets for roundness. (R. Feynman: "What do you care what other people think?", 1988.)

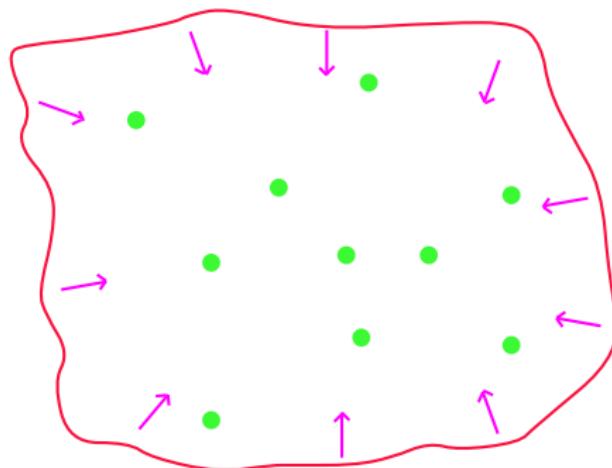
## Motivation: Convex Hull

- Given is a set  $S$  of  $n$  points in  $\mathbb{R}^2$ .



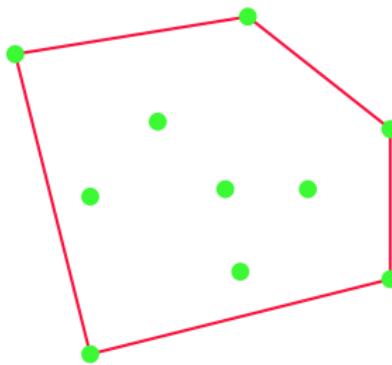
## Motivation: Convex Hull

- Given is a set  $S$  of  $n$  points in  $\mathbb{R}^2$ .
- Question: How efficiently can we determine the convex hull of  $S$ ?



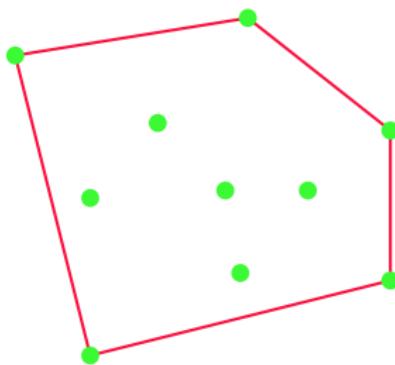
## Motivation: Convex Hull

- Given is a set  $S$  of  $n$  points in  $\mathbb{R}^2$ .
- Question: How efficiently can we determine the convex hull of  $S$ ?



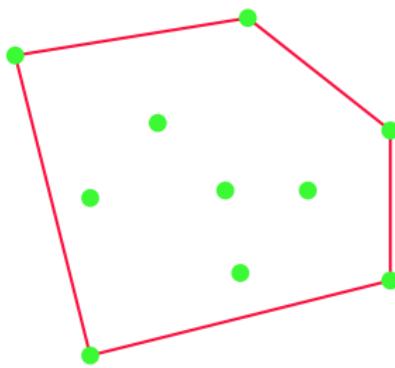
## Motivation: Convex Hull

- Answer: The convex hull of  $S$  can be computed in  $O(n \log n)$  steps.



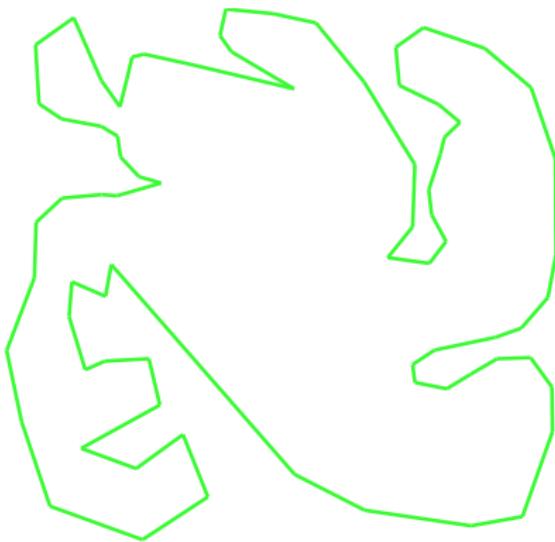
## Motivation: Convex Hull

- Answer: The convex hull of  $S$  can be computed in  $O(n \log n)$  steps.
- Lower bound: In the worst case,  $\Omega(n \log n)$  steps will be necessary.



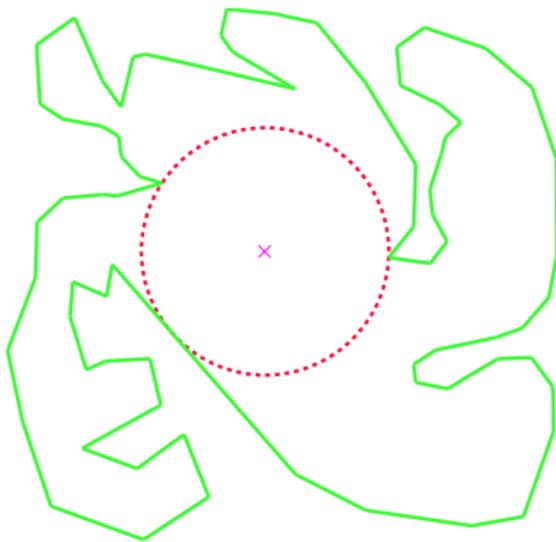
## Motivation: Maximum Inscribed Circle

- Given is a simple polygon  $\mathcal{P}$ . A circle is called inscribed to  $\mathcal{P}$  if it lies completely inside of  $\mathcal{P}$ .



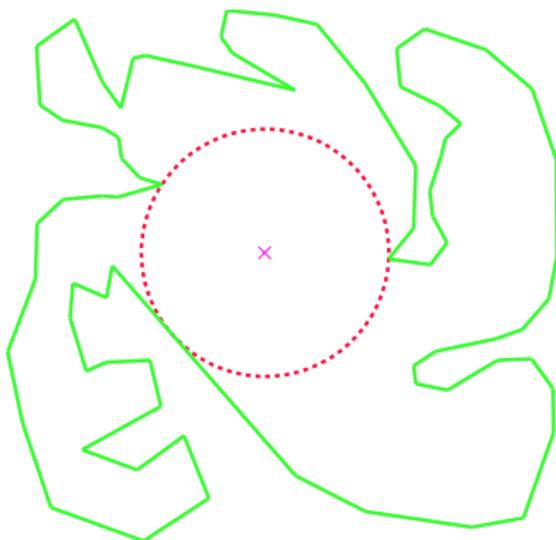
## Motivation: Maximum Inscribed Circle

- Given is a simple polygon  $\mathcal{P}$ . A circle is called inscribed to  $\mathcal{P}$  if it lies completely inside of  $\mathcal{P}$ .
- Question: How efficiently can we determine a maximum inscribed circle?



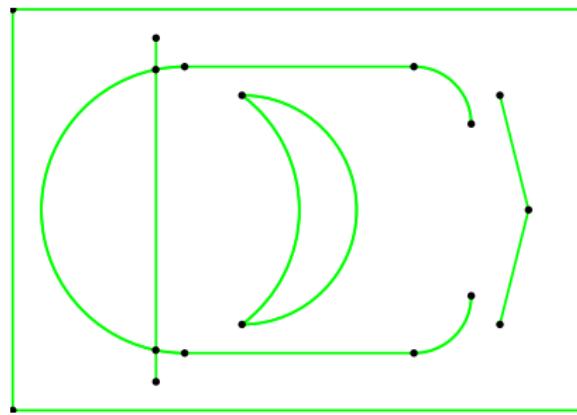
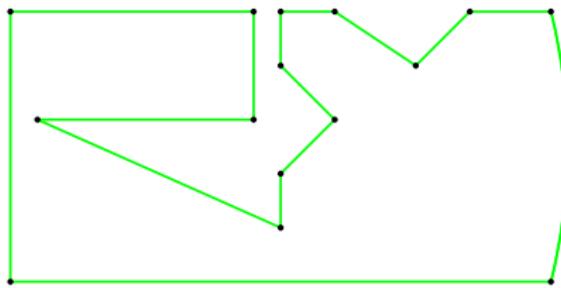
## Maximum Inscribed Circle

- Answer: In theory, a maximum inscribed circle can be computed in time linear in the number  $n$  of vertices of  $\mathcal{P}$ .
- Practical algorithms/implementations running in  $O(n \log n)$  time are available.



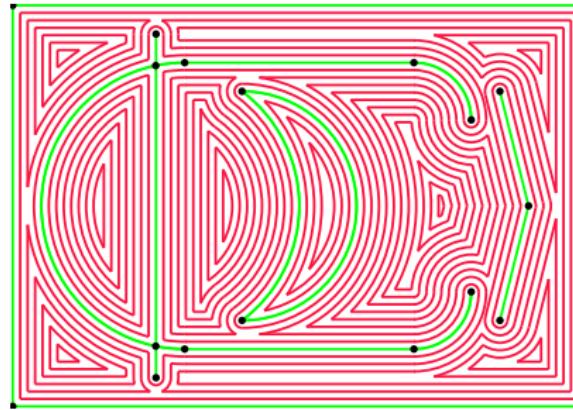
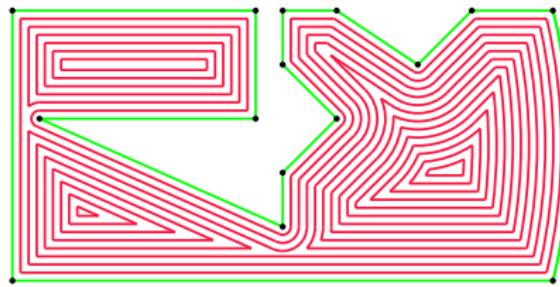
## Motivation: Computation of Offset Patterns

- How can we compute offset patterns reliably and efficiently?



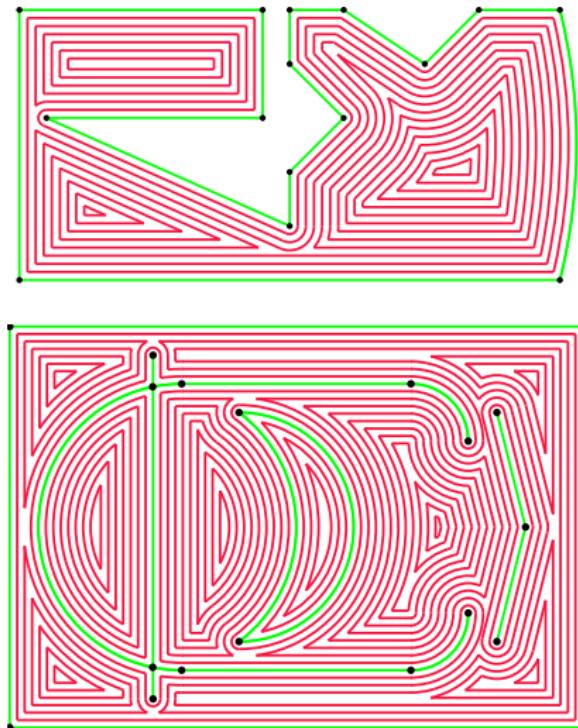
## Motivation: Computation of Offset Patterns

- How can we compute offset patterns reliably and efficiently?



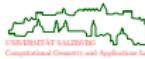
## Motivation: Computation of Offset Patterns

- [Persson 1978, Held 1991]: If the Voronoi diagram of the input is known, then all offset curves of one offset can be determined in linear time.



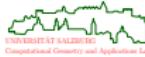
# Introduction

- Motivation
- History
- Asymptotic Notation



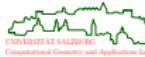
# History of Computational Geometry

- 1000BC: length, area and volume are known for simple objects (cube, box, cylinder).



# History of Computational Geometry

- 1000BC: length, area and volume are known for simple objects (cube, box, cylinder).
- Antiquity: move from empirical mathematics to deductive mathematics.
- Thales of Milet ( $\approx$  600BC): proved(!) that the two base angles of an isosceles triangle are identical.



# History of Computational Geometry

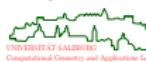
- 1000BC: length, area and volume are known for simple objects (cube, box, cylinder).
- Antiquity: move from empirical mathematics to deductive mathematics.
- Thales of Milet ( $\approx$  600BC): proved(!) that the two base angles of an isosceles triangle are identical.
- Euclid of Alexandria ( $\approx$  300BC): “The Elements”.
  - definitions,
  - five postulates,
  - five axioms,
  - 115 propositions.

English translation on the subsequent slides by Sir Thomas Heath, “The Thirteen Books of Euclid’s Elements”, Dover Publ., 1956.



# Euclid's Elements: Definitions and Postulates

- Definitions: e.g.,
  - A point is that which has no part.
  - A circle is a plane figure contained by one line such that all the straight lines falling from one point among those lying within the figure are equal to one another;
  - And the point is called the center of the circle.
- Five postulates:
  - It is possible to draw a straight line from any point to any point.
  - It is possible to continue a finite straight line continuously in a straight line.
  - It is possible to describe a circle with any center and radius.
  - All right angles are equal to one another.
  - If a straight line falling on two straight lines make the interior angles on the same side less than two right angles, the two straight lines, if continued indefinitely, meet on that side on which are the angles less than the two right angles.
- Fifth postulate (“parallel postulate”) rephrased: Given a line  $\ell$  and a point  $p$  not on the line, there is exactly one line through  $p$  which does not intersect  $\ell$ . Other postulates consistent with Postulates 1–4 are possible: Riemann, Lobačevskij.

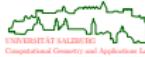


# Euclid's Elements: Axioms and Propositions

- Five axioms (“common notions”):
  - Things which are equal to the same thing are also equal to one another.
  - If equals be added to equals, the wholes are equal.
  - If equals be subtracted from equals, the remainders are equal.
  - Things which coincide with one another are equal to one another.
  - The whole is greater than the part.
- 115 Propositions: e.g.,
  - On a given finite straight line to construct an equilateral triangle.
  - In a given circle to inscribe a fifteen-angled figure which shall be both equilateral and equiangular.
- His “Euclidean constructions” used in the propositions introduced a highly stylized scheme consisting of algorithm *and* proof.

## History

- L. da Vinci (1452–1519) and others: introduced perspective and projective geometry.



## History

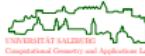
- L. da Vinci (1452–1519) and others: introduced perspective and projective geometry.
- R. Descartes (1596–1650) and P. de Fermat: coordinates and the foundation of analytical geometry.

## History

- L. da Vinci (1452–1519) and others: introduced perspective and projective geometry.
- R. Descartes (1596–1650) and P. de Fermat: coordinates and the foundation of analytical geometry.
- B. Riemann (1826–1866): differential geometry.
- H. Poincaré (1854–1912) and D. Hilbert (1862–1943): axiom-based mathematics, proved consistency of axioms.

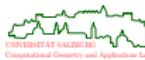
## History

- D. Knuth: “The Art of Computer Programming” published 1968–1973.



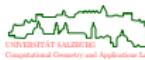
## History

- D. Knuth: “The Art of Computer Programming” published 1968–1973.
- Bézier, Forrest, Riesenfeld: modeling of spline curves and surfaces called “computational geometry”.
- Minsky and Papert: book entitled “Perceptrons” with chapter on “computational geometry”: which geometric properties of a figure can be recognized with neural networks?



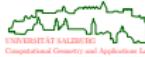
## History

- D. Knuth: “The Art of Computer Programming” published 1968–1973.
- Bézier, Forrest, Riesenfeld: modeling of spline curves and surfaces called “computational geometry”.
- Minsky and Papert: book entitled “Perceptrons” with chapter on “computational geometry”: which geometric properties of a figure can be recognized with neural networks?
- M.I. Shamos: PhD thesis “Computational Geometry” (Yale, 1978).



# Introduction

- Motivation
- History
- Asymptotic Notation



# How Can We Compare the Growth Rate of Functions?

- Let's consider  $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$  with  $f(n) := n$  and  $g(n) := 9n + 20$ .
- We get for all  $n \in \mathbb{N}$  with  $n \geq 20$

$$g(n) = 9n + 20 \leq 9n + n = 10n = 10f(n), \quad \text{that is } g(n) \leq 10f(n).$$

- Also for all  $n \in \mathbb{N}$
- $$f(n) \leq g(n).$$

# How Can We Compare the Growth Rate of Functions?

- Let's consider  $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$  with  $f(n) := n$  and  $g(n) := 9n + 20$ .
- We get for all  $n \in \mathbb{N}$  with  $n \geq 20$

$$g(n) = 9n + 20 \leq 9n + n = 10n = 10f(n), \quad \text{that is } \underbrace{g(n)}_{\text{in purple}} \leq 10f(n).$$

- Also for all  $n \in \mathbb{N}$

$$\underbrace{f(n)}_{\text{in purple}} \leq g(n).$$

Thus, we have

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

$\left\{ \begin{array}{l} \text{for all } n \geq n_0 \\ \text{where } n_0 := 20, \\ c_1 := 1, c_2 := 10. \end{array} \right.$



# How Can We Compare the Growth Rate of Functions?

- Let's consider  $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$  with  $f(n) := n$  and  $g(n) := 9n + 20$ .
- We get for all  $n \in \mathbb{N}$  with  $n \geq 20$

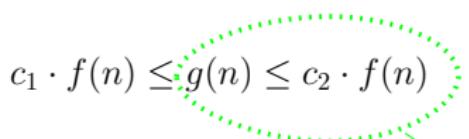
$$g(n) = 9n + 20 \leq 9n + n = 10n = 10f(n), \quad \text{that is } g(n) \leq 10f(n).$$

- Also for all  $n \in \mathbb{N}$

$$f(n) \leq g(n).$$

Thus, we have

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

  $\left\{ \begin{array}{l} \text{for all } n \geq n_0 \\ \text{where } n_0 := 20, \\ c_1 := 1, c_2 := 10. \end{array} \right.$

*g grows at most as fast as  $c_2 \cdot f$*

*f is an asymptotic upper bound on g*

*we'll say that  $g \in O(f)$*

# How Can We Compare the Growth Rate of Functions?

- Let's consider  $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$  with  $f(n) := n$  and  $g(n) := 9n + 20$ .
- We get for all  $n \in \mathbb{N}$  with  $n \geq 20$

$$g(n) = 9n + 20 \leq 9n + n = 10n = 10f(n), \quad \text{that is } g(n) \leq 10f(n).$$

- Also for all  $n \in \mathbb{N}$

$$f(n) \leq g(n).$$

Thus, we have

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

$\left\{ \begin{array}{l} \text{for all } n \geq n_0 \\ \text{where } n_0 := 20, \\ c_1 := 1, c_2 := 10. \end{array} \right.$

$g$  grows at least as fast as  $c_1 \cdot f$

$f$  is an asymptotic lower bound on  $g$

we'll say that  $g \in \Omega(f)$



# How Can We Compare the Growth Rate of Functions?

- Let's consider  $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$  with  $f(n) := n$  and  $g(n) := 9n + 20$ .
- We get for all  $n \in \mathbb{N}$  with  $n \geq 20$

$$g(n) = 9n + 20 \leq 9n + n = 10n = 10f(n), \quad \text{that is } g(n) \leq 10f(n).$$

- Also for all  $n \in \mathbb{N}$
- $$f(n) \leq g(n).$$

Thus, we have

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

$\left\{ \begin{array}{l} \text{for all } n \geq n_0 \\ \text{where } n_0 := 20, \\ c_1 := 1, c_2 := 10. \end{array} \right.$

$g$  grows at least as fast as  $c_1 \cdot f$   
 $f$  is an asymptotic lower bound on  $g$   
we'll say that  $g \in \Omega(f)$

$g$  grows at most as fast as  $c_2 \cdot f$   
 $f$  is an asymptotic upper bound on  $g$   
we'll say that  $g \in O(f)$

$g$  has same growth rate as  $f$   
we'll say that  $g \in \Theta(f)$

# Asymptotic Notation: Big-O

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

$\left\{ \begin{array}{l} \text{for all } n \geq n_0 \text{ and} \\ \text{fixed } c_1, c_2 \in \mathbb{R}^+. \end{array} \right.$

$g$  grows at most as fast as  $c_2 \cdot f$

$f$  is an asymptotic upper bound on  $g$

we'll say that  $g \in O(f)$

## Definition 1 (Big-O, Dt.: GroB-O)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $O(f)$  is defined as

$$O(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_2 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq c_2 \cdot f(n)\}.$$

# Asymptotic Notation: Big-O

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

$\left\{ \begin{array}{l} \text{for all } n \geq n_0 \text{ and} \\ \text{fixed } c_1, c_2 \in \mathbb{R}^+. \end{array} \right.$

$g$  grows at most as fast as  $c_2 \cdot f$

$f$  is an asymptotic upper bound on  $g$

we'll say that  $g \in O(f)$

## Definition 1 (Big-O, Dt.: GroB-O)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $O(f)$  is defined as

$$O(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_2 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq c_2 \cdot f(n)\}.$$

- Note:  $O(f)$  is a set of functions!
- Definitions of the form

$$O(f(n)) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_2 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq c_2 \cdot f(n)\}$$

are a (wide-spread) formal nonsense.

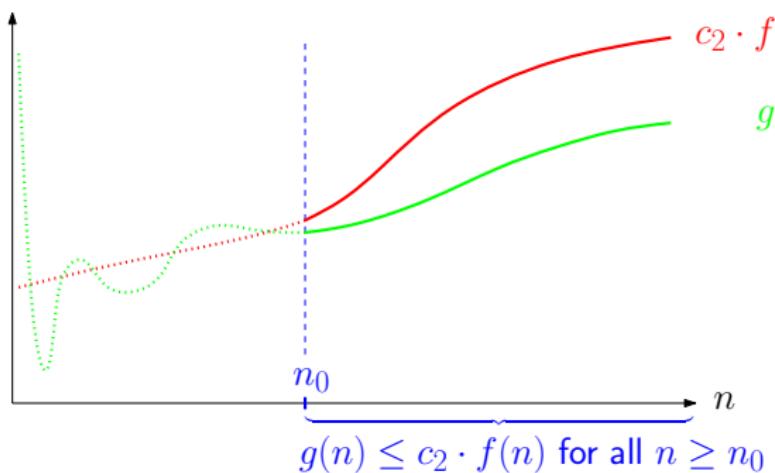


# Graphical Illustration of $O(f)$

## Definition 1 (Big-O, Dt.: Gro $\beta$ -O)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $O(f)$  is defined as

$$O(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_2 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq c_2 \cdot f(n)\}.$$



## Sample Proof of $g \in O(f)$

- We prove  $g \in O(f)$  for  $f(n) := \frac{1}{4}n^2 + 3n - \sqrt{n}$  and  $g(n) := n^2$ .

*Proof:*

- We observe that  $3n - \sqrt{n} \geq 0$  for all  $n \in \mathbb{N}$ , and get for all  $n \in \mathbb{N}$

$$g(n) = n^2 = 4\left(\frac{1}{4}n^2\right) \leq 4\left(\frac{1}{4}n^2 + 3n - \sqrt{n}\right) = 4f(n), \quad \text{that is } g(n) \leq 4f(n).$$

## Sample Proof of $g \in O(f)$

- We prove  $g \in O(f)$  for  $f(n) := \frac{1}{4}n^2 + 3n - \sqrt{n}$  and  $g(n) := n^2$ .

*Proof:*

- We observe that  $3n - \sqrt{n} \geq 0$  for all  $n \in \mathbb{N}$ , and get for all  $n \in \mathbb{N}$

$$g(n) = n^2 = 4\left(\frac{1}{4}n^2\right) \leq 4\left(\frac{1}{4}n^2 + 3n - \sqrt{n}\right) = 4f(n), \quad \text{that is } g(n) \leq 4f(n).$$

- Thus,  $g \in O(f)$  with  $c_2 := 4$  and  $n_0 := 1$ .

□

## Sample Proof of $g \in O(f)$

- We prove  $g \in O(f)$  for  $f(n) := \frac{1}{4}n^2 + 3n - \sqrt{n}$  and  $g(n) := n^2$ .

*Proof:*

- We observe that  $3n - \sqrt{n} \geq 0$  for all  $n \in \mathbb{N}$ , and get for all  $n \in \mathbb{N}$

$$g(n) = n^2 = 4\left(\frac{1}{4}n^2\right) \leq 4\left(\frac{1}{4}n^2 + 3n - \sqrt{n}\right) = 4f(n), \quad \text{that is } g(n) \leq 4f(n).$$

- Thus,  $g \in O(f)$  with  $c_2 := 4$  and  $n_0 := 1$ .

□

- Note that there is no need to try to obtain the smallest-possible values for  $n_0$  and  $c_2$  when establishing  $g \in O(f)$ .

## Sample Proof of $g \in O(f)$

- We prove  $g \in O(f)$  for  $f(n) := \frac{1}{4}n^2 + 3n - \sqrt{n}$  and  $g(n) := n^2$ .

*Proof:*

- We observe that  $3n - \sqrt{n} \geq 0$  for all  $n \in \mathbb{N}$ , and get for all  $n \in \mathbb{N}$

$$g(n) = n^2 = 4(\frac{1}{4}n^2) \leq 4(\frac{1}{4}n^2 + 3n - \sqrt{n}) = 4f(n), \quad \text{that is } g(n) \leq 4f(n).$$

- Thus,  $g \in O(f)$  with  $c_2 := 4$  and  $n_0 := 1$ .

□

- Note that there is no need to try to obtain the smallest-possible values for  $n_0$  and  $c_2$  when establishing  $g \in O(f)$ .
- Can we also prove  $h \in O(f)$  for  $h(n) := n^3$ ? We get, for all  $n \in \mathbb{N}$ ,

$$\frac{h(n)}{f(n)} = \frac{n^3}{\frac{1}{4}n^2 + 3n - \sqrt{n}} = \frac{n}{\frac{1}{4} + \frac{3}{n} - \frac{1}{n\sqrt{n}}} \geq \frac{1}{4}n.$$

- Thus,  $\frac{h(n)}{f(n)}$  grows unboundedly as  $n$  grows, while it ought to be bounded by some constant  $c_2$  for all  $n \geq n_0$ , for some fixed  $n_0 \in \mathbb{N}$ . We conclude that  $h \notin O(f)$ .



## Asymptotic Notation: Big-Omega

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

$\left\{ \begin{array}{l} \text{for all } n \geq n_0 \text{ and} \\ \text{fixed } c_1, c_2 \in \mathbb{R}^+. \end{array} \right.$



$g$  grows at least as fast as  $c_1 \cdot f$

$f$  is an asymptotic lower bound on  $g$

we'll say that  $g \in \Omega(f)$

### Definition 2 (Big-Omega, Dt.: Groß-Omega)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $\Omega(f)$  is defined as

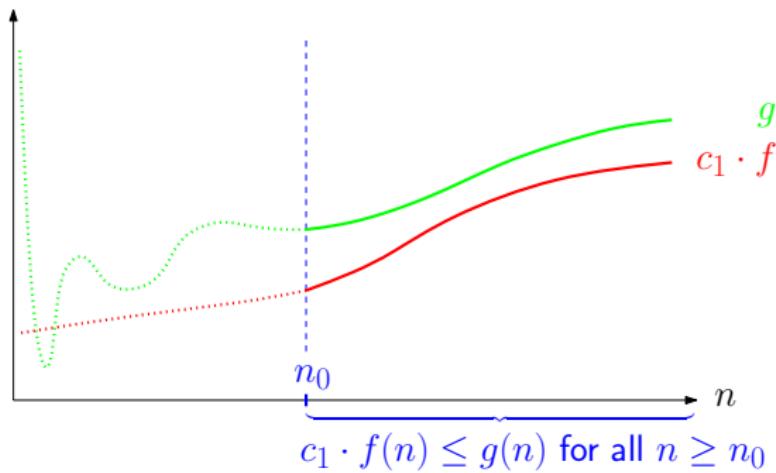
$$\Omega(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad c_1 \cdot f(n) \leq g(n)\}.$$

## Graphical Illustration of $\Omega(f)$

### Definition 2 (Big-Omega, Dt.: Groß-Omega)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $\Omega(f)$  is defined as

$$\Omega(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad c_1 \cdot f(n) \leq g(n)\}.$$



# Asymptotic Notation: Big-Theta

$$\underbrace{c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)}_{\text{$g$ has same growth rate as $f$}} \quad \left\{ \begin{array}{l} \text{for all } n \geq n_0 \text{ and} \\ \text{fixed } c_1, c_2 \in \mathbb{R}^+. \end{array} \right.$$

*g* has same growth rate as *f*

we'll say that  $g \in \Theta(f)$

## Definition 3 (Big-Theta, Dt.: Groß-Theta)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $\Theta(f)$  is defined as

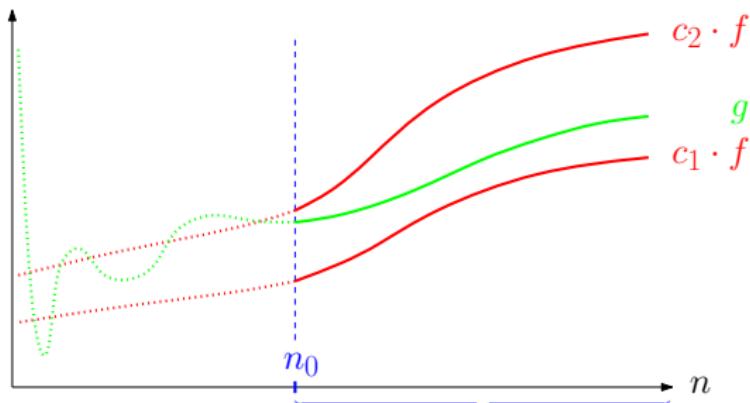
$$\Theta(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0$$
$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}.$$

# Graphical Illustration of $\Theta(f)$

## Definition 3 (Big-Theta, Dt.: Groß-Theta)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $\Theta(f)$  is defined as

$$\Theta(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \\ c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}.$$



$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n) \text{ for all } n \geq n_0$$

$$\text{which is equivalent to } c_1 \leq \frac{g(n)}{f(n)} \leq c_2 \text{ for all } n \geq n_0$$

# Asymptotic Notation: Small-Oh

## Definition 4 (Small-Oh, Dt.: Klein-O)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $o(f)$  is defined as

$$o(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) < c \cdot f(n)\}.$$



## Definition 4 (Small-Oh, Dt.: Klein-O)

Let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ . Then the set  $o(f)$  is defined as

$$o(f) := \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) < c \cdot f(n)\}.$$

- It is trivial to extend Definitions 1–4 such that  $\mathbb{N}_0$  rather than  $\mathbb{N}$  is taken as the base set.
- Similarly, we can replace  $\mathbb{R}^+$  by  $\mathbb{R}_0^+$  (or even  $\mathbb{R}$ ) provided that all functions are eventually positive.
- The same comments apply to the subsequent slides.

## Geometric Searching

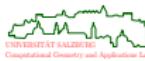
- Introduction
- Range Searching
  - Report Query
  - kd-Tree
- Point Inclusion
  - Brute-Force Method
  - Point-in-Polygon Query
  - Slab Method
  - Paradigm: Plane Sweep
  - Chain Method
  - Triangulation Refinement Technique

## 2 Geometric Searching

- Introduction
- Range Searching
- Point Inclusion

# Introduction to Geometric Searching

**Point-Inclusion Query:** In which cell does a query point lie?



# Introduction to Geometric Searching

**Point-Inclusion Query:** In which cell does a query point lie?

**Range Searching:**

- **Report Query:** Which points are within a query object (rectangle, circle, ...)?
- **Count Query:** Only the number of points within an object matters.

# Introduction to Geometric Searching

**Point-Inclusion Query:** In which cell does a query point lie?

**Range Searching:**

- **Report Query:** Which points are within a query object (rectangle, circle, ...)?
  - **Count Query:** Only the number of points within an object matters.
- 
- Another way to distinguish geometric searching queries:

**Single-Shot Query:** Only one query per data set.

**Repetitive-Mode Query:** Many queries per data set; preprocessing may make sense.



# Introduction to Geometric Searching

**Point-Inclusion Query:** In which cell does a query point lie?

**Range Searching:**

- **Report Query:** Which points are within a query object (rectangle, circle, ...)?
  - **Count Query:** Only the number of points within an object matters.
- 
- Another way to distinguish geometric searching queries:

**Single-Shot Query:** Only one query per data set.

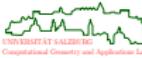
**Repetitive-Mode Query:** Many queries per data set; preprocessing may make sense.

- The complexity of a query is determined relative to four cost measures:
  - query time,
  - preprocessing time,
  - memory consumption,
  - update time (in the case of dynamic data sets).



## 2 Geometric Searching

- Introduction
- Range Searching
  - Report Query
  - kd-Tree
- Point Inclusion

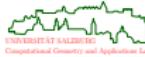


## Range Searching: Report Query

- Given a set of  $n$  points in  $\mathbb{R}^k$ , we want to find the points which are within a specific (hyper-)rectangle.

## Range Searching: Report Query

- Given a set of  $n$  points in  $\mathbb{R}^k$ , we want to find the points which are within a specific (hyper-)rectangle.
- Case  $k = 1$ : The rectangle is an interval.
  - As preprocessing we sort the points. This needs  $O(n \log n)$  time.



## Range Searching: Report Query

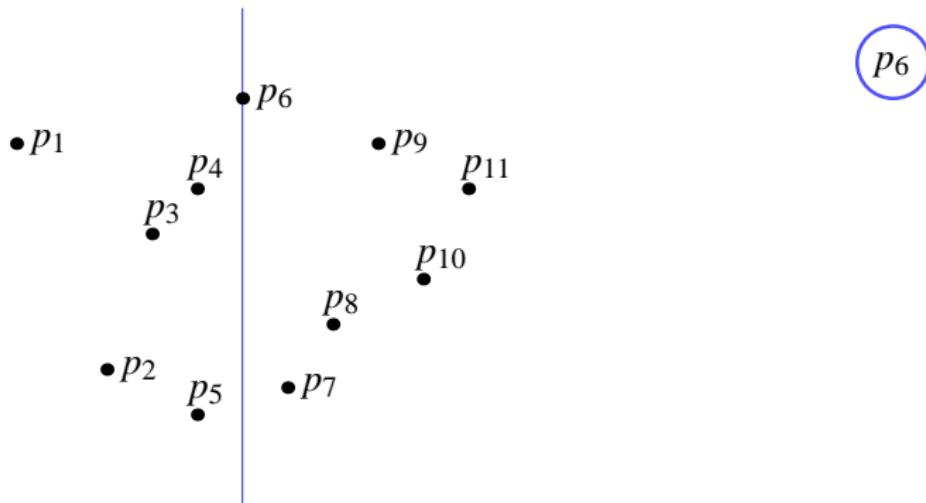
- Given a set of  $n$  points in  $\mathbb{R}^k$ , we want to find the points which are within a specific (hyper-)rectangle.
- Case  $k = 1$ : The rectangle is an interval.
  - As preprocessing we sort the points. This needs  $O(n \log n)$  time.
  - A query is solved by a binary search which needs  $O(\log n + m)$  time, where  $m$  is the number of points returned.

## Range Searching: Report Query

- Given a set of  $n$  points in  $\mathbb{R}^k$ , we want to find the points which are within a specific (hyper-)rectangle.
- Case  $k = 1$ : The rectangle is an interval.
  - As preprocessing we sort the points. This needs  $O(n \log n)$  time.
  - A query is solved by a binary search which needs  $O(\log n + m)$  time, where  $m$  is the number of points returned.
- Case  $k \geq 2$ : The goal is to “extend” binary search to higher dimensions.

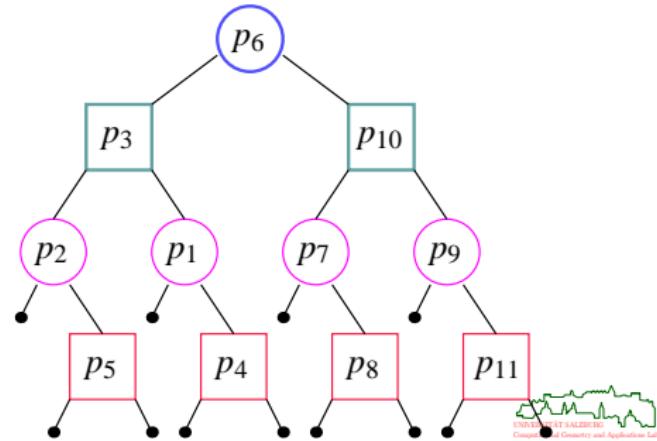
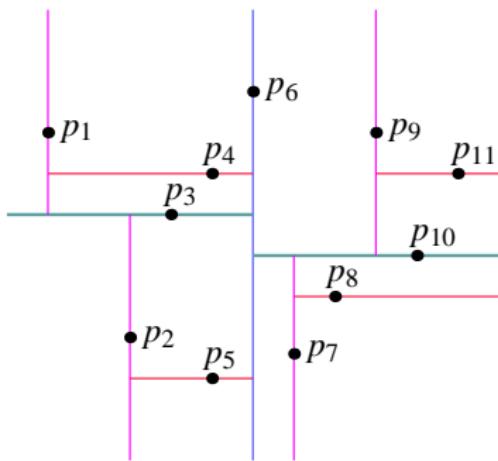
## Range Searching: 2D kd-Tree

- For points  $p_1, \dots, p_n$  in  $\mathbb{R}^2$  we build a 2D kd-tree (“two-dimensional binary search tree”) as the preprocessing:
  - We start by finding the median  $p_m$  of the points with respect to their  $x$ -coordinates. (W.l.o.g.: “general position assumed!”)
  - The point  $p_m$  becomes the root of the tree; it is labeled “vertical”.
  - We divide the plane by a vertical straight line through  $p_m$  into two half-planes.

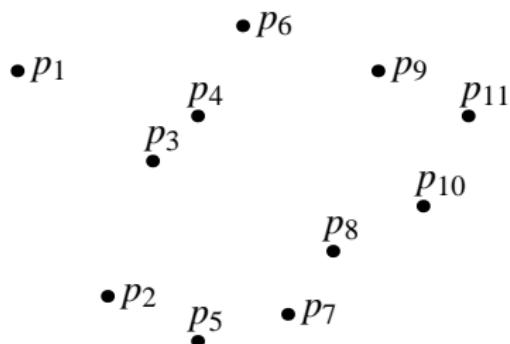


## Range Searching: 2D kd-Tree

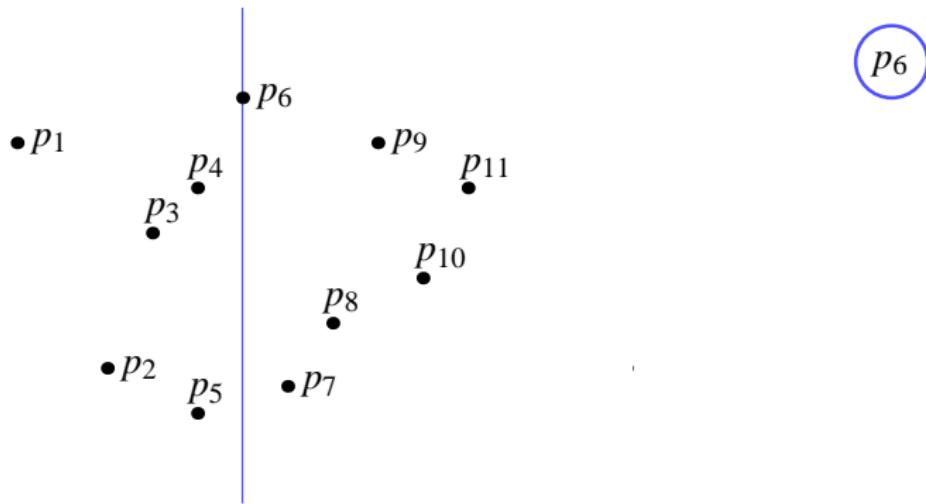
- For points  $p_1, \dots, p_n$  in  $\mathbb{R}^2$  we build a 2D kd-tree as the preprocessing:
  - Within each half-plane we find the medians with respect to the  $y$ -coordinates of the respective points.
  - These two points are called “horizontal” nodes and become the left and the right child of the root.
  - The recursive subdivision continues until all points form nodes of the tree: for “vertical” nodes we use a vertical line, and a horizontal line for “horizontal” nodes.



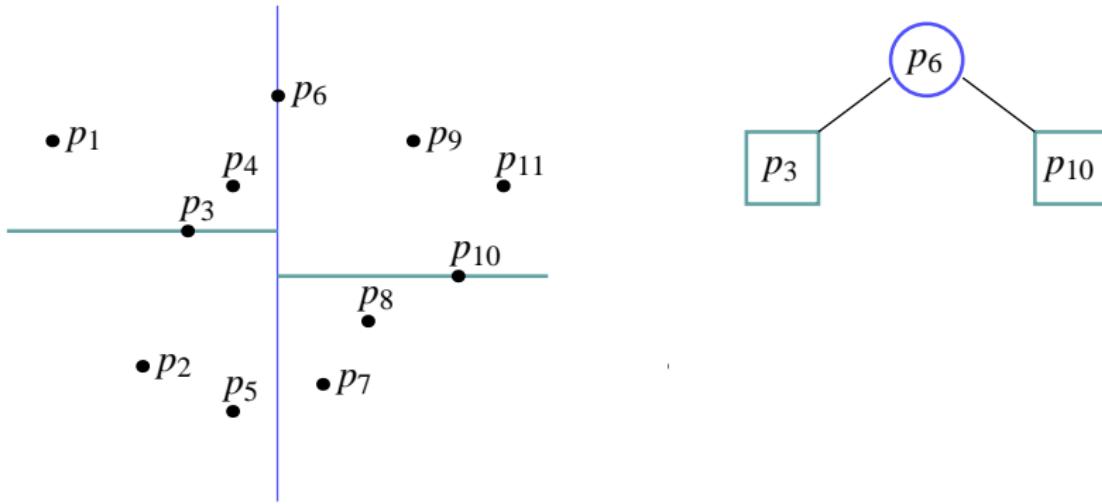
## Sample Construction of 2D kd-Tree



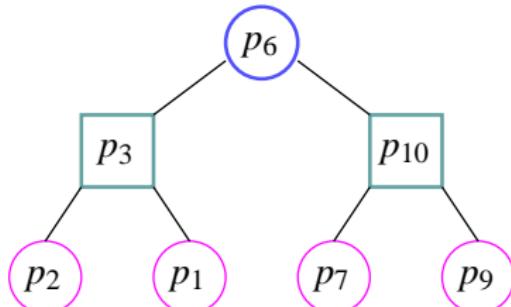
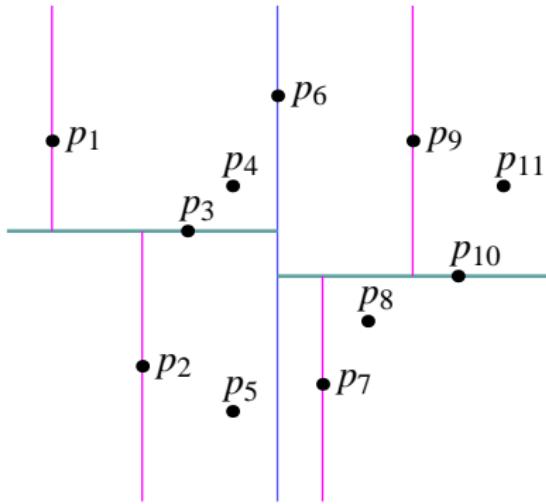
## Sample Construction of 2D kd-Tree



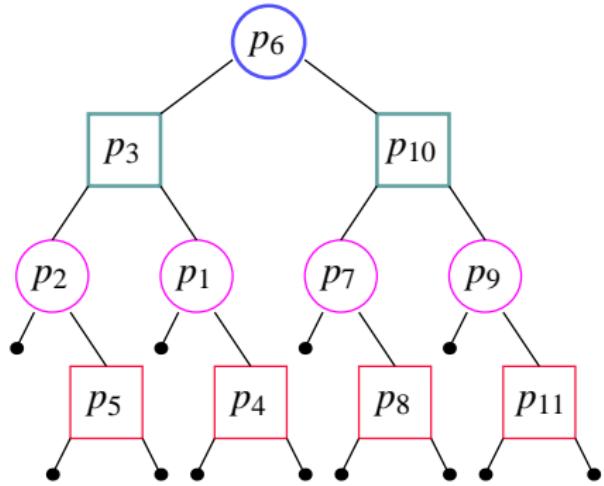
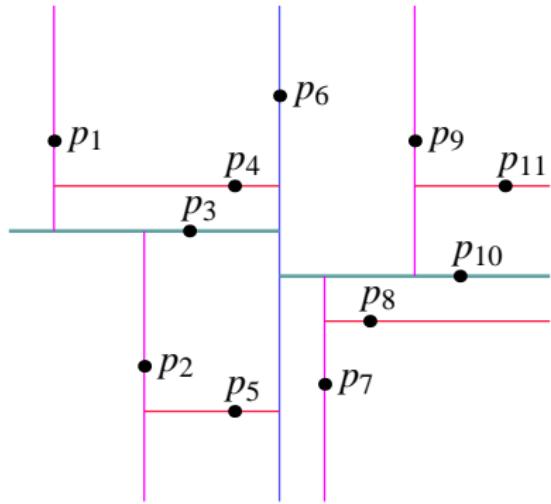
## Sample Construction of 2D kd-Tree



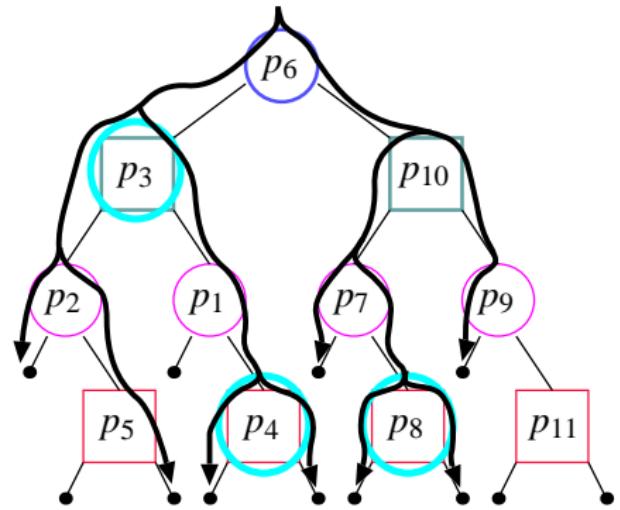
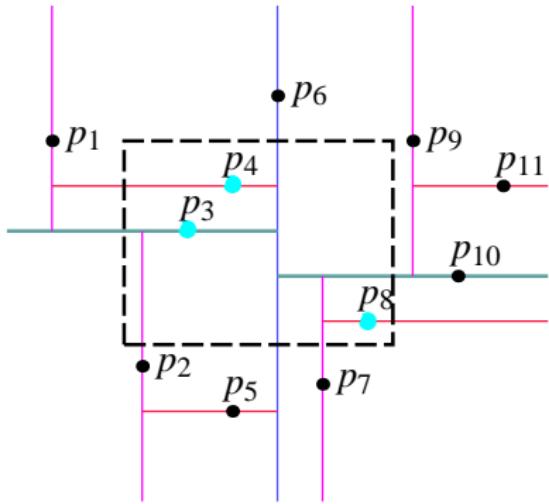
## Sample Construction of 2D kd-Tree



## Sample Construction of 2D kd-Tree



## Range Search: Traversing a 2D kd-Tree



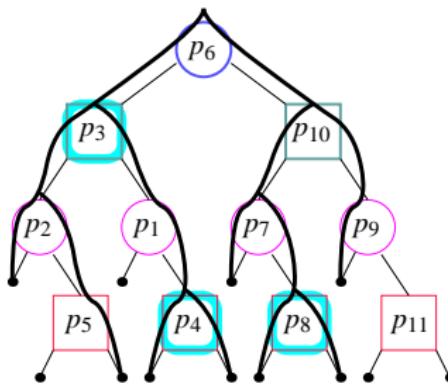
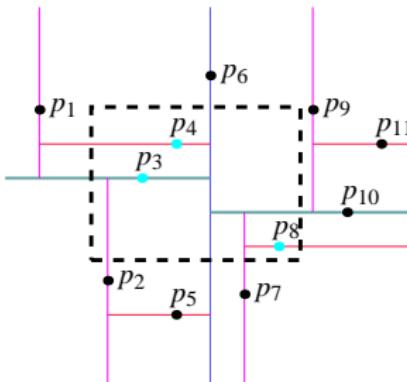
## Range Searching: Traversing a 2D kd-Tree

- We denote by

- $v$  the currently tested node,
- $D = [x_1, x_2] \times [y_1, y_2]$  the rectangle,
- $t(v)$  the type of the node (which can be “vertical” or “horizontal”),
- $P(v)$  the point which corresponds to the node  $v$ ,
- $M(v)$  the  $x$ -coordinate of  $P(v)$  if  $v$  is a “vertical” node, else the  $y$ -coordinate,
- $l(v)$  the left child of node  $v$ ,
- $r(v)$  the right child of node  $v$ .

# Range Search: Traversing a 2D kd-Tree

```
procedure search( $v$ ,  $D$ )
if  $t(v)$  is “vertical” then
     $[a, b] := [x_1, x_2]$ 
else
     $[a, b] := [y_1, y_2]$ 
end if
if  $a \leq M(v) \leq b$  then
    if  $P(v) \in D$  then
        report  $P(v)$ 
    end if
end if
if  $v$  is no leaf then
    if  $a < M(v)$  then
        search( $l(v)$ ,  $D$ )
    end if
    if  $M(v) < b$  then
        search( $r(v)$ ,  $D$ )
    end if
end if
```



## Theorem 5

Range searching based on a 2-dimensional kd-tree needs  $O(n \log n)$  preprocessing time, with  $O(n)$  space complexity. A query can be carried out in  $O(\sqrt{n} + m)$  time, where  $m$  is the number of nodes visited.

# Complexity of Range Searching Based on a kd-Tree

## Theorem 5

Range searching based on a 2-dimensional kd-tree needs  $O(n \log n)$  preprocessing time, with  $O(n)$  space complexity. A query can be carried out in  $O(\sqrt{n} + m)$  time, where  $m$  is the number of nodes visited.

- This complexity does not change if no pre-sorting is carried out and a linear-time algorithm for median finding is used for identifying the nodes of the tree. (It just makes the implementation more tedious.)

# Complexity of Range Searching Based on a kd-Tree

## Theorem 5

Range searching based on a 2-dimensional kd-tree needs  $O(n \log n)$  preprocessing time, with  $O(n)$  space complexity. A query can be carried out in  $O(\sqrt{n} + m)$  time, where  $m$  is the number of nodes visited.

- This complexity does not change if no pre-sorting is carried out and a linear-time algorithm for median finding is used for identifying the nodes of the tree. (It just makes the implementation more tedious.)
- More efficient methods are known. But kd-trees are a very versatile tool!

# Complexity of Range Searching Based on a kd-Tree

## Theorem 5

Range searching based on a 2-dimensional kd-tree needs  $O(n \log n)$  preprocessing time, with  $O(n)$  space complexity. A query can be carried out in  $O(\sqrt{n} + m)$  time, where  $m$  is the number of nodes visited.

- This complexity does not change if no pre-sorting is carried out and a linear-time algorithm for median finding is used for identifying the nodes of the tree. (It just makes the implementation more tedious.)
- More efficient methods are known. But kd-trees are a very versatile tool!
- For a range search in  $\mathbb{R}^k$  we split the space alternatingly by straight lines ( $k = 2$ ), planes ( $k = 3$ ), or hyper-planes (for  $k \geq 4$ ).

# Complexity of Range Searching Based on a kd-Tree

## Theorem 5

Range searching based on a 2-dimensional kd-tree needs  $O(n \log n)$  preprocessing time, with  $O(n)$  space complexity. A query can be carried out in  $O(\sqrt{n} + m)$  time, where  $m$  is the number of nodes visited.

- This complexity does not change if no pre-sorting is carried out and a linear-time algorithm for median finding is used for identifying the nodes of the tree. (It just makes the implementation more tedious.)
- More efficient methods are known. But kd-trees are a very versatile tool!
- For a range search in  $\mathbb{R}^k$  we split the space alternatingly by straight lines ( $k = 2$ ), planes ( $k = 3$ ), or hyper-planes (for  $k \geq 4$ ).

## Theorem 6

Range searching based on a  $k$ -dimensional kd-tree needs  $O(k \cdot n \log n)$  preprocessing time, with  $O(n)$  space complexity. A query can be carried out in  $O(k \cdot n^{1-1/k} + m)$  time, where  $m$  is the number of nodes visited.



# Complexity of Range Searching Based on a kd-Tree

## Theorem 5

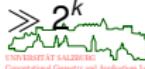
Range searching based on a 2-dimensional kd-tree needs  $O(n \log n)$  preprocessing time, with  $O(n)$  space complexity. A query can be carried out in  $O(\sqrt{n} + m)$  time, where  $m$  is the number of nodes visited.

- This complexity does not change if no pre-sorting is carried out and a linear-time algorithm for median finding is used for identifying the nodes of the tree. (It just makes the implementation more tedious.)
- More efficient methods are known. But kd-trees are a very versatile tool!
- For a range search in  $\mathbb{R}^k$  we split the space alternatingly by straight lines ( $k = 2$ ), planes ( $k = 3$ ), or hyper-planes (for  $k \geq 4$ ).

## Theorem 6

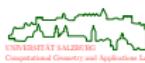
Range searching based on a  $k$ -dimensional kd-tree needs  $O(k \cdot n \log n)$  preprocessing time, with  $O(n)$  space complexity. A query can be carried out in  $O(k \cdot n^{1-1/k} + m)$  time, where  $m$  is the number of nodes visited.

- Note the curse of dimensionality for large values of  $k$ : We ought to have  $n > 2^k$  in order to make kd-trees practical!



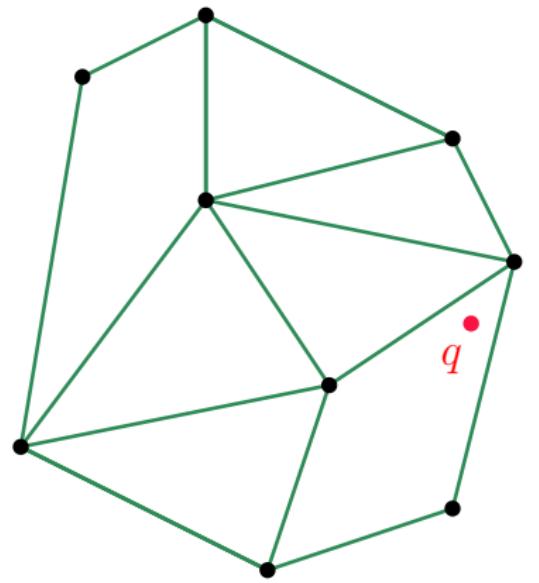
## 2 Geometric Searching

- Introduction
- Range Searching
- Point Inclusion
  - Brute-Force Method
  - Point-in-Polygon Query
  - Slab Method
  - Paradigm: Plane Sweep
  - Chain Method
  - Triangulation Refinement Technique



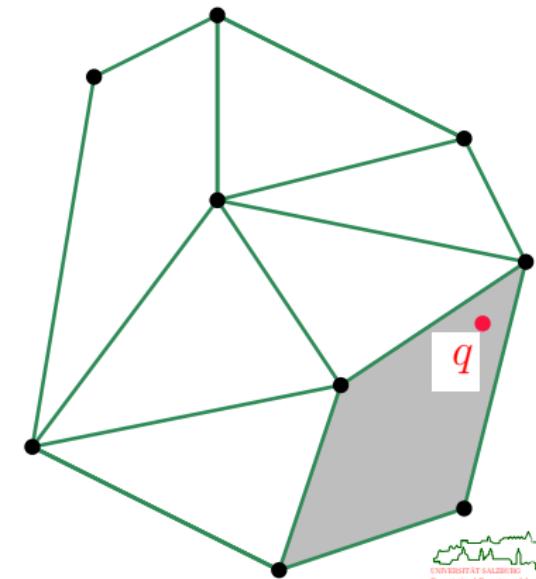
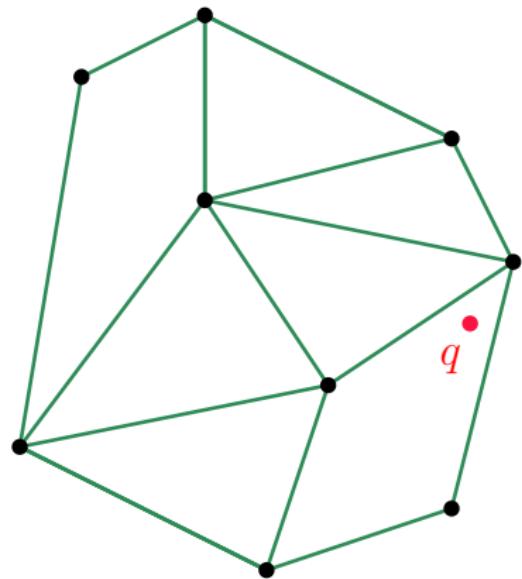
## Point Inclusion

- Given: Decomposition  $\mathcal{G}$  of the plane into polygonal regions, and a query point  $q$ .



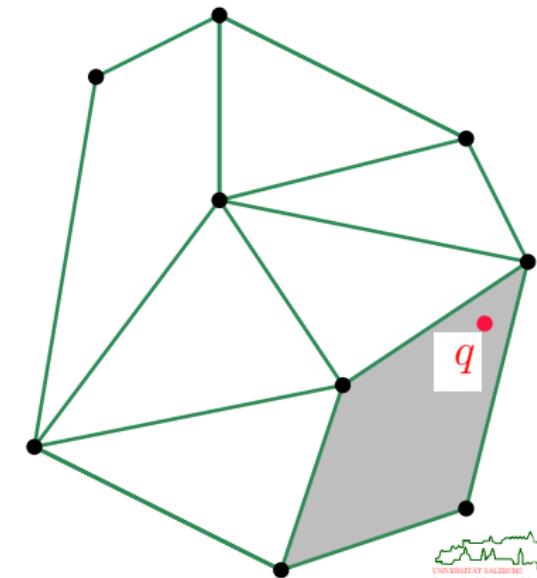
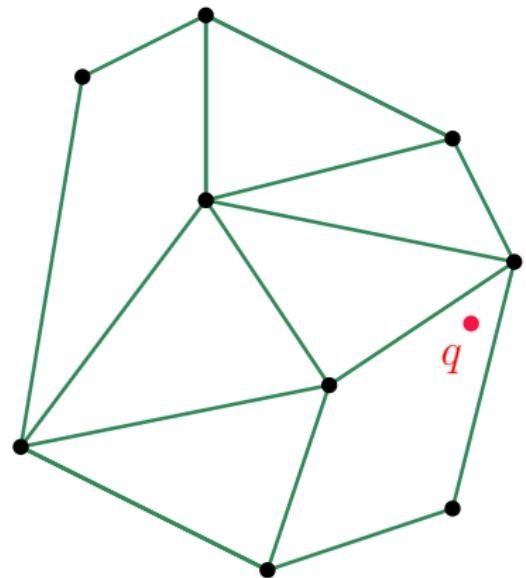
## Point Inclusion

- Given: Decomposition  $\mathcal{G}$  of the plane into polygonal regions, and a query point  $q$ .
- Problem: Which face of  $\mathcal{G}$  contains  $q$ ?



## Point Inclusion

- Given: Decomposition  $\mathcal{G}$  of the plane into polygonal regions, and a query point  $q$ .
- Problem: Which face of  $\mathcal{G}$  contains  $q$ ?
- Aka “point location”. Obviously, point-inclusion problems can also be studied in higher dimensions.



## Brute-Force Method

- Given is an  $n$ -vertex planar subdivision  $\mathcal{G}$ .
- Brute-force approach:
  - for each polygon  $\mathcal{P}$  of  $\mathcal{G}$ :
    - solve point-in-polygon problem for  $q$  within  $\mathcal{P}$ .

## Brute-Force Method

- Given is an  $n$ -vertex planar subdivision  $\mathcal{G}$ .
- Brute-force approach:
  - for each polygon  $\mathcal{P}$  of  $\mathcal{G}$ :
    - solve point-in-polygon problem for  $q$  within  $\mathcal{P}$ .

### Theorem 7

A brute-force point location within an  $n$ -vertex planar subdivision can be carried out in  $O(n)$  time, using  $O(n)$  space.

## Brute-Force Method

- Given is an  $n$ -vertex planar subdivision  $\mathcal{G}$ .
- Brute-force approach:
  - for each polygon  $\mathcal{P}$  of  $\mathcal{G}$ :
    - solve point-in-polygon problem for  $q$  within  $\mathcal{P}$ .

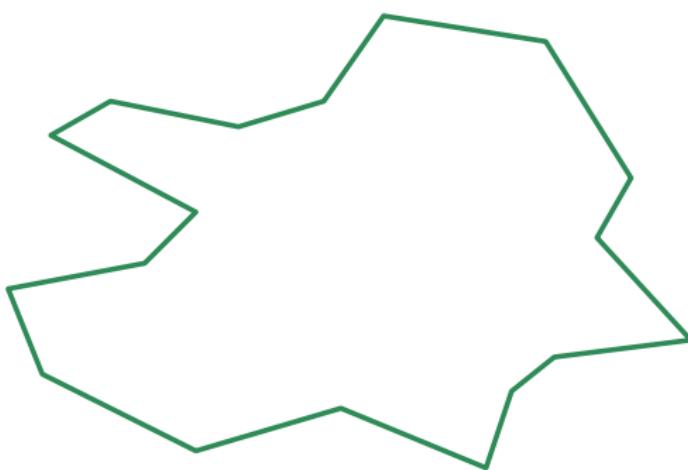
### Theorem 7

A brute-force point location within an  $n$ -vertex planar subdivision can be carried out in  $O(n)$  time, using  $O(n)$  space.

- This is not an efficient solution for repetitive-mode queries!
- Goal: Create geometric data structure that supports some kind of binary search.

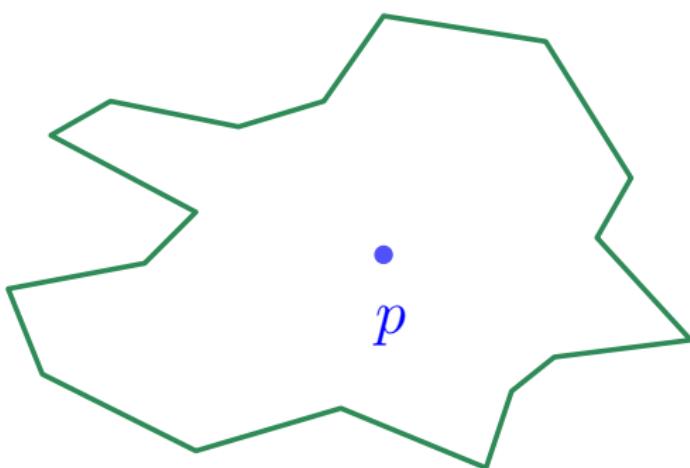
# Point-in-Polygon Query for Star-Shaped Polygons

## ① Preprocessing:



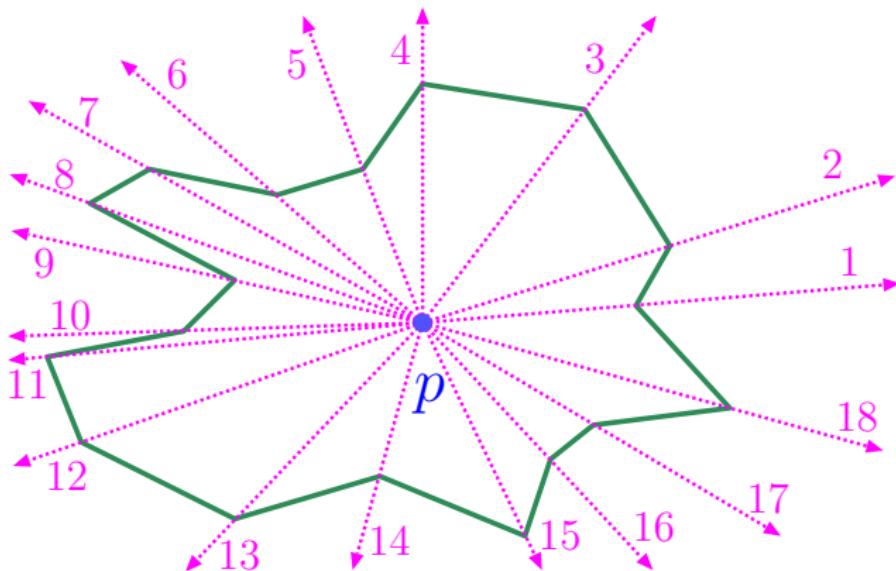
## Point-in-Polygon Query for Star-Shaped Polygons

- ① Preprocessing: Find point  $p$  within nucleus of polygon in  $O(n)$  time.



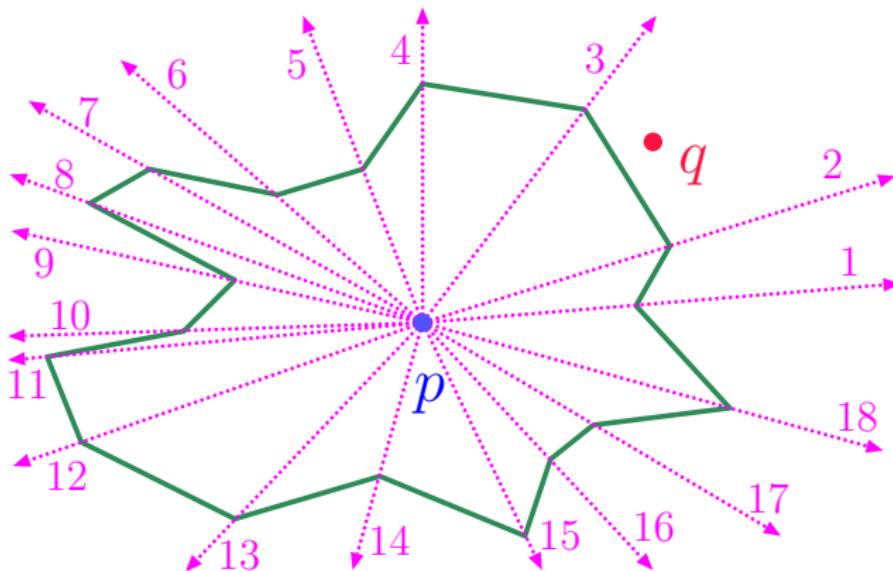
# Point-in-Polygon Query for Star-Shaped Polygons

- ① Preprocessing: Find point  $p$  within nucleus of polygon in  $O(n)$  time.
- ② Preprocessing: Shoot rays starting at  $p$  through each vertex, in  $O(n)$  time.



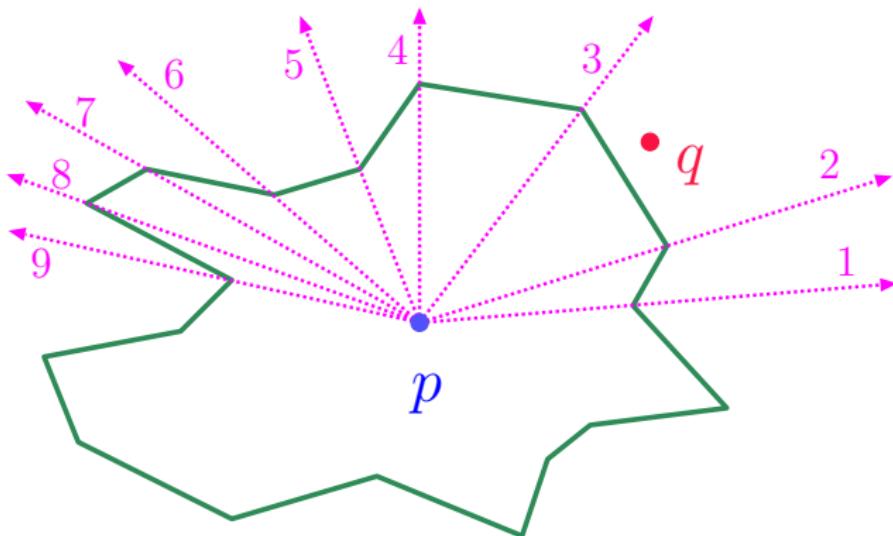
# Point-in-Polygon Query for Star-Shaped Polygons

- ① Preprocessing: Find point  $p$  within nucleus of polygon in  $O(n)$  time.
- ② Preprocessing: Shoot rays starting at  $p$  through each vertex, in  $O(n)$  time.
- ③ For a query point  $q$ :



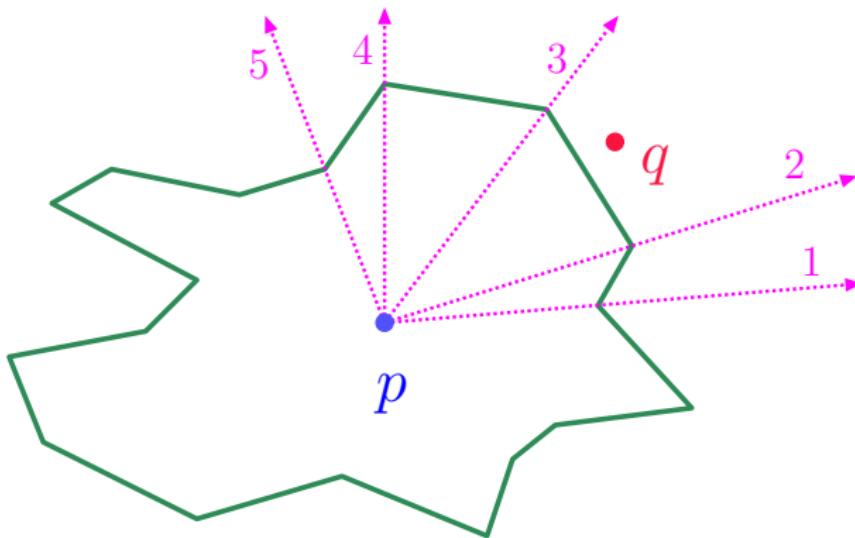
## Point-in-Polygon Query for Star-Shaped Polygons

- ① Preprocessing: Find point  $p$  within nucleus of polygon in  $O(n)$  time.
- ② Preprocessing: Shoot rays starting at  $p$  through each vertex, in  $O(n)$  time.
- ③ For a query point  $q$ : Perform binary search in  $O(\log n)$ .



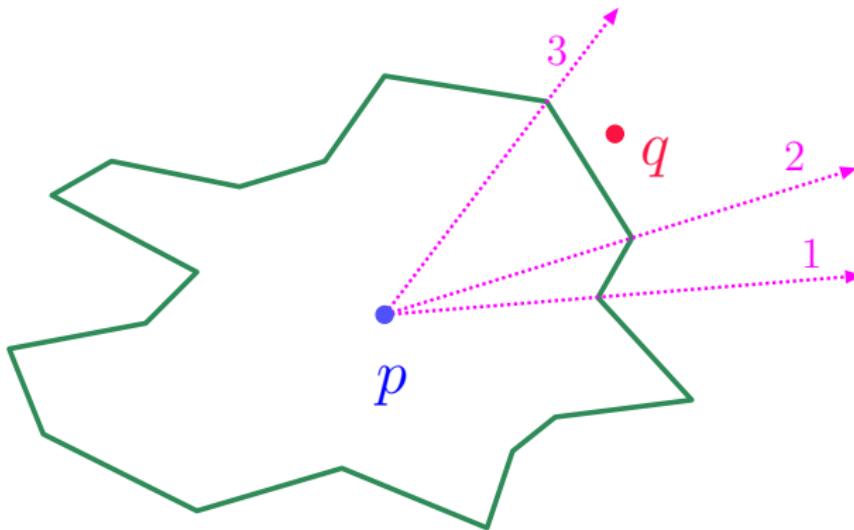
## Point-in-Polygon Query for Star-Shaped Polygons

- ① Preprocessing: Find point  $p$  within nucleus of polygon in  $O(n)$  time.
- ② Preprocessing: Shoot rays starting at  $p$  through each vertex, in  $O(n)$  time.
- ③ For a query point  $q$ : Perform binary search in  $O(\log n)$ .



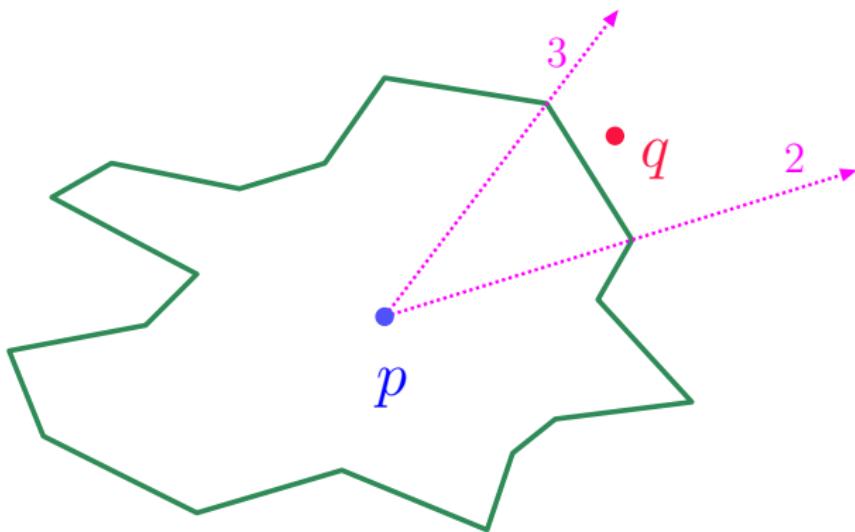
## Point-in-Polygon Query for Star-Shaped Polygons

- ① Preprocessing: Find point  $p$  within nucleus of polygon in  $O(n)$  time.
- ② Preprocessing: Shoot rays starting at  $p$  through each vertex, in  $O(n)$  time.
- ③ For a query point  $q$ : Perform binary search in  $O(\log n)$ .



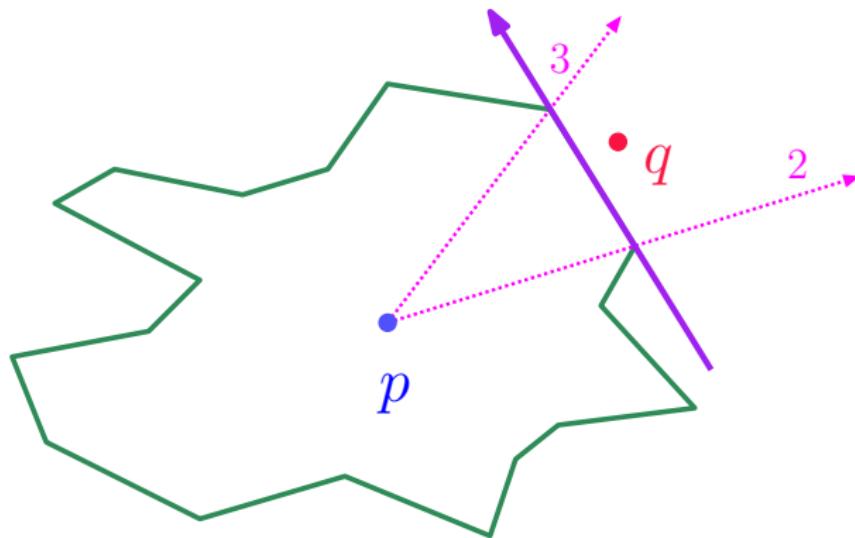
## Point-in-Polygon Query for Star-Shaped Polygons

- ① Preprocessing: Find point  $p$  within nucleus of polygon in  $O(n)$  time.
- ② Preprocessing: Shoot rays starting at  $p$  through each vertex, in  $O(n)$  time.
- ③ For a query point  $q$ : Perform binary search in  $O(\log n)$ .



## Point-in-Polygon Query for Star-Shaped Polygons

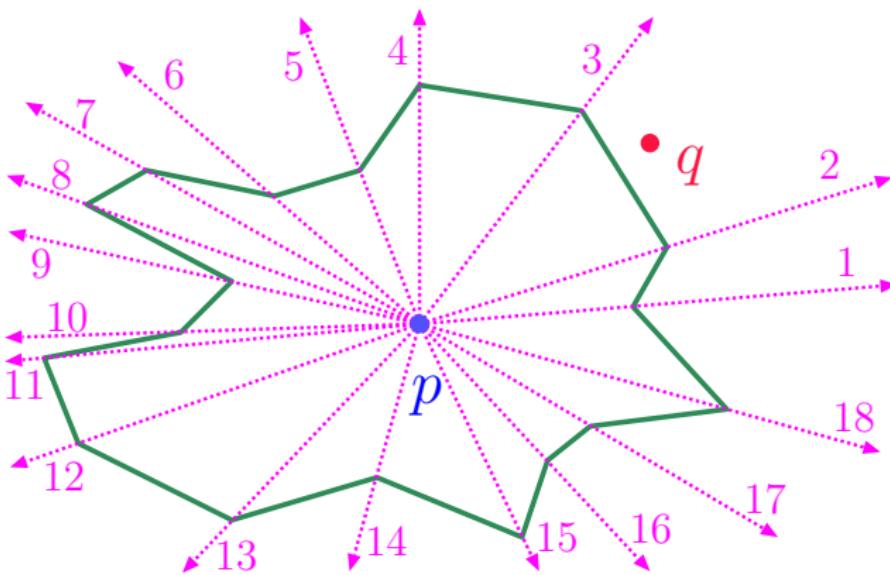
- ① Preprocessing: Find point  $p$  within nucleus of polygon in  $O(n)$  time.
- ② Preprocessing: Shoot rays starting at  $p$  through each vertex, in  $O(n)$  time.
- ③ For a query point  $q$ : Perform binary search in  $O(\log n)$ .
- ④ Determine sidedness relative to one edge of the polygon.



# Point-in-Polygon Query for Star-Shaped Polygons

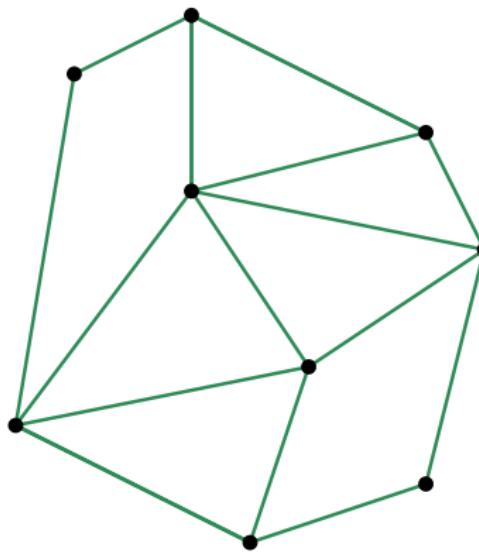
## Theorem 8

For an  $n$ -vertex star-shaped polygon, a point-location query can be answered in  $O(\log n)$  query time, after  $O(n)$  preprocessing and within  $O(n)$  space.



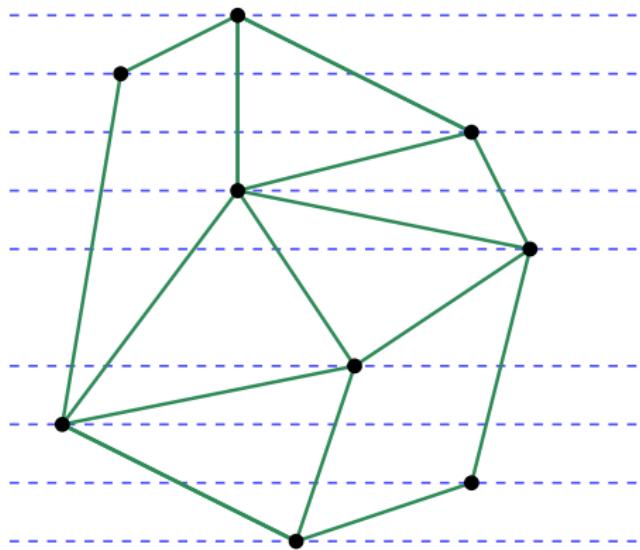
## Slab Method

- Planar straight-line graph (PSLG): An embedding of a planar graph such that all edges are straight lines. (Fáry's Theorem [1948] states that every planar graph has such an embedding.) W.l.o.g.: The graph is connected.

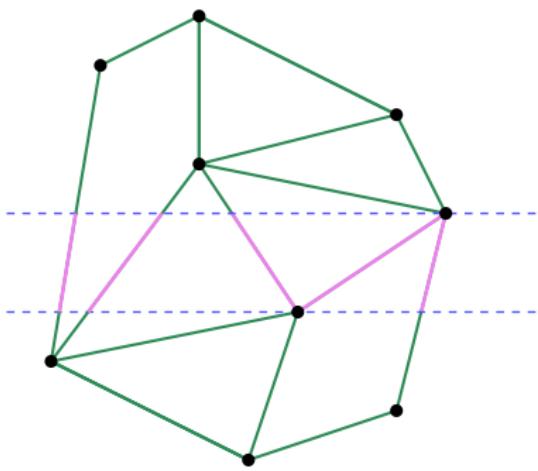


## Slab Method

- Planar straight-line graph (PSLG): An embedding of a planar graph such that all edges are straight lines. (Fáry's Theorem [1948] states that every planar graph has such an embedding.) W.l.o.g.: The graph is connected.
- Draw a horizontal line through each vertex: We get (at most)  $(n + 1)$  slabs.



## Slab Method

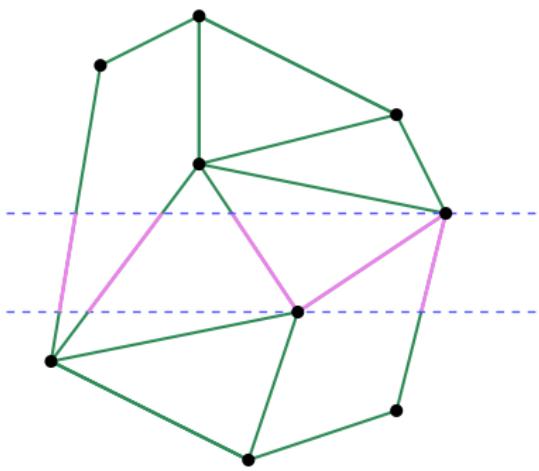


### Lemma 9

A slab has the following properties:

- (1) A slab contains no other vertex of the PSLG.

## Slab Method

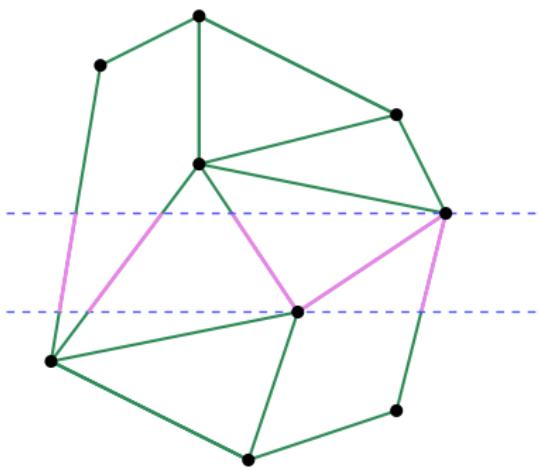


### Lemma 9

A slab has the following properties:

- (1) A slab contains no other vertex of the PSLG.
- (2) All finite faces define trapezoids, which may degenerate to triangles.

## Slab Method



### Lemma 9

A slab has the following properties:

- (1) A slab contains no other vertex of the PSLG.
- (2) All finite faces define trapezoids, which may degenerate to triangles.
- (3) There are no intersections of edges within a slab.

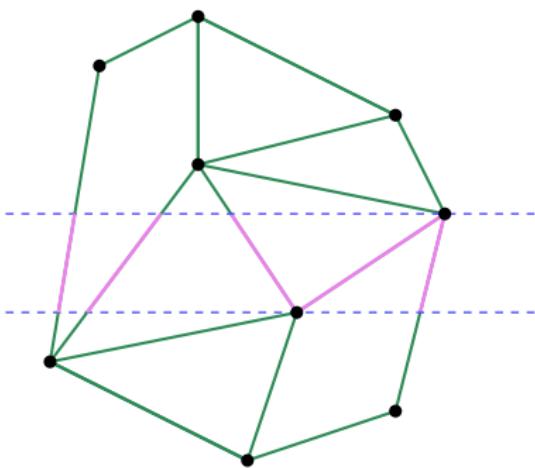
## Theorem 10

The preprocessing of the slab method takes  $O(n^2 \log n)$  time and needs  $O(n^2)$  space.  
A query can be carried out in  $O(\log n)$  time.

## Theorem 10

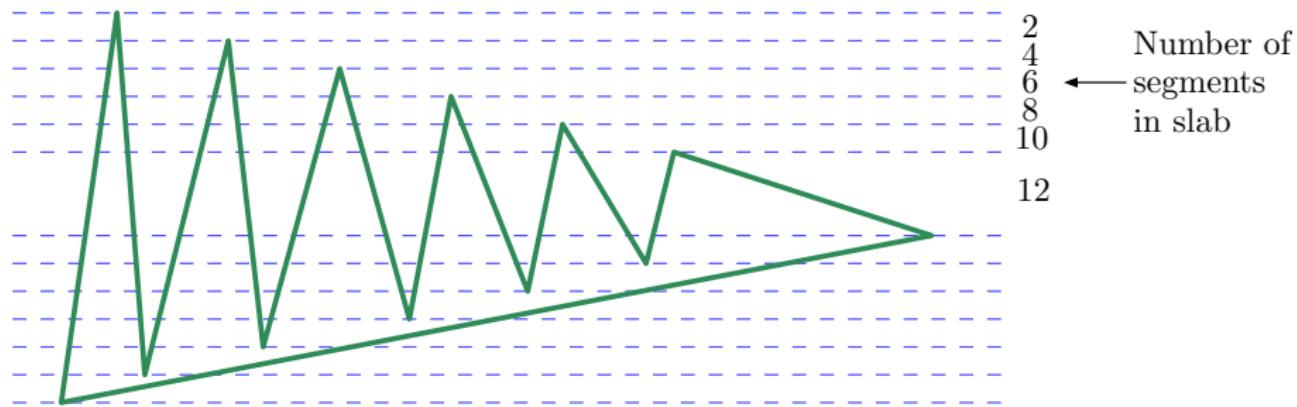
The preprocessing of the slab method takes  $O(n^2 \log n)$  time and needs  $O(n^2)$  space. A query can be carried out in  $O(\log n)$  time.

*Proof:* There are  $O(n)$  slabs, and each slab can have as many as  $O(n)$  segments, resulting in an  $O(n \log n)$  sort per slab. □



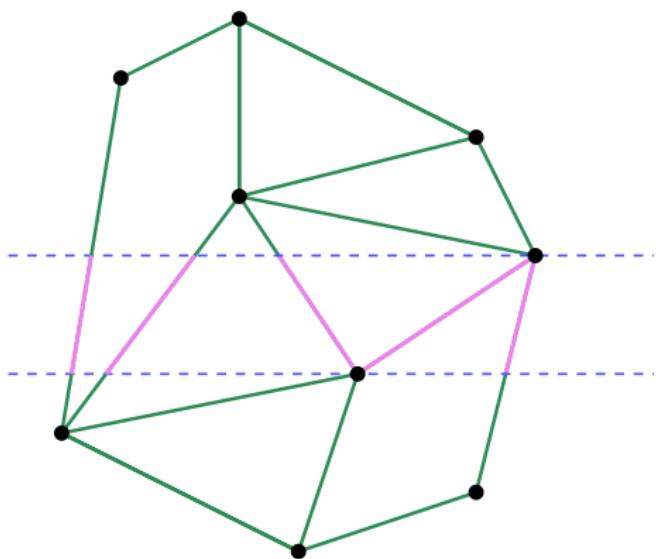
## Slab Method: Worst-Case Memory Consumption

- The  $O(n^2)$  space complexity can be achieved. I.e., we have  $\Theta(n^2)$  space.



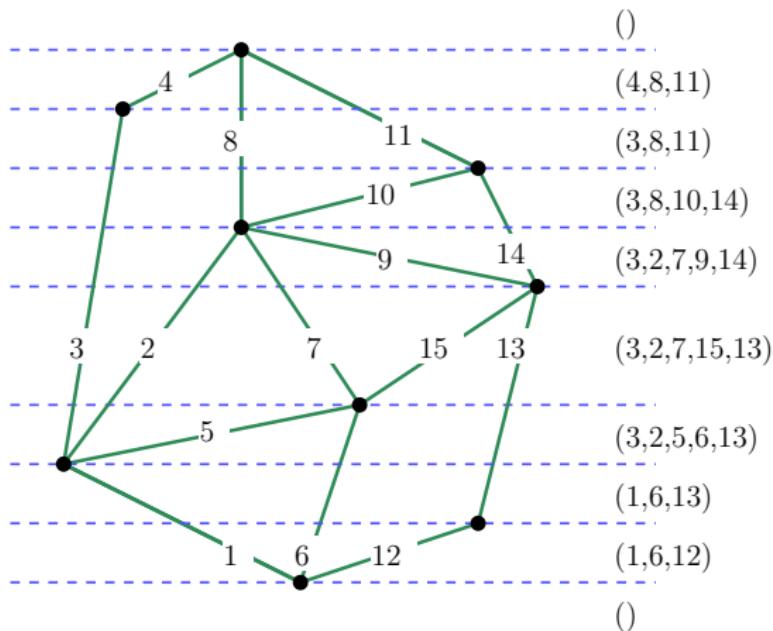
## Paradigm: Plane-Sweep Algorithm

- Observation: Some edges of PSLG cross several slabs, and their relative left-to-right order does not change.



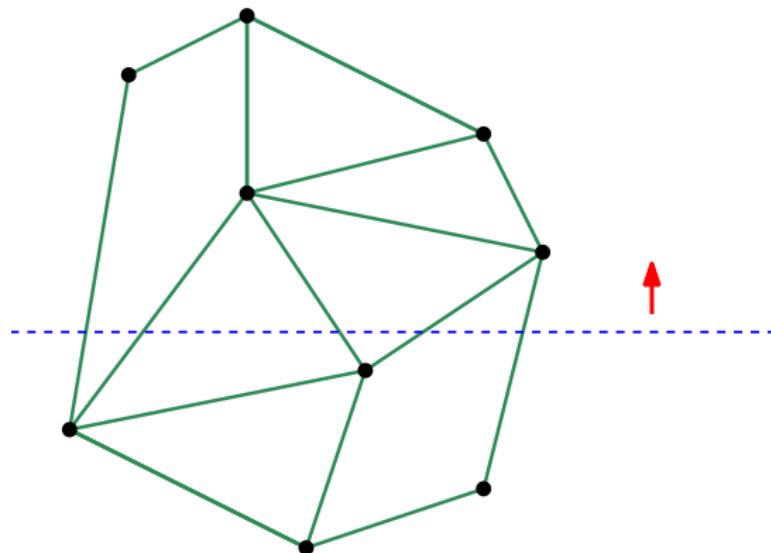
## Paradigm: Plane-Sweep Algorithm

- Observation: Some edges of PSLG cross several slabs, and their relative left-to-right order does not change.
- Goal: Exploit coherence!



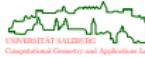
## Paradigm: Plane-Sweep Algorithm

- Observation: Some edges of PSLG cross several slabs, and their relative left-to-right order does not change.
- Goal: Exploit coherence!
- The essential idea is to sweep a horizontal line over the PSLG.



# Plane-Sweep Algorithm

- A *plane-sweep algorithm* uses two data structures:
  - ➊ *Event-point schedule*: Sequence of positions to be assumed by the sweep line.

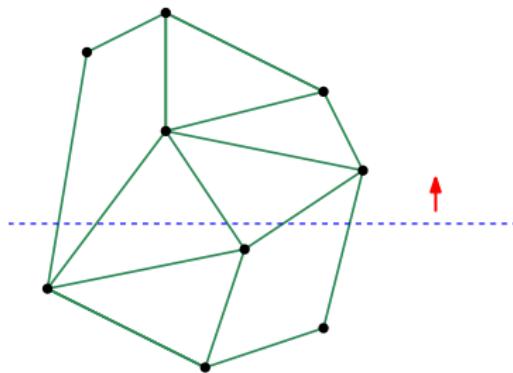


## Plane-Sweep Algorithm

- A *plane-sweep algorithm* uses two data structures:
  - ① *Event-point schedule*: Sequence of positions to be assumed by the sweep line.
  - ② *Sweep-line status*: Description of the intersection of the sweep line with the geometric object(s) being swept at the current event.

## Plane-Sweep Algorithm

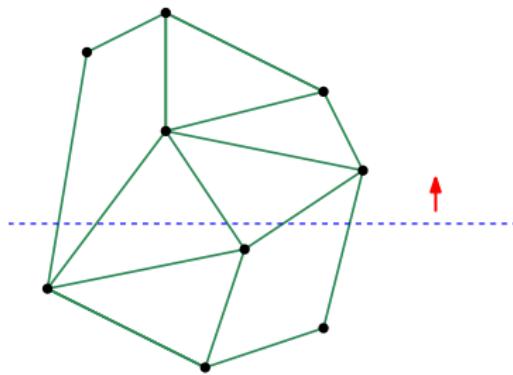
- A *plane-sweep algorithm* uses two data structures:
  - ① *Event-point schedule*: Sequence of positions to be assumed by the sweep line.
  - ② *Sweep-line status*: Description of the intersection of the sweep line with the geometric object(s) being swept at the current event.



- Plane sweep applied to preprocessing for the slab method:
  - ① Event-point schedule: vertices of  $\mathcal{G}$ , arranged according to ascending  $y$ -coordinates. (The sweep is bottom-to-top.)

## Plane-Sweep Algorithm

- A *plane-sweep algorithm* uses two data structures:
  - ① *Event-point schedule*: Sequence of positions to be assumed by the sweep line.
  - ② *Sweep-line status*: Description of the intersection of the sweep line with the geometric object(s) being swept at the current event.



- Plane sweep applied to preprocessing for the slab method:
  - ① Event-point schedule: vertices of  $\mathcal{G}$ , arranged according to ascending  $y$ -coordinates. (The sweep is bottom-to-top.)
  - ② Sweep-line status: left-to-right sequence of edges of  $\mathcal{G}$  that intersect the sweep line.

## Slab Method Based on Plane Sweep

- Event processing: At each event, i.e., vertex  $v \in \mathcal{G}$ ,
  - ➊ delete the edges terminating at  $v$  from the sweep-line status;

## Slab Method Based on Plane Sweep

- Event processing: At each event, i.e., vertex  $v \in \mathcal{G}$ ,
  - ➊ delete the edges terminating at  $v$  from the sweep-line status;
  - ➋ add the edges originating at  $v$  to the sweep-line status in sorted order;

## Slab Method Based on Plane Sweep

- Event processing: At each event, i.e., vertex  $v \in \mathcal{G}$ ,
  - ➊ delete the edges terminating at  $v$  from the sweep-line status;
  - ➋ add the edges originating at  $v$  to the sweep-line status in sorted order;
  - ➌ output the sweep-line status and construct a slab.

## Slab Method Based on Plane Sweep

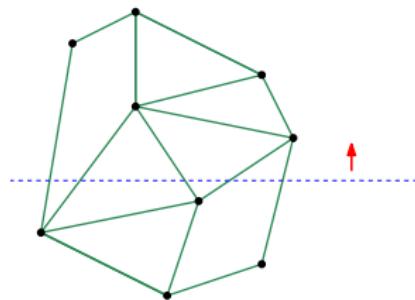
- Event processing: At each event, i.e., vertex  $v \in \mathcal{G}$ ,
  - ① delete the edges terminating at  $v$  from the sweep-line status;
  - ② add the edges originating at  $v$  to the sweep-line status in sorted order;
  - ③ output the sweep-line status and construct a slab.
- Query time: Stays at  $O(\log n)$ .
- Preprocessing time: Reduced to  $O(n^2)$ , since we still have  $O(n)$  slabs each requiring  $O(n)$  time per slab.
- Space: Stays at  $O(n^2)$ . ([Sarnak&Tarjan 1986]: Can be brought down to  $O(n)$ .)

# Slab Method Based on Plane Sweep

- Event processing: At each event, i.e., vertex  $v \in \mathcal{G}$ ,
  - ① delete the edges terminating at  $v$  from the sweep-line status;
  - ② add the edges originating at  $v$  to the sweep-line status in sorted order;
  - ③ output the sweep-line status and construct a slab.
- Query time: Stays at  $O(\log n)$ .
- Preprocessing time: Reduced to  $O(n^2)$ , since we still have  $O(n)$  slabs each requiring  $O(n)$  time per slab.
- Space: Stays at  $O(n^2)$ . ([Sarnak&Tarjan 1986]: Can be brought down to  $O(n)$ .)

## Theorem 11 (Dobkin&Lipton 1976)

For an  $n$ -vertex PSLG, the slab method supports point-location queries in  $O(\log n)$  query time, after  $O(n^2)$  preprocessing and within  $O(n^2)$  space.



## Definition 12 (Polygonal chain)

A *chain*  $c = (v_1, v_2, \dots, v_p)$  is a polygonal curve with vertex set  $\{v_1, v_2, \dots, v_p\}$  and edge set  $\{(v_i, v_{i+1}) : 1 \leq i < p\}$ .

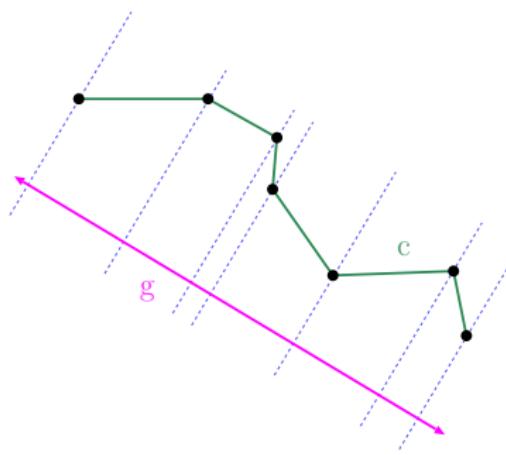
# Chain Method

## Definition 12 (Polygonal chain)

A  $chain c = (v_1, v_2, \dots, v_p)$  is a polygonal curve with vertex set  $\{v_1, v_2, \dots, v_p\}$  and edge set  $\{(v_i, v_{i+1}) : 1 \leq i < p\}$ .

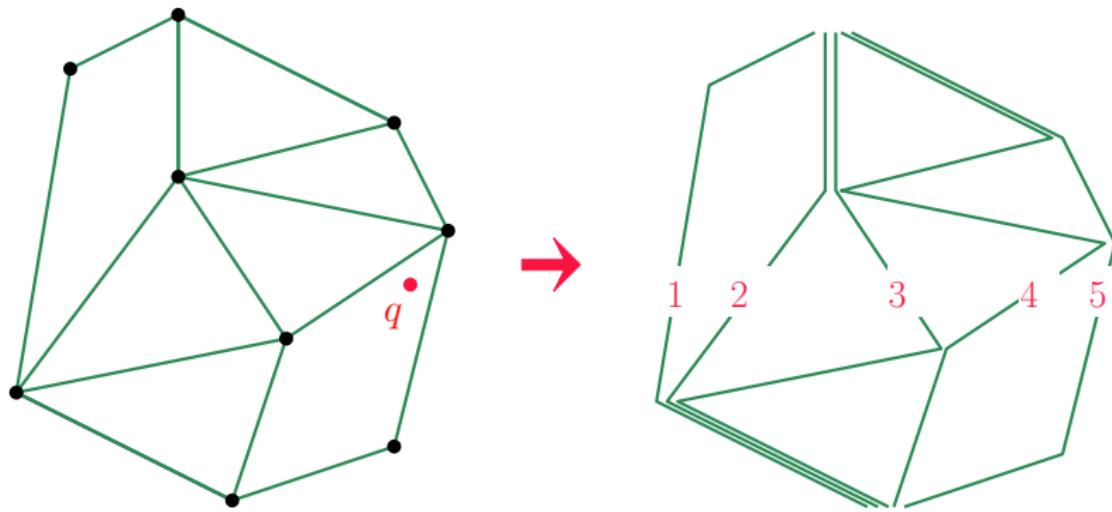
## Definition 13 (Monotone chain)

A chain  $c = (v_1, v_2, \dots, v_p)$  is *monotone* with respect to a line  $g$  if every line orthogonal to  $g$  intersects  $c$  in at most one point.



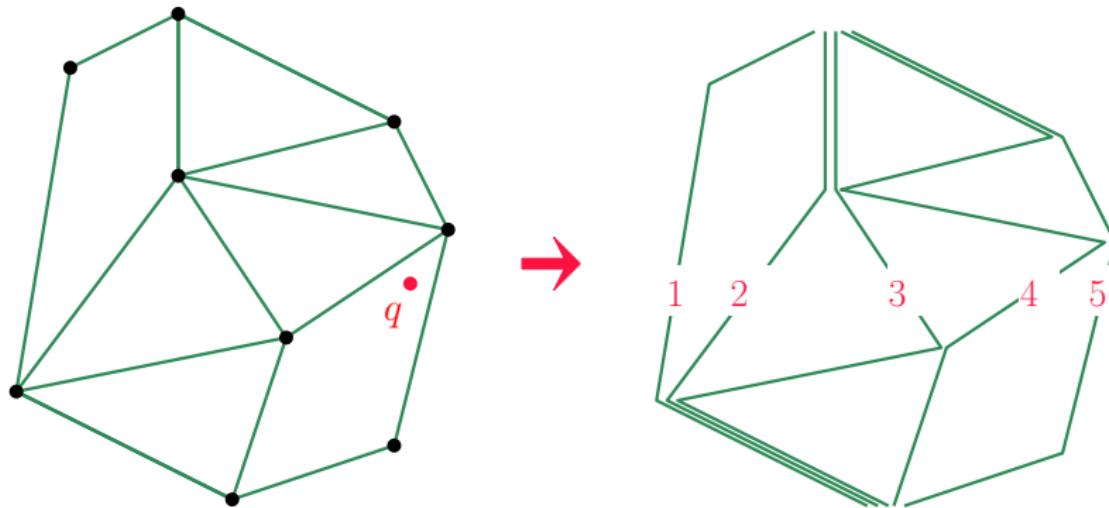
## Chain Method

- Preprocessing: Decompose the PSLG into a set of monotone chains.



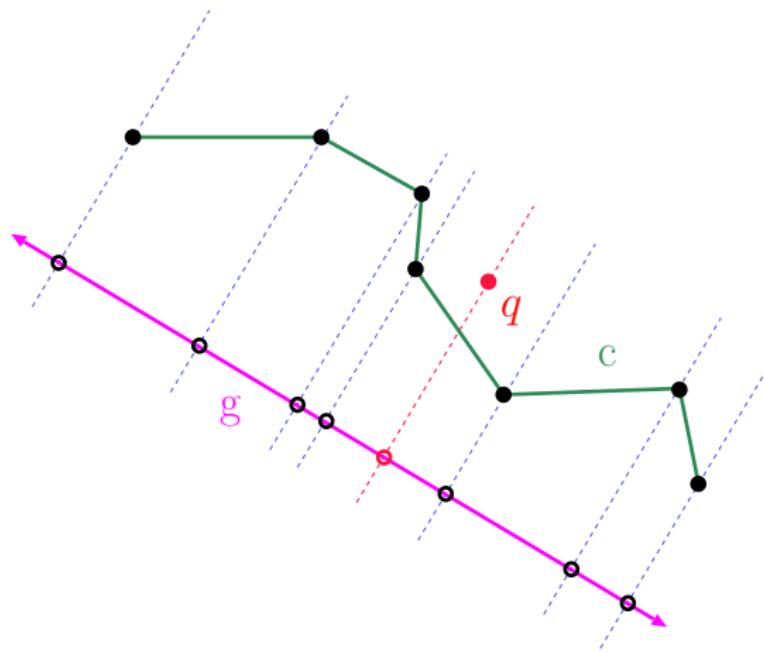
## Chain Method

- Preprocessing: Decompose the PSLG into a set of monotone chains.
- Query: Binary search on the chains to find the two chains that are on either side of the query point, followed by binary search on one of those chains to determine the face.



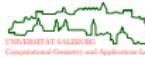
## Point-Chain Discrimination

- ① Compute orthogonal projection of  $q$  on  $g$  in  $O(1)$  time.
- ② Binary search on  $g$  among the orthogonal projections of the vertices of  $c$ , in  $O(\log k)$  time for a  $k$ -vertex chain.
- ③ Locate  $q$  relative to one edge of  $c$ .



## Analysis of Chain Method

- Let  $C = \{c_1, c_2, \dots, c_r\}$  be a set of chains which are monotone relative to the same line  $g$  and which exhibit the following two properties:



## Analysis of Chain Method

- Let  $C = \{c_1, c_2, \dots, c_r\}$  be a set of chains which are monotone relative to the same line  $g$  and which exhibit the following two properties:
  - The union of the members of  $C$  contains the PSLG  $\mathcal{G}$ . (Some edge of  $\mathcal{G}$  may be in more than one chain of  $C$  ).



## Analysis of Chain Method

- Let  $C = \{c_1, c_2, \dots, c_r\}$  be a set of chains which are monotone relative to the same line  $g$  and which exhibit the following two properties:
  - The union of the members of  $C$  contains the PSLG  $\mathcal{G}$ . (Some edge of  $\mathcal{G}$  may be in more than one chain of  $C$  ).
  - For any two chains  $c_i$  and  $c_j$  of  $C$ , the vertices of  $c_i$  which are not also members of  $c_j$  lie on the same side of  $c_j$ .

## Analysis of Chain Method

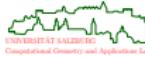
- Let  $C = \{c_1, c_2, \dots, c_r\}$  be a set of chains which are monotone relative to the same line  $g$  and which exhibit the following two properties:
  - The union of the members of  $C$  contains the PSLG  $\mathcal{G}$ . (Some edge of  $\mathcal{G}$  may be in more than one chain of  $C$  ).
  - For any two chains  $c_i$  and  $c_j$  of  $C$ , the vertices of  $c_i$  which are not also members of  $c_j$  lie on the same side of  $c_j$ .
- Due to this order, a binary search can be applied to  $C$  using the point-chain discrimination operation as the comparison operator.

## Analysis of Chain Method

- Let  $C = \{c_1, c_2, \dots, c_r\}$  be a set of chains which are monotone relative to the same line  $g$  and which exhibit the following two properties:
  - The union of the members of  $C$  contains the PSLG  $\mathcal{G}$ . (Some edge of  $\mathcal{G}$  may be in more than one chain of  $C$  ).
  - For any two chains  $c_i$  and  $c_j$  of  $C$ , the vertices of  $c_i$  which are not also members of  $c_j$  lie on the same side of  $c_j$ .
- Due to this order, a binary search can be applied to  $C$  using the point-chain discrimination operation as the comparison operator.
- Complexity of a query:
  - For  $r$  chains with up to  $p$  vertices, the search uses  $O(\log r \cdot \log p)$  time.
  - Since  $r = O(n)$  and  $p = O(n)$ , the query complexity is  $O(\log^2 n)$ .

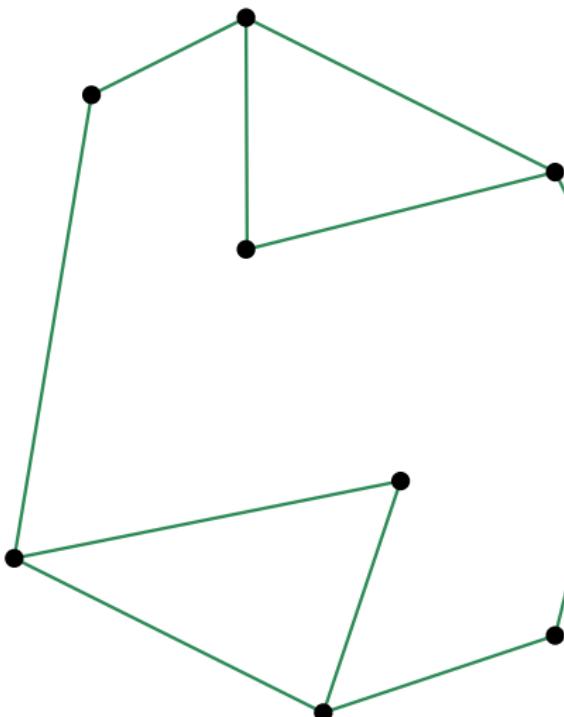
## Analysis of Chain Method

- Can every PSLG be decomposed into a set of monotone chains that are monotone relative to the  $y$ -axis?



## Analysis of Chain Method

- Can every PSLG be decomposed into a set of monotone chains that are monotone relative to the  $y$ -axis?



## Analysis of Chain Method

- Assume that the vertices  $\{v_1, v_2, \dots, v_n\}$  of  $\mathcal{G}$  are indexed as follows:

$$\forall(1 \leq i < j \leq n) \quad [y(v_i) < y(v_j)] \quad \vee \quad [(y(v_i) = y(v_j)) \wedge (x(v_i) < x(v_j))],$$

where  $x(v_i)$  denotes the  $x$ -coordinate of vertex  $v_i$ .

## Analysis of Chain Method

- Assume that the vertices  $\{v_1, v_2, \dots, v_n\}$  of  $\mathcal{G}$  are indexed as follows:

$$\forall (1 \leq i < j \leq n) \quad [y(v_i) < y(v_j)] \quad \vee \quad [(y(v_i) = y(v_j)) \wedge (x(v_i) < x(v_j))],$$

where  $x(v_i)$  denotes the  $x$ -coordinate of vertex  $v_i$ .

### Definition 14

A vertex  $v_i$  is called *regular* if there are vertices  $v_j$  and  $v_k$ , with  $j < i < k$ , such that the edges  $v_j v_i$  and  $v_i v_k$  belong to  $\mathcal{G}$ .

# Analysis of Chain Method

- Assume that the vertices  $\{v_1, v_2, \dots, v_n\}$  of  $\mathcal{G}$  are indexed as follows:

$$\forall (1 \leq i < j \leq n) \quad [y(v_i) < y(v_j)] \quad \vee \quad [(y(v_i) = y(v_j)) \wedge (x(v_i) < x(v_j))],$$

where  $x(v_i)$  denotes the  $x$ -coordinate of vertex  $v_i$ .

## Definition 14

A vertex  $v_i$  is called *regular* if there are vertices  $v_j$  and  $v_k$ , with  $j < i < k$ , such that the edges  $v_j v_i$  and  $v_i v_k$  belong to  $\mathcal{G}$ .

## Definition 15

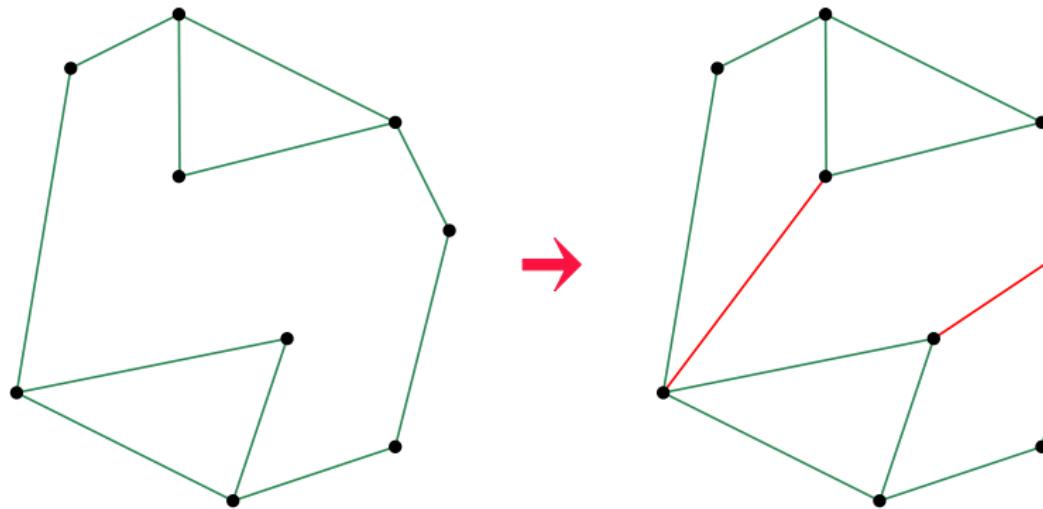
A PSLG is called *regular* if each of its vertices is regular, except for the two extreme vertices  $v_1$  and  $v_n$ .

# Analysis of Chain Method

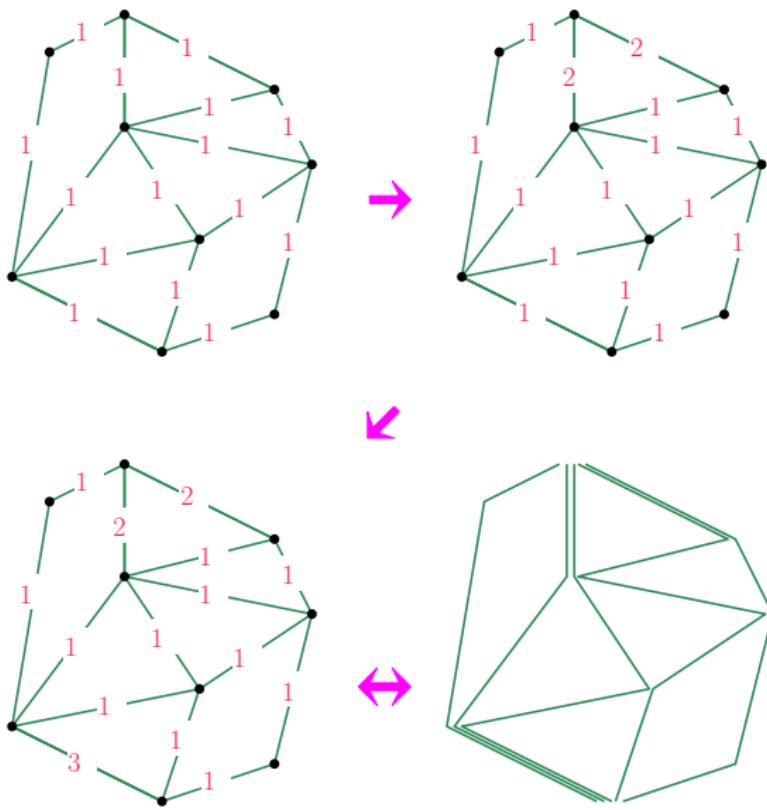
## Lemma 16

Regularization can be carried out in  $O(n \log n)$  time.

*Proof:* Weight balancing by means of plane sweep. □

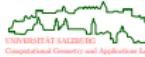


# Weight Balancing for Decomposition into Monotone Chains



## Analysis of Chain Method

- Preprocessing: The regularization of an  $n$ -vertex PSLG can be carried out in time  $O(n \log n)$  and space  $O(n)$ .



## Analysis of Chain Method

- Preprocessing: The regularization of an  $n$ -vertex PSLG can be carried out in time  $O(n \log n)$  and space  $O(n)$ .
- Query time:  $O(\log^2 n)$ , but can be brought down to  $O(\log n)$  by means of fractional cascading.

## Analysis of Chain Method

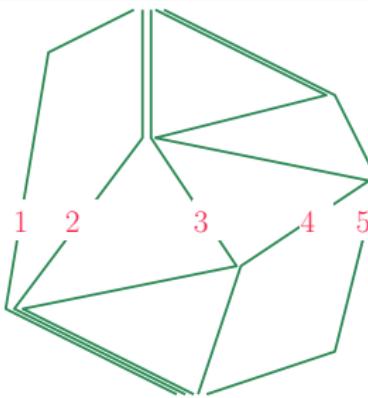
- Preprocessing: The regularization of an  $n$ -vertex PSLG can be carried out in time  $O(n \log n)$  and space  $O(n)$ .
- Query time:  $O(\log^2 n)$ , but can be brought down to  $O(\log n)$  by means of fractional cascading.
- Space:  $O(n^2)$  if all chains are stored naively, but can be brought down to  $O(n)$  [Edelsbrunner&Guibas&Stolfi 1986]).

## Analysis of Chain Method

- Preprocessing: The regularization of an  $n$ -vertex PSLG can be carried out in time  $O(n \log n)$  and space  $O(n)$ .
- Query time:  $O(\log^2 n)$ , but can be brought down to  $O(\log n)$  by means of fractional cascading.
- Space:  $O(n^2)$  if all chains are stored naively, but can be brought down to  $O(n)$  [Edelsbrunner&Guibas&Stolfi 1986]).

### Theorem 17

For an  $n$ -vertex PSLG, the chain method supports point-location queries in  $O(\log n)$  query time, after  $O(n \log n)$  preprocessing and within  $O(n)$  space.



## Triangulation Refinement Technique

- Aka: Dobkin-Kirkpatrick hierarchy [1990] in 3D, based on Kirkpatrick [1983].
- For a given  $n$ -vertex PSLG  $\mathcal{G}$ , generate a PSLG  $\mathcal{G}'$  with the following properties:
  - ①  $\mathcal{G}'$  is a super-graph of  $\mathcal{G}$ ,
  - ②  $\mathcal{G}'$  is a triangulation,
  - ③  $\mathcal{G}'$  has a triangular boundary, and, by Euler's formula,
  - ④  $\mathcal{G}'$  has exactly  $3n - 6$  edges.

## Triangulation Refinement Technique

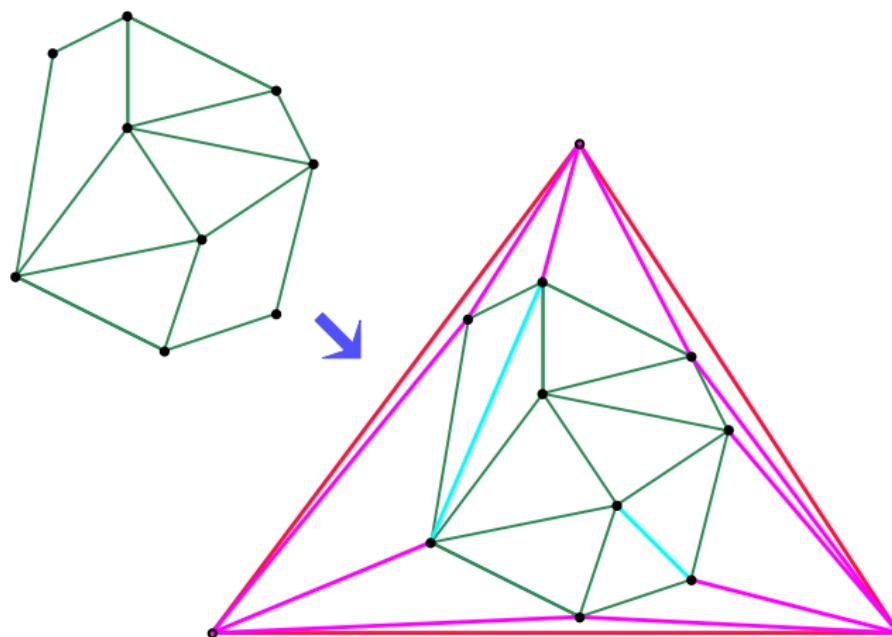
- Aka: Dobkin-Kirkpatrick hierarchy [1990] in 3D, based on Kirkpatrick [1983].
- For a given  $n$ -vertex PSLG  $\mathcal{G}$ , generate a PSLG  $\mathcal{G}'$  with the following properties:
  - ①  $\mathcal{G}'$  is a super-graph of  $\mathcal{G}$ ,
  - ②  $\mathcal{G}'$  is a triangulation,
  - ③  $\mathcal{G}'$  has a triangular boundary, and, by Euler's formula,
  - ④  $\mathcal{G}'$  has exactly  $3n - 6$  edges.
- Construct hierarchy of triangulations above  $\mathcal{G}'$ , and set up a directed acyclic search graph  $\mathcal{T}$ , in time  $O(n \log n)$  and space  $O(n)$ .

## Triangulation Refinement Technique

- Aka: Dobkin-Kirkpatrick hierarchy [1990] in 3D, based on Kirkpatrick [1983].
- For a given  $n$ -vertex PSLG  $\mathcal{G}$ , generate a PSLG  $\mathcal{G}'$  with the following properties:
  - ①  $\mathcal{G}'$  is a super-graph of  $\mathcal{G}$ ,
  - ②  $\mathcal{G}'$  is a triangulation,
  - ③  $\mathcal{G}'$  has a triangular boundary, and, by Euler's formula,
  - ④  $\mathcal{G}'$  has exactly  $3n - 6$  edges.
- Construct hierarchy of triangulations above  $\mathcal{G}'$ , and set up a directed acyclic search graph  $\mathcal{T}$ , in time  $O(n \log n)$  and space  $O(n)$ .
- Perform point-location queries within  $\mathcal{T}$  in time  $O(\log n)$ .

# Triangulation Refinement Technique

- Convert  $\mathcal{G}$  into a triangulation  $\mathcal{G}'$ .

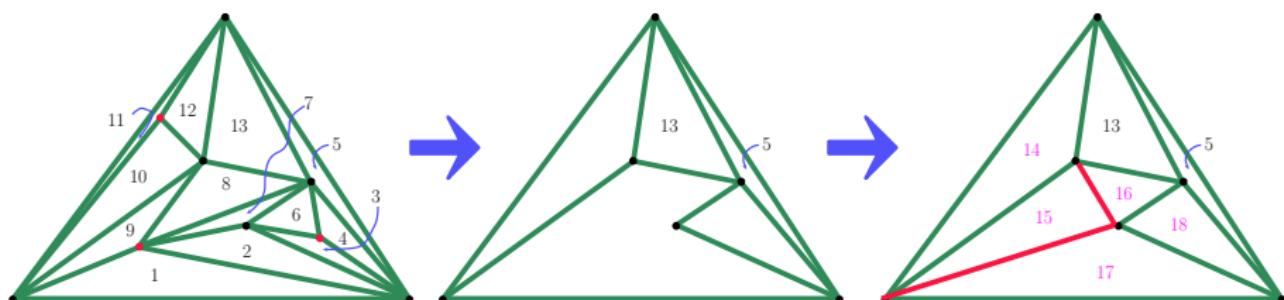


## Triangulation Refinement: Hierarchy of Triangulations

- Consider a triangulated PSLG  $\mathcal{G}'$  on  $n$  vertices.
- We construct a hierarchy of triangulations  $S_1, S_2, \dots, S_{h(n)}$ ,

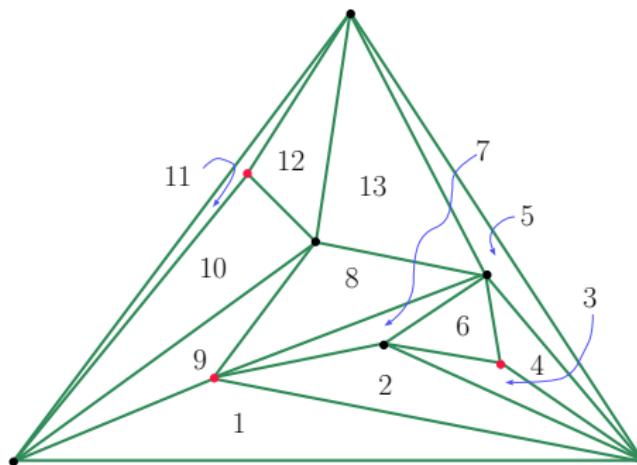
## Triangulation Refinement: Hierarchy of Triangulations

- Consider a triangulated PSLG  $\mathcal{G}'$  on  $n$  vertices.
- We construct a hierarchy of triangulations  $S_1, S_2, \dots, S_{h(n)}$ , where  $S_1 := \mathcal{G}'$  and  $S_i$  is obtained from  $S_{i-1}$  as follows:
  - Step 1:** Remove a maximal independent set of non-boundary vertices of  $S_{i-1}$  together with their incident edges.
  - Step 2:** Re-triangulate the holes arising from the removal of those vertices and edges.
- The final triangulation in the hierarchy,  $S_{h(n)}$ , has no internal vertices; it is just one triangle.



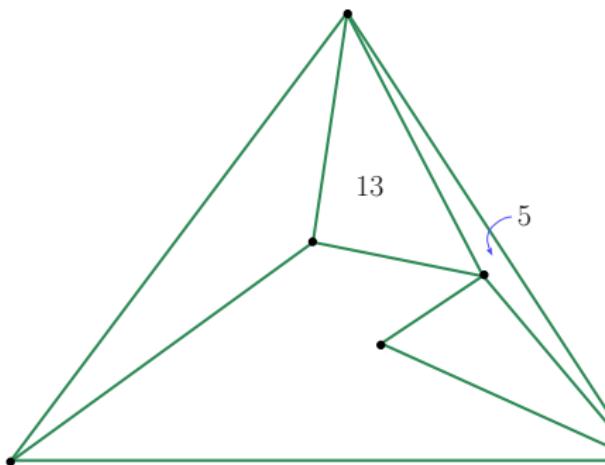
# Animation of Construction of Triangulation Hierarchy

- **Step 1:** Remove a maximal independent set of non-boundary vertices of  $S_{i-1}$  together with their incident edges.
- **Step 2:** Re-triangulate the holes arising from the removal of those vertices and edges.



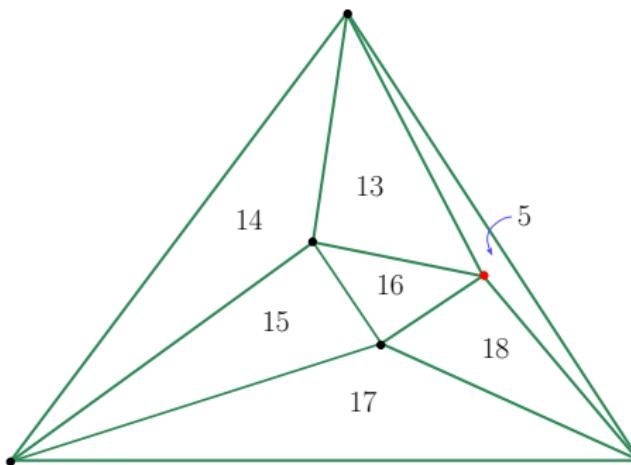
## Animation of Construction of Triangulation Hierarchy

- **Step 1:** Remove a maximal independent set of non-boundary vertices of  $S_{i-1}$  together with their incident edges.
- **Step 2:** Re-triangulate the holes arising from the removal of those vertices and edges.



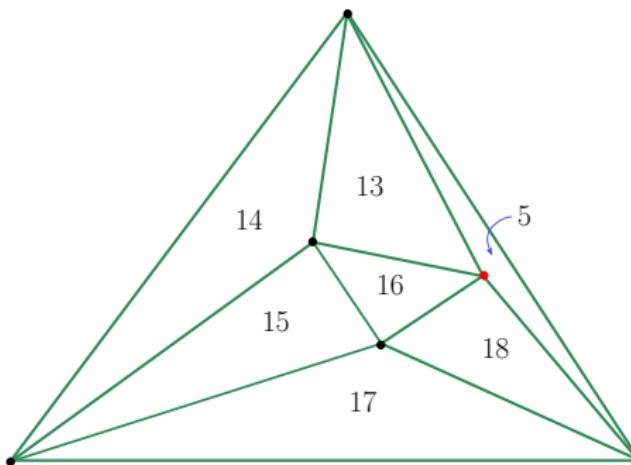
# Animation of Construction of Triangulation Hierarchy

- **Step 1:** Remove a maximal independent set of non-boundary vertices of  $S_{i-1}$  together with their incident edges.
- **Step 2:** Re-triangulate the holes arising from the removal of those vertices and edges.



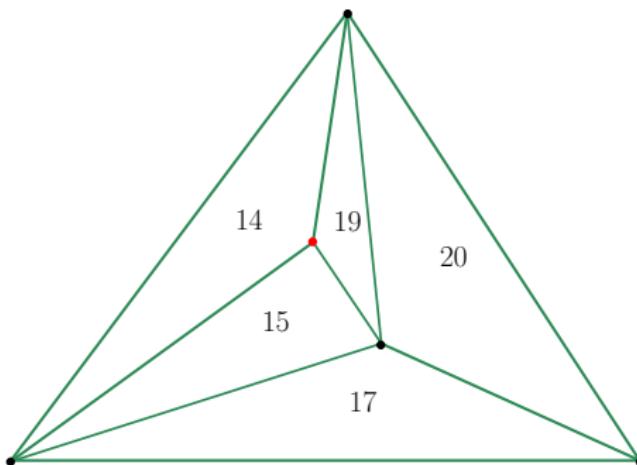
# Animation of Construction of Triangulation Hierarchy

- **Step 1:** Remove a maximal independent set of non-boundary vertices of  $S_{i-1}$  together with their incident edges.
- **Step 2:** Re-triangulate the holes arising from the removal of those vertices and edges.



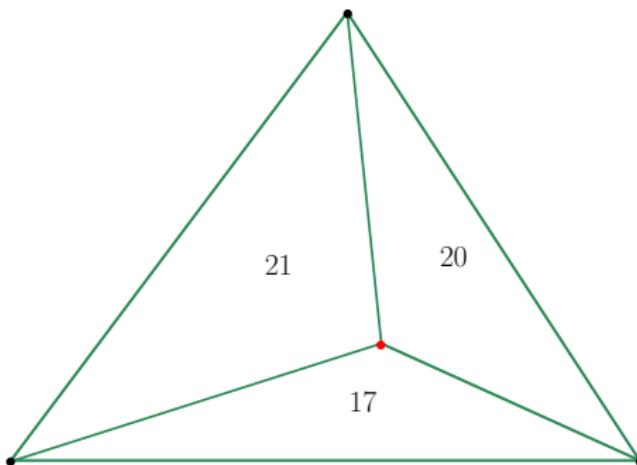
## Animation of Construction of Triangulation Hierarchy

- **Step 1:** Remove a maximal independent set of non-boundary vertices of  $S_{i-1}$  together with their incident edges.
- **Step 2:** Re-triangulate the holes arising from the removal of those vertices and edges.



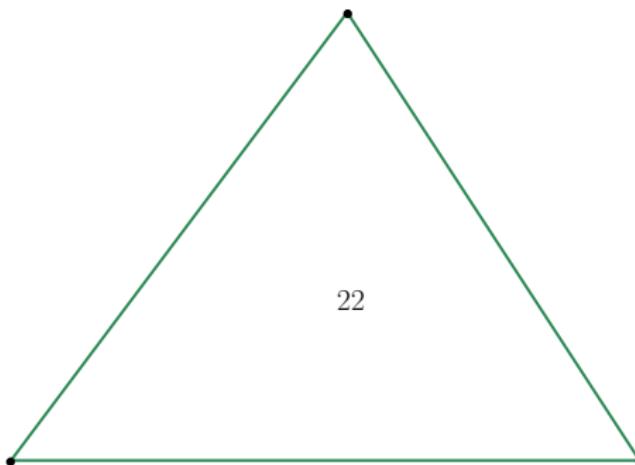
## Animation of Construction of Triangulation Hierarchy

- **Step 1:** Remove a maximal independent set of non-boundary vertices of  $S_{i-1}$  together with their incident edges.
- **Step 2:** Re-triangulate the holes arising from the removal of those vertices and edges.

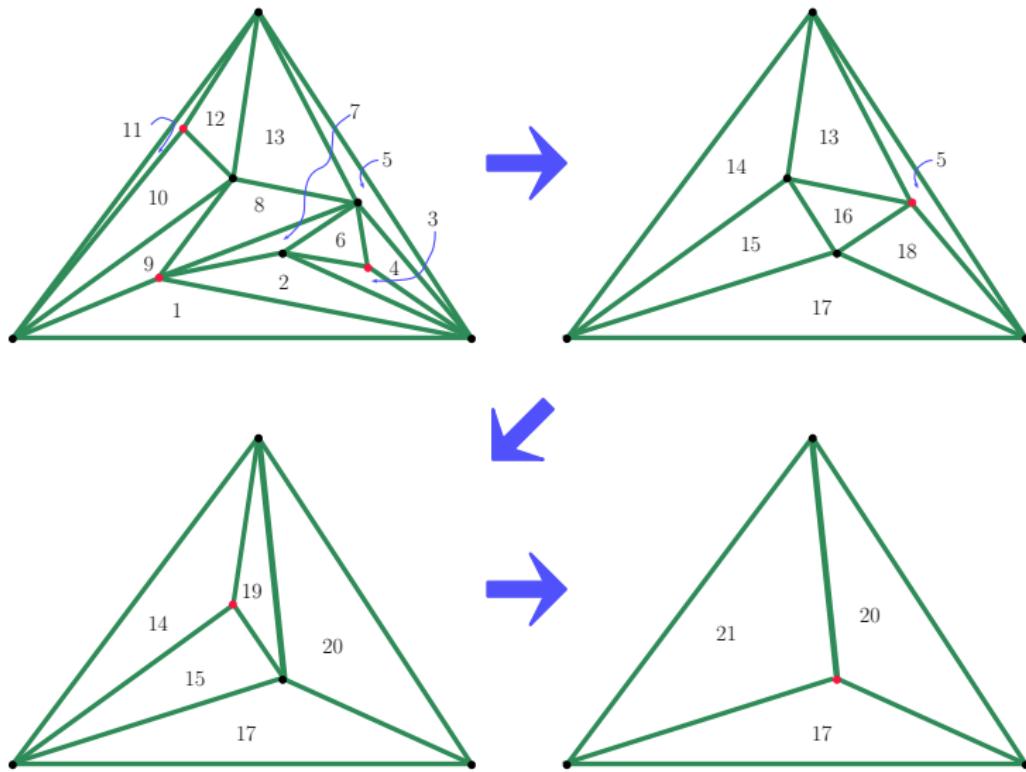


## Animation of Construction of Triangulation Hierarchy

- **Step 1:** Remove a maximal independent set of non-boundary vertices of  $S_{i-1}$  together with their incident edges.
- **Step 2:** Re-triangulate the holes arising from the removal of those vertices and edges.

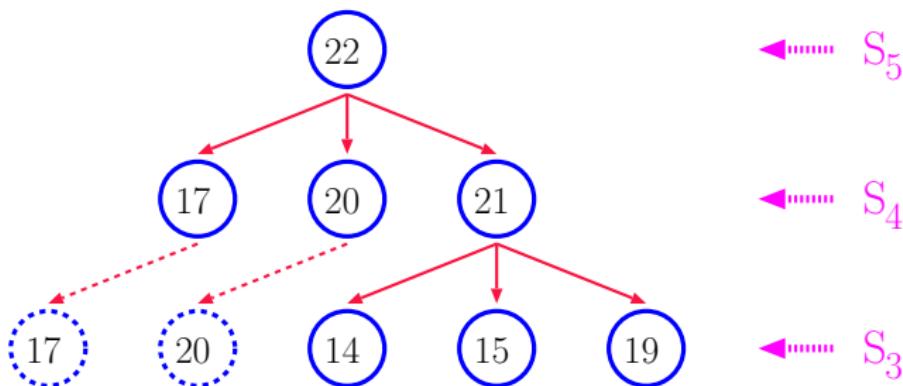


# Triangulation Refinement: Hierarchy of Triangulations



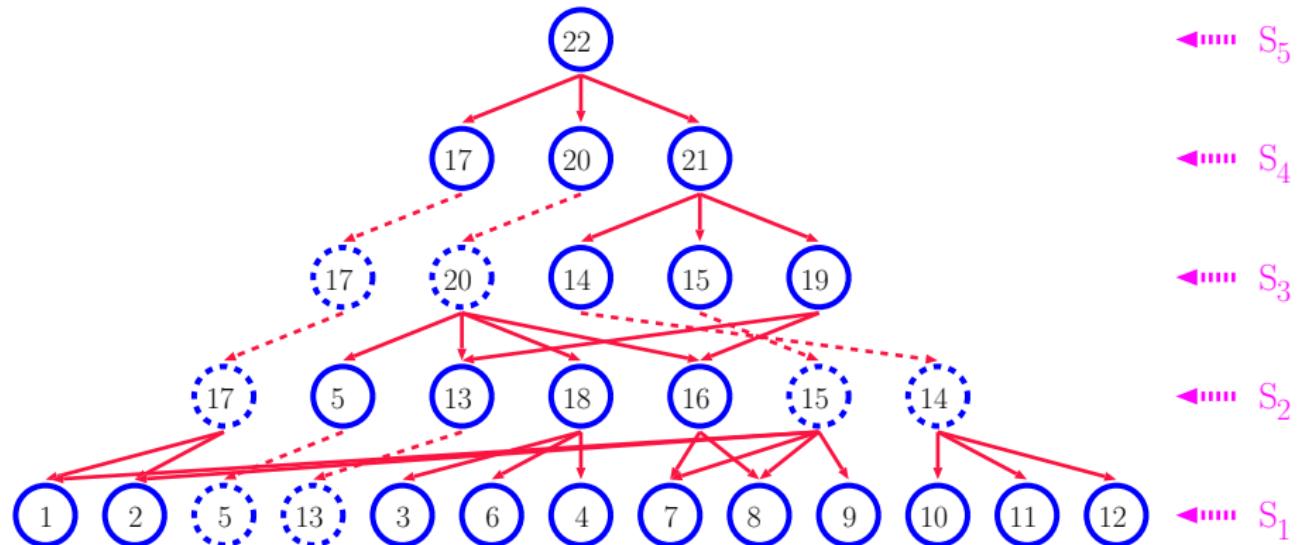
## Triangulation Refinement: Directed Acyclic Search Graph (DAG)

- We set up an directed acyclic search graph  $\mathcal{T}$ .
- The graph  $\mathcal{T}$  contains an edge from triangle  $R_k$  to triangle  $R_j$  if, when constructing triangulation  $S_i$  from triangulation  $S_{i-1}$ , we have:
  - ①  $R_j$  is removed from  $S_{i-1}$  in **Step 1**.
  - ②  $R_k$  is created in  $S_i$  in **Step 2**.
  - ③  $R_j \cap R_k \neq \emptyset$ .



# Triangulation Refinement: Directed Acyclic Search Graph

- We construct a hierarchy of triangulations  $S_1, S_2, \dots, S_{h(n)}$ .



## Triangulation Refinement Technique: Query

- For a query point  $q$ , perform a point-in-triangle test of  $q$  relative to the root triangle of  $\mathcal{T}$ .
- If  $q$  is
  - outside** then  $q$  lies in the unbounded (exterior) face.
  - inside** then  $q$  is tested for triangle inclusion with each of the descendants of the current node.

## Triangulation Refinement Technique: Query

- For a query point  $q$ , perform a point-in-triangle test of  $q$  relative to the root triangle of  $\mathcal{T}$ .
- If  $q$  is
  - outside** then  $q$  lies in the unbounded (exterior) face.
  - inside** then  $q$  is tested for triangle inclusion with each of the descendants of the current node.
- This scheme is applied recursively until a leaf of  $\mathcal{T}$  is reached.

## Triangulation Refinement Technique: Query

- For a query point  $q$ , perform a point-in-triangle test of  $q$  relative to the root triangle of  $\mathcal{T}$ .
- If  $q$  is
  - outside** then  $q$  lies in the unbounded (exterior) face.
  - inside** then  $q$  is tested for triangle inclusion with each of the descendants of the current node.
- This scheme is applied recursively until a leaf of  $\mathcal{T}$  is reached.
- What is the complexity of a query? This depends on
  - the height  $h(n)$  of  $\mathcal{T}$ ,
  - the maximum number  $m$  of point-in-triangle tests needed per node.

## Triangulation Refinement Technique: Query

- For a query point  $q$ , perform a point-in-triangle test of  $q$  relative to the root triangle of  $\mathcal{T}$ .
- If  $q$  is
  - outside** then  $q$  lies in the unbounded (exterior) face.
  - inside** then  $q$  is tested for triangle inclusion with each of the descendants of the current node.
- This scheme is applied recursively until a leaf of  $\mathcal{T}$  is reached.
- What is the complexity of a query? This depends on
  - the height  $h(n)$  of  $\mathcal{T}$ ,
  - the maximum number  $m$  of point-in-triangle tests needed per node.
- Both terms seem to depend on how we select the vertices of  $S_{i-1}$  that will not be part of  $S_i$ .
- Goal: Construct  $\mathcal{T}$  such that  $m = O(1)$  and  $h(n) = O(\log n)$ .

## Analysis of Triangulation Refinement Technique

- Let  $N_i$  denote the number of vertices of triangulation  $S_i$ .
- Criterion for selecting vertices that are to be removed:

*Remove a set of non-adjacent vertices of degree less than  $K$ ,*

where  $K := 12$ .

# Analysis of Triangulation Refinement Technique

- Let  $N_i$  denote the number of vertices of triangulation  $S_i$ .
- Criterion for selecting vertices that are to be removed:

*Remove a set of non-adjacent vertices of degree less than  $K$ ,*

where  $K := 12$ .

- Easy to prove: This criterion allows us to delete at least

$$\frac{1}{12} \left( \frac{N_{i-1}}{2} - 3 \right)$$

vertices within  $S_{i-1}$ .

# Analysis of Triangulation Refinement Technique

- Let  $N_i$  denote the number of vertices of triangulation  $S_i$ .
- Criterion for selecting vertices that are to be removed:

*Remove a set of non-adjacent vertices of degree less than  $K$ ,*

where  $K := 12$ .

- Easy to prove: This criterion allows us to delete at least

$$\frac{1}{12} \left( \frac{N_{i-1}}{2} - 3 \right)$$

vertices within  $S_{i-1}$ .

- We get:

- $N_i \leq \alpha N_{i-1}$ , where  $\alpha \approx \frac{23}{24}$ .
- $h(n) \leq \lceil \log_{1/\alpha} n \rceil \approx 16 \log n$ , and only  $O(n)$  triangles need to be stored.
- Finally,  $m = (K - 1) - 2 = 9$ .

# Analysis of Triangulation Refinement Technique

- The key step in the preprocessing is the initial triangulation of the PSLG, which takes  $O(n \log n)$  time (or  $O(n)$  time if Chazelle's linear-time algorithm is used). All other triangulation operations can easily be carried out in time linear in the number of vertices involved.

# Analysis of Triangulation Refinement Technique

- The key step in the preprocessing is the initial triangulation of the PSLG, which takes  $O(n \log n)$  time (or  $O(n)$  time if Chazelle's linear-time algorithm is used). All other triangulation operations can easily be carried out in time linear in the number of vertices involved.

## Theorem 18 (Kirkpatrick 1983)

For an  $n$ -vertex PSLG, triangulation refinement supports point-location queries in  $O(\log n)$  query time, after  $O(n)$  preprocessing and within  $O(n)$  space.

# Analysis of Triangulation Refinement Technique

- The key step in the preprocessing is the initial triangulation of the PSLG, which takes  $O(n \log n)$  time (or  $O(n)$  time if Chazelle's linear-time algorithm is used). All other triangulation operations can easily be carried out in time linear in the number of vertices involved.

## Theorem 18 (Kirkpatrick 1983)

For an  $n$ -vertex PSLG, triangulation refinement supports point-location queries in  $O(\log n)$  query time, after  $O(n)$  preprocessing and within  $O(n)$  space.

- Other choices for  $K$  yield tighter bounds! E.g.,  $K := 9$  yields the slightly better bounds  $\alpha \approx \frac{17}{18}$  and  $12 \log n$  per query, and more elaborate choices for the vertices to be deleted bring down the query complexity to roughly  $\frac{9}{2} \log n$ .

# Analysis of Triangulation Refinement Technique

- The key step in the preprocessing is the initial triangulation of the PSLG, which takes  $O(n \log n)$  time (or  $O(n)$  time if Chazelle's linear-time algorithm is used). All other triangulation operations can easily be carried out in time linear in the number of vertices involved.

## Theorem 18 (Kirkpatrick 1983)

For an  $n$ -vertex PSLG, triangulation refinement supports point-location queries in  $O(\log n)$  query time, after  $O(n)$  preprocessing and within  $O(n)$  space.

- Other choices for  $K$  yield tighter bounds! E.g.,  $K := 9$  yields the slightly better bounds  $\alpha \approx \frac{17}{18}$  and  $12 \log n$  per query, and more elaborate choices for the vertices to be deleted bring down the query complexity to roughly  $\frac{9}{2} \log n$ .
- Although this point-inclusion algorithm is optimum in terms of the  $O$ -notation, it is not very practical and better (but more elaborate) algorithms are known.

## Convex Hulls

- Basics
- Graham's Scan
- Gift Wrapping
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls

## Convex Hulls

### Basics

- Graham's Scan
- Gift Wrapping
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls

# Basics of Convex Hulls

## Definition 19 (Convex hull, Dt.: konvexe Hülle)

Given a set  $S = \{p_1, p_2, \dots, p_n\}$  of points in the plane, the convex hull  $CH(S)$  is the smallest convex set in the plane that contains all the points of  $S$ .



# Basics of Convex Hulls

## Definition 19 (Convex hull, Dt.: konvexe Hülle)

Given a set  $S = \{p_1, p_2, \dots, p_n\}$  of points in the plane, the convex hull  $CH(S)$  is the smallest convex set in the plane that contains all the points of  $S$ .



- This definition is generalized easily to convex hulls of infinite sets and to higher dimensions. (However, some claims need to be modified in the general setting.)

## Basics of Convex Hulls

### Definition 19 (Convex hull, Dt.: konvexe Hülle)

Given a set  $S = \{p_1, p_2, \dots, p_n\}$  of points in the plane, the convex hull  $CH(S)$  is the smallest convex set in the plane that contains all the points of  $S$ .



- This definition is generalized easily to convex hulls of infinite sets and to higher dimensions. (However, some claims need to be modified in the general setting.)

### Lemma 20

For a set  $S$  of points in the plane,  $CH(S)$  is given by the intersection of all convex sets that contain  $S$ .

# Complexity of Computing Convex Hulls

## Lemma 21

For a set  $S$  of points in the plane,  $CH(S)$  is a convex polygon.



# Complexity of Computing Convex Hulls

## Lemma 21

For a set  $S$  of points in the plane,  $CH(S)$  is a convex polygon.

- Problem CONVEXHULL:
  - Given: a set  $S = \{p_1, p_2, \dots, p_n\}$  of points in the plane.
  - Output: the convex hull  $CH(S)$ , as an ordered list of vertices.
- Question: Can we state a worst-case lower bound on the time complexity of CONVEXHULL, i.e., for computing  $CH(S)$ ?



# Complexity of Computing Convex Hulls

## Lemma 21

For a set  $S$  of points in the plane,  $CH(S)$  is a convex polygon.

- Problem CONVEXHULL:
  - Given: a set  $S = \{p_1, p_2, \dots, p_n\}$  of points in the plane.
  - Output: the convex hull  $CH(S)$ , as an ordered list of vertices.
- Question: Can we state a worst-case lower bound on the time complexity of CONVEXHULL, i.e., for computing  $CH(S)$ ?

## Theorem 22

SORTING is linear-time transformable to CONVEXHULL.

# Complexity of Computing Convex Hulls

## Lemma 21

For a set  $S$  of points in the plane,  $CH(S)$  is a convex polygon.

- Problem CONVEXHULL:
  - Given: a set  $S = \{p_1, p_2, \dots, p_n\}$  of points in the plane.
  - Output: the convex hull  $CH(S)$ , as an ordered list of vertices.
- Question: Can we state a worst-case lower bound on the time complexity of CONVEXHULL, i.e., for computing  $CH(S)$ ?

## Theorem 22

SORTING is linear-time transformable to CONVEXHULL.

## Corollary 23

Solving CONVEXHULL for  $n$  points requires at least  $\Omega(n \log n)$  time.

# Complexity of Computing Convex Hulls

## Lemma 21

For a set  $S$  of points in the plane,  $CH(S)$  is a convex polygon.

- Problem CONVEXHULL:
  - Given: a set  $S = \{p_1, p_2, \dots, p_n\}$  of points in the plane.
  - Output: the convex hull  $CH(S)$ , as an ordered list of vertices.
- Question: Can we state a worst-case lower bound on the time complexity of CONVEXHULL, i.e., for computing  $CH(S)$ ?

## Theorem 22

SORTING is linear-time transformable to CONVEXHULL.

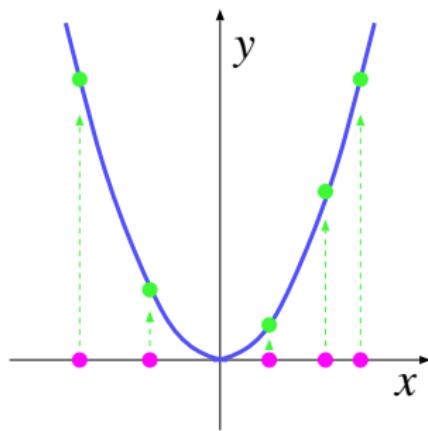
## Corollary 23

Solving CONVEXHULL for  $n$  points requires at least  $\Omega(n \log n)$  time.

- Note: These lower bounds also apply if only the unordered set of hull vertices is sought.

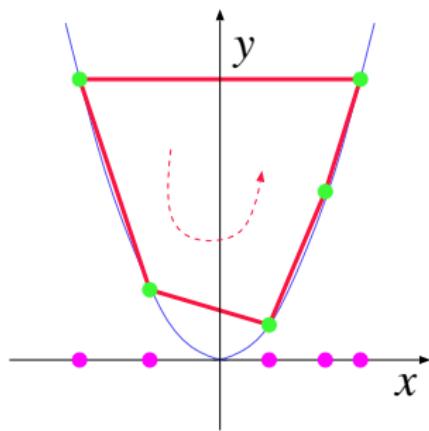
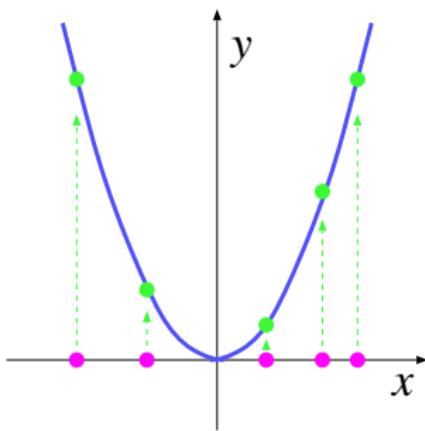
## Reduction From Sorting to Convex Hulls

- Suppose the instance of **SORTING** is the set of  $S' := \{x_1, x_2, \dots, x_n\} \subset \mathbb{R}$ .
- We transform  $S'$  into an instance of **CONVEXHULL** by mapping each real number  $x_i$  to the point  $(x_i, x_i^2)$ . All points of the resulting set  $S$  of points lie on the parabola  $y = x^2$ .



## Reduction From Sorting to Convex Hulls

- Suppose the instance of **SORTING** is the set of  $S' := \{x_1, x_2, \dots, x_n\} \subset \mathbb{R}$ .
- We transform  $S'$  into an instance of **CONVEXHULL** by mapping each real number  $x_i$  to the point  $(x_i, x_i^2)$ . All points of the resulting set  $S$  of points lie on the parabola  $y = x^2$ .
- The convex hull of  $S$  contains a list of vertices sorted by  $x$ -coordinates.
- One pass through this list will find the smallest element. The sorted numbers can be obtained by a second pass through this list.



## Complexity of Computing Convex Hulls

- If also the size  $h$  of the output is considered (in addition to the input size  $n$ ), then one can prove the lower bound  $\Omega(n \log h)$ .

## Complexity of Computing Convex Hulls

- If also the size  $h$  of the output is considered (in addition to the input size  $n$ ), then one can prove the lower bound  $\Omega(n \log h)$ .
- This lower bound is matched by a “marriage-before-conquest” algorithm and by Timothy Chan’s algorithm. (The latter algorithm is simpler and also extends to 3D.)

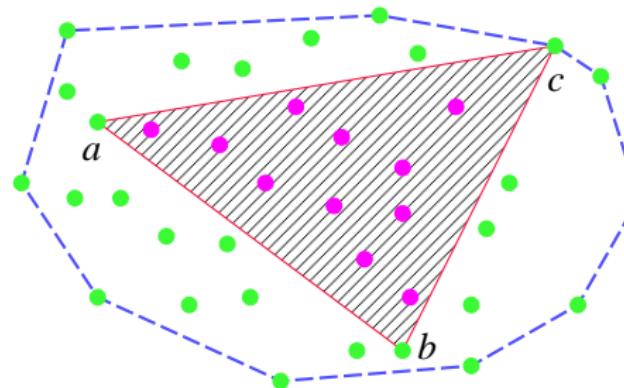
### Theorem 24 (Kirkpatrick&Seidel (1986), Chan (1996))

The convex hull of  $n$  points in the plane can be computed in  $O(n \log h)$  time and within  $O(n)$  storage, where  $h$  denotes the number of vertices of  $CH(S)$ .

# Discarding Internal Points

## Lemma 25

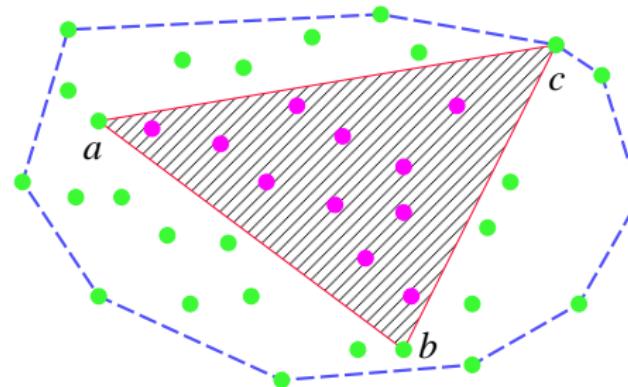
Consider three points  $a, b, c \in CH(S)$ . Then every point  $q$  that lies strictly within  $\Delta(a, b, c)$  is internal to  $CH(S)$ .



# Discarding Internal Points

## Lemma 25

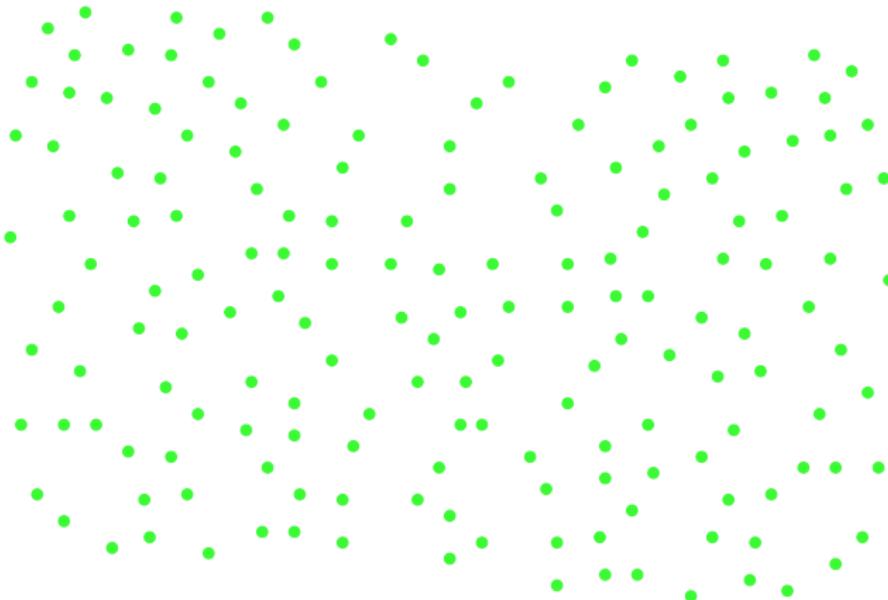
Consider three points  $a, b, c \in CH(S)$ . Then every point  $q$  that lies strictly within  $\Delta(a, b, c)$  is internal to  $CH(S)$ .



- In particular, no point strictly within  $\Delta(a, b, c)$  can be a vertex of the convex hull.
- This lemma can be generalized to any convex quadrangle (or polygon) whose vertices lie within  $CH(S)$ .

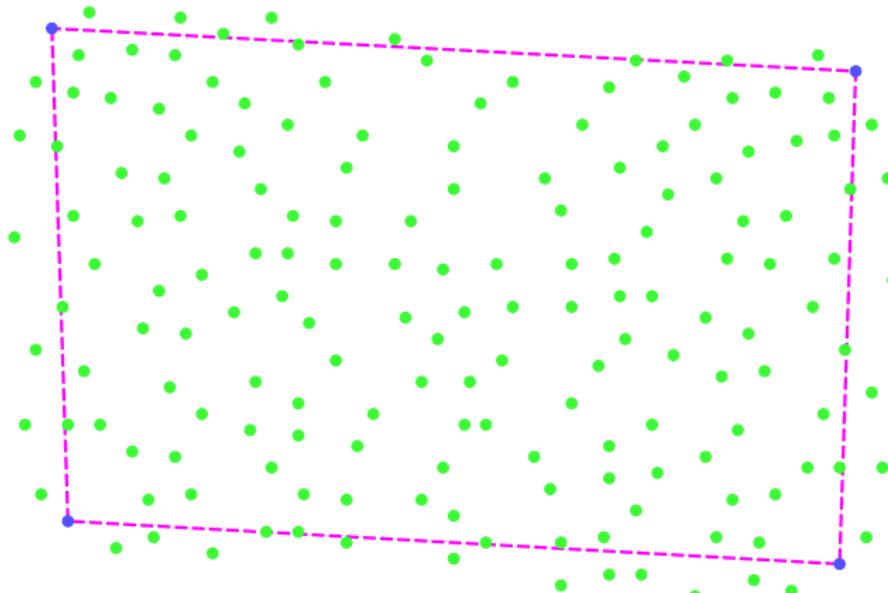
## Discarding Internal Points: Interior Elimination

- Discard all points within a large (axis-aligned) rectangle.



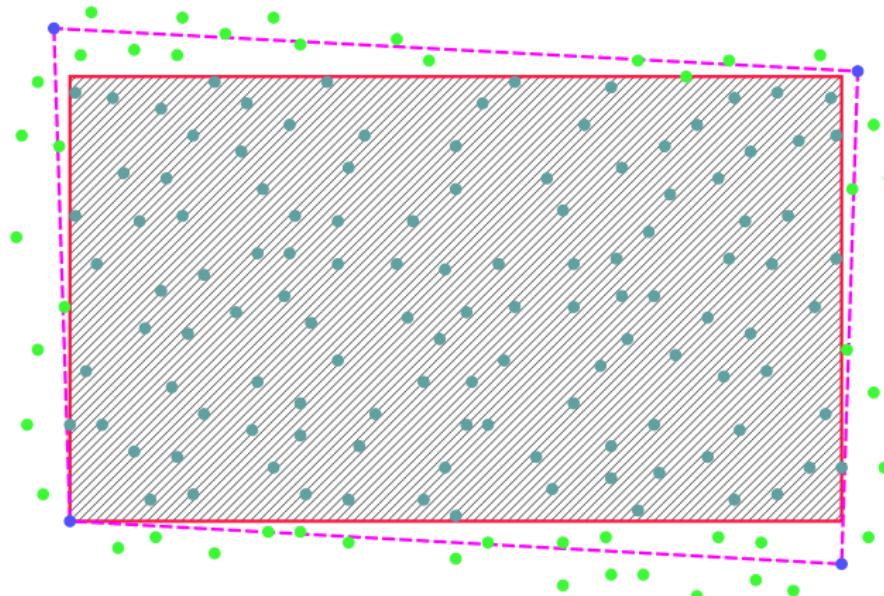
## Discarding Internal Points: Interior Elimination

- Discard all points within a large (axis-aligned) rectangle.



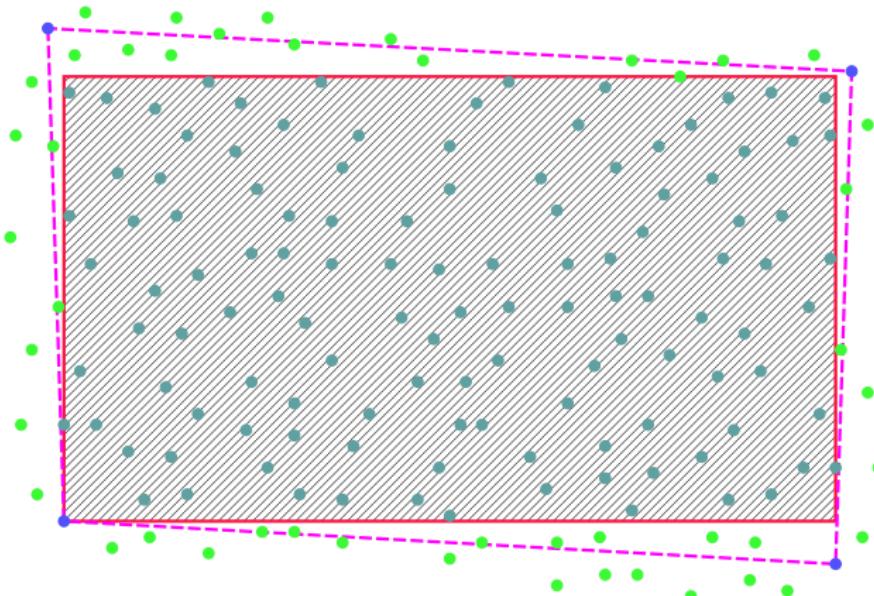
## Discarding Internal Points: Interior Elimination

- Discard all points within a large (axis-aligned) rectangle.



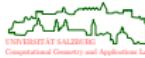
## Discarding Internal Points: Interior Elimination

- Discard all points within a large (axis-aligned) rectangle.
- Heuristic improvement; does not change worst-case complexity.



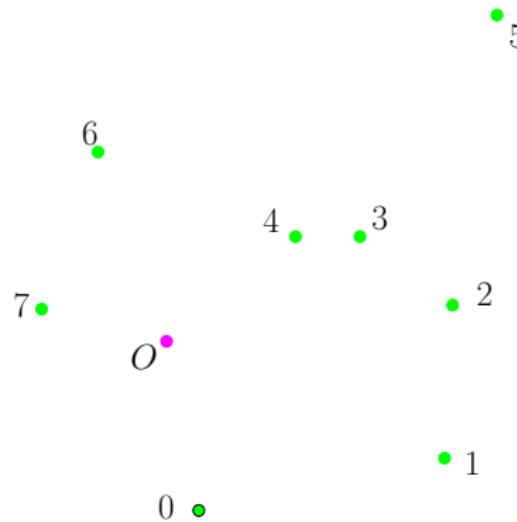
## Convex Hulls

- Basics
- **Graham's Scan**
- Gift Wrapping
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls



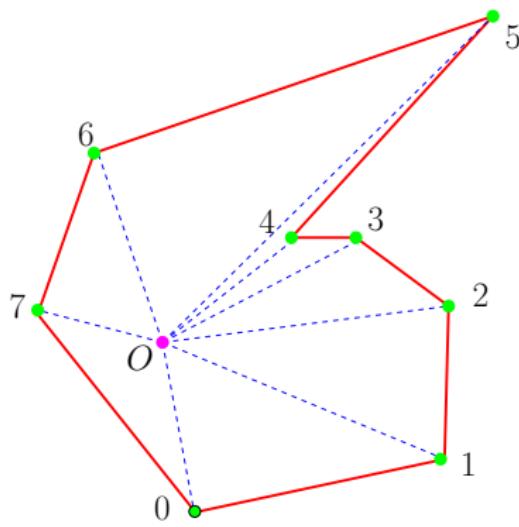
## Graham's Scan

- Find a point  $O$  internal to  $CH(S)$ , e.g. the center of three points of  $S$ .



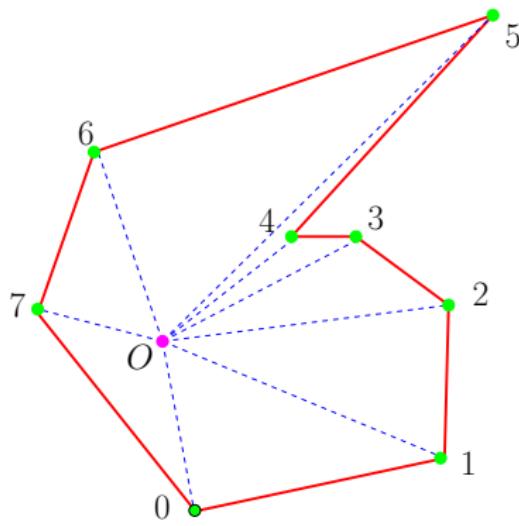
## Graham's Scan

- Find a point  $O$  internal to  $CH(S)$ , e.g, the center of three points of  $S$ .
- Sort the  $n$  points of  $S$  lexicographically on
  - ① polar angle relative to  $O$ ,
  - ② distance from  $O$ .



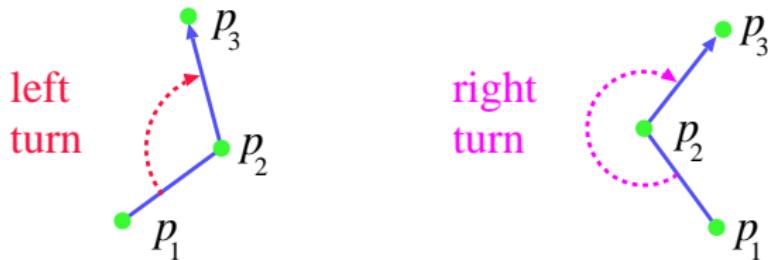
## Graham's Scan

- Find a point  $O$  internal to  $CH(S)$ , e.g., the center of three points of  $S$ .
- Sort the  $n$  points of  $S$  lexicographically on
  - ① polar angle relative to  $O$ ,
  - ② distance from  $O$ .
- Choose a point  $p_0 \in S$  guaranteed to be a vertex of  $CH(S)$ , and re-number the points.



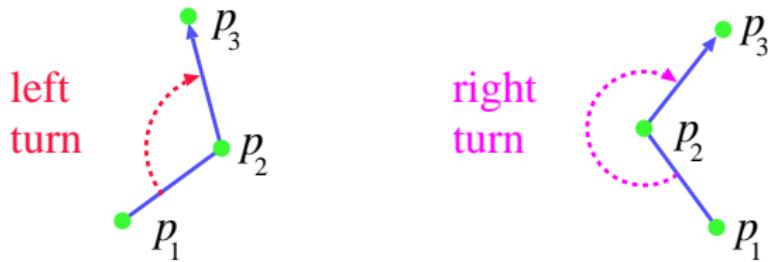
## Graham's Scan

- CCW scan algorithm: The algorithm repeatedly examines triangles defined by triples of consecutive points  $\triangle(p_i, p_{i+1}, p_{i+2})$ :
  - If  $\triangle(p_i, p_{i+1}, p_{i+2})$  is a left turn, advance to  $\triangle(p_{i+1}, p_{i+2}, p_{i+3})$ .



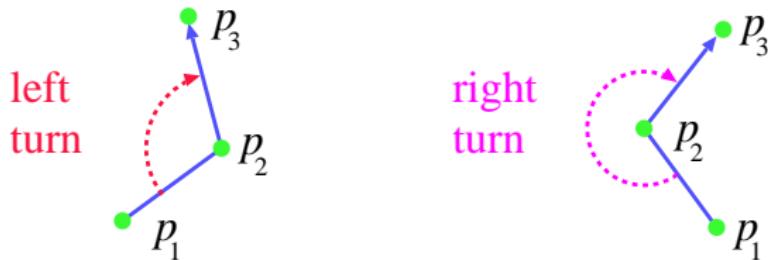
## Graham's Scan

- CCW scan algorithm: The algorithm repeatedly examines triangles defined by triples of consecutive points  $\triangle(p_i, p_{i+1}, p_{i+2})$ :
  - If  $\triangle(p_i, p_{i+1}, p_{i+2})$  is a left turn, advance to  $\triangle(p_{i+1}, p_{i+2}, p_{i+3})$ .
  - If  $\triangle(p_i, p_{i+1}, p_{i+2})$  is a right turn, eliminate  $p_{i+1}$  from  $S$  and backtrack to  $\triangle(p_{i-1}, p_i, p_{i+2})$ .



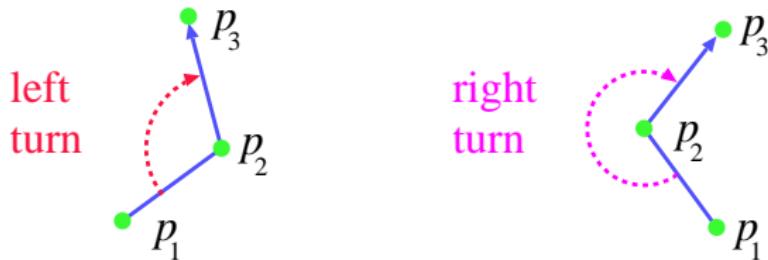
## Graham's Scan

- CCW scan algorithm: The algorithm repeatedly examines triangles defined by triples of consecutive points  $\triangle(p_i, p_{i+1}, p_{i+2})$ :
  - If  $\triangle(p_i, p_{i+1}, p_{i+2})$  is a left turn, advance to  $\triangle(p_{i+1}, p_{i+2}, p_{i+3})$ .
  - If  $\triangle(p_i, p_{i+1}, p_{i+2})$  is a right turn, eliminate  $p_{i+1}$  from  $S$  and backtrack to  $\triangle(p_{i-1}, p_i, p_{i+2})$ .
  - If  $p_i, p_{i+1}, p_{i+2}$  are collinear then eliminate  $p_{i+1}$  and advance to  $\triangle(p_i, p_{i+2}, p_{i+3})$ .



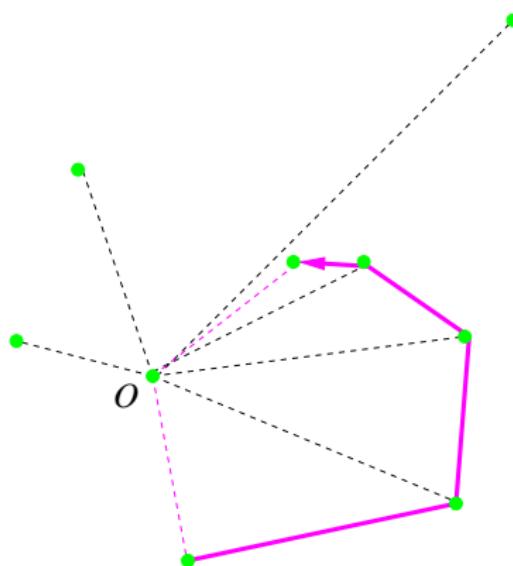
## Graham's Scan

- CCW scan algorithm: The algorithm repeatedly examines triangles defined by triples of consecutive points  $\triangle(p_i, p_{i+1}, p_{i+2})$ :
  - If  $\triangle(p_i, p_{i+1}, p_{i+2})$  is a left turn, advance to  $\triangle(p_{i+1}, p_{i+2}, p_{i+3})$ .
  - If  $\triangle(p_i, p_{i+1}, p_{i+2})$  is a right turn, eliminate  $p_{i+1}$  from  $S$  and backtrack to  $\triangle(p_{i-1}, p_i, p_{i+2})$ .
  - If  $p_i, p_{i+1}, p_{i+2}$  are collinear then eliminate  $p_{i+1}$  and advance to  $\triangle(p_i, p_{i+2}, p_{i+3})$ .
  - Scan ends when it returns to  $p_0$ .



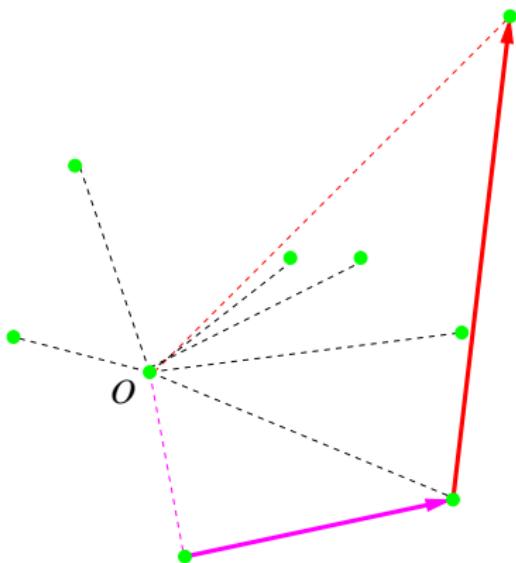
## Graham's Scan: Advancing and Backtracking

- Backtracking may occur more than once in succession, eliminating a sequence of points.



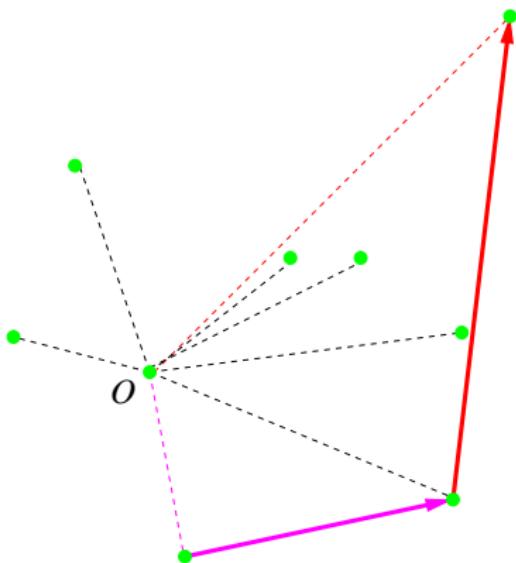
## Graham's Scan: Advancing and Backtracking

- Backtracking may occur more than once in succession, eliminating a sequence of points.



## Graham's Scan: Advancing and Backtracking

- Backtracking may occur more than once in succession, eliminating a sequence of points.
- Backtracking sure to stop at  $p_0$ .

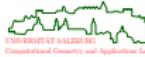


# Animation of Graham's Scan

# Analysis of Graham's Scan

## Theorem 26 (Complexity of Graham's Scan)

Graham's Scan computes the convex hull of  $n$  points in the plane in  $O(n \log n)$  time and within  $O(n)$  storage.



## Theorem 26 (Complexity of Graham's Scan)

Graham's Scan computes the convex hull of  $n$  points in the plane in  $O(n \log n)$  time and within  $O(n)$  storage.

*Proof:*

- ① Find a point within  $CH(S)$ :  $O(1)$ .
- ② Sort the points:  $O(n \log n)$ .
- ③ Find a point on  $CH(S)$ :  $O(n)$ .
- ④ Scan algorithm:  $O(n)$ .



# Analysis of Graham's Scan

## Theorem 26 (Complexity of Graham's Scan)

Graham's Scan computes the convex hull of  $n$  points in the plane in  $O(n \log n)$  time and within  $O(n)$  storage.

*Proof:*

- ① Find a point within  $CH(S)$ :  $O(1)$ .
- ② Sort the points:  $O(n \log n)$ .
- ③ Find a point on  $CH(S)$ :  $O(n)$ .
- ④ Scan algorithm:  $O(n)$ .

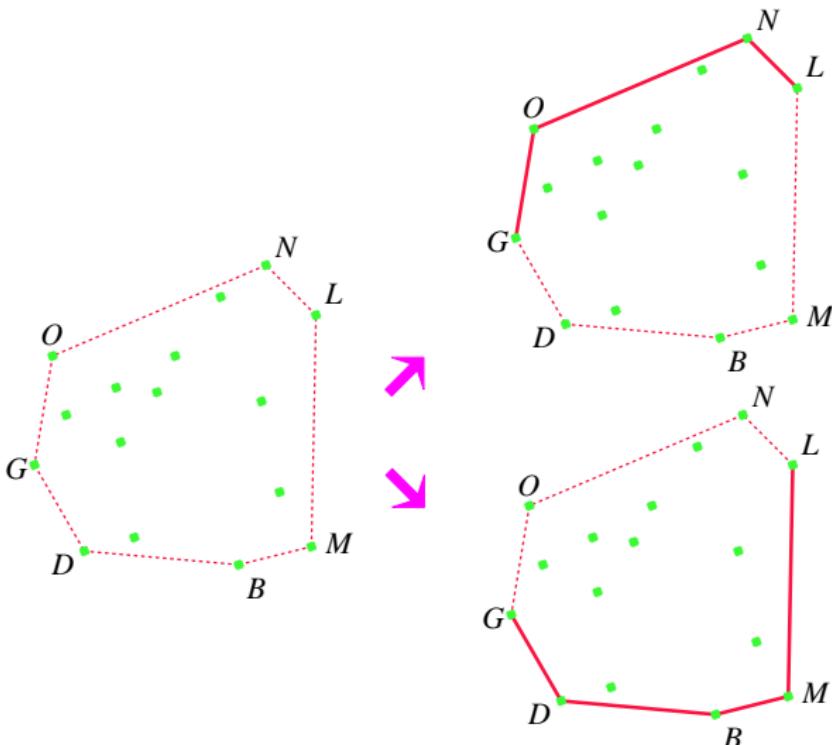
□

## Corollary 27

Graham's Scan computes the convex hull of a star-shaped polygon in linear time.

## Practice-Minded Simplification of Graham's Scan

- Compute upper and lower convex hull separately: then a conventional lexicographical sort with respect to  $x$ -coordinates (and  $y$ -coordinates) suffices.

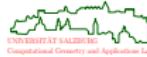


## Convex Hulls

- Basics
- Graham's Scan
- **Gift Wrapping**
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls

## Gift Wrapping

- Aka: Jarvis' March. Gift wrapping finds consecutive edges of  $CH(S)$ .

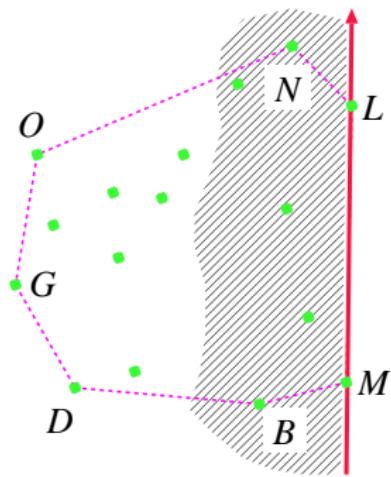


# Gift Wrapping

- Aka: Jarvis' March. Gift wrapping finds consecutive edges of  $CH(S)$ .

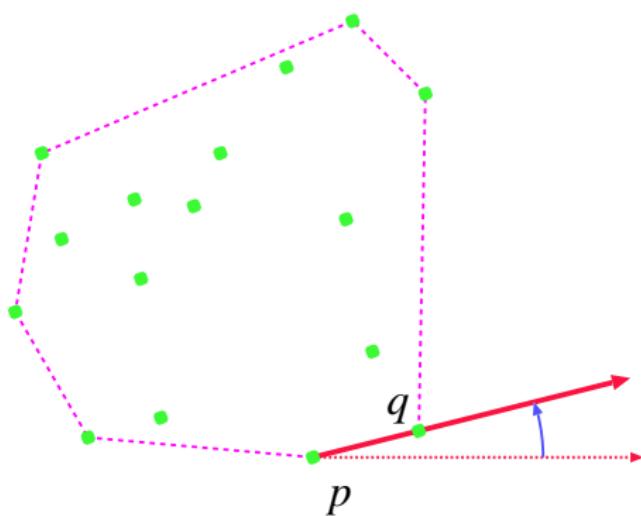
## Lemma 28

For  $p, q \in S$ , the line segment  $\overline{pq}$  is an edge of the convex hull  $CH(S)$  if and only if all other points of  $S$  lie on the line segment or to one side of it.



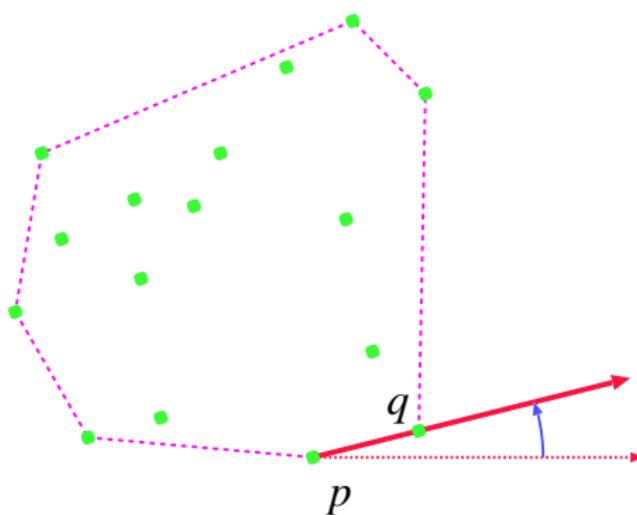
## Gift Wrapping: Basic Algorithm

- ① Find the point  $p$  of  $S$  which has minimum  $y$ -coordinate. Call it “pivot” point.



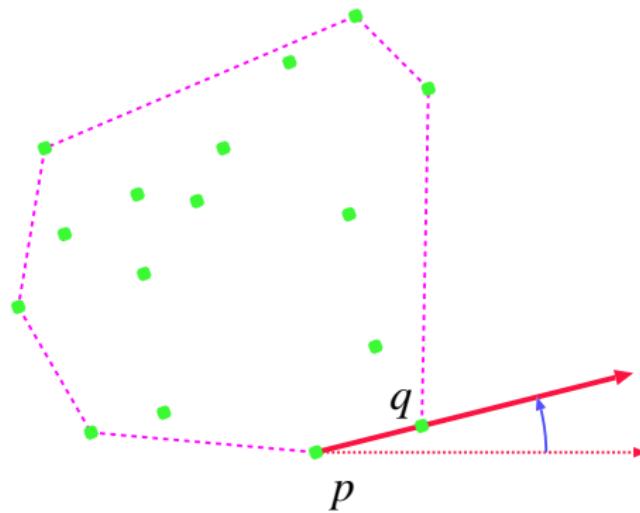
## Gift Wrapping: Basic Algorithm

- ① Find the point  $p$  of  $S$  which has minimum  $y$ -coordinate. Call it “pivot” point.
- ② Shoot a ray in direction of the positive  $x$ -axis, starting at the pivot point  $p$ .



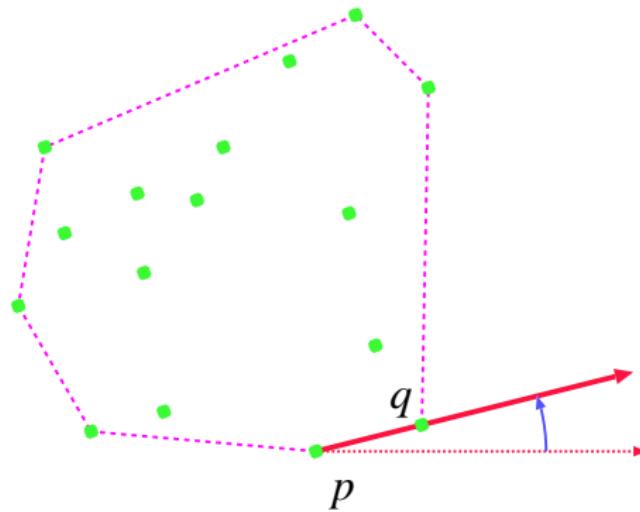
## Gift Wrapping: Basic Algorithm

- ① Find the point  $p$  of  $S$  which has minimum  $y$ -coordinate. Call it “pivot” point.
- ② Shoot a ray in direction of the positive  $x$ -axis, starting at the pivot point  $p$ .
- ③ Rotate the ray in CCW direction around  $p$  until another point  $q$  of  $S$  is met.

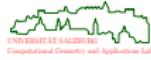


## Gift Wrapping: Basic Algorithm

- ① Find the point  $p$  of  $S$  which has minimum  $y$ -coordinate. Call it “pivot” point.
- ② Shoot a ray in direction of the positive  $x$ -axis, starting at the pivot point  $p$ .
- ③ Rotate the ray in CCW direction around  $p$  until another point  $q$  of  $S$  is met.
- ④ This yields the edge  $\overline{pq}$  of  $CH(S)$ , and  $q$  becomes our new pivot.



# Animation of Gift Wrapping



## Gift Wrapping: Details of the Rotation

- ① Suppose that  $p$  is the pivot, and that we rotate CCW. As usual, general position assumed.
- ② Choose a point  $q \in S \setminus \{p\}$  (at random), and let  $S' := S \setminus \{p, q\}$ .

## Gift Wrapping: Details of the Rotation

- ① Suppose that  $p$  is the pivot, and that we rotate CCW. As usual, general position assumed.
- ② Choose a point  $q \in S \setminus \{p\}$  (at random), and let  $S' := S \setminus \{p, q\}$ .
- ③ Test whether all points of  $S'$  lie on the left side of the line  $\overline{pq}$ :
  - a Choose  $r \in S'$  (at random).
  - b Delete it from  $S'$ .
  - c If  $r$  is right of  $\overline{pq}$  then let  $q := r$ .
  - d While  $S' \neq \emptyset$  go to Step 3a.

## Gift Wrapping: Details of the Rotation

- ① Suppose that  $p$  is the pivot, and that we rotate CCW. As usual, general position assumed.
- ② Choose a point  $q \in S \setminus \{p\}$  (at random), and let  $S' := S \setminus \{p, q\}$ .
- ③ Test whether all points of  $S'$  lie on the left side of the line  $\overline{pq}$ :
  - a Choose  $r \in S'$  (at random).
  - b Delete it from  $S'$ .
  - c If  $r$  is right of  $\overline{pq}$  then let  $q := r$ .
  - d While  $S' \neq \emptyset$  go to Step 3a.
- ④ If  $S'$  is empty then all points of  $S \setminus \{p, q\}$  lie to the left of  $\overline{pq}$ .

## Theorem 29 (Complexity of gift wrapping)

Gift wrapping computes the convex hull of  $n$  points in the plane in  $O(n \cdot h)$  time and within  $O(n)$  storage, where  $h$  denotes the number of vertices of  $CH(S)$ .

## Theorem 29 (Complexity of gift wrapping)

Gift wrapping computes the convex hull of  $n$  points in the plane in  $O(n \cdot h)$  time and within  $O(n)$  storage, where  $h$  denotes the number of vertices of  $CH(S)$ .

*Proof:*

- Let  $n$  be the total number of points in  $S$ , and  $h$  the number of vertices of  $CH(S)$ .
- Each new vertex of the hull is discovered in  $O(n)$  time.
- Thus, the entire algorithm has a complexity of  $O(n \cdot h)$ .



## Theorem 29 (Complexity of gift wrapping)

Gift wrapping computes the convex hull of  $n$  points in the plane in  $O(n \cdot h)$  time and within  $O(n)$  storage, where  $h$  denotes the number of vertices of  $CH(S)$ .

*Proof:*

- Let  $n$  be the total number of points in  $S$ , and  $h$  the number of vertices of  $CH(S)$ .
  - Each new vertex of the hull is discovered in  $O(n)$  time.
  - Thus, the entire algorithm has a complexity of  $O(n \cdot h)$ .
- 
- Obviously, gift wrapping will benefit from interior elimination.

## Convex Hulls

- Basics
- Graham's Scan
- Gift Wrapping
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls

## Divide-and-Conquer Convex Hull

- ① If  $|S| \leq k_0$ , where  $k_0$  is a small integer (e.g.,  $k_0 = 3$ ), then construct the convex hull  $CH(S)$  directly by some method and stop, else go to Step 2.

## Divide-and-Conquer Convex Hull

- ① If  $|S| \leq k_0$ , where  $k_0$  is a small integer (e.g.,  $k_0 = 3$ ), then construct the convex hull  $CH(S)$  directly by some method and stop, else go to Step 2.
- ② Partition the set  $S$  arbitrarily into two subsets  $S_1$  and  $S_2$  of approximately equal sizes.



## Divide-and-Conquer Convex Hull

- ① If  $|S| \leq k_0$ , where  $k_0$  is a small integer (e.g.,  $k_0 = 3$ ), then construct the convex hull  $CH(S)$  directly by some method and stop, else go to Step 2.
- ② Partition the set  $S$  arbitrarily into two subsets  $S_1$  and  $S_2$  of approximately equal sizes.
- ③ Recursively find the convex hulls  $CH(S_1)$  and  $CH(S_2)$ .



## Divide-and-Conquer Convex Hull

- ① If  $|S| \leq k_0$ , where  $k_0$  is a small integer (e.g.,  $k_0 = 3$ ), then construct the convex hull  $CH(S)$  directly by some method and stop, else go to Step 2.
- ② Partition the set  $S$  arbitrarily into two subsets  $S_1$  and  $S_2$  of approximately equal sizes.
- ③ Recursively find the convex hulls  $CH(S_1)$  and  $CH(S_2)$ .
- ④ Merge the two hulls together to form  $CH(S)$ .

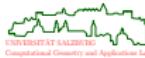


## Divide-and-Conquer Convex Hull

- ① If  $|S| \leq k_0$ , where  $k_0$  is a small integer (e.g.,  $k_0 = 3$ ), then construct the convex hull  $CH(S)$  directly by some method and stop, else go to Step 2.
- ② Partition the set  $S$  arbitrarily into two subsets  $S_1$  and  $S_2$  of approximately equal sizes.
- ③ Recursively find the convex hulls  $CH(S_1)$  and  $CH(S_2)$ .
- ④ Merge the two hulls together to form  $CH(S)$ .

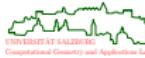
**function**  $CH(S)$

01.     $n := |S|$
02.    **if**  $n \leq 3$  **then**
03.        construct  $CH(S)$  by brute force;
04.        **return**  $CH(S)$ ;
05.    **else**
06.         $S_1 := \{p_1, p_2, \dots, p_{\lfloor n/2 \rfloor}\}$  and  $S_2 := \{p_{\lfloor n/2 \rfloor + 1}, \dots, p_n\}$ ;
07.         $P_1 := CH(S_1)$  and  $P_2 := CH(S_2)$ ;
08.         $P := MERGE\_CH(P_1, P_2)$ ;
09.        **return**  $P$ ;
10.    **end**



## Divide-and-Conquer Convex Hull: Algorithm

- The convex hull of the union of the two subsets is the same as the convex hull of the union of the convex hulls of the two subsets.
- Computing the convex hull of  $CH(S_1) \cup CH(S_2)$  is relatively simple since  $CH(S_1)$  and  $CH(S_2)$  are convex polygons  $P_1, P_2$  and, thus, have a natural ordering of their vertices.



## Definition 30 (Supporting line, Dt.: Stützgerade)

A *supporting line* of a convex polygon  $P$  is a straight line  $\ell$  passing through a vertex of  $P$  such that the interior of  $P$  lies entirely to one side of  $\ell$ .

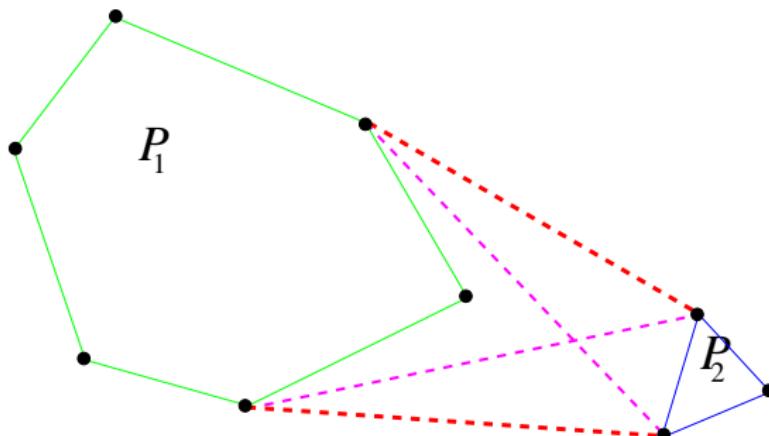


## Divide-and-Conquer Convex Hull: Supporting Lines

### Definition 30 (Supporting line, Dt.: Stützgerade)

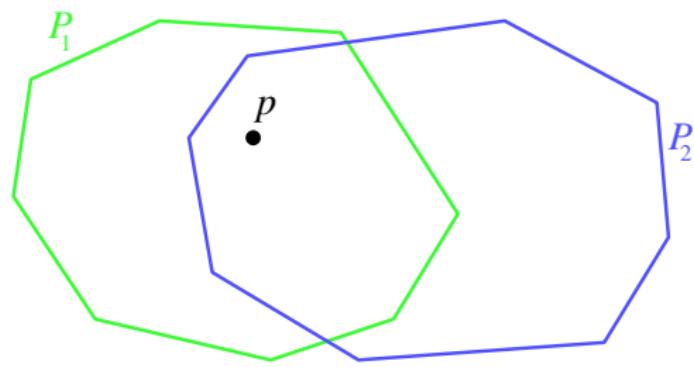
A *supporting line* of a convex polygon  $P$  is a straight line  $\ell$  passing through a vertex of  $P$  such that the interior of  $P$  lies entirely to one side of  $\ell$ .

- This definition is readily generalized to general convex sets.
- Two convex polygons  $P_1$  and  $P_2$ , where no polygon is entirely contained within the other polygon, have up to four *common supporting lines*.



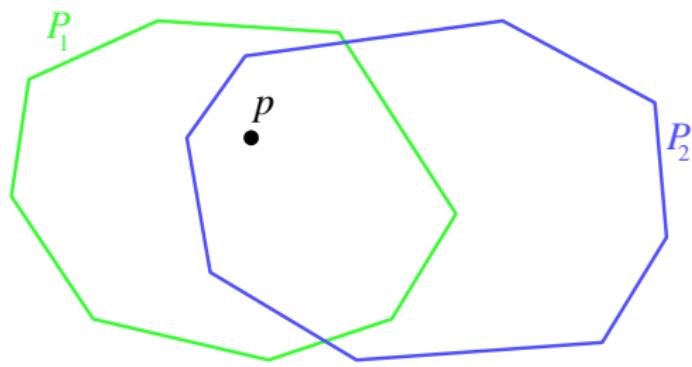
## Divide-and-Conquer Convex Hull: Merge

- ① Find a point  $p$  that is internal to  $P_1$ ; e.g., the centroid. Note that this point  $p$  will be internal to  $CH(P_1 \cup P_2)$ .



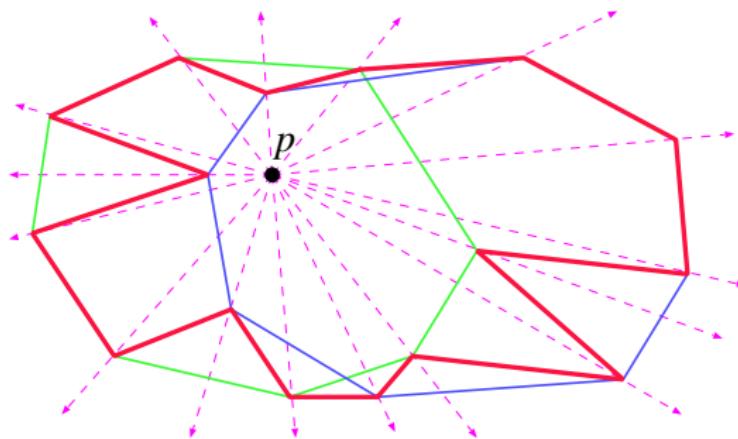
## Divide-and-Conquer Convex Hull: Merge

- ① Find a point  $p$  that is internal to  $P_1$ ; e.g., the centroid. Note that this point  $p$  will be internal to  $CH(P_1 \cup P_2)$ .
- ② Determine whether or not  $p$  is internal to  $P_2$ .



## Divide-and-Conquer Convex Hull: Merge

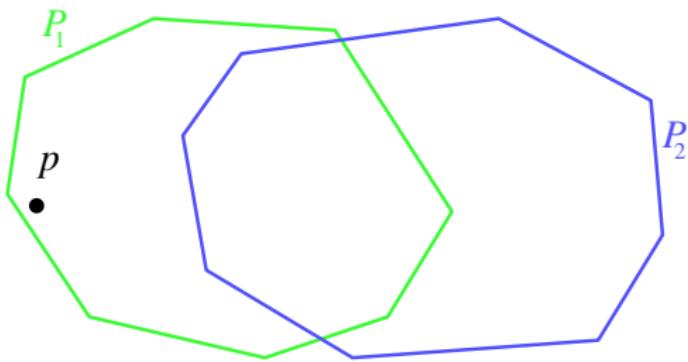
- ① Find a point  $p$  that is internal to  $P_1$ ; e.g., the centroid. Note that this point  $p$  will be internal to  $CH(P_1 \cup P_2)$ .
- ② Determine whether or not  $p$  is internal to  $P_2$ .
- ③ Case: Point  $p$  is internal to  $P_2$ :
  - (a) Merge  $P_1$  and  $P_2$  into one polygon that is star-shaped, with  $p$  within its nucleus.
  - (b) Go to Step 5 (Graham's Scan).



## Divide-and-Conquer Convex Hull: Merge

④ Case: Point  $p$  is not internal to  $P_2$ :

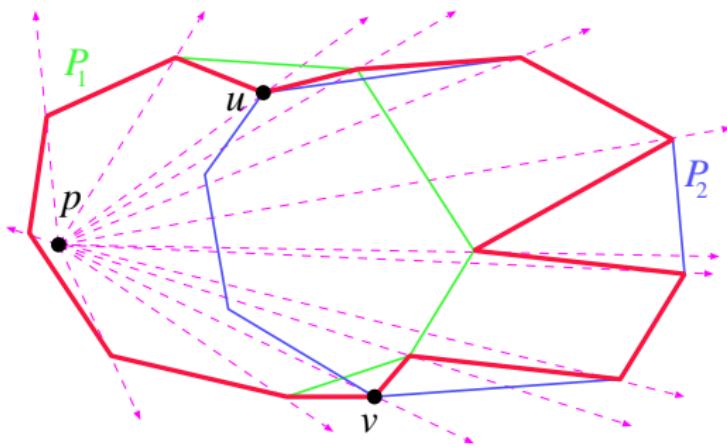
- Find vertices  $u$  and  $v$  on  $P_2$  such that  $\overline{pu}$  and  $\overline{pv}$  are supporting lines of  $P_2$ .
- Split  $P_2$  into two chains at  $u$  and  $v$ .
- Merge  $P_1$  and one chain of  $P_2$  into one polygon that is star-shaped, with  $p$  within its nucleus.



## Divide-and-Conquer Convex Hull: Merge

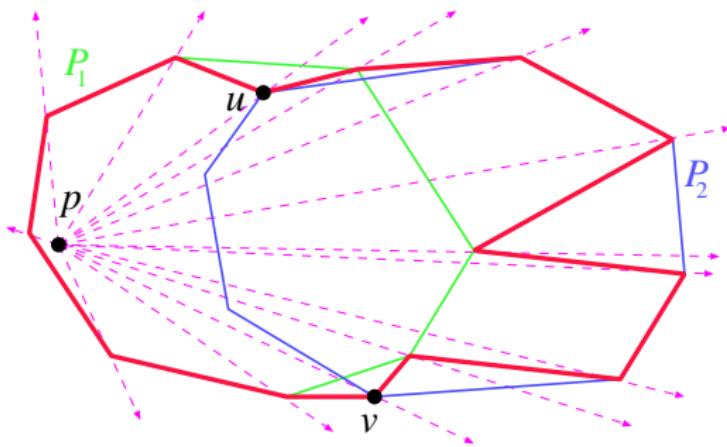
④ Case: Point  $p$  is not internal to  $P_2$ :

- Find vertices  $u$  and  $v$  on  $P_2$  such that  $\overline{pu}$  and  $\overline{pv}$  are supporting lines of  $P_2$ .
- Split  $P_2$  into two chains at  $u$  and  $v$ .
- Merge  $P_1$  and one chain of  $P_2$  into one polygon that is star-shaped, with  $p$  within its nucleus.



## Divide-and-Conquer Convex Hull: Merge

- ④ Case: Point  $p$  is not internal to  $P_2$ :
  - (a) Find vertices  $u$  and  $v$  on  $P_2$  such that  $\overline{pu}$  and  $\overline{pv}$  are supporting lines of  $P_2$ .
  - (b) Split  $P_2$  into two chains at  $u$  and  $v$ .
  - (c) Merge  $P_1$  and one chain of  $P_2$  into one polygon that is star-shaped, with  $p$  within its nucleus.
- ⑤ Apply Graham's Scan to the resulting polygon.



## Divide-and-Conquer Convex Hull: Analysis

- If polygon  $P_1$  has  $n_1$  vertices and polygon  $P_2$  has  $n_2$  vertices, then the merge algorithm computes  $CH(P_1 \cup P_2)$  in  $O(n_1 + n_2)$  time.
- Obviously, an  $O(n)$  merge yields an  $O(n \log n)$  time bound for this divide-and-conquer algorithm.



## Divide-and-Conquer Convex Hull: Analysis

- If polygon  $P_1$  has  $n_1$  vertices and polygon  $P_2$  has  $n_2$  vertices, then the merge algorithm computes  $CH(P_1 \cup P_2)$  in  $O(n_1 + n_2)$  time.
- Obviously, an  $O(n)$  merge yields an  $O(n \log n)$  time bound for this divide-and-conquer algorithm.

### Theorem 31 (Complexity of divide&conquer convex hull)

The divide&conquer algorithm computes the convex hull of  $n$  points in the plane in  $O(n \log n)$  time and within  $O(n)$  storage.



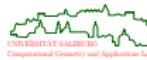
## Convex Hulls

- Basics
- Graham's Scan
- Gift Wrapping
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls

# Online Convex Hulls

- Terminology:

**static data:** a set  $S$  of objects remains fixed between operations (e.g., between queries in repetitive mode).

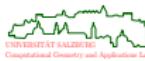


# Online Convex Hulls

- Terminology:

**static data:** a set  $S$  of objects remains fixed between operations (e.g., between queries in repetitive mode).

**online data:** a set  $S$  of objects is not known completely a-priori; rather, insertions of objects are permitted; this implies that  $S$  must be in some updatable “incremental” data structure that supports insertions.



# Online Convex Hulls

- Terminology:

**static data:** a set  $S$  of objects remains fixed between operations (e.g., between queries in repetitive mode).

**online data:** a set  $S$  of objects is not known completely a-priori; rather, insertions of objects are permitted; this implies that  $S$  must be in some updatable “incremental” data structure that supports insertions.

**dynamic data:** a set  $S$  of objects may change between operations due to insertions and deletions of objects; this implies that  $S$  must be in some updatable “dynamic” data structure.

# Online Convex Hulls

- Terminology:

**static data:** a set  $S$  of objects remains fixed between operations (e.g., between queries in repetitive mode).

**online data:** a set  $S$  of objects is not known completely a-priori; rather, insertions of objects are permitted; this implies that  $S$  must be in some updatable “incremental” data structure that supports insertions.

**dynamic data:** a set  $S$  of objects may change between operations due to insertions and deletions of objects; this implies that  $S$  must be in some updatable “dynamic” data structure.

**mobile data:** individual or all members of a set  $S$  of objects may move between operations; this implies that  $S$  must be in some updatable “kinetic” data structure.



# Online Convex Hulls

- Terminology:

**static data:** a set  $S$  of objects remains fixed between operations (e.g., between queries in repetitive mode).

**online data:** a set  $S$  of objects is not known completely a-priori; rather, insertions of objects are permitted; this implies that  $S$  must be in some updatable “incremental” data structure that supports insertions.

**dynamic data:** a set  $S$  of objects may change between operations due to insertions and deletions of objects; this implies that  $S$  must be in some updatable “dynamic” data structure.

**mobile data:** individual or all members of a set  $S$  of objects may move between operations; this implies that  $S$  must be in some updatable “kinetic” data structure.

- Lower bounds for the “static” version of a problem extend trivially to all other versions where objects are allowed to change.



## Theorem 32

The convex hull of  $n$  points in the plane can be computed incrementally (and online) in  $O(n \log n)$  time and within  $O(n)$  space.

*Proof:*

- Assume the points are inserted in the sequence  $p_1, p_2, \dots, p_n$ . Let  $p_i$  be the current point and  $C_{i-1} := CH(\{p_1, p_2, \dots, p_{i-1}\})$ .
- Case:  $p_i$  is internal to  $C_{i-1}$ . Then  $C_i = C_{i-1}$ .
- Case:  $p_i$  is external to  $C_{i-1}$ . We find the supporting lines from  $p_i$  to  $C_{i-1}$ .
- Both cases can be handled in time  $O(\log n)$ , based on a hierarchical representation of  $C_{i-1}$ .

□

## Theorem 32

The convex hull of  $n$  points in the plane can be computed incrementally (and online) in  $O(n \log n)$  time and within  $O(n)$  space.

### Proof:

- Assume the points are inserted in the sequence  $p_1, p_2, \dots, p_n$ . Let  $p_i$  be the current point and  $C_{i-1} := CH(\{p_1, p_2, \dots, p_{i-1}\})$ .
  - Case:  $p_i$  is internal to  $C_{i-1}$ . Then  $C_i = C_{i-1}$ .
  - Case:  $p_i$  is external to  $C_{i-1}$ . We find the supporting lines from  $p_i$  to  $C_{i-1}$ .
  - Both cases can be handled in time  $O(\log n)$ , based on a hierarchical representation of  $C_{i-1}$ .
- 
- If insertions and deletions are to be supported then the update time goes up to  $O(\log^2 n)$ . Note that points may re-surface on the convex hull when a hull vertex is deleted!



## Convex Hulls

- Basics
- Graham's Scan
- Gift Wrapping
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls

## Convex Hull of a Simple Polygon

- Given is the sequence  $(p_1, p_2, \dots, p_n)$  of  $n$  points in  $\mathbb{R}^2$  which form the vertices of a simple polygon  $P$ .
- Obviously,  $CH(P)$  can be computed in  $O(n \log n)$  time.
- Can we do any better?



## Convex Hull of a Simple Polygon

- Given is the sequence  $(p_1, p_2, \dots, p_n)$  of  $n$  points in  $\mathbb{R}^2$  which form the vertices of a simple polygon  $P$ .
- Obviously,  $CH(P)$  can be computed in  $O(n \log n)$  time.
- Can we do any better? Note: The lower bound for computing the convex hull of  $n$  points does not carry over to this problem!



## Convex Hull of a Simple Polygon

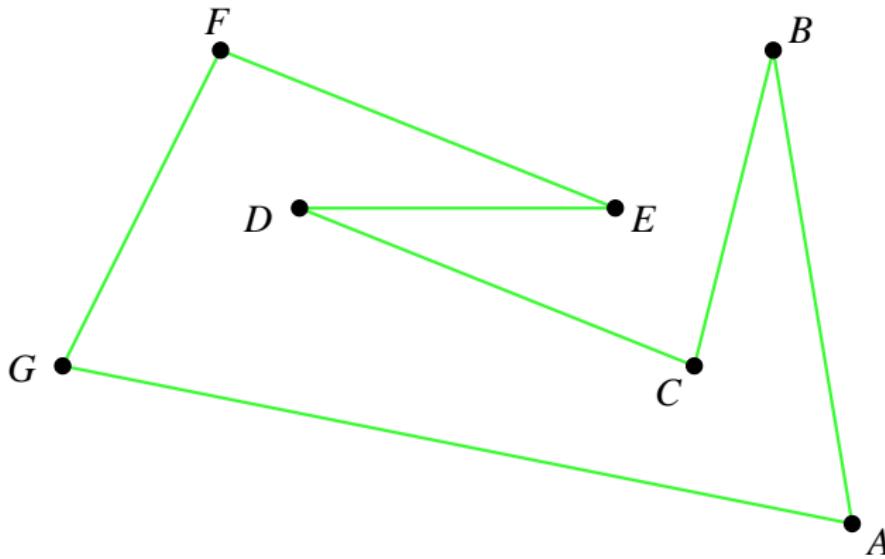
- Given is the sequence  $(p_1, p_2, \dots, p_n)$  of  $n$  points in  $\mathbb{R}^2$  which form the vertices of a simple polygon  $P$ .
- Obviously,  $CH(P)$  can be computed in  $O(n \log n)$  time.
- Can we do any better? Note: The lower bound for computing the convex hull of  $n$  points does not carry over to this problem!
- Recall that Graham's Scan runs in linear time when applied to a star-shaped polygon.
- Thus, the fact that the points are vertices of a polygon can be expected to help when designing a linear-time algorithm.

## Convex Hull of a Simple Polygon

- Given is the sequence  $(p_1, p_2, \dots, p_n)$  of  $n$  points in  $\mathbb{R}^2$  which form the vertices of a simple polygon  $P$ .
- Obviously,  $CH(P)$  can be computed in  $O(n \log n)$  time.
- Can we do any better? Note: The lower bound for computing the convex hull of  $n$  points does not carry over to this problem!
- Recall that Graham's Scan runs in linear time when applied to a star-shaped polygon.
- Thus, the fact that the points are vertices of a polygon can be expected to help when designing a linear-time algorithm.
- Caveats:
  - Several invalid linear-time “algorithms” were published in the early days of computational geometry.
  - Graham's Scan does not work properly for arbitrary simple polygons!

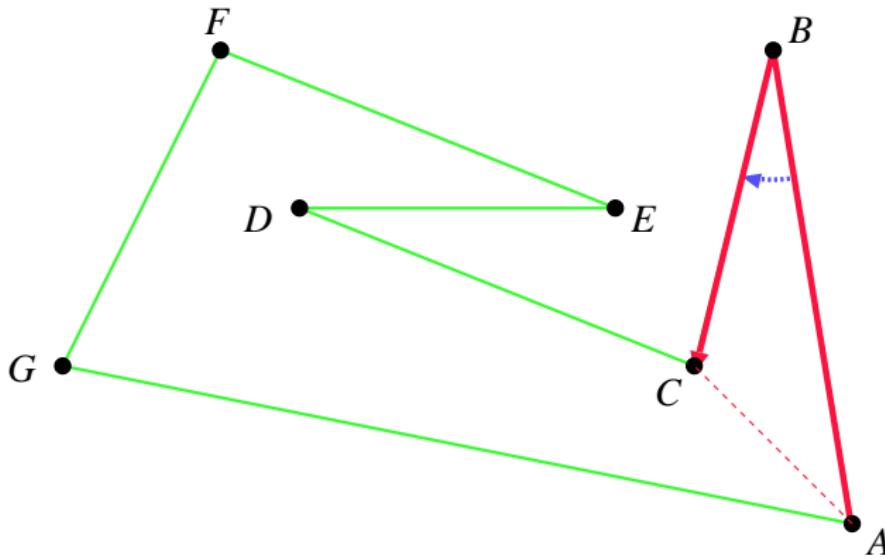
## Convex Hull of a Simple Polygon

- Graham's Scan does not work properly for arbitrary simple polygons!



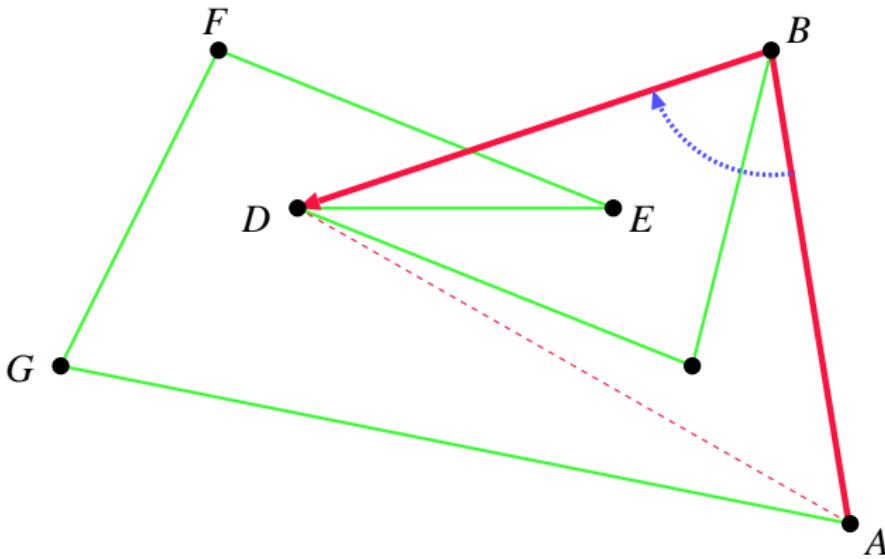
## Convex Hull of a Simple Polygon

- Graham's Scan does not work properly for arbitrary simple polygons!



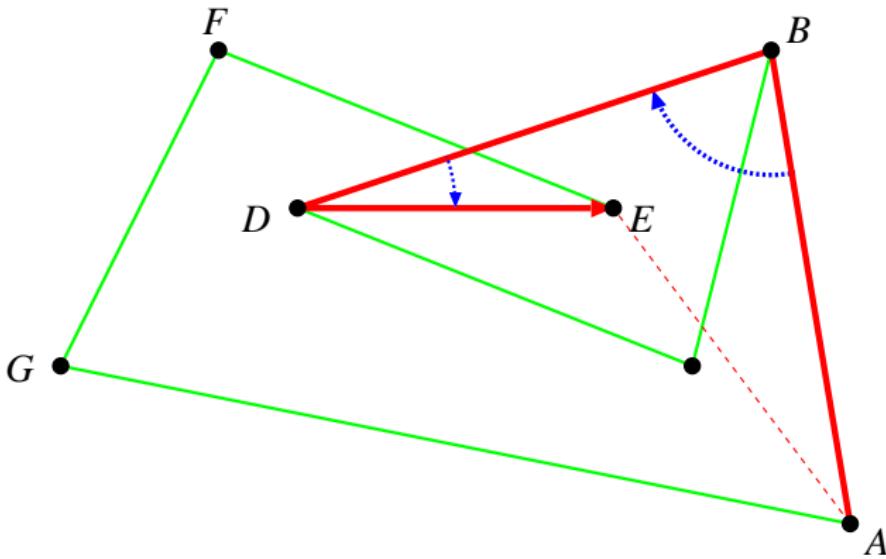
## Convex Hull of a Simple Polygon

- Graham's Scan does not work properly for arbitrary simple polygons!



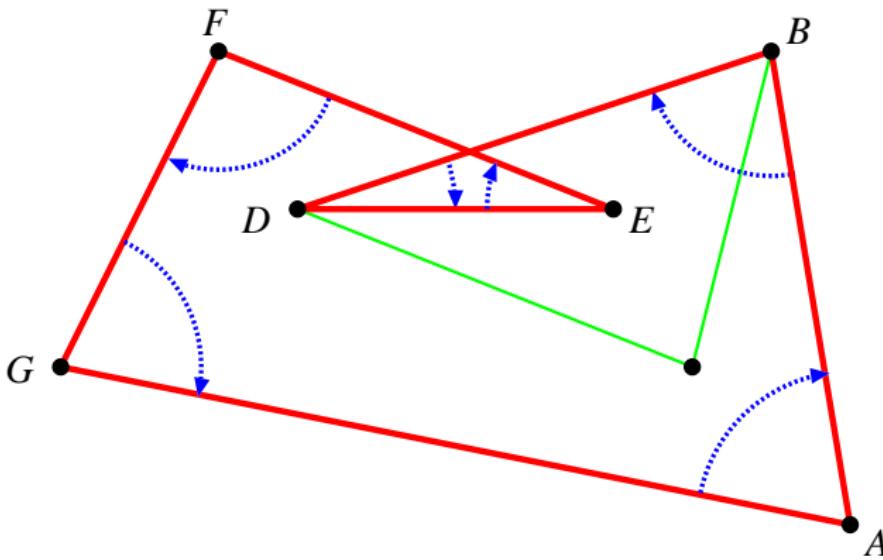
## Convex Hull of a Simple Polygon

- Graham's Scan does not work properly for arbitrary simple polygons!



## Convex Hull of a Simple Polygon

- Graham's Scan does not work properly for arbitrary simple polygons!



## Convex Hull of a Simple Polygon: Melkman's Algorithm

- Melkman's algorithm operates on a double-ended queue ("deque")  
 $< d_b, \dots, d_t >$ , with  $d_b = d_t$ ; the  $d_i$ 's will represent vertices of the convex hull.
- Deque operations:
  - Push( $v$ ) increments  $t$  by one, and inserts  $v$  at the new top;
  - Pop( $d_t$ ) deletes the top element and decrements  $t$  by one;
  - Insert( $v$ ) decrements  $b$  by one, and inserts  $v$  at the new bottom;
  - Delete( $d_b$ ) deletes the bottom element and increments  $b$  by one.



## Convex Hull of a Simple Polygon: Melkman's Algorithm

- Melkman's algorithm operates on a double-ended queue ("deque")  
 $< d_b, \dots, d_t >$ , with  $d_b = d_t$ ; the  $d_i$ 's will represent vertices of the convex hull.
- Deque operations:
  - Push( $v$ ) increments  $t$  by one, and inserts  $v$  at the new top;
  - Pop( $d_t$ ) deletes the top element and decrements  $t$  by one;
  - Insert( $v$ ) decrements  $b$  by one, and inserts  $v$  at the new bottom;
  - Delete( $d_b$ ) deletes the bottom element and increments  $b$  by one.
- Melkman's algorithm incrementally computes the convex hull of the polygon by adding one vertex at a time.
- A deque  $D$  is used to maintain the vertices of the convex hull constructed so far in CW order.

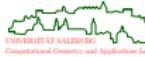
## Convex Hull of a Simple Polygon: Melkman's Algorithm

- Melkman's algorithm operates on a double-ended queue ("deque")  
 $< d_b, \dots, d_t >$ , with  $d_b = d_t$ ; the  $d_i$ 's will represent vertices of the convex hull.
- Deque operations:
  - Push( $v$ ) increments  $t$  by one, and inserts  $v$  at the new top;
  - Pop( $d_t$ ) deletes the top element and decrements  $t$  by one;
  - Insert( $v$ ) decrements  $b$  by one, and inserts  $v$  at the new bottom;
  - Delete( $d_b$ ) deletes the bottom element and increments  $b$  by one.
- Melkman's algorithm incrementally computes the convex hull of the polygon by adding one vertex at a time.
- A deque  $D$  is used to maintain the vertices of the convex hull constructed so far in CW order.
- The input polygon needs to be oriented CW.
- In the pseudo-code the vertices are retrieved online from "input", and an actual implementation needs to check for an end of the input data.

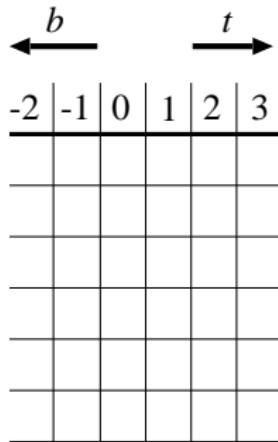
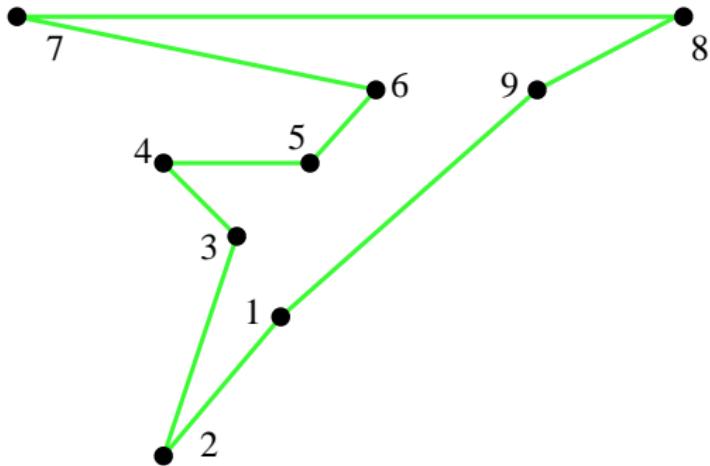
# Convex Hull of a Simple Polygon: Melkman's Algorithm

## Algorithm Melkman's Algorithm

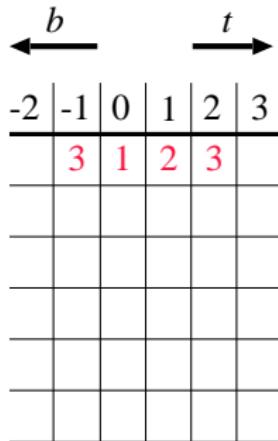
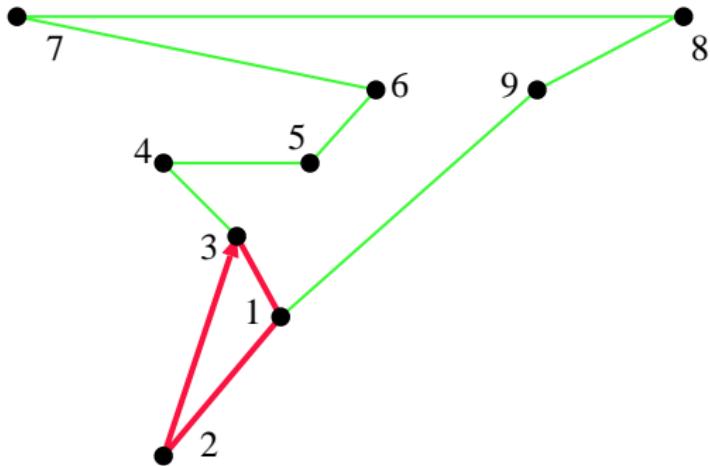
1.  $t \leftarrow -1; b \leftarrow 0;$  (\* The current convex hull is maintained in the deque  $D$  \*)
2.  $v_1 \leftarrow \text{input}; v_2 \leftarrow \text{input}; v_3 \leftarrow \text{input};$  (\* Obtain vertices in CW order \*)
3. **if**  $\det(v_1, v_2, v_3) < 0$  **then** (\* Initialize  $D$  \*)
4.   Push( $v_1$ ); Push( $v_2$ );
5. **else**
6.   Push( $v_2$ ); Push( $v_1$ );
7. Push( $v_3$ ); Insert( $v_3$ );
8. **repeat**
9.   **repeat**
10.     $v \leftarrow \text{input};$
11.    **until**  $\det(d_b, d_{b+1}, v) > 0$  **or**  $\det(d_{t-1}, d_t, v) > 0$  (\* Skip  $v$  if interior to  $D$  \*)
12.   **while**  $\det(d_{t-1}, d_t, v) > 0$  **do**
13.     Pop( $d_t$ ); (\* Delete interior vertices from top of  $D$  \*)
14.     Push( $v$ ); (\* Insert  $v$  at top of  $D$  \*)
15.   **while**  $\det(d_b, d_{b+1}, v) > 0$  **do**
16.     Delete( $d_b$ ); (\* Delete interior vertices from bottom of  $D$  \*)
17.     Insert( $v$ );
18. **until** input is empty.



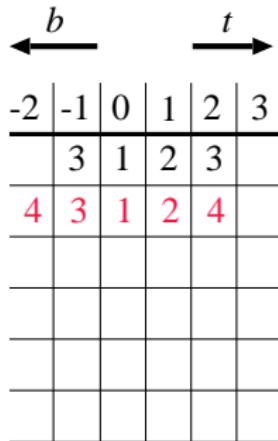
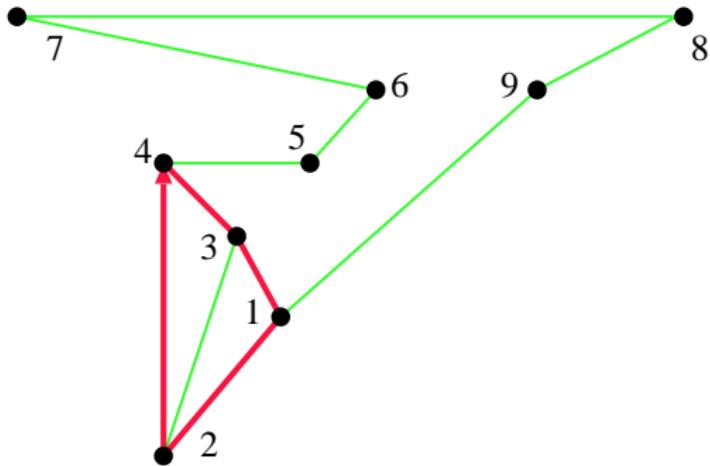
# Animation of Melkman's Algorithm



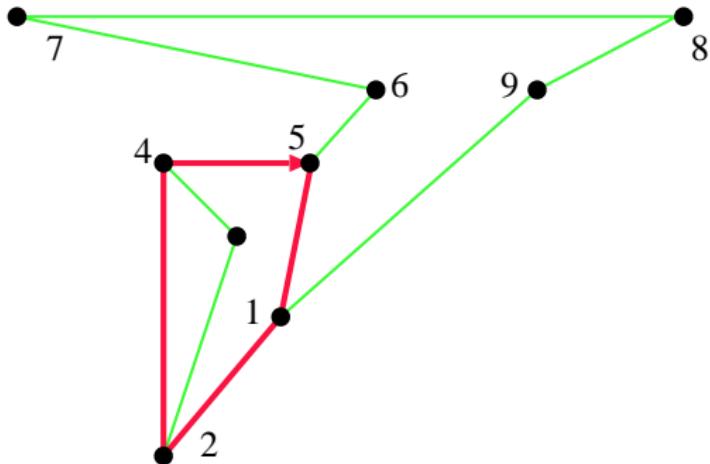
# Animation of Melkman's Algorithm



# Animation of Melkman's Algorithm



# Animation of Melkman's Algorithm

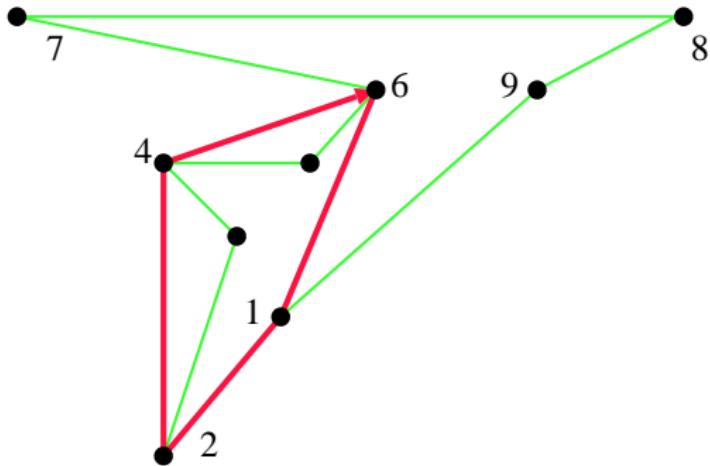


$b$  ← →  $t$

-2	-1	0	1	2	3
	3	1	2	3	
4	3	1	2	4	
	5	1	2	4	5



# Animation of Melkman's Algorithm

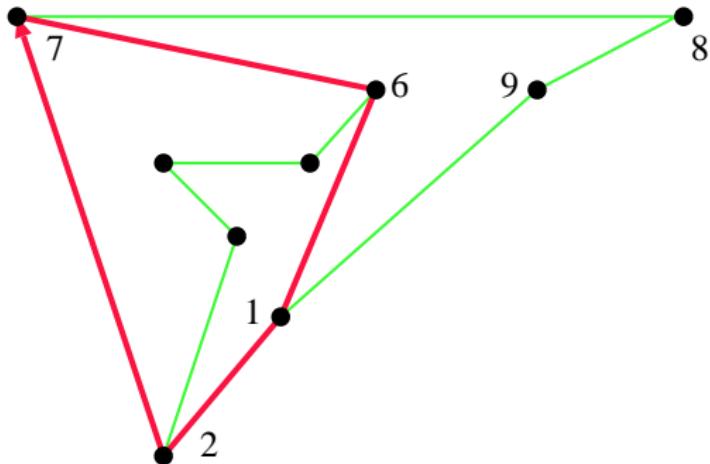


$b$  ← →  $t$

-2	-1	0	1	2	3
3	1	2	3		
4	3	1	2	4	
5	1	2	4	5	
6	1	2	4	6	



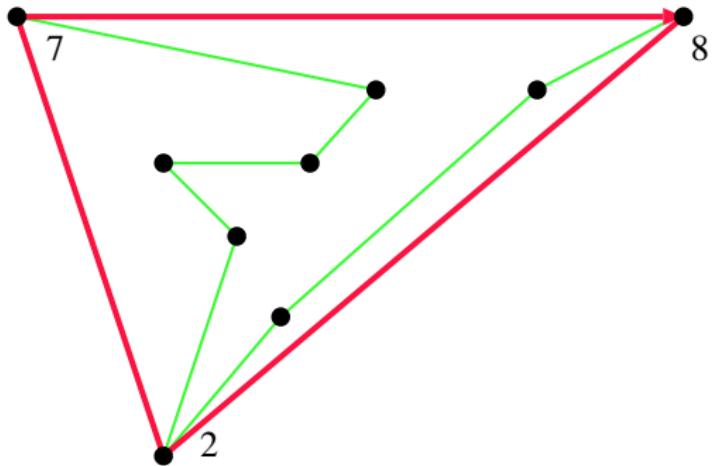
# Animation of Melkman's Algorithm



$b \leftarrow$   $t \rightarrow$

-2	-1	0	1	2	3
	3	1	2	3	
4	3	1	2	4	
	5	1	2	4	5
6	1	2	4	6	
7	6	1	2	7	

# Animation of Melkman's Algorithm



$b$  ← →  $t$

-2	-1	0	1	2	3
3	1	2	3		
4	3	1	2	4	
5	1	2	4	5	
6	1	2	4	6	
7	6	1	2	7	
	8	2	7	8	

# Convex Hull of a Simple Polygon: Analysis of Melkman's Algorithm

## Theorem 33 (Melkman (1987))

Melkman's algorithm computes the convex hull of a simple  $n$ -vertex polygon in  $O(n)$  time.

# Convex Hull of a Simple Polygon: Analysis of Melkman's Algorithm

## Theorem 33 (Melkman (1987))

Melkman's algorithm computes the convex hull of a simple  $n$ -vertex polygon in  $O(n)$  time.

*Proof:* Similar to the analysis of Graham's Scan:

- Each vertex of the polygon is classified as either interior or exterior to the current hull in  $O(1)$  time.
- If vertex  $v_i$  is exterior to the current hull then  $k_i$  other vertices may end up being deleted, with  $O(1)$  time per each vertex that is deleted.
- Since  $\sum_{i=1}^n k_i \leq n - 3$ , the entire algorithm runs in  $O(n)$  time.

□

# Convex Hull of a Simple Polygon: Analysis of Melkman's Algorithm

## Theorem 33 (Melkman (1987))

Melkman's algorithm computes the convex hull of a simple  $n$ -vertex polygon in  $O(n)$  time.

*Proof:* Similar to the analysis of Graham's Scan:

- Each vertex of the polygon is classified as either interior or exterior to the current hull in  $O(1)$  time.
- If vertex  $v_i$  is exterior to the current hull then  $k_i$  other vertices may end up being deleted, with  $O(1)$  time per each vertex that is deleted.
- Since  $\sum_{i=1}^n k_i \leq n - 3$ , the entire algorithm runs in  $O(n)$  time.

□

- The first correct convex-hull algorithm for polygons is due to [McCallum&Avis 1979].

## Convex Hulls

- Basics
- Graham's Scan
- Gift Wrapping
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls

## Convex Hulls in 3D

- Consider a set  $S$  of  $n$  points  $\{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^3$ . We want to compute  $CH(S)$ .



## Convex Hulls in 3D

- Consider a set  $S$  of  $n$  points  $\{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^3$ . We want to compute  $CH(S)$ .
- Easy to prove:  $CH(S)$  is a convex polytope.
- The  $\Omega(n \log n)$  lower bound extends trivially from 2D to 3D.



## Convex Hulls in 3D

- Consider a set  $S$  of  $n$  points  $\{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^3$ . We want to compute  $CH(S)$ .
- Easy to prove:  $CH(S)$  is a convex polytope.
- The  $\Omega(n \log n)$  lower bound extends trivially from 2D to 3D.
- Divide-and-conquer algorithm for computing convex hulls in 3D:
  - Sort (and re-number) the points of  $S$  according to their  $x$ -coordinates.

## Convex Hulls in 3D

- Consider a set  $S$  of  $n$  points  $\{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^3$ . We want to compute  $CH(S)$ .
- Easy to prove:  $CH(S)$  is a convex polytope.
- The  $\Omega(n \log n)$  lower bound extends trivially from 2D to 3D.
- Divide-and-conquer algorithm for computing convex hulls in 3D:
  - Sort (and re-number) the points of  $S$  according to their  $x$ -coordinates.
  - Partition the set  $S$  into two subsets:
$$S_1 := \{p_1, p_2, \dots, p_{\lfloor n/2 \rfloor}\}, \text{ and}$$
$$S_2 := \{p_{\lfloor n/2 \rfloor + 1}, \dots, p_n\}.$$



## Convex Hulls in 3D

- Consider a set  $S$  of  $n$  points  $\{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^3$ . We want to compute  $CH(S)$ .
- Easy to prove:  $CH(S)$  is a convex polytope.
- The  $\Omega(n \log n)$  lower bound extends trivially from 2D to 3D.
- Divide-and-conquer algorithm for computing convex hulls in 3D:
  - Sort (and re-number) the points of  $S$  according to their  $x$ -coordinates.
  - Partition the set  $S$  into two subsets:  
 $S_1 := \{p_1, p_2, \dots, p_{\lfloor n/2 \rfloor}\}$ , and  
 $S_2 := \{p_{\lfloor n/2 \rfloor + 1}, \dots, p_n\}$ .
  - Recursively find the convex hulls  $CH(S_1)$  and  $CH(S_2)$ .



## Convex Hulls in 3D

- Consider a set  $S$  of  $n$  points  $\{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^3$ . We want to compute  $CH(S)$ .
- Easy to prove:  $CH(S)$  is a convex polytope.
- The  $\Omega(n \log n)$  lower bound extends trivially from 2D to 3D.
- Divide-and-conquer algorithm for computing convex hulls in 3D:
  - Sort (and re-number) the points of  $S$  according to their  $x$ -coordinates.
  - Partition the set  $S$  into two subsets:
$$S_1 := \{p_1, p_2, \dots, p_{\lfloor n/2 \rfloor}\}, \text{ and}$$
$$S_2 := \{p_{\lfloor n/2 \rfloor + 1}, \dots, p_n\}.$$
  - Recursively find the convex hulls  $CH(S_1)$  and  $CH(S_2)$ .
  - Merge the two hulls together to form  $CH(S)$ .



## Convex Hulls in 3D: Divide-and-Conquer Algorithm

- The key idea of the merge step is similar to gift wrapping in 2D:
  - ➊ Assume one facet of the convex hull of the two polytopes  $P_1$  and  $P_2$  is given.  
(This relies on the projection of  $P_1$  and  $P_2$  to 2D.)
  - ➋ Find a neighboring facet of the hull by "wrapping" a 2D plane (paper) around the two polytopes.
  - ➌ Continue from each facet to its neighbors until all facets are found.



# Convex Hulls in 3D: Divide-and-Conquer Algorithm

- The key idea of the merge step is similar to gift wrapping in 2D:
  - ➊ Assume one facet of the convex hull of the two polytopes  $P_1$  and  $P_2$  is given. (This relies on the projection of  $P_1$  and  $P_2$  to 2D.)
  - ➋ Find a neighboring facet of the hull by "wrapping" a 2D plane (paper) around the two polytopes.
  - ➌ Continue from each facet to its neighbors until all facets are found.

## Theorem 34

The merge step of the divide&conquer algorithm can be carried out in  $O(n)$  time. Thus, the convex hull of  $n$  points in  $\mathbb{R}^3$  can be computed in  $O(n \log n)$  time.



# Convex Hulls in 3D: Divide-and-Conquer Algorithm

- The key idea of the merge step is similar to gift wrapping in 2D:
  - ➊ Assume one facet of the convex hull of the two polytopes  $P_1$  and  $P_2$  is given. (This relies on the projection of  $P_1$  and  $P_2$  to 2D.)
  - ➋ Find a neighboring facet of the hull by "wrapping" a 2D plane (paper) around the two polytopes.
  - ➌ Continue from each facet to its neighbors until all facets are found.

## Theorem 34

The merge step of the divide&conquer algorithm can be carried out in  $O(n)$  time. Thus, the convex hull of  $n$  points in  $\mathbb{R}^3$  can be computed in  $O(n \log n)$  time.

## Lemma 35 (Seidel (1984))

The computation of the convex hull of a star-shaped polytope in  $\mathbb{R}^3$  with  $n$  vertices requires  $\Omega(n \log n)$  time in the worst case.



## Convex Hulls in 3D: Divide-and-Conquer Algorithm

- The key idea of the merge step is similar to gift wrapping in 2D:
  - ➊ Assume one facet of the convex hull of the two polytopes  $P_1$  and  $P_2$  is given. (This relies on the projection of  $P_1$  and  $P_2$  to 2D.)
  - ➋ Find a neighboring facet of the hull by "wrapping" a 2D plane (paper) around the two polytopes.
  - ➌ Continue from each facet to its neighbors until all facets are found.

### Theorem 34

The merge step of the divide&conquer algorithm can be carried out in  $O(n)$  time. Thus, the convex hull of  $n$  points in  $\mathbb{R}^3$  can be computed in  $O(n \log n)$  time.

### Lemma 35 (Seidel (1984))

The computation of the convex hull of a star-shaped polytope in  $\mathbb{R}^3$  with  $n$  vertices requires  $\Omega(n \log n)$  time in the worst case.

### Lemma 36

The convex hull of  $n$  points in  $\mathbb{R}^d$  can have  $\Omega(n^{\lfloor d/2 \rfloor})$  facets.



## Convex Hulls

- Basics
- Graham's Scan
- Gift Wrapping
- Divide-and-Conquer Algorithm
- Online Convex Hulls
- Convex Hull of Polygons
- Convex Hulls in 3D
- Sample Applications of Convex Hulls

## Sample Applications of Convex Hulls: Onion Layers

- Consider a set  $S$  of  $n$  points in  $\mathbb{R}^2$ .
- Let  $S_0 \subseteq S$  be the set of all vertices of  $CH(S)$ . (General position assumed.)
- The points of  $S_0$  are said to have *depth* 0.



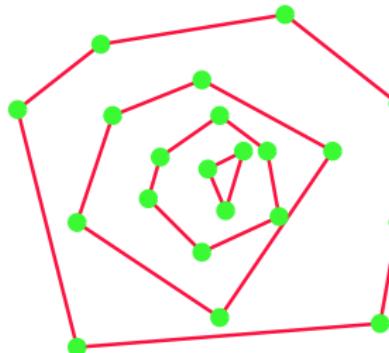
## Sample Applications of Convex Hulls: Onion Layers

- Consider a set  $S$  of  $n$  points in  $\mathbb{R}^2$ .
- Let  $S_0 \subseteq S$  be the set of all vertices of  $CH(S)$ . (General position assumed.)
- The points of  $S_0$  are said to have *depth 0*.
- Now let  $S := S \setminus S_0$ , and re-consider  $CH(S)$ .
- All points of  $S$  that are on  $CH(S)$  are said to have *depth 1*, and are assigned to  $S_1$ .



## Sample Applications of Convex Hulls: Onion Layers

- Consider a set  $S$  of  $n$  points in  $\mathbb{R}^2$ .
- Let  $S_0 \subseteq S$  be the set of all vertices of  $CH(S)$ . (General position assumed.)
- The points of  $S_0$  are said to have *depth 0*.
- Now let  $S := S \setminus S_0$ , and re-consider  $CH(S)$ .
- All points of  $S$  that are on  $CH(S)$  are said to have *depth 1*, and are assigned to  $S_1$ .
- Similarly for depths  $2, 3, \dots, k$ , where  $S_k \neq \emptyset$  and  $S_{k+1} = \emptyset$ .
- The sets  $S_0, S_1, S_2, \dots$  are called *shells* or *onion layers* of  $S$ .



# Sample Applications of Convex Hulls: Onion Layers

## Lemma 37

It takes  $\Omega(n \log n)$  time to compute all depths of  $n$  points in  $\mathbb{R}^2$ .

# Sample Applications of Convex Hulls: Onion Layers

## Lemma 37

It takes  $\Omega(n \log n)$  time to compute all depths of  $n$  points in  $\mathbb{R}^2$ .

## Lemma 38

All depths of  $n$  points in  $\mathbb{R}^2$ , together with their onion layers, can be computed in time  $O(n \log n)$ .

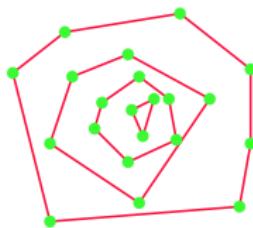
# Sample Applications of Convex Hulls: Onion Layers

## Lemma 37

It takes  $\Omega(n \log n)$  time to compute all depths of  $n$  points in  $\mathbb{R}^2$ .

## Lemma 38

All depths of  $n$  points in  $\mathbb{R}^2$ , together with their onion layers, can be computed in time  $O(n \log n)$ .



- Statistics: The points of  $S_k, S_{k-1}, S_{k-2}, \dots$  lie close to the “center” of  $S$ , and computing their mean tends to discard “outliers”, thus yielding a more robust statistical estimator of the mean of  $S$  than the mean of all point samples.



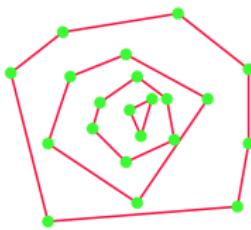
# Sample Applications of Convex Hulls: Onion Layers

## Lemma 37

It takes  $\Omega(n \log n)$  time to compute all depths of  $n$  points in  $\mathbb{R}^2$ .

## Lemma 38

All depths of  $n$  points in  $\mathbb{R}^2$ , together with their onion layers, can be computed in time  $O(n \log n)$ .



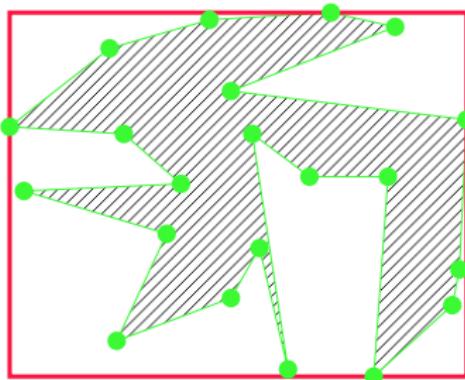
- Statistics: The points of  $S_k, S_{k-1}, S_{k-2}, \dots$  lie close to the “center” of  $S$ , and computing their mean tends to discard “outliers”, thus yielding a more robust statistical estimator of the mean of  $S$  than the mean of all point samples.
- Rendering: Onion layers can be used to generate Hamiltonian triangulations.



## Sample Applications of Convex Hulls: Kinetic AABB

### Definition 39 (AABB)

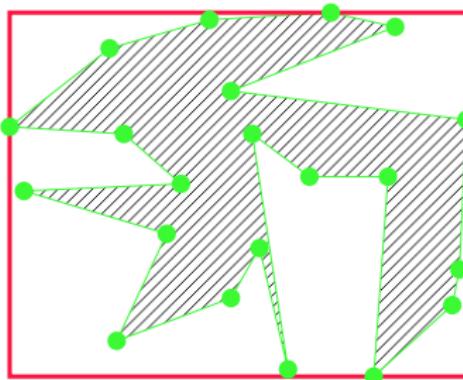
The (*axis-aligned*) *bounding box* (AABB) of a set  $S \subset \mathbb{R}^d$ , denoted by  $\text{AABB}(S)$ , is the smallest box (with sides parallel to the coordinate planes) which contains  $S$ .



## Sample Applications of Convex Hulls: Kinetic AABB

### Definition 39 (AABB)

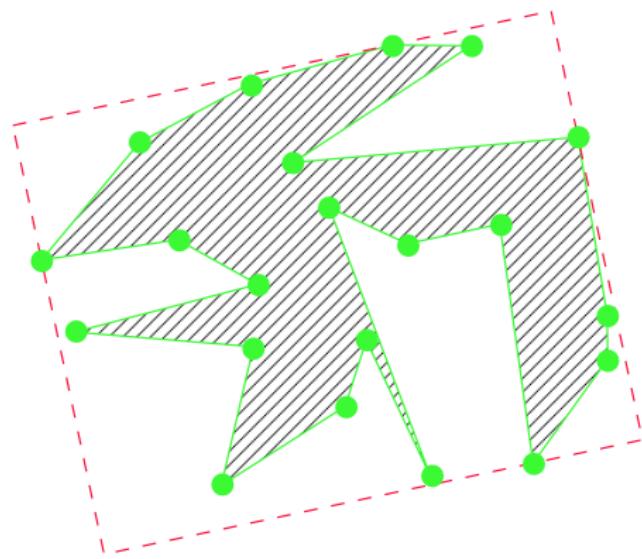
The (*axis-aligned*) *bounding box* (AABB) of a set  $S \subset \mathbb{R}^d$ , denoted by  $\text{AABB}(S)$ , is the smallest box (with sides parallel to the coordinate planes) which contains  $S$ .



- If  $S$  can be described by a set of  $n$  vertices then  $\text{AABB}(S)$  can be computed in  $O(d \cdot n)$  time in a straightforward manner.

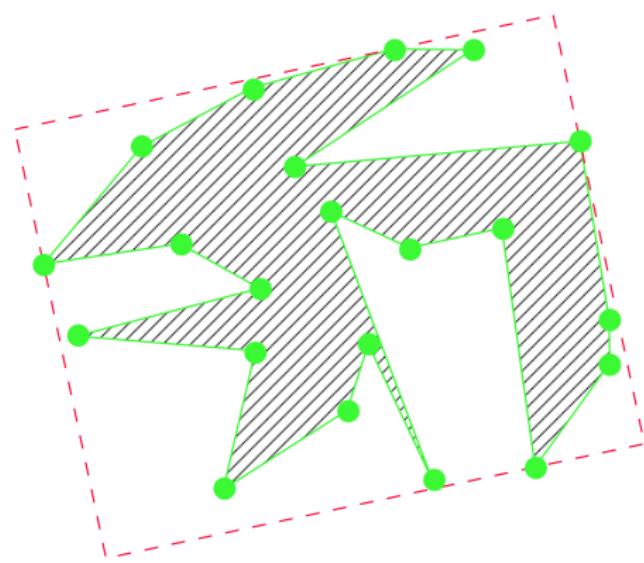
## Sample Applications of Convex Hulls: Kinetic AABB

- What happens if  $S$  moves? We observe that  $AABB(S)$  equals  $AABB(CH(S))$ : up to six vertices  $v_1, v_2, \dots, v_6$  of  $CH(S)$  determine  $AABB(S)$  in  $\mathbb{R}^3$ .



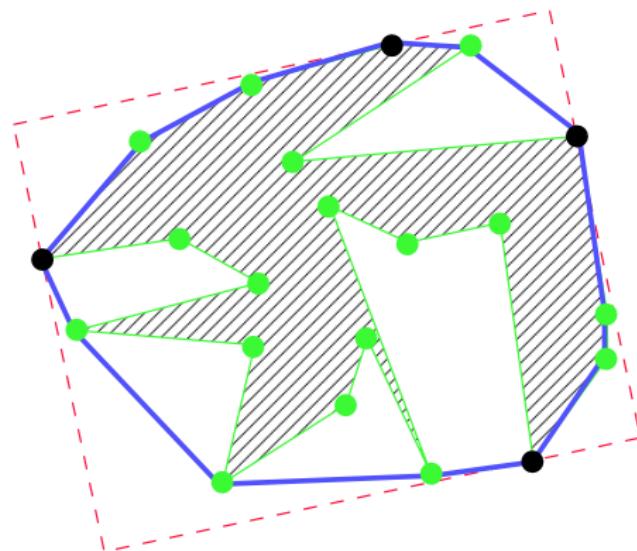
## Sample Applications of Convex Hulls: Kinetic AABB

- What happens if  $S$  moves? We observe that  $AABB(S)$  equals  $AABB(CH(S))$ : up to six vertices  $v_1, v_2, \dots, v_6$  of  $CH(S)$  determine  $AABB(S)$  in  $\mathbb{R}^3$ .
- Goal: Avoid re-scanning all vertices of  $S$  in order to re-compute the axis-aligned bounding box from scratch.



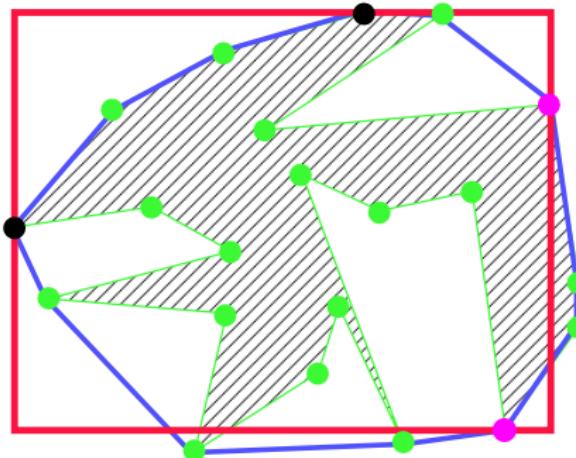
## Sample Applications of Convex Hulls: Kinetic AABB

- What happens if  $S$  moves? We observe that  $AABB(S)$  equals  $AABB(CH(S))$ : up to six vertices  $v_1, v_2, \dots, v_6$  of  $CH(S)$  determine  $AABB(S)$  in  $\mathbb{R}^3$ .
- We can exploit coherence by applying a hill-climbing algorithm, starting at each of these six vertices (resp. four vertices in 2D).



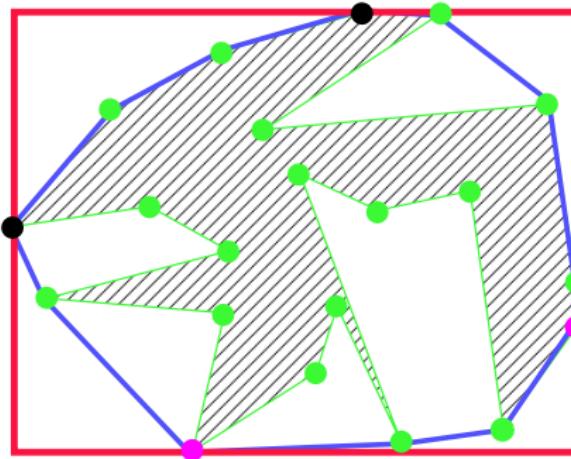
## Sample Applications of Convex Hulls: Kinetic AABB

- What happens if  $S$  moves? We observe that  $AABB(S)$  equals  $AABB(CH(S))$ : up to six vertices  $v_1, v_2, \dots, v_6$  of  $CH(S)$  determine  $AABB(S)$  in  $\mathbb{R}^3$ .
- Hill-climbing means to move from one vertex to a neighboring vertex of  $CH(S)$  if it has a smaller/larger  $x$ -coordinate,  $y$ -coordinate, ...



## Sample Applications of Convex Hulls: Kinetic AABB

- What happens if  $S$  moves? We observe that  $AABB(S)$  equals  $AABB(CH(S))$ : up to six vertices  $v_1, v_2, \dots, v_6$  of  $CH(S)$  determine  $AABB(S)$  in  $\mathbb{R}^3$ .
- If  $S$  has moved only a little then few steps of the hill-climbing algorithm will suffice. Of course, this scheme can be extended to  $k$ -dops.



## Voronoi Diagrams of Points

- Definition and Properties
  - Proximity Problems and Lower Bounds
  - Definitions
  - Properties
  - Delaunay Triangulation
- Algorithms
  - Divide&Conquer Algorithm
  - Incremental Construction
  - Sweep-Line Algorithm
  - Construction via Lifting to 3D
  - Approximate Voronoi Diagram by Means of Graphics Hardware

## Voronoi Diagrams of Points

- Definition and Properties
  - Proximity Problems and Lower Bounds
  - Definitions
  - Properties
  - Delaunay Triangulation
- Algorithms

## A Set of Proximity Problems

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points in  $\mathbb{R}^2$  under the Euclidean metric.  
(Unless mentioned explicitly, we will always deal with the Euclidean metric.)



## A Set of Proximity Problems

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points in  $\mathbb{R}^2$  under the Euclidean metric.  
(Unless mentioned explicitly, we will always deal with the Euclidean metric.)

**CLOSESTPAIR:** Determine two points of  $S$  whose mutual distance is smallest.

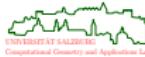


## A Set of Proximity Problems

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points in  $\mathbb{R}^2$  under the Euclidean metric.  
(Unless mentioned explicitly, we will always deal with the Euclidean metric.)

**CLOSESTPAIR:** Determine two points of  $S$  whose mutual distance is smallest.

**ALLNEARESTNEIGHBORS:** Determine the “nearest neighbor” (point of minimum distance within  $S$ ) for each point in  $S$ .



## A Set of Proximity Problems

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points in  $\mathbb{R}^2$  under the Euclidean metric.  
(Unless mentioned explicitly, we will always deal with the Euclidean metric.)

**CLOSESTPAIR:** Determine two points of  $S$  whose mutual distance is smallest.

**ALLNEARESTNEIGHBORS:** Determine the “nearest neighbor” (point of minimum distance within  $S$ ) for each point in  $S$ .

**EUCLIDEANMINIMUMSPANNINGTREE (EMST):** Construct a tree of minimum total (Euclidean) length whose vertices are the points of  $S$ . (No Steiner points allowed.)



## A Set of Proximity Problems

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points in  $\mathbb{R}^2$  under the Euclidean metric.  
(Unless mentioned explicitly, we will always deal with the Euclidean metric.)

**CLOSESTPAIR:** Determine two points of  $S$  whose mutual distance is smallest.

**ALLNEARESTNEIGHBORS:** Determine the “nearest neighbor” (point of minimum distance within  $S$ ) for each point in  $S$ .

**EUCLIDEANMINIMUMSPANNINGTREE (EMST):** Construct a tree of minimum total (Euclidean) length whose vertices are the points of  $S$ . (No Steiner points allowed.)

**MAXIMUMEMPTYCIRCLE:** Find a circle with largest radius which does not contain a point of  $S$  in its interior and whose center lies within  $CH(S)$ .



## A Set of Proximity Problems

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points in  $\mathbb{R}^2$  under the Euclidean metric.  
(Unless mentioned explicitly, we will always deal with the Euclidean metric.)

**CLOSESTPAIR:** Determine two points of  $S$  whose mutual distance is smallest.

**ALLNEARESTNEIGHBORS:** Determine the “nearest neighbor” (point of minimum distance within  $S$ ) for each point in  $S$ .

**EUCLIDEANMINIMUMSPANNINGTREE (EMST):** Construct a tree of minimum total (Euclidean) length whose vertices are the points of  $S$ . (No Steiner points allowed.)

**MAXIMUMEMPTYCIRCLE:** Find a circle with largest radius which does not contain a point of  $S$  in its interior and whose center lies within  $CH(S)$ .

**TRIANGULATION:** Join the points in  $S$  by non-intersecting straight-line segments so that every region internal to the convex hull of  $S$  is a triangle.



# A Set of Proximity Problems

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points in  $\mathbb{R}^2$  under the Euclidean metric.  
(Unless mentioned explicitly, we will always deal with the Euclidean metric.)

**CLOSESTPAIR:** Determine two points of  $S$  whose mutual distance is smallest.

**ALLNEARESTNEIGHBORS:** Determine the “nearest neighbor” (point of minimum distance within  $S$ ) for each point in  $S$ .

**EUCLIDEANMINIMUMSPANNINGTREE (EMST):** Construct a tree of minimum total (Euclidean) length whose vertices are the points of  $S$ . (No Steiner points allowed.)

**MAXIMUMEMPTYCIRCLE:** Find a circle with largest radius which does not contain a point of  $S$  in its interior and whose center lies within  $CH(S)$ .

**TRIANGULATION:** Join the points in  $S$  by non-intersecting straight-line segments so that every region internal to the convex hull of  $S$  is a triangle.

**NEARESTNEIGHBORSEARCH:** Given a query point  $q$ , determine which point  $p \in S$  is closest to  $q$ .



# Lower Bounds

## Lemma 40

NEARESTNEIGHBORSEARCH among  $n$  points in  $\mathbb{R}^2$  has an  $\Omega(\log n)$  lower bound;  
CLOSESTPAIR, ALLNEARESTNEIGHBORS, EMST, MAXIMUMEMPTYCIRCLE and  
TRIANGULATION all have  $\Omega(n \log n)$  lower bounds (in the ADT model of computation).



# Lower Bounds

## Lemma 40

NEARESTNEIGHBORSEARCH among  $n$  points in  $\mathbb{R}^2$  has an  $\Omega(\log n)$  lower bound;  
CLOSESTPAIR, ALLNEARESTNEIGHBORS, EMST, MAXIMUMEMPTYCIRCLE and  
TRIANGULATION all have  $\Omega(n \log n)$  lower bounds (in the ADT model of computation).

*Proof:*

- NEARESTNEIGHBORSEARCH: standard argument yields  $\Omega(\log n)$  comparisons.



# Lower Bounds

## Lemma 40

NEARESTNEIGHBORSEARCH among  $n$  points in  $\mathbb{R}^2$  has an  $\Omega(\log n)$  lower bound;  
CLOSESTPAIR, ALLNEARESTNEIGHBORS, EMST, MAXIMUMEMPTYCIRCLE and  
TRIANGULATION all have  $\Omega(n \log n)$  lower bounds (in the ADT model of computation).

### *Proof:*

- NEARESTNEIGHBORSEARCH: standard argument yields  $\Omega(\log n)$  comparisons.
- CLOSESTPAIR: ELEMENTUNIQUENESS, which requires  $\Omega(n \log n)$  time in the  
ADT model of computation, is linearly transformable to CLOSESTPAIR.



# Lower Bounds

## Lemma 40

NEARESTNEIGHBORSEARCH among  $n$  points in  $\mathbb{R}^2$  has an  $\Omega(\log n)$  lower bound;  
CLOSESTPAIR, ALLNEARESTNEIGHBORS, EMST, MAXIMUMEMPTYCIRCLE and  
TRIANGULATION all have  $\Omega(n \log n)$  lower bounds (in the ADT model of computation).

*Proof:*

- NEARESTNEIGHBORSEARCH: standard argument yields  $\Omega(\log n)$  comparisons.
- CLOSESTPAIR: ELEMENTUNIQUENESS, which requires  $\Omega(n \log n)$  time in the ADT model of computation, is linearly transformable to CLOSESTPAIR.
- ALLNEARESTNEIGHBORS: CLOSESTPAIR is linearly transformable to ALLNEARESTNEIGHBORS.



## Lemma 40

NEARESTNEIGHBORSEARCH among  $n$  points in  $\mathbb{R}^2$  has an  $\Omega(\log n)$  lower bound;  
CLOSESTPAIR, ALLNEARESTNEIGHBORS, EMST, MAXIMUMEMPTYCIRCLE and  
TRIANGULATION all have  $\Omega(n \log n)$  lower bounds (in the ADT model of computation).

### Proof:

- NEARESTNEIGHBORSEARCH: standard argument yields  $\Omega(\log n)$  comparisons.
- CLOSESTPAIR: ELEMENTUNIQUENESS, which requires  $\Omega(n \log n)$  time in the ADT model of computation, is linearly transformable to CLOSESTPAIR.
- ALLNEARESTNEIGHBORS: CLOSESTPAIR is linearly transformable to ALLNEARESTNEIGHBORS.
- EMST: CLOSESTPAIR is linearly transformable to EMST. (Also, SORTING can be transformed linearly to EMST.)



# Lower Bounds

## Lemma 40

NEARESTNEIGHBORSEARCH among  $n$  points in  $\mathbb{R}^2$  has an  $\Omega(\log n)$  lower bound;  
CLOSESTPAIR, ALLNEARESTNEIGHBORS, EMST, MAXIMUMEMPTYCIRCLE and  
TRIANGULATION all have  $\Omega(n \log n)$  lower bounds (in the ADT model of computation).

### Proof:

- NEARESTNEIGHBORSEARCH: standard argument yields  $\Omega(\log n)$  comparisons.
- CLOSESTPAIR: ELEMENTUNIQUENESS, which requires  $\Omega(n \log n)$  time in the ADT model of computation, is linearly transformable to CLOSESTPAIR.
- ALLNEARESTNEIGHBORS: CLOSESTPAIR is linearly transformable to ALLNEARESTNEIGHBORS.
- EMST: CLOSESTPAIR is linearly transformable to EMST. (Also, SORTING can be transformed linearly to EMST.)
- MAXIMUMEMPTYCIRCLE in 1D solves MAXGAP, which establishes the  $\Omega(n \log n)$  lower bound.



# Lower Bounds

## Lemma 40

NEARESTNEIGHBORSEARCH among  $n$  points in  $\mathbb{R}^2$  has an  $\Omega(\log n)$  lower bound;  
CLOSESTPAIR, ALLNEARESTNEIGHBORS, EMST, MAXIMUMEMPTYCIRCLE and  
TRIANGULATION all have  $\Omega(n \log n)$  lower bounds (in the ADT model of computation).

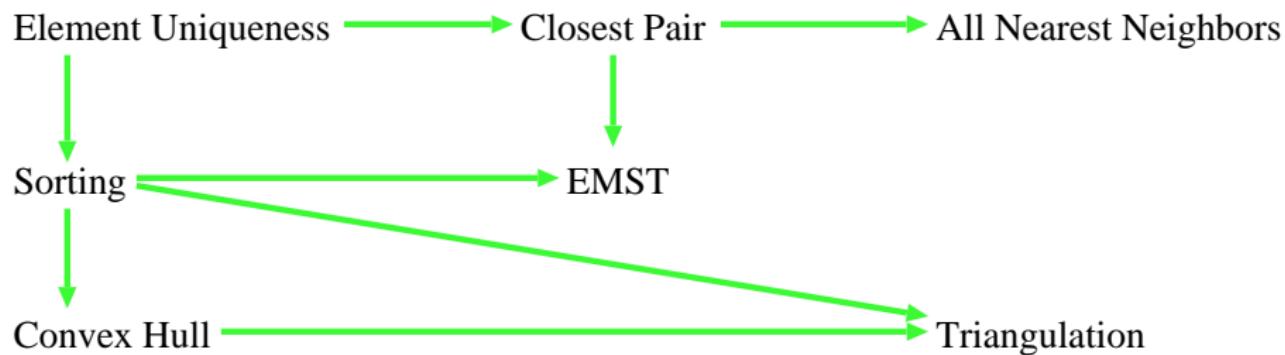
### Proof:

- NEARESTNEIGHBORSEARCH: standard argument yields  $\Omega(\log n)$  comparisons.
- CLOSESTPAIR: ELEMENTUNIQUENESS, which requires  $\Omega(n \log n)$  time in the ADT model of computation, is linearly transformable to CLOSESTPAIR.
- ALLNEARESTNEIGHBORS: CLOSESTPAIR is linearly transformable to ALLNEARESTNEIGHBORS.
- EMST: CLOSESTPAIR is linearly transformable to EMST. (Also, SORTING can be transformed linearly to EMST.)
- MAXIMUMEMPTYCIRCLE in 1D solves MAXGAP, which establishes the  $\Omega(n \log n)$  lower bound.
- TRIANGULATION: CONVEXHULL is linearly transformable to TRIANGULATION.



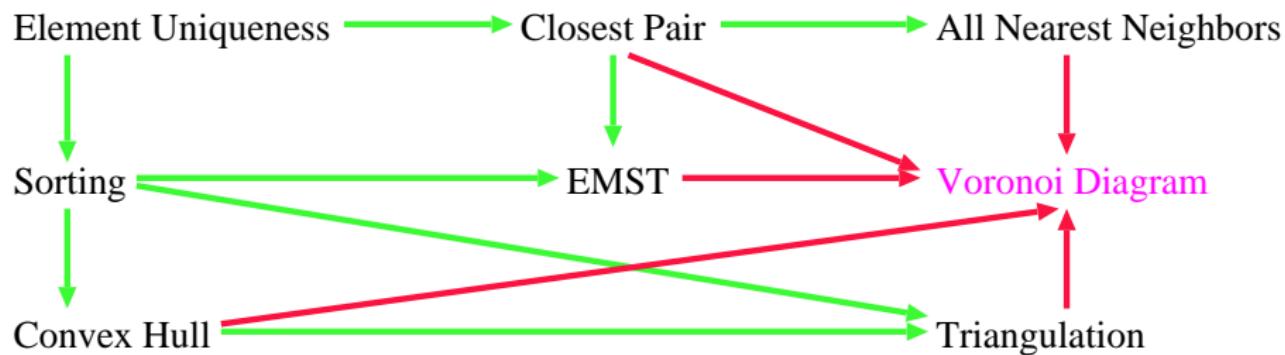
## Lower Bounds: Summary of Reductions

- Thus, we have  $\Omega(n \log n)$  lower bounds due to a variety of reductions.



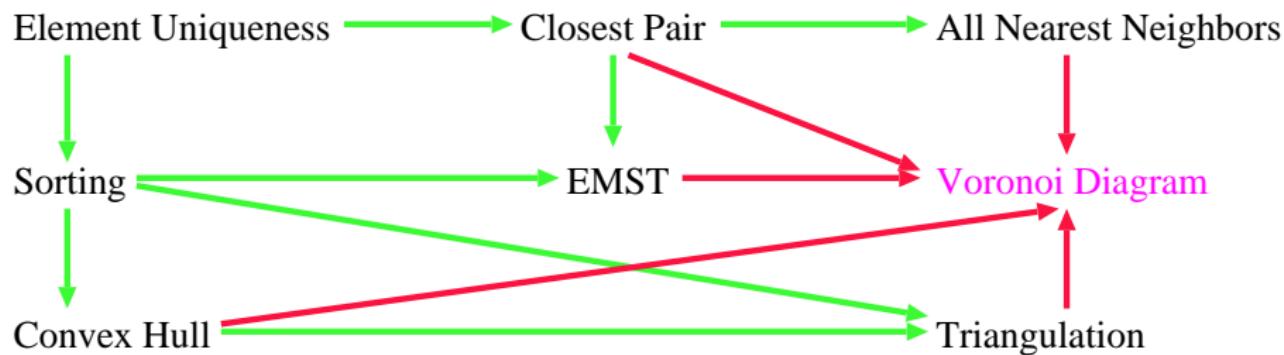
## Lower Bounds: Summary of Reductions

- If Voronoi diagram is available then these problems can be solved in  $O(n)$  time!



## Lower Bounds: Summary of Reductions

- If Voronoi diagram is available then these problems can be solved in  $O(n)$  time!



### Theorem 41

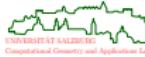
The computation of the Voronoi diagram of  $n$  points in  $\mathbb{R}^2$  requires  $\Omega(n \log n)$  time.



# Voronoi Diagram: Motivation

## Voronoi experiment

Let's throw a ball into "ideal" water (with no boundaries and no initial waves), and watch the wave patterns that emerge (within a finite portion of the water).



## Voronoi experiment

Let's now throw two identical balls at the same time into the water: As the waves meet, the **bisector** between the two balls is traced out.



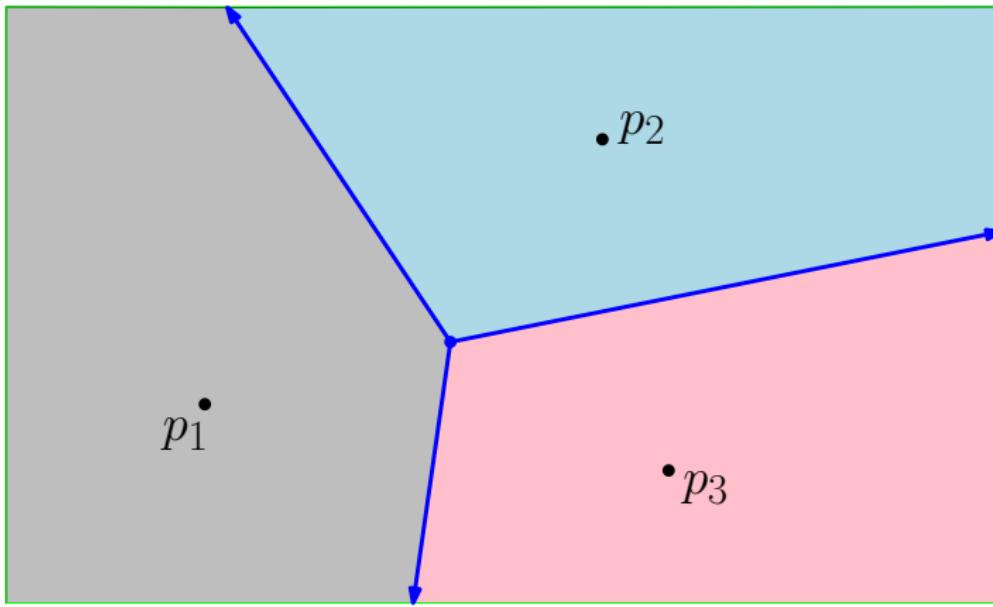
## Voronoi experiment

We repeat the experiment with three identical balls thrown at the same time into the water: Again, the waves trace out the bisectors between the balls as they meet.

# Voronoi Diagram: Motivation

## Voronoi Cells

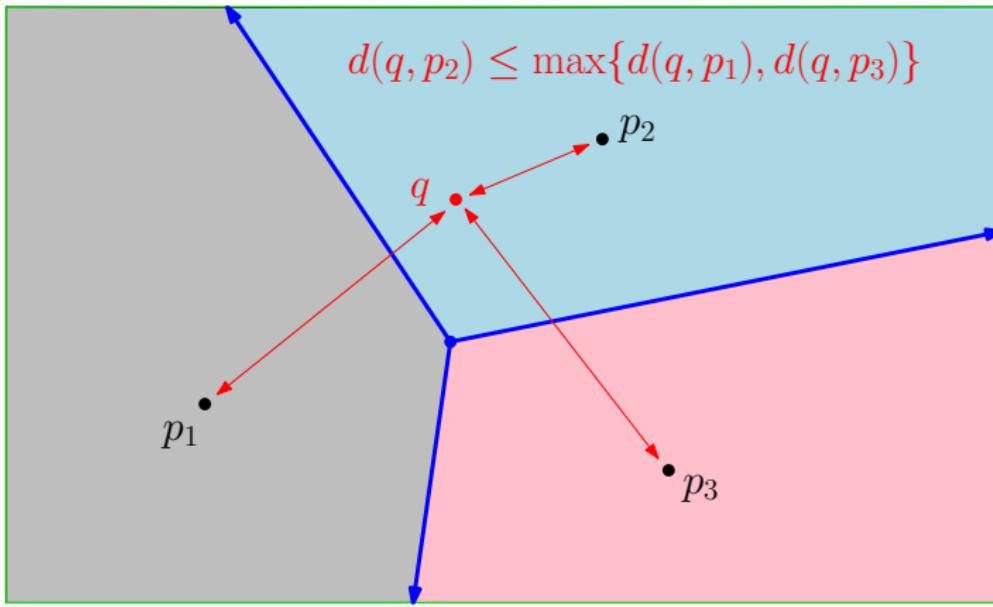
The blue bisectors defined by the three balls partition the water into three Voronoi cells:



# Voronoi Diagram: Motivation

## Voronoi Cells

The blue bisectors defined by the three balls partition the water into three Voronoi cells: Each cell is the loci of points  $q$  closer to its defining ball than to any other ball.



## Voronoi Diagram: Definition

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  distinct points in  $\mathbb{R}^2$  and denote the Euclidean distance by  $d(\cdot, \cdot)$ , with  $d(q, S) := \min\{d(q, p) : p \in S\}$ .



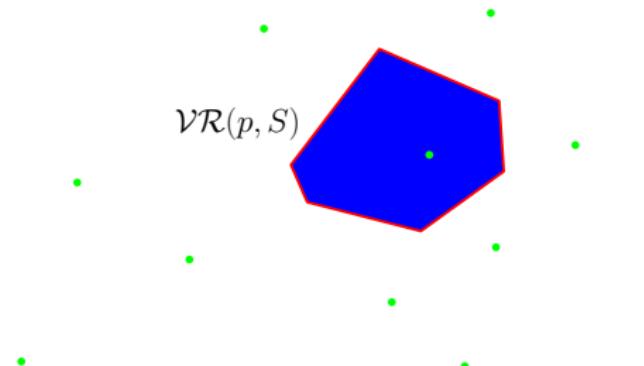
# Voronoi Diagram: Definition

- Consider a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  distinct points in  $\mathbb{R}^2$  and denote the Euclidean distance by  $d(\cdot, \cdot)$ , with  $d(q, S) := \min\{d(q, p) : p \in S\}$ .

## Definition 42 (Voronoi region, Dt.: Voronoi-Zelle)

The *Voronoi region* (VR, aka “Voronoi cell”) of a point  $p \in S$  is the locus of points of  $\mathbb{R}^2$  whose distance to  $p$  is not greater than the distance to any other point of  $S$ :

$$\mathcal{VR}(p, S) := \{q \in \mathbb{R}^2 : d(q, p) \leq d(q, S)\}.$$



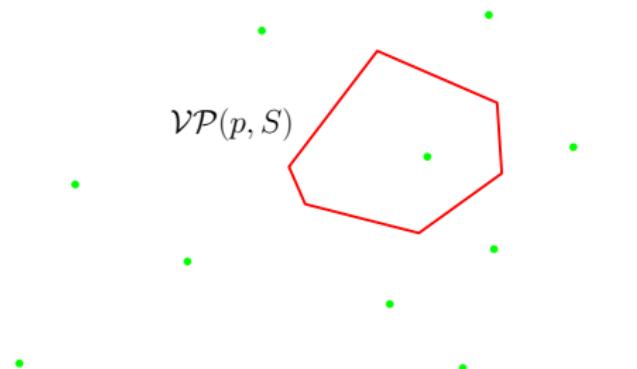
# Voronoi Diagram: Definition

## Definition 43 (Voronoi polygon)

The *Voronoi polygon* (VP) of  $p \in S$  is defined as

$$\mathcal{VP}(p, S) := \partial \mathcal{VR}(p, S).$$

The segments of a Voronoi polygon are called *Voronoi edges*.

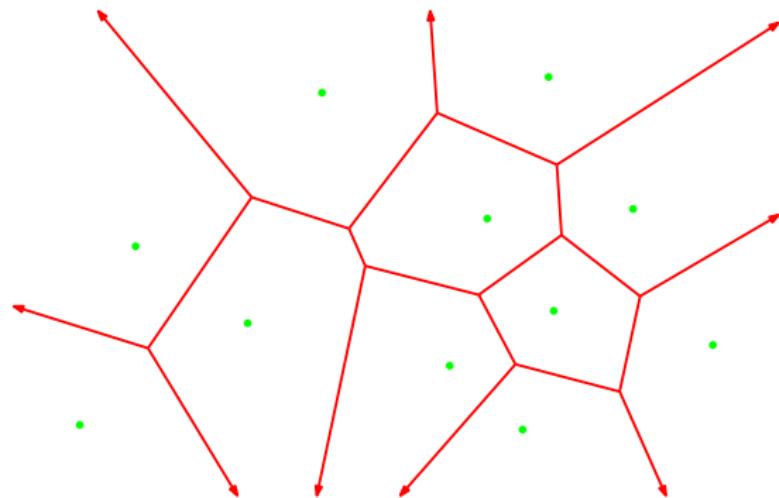


# Voronoi Diagram: Definition

## Definition 44 (Voronoi diagram)

The *Voronoi diagram* (VD) of  $S$  is defined as

$$\mathcal{VD}(S) := \bigcup_{1 \leq i \leq n} \mathcal{VP}(p_i, S).$$

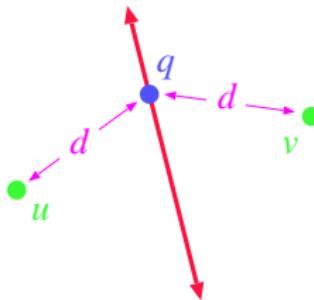


# Voronoi Diagram: Definition

## Definition 45 (Bisector)

The *bisector* of two points  $u, v \in \mathbb{R}^2$  is the set of points of  $\mathbb{R}^2$  which are equidistant to  $u$  and  $v$ :

$$b(u, v) := \{q \in \mathbb{R}^2 : d(u, q) = d(v, q)\}.$$

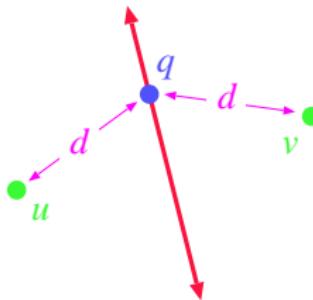


# Voronoi Diagram: Definition

## Definition 45 (Bisector)

The *bisector* of two points  $u, v \in \mathbb{R}^2$  is the set of points of  $\mathbb{R}^2$  which are equidistant to  $u$  and  $v$ :

$$b(u, v) := \{q \in \mathbb{R}^2 : d(u, q) = d(v, q)\}.$$



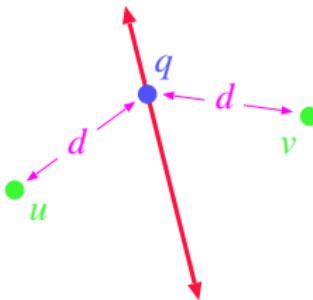
- A Voronoi edge always lies on a bisector. Thus, points on a Voronoi edge are equidistant to two points of  $S$ .

# Voronoi Diagram: Definition

## Definition 45 (Bisector)

The *bisector* of two points  $u, v \in \mathbb{R}^2$  is the set of points of  $\mathbb{R}^2$  which are equidistant to  $u$  and  $v$ :

$$b(u, v) := \{q \in \mathbb{R}^2 : d(u, q) = d(v, q)\}.$$



- A Voronoi edge always lies on a bisector. Thus, points on a Voronoi edge are equidistant to two points of  $S$ .

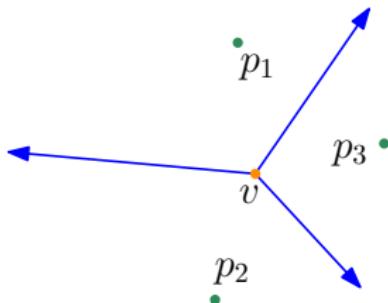
## Lemma 46

For  $p \in S$  we get  $\mathcal{VP}(p, S) = \{q \in \mathbb{R}^2 : d(q, p) = d(q, S \setminus \{p\})\}$ .

# Voronoi Diagram: Definition

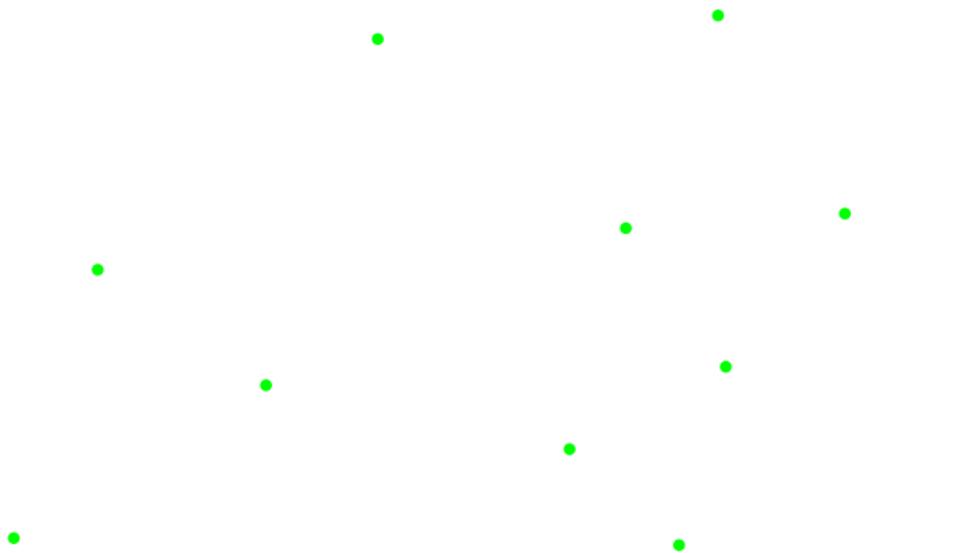
## Definition 47 (Voronoi node, Dt.: Voronoi-Knoten)

Intersections of Voronoi edges are called *Voronoi nodes*.



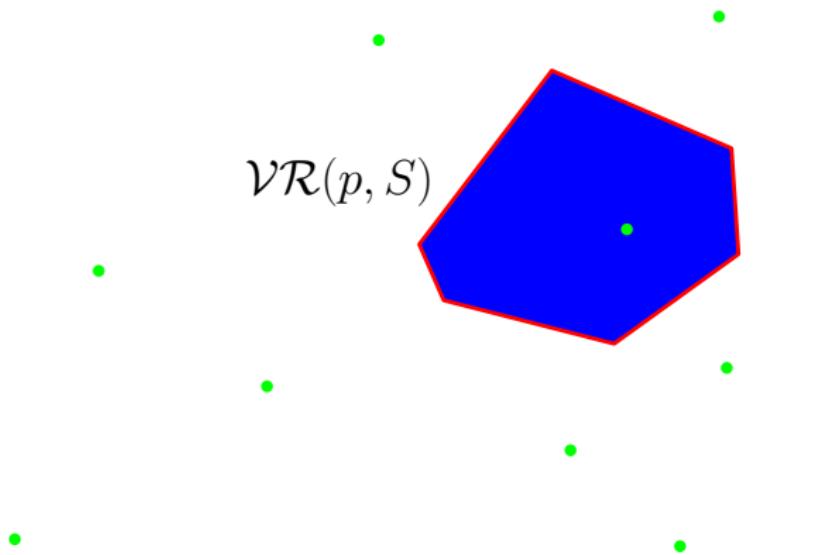
# Sample Voronoi Diagram

- Input  $S$ .



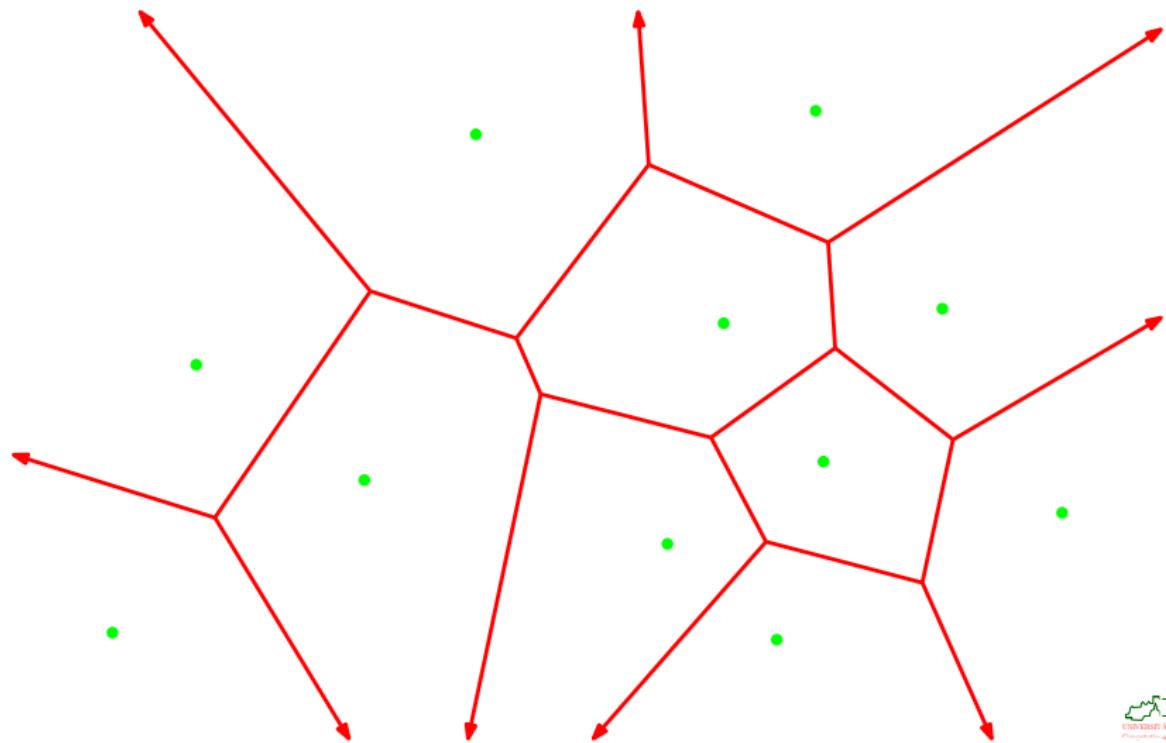
# Sample Voronoi Diagram

- Input  $S$ , Voronoi region.



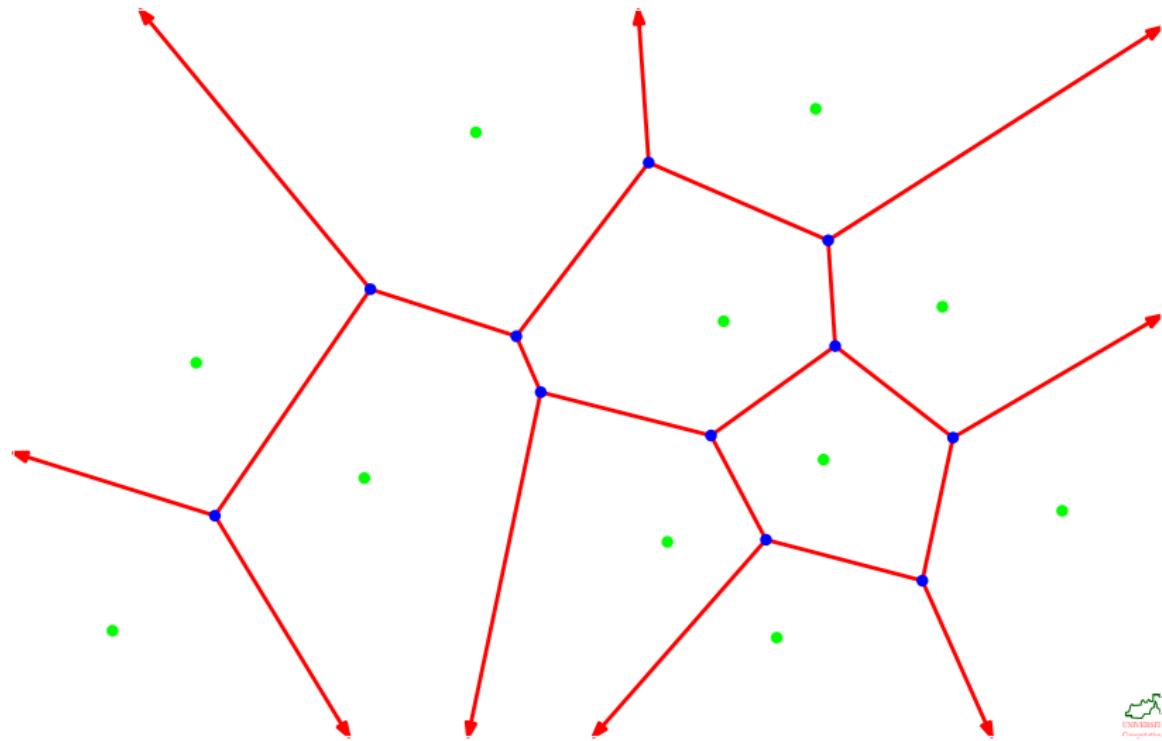
## Sample Voronoi Diagram

- Input  $S$ , Voronoi region, Voronoi diagram.



## Sample Voronoi Diagram

- Input  $S$ , Voronoi region, Voronoi diagram, Voronoi nodes.



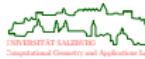
## Historical Remarks

- René Descartes (1596–1650): He drew Voronoi-like diagrams to illustrate the subdivision of space by celestial bodies [Descartes 1644].
- Gustav Lejeune Dirichlet (1805–1859) provided the first formal definition of Voronoi diagrams in two dimensions [Dirichlet 1850].



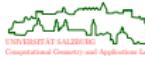
## Historical Remarks

- René Descartes (1596–1650): He drew Voronoi-like diagrams to illustrate the subdivision of space by celestial bodies [Descartes 1644].
- Gustav Lejeune Dirichlet (1805–1859) provided the first formal definition of Voronoi diagrams in two dimensions [Dirichlet 1850].
- Georgy Feodosevich Voronoi (1868–1908) generalized them to  $n$  dimensions [Voronoi 1908].
  - Several other Latin spellings of his name: Voronoï, Voronoy, Woronoi.
  - Born at Zhuravky (near Kiev).
  - Studied at Saint Petersburg University as a student of Andrey Markov.
  - Professor at the University of Warsaw.
  - Students (among others): Boris Delaunay (Kiev) and Waclaw Sierpiński (Warsaw).



## General Position Assumed

- General position assumed (GPA): No four points of  $S$  are co-circular!



## General Position Assumed

- General position assumed (GPA): No four points of  $S$  are co-circular!
- Note that several (standard) Voronoi-related claims need to be weakened or become more tedious to state if general position may not be assumed.



## General Position Assumed

- General position assumed (GPA): No four points of  $S$  are co-circular!
- Note that several (standard) Voronoi-related claims need to be weakened or become more tedious to state if general position may not be assumed.
- GPA facilitates the description of algorithms and is widely (implicitly) assumed in computational geometry:
  - “No three points are collinear”,
  - “No three lines intersect in the same point”,
  - “No two points have the same  $y$ -coordinate”,
  - ...

## General Position Assumed

- General position assumed (GPA): No four points of  $S$  are co-circular!
- Note that several (standard) Voronoi-related claims need to be weakened or become more tedious to state if general position may not be assumed.
- GPA facilitates the description of algorithms and is widely (implicitly) assumed in computational geometry:
  - “No three points are collinear”,
  - “No three lines intersect in the same point”,
  - “No two points have the same  $y$ -coordinate”,
  - ...

### Warning

A general-position assumption will not hold for most real-world data. Thus, make sure to check whether an algorithm is indeed applicable to your problem if its description makes use of GPA — or be prepared to fill in the (potentially non-trivial) gaps!

# Voronoi Diagram: Properties

## Lemma 48

The Voronoi region  $\mathcal{VR}(p_i, S)$  is the intersection of half-planes defined by bisectors between  $p_i \in S$  and the other points of  $S$ :

$$\mathcal{VR}(p_i) = \bigcap_{\substack{1 \leq j \leq n \\ j \neq i}} H(p_i, p_j),$$

where  $H(p_i, p_j)$  is the half-space that contains  $p_i$ .

# Voronoi Diagram: Properties

## Lemma 48

The Voronoi region  $\mathcal{VR}(p_i, S)$  is the intersection of half-planes defined by bisectors between  $p_i \in S$  and the other points of  $S$ :

$$\mathcal{VR}(p_i) = \bigcap_{\substack{1 \leq j \leq n \\ j \neq i}} H(p_i, p_j),$$

where  $H(p_i, p_j)$  is the half-space that contains  $p_i$ .

## Corollary 49

Every Voronoi region is a convex polygonal area.



# Voronoi Diagram: Properties

## Lemma 48

The Voronoi region  $\mathcal{VR}(p_i, S)$  is the intersection of half-planes defined by bisectors between  $p_i \in S$  and the other points of  $S$ :

$$\mathcal{VR}(p_i) = \bigcap_{\substack{1 \leq j \leq n \\ j \neq i}} H(p_i, p_j),$$

where  $H(p_i, p_j)$  is the half-space that contains  $p_i$ .

## Corollary 49

Every Voronoi region is a convex polygonal area.

## Lemma 50

Every point of  $S$  has its own Voronoi region that is not empty.

## Voronoi Diagram: Properties

### Lemma 48

The Voronoi region  $\mathcal{VR}(p_i, S)$  is the intersection of half-planes defined by bisectors between  $p_i \in S$  and the other points of  $S$ :

$$\mathcal{VR}(p_i) = \bigcap_{\substack{1 \leq j \leq n \\ j \neq i}} H(p_i, p_j),$$

where  $H(p_i, p_j)$  is the half-space that contains  $p_i$ .

### Corollary 49

Every Voronoi region is a convex polygonal area.

### Lemma 50

Every point of  $S$  has its own Voronoi region that is not empty.

### Lemma 51

The (topological) interiors of Voronoi regions of distinct points of  $S$  are disjoint.



# Voronoi Diagram: Properties

## Lemma 52

A Voronoi node is the common intersection of exactly three Voronoi edges. It is equidistant to the three points of  $S$  which lie in the Voronoi regions it belongs to.



## Voronoi Diagram: Properties

### Lemma 52

A Voronoi node is the common intersection of exactly three Voronoi edges. It is equidistant to the three points of  $S$  which lie in the Voronoi regions it belongs to.

*Proof:*

- Let a Voronoi node  $v$  be the intersection of  $k$  edges  $e_1, e_2, \dots, e_k$ , with  $k \geq 2$ , which are ordered clockwise around  $v$ . Then  $e_1$  is equidistant to some points  $p_1$  and  $p_2$ ,  $e_2$  is equidistant to  $p_2$  and  $p_3$ , and so on, and  $e_k$  is equidistant to  $p_k$  and  $p_1$ .
- Thus,  $v$  is equidistant to  $p_1, p_2, \dots, p_k$ .
- Since a Voronoi region is convex, all points  $p_1, p_2, \dots, p_k$  are distinct.
- Based on our assumption that no more than three points are co-circular we conclude  $k \leq 3$ .
- However,  $k = 2$  would mean  $e_1$  is equidistant to  $p_1$  and  $p_2$ , and  $e_2$  is equidistant to  $p_2$  and  $p_1$ . Therefore, they would lie on the same bisector, and could not intersect in  $v$ .



## Voronoi Diagram: Properties

### Lemma 52

A Voronoi node is the common intersection of exactly three Voronoi edges. It is equidistant to the three points of  $S$  which lie in the Voronoi regions it belongs to.

*Proof:*

- Let a Voronoi node  $v$  be the intersection of  $k$  edges  $e_1, e_2, \dots, e_k$ , with  $k \geq 2$ , which are ordered clockwise around  $v$ . Then  $e_1$  is equidistant to some points  $p_1$  and  $p_2$ ,  $e_2$  is equidistant to  $p_2$  and  $p_3$ , and so on, and  $e_k$  is equidistant to  $p_k$  and  $p_1$ .
- Thus,  $v$  is equidistant to  $p_1, p_2, \dots, p_k$ .
- Since a Voronoi region is convex, all points  $p_1, p_2, \dots, p_k$  are distinct.
- Based on our assumption that no more than three points are co-circular we conclude  $k \leq 3$ .
- However,  $k = 2$  would mean  $e_1$  is equidistant to  $p_1$  and  $p_2$ , and  $e_2$  is equidistant to  $p_2$  and  $p_1$ . Therefore, they would lie on the same bisector, and could not intersect in  $v$ .



### Corollary 53

A Voronoi diagram is a 3-regular (planar) graph.



# Voronoi Diagram: Properties

## Lemma 54

The circle  $C$  centered at a Voronoi node  $v$  that passes through the node's three equidistant points  $p_1, p_2, p_3 \in S$  contains no other points of  $S$  in its interior.



# Voronoi Diagram: Properties

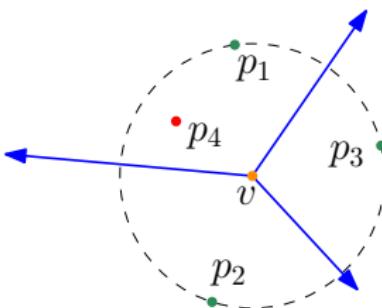
## Lemma 54

The circle  $C$  centered at a Voronoi node  $v$  that passes through the node's three equidistant points  $p_1, p_2, p_3 \in S$  contains no other points of  $S$  in its interior.

*Proof:*

- Assume that  $C$  contains another point  $p_4 \in S$  in its interior.
- Then  $v$  would be closer to  $p_4$  than to any of  $p_1, p_2, p_3$ . Therefore,  $v$  would lie in the interior of the Voronoi region of  $p_4$ .
- This is a contradiction because  $v$  lies on the common boundary of the Voronoi regions of  $p_1, p_2, p_3$ .

□



# Voronoi Diagram: Properties

## Lemma 55

For  $p_i \in S$ , every nearest neighbor of  $p_i$  defines an edge of  $\mathcal{VP}(p_i, S)$ .



# Voronoi Diagram: Properties

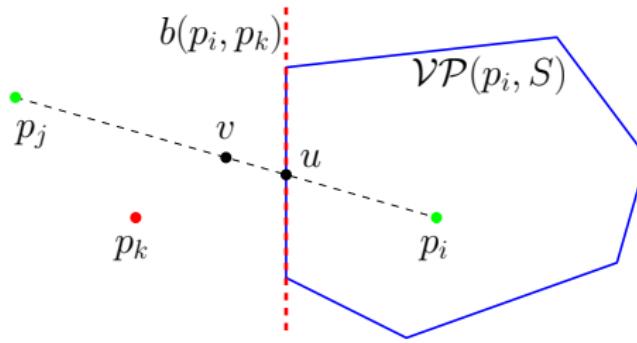
## Lemma 55

For  $p_i \in S$ , every nearest neighbor of  $p_i$  defines an edge of  $\mathcal{VP}(p_i, S)$ .

*Proof:*

- Let  $p_j \in S$  be a nearest neighbor of  $p_i$ , and let  $v$  be their midpoint.
- Suppose that  $v$  does not lie on the boundary of  $\mathcal{VP}(p_i, S)$ .
- Then the line segment  $\overline{p_iv}$  would intersect some edge of  $\mathcal{VP}(p_i, S)$ . Assume that it intersects the bisector of  $\overline{p_ip_k}$  in the point  $u$ . Now  $|\overline{p_iu}| < |\overline{p_iv}|$ , and therefore  $|\overline{p_ip_k}| \leq 2|\overline{p_iu}| < 2|\overline{p_iv}| = |\overline{p_ip_j}|$ , and we would have  $p_k$  closer to  $p_i$  than  $p_j$ , which is a contradiction.

□



# Delaunay Triangulation: Definition and Properties

## Definition 56 (Delaunay triangulation)

A *Delaunay triangulation* (DT),  $\mathcal{DT}(S)$ , of  $S$  is a geometric graph that is *dual* to the Voronoi diagram of  $S$ :

- The nodes of the graph are given by the points of  $S$ .
- Two points are connected by a line segment, and form an edge of  $\mathcal{DT}(S)$ , exactly if they share a Voronoi edge of  $\mathcal{VD}(S)$ .

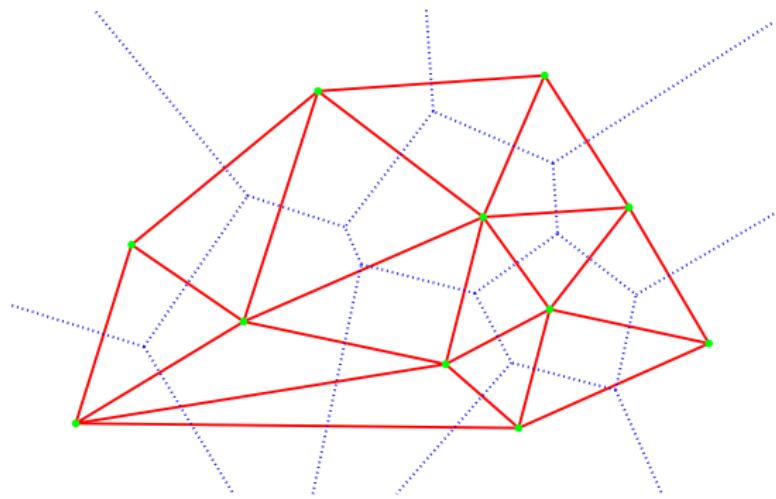


# Delaunay Triangulation: Definition and Properties

## Definition 56 (Delaunay triangulation)

A *Delaunay triangulation* (DT),  $\mathcal{DT}(S)$ , of  $S$  is a geometric graph that is *dual* to the Voronoi diagram of  $S$ :

- The nodes of the graph are given by the points of  $S$ .
- Two points are connected by a line segment, and form an edge of  $\mathcal{DT}(S)$ , exactly if they share a Voronoi edge of  $\mathcal{VD}(S)$ .



# Delaunay Triangulation: Definition and Properties

## Lemma 57

The structure  $\mathcal{DT}(S)$  does indeed form a triangulation of  $S$ .



# Delaunay Triangulation: Definition and Properties

## Lemma 57

The structure  $\mathcal{DT}(S)$  does indeed form a triangulation of  $S$ .

- Thus, the interior faces of  $\mathcal{DT}(S)$  are defined by triples of  $S$  which correspond to nodes of  $\mathcal{VD}(S)$ .
- By definition, every edge of the Delaunay triangulation has a corresponding edge in the Voronoi diagram.
- $\mathcal{DT}(S)$  is called the *straight-line dual* of  $\mathcal{VD}(S)$ .

# Delaunay Triangulation: Definition and Properties

## Lemma 57

The structure  $\mathcal{DT}(S)$  does indeed form a triangulation of  $S$ .

- Thus, the interior faces of  $\mathcal{DT}(S)$  are defined by triples of  $S$  which correspond to nodes of  $\mathcal{VD}(S)$ .
- By definition, every edge of the Delaunay triangulation has a corresponding edge in the Voronoi diagram.
- $\mathcal{DT}(S)$  is called the *straight-line dual* of  $\mathcal{VD}(S)$ .
- Note: An edge of  $\mathcal{DT}(S)$  need not intersect its dual Voronoi edge.
- If no four points of a point set are co-circular then its Delaunay triangulation is unique.
- Named after Boris Nikolaevich Delaunay (1890–1980).



# Complexity of Voronoi Diagram and Delaunay Triangulation

## Lemma 58

The Delaunay triangulation of  $n$  points has at most  $3n - 6$  edges and at most  $2n - 4$  faces.

# Complexity of Voronoi Diagram and Delaunay Triangulation

## Lemma 58

The Delaunay triangulation of  $n$  points has at most  $3n - 6$  edges and at most  $2n - 4$  faces.

*Proof:* Recall that a Delaunay triangulation forms a very special planar graph on  $n$  nodes, to which Euler's formula  $V - E + F = 2$  can be applied, with  $V = n$ . The planarity implies that we have

- at least  $E \geq \frac{3}{2}F$  edges,
- at most  $F \leq \frac{2}{3}E$  faces.

# Complexity of Voronoi Diagram and Delaunay Triangulation

## Lemma 58

The Delaunay triangulation of  $n$  points has at most  $3n - 6$  edges and at most  $2n - 4$  faces.

*Proof:* Recall that a Delaunay triangulation forms a very special planar graph on  $n$  nodes, to which Euler's formula  $V - E + F = 2$  can be applied, with  $V = n$ . The planarity implies that we have

- at least  $E \geq \frac{3}{2}F$  edges,
- at most  $F \leq \frac{2}{3}E$  faces.

We conclude that

$$\begin{aligned} \mathcal{D}\mathcal{T}: \quad & \leq 3n - 6 \text{ edges} & \text{and thus} & \quad \mathcal{V}\mathcal{D}: \quad \leq 3n - 6 \text{ edges}, \\ \mathcal{D}\mathcal{T}: \quad & \leq 2n - 4 \text{ faces} & \text{and thus} & \quad \mathcal{V}\mathcal{D}: \quad \leq 2n - 5 \text{ nodes}. \end{aligned}$$



# Complexity of Voronoi Diagram and Delaunay Triangulation

## Lemma 58

The Delaunay triangulation of  $n$  points has at most  $3n - 6$  edges and at most  $2n - 4$  faces.

*Proof:* Recall that a Delaunay triangulation forms a very special planar graph on  $n$  nodes, to which Euler's formula  $V - E + F = 2$  can be applied, with  $V = n$ . The planarity implies that we have

- at least  $E \geq \frac{3}{2}F$  edges,
- at most  $F \leq \frac{2}{3}E$  faces.

We conclude that

$$\begin{aligned} \mathcal{D}\mathcal{T}: & \leq 3n - 6 \text{ edges} & \text{and thus} & \quad \mathcal{V}\mathcal{D}: \leq 3n - 6 \text{ edges}, \\ \mathcal{D}\mathcal{T}: & \leq 2n - 4 \text{ faces} & \text{and thus} & \quad \mathcal{V}\mathcal{D}: \leq 2n - 5 \text{ nodes}. \end{aligned}$$

□

## Lemma 59

The Voronoi diagram of  $n$  points has at most  $3n - 6$  edges and at most  $2n - 5$  nodes.

# Complexity of Voronoi Diagram and Delaunay Triangulation

## Lemma 58

The Delaunay triangulation of  $n$  points has at most  $3n - 6$  edges and at most  $2n - 4$  faces.

*Proof:* Recall that a Delaunay triangulation forms a very special planar graph on  $n$  nodes, to which Euler's formula  $V - E + F = 2$  can be applied, with  $V = n$ . The planarity implies that we have

- at least  $E \geq \frac{3}{2}F$  edges,
- at most  $F \leq \frac{2}{3}E$  faces.

We conclude that

$$\begin{aligned}\mathcal{D}\mathcal{T}: \quad &\leq 3n - 6 \text{ edges} & \text{and thus} & \quad \mathcal{V}\mathcal{D}: \quad \leq 3n - 6 \text{ edges}, \\ \mathcal{D}\mathcal{T}: \quad &\leq 2n - 4 \text{ faces} & \text{and thus} & \quad \mathcal{V}\mathcal{D}: \quad \leq 2n - 5 \text{ nodes.}\end{aligned}$$



## Lemma 59

The Voronoi diagram of  $n$  points has at most  $3n - 6$  edges and at most  $2n - 5$  nodes.

## Corollary 60

A Voronoi polygon has at most  $n - 1$  edges, but only six edges on average.



## Proximity Problems Solved by Voronoi Diagrams

- The fact that the Voronoi polygons of nearest neighbors always have a Voronoi edge in common implies that it is sufficient to check all points in adjacent Voronoi regions to find a nearest neighbor of a point  $p_i$ .
- Thus, knowledge of the Voronoi diagram helps to solve CLOSESTPAIR and ALLNEARESTNEIGHBORS in  $O(n)$  time.



## Proximity Problems Solved by Voronoi Diagrams

- The fact that the Voronoi polygons of nearest neighbors always have a Voronoi edge in common implies that it is sufficient to check all points in adjacent Voronoi regions to find a nearest neighbor of a point  $p_i$ .
- Thus, knowledge of the Voronoi diagram helps to solve CLOSESTPAIR and ALLNEARESTNEIGHBORS in  $O(n)$  time.
- The Voronoi polygon of  $p_i$  is unbounded if and only if  $p_i$  is a point of the convex hull of the set  $S$ . (Proof: See Preparata&Shamos.) This means that the vertices of  $CH(S)$  are those points of  $S$  which have unbounded Voronoi polygons.
- Thus, knowledge of the Voronoi diagram allows to solve CONVEXHULL in  $O(n)$  time.



## Proximity Problems Solved by Voronoi Diagrams

- The fact that the Voronoi polygons of nearest neighbors always have a Voronoi edge in common implies that it is sufficient to check all points in adjacent Voronoi regions to find a nearest neighbor of a point  $p_i$ .
- Thus, knowledge of the Voronoi diagram helps to solve CLOSESTPAIR and ALLNEARESTNEIGHBORS in  $O(n)$  time.
- The Voronoi polygon of  $p_i$  is unbounded if and only if  $p_i$  is a point of the convex hull of the set  $S$ . (Proof: See Preparata&Shamos.) This means that the vertices of  $CH(S)$  are those points of  $S$  which have unbounded Voronoi polygons.
- Thus, knowledge of the Voronoi diagram allows to solve CONVEXHULL in  $O(n)$  time.
- A MAXIMUMEMPTYCIRCLE can be found in  $O(n)$  time by scanning all nodes of the Voronoi diagram; see later.
- After  $O(n)$  preprocessing for building a search data structure of size  $O(n)$  on top of the Voronoi diagram, NEARESTNEIGHBORSEARCH queries can be handled in  $O(\log n)$  time. (However, the constants are high — better techniques are known for point sites!)

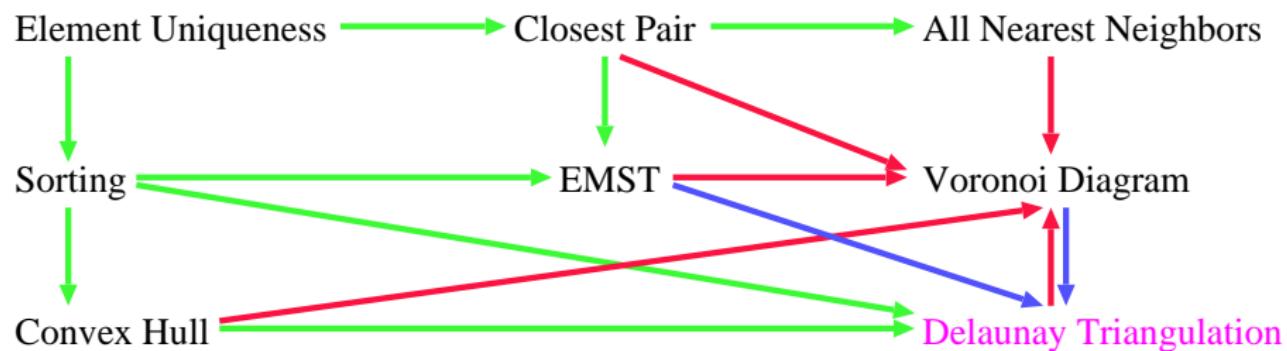
## Lemma 61

The Voronoi diagram of  $n$  points in  $\mathbb{R}^2$  can be obtained in  $O(n)$  time from the Delaunay triangulation, and the Delaunay triangulation can be obtained in  $O(n)$  time from the Voronoi diagram.

# Reductions Among Proximity Problems

## Lemma 61

The Voronoi diagram of  $n$  points in  $\mathbb{R}^2$  can be obtained in  $O(n)$  time from the Delaunay triangulation, and the Delaunay triangulation can be obtained in  $O(n)$  time from the Voronoi diagram.



## Voronoi Diagrams of Points

- Definition and Properties
- Algorithms
  - Divide&Conquer Algorithm
  - Incremental Construction
  - Sweep-Line Algorithm
  - Construction via Lifting to 3D
  - Approximate Voronoi Diagram by Means of Graphics Hardware

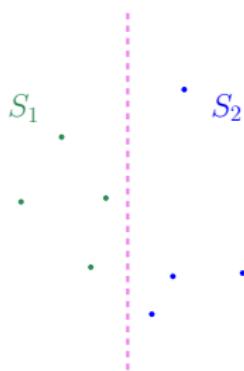
## Divide&Conquer Algorithm

- Preprocessing: Sort the points of  $S$  by  $x$ -coordinates. This takes  $O(n \log n)$  time.



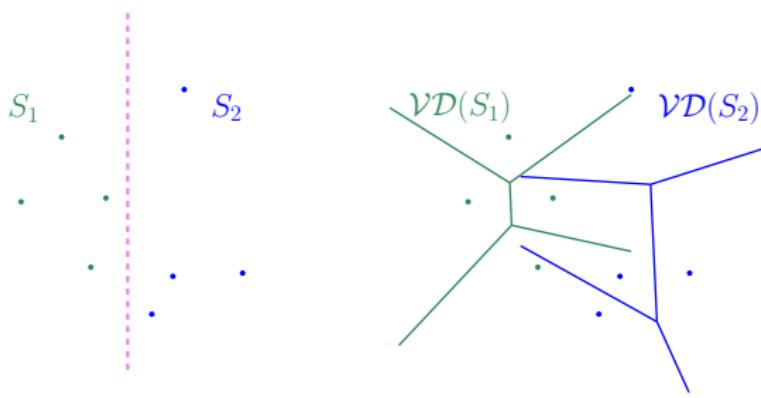
## Divide&Conquer Algorithm

- Preprocessing: Sort the points of  $S$  by  $x$ -coordinates. This takes  $O(n \log n)$  time.
- Divide:
  - Divide  $S$  into two subsets  $S_1$  and  $S_2$  of roughly equal size such that the points in  $S_1$  lie to the left and the points in  $S_2$  lie to the right of a vertical line.
  - This step can be carried out in  $O(n)$  time.



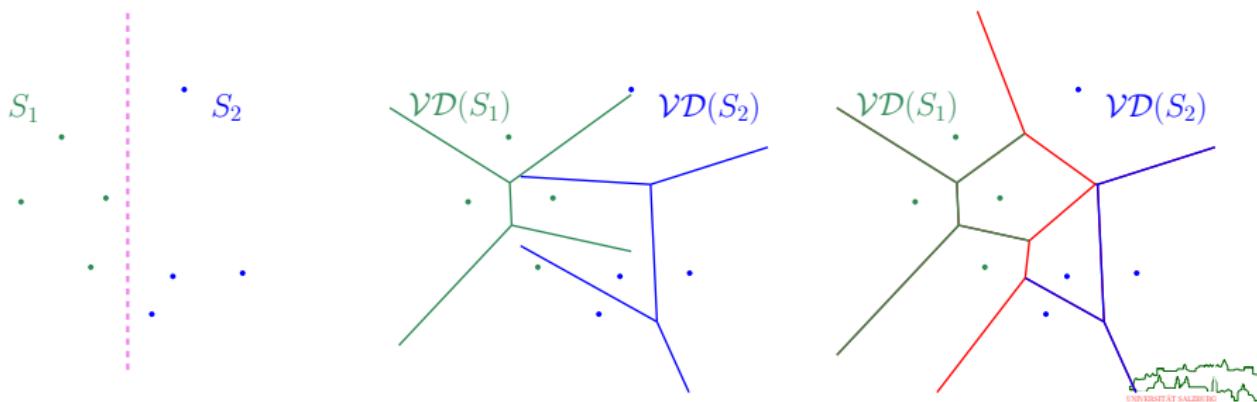
# Divide & Conquer Algorithm

- Preprocessing: Sort the points of  $S$  by  $x$ -coordinates. This takes  $O(n \log n)$  time.
- Divide:
  - Divide  $S$  into two subsets  $S_1$  and  $S_2$  of roughly equal size such that the points in  $S_1$  lie to the left and the points in  $S_2$  lie to the right of a vertical line.
  - This step can be carried out in  $O(n)$  time.
- Conquer (aka “Merge”):
  - Assume that  $\text{VD}(S_1)$  and  $\text{VD}(S_2)$  are known.



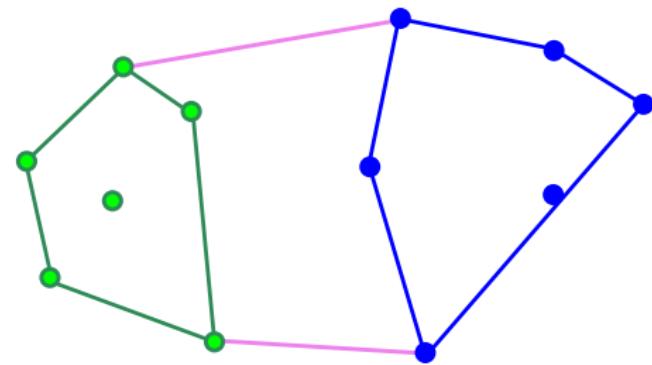
## Divide&Conquer Algorithm

- Preprocessing: Sort the points of  $S$  by  $x$ -coordinates. This takes  $O(n \log n)$  time.
- Divide:
  - Divide  $S$  into two subsets  $S_1$  and  $S_2$  of roughly equal size such that the points in  $S_1$  lie to the left and the points in  $S_2$  lie to the right of a vertical line.
  - This step can be carried out in  $O(n)$  time.
- Conquer (aka “Merge”):
  - Assume that  $\mathcal{VD}(S_1)$  and  $\mathcal{VD}(S_2)$  are known.
  - Clip those parts of  $\mathcal{VD}(S_1)$  that lie to the “right” of a so-called *dividing chain*.
  - Analogously for  $\mathcal{VD}(S_2)$ .



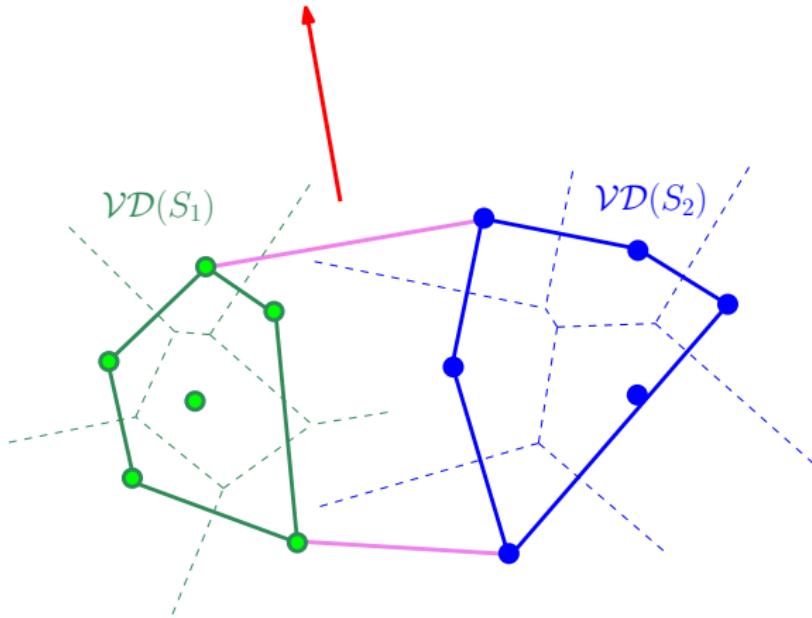
## Divide&Conquer Algorithm: Merge

- 1 Find upper and lower bridges of the convex hull.



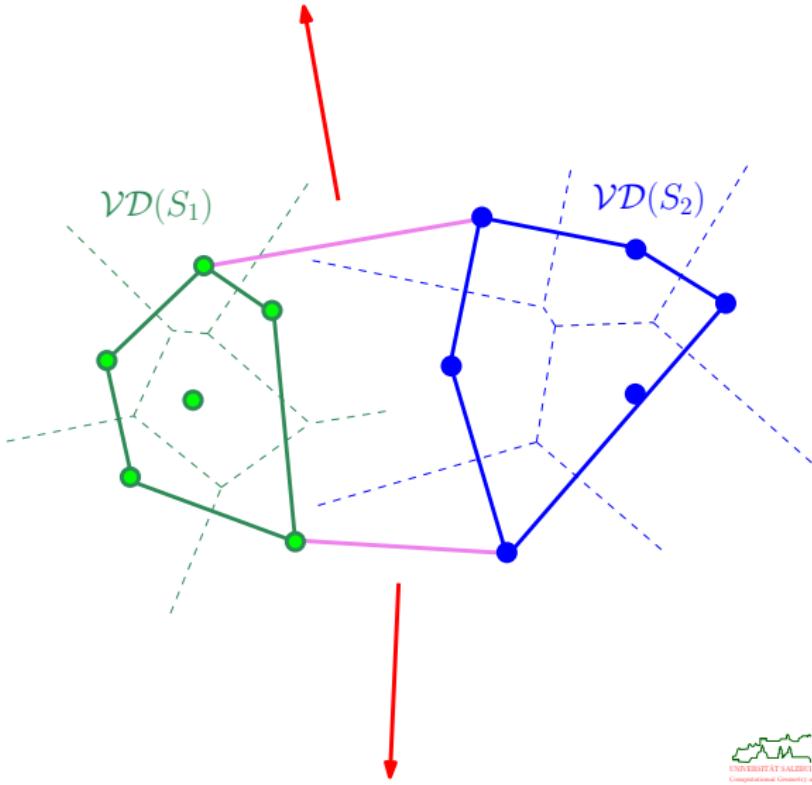
## Divide&Conquer Algorithm: Merge

- ① Find upper and lower bridges of the convex hull.
  - Bisector (ray) defined by upper bridge of convex hull is part of the dividing chain.



## Divide&Conquer Algorithm: Merge

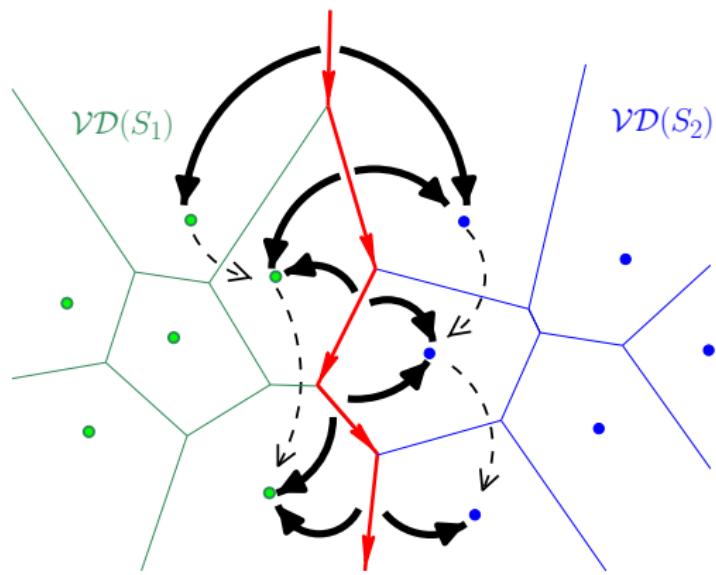
- ① Find upper and lower bridges of the convex hull.
  - Bisector (ray) defined by upper bridge of convex hull is part of the dividing chain.
  - Bisector (ray) defined by lower bridge of convex hull is part of the dividing chain.



## Divide&Conquer Algorithm: Merge

② Build dividing chain from top to bottom:

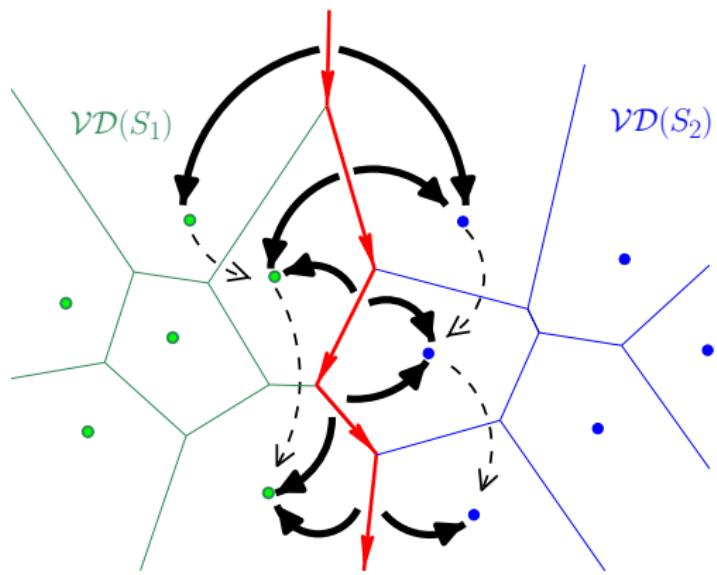
- Start by walking down along the upper ray.
- Intersect the ray with  $\mathcal{VD}(S_1)$  and  $\mathcal{VD}(S_2)$ .
- Pick the first intersection as new Voronoi node.



## Divide&Conquer Algorithm: Merge

② Build dividing chain from top to bottom:

- Start by walking down along the upper ray.
- Intersect the ray with  $\mathcal{VD}(S_1)$  and  $\mathcal{VD}(S_2)$ .
- Pick the first intersection as new Voronoi node.
- The next ray is the new bisector originating at this node.
- Continue this jagged walk until the lower ray is reached.



## Lemma 62

The merge can be carried out in  $O(n)$  time, based on the Shamos-Hoey scanning scheme that prevents Voronoi edges from being searched for an intersection for more than a constant number of times.

## Lemma 62

The merge can be carried out in  $O(n)$  time, based on the Shamos-Hoey scanning scheme that prevents Voronoi edges from being searched for an intersection for more than a constant number of times.

- If the merge is carried out in linear time then we get a familiar recurrence relation for the time  $T$ :

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n), \quad \text{and thus } T \in O(n \log n).$$

## Lemma 62

The merge can be carried out in  $O(n)$  time, based on the Shamos-Hoey scanning scheme that prevents Voronoi edges from being searched for an intersection for more than a constant number of times.

- If the merge is carried out in linear time then we get a familiar recurrence relation for the time  $T$ :

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n), \quad \text{and thus } T \in O(n \log n).$$

## Theorem 63

The divide&conquer algorithm can compute  $\mathcal{VD}(S)$  for a set  $S$  of  $n$  points in optimal  $O(n \log n)$  time.



## Incremental Construction

- We compute the Voronoi diagram  $\mathcal{VD}(S)$  of a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points by inserting the  $i$ -th point  $p_i$  into  $\mathcal{VD}(\{p_1, p_2, \dots, p_{i-1}\})$ , for  $1 \leq i \leq n$ .
- Let  $S' := \{p_1, p_2, \dots, p_{i-1}\}$ .



## Incremental Construction

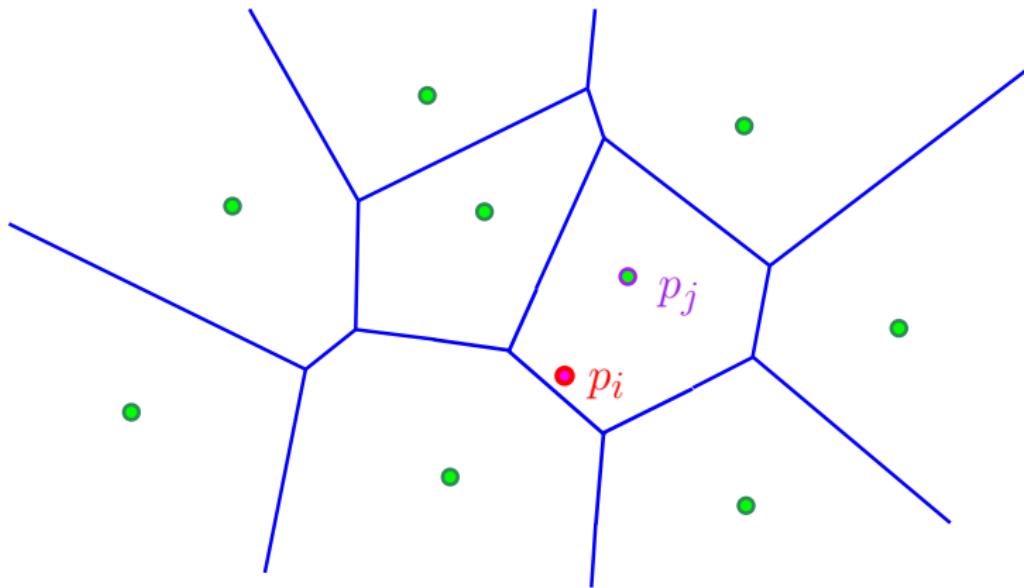
- We compute the Voronoi diagram  $\mathcal{VD}(S)$  of a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points by inserting the  $i$ -th point  $p_i$  into  $\mathcal{VD}(\{p_1, p_2, \dots, p_{i-1}\})$ , for  $1 \leq i \leq n$ .
- Let  $S' := \{p_1, p_2, \dots, p_{i-1}\}$ .
- If we could achieve constant complexity per insertion then a linear algorithm would result:
  - Best case:  $O(n)$ .
- An insertion could, however, affect all other sites:
  - Worst case:  $O(n^2)$ , or even worse.

## Incremental Construction

- We compute the Voronoi diagram  $\mathcal{VD}(S)$  of a set  $S := \{p_1, p_2, \dots, p_n\}$  of  $n$  points by inserting the  $i$ -th point  $p_i$  into  $\mathcal{VD}(\{p_1, p_2, \dots, p_{i-1}\})$ , for  $1 \leq i \leq n$ .
- Let  $S' := \{p_1, p_2, \dots, p_{i-1}\}$ .
- If we could achieve constant complexity per insertion then a linear algorithm would result:
  - Best case:  $O(n)$ .
- An insertion could, however, affect all other sites:
  - Worst case:  $O(n^2)$ , or even worse.
- Since, on average, every Voronoi region is bounded by six Voronoi edges there is reason to hope that a close-to-linear time complexity can be achieved.

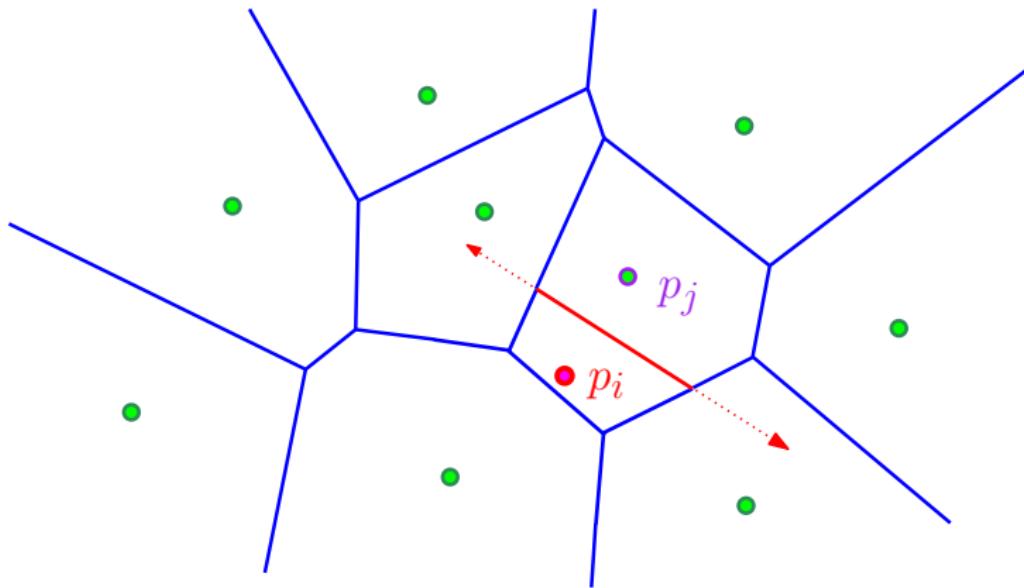
## Incremental Construction: Basic Algorithm

- ① Nearest-neighbor search among  $S'$ : Determine  $1 \leq j < i$  such that the new point  $p_i$  lies in  $\mathcal{VR}(p_j, S')$ .



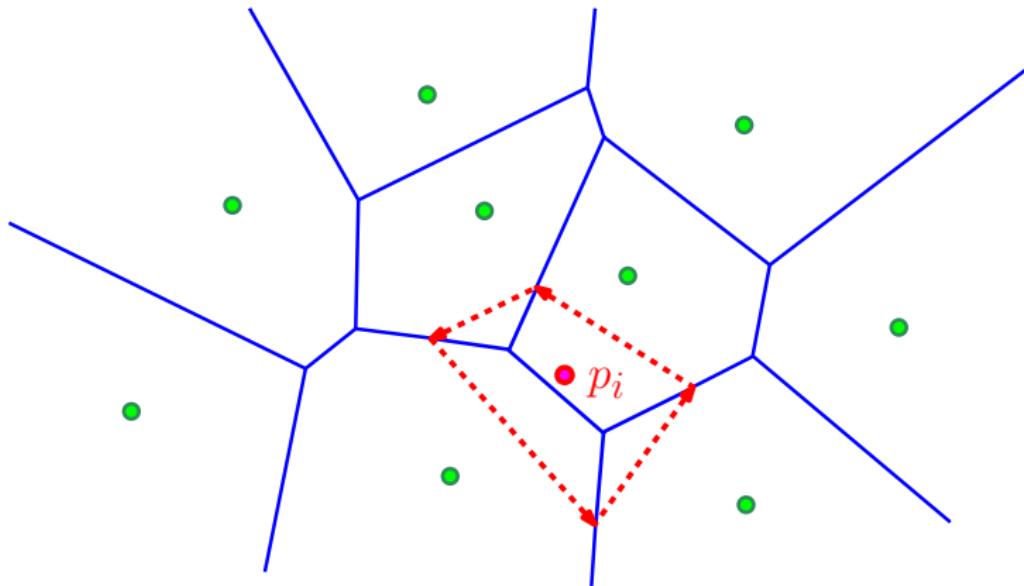
## Incremental Construction: Basic Algorithm

- ② Construct the bisector  $b(p_i, p_j)$  between  $p_i$  and  $p_j$ , intersect it with  $\mathcal{VP}(p_j, S')$ , and clip that portion of  $\mathcal{VP}(p_j, S')$  which is closer to  $p_i$  than to  $p_j$ .



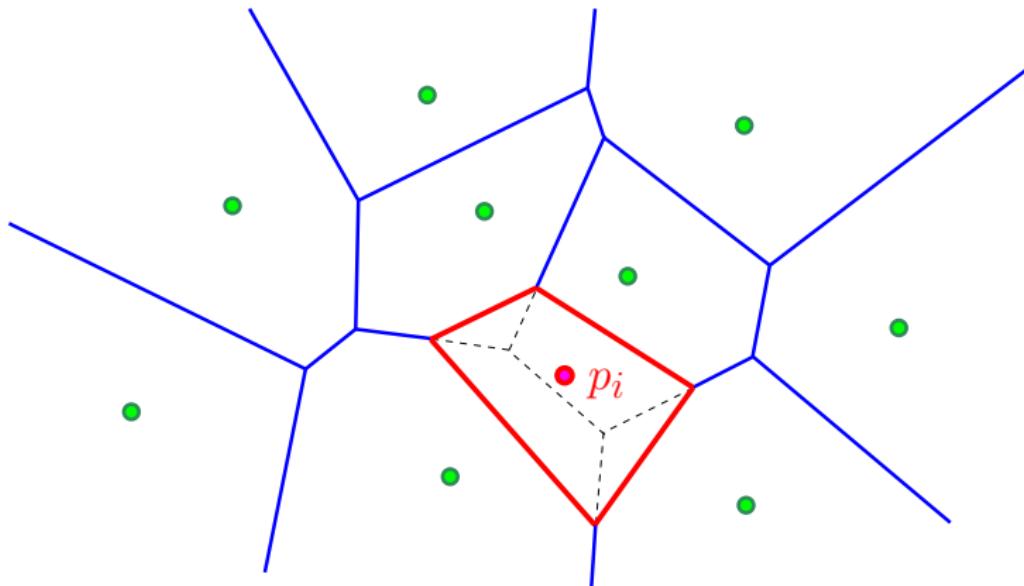
## Incremental Construction: Basic Algorithm

- ③ Generate  $\mathcal{VP}(p_i, \{p_1, p_2, \dots, p_i\})$  by a circular scan around  $p_i$ , similar to the construction of the dividing chain in the divide&conquer algorithm.



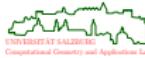
## Incremental Construction: Basic Algorithm

- The scan is finished once it returns to  $\mathcal{VR}(p_j, S')$ . What is the complexity of one insertion?



## Incremental Construction: Complexity of Nearest-Neighbor Search

- The complexity mainly depends on the complexity of the nearest-neighbor search and on the number of edges generated/deleted during the scan.



## Incremental Construction: Complexity of Nearest-Neighbor Search

- The complexity mainly depends on the complexity of the nearest-neighbor search and on the number of edges generated/deleted during the scan.
- Nearest-neighbor search by brute force:
  - Compute all possible distances.
  - Not very practical:  $O(n)$  per nearest-neighbor query.

## Incremental Construction: Complexity of Nearest-Neighbor Search

- The complexity mainly depends on the complexity of the nearest-neighbor search and on the number of edges generated/deleted during the scan.
- Nearest-neighbor search by brute force:
  - Compute all possible distances.
  - Not very practical:  $O(n)$  per nearest-neighbor query.
- Nearest-neighbor search akin to “steepest ascent”:
  - Guess an initial candidate for  $p_j$ .
  - Compare  $d(p_i, p_j)$  to the distances between  $p_i$  and the neighbors of  $p_j$ .
  - Select the closest neighbor point as new candidate  $p_j$ .
  - Continue with the new point, until all neighbors have a distance from  $p_i$  that is larger than  $d(p_i, p_j)$ .
  - This approach heavily depends on the availability of a good initial guess.

## Incremental Construction: Complexity of Nearest-Neighbor Search

- The complexity mainly depends on the complexity of the nearest-neighbor search and on the number of edges generated/deleted during the scan.
- Nearest-neighbor search by brute force:
  - Compute all possible distances.
  - Not very practical:  $O(n)$  per nearest-neighbor query.
- Nearest-neighbor search akin to “steepest ascent”:
  - Guess an initial candidate for  $p_j$ .
  - Compare  $d(p_i, p_j)$  to the distances between  $p_i$  and the neighbors of  $p_j$ .
  - Select the closest neighbor point as new candidate  $p_j$ .
  - Continue with the new point, until all neighbors have a distance from  $p_i$  that is larger than  $d(p_i, p_j)$ .
  - This approach heavily depends on the availability of a good initial guess.
- Nearest-neighbor search based on *geometric hashing*:
  - Register  $p_1, p_2, p_{i-1}$  in a uniform grid.
  - Locate the grid cell that contains  $p_i$ , and find nearest neighbor.
  - This approach works best for a uniform distribution of the points.

## Geometric Hashing for Nearest-Neighbor Searching

- The bounding box of the input points is partitioned into rectangular cells of uniform size by means of a regular grid.
- For every cell  $c$ , all points of  $\{p_1, p_2, \dots, p_{i-1}\}$  that lie in  $c$  are stored with  $c$ .  
(Alternatively, only one point is stored per cell.)



## Geometric Hashing for Nearest-Neighbor Searching

- The bounding box of the input points is partitioned into rectangular cells of uniform size by means of a regular grid.
- For every cell  $c$ , all points of  $\{p_1, p_2, \dots, p_{i-1}\}$  that lie in  $c$  are stored with  $c$ . (Alternatively, only one point is stored per cell.)
- To find the point  $p_j$  nearest to point  $p_i$ :
  - Determine the cell  $c$  that in which  $p_i$  lies.
  - By searching in  $c$  (and possibly in its neighboring cells, if  $c$  is empty), we find a first candidate for the nearest neighbor.
  - Let  $\delta$  be the distance from  $p_i$  to this point.



## Geometric Hashing for Nearest-Neighbor Searching

- The bounding box of the input points is partitioned into rectangular cells of uniform size by means of a regular grid.
- For every cell  $c$ , all points of  $\{p_1, p_2, \dots, p_{i-1}\}$  that lie in  $c$  are stored with  $c$ . (Alternatively, only one point is stored per cell.)
- To find the point  $p_j$  nearest to point  $p_i$ :
  - Determine the cell  $c$  that in which  $p_i$  lies.
  - By searching in  $c$  (and possibly in its neighboring cells, if  $c$  is empty), we find a first candidate for the nearest neighbor.
  - Let  $\delta$  be the distance from  $p_i$  to this point.
  - We continue searching in  $c$  and in those cells around  $c$  which are intersected by a clearance circle with radius  $\delta$  centered at  $p_i$ .
  - Whenever a point that is closer is found, we update  $\delta$  appropriately.



## Geometric Hashing for Nearest-Neighbor Searching

- The bounding box of the input points is partitioned into rectangular cells of uniform size by means of a regular grid.
- For every cell  $c$ , all points of  $\{p_1, p_2, \dots, p_{i-1}\}$  that lie in  $c$  are stored with  $c$ . (Alternatively, only one point is stored per cell.)
- To find the point  $p_j$  nearest to point  $p_i$ :
  - Determine the cell  $c$  that in which  $p_i$  lies.
  - By searching in  $c$  (and possibly in its neighboring cells, if  $c$  is empty), we find a first candidate for the nearest neighbor.
  - Let  $\delta$  be the distance from  $p_i$  to this point.
  - We continue searching in  $c$  and in those cells around  $c$  which are intersected by a clearance circle with radius  $\delta$  centered at  $p_i$ .
  - Whenever a point that is closer is found, we update  $\delta$  appropriately.
  - The search stops once no unsearched cell exists that is intersected by the clearance circle.



## Geometric Hashing for Nearest-Neighbor Searching

- If all points are stored per cell then the true nearest neighbor is found. If only one point is stored per cell then this approach yields a (hopefully) good initial candidate for the nearest neighbor.



## Geometric Hashing for Nearest-Neighbor Searching

- If all points are stored per cell then the true nearest neighbor is found. If only one point is stored per cell then this approach yields a (hopefully) good initial candidate for the nearest neighbor.
- What is a suitable resolution of the grid? There is no universally valid answer. In any case, the grid should not use more than  $O(n)$  memory!



## Geometric Hashing for Nearest-Neighbor Searching

- If all points are stored per cell then the true nearest neighbor is found. If only one point is stored per cell then this approach yields a (hopefully) good initial candidate for the nearest neighbor.
- What is a suitable resolution of the grid? There is no universally valid answer. In any case, the grid should not use more than  $O(n)$  memory!
- Personal experience:
  - Grids of the form  $(w \cdot \sqrt{n}) \times (h \cdot \sqrt{n})$  seem to work nicely, with  $w \cdot h = c$  for some constant  $c$ .
  - The parameters  $w, h$  are chosen to adapt the resolution of the grid to the aspect ratio of the bounding box of the points.
  - By experiment:  $1 \leq c \leq 2$ .



## Geometric Hashing for Nearest-Neighbor Searching

- If all points are stored per cell then the true nearest neighbor is found. If only one point is stored per cell then this approach yields a (hopefully) good initial candidate for the nearest neighbor.
- What is a suitable resolution of the grid? There is no universally valid answer. In any case, the grid should not use more than  $O(n)$  memory!
- Personal experience:
  - Grids of the form  $(w \cdot \sqrt{n}) \times (h \cdot \sqrt{n})$  seem to work nicely, with  $w \cdot h = c$  for some constant  $c$ .
  - The parameters  $w, h$  are chosen to adapt the resolution of the grid to the aspect ratio of the bounding box of the points.
  - By experiment:  $1 \leq c \leq 2$ .
- This basic scheme can be tuned considerably, e.g., by switching to 2D-trees if a small sample of the points indicates that the points are distributed highly non-uniformly.



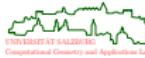
## Geometric Hashing for Nearest-Neighbor Searching

- If all points are stored per cell then the true nearest neighbor is found. If only one point is stored per cell then this approach yields a (hopefully) good initial candidate for the nearest neighbor.
- What is a suitable resolution of the grid? There is no universally valid answer. In any case, the grid should not use more than  $O(n)$  memory!
- Personal experience:
  - Grids of the form  $(w \cdot \sqrt{n}) \times (h \cdot \sqrt{n})$  seem to work nicely, with  $w \cdot h = c$  for some constant  $c$ .
  - The parameters  $w, h$  are chosen to adapt the resolution of the grid to the aspect ratio of the bounding box of the points.
  - By experiment:  $1 \leq c \leq 2$ .
- This basic scheme can be tuned considerably, e.g., by switching to 2D-trees if a small sample of the points indicates that the points are distributed highly non-uniformly.
- Note: Hash-based nearest-neighbor searching will work best for points that are distributed uniformly, and will fail miserably if all points end up in one cell!



## Randomized Incremental Construction

- Nearest-neighbor search based on a *history DAG* and *randomized incremental insertion*:
  - Assume that the points are inserted in random order.



## Randomized Incremental Construction

- Nearest-neighbor search based on a *history DAG* and *randomized incremental insertion*:
  - Assume that the points are inserted in random order.
  - One can prove (using backwards analysis): the history DAG has  $O(n)$  expected size, and supports nearest-neighbor queries in  $O(\log n)$  expected time.



# Randomized Incremental Construction

- Nearest-neighbor search based on a *history DAG* and *randomized incremental insertion*:
  - Assume that the points are inserted in random order.
  - One can prove (using backwards analysis): the history DAG has  $O(n)$  expected size, and supports nearest-neighbor queries in  $O(\log n)$  expected time.
- Randomized incremental insertion:
  - One can prove (using backwards analysis): the update necessary for inserting one point can be done in  $O(1)$  expected time.
  - Note: This is independent of the point distribution, as long as the order is random!



# Randomized Incremental Construction

- Nearest-neighbor search based on a *history DAG* and *randomized incremental insertion*:
  - Assume that the points are inserted in random order.
  - One can prove (using backwards analysis): the history DAG has  $O(n)$  expected size, and supports nearest-neighbor queries in  $O(\log n)$  expected time.
- Randomized incremental insertion:
  - One can prove (using backwards analysis): the update necessary for inserting one point can be done in  $O(1)$  expected time.
  - Note: This is independent of the point distribution, as long as the order is random!
- Worst-case insertion order:
  - It is fairly easy to pick  $n$  points and number them “appropriately” such that the insertion of the  $i$ -th point requires the generation of  $O(i)$  Voronoi edges!

## Theorem 64

If randomized incremental construction is used then one can compute a Voronoi diagram of  $n$  points incrementally in  $O(n \log n)$  expected time.

## Incremental Construction: Overall Complexity

### Theorem 64

If randomized incremental construction is used then one can compute a Voronoi diagram of  $n$  points incrementally in  $O(n \log n)$  expected time.

- If points are uniformly distributed then geometric hashing tends to answer a nearest-neighbor query in  $O(1)$  time.
- Thus, randomized incremental insertion of  $n$  uniformly distributed points can be assumed to generate the Voronoi diagram in linear or slightly super-linear time. (And this claim is supported by practical experiments, even clustered points!)
- However, the worst case still is  $O(n^2)$ , no matter how efficiently all nearest-neighbor queries are answered!



## Sweep-Line Algorithm

- Can a sweep-line algorithm be applied to compute the Voronoi diagram?



## Sweep-Line Algorithm

- Can a sweep-line algorithm be applied to compute the Voronoi diagram?
- Principal problem: When the sweep line reaches an extreme (e.g., top-most) vertex of  $\mathcal{VP}(p_i, S)$ , it has not yet moved over  $p_i$ .
- Thus, the information on the point sites is missing when a Voronoi polygon is first encountered and Voronoi nodes are to be computed.
- This problem is independent of the sweep direction chosen. Thus, w.l.o.g., we move the sweep line  $\ell$  from top to bottom.

## Sweep-Line Algorithm

- Can a sweep-line algorithm be applied to compute the Voronoi diagram?
- Principal problem: When the sweep line reaches an extreme (e.g., top-most) vertex of  $\mathcal{VP}(p_i, S)$ , it has not yet moved over  $p_i$ .
- Thus, the information on the point sites is missing when a Voronoi polygon is first encountered and Voronoi nodes are to be computed.
- This problem is independent of the sweep direction chosen. Thus, w.l.o.g., we move the sweep line  $\ell$  from top to bottom.
- Remarkable idea (by S. Fortune): Rather than keeping the actual intersection of the Voronoi diagram with  $\ell$ , we maintain information on that part of the Voronoi diagram of the points above  $\ell$  that is not affected by points below  $\ell$ .

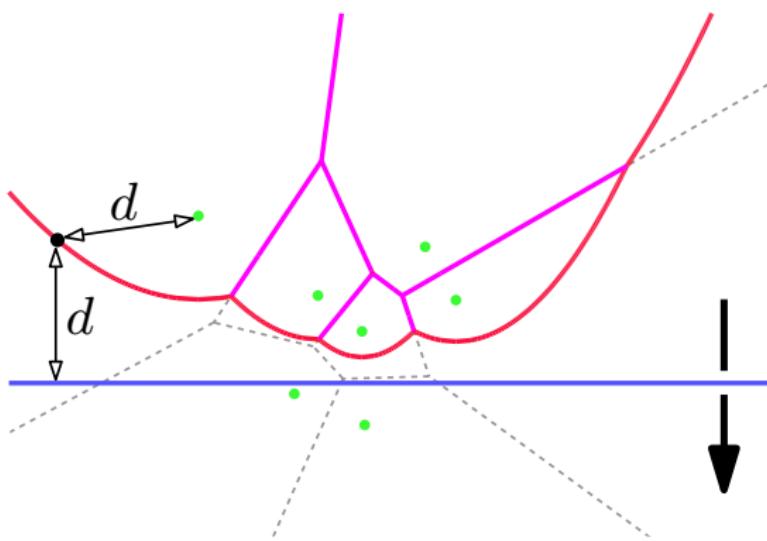
## Sweep-Line Algorithm

- Can a sweep-line algorithm be applied to compute the Voronoi diagram?
- Principal problem: When the sweep line reaches an extreme (e.g., top-most) vertex of  $\mathcal{VP}(p_i, S)$ , it has not yet moved over  $p_i$ .
- Thus, the information on the point sites is missing when a Voronoi polygon is first encountered and Voronoi nodes are to be computed.
- This problem is independent of the sweep direction chosen. Thus, w.l.o.g., we move the sweep line  $\ell$  from top to bottom.
- Remarkable idea (by S. Fortune): Rather than keeping the actual intersection of the Voronoi diagram with  $\ell$ , we maintain information on that part of the Voronoi diagram of the points above  $\ell$  that is not affected by points below  $\ell$ .
- That part of the Voronoi diagram under construction lies above a *beach line* consisting of parabolic arcs: Each parabolic arc is defined by  $\ell$  and a point above  $\ell$ .



## Sweep-Line Algorithm: Beach Line

- The part of the Voronoi diagram that will not change any more as the sweep line continues to move downwards lies above the beach line formed by the lower envelope of parabolic arcs.



## Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.  
A full sweep reveals the complete Voronoi diagram.

## Sweep-Line Algorithm: Events

- The following two events need to be considered for the event-point schedule:
  - ➊ Site event:
    - The sweep line  $\ell$  passes through an input point, and a new parabolic arc needs to be inserted into the beach line.



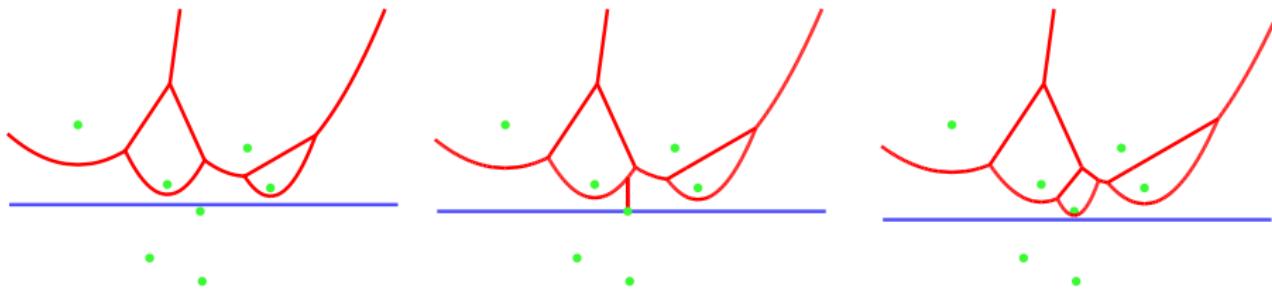
## Sweep-Line Algorithm: Events

- The following two events need to be considered for the event-point schedule:
  - ➊ Site event:
    - The sweep line  $\ell$  passes through an input point, and a new parabolic arc needs to be inserted into the beach line.
  - ➋ Circle event:
    - A parabolic arc of the beach line vanishes, i.e., degenerates to a point  $v$ , and a new Voronoi node has to be inserted at  $v$ .
    - What does this mean for the sweep line  $\ell$ ? What is the appropriate  $y$ -position of  $\ell$  to catch this event?



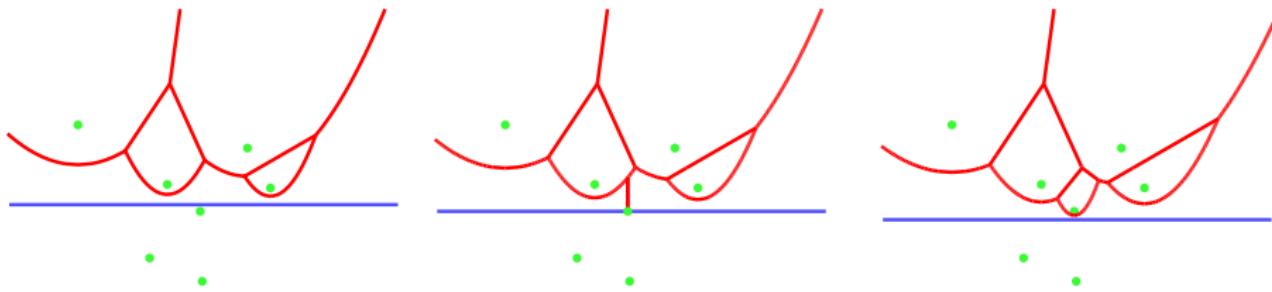
## Sweep-Line Algorithm: Site Event

- If the sweep line  $\ell$  passes through an input point then a new parabolic arc needs to be inserted into the beach line. Initially, this arc is degenerate.



## Sweep-Line Algorithm: Site Event

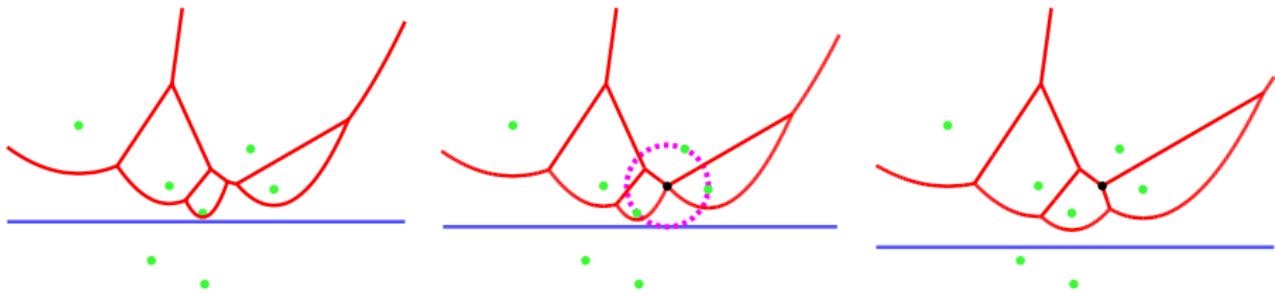
- If the sweep line  $\ell$  passes through an input point then a new parabolic arc needs to be inserted into the beach line. Initially, this arc is degenerate.



- This event occurs whenever the sweep line  $\ell$  passes through an input point  $p_i$ .
- It is responsible for the initialization of a new Voronoi region that will become  $\mathcal{VR}(p_i, S)$ .

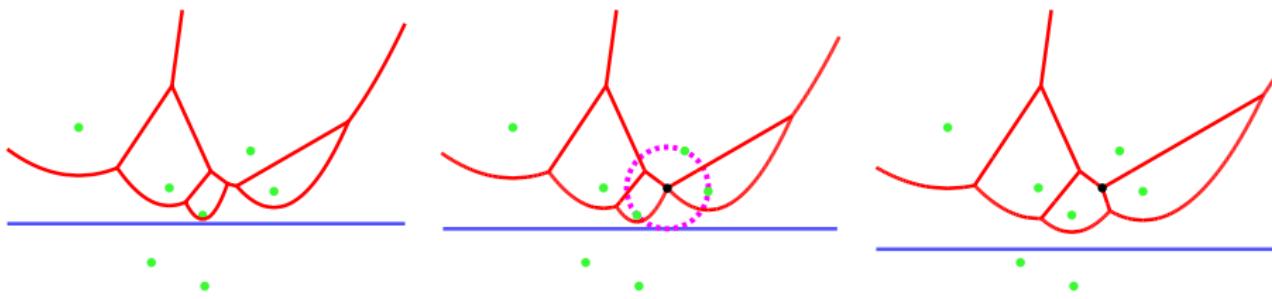
## Sweep-Line Algorithm: Circle Event

- If a parabolic arc of the beach line degenerates to a point  $v$  then a new Voronoi node needs to be inserted at  $v$ .



## Sweep-Line Algorithm: Circle Event

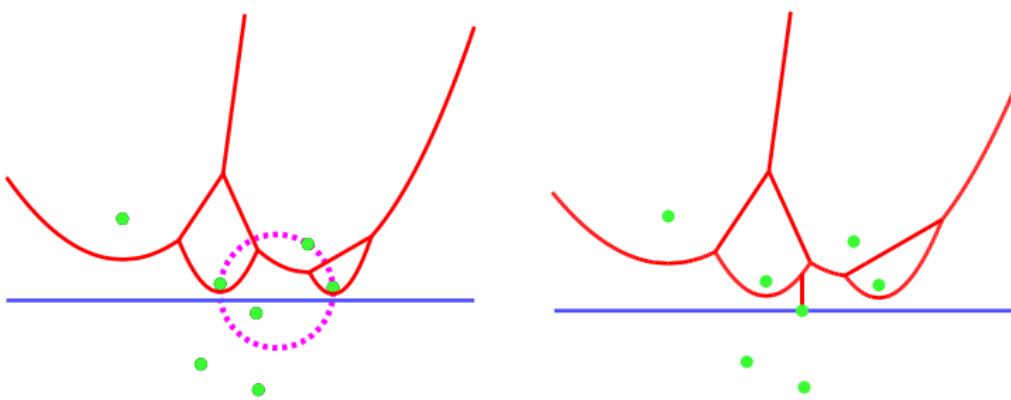
- If a parabolic arc of the beach line degenerates to a point  $v$  then a new Voronoi node needs to be inserted at  $v$ .



- A circle event occurs when the sweep line  $\ell$  passes over the south pole of a circle through the three defining input points  $p_i, p_j, p_k$  of three consecutive parabolic arcs of the beach line.
- The center  $v$  of such a circle is equidistant to  $p_i, p_j, p_k$  and also to  $\ell$ ; it becomes a new node of the Voronoi diagram.

## Sweep-Line Algorithm: False Alarms

- Not all scheduled circle events correspond to valid new Voronoi nodes: A circle event has to be processed only if its defining three parabolic arcs still are consecutive members of the beach line at the time when the sweep line  $\ell$  passes over the south pole of the circle.



## Sweep-Line Algorithm: Event-Point Schedule and Sweep-Line Status

- All input points are stored in sorted order (according to  $y$ -coordinates) in the event-point schedule.
- Whenever three parabolic arcs become consecutive for the first time — when a site event occurs — the  $y$ -coordinate of the corresponding circle event is inserted into the event-point schedule at the appropriate place.



## Sweep-Line Algorithm: Event-Point Schedule and Sweep-Line Status

- All input points are stored in sorted order (according to  $y$ -coordinates) in the event-point schedule.
- Whenever three parabolic arcs become consecutive for the first time — when a site event occurs — the  $y$ -coordinate of the corresponding circle event is inserted into the event-point schedule at the appropriate place.
- Parabolic arcs have to be inserted into the beach line when processing site events, and have to be deleted when processing circle events.



## Sweep-Line Algorithm: Event-Point Schedule and Sweep-Line Status

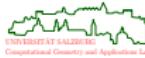
- All input points are stored in sorted order (according to  $y$ -coordinates) in the event-point schedule.
- Whenever three parabolic arcs become consecutive for the first time — when a site event occurs — the  $y$ -coordinate of the corresponding circle event is inserted into the event-point schedule at the appropriate place.
- Parabolic arcs have to be inserted into the beach line when processing site events, and have to be deleted when processing circle events.
- Both structures are best represented as balanced binary search trees, since this allows logarithmic insertion/deletion.



# Sweep-Line Algorithm: Analysis

## Lemma 65

The beach line is monotone with respect to the  $x$ -axis.



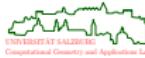
# Sweep-Line Algorithm: Analysis

## Lemma 65

The beach line is monotone with respect to the  $x$ -axis.

## Lemma 66

An arc can appear on the beach line only through a site event.



# Sweep-Line Algorithm: Analysis

## Lemma 65

The beach line is monotone with respect to the  $x$ -axis.

## Lemma 66

An arc can appear on the beach line only through a site event.

## Corollary 67

The beach line is a sequence of at most  $2n - 1$  parabolic arcs.



# Sweep-Line Algorithm: Analysis

## Lemma 65

The beach line is monotone with respect to the  $x$ -axis.

## Lemma 66

An arc can appear on the beach line only through a site event.

## Corollary 67

The beach line is a sequence of at most  $2n - 1$  parabolic arcs.

## Lemma 68

An arc can disappear from the beach line only through a circle event.



# Sweep-Line Algorithm: Analysis

## Lemma 65

The beach line is monotone with respect to the  $x$ -axis.

## Lemma 66

An arc can appear on the beach line only through a site event.

## Corollary 67

The beach line is a sequence of at most  $2n - 1$  parabolic arcs.

## Lemma 68

An arc can disappear from the beach line only through a circle event.

## Theorem 69 (Fortune 1986,1987)

The sweep-line algorithm computes the Voronoi diagram of  $n$  points in  $O(n \log n)$  time, using  $O(n)$  storage.



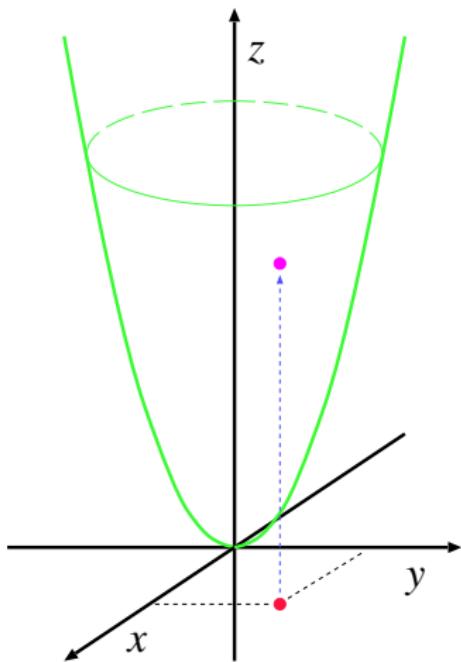
## Construction via Lifting to 3D

- Consider the transformation that maps a point  $p = (p_x, p_y)$  to the non-vertical plane  $h(p) \equiv z = 2p_x x + 2p_y y - (p_x^2 + p_y^2)$  in  $\mathbb{R}^3$ .



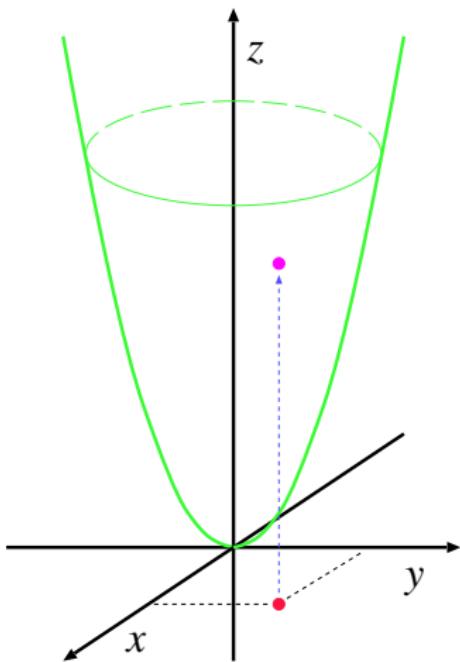
## Construction via Lifting to 3D

- Consider the transformation that maps a point  $p = (p_x, p_y)$  to the non-vertical plane  $h(p) \equiv z = 2p_x x + 2p_y y - (p_x^2 + p_y^2)$  in  $\mathbb{R}^3$ .
- This plane is tangent to the unit paraboloid  $z = x^2 + y^2$  at the point  $(p_x, p_y, p_x^2 + p_y^2)$ .



## Construction via Lifting to 3D

- Consider the transformation that maps a point  $p = (p_x, p_y)$  to the non-vertical plane  $h(p) \equiv z = 2p_x x + 2p_y y - (p_x^2 + p_y^2)$  in  $\mathbb{R}^3$ .
- This plane is tangent to the unit paraboloid  $z = x^2 + y^2$  at the point  $(p_x, p_y, p_x^2 + p_y^2)$ .
- Let  $h^+(p)$  be the half-space defined by  $h(p)$  which contains the unit paraboloid.

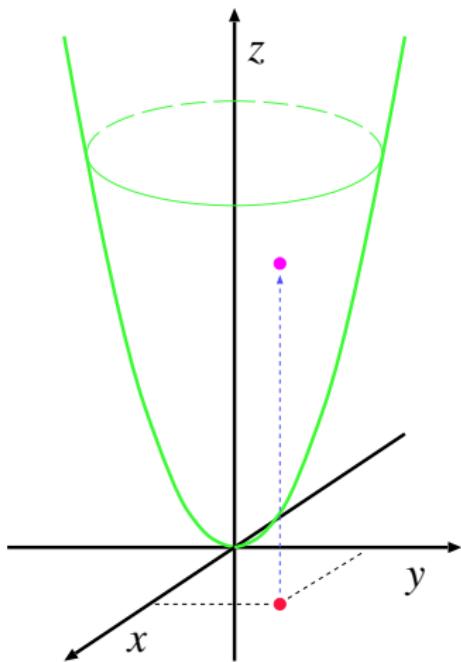


## Construction via Lifting to 3D

- Consider the transformation that maps a point  $p = (p_x, p_y)$  to the non-vertical plane  $h(p) \equiv z = 2p_x x + 2p_y y - (p_x^2 + p_y^2)$  in  $\mathbb{R}^3$ .
- This plane is tangent to the unit paraboloid  $z = x^2 + y^2$  at the point  $(p_x, p_y, p_x^2 + p_y^2)$ .
- Let  $h^+(p)$  be the half-space defined by  $h(p)$  which contains the unit paraboloid.

### Lemma 70

For  $S := \{p_1, p_2, \dots, p_n\}$ , consider the convex polyhedron  $\mathcal{P} := \cap_{1 \leq i \leq n} h^+(p_i)$ . The normal projection of the vertices and edges of  $\mathcal{P}$  onto the  $xy$ -plane yields  $\mathcal{VD}(S)$ .



## Construction via Lifting to 3D

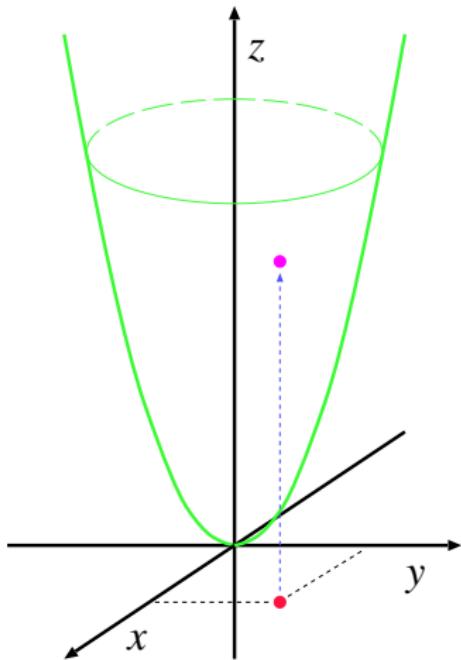
- Consider the transformation that maps a point  $p = (p_x, p_y)$  to the non-vertical plane  $h(p) \equiv z = 2p_x x + 2p_y y - (p_x^2 + p_y^2)$  in  $\mathbb{R}^3$ .
- This plane is tangent to the unit paraboloid  $z = x^2 + y^2$  at the point  $(p_x, p_y, p_x^2 + p_y^2)$ .
- Let  $h^+(p)$  be the half-space defined by  $h(p)$  which contains the unit paraboloid.

### Lemma 70

For  $S := \{p_1, p_2, \dots, p_n\}$ , consider the convex polyhedron  $\mathcal{P} := \cap_{1 \leq i \leq n} h^+(p_i)$ . The normal projection of the vertices and edges of  $\mathcal{P}$  onto the  $xy$ -plane yields  $\mathcal{VD}(S)$ .

### Corollary 71

This lifting allows to construct Voronoi diagrams in  $O(n \log n)$  time.



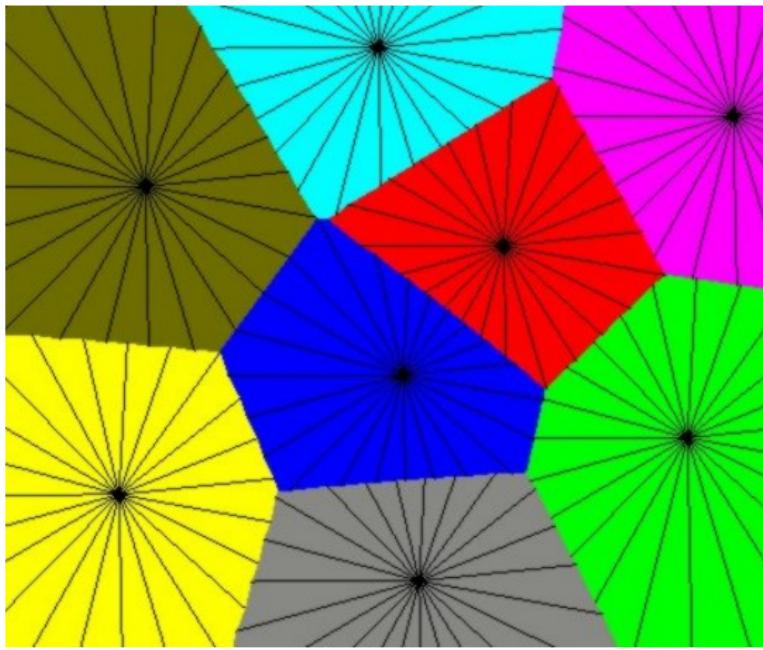
## Approximate Voronoi Diagram by Means of Graphics Hardware

- Regard  $\mathbb{R}^2$  as the  $xy$ -plane of  $\mathbb{R}^3$ , and construct upright circular unit cones at every point of  $S$ . (All cones point upwards, are of the same size and form the same angle with the  $xy$ -plane!) Assign a unique color to every cone.



## Approximate Voronoi Diagram by Means of Graphics Hardware

- Look at the cones from below the  $xy$ -plane, and use normal projection to project them on the  $xy$ -plane. This yields a colored subdivision of the  $xy$ -plane, i.e., of  $\mathbb{R}^2$ , where each cell corresponds to a Voronoi region of a point of  $S$ .



## Generalized Voronoi Diagrams

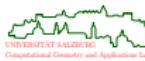
- Definitions and Properties
  - Definitions
  - Properties
- Algorithms
  - Algorithmic State-of-the-Art
  - Randomized Incremental Construction
- Applications of Voronoi Diagrams
  - Statistical Classification
  - Euclidean Minimum Spanning Tree
  - Euclidean Traveling Salesman Tour
  - Natural-Neighbor Interpolation
  - Maximum Empty/Inscribed Circle
  - Offsetting/Buffering
  - Generation of Tool Paths
  - Finding a Gouge-Free Path
  - Topologically Consistent Watermarking
  - Analysis of Drainage Displacements
  - Voronoi Diagrams in Nature

## Generalized Voronoi Diagrams

- Definitions and Properties
  - Definitions
  - Properties
- Algorithms
- Applications of Voronoi Diagrams

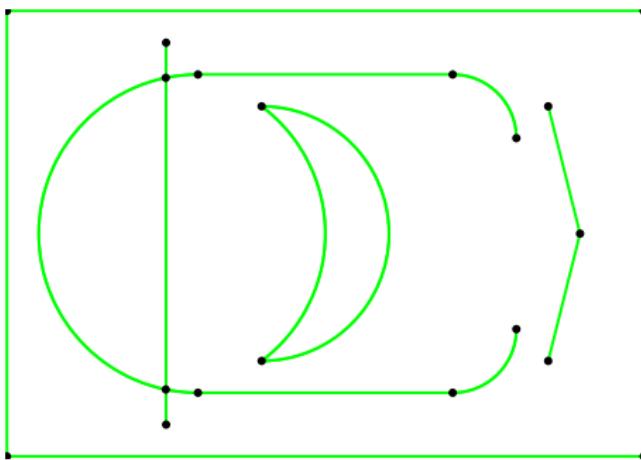
## Generalized Voronoi Diagram

- The definition of a Voronoi region allows **generalizations** in three different directions.



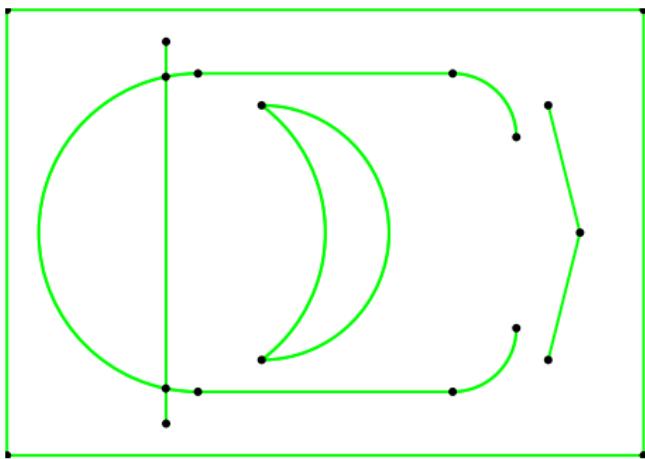
## Generalized Voronoi Diagram

- We consider a set  $S$  of  $n$  “sites” (points, straight-line segments, and circular arcs).



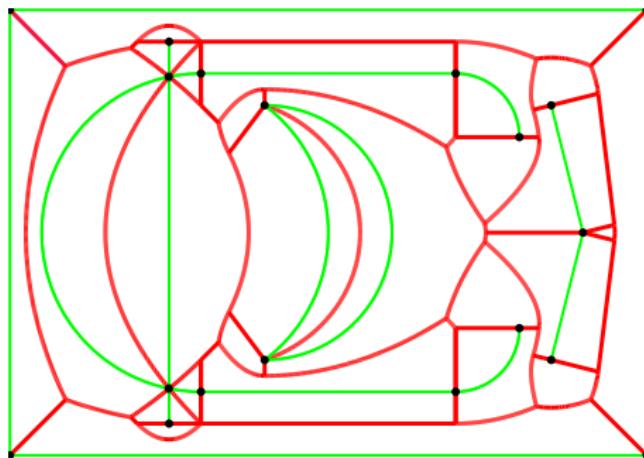
## Generalized Voronoi Diagram

- We consider a set  $S$  of  $n$  “sites” (points, straight-line segments, and circular arcs).
- For technical reasons we assume that all end-points of all segments and arcs are members of  $S$ . Furthermore, the segments and arcs are allowed to intersect only at common end-points. Such a set of sites is called “admissible”.



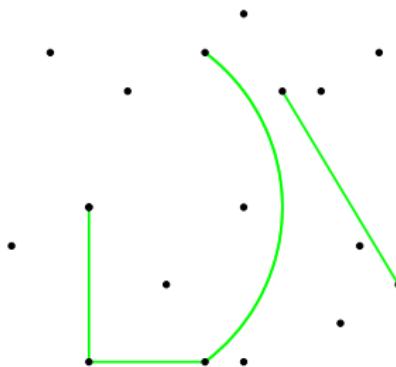
## Generalized Voronoi Diagram

- We consider a set  $S$  of  $n$  “sites” (points, straight-line segments, and circular arcs).
- For technical reasons we assume that all end-points of all segments and arcs are members of  $S$ . Furthermore, the segments and arcs are allowed to intersect only at common end-points. Such a set of sites is called “admissible”.
- Intuitively, the Voronoi diagram of  $S$  partitions the Euclidean plane into regions that are closer to one site than to any other.



# Generalized Voronoi Diagram

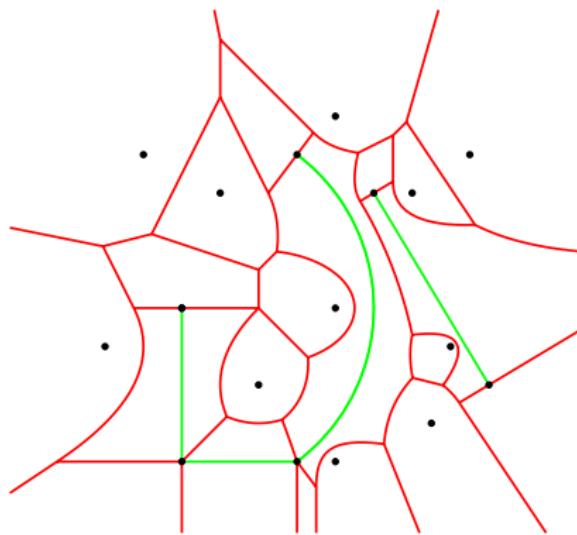
- Problem studied:
  - Given: Set  $S$  of points, line segments and circular arcs as input sites in 2D;



# Generalized Voronoi Diagram

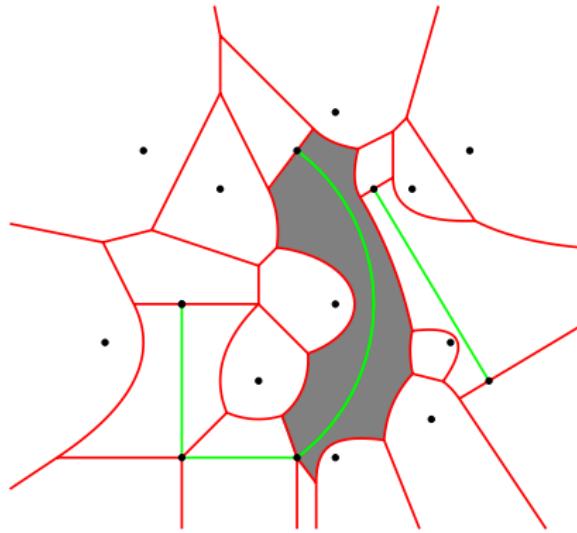
- Problem studied:

- Given: Set  $S$  of points, line segments and circular arcs as input sites in 2D;
- Sought: Voronoi diagram  $\mathcal{VD}(S)$  of the sites under the Euclidean distance  $d(\cdot, \cdot)$ .



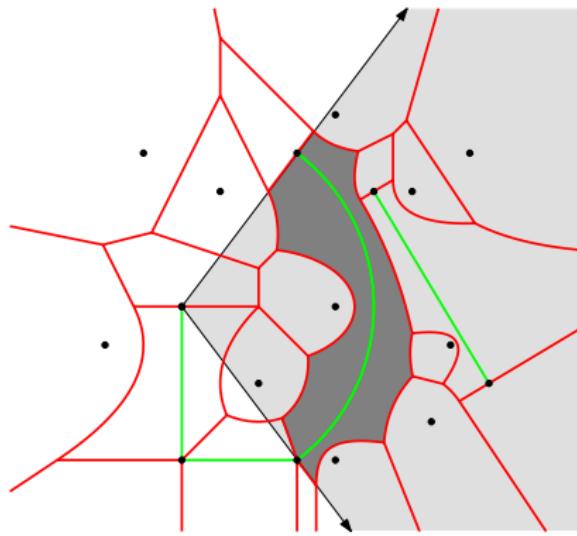
# Generalized Voronoi Diagram

- Problem studied:
  - Given: Set  $S$  of points, line segments and circular arcs as input sites in 2D;
  - Sought: Voronoi diagram  $\mathcal{VD}(S)$  of the sites under the Euclidean distance  $d(\cdot, \cdot)$ .
- Natural generalization of VDs of points, but Voronoi regions are now bounded by conics and need not be convex.



# Generalized Voronoi Diagram

- Problem studied:
  - Given: Set  $S$  of points, line segments and circular arcs as input sites in 2D;
  - Sought: Voronoi diagram  $\mathcal{VD}(S)$  of the sites under the Euclidean distance  $d(\cdot, \cdot)$ .
- Technical problem: Voronoi region of segment or arc  $s$  is restricted to strip resp. cone perpendicular to it — cone of influence  $\mathcal{CI}(s)$ .



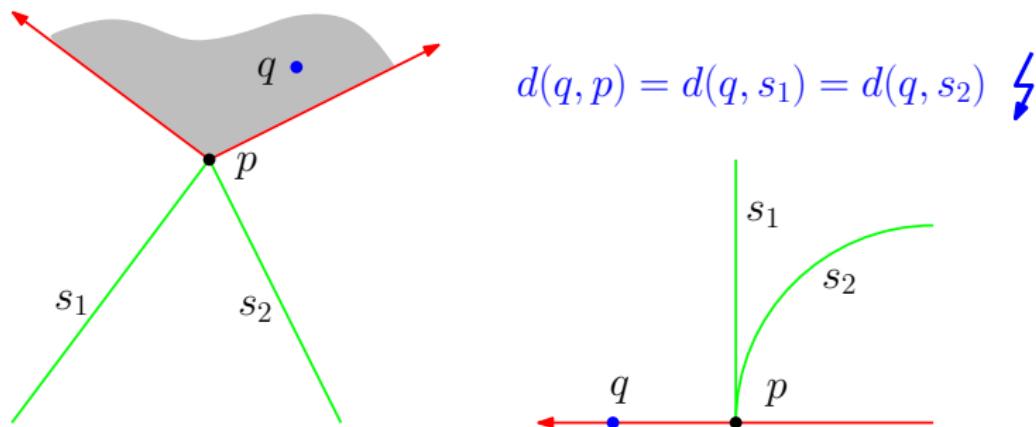
## Generalized Voronoi Diagram: Technical Problem

- Consider an admissible set  $S$  of  $n$  points, line segments and circular arcs as input sites in 2D, and two sites  $s_1, s_2 \in S$ .



## Generalized Voronoi Diagram: Technical Problem

- Consider an admissible set  $S$  of  $n$  points, line segments and circular arcs as input sites in 2D, and two sites  $s_1, s_2 \in S$ .
- Problem: We need to avoid “two-dimensional” bisectors.

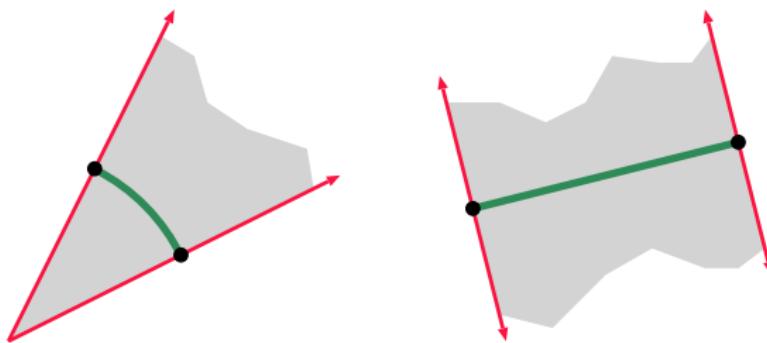


# Generalized Voronoi Diagram: Definitions

## Definition 72 (Cone of influence)

The *cone of influence*,  $\mathcal{CI}(s)$ , of

- a circular arc  $s$  is the closure of the cone bounded by the pair of rays originating in the arc's center and extending through its endpoints;

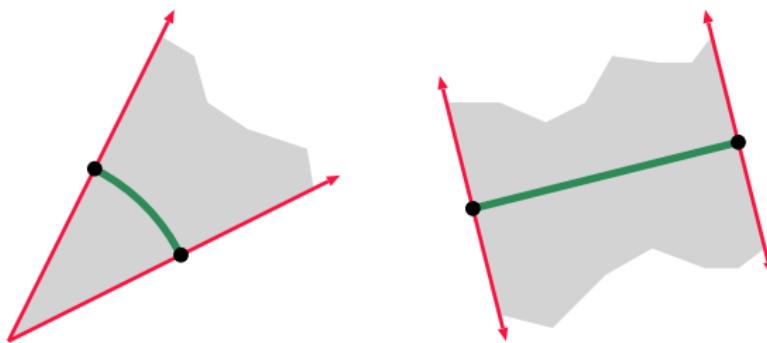


# Generalized Voronoi Diagram: Definitions

## Definition 72 (Cone of influence)

The *cone of influence*,  $\mathcal{CI}(s)$ , of

- a circular arc  $s$  is the closure of the cone bounded by the pair of rays originating in the arc's center and extending through its endpoints;
- a straight-line segment  $s$  is the closure of the strip bounded by the normals through its endpoints;

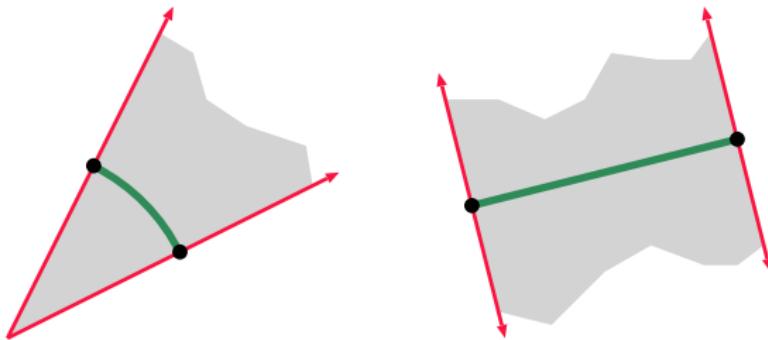


# Generalized Voronoi Diagram: Definitions

## Definition 72 (Cone of influence)

The *cone of influence*,  $\mathcal{CI}(s)$ , of

- a circular arc  $s$  is the closure of the cone bounded by the pair of rays originating in the arc's center and extending through its endpoints;
- a straight-line segment  $s$  is the closure of the strip bounded by the normals through its endpoints;
- a point  $s$  is the entire plane.



## Generalized Voronoi Diagram: Definitions

### Definition 73 ((Generalized) Voronoi region)

The (*generalized*) Voronoi region of  $s_i \in S$  relative to  $S$  is defined as

$$\mathcal{VR}(s_i, S) := \text{cl}\{q \in \text{int } \mathcal{CI}(s_i) : d(q, s_i) \leq d(q, S)\}.$$

## Generalized Voronoi Diagram: Definitions

### Definition 73 ((Generalized) Voronoi region)

The (*generalized*) Voronoi region of  $s_i \in S$  relative to  $S$  is defined as

$$\mathcal{VR}(s_i, S) := \text{cl}\{q \in \text{int } \mathcal{CI}(s_i) : d(q, s_i) \leq d(q, S)\}.$$

- It is common to drop the attribute “generalized” if the meaning is clear.



# Generalized Voronoi Diagram: Definitions

## Definition 73 ((Generalized) Voronoi region)

The (*generalized*) Voronoi region of  $s_i \in S$  relative to  $S$  is defined as

$$\mathcal{VR}(s_i, S) := \text{cl}\{q \in \text{int } \mathcal{CI}(s_i) : d(q, s_i) \leq d(q, S)\}.$$

- It is common to drop the attribute “generalized” if the meaning is clear.

## Definition 74 ((Generalized) Voronoi polygon)

The (*generalized*) Voronoi polygon of  $s_i \in S$  relative to  $S$  is defined as

$$\mathcal{VP}(s_i, S) := \partial \mathcal{VR}(s_i, S).$$



## Generalized Voronoi Diagram: Definitions

### Definition 73 ((Generalized) Voronoi region)

The (*generalized*) Voronoi region of  $s_i \in S$  relative to  $S$  is defined as

$$\mathcal{VR}(s_i, S) := \text{cl}\{q \in \text{int } \mathcal{CI}(s_i) : d(q, s_i) \leq d(q, S)\}.$$

- It is common to drop the attribute “generalized” if the meaning is clear.

### Definition 74 ((Generalized) Voronoi polygon)

The (*generalized*) Voronoi polygon of  $s_i \in S$  relative to  $S$  is defined as

$$\mathcal{VP}(s_i, S) := \partial \mathcal{VR}(s_i, S).$$

### Definition 75 ((Generalized) Voronoi diagram)

The (*generalized*) Voronoi diagram of  $S$  is defined as

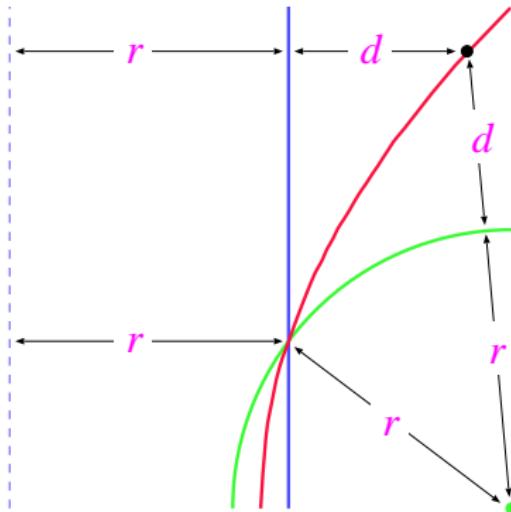
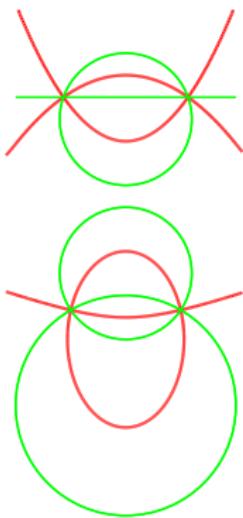
$$\mathcal{VD}(S) := \bigcup_{1 \leq i \leq n} \mathcal{VP}(s_i, S).$$



# Generalized Voronoi Diagram: Bisectors

## Lemma 76

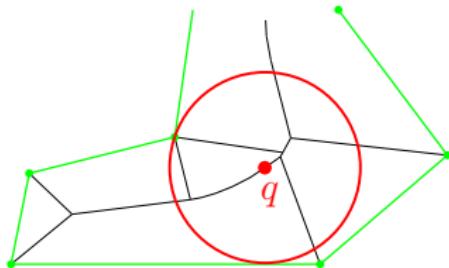
The structure  $\mathcal{VD}(S)$  is a planar graph and consists of  $O(n)$  parabolic, hyperbolic, elliptic and straight-line edges.



# Generalized Voronoi Diagram: Medial Axis

## Definition 77 (Clearance)

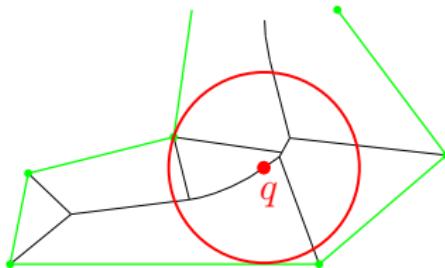
The *clearance* of a point  $q$  relative to  $S$  is the radius of the largest disk (*clearance disk*) centered at  $q$  which does not contain any site of  $S$  in its interior.



# Generalized Voronoi Diagram: Medial Axis

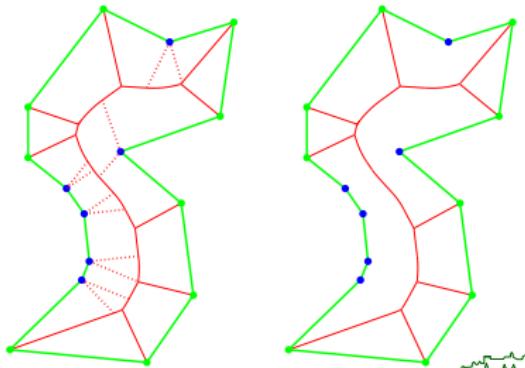
## Definition 77 (Clearance)

The *clearance* of a point  $q$  relative to  $S$  is the radius of the largest disk (*clearance disk*) centered at  $q$  which does not contain any site of  $S$  in its interior.



## Definition 78 (Medial axis)

A point in the interior of a (multiply-connected) polygonal area belongs to the *medial axis* (MA) of the area if and only if its clearance disk touches the boundary in at least two disjoint points.



## Generalized Voronoi Diagrams

- Definitions and Properties
- Algorithms
  - Algorithmic State-of-the-Art
  - Randomized Incremental Construction
- Applications of Voronoi Diagrams

# Generalized Voronoi Diagram: Algorithms

## Theorem 79 (Fortune 1987)

The Voronoi diagram of  $n$  points and straight-line segments can be constructed in  $O(n \log n)$  time by means of a sweep-line algorithm.



# Generalized Voronoi Diagram: Algorithms

## Theorem 79 (Fortune 1987)

The Voronoi diagram of  $n$  points and straight-line segments can be constructed in  $O(n \log n)$  time by means of a sweep-line algorithm.

## Theorem 80 (Yap 1987)

The Voronoi diagram of  $n$  points, straight-line segments and circular arcs can be constructed in  $O(n \log n)$  time by means of a divide&conquer algorithm.



# Generalized Voronoi Diagram: Algorithms

## Theorem 79 (Fortune 1987)

The Voronoi diagram of  $n$  points and straight-line segments can be constructed in  $O(n \log n)$  time by means of a sweep-line algorithm.

## Theorem 80 (Yap 1987)

The Voronoi diagram of  $n$  points, straight-line segments and circular arcs can be constructed in  $O(n \log n)$  time by means of a divide&conquer algorithm.

## Theorem 81 (Aichholzer et alii 2009)

The Voronoi diagram of  $n$  points, straight-line segments and circular arcs can be constructed in  $O(n \log^2 n)$  expected time by means of randomization combined with a divide&conquer algorithm.



## Generalized Voronoi Diagram: Algorithms

### Theorem 79 (Fortune 1987)

The Voronoi diagram of  $n$  points and straight-line segments can be constructed in  $O(n \log n)$  time by means of a sweep-line algorithm.

### Theorem 80 (Yap 1987)

The Voronoi diagram of  $n$  points, straight-line segments and circular arcs can be constructed in  $O(n \log n)$  time by means of a divide&conquer algorithm.

### Theorem 81 (Aichholzer et alii 2009)

The Voronoi diagram of  $n$  points, straight-line segments and circular arcs can be constructed in  $O(n \log^2 n)$  expected time by means of randomization combined with a divide&conquer algorithm.

### Theorem 82 (Held&Huber 2009, based on Held 2001)

The Voronoi diagram of  $n$  points, straight-line segments and circular arcs can be constructed in  $O(n \log n)$  expected time by means of randomized incremental construction.



## Generalized Voronoi Diagram: Algorithms

- Several other  $O(n \log n)$  expected-time algorithms for polygons and/or line segments ...
- What about Voronoi diagrams of polygons? Can one achieve  $o(n \log n)$ ?



## Generalized Voronoi Diagram: Algorithms

- Several other  $O(n \log n)$  expected-time algorithms for polygons and/or line segments ...
- What about Voronoi diagrams of polygons? Can one achieve  $o(n \log n)$ ?

### Theorem 83 (Aggarwal et alii 1989)

The Voronoi diagram of a convex polygon can be constructed in linear time.

## Generalized Voronoi Diagram: Algorithms

- Several other  $O(n \log n)$  expected-time algorithms for polygons and/or line segments ...
- What about Voronoi diagrams of polygons? Can one achieve  $o(n \log n)$ ?

### Theorem 83 (Aggarwal et alii 1989)

The Voronoi diagram of a convex polygon can be constructed in linear time.

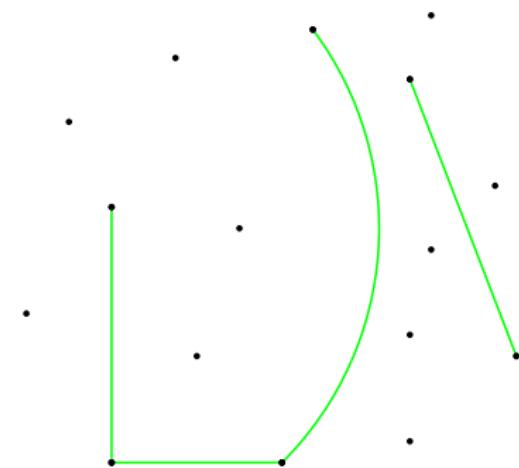
### Theorem 84 (Chin et alii 1999)

The Voronoi diagram of a simple polygon can be constructed in linear time.



# Randomized Incremental Construction

- How can we construct the (generalized) Voronoi diagram of these sites?



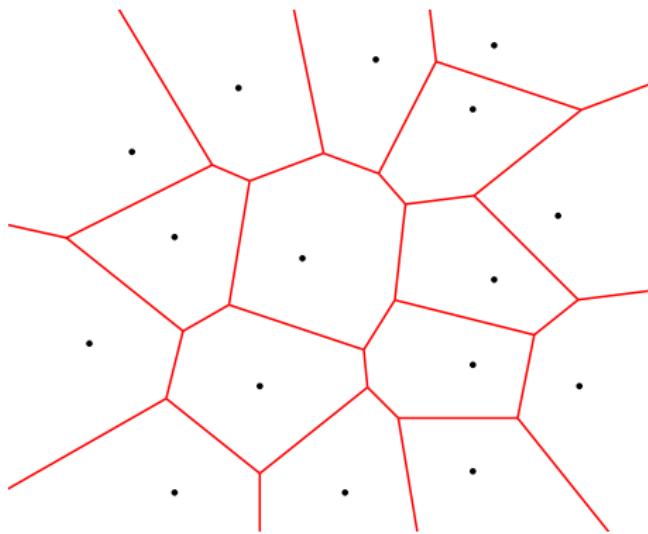
# Randomized Incremental Construction

- Start with the vertices of  $S$ .



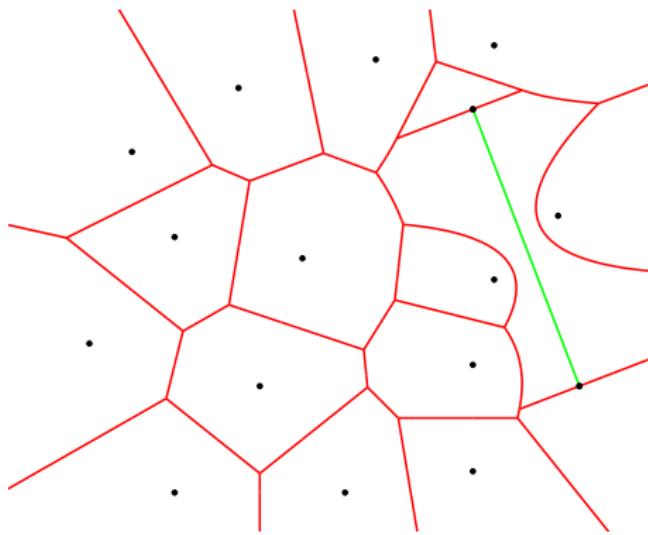
# Randomized Incremental Construction

- Start with the vertices of  $S$ , and compute their Voronoi diagram.



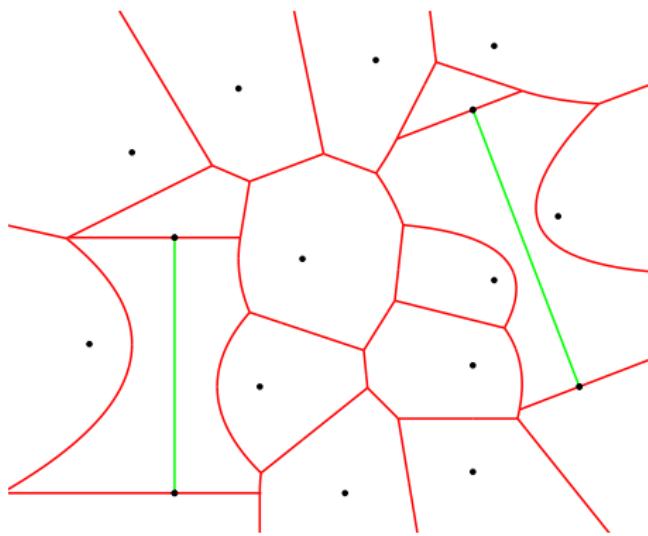
## Randomized Incremental Construction

- Start with the vertices of  $S$ , and compute their Voronoi diagram. Insert segments.



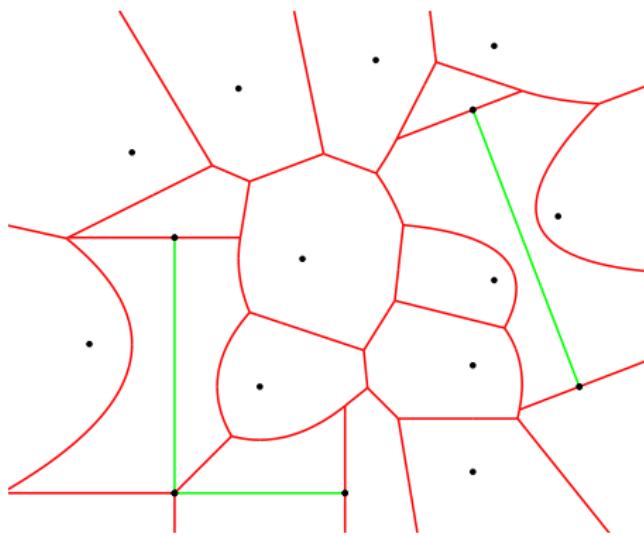
## Randomized Incremental Construction

- Start with the vertices of  $S$ , and compute their Voronoi diagram. Insert segments, randomly.



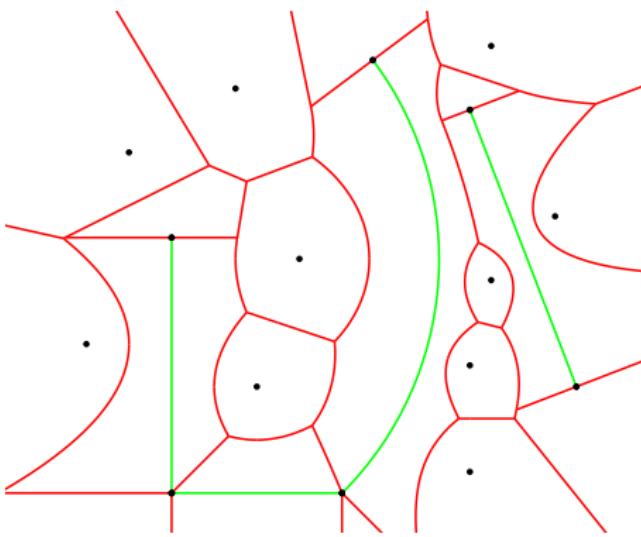
## Randomized Incremental Construction

- Start with the vertices of  $S$ , and compute their Voronoi diagram. Insert segments, randomly, one after the other.



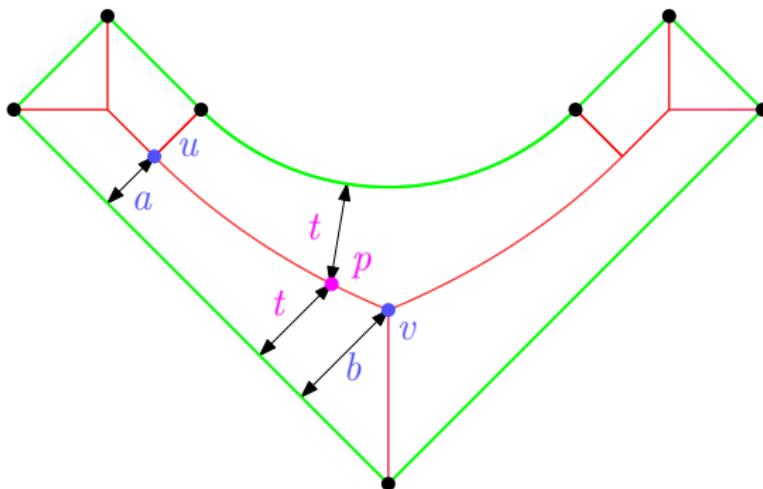
## Randomized Incremental Construction

- Start with the vertices of  $S$ , and compute their Voronoi diagram. Insert segments, randomly, one after the other. Same for the arcs.



## Generalized Voronoi Diagram: Parameterization of Voronoi Edges

- We assign a clearance-based parameterization  $f : [a, b] \rightarrow \mathbb{R}^2$  to every edge  $e$ , where  $a$  is the minimum and  $b$  is the maximum clearance of points of  $e$ .
- The coordinates of a point  $p$  of  $e$  with clearance  $t$  are obtained by evaluating  $f$ : we have  $p = f(t)$ .



## Generalized Voronoi Diagrams

- Definitions and Properties
- Algorithms
- Applications of Voronoi Diagrams
  - Statistical Classification
  - Euclidean Minimum Spanning Tree
  - Euclidean Traveling Salesman Tour
  - Natural-Neighbor Interpolation
  - Maximum Empty/Inscribed Circle
  - Offsetting/Buffering
  - Generation of Tool Paths
  - Finding a Gouge-Free Path
  - Topologically Consistent Watermarking
  - Analysis of Drainage Displacements
  - Voronoi Diagrams in Nature

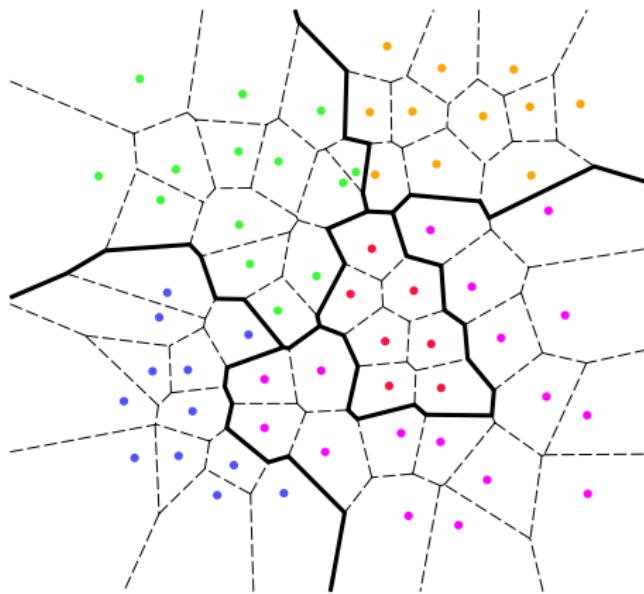
# Statistical Classification and Shape Estimation

- Given are sets of differently colored points in the plane. What is a suitable partition of the plane according to the colors of the points?



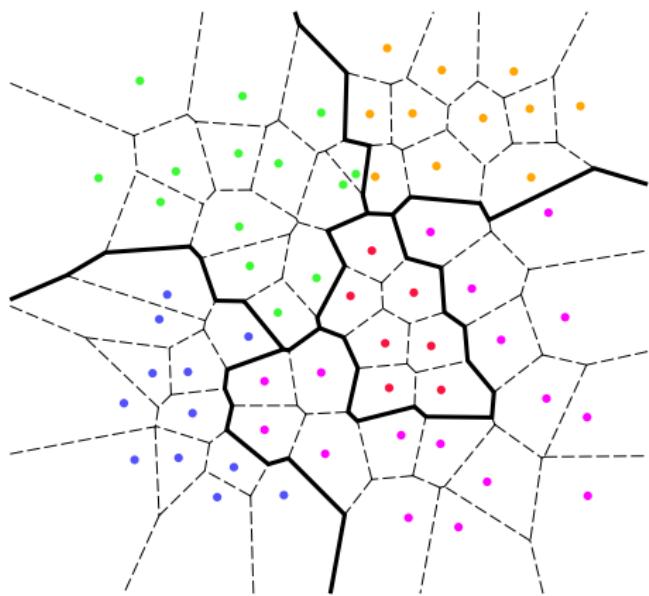
# Statistical Classification and Shape Estimation

- Given are sets of differently colored points in the plane. What is a suitable partition of the plane according to the colors of the points?
- Well-known idea: Compute the Voronoi diagram and color every Voronoi region with its point's color.



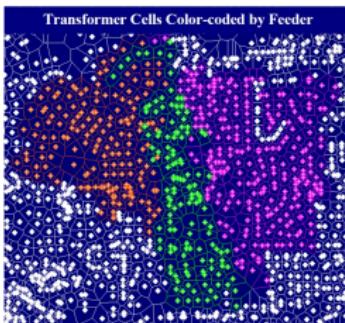
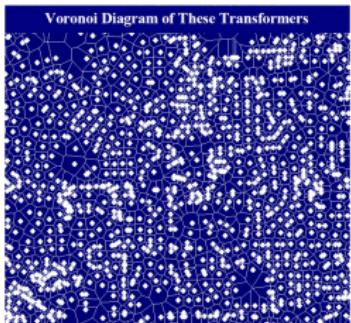
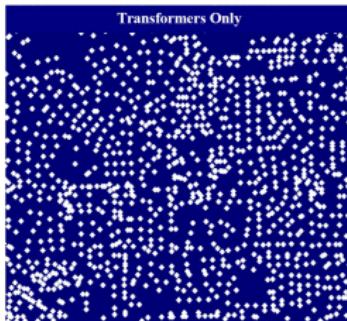
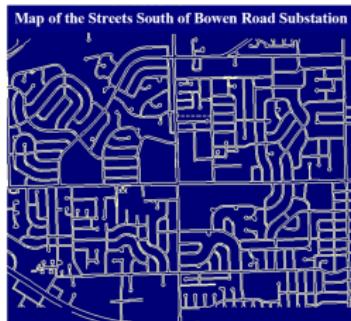
# Estimating Electrical Distribution Boundaries

- TXU Energy (Dallas, TX, USA):
  - Which area is serviced by a particular electric device?
  - How can we display (feeder-level) statistical information within a geographic context?



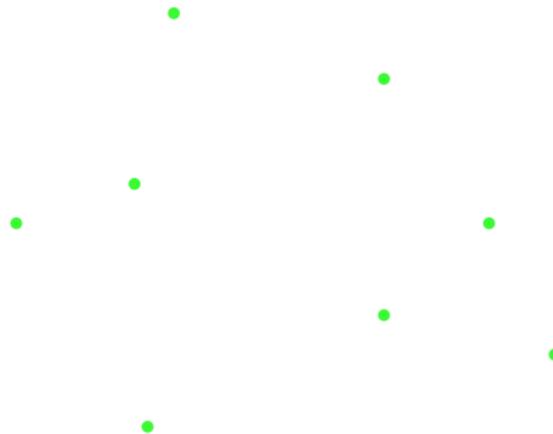
# Estimating Electrical Distribution Boundaries

- [Held&Williamson 2004] generate distribution boundaries as boundaries of unions of Voronoi regions of basic devices (e.g., transformers) and integrate them into TXU's geographic information system.



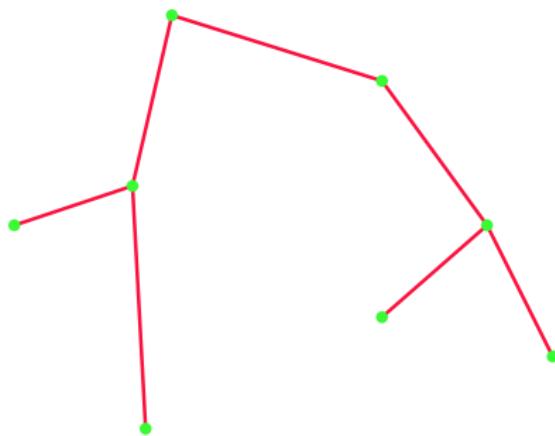
# Euclidean Minimum Spanning Tree

- Consider a set  $S := \{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^2$  of  $n$  points, and assume that we want to compute a Euclidean minimum spanning tree (EMST) of  $S$ .



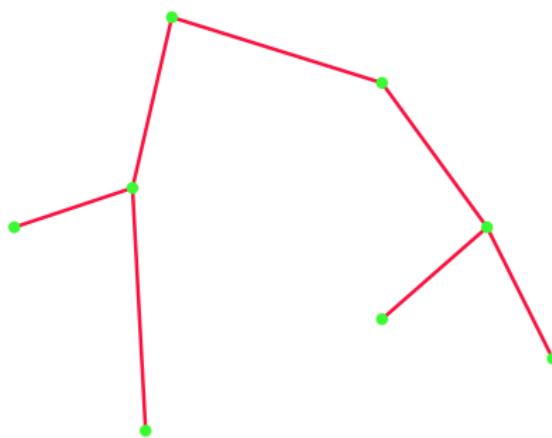
## Euclidean Minimum Spanning Tree

- Consider a set  $S := \{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^2$  of  $n$  points, and assume that we want to compute a Euclidean minimum spanning tree (EMST) of  $S$ .



## Euclidean Minimum Spanning Tree

- Consider a set  $S := \{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^2$  of  $n$  points, and assume that we want to compute a Euclidean minimum spanning tree (EMST) of  $S$ .



- Note: An EMST is unique if all inter-point distances on  $S$  are distinct.

## Euclidean Minimum Spanning Tree

- Obviously, we could apply standard techniques of graph theory by computing an EMST on the graph  $\mathcal{G} := (V, E)$ , where  $V := S$  and  $E := S \times S$ , and where the Euclidean length of an edge is taken as its weight.

# Euclidean Minimum Spanning Tree

- Obviously, we could apply standard techniques of graph theory by computing an EMST on the graph  $\mathcal{G} := (V, E)$ , where  $V := S$  and  $E := S \times S$ , and where the Euclidean length of an edge is taken as its weight.

## Lemma 85 (Jarnik 1930, Prim 1957, Dijkstra 1959)

Assume that  $\mathcal{G}$  is connected, and let  $V_1, V_2$  be a partition of  $V$ . There is a minimum spanning tree of  $\mathcal{G}$  which contains the shortest of the edges with one terminal in  $V_1$  and the other in  $V_2$ .

## Euclidean Minimum Spanning Tree

- Obviously, we could apply standard techniques of graph theory by computing an EMST on the graph  $\mathcal{G} := (V, E)$ , where  $V := S$  and  $E := S \times S$ , and where the Euclidean length of an edge is taken as its weight.

### Lemma 85 (Jarnik 1930, Prim 1957, Dijkstra 1959)

Assume that  $\mathcal{G}$  is connected, and let  $V_1, V_2$  be a partition of  $V$ . There is a minimum spanning tree of  $\mathcal{G}$  which contains the shortest of the edges with one terminal in  $V_1$  and the other in  $V_2$ .

- Prim's algorithm* starts with a small tree  $\mathcal{T}$  and grows it until it contains all nodes of  $\mathcal{G}$ . Initially,  $\mathcal{T}$  contains just one arbitrary node of  $V$ . At each stage one node not yet in  $\mathcal{T}$  but closest to (a node of)  $\mathcal{T}$  is added to  $\mathcal{T}$ . Prim's algorithm can be implemented to run in  $O(|V|^2)$  time.

## Euclidean Minimum Spanning Tree

- Obviously, we could apply standard techniques of graph theory by computing an EMST on the graph  $\mathcal{G} := (V, E)$ , where  $V := S$  and  $E := S \times S$ , and where the Euclidean length of an edge is taken as its weight.

### Lemma 85 (Jarnik 1930, Prim 1957, Dijkstra 1959)

Assume that  $\mathcal{G}$  is connected, and let  $V_1, V_2$  be a partition of  $V$ . There is a minimum spanning tree of  $\mathcal{G}$  which contains the shortest of the edges with one terminal in  $V_1$  and the other in  $V_2$ .

- Prim's algorithm* starts with a small tree  $\mathcal{T}$  and grows it until it contains all nodes of  $\mathcal{G}$ . Initially,  $\mathcal{T}$  contains just one arbitrary node of  $V$ . At each stage one node not yet in  $\mathcal{T}$  but closest to (a node of)  $\mathcal{T}$  is added to  $\mathcal{T}$ . Prim's algorithm can be implemented to run in  $O(|V|^2)$  time.
- Kruskal's algorithm* begins with a spanning forest, where each forest is initialized with one node of  $V$ . It repeatedly joins two trees together by picking the shortest edge between them until a spanning tree of the entire graph is obtained.  
Kruskal's algorithm can be implemented to run in  $O(|E| \log |E|)$  time.

## Euclidean Minimum Spanning Tree

- Obviously, we could apply standard techniques of graph theory by computing an EMST on the graph  $\mathcal{G} := (V, E)$ , where  $V := S$  and  $E := S \times S$ , and where the Euclidean length of an edge is taken as its weight.

### Lemma 85 (Jarnik 1930, Prim 1957, Dijkstra 1959)

Assume that  $\mathcal{G}$  is connected, and let  $V_1, V_2$  be a partition of  $V$ . There is a minimum spanning tree of  $\mathcal{G}$  which contains the shortest of the edges with one terminal in  $V_1$  and the other in  $V_2$ .

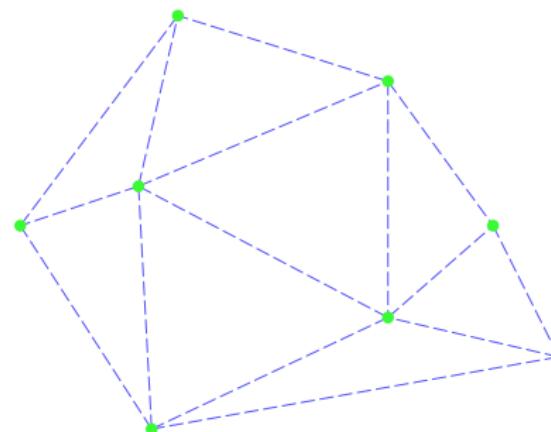
- Prim's algorithm* starts with a small tree  $\mathcal{T}$  and grows it until it contains all nodes of  $\mathcal{G}$ . Initially,  $\mathcal{T}$  contains just one arbitrary node of  $V$ . At each stage one node not yet in  $\mathcal{T}$  but closest to (a node of)  $\mathcal{T}$  is added to  $\mathcal{T}$ . Prim's algorithm can be implemented to run in  $O(|V|^2)$  time.
- Kruskal's algorithm* begins with a spanning forest, where each forest is initialized with one node of  $V$ . It repeatedly joins two trees together by picking the shortest edge between them until a spanning tree of the entire graph is obtained. Kruskal's algorithm can be implemented to run in  $O(|E| \log |E|)$  time.
- Can we do any better than  $O(n^2)$  when computing EMSTs?

# Euclidean Minimum Spanning Tree

- Can we do any better than  $O(n^2)$  when computing EMSTs?

## Lemma 86

The EMST of a set  $S$  of points is a sub-graph of  $\mathcal{DT}(S)$ .

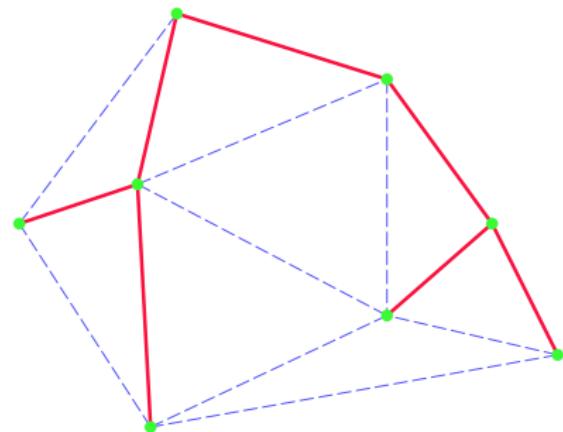


# Euclidean Minimum Spanning Tree

- Can we do any better than  $O(n^2)$  when computing EMSTs?

## Lemma 86

The EMST of a set  $S$  of points is a sub-graph of  $\mathcal{DT}(S)$ .



## Euclidean Minimum Spanning Tree

- Thus, there is no need to consider the full graph on  $S$ .
- Rather, we can apply Kruskal's algorithm to  $\mathcal{DT}(S)$ , and obtain an  $O(n \log n)$  algorithm for computing EMSTs.

# Euclidean Minimum Spanning Tree

- Thus, there is no need to consider the full graph on  $S$ .
- Rather, we can apply Kruskal's algorithm to  $\mathcal{DT}(S)$ , and obtain an  $O(n \log n)$  algorithm for computing EMSTs.

## Lemma 87

An EMST of  $S$  can be computed in time  $O(n \log n)$ .

# Euclidean Minimum Spanning Tree

- Thus, there is no need to consider the full graph on  $S$ .
- Rather, we can apply Kruskal's algorithm to  $\mathcal{DT}(S)$ , and obtain an  $O(n \log n)$  algorithm for computing EMSTs.

## Lemma 87

An EMST of  $S$  can be computed in time  $O(n \log n)$ .

## Theorem 88

An EMST of  $S$  can be computed from the Delaunay triangulation of  $S$  in time  $O(n)$ .

# Euclidean Minimum Spanning Tree

- Thus, there is no need to consider the full graph on  $S$ .
- Rather, we can apply Kruskal's algorithm to  $\mathcal{DT}(S)$ , and obtain an  $O(n \log n)$  algorithm for computing EMSTs.

## Lemma 87

An EMST of  $S$  can be computed in time  $O(n \log n)$ .

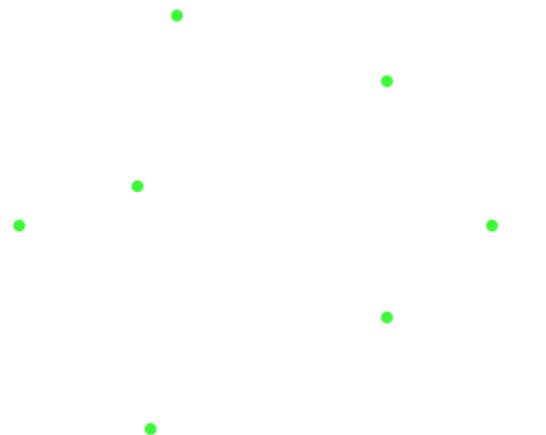
## Theorem 88

An EMST of  $S$  can be computed from the Delaunay triangulation of  $S$  in time  $O(n)$ .

*Proof:* Observe that  $\mathcal{DT}(S)$  is a planar graph, and use Cherdon and Tarjan's "clean-up refinement" of Kruskal's algorithm. □

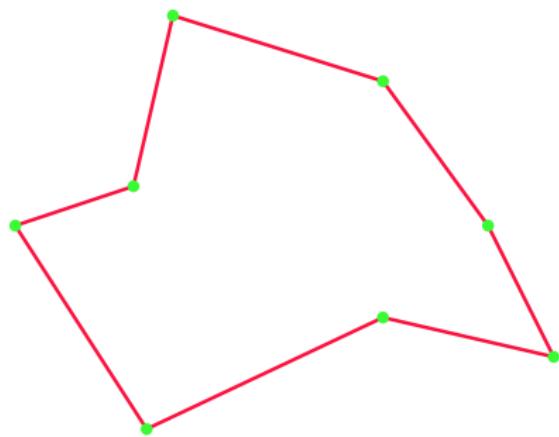
## Approximate Traveling Salesman Tour

- The EUCLIDEAN TRAVELING SALESMAN PROBLEM (ETSP) asks to compute a shortest closed path on  $S \subset \mathbb{R}^2$  that visits all points of  $S$ .



## Approximate Traveling Salesman Tour

- The EUCLIDEAN TRAVELING SALESMAN PROBLEM (ETSP) asks to compute a shortest closed path on  $S \subset \mathbb{R}^2$  that visits all points of  $S$ .



## Theorem 89

TSP is  $\mathcal{NP}$ -complete, and ETSP is  $\mathcal{NP}$ -hard.



## Theorem 89

TSP is  $\mathcal{NP}$ -complete, and ETSP is  $\mathcal{NP}$ -hard.

- Intuitively, ETSP ought to be  $\mathcal{NP}$ -complete, too.
- And, indeed, the  $\mathcal{NP}$ -completeness of ETSP is claimed in many publications . . .



# Approximate Traveling Salesman Tour

## Theorem 89

TSP is  $\mathcal{NP}$ -complete, and ETSP is  $\mathcal{NP}$ -hard.

- Intuitively, ETSP ought to be  $\mathcal{NP}$ -complete, too.
- And, indeed, the  $\mathcal{NP}$ -completeness of ETSP is claimed in many publications . . .
- However, this claim is wrong! (The title of [Papadimitriou 1977] is misleading!)  
ETSP, and several other optimization problems involving Euclidean distance, are not known to be in  $\mathcal{NP}$  due to a “technical twist”: For ETSP, the length of a tour on  $n$  points is a sum of  $n$  square roots. Comparing this sum to the number  $c$  may require very high precision, and no polynomial-time algorithm is known for solving this problem. (E.g., repeated squaring may lead to numbers that need  $2^n$  bits to store.)
- Open problem: Can the sum of  $n$  square roots of integers be compared to another integer in polynomial time?

## Approximate Traveling Salesman Tour

- Let  $OPT$  be the true length of a TSP tour, and let  $APX$  be the length of an approximate solution.

# Approximate Traveling Salesman Tour

- Let  $OPT$  be the true length of a TSP tour, and let  $APX$  be the length of an approximate solution.

## Definition 90 (Constant-factor approximation)

An approximation algorithm provides a *constant-factor approximation* for TSP if a constant  $c \in \mathbb{R}^+$  exists such that  $APX \leq c \cdot OPT$  holds for all inputs.

# Approximate Traveling Salesman Tour

- Let  $OPT$  be the true length of a TSP tour, and let  $APX$  be the length of an approximate solution.

## Definition 90 (Constant-factor approximation)

An approximation algorithm provides a *constant-factor approximation* for TSP if a constant  $c \in \mathbb{R}^+$  exists such that  $APX \leq c \cdot OPT$  holds for all inputs.

- Simple constant-factor approximations to ETSP:
  - Doubling-the-EMST heuristic:  $c = 2$ ; runs in  $O(n \log n)$  time.
  - Christofides' heuristic:  $c = 3/2$ ; runs in  $O(n^3)$  time.

## Approximate Traveling Salesman Tour

- Note that the Euclidean metric obeys the triangle inequality.
- Recent *polynomial-time approximation schemes* (PTAS):
  - Arora (1996), Mitchell (1996), Rao and Smith (1998).
  - $c = 1 + \varepsilon$  (for  $\varepsilon \in \mathbb{R}^+$ ).
  - For every fixed  $\varepsilon$ , a PTAS is required to run in time polynomial in  $n$ .
  - Common to these algorithms is the fact that  $O(1/\varepsilon)$  is allowed to appear as exponent of  $n$  or  $\log n$ .



## Approximate Traveling Salesman Tour

- Note that the Euclidean metric obeys the triangle inequality.
- Recent *polynomial-time approximation schemes* (PTAS):
  - Arora (1996), Mitchell (1996), Rao and Smith (1998).
  - $c = 1 + \varepsilon$  (for  $\varepsilon \in \mathbb{R}^+$ ).
  - For every fixed  $\varepsilon$ , a PTAS is required to run in time polynomial in  $n$ .
  - Common to these algorithms is the fact that  $O(1/\varepsilon)$  is allowed to appear as exponent of  $n$  or  $\log n$ .

### Theorem 91 (Arora 1996, Mitchell 1996, Rao&Smith 1998)

There exists a polynomial-time approximation scheme for solving ETSP.

# Approximate Traveling Salesman Tour

- Note that the Euclidean metric obeys the triangle inequality.
- Recent *polynomial-time approximation schemes* (PTAS):
  - Arora (1996), Mitchell (1996), Rao and Smith (1998).
  - $c = 1 + \varepsilon$  (for  $\varepsilon \in \mathbb{R}^+$ ).
  - For every fixed  $\varepsilon$ , a PTAS is required to run in time polynomial in  $n$ .
  - Common to these algorithms is the fact that  $O(1/\varepsilon)$  is allowed to appear as exponent of  $n$  or  $\log n$ .

## Theorem 91 (Arora 1996, Mitchell 1996, Rao&Smith 1998)

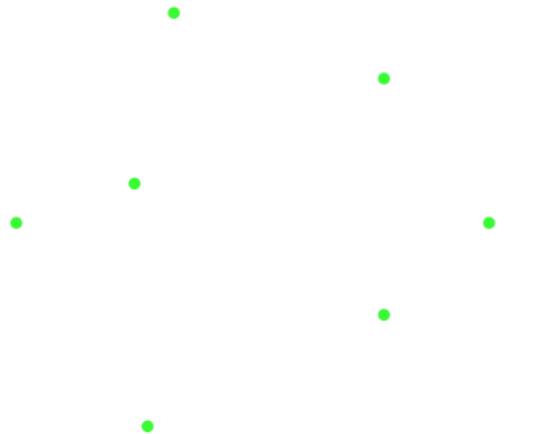
There exists a polynomial-time approximation scheme for solving ETSP.

## Theorem 92 (Papadimitriou&Vempala 2000)

For non-Euclidean TSPs with symmetric metric no polynomial-time constant-factor approximation algorithm exists which achieves  $c \leq (1 + 1/219)$ , unless  $\mathcal{P} = \mathcal{NP}$ .

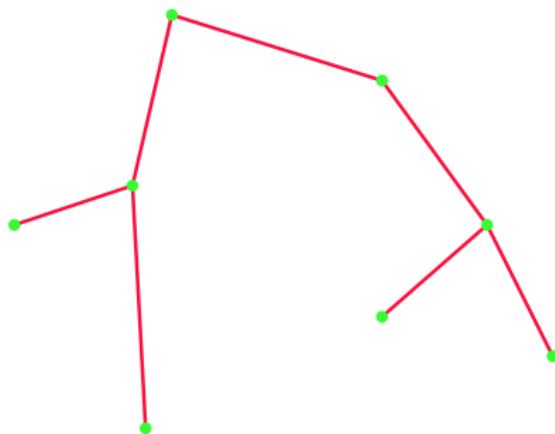


# Approximate TST: Doubling-the-EMST Heuristic



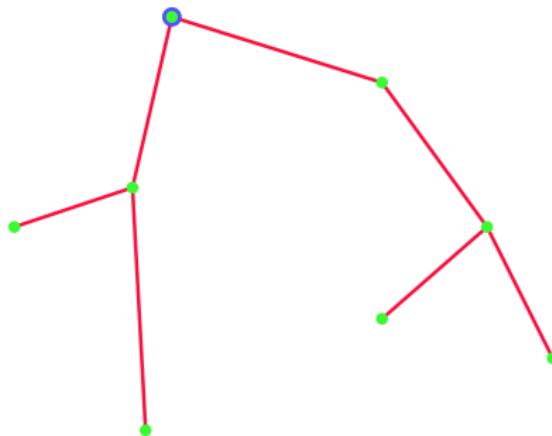
## Approximate TST: Doubling-the-EMST Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .



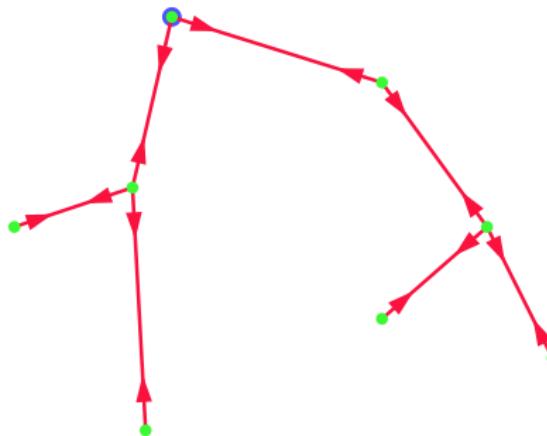
## Approximate TST: Doubling-the-EMST Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .
- ② Select an arbitrary node  $v$  of  $\mathcal{T}(S)$  as root.



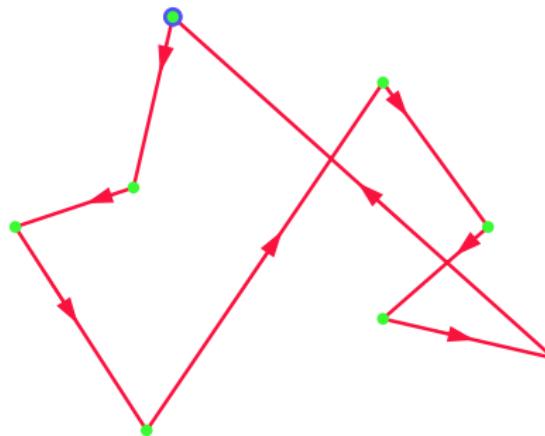
## Approximate TST: Doubling-the-EMST Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .
- ② Select an arbitrary node  $v$  of  $\mathcal{T}(S)$  as root.
- ③ Compute an in-order traversal of  $\mathcal{T}(S)$  rooted at  $v$  to obtain a tour  $\mathcal{C}(S)$ .



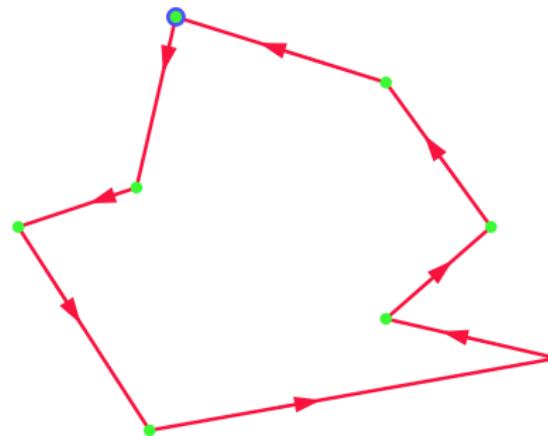
## Approximate TST: Doubling-the-EMST Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .
- ② Select an arbitrary node  $v$  of  $\mathcal{T}(S)$  as root.
- ③ Compute an in-order traversal of  $\mathcal{T}(S)$  rooted at  $v$  to obtain a tour  $\mathcal{C}(S)$ .
- ④ By-pass points already visited, thus shortening  $\mathcal{C}(S)$ .



## Approximate TST: Doubling-the-EMST Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .
- ② Select an arbitrary node  $v$  of  $\mathcal{T}(S)$  as root.
- ③ Compute an in-order traversal of  $\mathcal{T}(S)$  rooted at  $v$  to obtain a tour  $\mathcal{C}(S)$ .
- ④ By-pass points already visited, thus shortening  $\mathcal{C}(S)$ .
- ⑤ Apply 2-opt moves (at additional computational cost).



## Approximate TST: Doubling-the-EMST Heuristic

- Time complexity:  $O(n \log n)$  for computing the EMST  $\mathcal{T}(S)$ .
- Factor of approximation:  $c = 2$ .



## Approximate TST: Doubling-the-EMST Heuristic

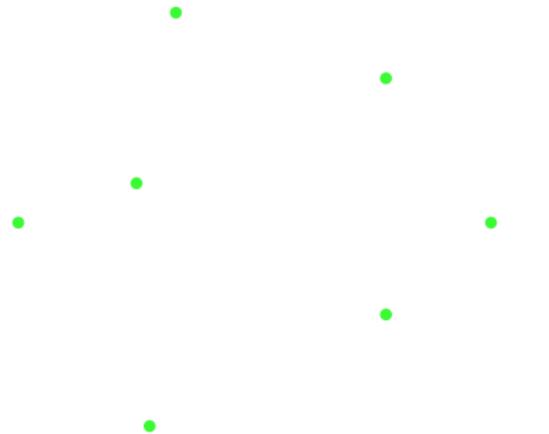
- Time complexity:  $O(n \log n)$  for computing the EMST  $\mathcal{T}(S)$ .
- Factor of approximation:  $c = 2$ .

### Theorem 93

The doubling-the-EMST heuristic computes a tour on  $n$  points within  $O(n \log n)$  time that is at most 100% longer than the shortest tour.

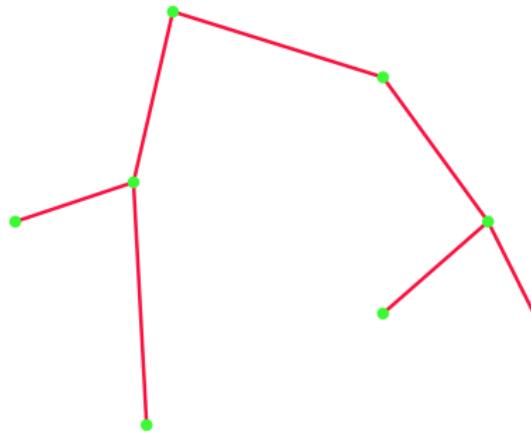


# Approximate TST: Christofides' Heuristic



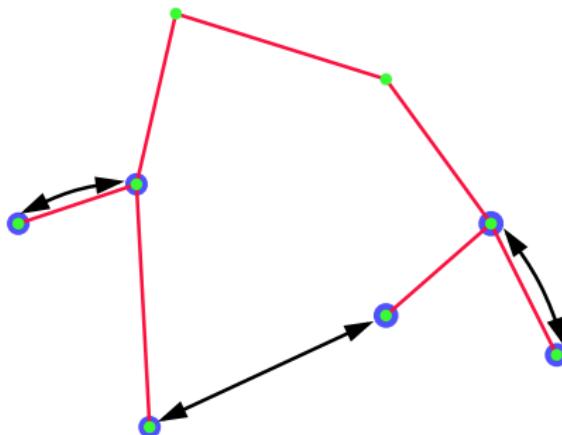
## Approximate TST: Christofides' Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .



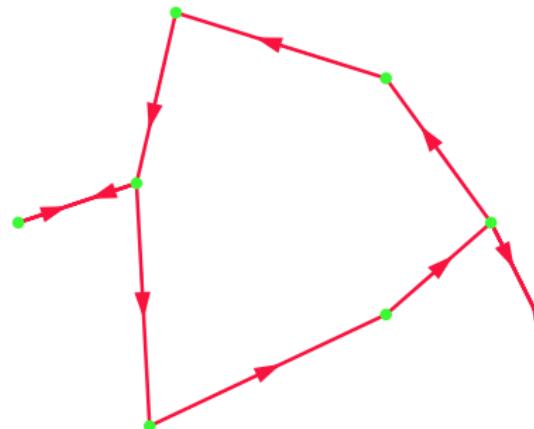
## Approximate TST: Christofides' Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .
- ② Get a minimum Euclidean matching  $\mathcal{M}$  on the vertices of odd degree in  $\mathcal{T}(S)$ .



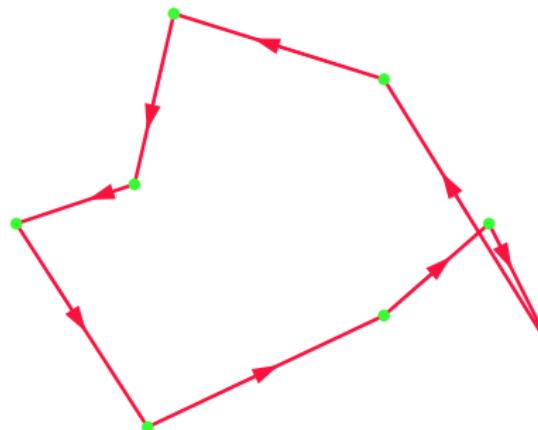
## Approximate TST: Christofides' Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .
- ② Get a minimum Euclidean matching  $\mathcal{M}$  on the vertices of odd degree in  $\mathcal{T}(S)$ .
- ③ Compute an Eulerian tour  $\mathcal{C}$  on  $\mathcal{T} \cup \mathcal{M}$ .



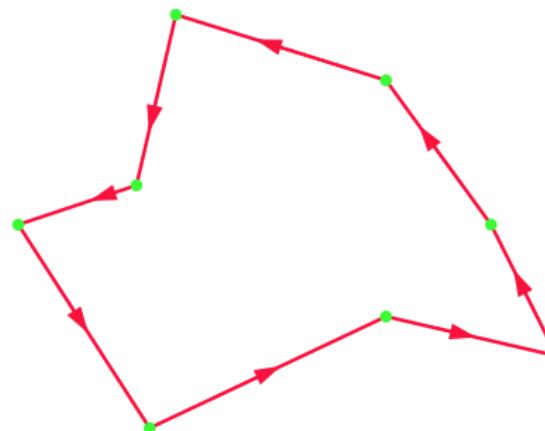
## Approximate TST: Christofides' Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .
- ② Get a minimum Euclidean matching  $\mathcal{M}$  on the vertices of odd degree in  $\mathcal{T}(S)$ .
- ③ Compute an Eulerian tour  $\mathcal{C}$  on  $\mathcal{T} \cup \mathcal{M}$ .
- ④ By-pass points already visited, thus shortening  $\mathcal{C}$ .



## Approximate TST: Christofides' Heuristic

- ① Compute the Euclidean minimum spanning tree  $\mathcal{T}(S)$  of  $S$ .
- ② Get a minimum Euclidean matching  $\mathcal{M}$  on the vertices of odd degree in  $\mathcal{T}(S)$ .
- ③ Compute an Eulerian tour  $\mathcal{C}$  on  $\mathcal{T} \cup \mathcal{M}$ .
- ④ By-pass points already visited, thus shortening  $\mathcal{C}$ .
- ⑤ Apply 2-opt moves (at additional computational cost).



## Approximate TST: Christofides' Heuristic

- Time complexity:  $O(n^3)$  for computing the Euclidean matching.
- Factor of approximation:  $c = \frac{3}{2}$ .

## Approximate TST: Christofides' Heuristic

- Time complexity:  $O(n^3)$  for computing the Euclidean matching.
- Factor of approximation:  $c = \frac{3}{2}$ .

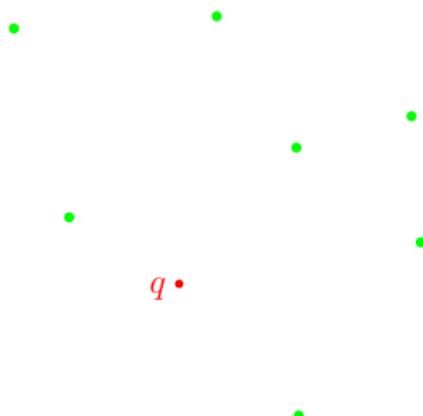
### Theorem 94

Christofides' heuristic computes a tour on  $n$  points within  $O(n^3)$  time that is at most 50% longer than the shortest tour.

# Natural-Neighbor Interpolation

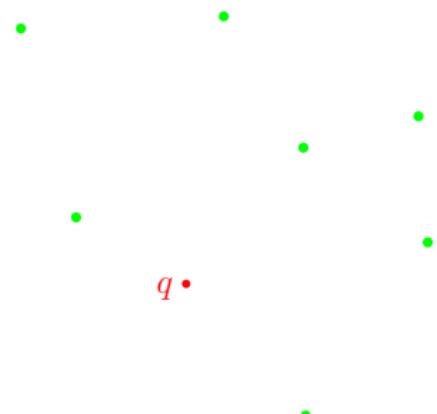
- Problem studied:

- Given:  $S$  of  $k$  sites  $p_1, p_2, \dots, p_k \in \mathbb{R}^2$  with associated (scalar or vector-valued) "data"  $v_1, v_2, \dots, v_k$ , and  $q \in CH(S)$ .



# Natural-Neighbor Interpolation

- Problem studied:
  - Given:  $S$  of  $k$  sites  $p_1, p_2, \dots, p_k \in \mathbb{R}^2$  with associated (scalar or vector-valued) “data”  $v_1, v_2, \dots, v_k$ , and  $q \in CH(S)$ .
  - Sought: An estimate  $f(q)$  of the data at  $q$ , obtained by interpolation of  $v_1, \dots, v_k$ .



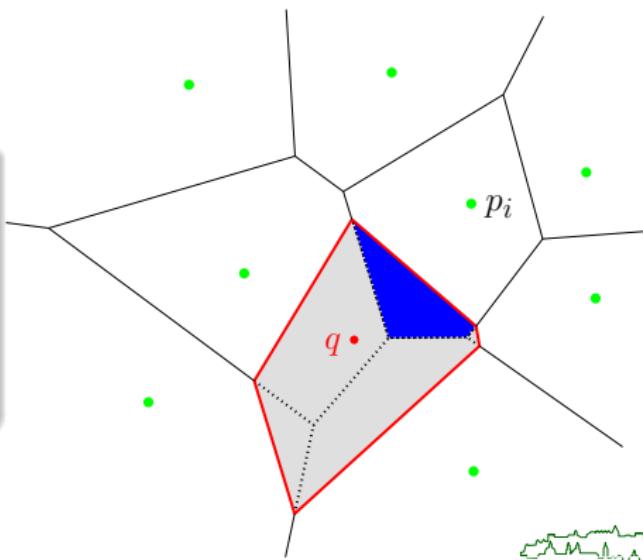
# Natural-Neighbor Interpolation

- Problem studied:
  - Given:  $S$  of  **$k$  sites**  $p_1, p_2, \dots, p_k \in \mathbb{R}^2$  with associated (scalar or vector-valued) “data”  $v_1, v_2, \dots, v_k$ , and  $q \in CH(S)$ .
  - Sought: An estimate  $f(q)$  of the data at  $q$ , obtained by interpolation of  $v_1, \dots, v_k$ .

## Natural-neighbor interpolation

[Sibson 1981]: Use ratios of Voronoi areas as weights  $\lambda_i(q)$  in the interpolation:

$$f(q) := \sum_{i=1}^k v_i \lambda_i(q).$$



# Natural-Neighbor Interpolation

## Definition 95 (NNI)

For a set  $S$  of  $k$  sites  $p_1, p_2, \dots, p_k \in \mathbb{R}^2$  with associated (scalar or vector-valued) “data”  $v_1, v_2, \dots, v_k$ , and  $q \in CH(S)$ ,

$$f(q) := \sum_{i=1}^k v_i \lambda_i(q)$$

gives the interpolated data for  $q$  obtained by natural-neighbor interpolation (NNI), where

$$\lambda_i(q) := \frac{|\mathcal{VR}(q, p_i, S')|}{|\mathcal{VR}(q, S')|} \text{ with } S' := S \cup \{q\}.$$

# Natural-Neighbor Interpolation

## Definition 95 (NNI)

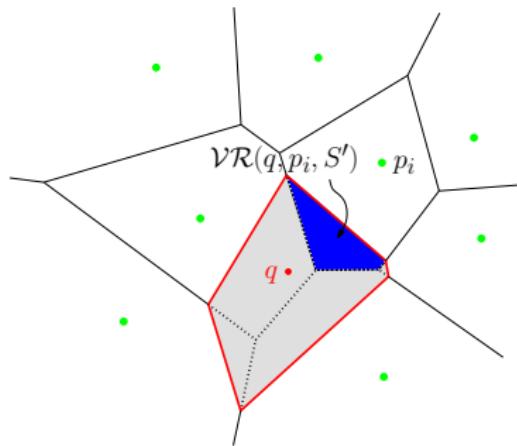
For a set  $S$  of  $k$  sites  $p_1, p_2, \dots, p_k \in \mathbb{R}^2$  with associated (scalar or vector-valued) "data"  $v_1, v_2, \dots, v_k$ , and  $q \in CH(S)$ ,

$$f(q) := \sum_{i=1}^k v_i \lambda_i(q)$$

gives the interpolated data for  $q$  obtained by natural-neighbor interpolation (NNI), where

$$\lambda_i(q) := \frac{|\mathcal{VR}(q, p_i, S')|}{|\mathcal{VR}(q, S')|} \text{ with } S' := S \cup \{q\}.$$

Here,  $|\mathcal{VR}(q, S')|$  denotes the area of the Voronoi cell of  $q$  within  $S'$ , and  $|\mathcal{VR}(q, p_i, S')|$  corresponds to the area of the second-order Voronoi cell of  $q$  and  $p_i$ .



# Improved Laser Sintering Based on Natural-Neighbor Interpolation

- Laser sintering is a manufacturing process used in rapid prototyping:
  - A laser is used to manufacture a part by sintering powder-based materials layer by layer.
  - Small-series production is possible.
  - Snap fits and living hinges can be produced.



Images courtesy of EOS GmbH

# Improved Laser Sintering Based on Natural-Neighbor Interpolation

- Major problem:

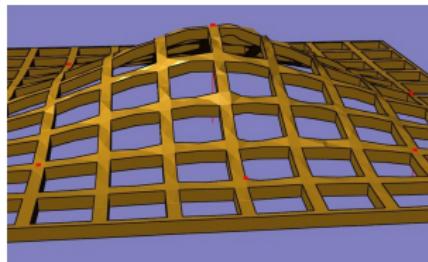
- The laser-induced heating and subsequent cooling down of the material may cause the “warpage” phenomenon.
- Warpage is the result of a change in the morphology of the molten powder: amorphous to part-crystalline.
- Crystalline regions have a higher density than the amorphous regions, leading to a loss of volume.
- Different layers undergo different loss in volume, leading to inter-layer tension.



# Improved Laser Sintering Based on Natural-Neighbor Interpolation

- Major problem:

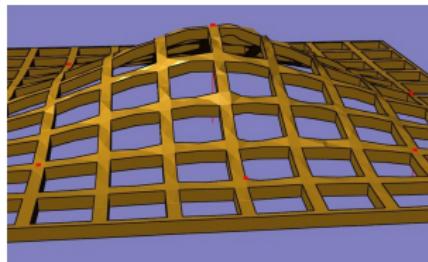
- The laser-induced heating and subsequent cooling down of the material may cause the “warpage” phenomenon.
- Warpage is the result of a change in the morphology of the molten powder: amorphous to part-crystalline.
- Crystalline regions have a higher density than the amorphous regions, leading to a loss of volume.
- Different layers undergo different loss in volume, leading to inter-layer tension.
- This tension may result in a bimetallic effect: “curl”.



# Improved Laser Sintering Based on Natural-Neighbor Interpolation

- Major problem:

- The laser-induced heating and subsequent cooling down of the material may cause the “warpage” phenomenon.
- Warpage is the result of a change in the morphology of the molten powder: amorphous to part-crystalline.
- Crystalline regions have a higher density than the amorphous regions, leading to a loss of volume.
- Different layers undergo different loss in volume, leading to inter-layer tension.
- This tension may result in a bimetallic effect: “curl”.



- Bold idea: Apply a pre-deformation in order to manufacture an inversely deformed part!

# Improved Laser Sintering Based on Natural-Neighbor Interpolation

Given:

- An STL model of a polyhedral part  $\mathcal{P}$ , assumed to represent a triangulated closed polyhedral 2-manifold.
- Deformation vectors for  $k$  of the  $n$  vertices of  $\mathcal{P}$ , with  $k \ll n$ .

# Improved Laser Sintering Based on Natural-Neighbor Interpolation

Given:

- An STL model of a polyhedral part  $\mathcal{P}$ , assumed to represent a triangulated closed polyhedral 2-manifold.
- Deformation vectors for  $k$  of the  $n$  vertices of  $\mathcal{P}$ , with  $k \ll n$ .

Sought:

- Suitable deformation vectors for all  $n$  vertices of  $\mathcal{P}$ .

# Improved Laser Sintering Based on Natural-Neighbor Interpolation

## Given:

- An STL model of a polyhedral part  $\mathcal{P}$ , assumed to represent a triangulated closed polyhedral 2-manifold.
- Deformation vectors for  $k$  of the  $n$  vertices of  $\mathcal{P}$ , with  $k \ll n$ .

## Sought:

- Suitable deformation vectors for all  $n$  vertices of  $\mathcal{P}$ .
- It seems natural to use interpolation — but how shall we interpolate vectors on the surface of a polyhedron?

# Improved Laser Sintering Based on Natural-Neighbor Interpolation

## Given:

- An STL model of a polyhedral part  $\mathcal{P}$ , assumed to represent a triangulated closed polyhedral 2-manifold.
- Deformation vectors for  $k$  of the  $n$  vertices of  $\mathcal{P}$ , with  $k \ll n$ .

## Sought:

- Suitable deformation vectors for all  $n$  vertices of  $\mathcal{P}$ .
- It seems natural to use interpolation — but how shall we interpolate vectors on the surface of a polyhedron?
- [Held&Pfligersdorffer 2009]: Pre-deformation by means of approximate natural-neighbor interpolation (NNI) helps to reduce warpage by 90%.

# Improved Laser Sintering Based on Natural-Neighbor Interpolation

- Approximate VD on the surface of the Stanford dragon.



# Improved Laser Sintering Based on Natural-Neighbor Interpolation

- Approximate VD on the surface of the Stanford dragon.



- Pre-deformation works neatly for reasonably triangulated parts and a reasonable number of deformation vectors.

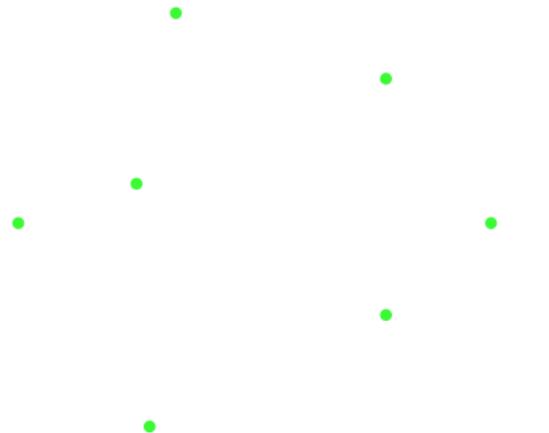
# Improved Laser Sintering Based on Natural-Neighbor Interpolation

- Approximate VD on the surface of the Stanford dragon.



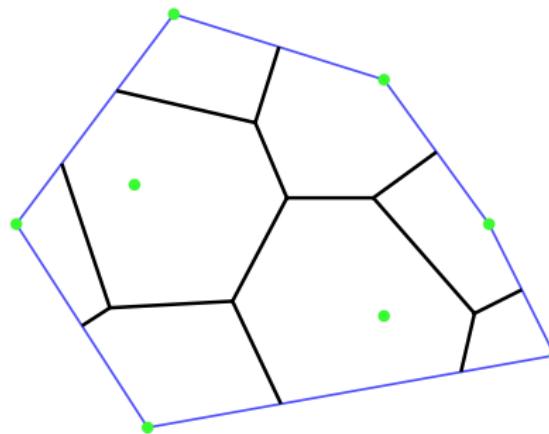
- Pre-deformation works neatly for reasonably triangulated parts and a reasonable number of deformation vectors.
- Open problems:
  - Dire need for a decent re-meshing software became apparent.
  - How can we generate the deformation vectors automatically?

# Maximum Empty Circle



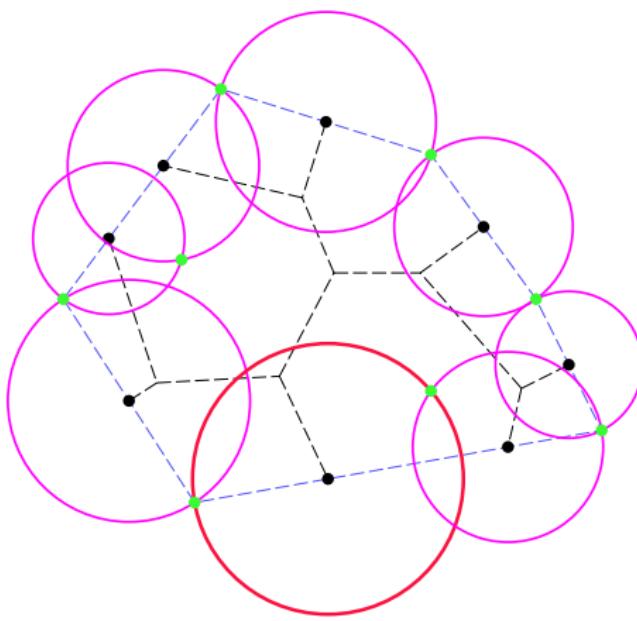
# Maximum Empty Circle

- ① Restrict  $\mathcal{VD}(S)$  to  $CH(S)$ .



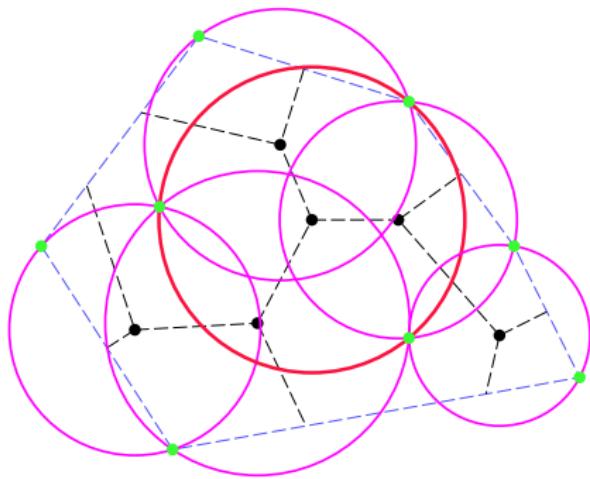
## Maximum Empty Circle

- ① Restrict  $\mathcal{VD}(S)$  to  $CH(S)$ .
- ② Determine the largest circle centered at an intersection of  $\mathcal{VD}(S)$  and  $CH(S)$ .



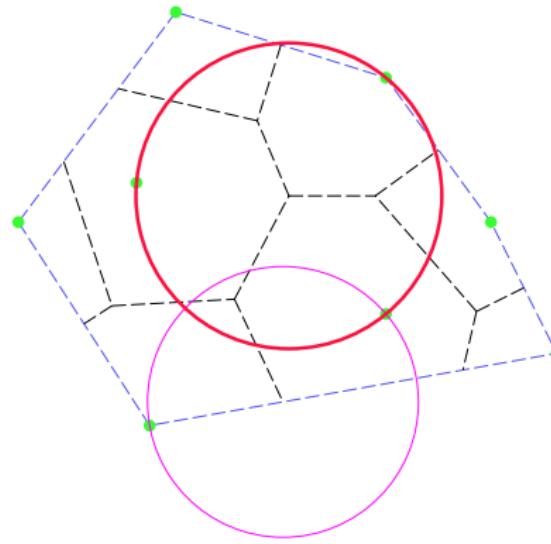
# Maximum Empty Circle

- ① Restrict  $\mathcal{VD}(S)$  to  $CH(S)$ .
- ② Determine the largest circle centered at an intersection of  $\mathcal{VD}(S)$  and  $CH(S)$ .
- ③ Determine the largest circle centered at an interior node of  $\mathcal{VD}(S)$ .



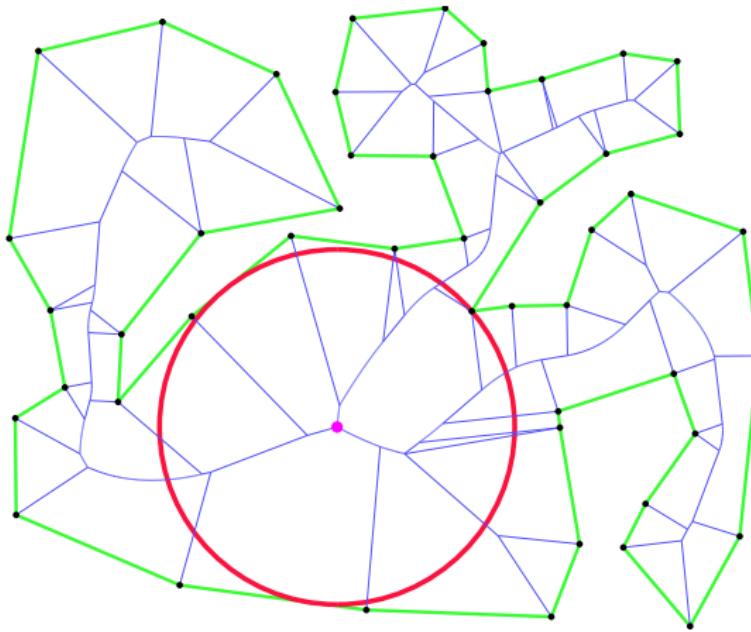
# Maximum Empty Circle

- ① Restrict  $\mathcal{VD}(S)$  to  $CH(S)$ .
- ② Determine the largest circle centered at an intersection of  $\mathcal{VD}(S)$  and  $CH(S)$ .
- ③ Determine the largest circle centered at an interior node of  $\mathcal{VD}(S)$ .
- ④ Pick the largest circle among those two categories of circles.



## Maximum Inscribed Circle

- Similarly, scanning the Voronoi nodes interior to a polygon yields a maximum inscribed circle in  $O(n)$  time.



## Offsetting: Minkowski Sum and Difference

- Let  $A, B$  be sets, and  $a, b$  denote points of  $A$  respectively  $B$ .
- We define the translation of  $A$  by the vector  $b$  as

$$A_b := \{a + b : a \in A\}.$$

## Offsetting: Minkowski Sum and Difference

- Let  $A, B$  be sets, and  $a, b$  denote points of  $A$  respectively  $B$ .
- We define the translation of  $A$  by the vector  $b$  as

$$A_b := \{a + b : a \in A\}.$$

- The *Minkowski sum* of  $A$  and  $B$  is defined as

$$A \oplus B := \bigcup_{b \in B} A_b.$$

## Offsetting: Minkowski Sum and Difference

- Let  $A, B$  be sets, and  $a, b$  denote points of  $A$  respectively  $B$ .
- We define the translation of  $A$  by the vector  $b$  as

$$A_b := \{a + b : a \in A\}.$$

- The *Minkowski sum* of  $A$  and  $B$  is defined as

$$A \oplus B := \bigcup_{b \in B} A_b.$$

- The *Minkowski difference* of  $A$  and  $B$  is defined as

$$A \ominus B := \bigcap_{b \in B} A_{-b}.$$

## Offsetting: Minkowski Sum and Difference

- Let  $A, B$  be sets, and  $a, b$  denote points of  $A$  respectively  $B$ .
- We define the translation of  $A$  by the vector  $b$  as

$$A_b := \{a + b : a \in A\}.$$

- The *Minkowski sum* of  $A$  and  $B$  is defined as

$$A \oplus B := \bigcup_{b \in B} A_b.$$

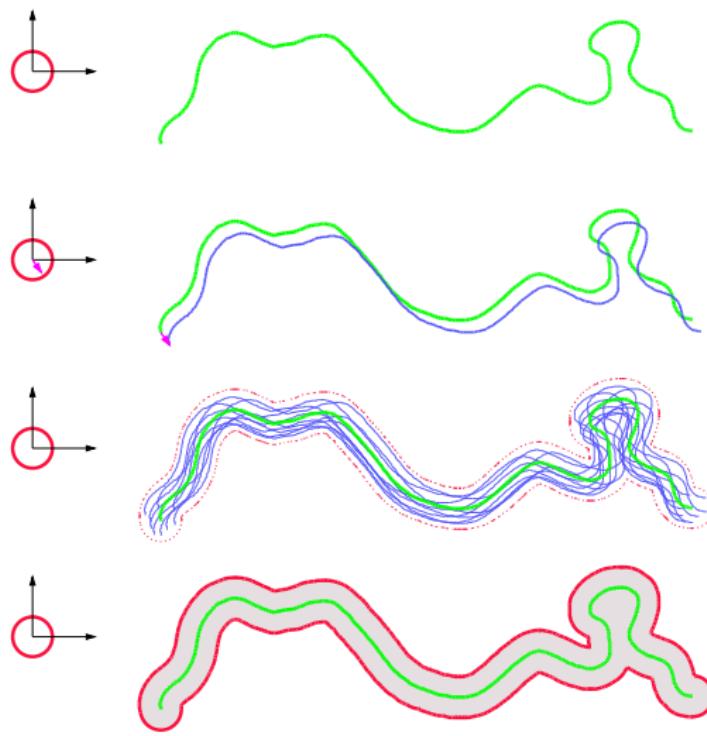
- The *Minkowski difference* of  $A$  and  $B$  is defined as

$$A \ominus B := \bigcap_{b \in B} A_{-b}.$$

- Note: In general,  $(A \oplus B) \ominus B \neq A$ .

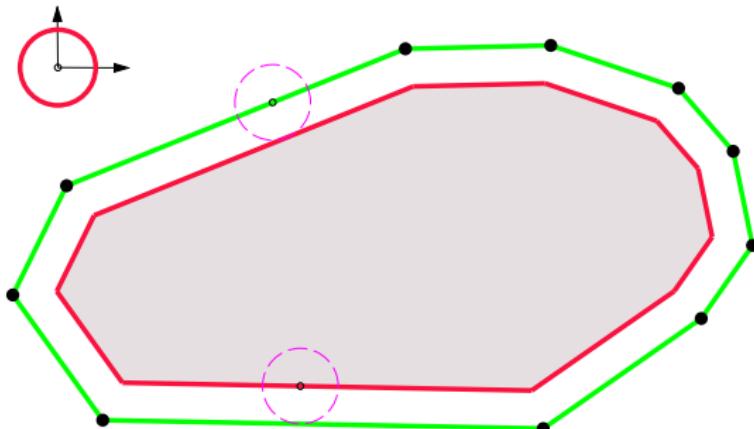
## Offsetting: Sample Minkowski Sum

- Let  $A$  be a curve, and  $B$  be a circular disk centered at the origin. What is  $A \oplus B$ ?



## Offsetting: Sample Minkowski Difference

- Let  $A$  be a polygon, and  $B$  be a circular disk centered at the origin. What is  $A \ominus B$ ?

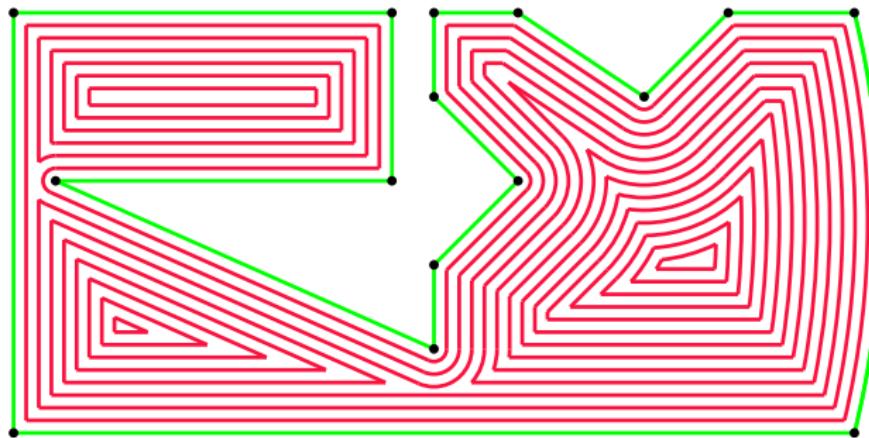


## Offsetting: Topological Changes

- Minkowski sums and differences of an area  $A$  with a circular disk  $B$  centered at the origin are also called *offsets* (in CAD/CAM) and *buffers* (in GIS).

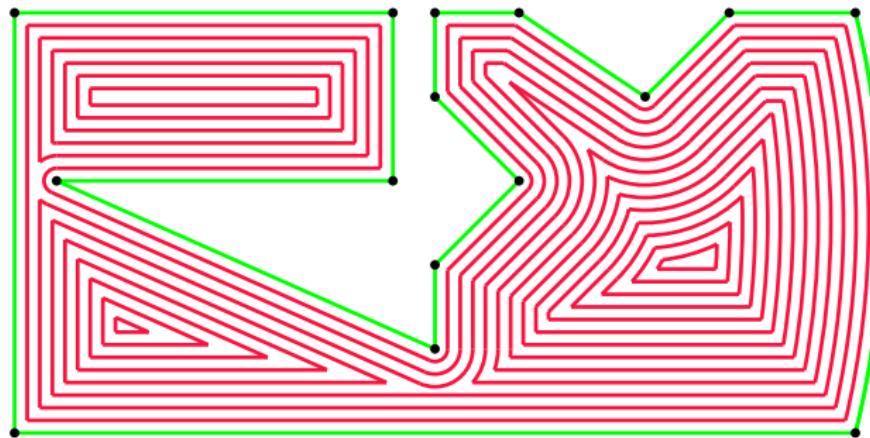
## Offsetting: Topological Changes

- Minkowski sums and differences of an area  $A$  with a circular disk  $B$  centered at the origin are also called *offsets* (in CAD/CAM) and *buffers* (in GIS).
- Note: The boundary of an offset may contain circular arcs even if the input is purely polygonal.



## Offsetting: Topological Changes

- Minkowski sums and differences of an area  $A$  with a circular disk  $B$  centered at the origin are also called *offsets* (in CAD/CAM) and *buffers* (in GIS).
- Note: The boundary of an offset may contain circular arcs even if the input is purely polygonal.
- Note: Offsetting may cause topological changes!



## Buffering in Geographical Information Systems

- Sample GIS application: Identify that portion of the territorial waters of Malta which is within three nautical miles of the baseline (coast) of Malta.



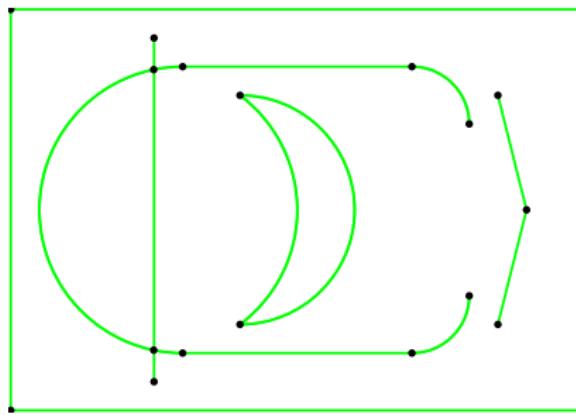
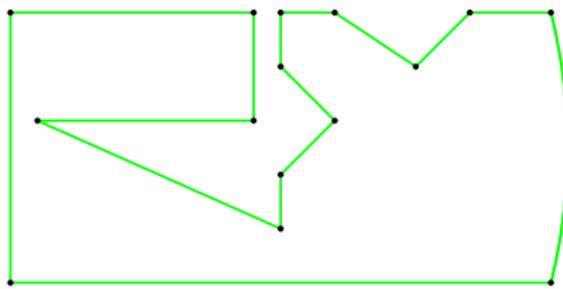
## Buffering in Geographical Information Systems

- Sample GIS application: Identify that portion of the territorial waters of Malta which is within three nautical miles of the baseline (coast) of Malta.



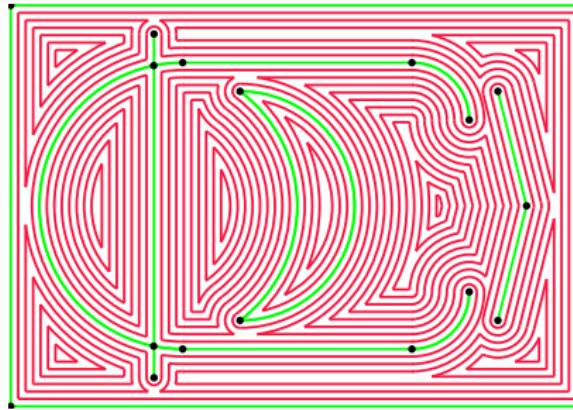
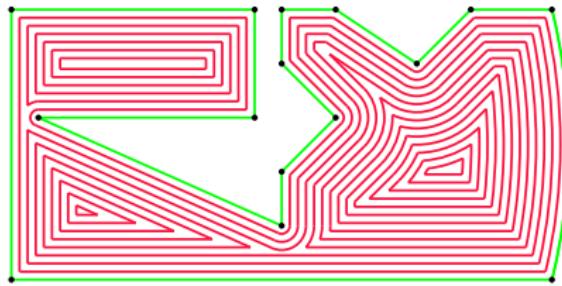
# Computation of Offset Patterns

- How can we compute offset patterns reliably and efficiently?



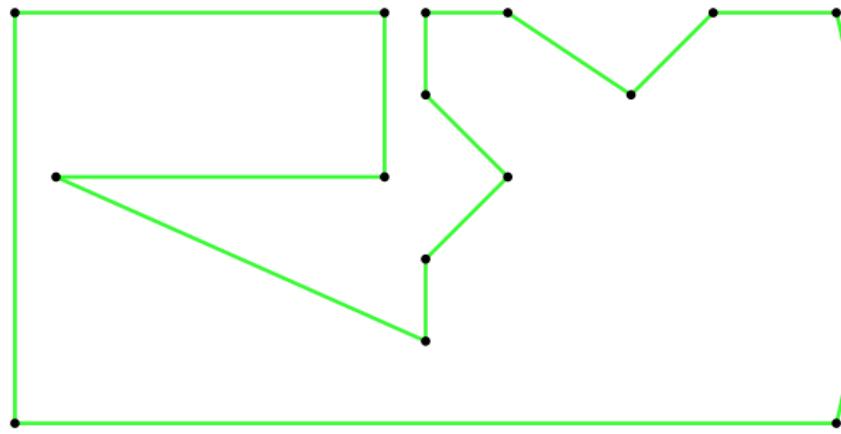
# Computation of Offset Patterns

- How can we compute offset patterns reliably and efficiently?



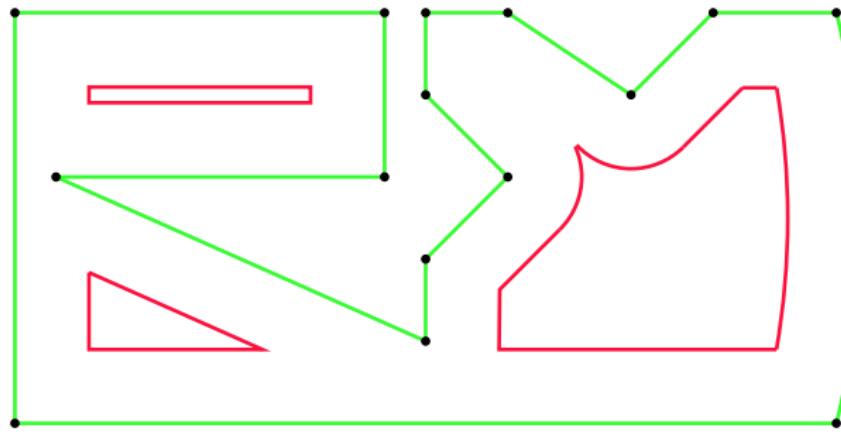
## Conventional Offsetting

- How can we compute even just one individual offset?

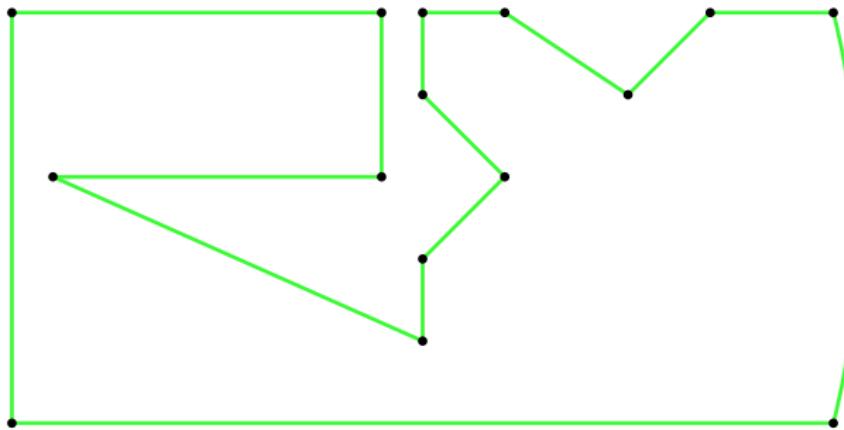


## Conventional Offsetting

- How can we compute even just one individual offset?

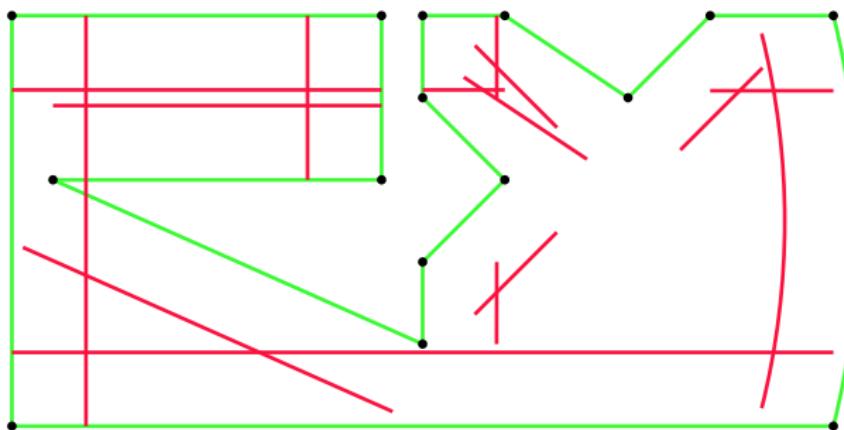


## Conventional Offsetting



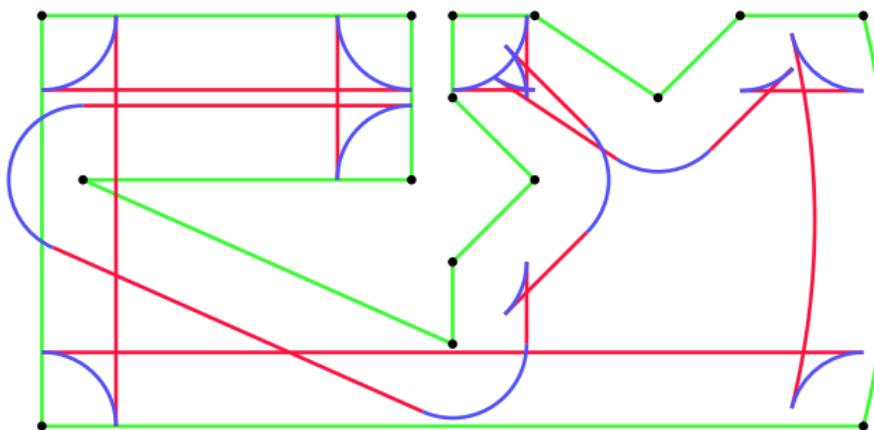
## Conventional Offsetting

- ① First, one computes offset elements for every input element.



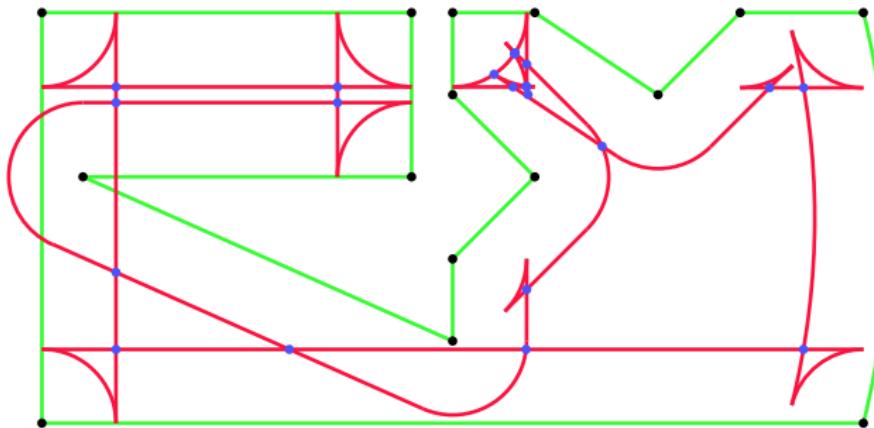
## Conventional Offsetting

- ① First, one computes offset elements for every input element.
- ② In order to get one closed loop, trimming arcs are inserted.



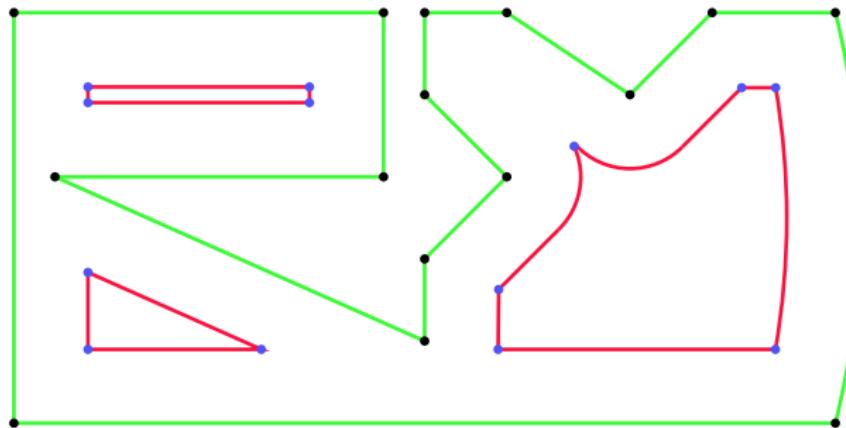
## Conventional Offsetting

- ① First, one computes offset elements for every input element.
- ② In order to get one closed loop, trimming arcs are inserted.
- ③ Next, all self-intersections are determined.



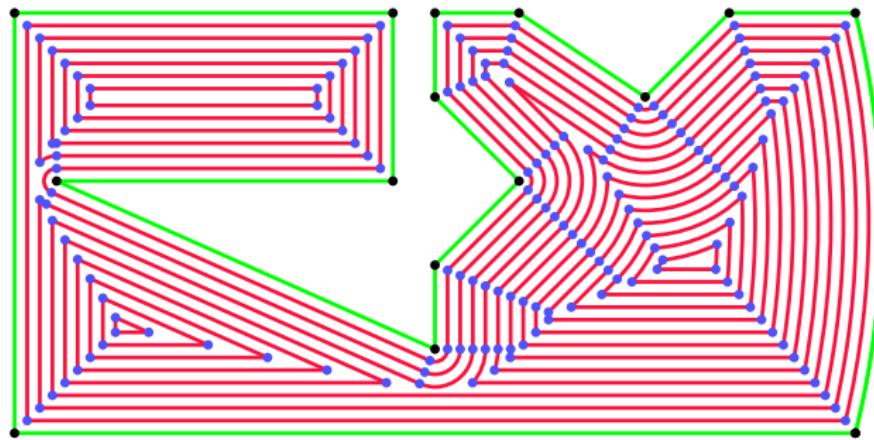
## Conventional Offsetting

- ① First, one computes offset elements for every input element.
- ② In order to get one closed loop, trimming arcs are inserted.
- ③ Next, all self-intersections are determined.
- ④ Finally, all incorrect loops of the offset are removed.



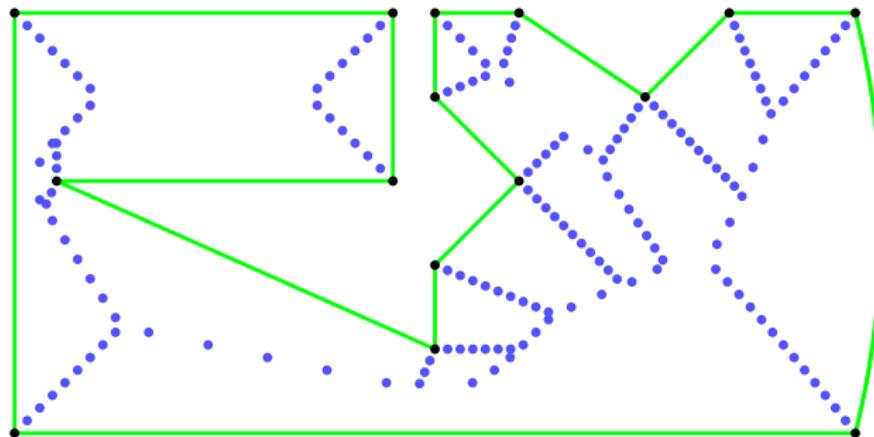
## Voronoi-Based Offsetting

- We start with analyzing the positions of the end-points of the offset segments.



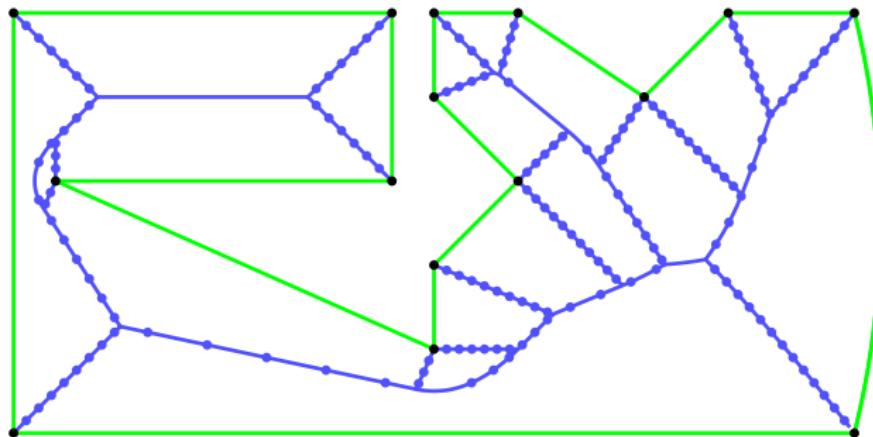
## Voronoi-Based Offsetting

- We start with analyzing the positions of the end-points of the offset segments.
- This looks familiar!



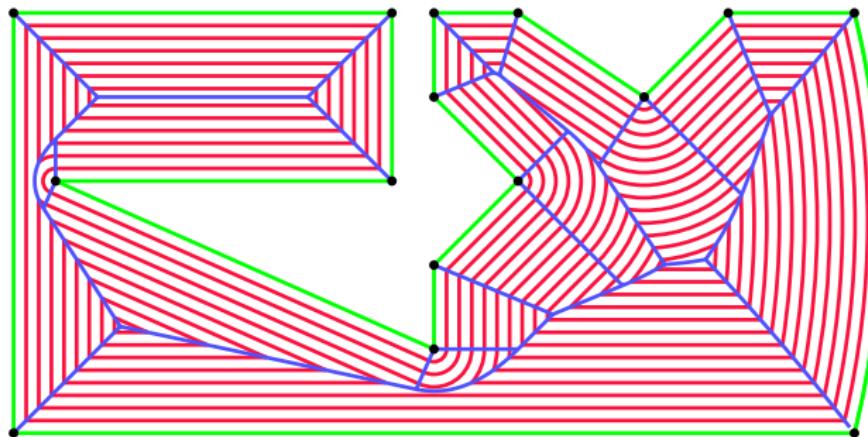
## Voronoi-Based Offsetting

- We start with analyzing the positions of the end-points of the offset segments.
- This looks familiar!
- Indeed, all end-points of offset segments lie on the Voronoi diagram!

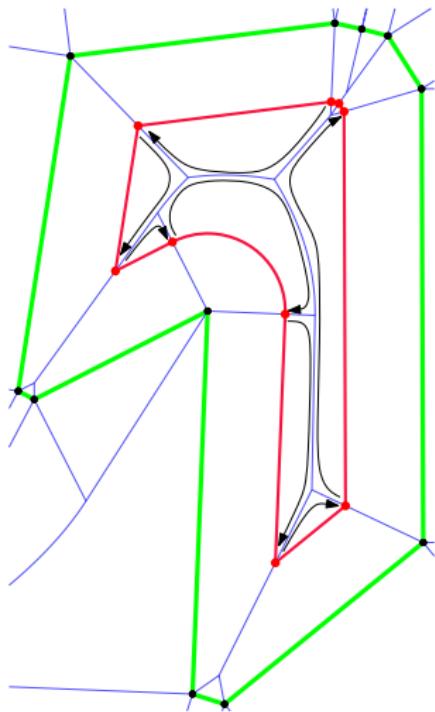


## Voronoi-Based Offsetting

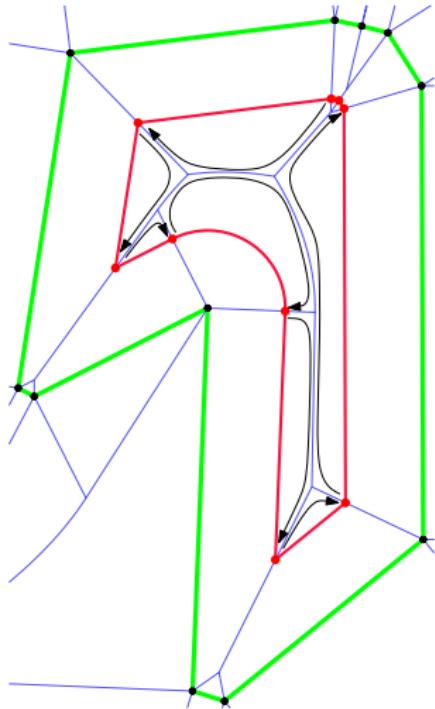
- We start with analyzing the positions of the end-points of the offset segments.
- This looks familiar!
- Indeed, all end-points of offset segments lie on the Voronoi diagram!
- A linear-time scan of the Voronoi diagram reveals the end-points of one offset.



# Voronoi-Based Offsetting



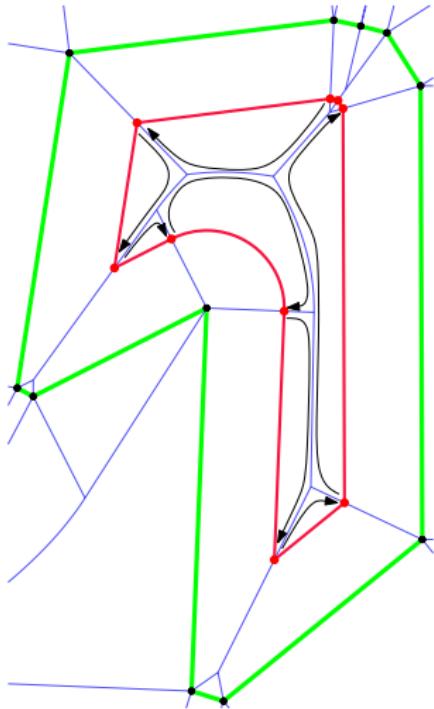
# Voronoi-Based Offsetting



## Theorem 96 (Persson 1978, Held 1991)

Let  $S$  be an admissible set of sites, and  $t \in \mathbb{R}^+$ . If  $\mathcal{VD}(S)$  is known then all offset curves of  $S$  at offset  $t$  can be determined in  $O(n)$  time.

# Voronoi-Based Offsetting



## Theorem 96 (Persson 1978, Held 1991)

Let  $S$  be an admissible set of sites, and  $t \in \mathbb{R}^+$ . If  $\mathcal{VD}(S)$  is known then all offset curves of  $S$  at offset  $t$  can be determined in  $O(n)$  time.

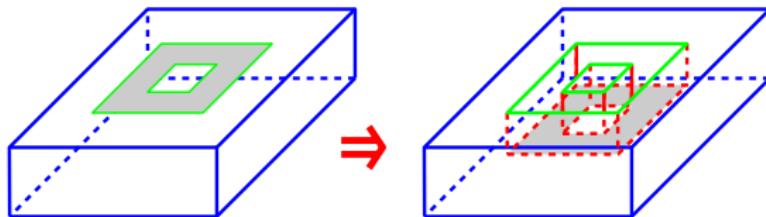
## Corollary 97

Let  $S$  be an admissible set of sites, and  $t \in \mathbb{R}^+$ . Then all offset curves of  $S$  at offset  $t$  can be determined in  $O(n \log n)$  time.

# Tool Paths for Pocket Machining

## Pocket Machining

**Pocket:** Interior recess that is cut into the surface of a workpiece.

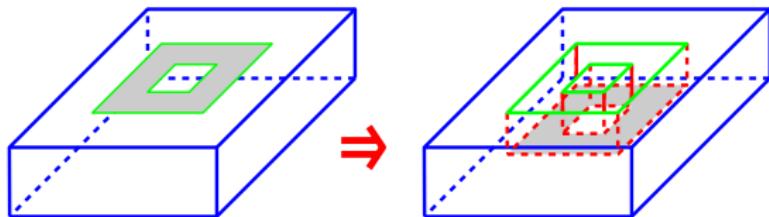


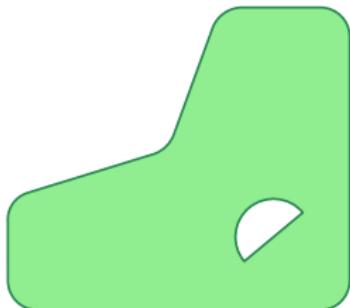
# Tool Paths for Pocket Machining

## Pocket Machining

**Pocket:** Interior recess that is cut into the surface of a workpiece.

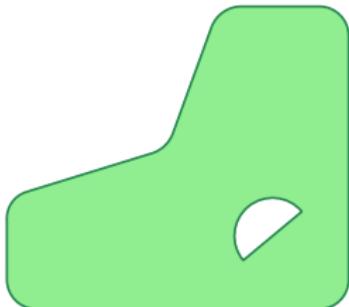
**Tool:** Can be regarded as a cylinder that rotates.





## Geometry of a pocket

- 2D area,
- straight-line segments and circular arcs as boundary elements,
- may contain islands.

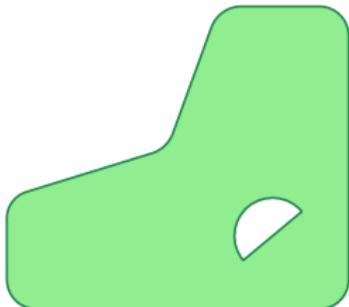


## Geometry of a pocket

- 2D area,
- straight-line segments and circular arcs as boundary elements,
- may contain islands.

## Geometry of a tool

- circular disk.



## Geometry of a pocket

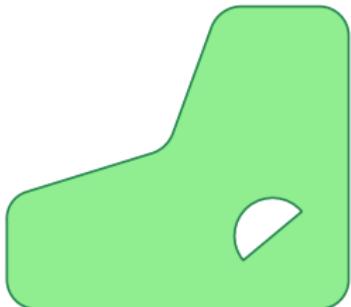
- 2D area,
- straight-line segments and circular arcs as boundary elements,
- may contain islands.

## Geometry of a tool

- circular disk.

## The goal is ...

- ... to compute a “good” tool path.



## Geometry of a pocket

- 2D area,
- straight-line segments and circular arcs as boundary elements,
- may contain islands.

## Geometry of a tool

- circular disk.

## The goal is ...

- ... to compute a “good” tool path.

- Similar path planning problems arise in many other applications that require “coverage” of an area by a disk-shaped object, e.g., layered manufacturing, spray painting, aerial surveillance.

# Voronoi-Based Generation of Tool Path

Persson (1978), Held (1991)

- Family of offset curves forms a tool path.
- Tool path computed by means of Voronoi diagram.



## Pros of offset-based machining

- Offset curves can be computed easily (based on Voronoi diagram),
- Reasonably short tool path.

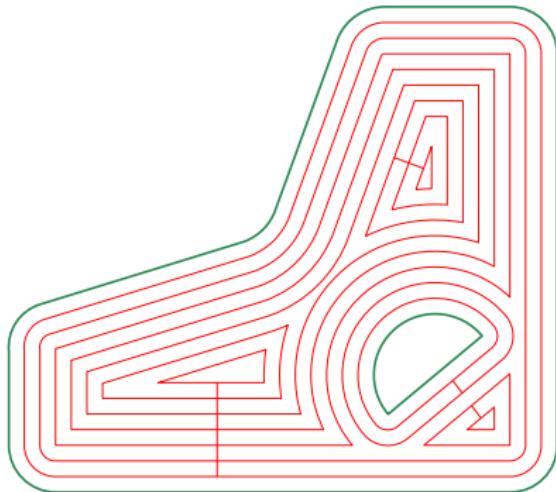
# Voronoi-Based Generation of Tool Path

## Pros of offset-based machining

- Offset curves can be computed easily (based on Voronoi diagram),
- Reasonably short tool path.

## Cons of offset-based machining

- Sharp corners,
- Highly varying material removal rate,
- Might require tool retractions,
- Not suitable for high-speed machining.



# Voronoi-Based Generation of Tool Path

## Pros of offset-based machining

- Offset curves can be computed easily (based on Voronoi diagram),
- Reasonably short tool path.

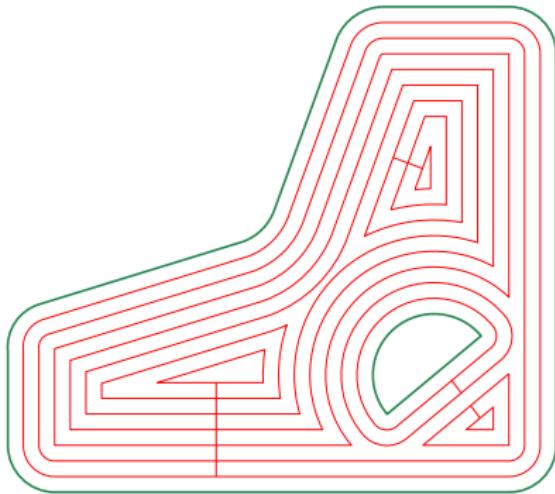
## Cons of offset-based machining

- Sharp corners,
- Highly varying material removal rate,
- Might require tool retractions,
- Not suitable for high-speed machining.

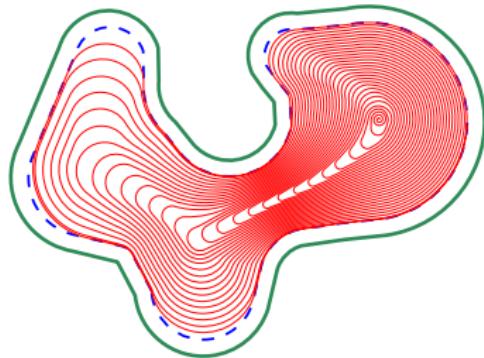
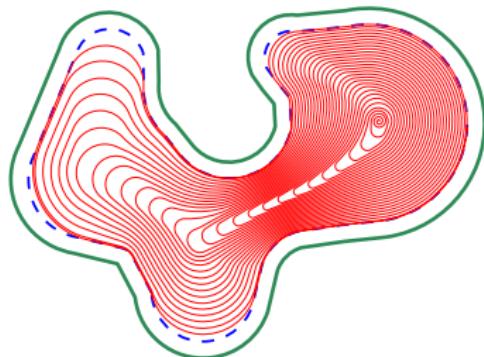
## High-speed machining (HSM)

Faster tool movement requires

- smooth tool paths,
- low variation of material removal rate.



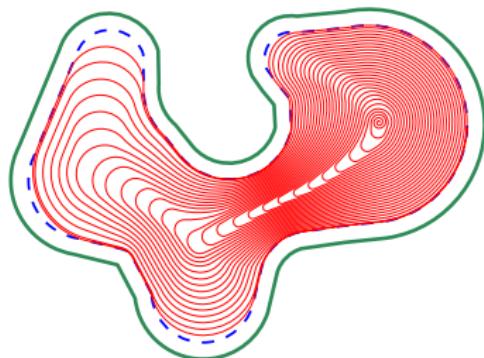
# Voronoi-Based Generation of Spiral Tool Paths



Held & Spielberger (2009)

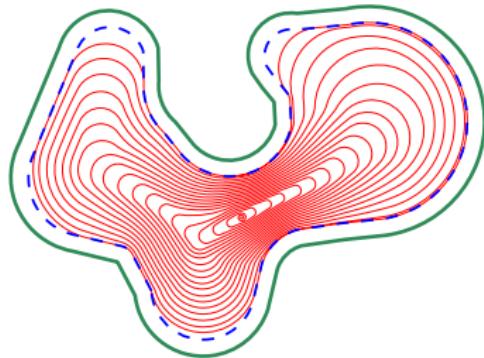
- Smooth spiral tool path for HSM.
- Handle general pockets without islands.

# Voronoi-Based Generation of Spiral Tool Paths



## Held&Spielberger (2009)

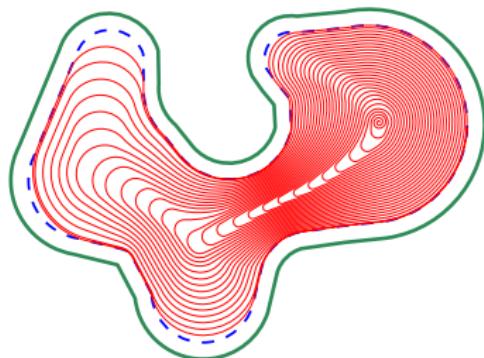
- Smooth spiral tool path for HSM.
- Handle general pockets without islands.



## Held&Spielberger (2013)

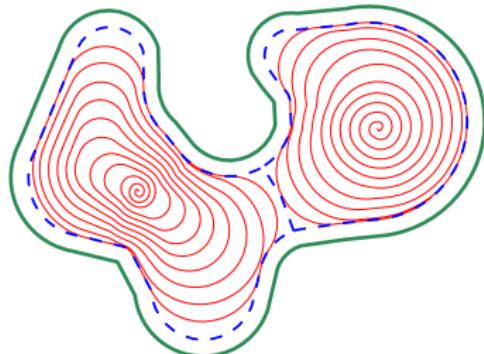
- Optimization of the start point of the spiral tool path.

# Voronoi-Based Generation of Spiral Tool Paths



## Held&Spielberger (2009)

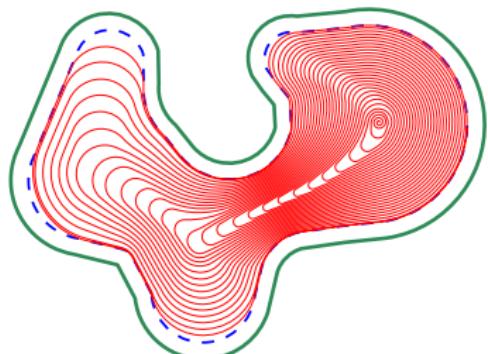
- Smooth spiral tool path for HSM.
- Handle general pockets without islands.



## Held&Spielberger (2013)

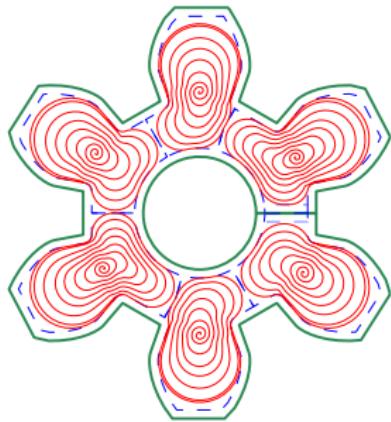
- Optimization of the start point of the spiral tool path.
- Decomposition of "complex" pockets.

# Voronoi-Based Generation of Spiral Tool Paths



## Held&Spielberger (2009)

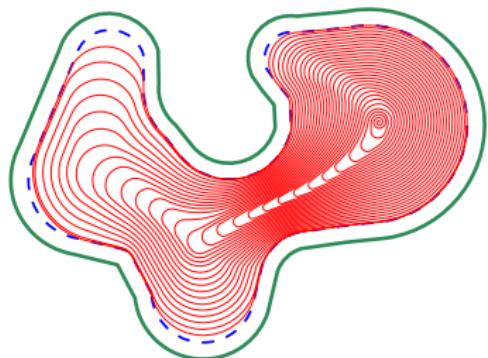
- Smooth spiral tool path for HSM.
- Handle general pockets without islands.



## Held&Spielberger (2013)

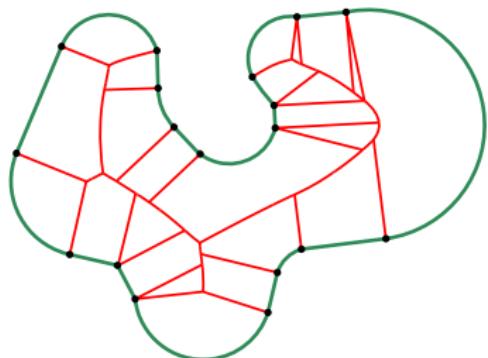
- Optimization of the start point of the spiral tool path.
- Decomposition of "complex" pockets.
- Handle pockets with islands.

# Voronoi-Based Generation of Spiral Tool Paths



## Held&Spielberger (2009)

- Smooth spiral tool path for HSM.
- Handle general pockets without islands.



## Held&Spielberger (2013)

- Optimization of the start point of the spiral tool path.
- Decomposition of "complex" pockets.
- Handle pockets with islands.

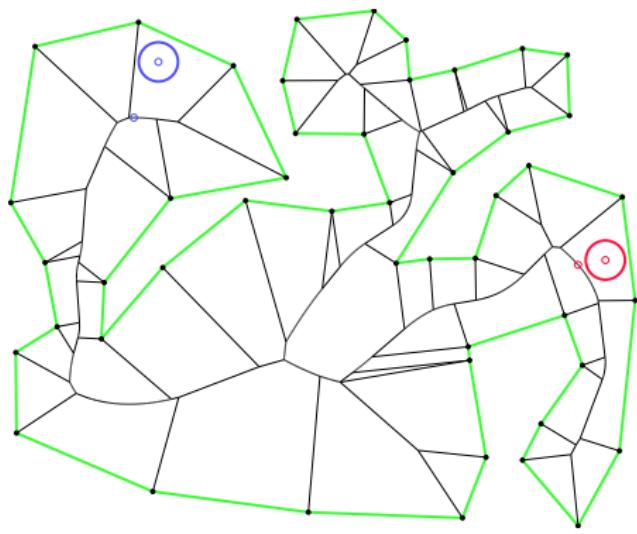
## Algorithmic vehicle

- Voronoi diagram and medial axis of pocket boundary.



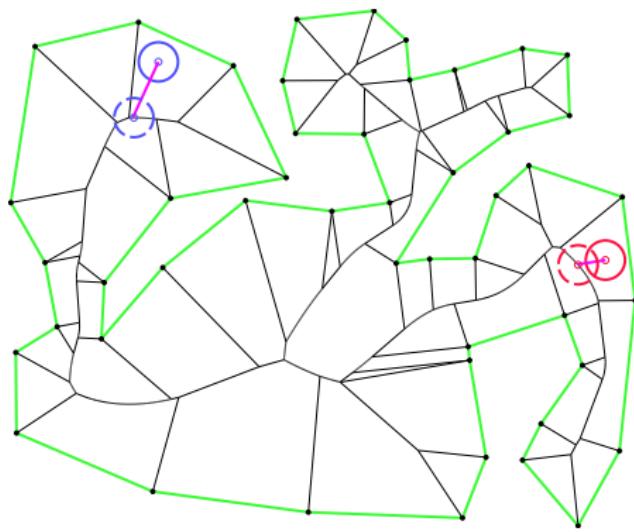
## Finding a Gouge-Free Path

- Can we move the disk within the polygon from the blue to the red position?



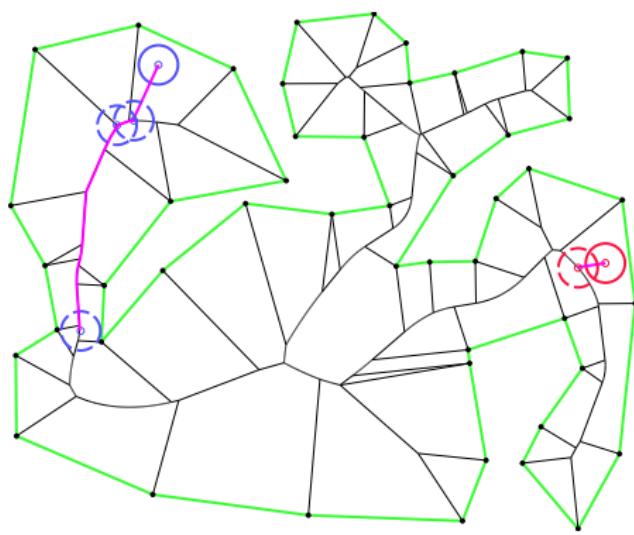
## Finding a Gouge-Free Path

- Can we move the disk within the polygon from the blue to the red position?
- Retraction method: Project red and blue centers onto the Voronoi diagram.



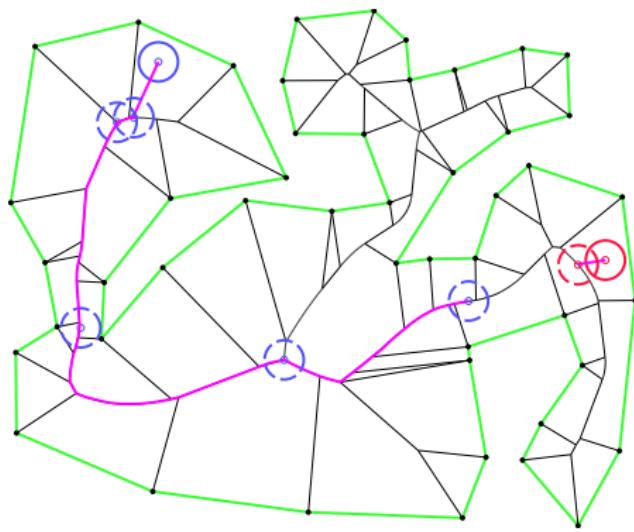
## Finding a Gouge-Free Path

- Can we move the disk within the polygon from the blue to the red position?
- Retraction method: Project red and blue centers onto the Voronoi diagram.
- Scan the Voronoi diagram to find a way from blue to red.



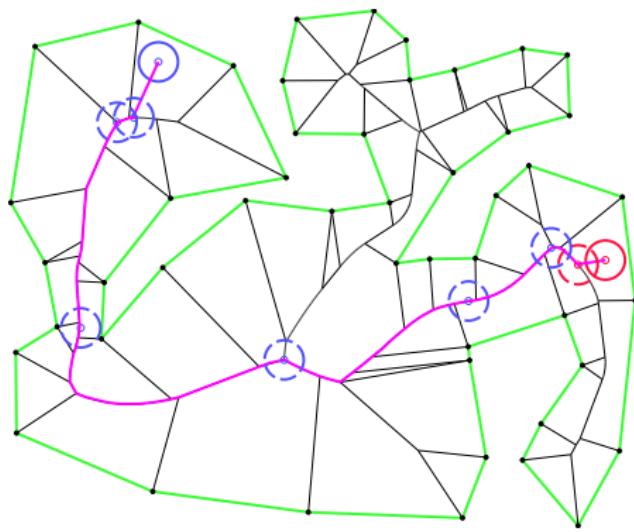
## Finding a Gouge-Free Path

- Can we move the disk within the polygon from the blue to the red position?
- Retraction method: Project red and blue centers onto the Voronoi diagram.
- Scan the Voronoi diagram to find a way from blue to red.
- Make sure to check the clearance while moving through a bottleneck.



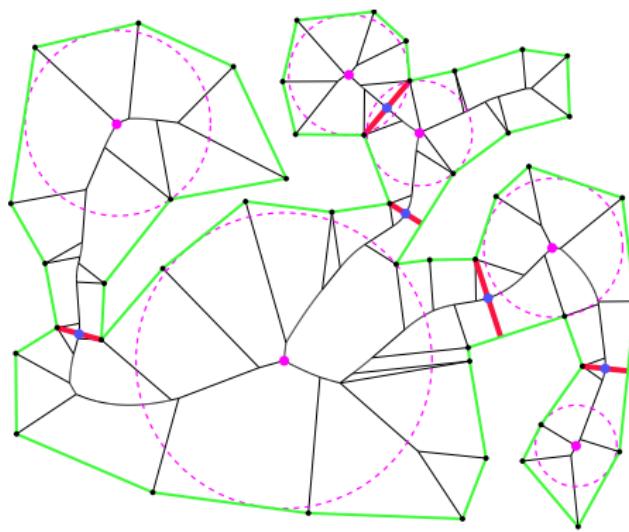
## Finding a Gouge-Free Path

- Can we move the disk within the polygon from the blue to the red position?
- Retraction method: Project red and blue centers onto the Voronoi diagram.
- Scan the Voronoi diagram to find a way from blue to red.
- Make sure to check the clearance while moving through a bottleneck.
- Indeed, this disk can be moved from blue to red!



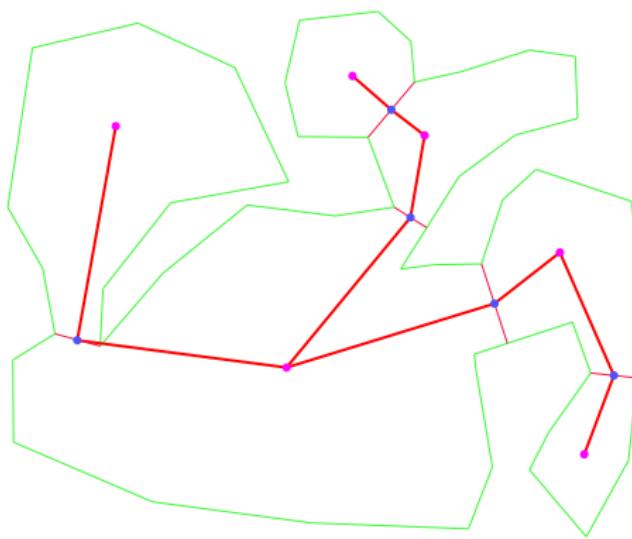
## Bottlenecks and Locally Inner-Most Points

- A linear-time scan of the VD reveals all bottlenecks and locally inner-most points.



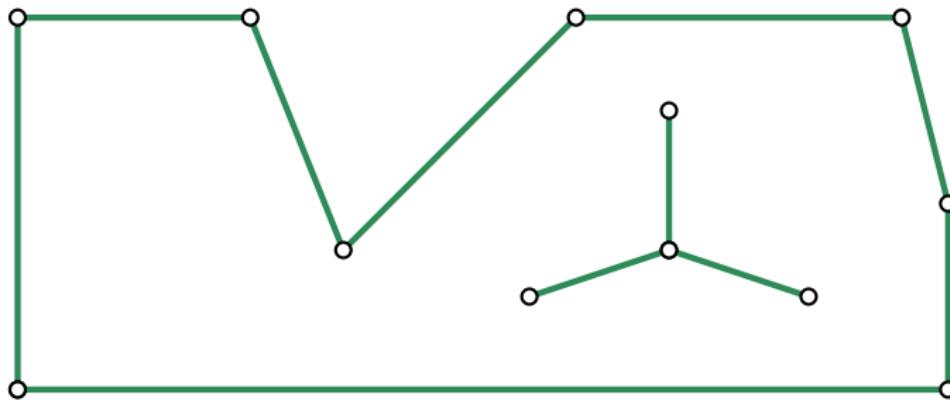
## Bottlenecks and Locally Inner-Most Points

- To save time, a graph search is performed on the graph of offset-connected areas.



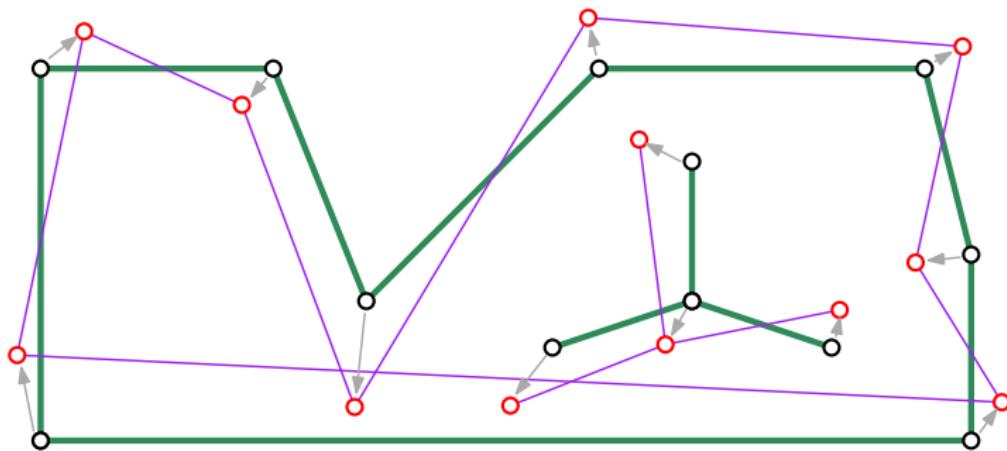
# Topology-Preserving Watermarking of Vector Graphics

- Watermarking techniques for vector graphics dislocate vertices in order to embed imperceptible, yet detectable, statistical features into the input data.



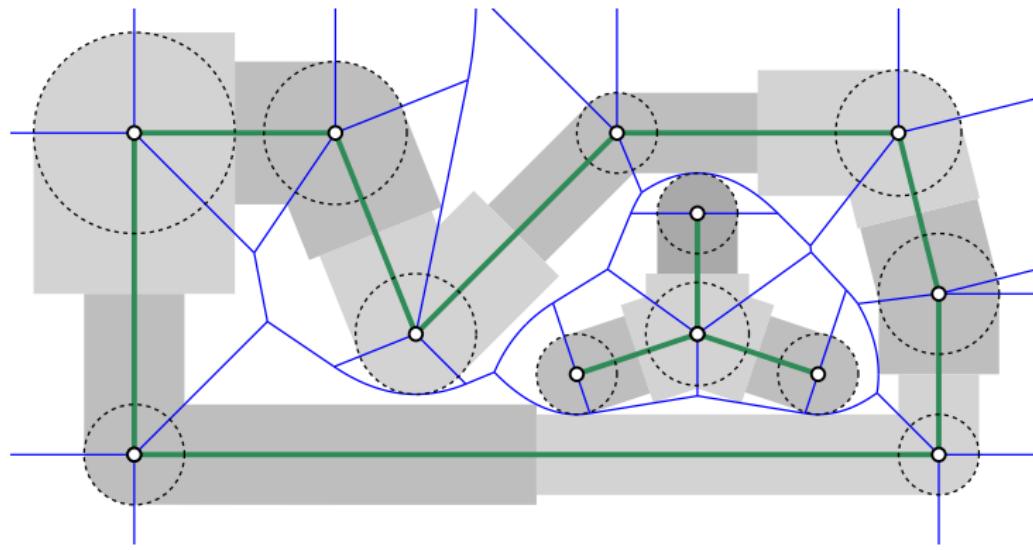
# Topology-Preserving Watermarking of Vector Graphics

- Watermarking techniques for vector graphics dislocate vertices in order to embed imperceptible, yet detectable, statistical features into the input data.
- Obvious problem: One needs to guarantee that the introduction of a watermark preserves the input topology.



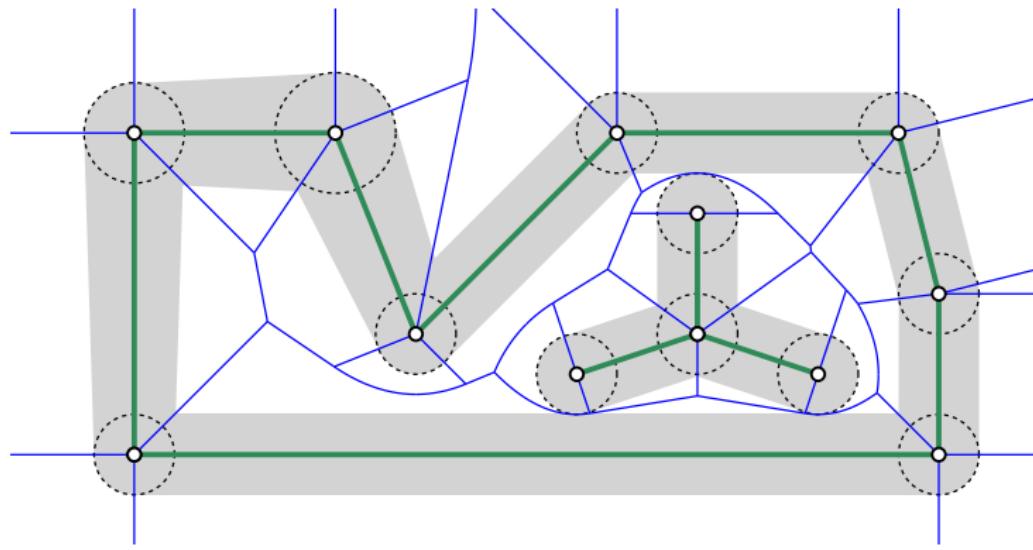
# Topology-Preserving Watermarking of Vector Graphics

- [Huber et alii 2013] compute for each vertex a disk-shaped maximum perturbation region (MPR), based on the Voronoi diagram of the input.



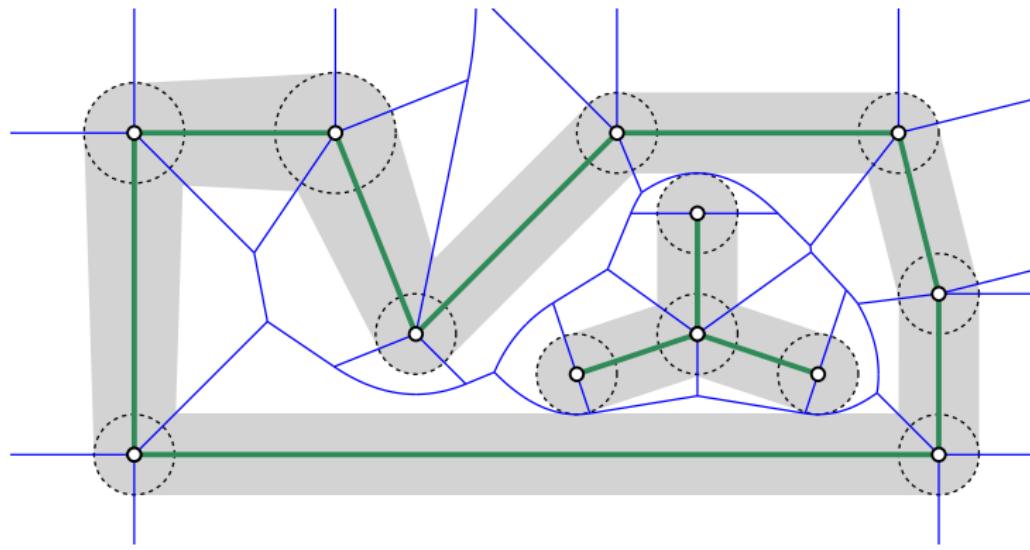
# Topology-Preserving Watermarking of Vector Graphics

- [Huber et alii 2013] compute for each vertex a disk-shaped maximum perturbation region (MPR), based on the Voronoi diagram of the input.
- Perturbing the vertices within their MPRs causes the edges to stay within their hoses and allows to preserve the input topology.



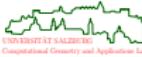
# Topology-Preserving Watermarking of Vector Graphics

- [Huber et alii 2013] compute for each vertex a disk-shaped maximum perturbation region (MPR), based on the Voronoi diagram of the input.
- Perturbing the vertices within their MPRs causes the edges to stay within their hoses and allows to preserve the input topology.
- This scheme can be extended to cover straight-line segments and circular arcs.



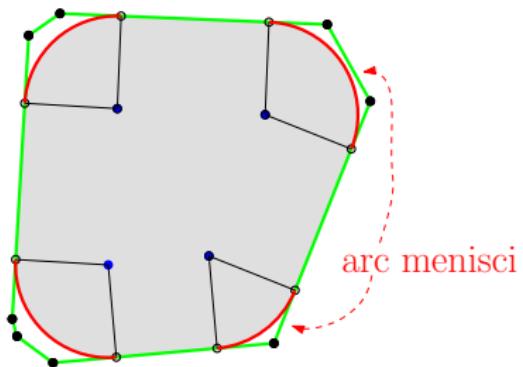
# Analysis of Drainage Displacements

- Drainage displacements in tubular cross sections: A non-wetting fluid displaces a wetting fluid in a porous medium of a prescribed (polygonal) cross section.



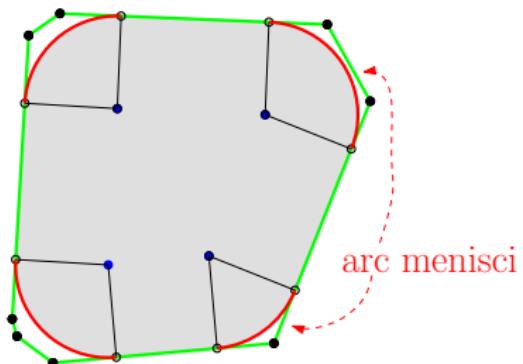
# Analysis of Drainage Displacements

- Drainage displacements in tubular cross sections: A non-wetting fluid displaces a wetting fluid in a porous medium of a prescribed (polygonal) cross section.
- If fluids are in equilibrium, and without gravity: The radii of the arc menisci which separate the non-wetting and wetting fluids have to be identical.
- Special case: The cross section is a convex polygon and the wetting angle  $\Theta$  equals  $90^\circ$ .



# Analysis of Drainage Displacements

- Drainage displacements in tubular cross sections: A non-wetting fluid displaces a wetting fluid in a porous medium of a prescribed (polygonal) cross section.
- If fluids are in equilibrium, and without gravity: The radii of the arc menisci which separate the non-wetting and wetting fluids have to be identical.
- Special case: The cross section is a convex polygon and the wetting angle  $\Theta$  equals  $90^\circ$ .



- Applications:

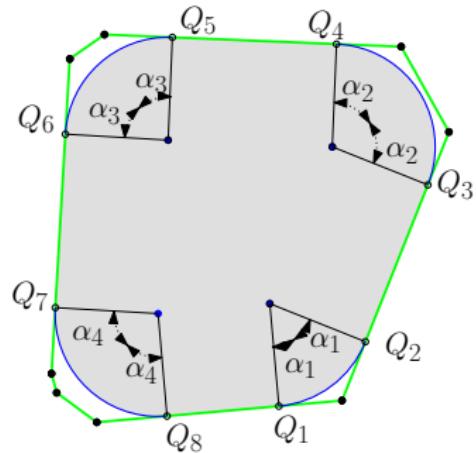
- Oil industry: flow of oil, gas and water in reservoirs is of great technological and economical importance.
- Hydrology and soil science: modelling of ground water resources and pollution effects of the acid deposition.



# Analysis of Drainage Displacements

- Terminology:

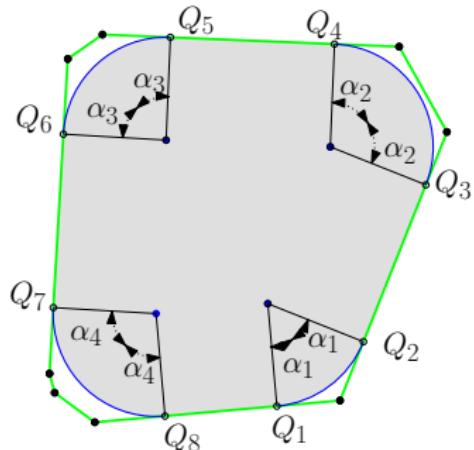
- $\alpha_i$  ... half of the opening angle of arc meniscus  $i$  (for some radius  $r$ )
- $\alpha(r) := \sum_i \alpha_i$
- $L_{Ns}(r)$  ... total length of contact between the cross section and the non-wetting fluid
- $A_N(r)$  ... area covered by the non-wetting fluid



# Analysis of Drainage Displacements

- Terminology:

- $\alpha_i$  ... half of the opening angle of arc meniscus  $i$  (for some radius  $r$ )
- $\alpha(r) := \sum_i \alpha_i$
- $L_{NS}(r)$  ... total length of contact between the cross section and the non-wetting fluid
- $A_N(r)$  ... area covered by the non-wetting fluid



- MS-P Equation [Lindquist 2006]:

Equilibrium holds if the radius  $r$  of the arc menisci fulfills the equation

$$2r^2 \cdot \alpha(r) + r \cdot L_{NS}(r) - A_N(r) = 0.$$

- Lindquist used a bisection scheme for computing an approximate solution.

# Analysis of Drainage Displacements

- Terminology:

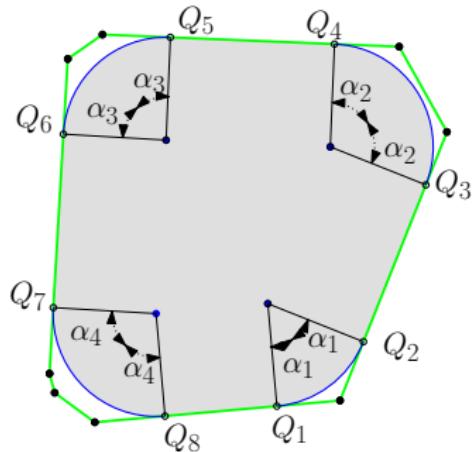
$\alpha_i$  ... half of the opening angle of arc meniscus  $i$  (for some radius  $r$ )

$$\alpha(r) := \sum_i \alpha_i$$

$L_{NS}(r)$  ... total length of contact between the cross section and the non-wetting fluid

$A_N(r)$  ... area covered by the non-wetting fluid

$\rho$  ... clearance of innermost node



- VD-based solution [Held 2010]:  $LHS(r) := 2r^2 \cdot \alpha(r) + r \cdot L_{NS}(r) - A_N(r)$ 
  - is continuous,
  - is strictly monotonously increasing, with  $LHS(0) < 0$  and  $LHS(\rho) > 0$ ,
  - is piecewise polynomial of degree two.

The one and only positive root of  $LHS(r) = 0$  can be determined in  $O(n \log n)$  time.

# Analysis of Drainage Displacements

- Cross section is partitioned into trapezoids defined by the VD and by the clearance lines of all VD nodes.
- For a trapezoid  $T$ :

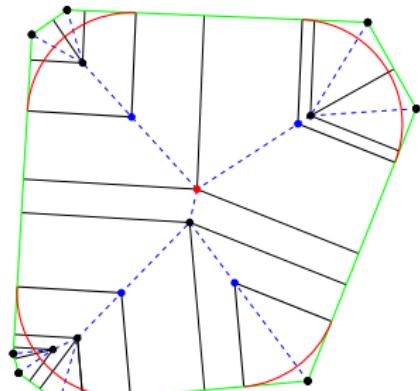
$r_{min}(T)$  ... minimum clearance of points on the VD edge of  $T$

$r_{max}(T)$  ... maximum clearance of points on the VD edge of  $T$

$h(T)$  ... base length of  $T$

$A(T)$  ... area of  $T$

$\rho$  ... clearance of innermost node



# Analysis of Drainage Displacements

- We have

$$2r^2 \cdot \alpha(r) = 2r^2\pi.$$

# Analysis of Drainage Displacements

- We have

$$2r^2 \cdot \alpha(r) = 2r^2\pi.$$

- Let  $r_0 < r_1 \leq r_2 \leq r_3 \dots \leq r_k = \rho$  be the sorted clearances of all  $k$  VD nodes, with  $r_0 := 0$ .



# Analysis of Drainage Displacements

- We have

$$2r^2 \cdot \alpha(r) = 2r^2\pi.$$

- Let  $r_0 < r_1 \leq r_2 \leq r_3 \dots \leq r_k = \rho$  be the sorted clearances of all  $k$  VD nodes, with  $r_0 := 0$ .
- For  $r_i < r \leq r_{i+1}$ :

$$r \cdot L_{NS}(r) = r \left( \sum_{r_{min}(T) \geq r_{i+1}} h(T) + \sum_{r_i < r_{max}(T) \leq r_{i+1}} h_r(T) \right)$$

and

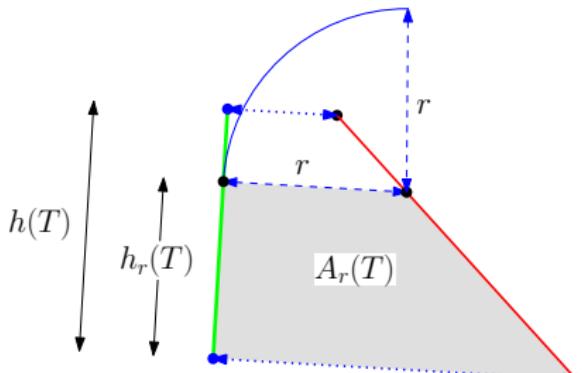
$$A_N(r) = r^2\pi + \sum_{r_{min}(T) \geq r_{i+1}} A(T) + \sum_{r_i < r_{max}(T) \leq r_{i+1}} A_r(T),$$

where  $h_r(T)$  and  $A_r(T)$  denote the appropriate portions of  $h(T)$  and  $A(T)$  in dependence on  $r$ .

# Analysis of Drainage Displacements

- Similar triangles yield

$$h_r(T) = h(T) \cdot \frac{r_{\max}(T) - r}{r_{\max}(T) - r_{\min}(T)}$$



and

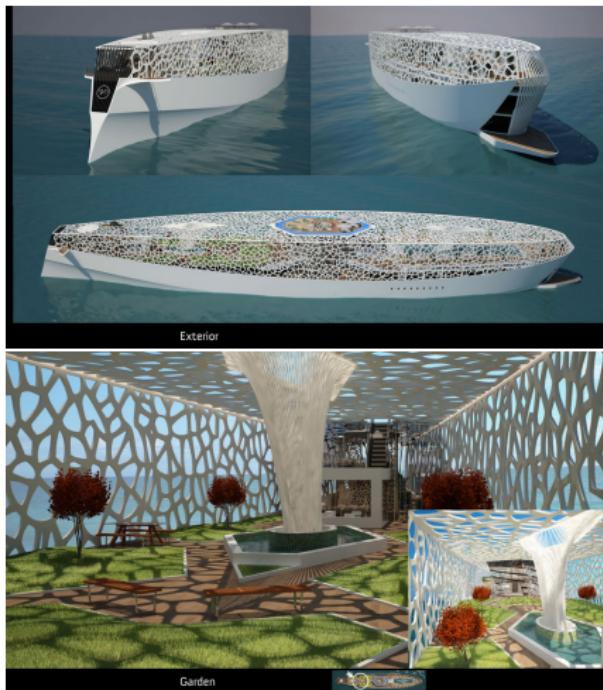
$$A_r(T) = h_r(T) \cdot \frac{r_{\max}(T) + r}{2} = h(T) \cdot \frac{r_{\max}^2(T) - r^2}{2(r_{\max}(T) - r_{\min}(T))},$$

where  $h(T)$ ,  $r_{\min}(T)$  and  $r_{\max}(T)$  are constants that depend only on the geometry of  $T$  but do not vary as  $r$  varies.

- Differentiation of LHS establishes its piecewise strict monotonicity.

# Voronoi Diagrams in Structural Design

- CNN (16-Aug-2011): “Stunning superyacht design inspired by nature’s hidden patterns”.



[Image credit: Hyun-Seok Kim, [www.yachtndesign.com](http://www.yachtndesign.com)]

# Voronoi Diagrams in Nature: Centroidal Voronoi Diagrams and Territorial Behavior

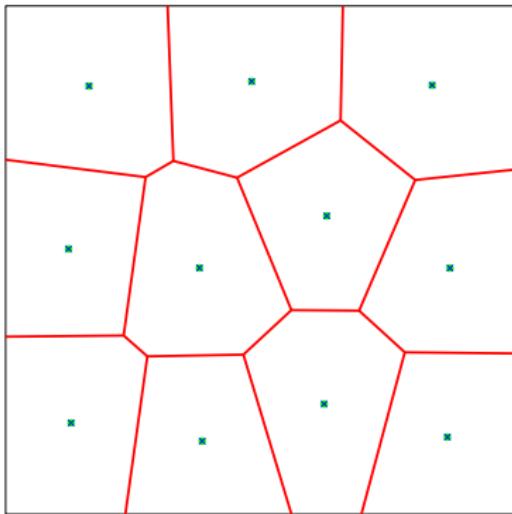
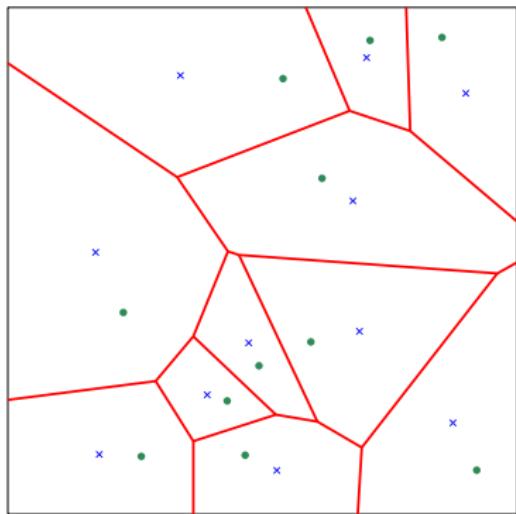
## Definition 98 (Centroidal Voronoi Diagram (CVD))

A Voronoi diagram of a set of points is called *centroidal* if the **points** are also **centroids** of the Voronoi regions, i.e., centers of mass with respect to a given density function.

# Voronoi Diagrams in Nature: Centroidal Voronoi Diagrams and Territorial Behavior

## Definition 98 (Centroidal Voronoi Diagram (CVD))

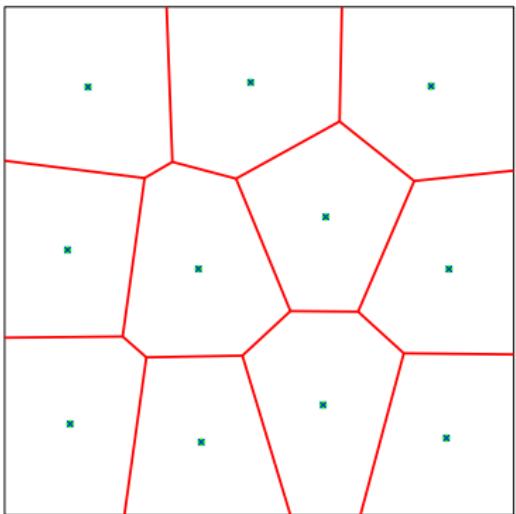
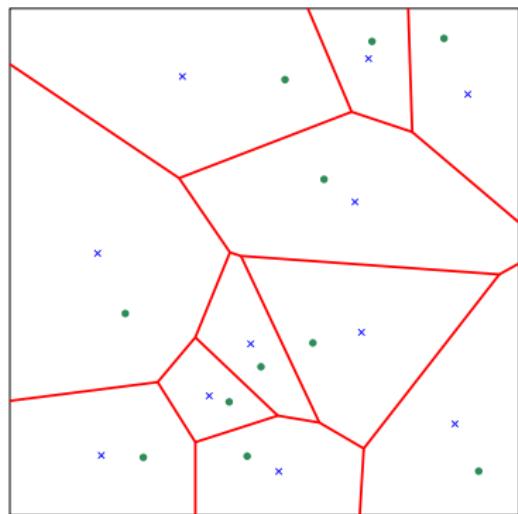
A Voronoi diagram of a set of points is called *centroidal* if the **points** are also **centroids** of the Voronoi regions, i.e., centers of mass with respect to a given density function.



# Voronoi Diagrams in Nature: Centroidal Voronoi Diagrams and Territorial Behavior

## Definition 98 (Centroidal Voronoi Diagram (CVD))

A Voronoi diagram of a set of points is called *centroidal* if the **points** are also **centroids** of the Voronoi regions, i.e., centers of mass with respect to a given density function.

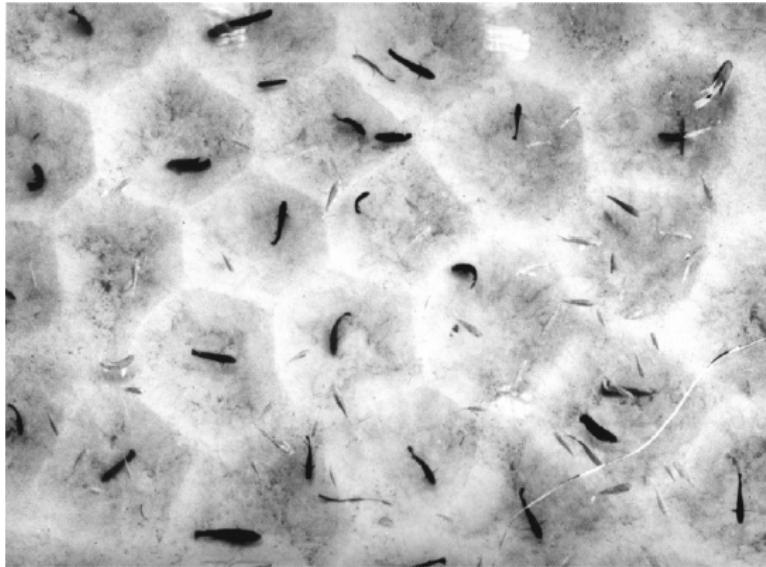


- Applications of CVDs: data compression, image segmentation, mesh generation, modeling of territorial behavior of animals, etc.



# Voronoi Diagrams in Nature: Centroidal Voronoi Diagrams and Territorial Behavior

- Tilapia mossambica:
  - The male fishes dig nesting pits into sandy grounds.
  - The centers and corners of the pits are adjusted until the final configuration resembles a centroidal Voronoi diagram.



[Image credit: G. Barlow, "Hexagonal Territories", Animal Behavior 22:876–878, 1974.]



## VDs in Nature: Evaporation as a Massively Parallel Algorithm?



- Salar de Atacama in the Chilean Andes: 3 000 km<sup>2</sup>, average elevation about 2 300 m asl., 3 500 milliliters annual evaporation, only a few milliliters of annual rainfall.

# Voronoi Diagram, Presumed Dead, Now Really Dead

by Mark Forez

The Voronoi diagram, wanted for multiple accounts of cliché, was found dead this week in Melbourne, Australia, finally putting an end to the mystery surrounding its whereabouts.

The missing diagram, responsible for decomposing metric space in multiple countries through the determination of boundaries between specified discrete points, was found upturned in a road-side latrine. Despite considerable bloating and partial consumption by dingos, forensic experts report that the diagram showed years of fatigue, abuse and overexposure to the elements prior to death.

A friend of the diagram, who asked to remain anonymous, presented an eulogy at the funeral: "We used to use [Voronoi diagram] all the damn time. When I was a student, we all were using it. It was easy, just a few clicks and bam—instant cool. I felt like Spiderman. No one ever thought much about it."

Donations to the Voronoi Memorial Foundation can be made via PayPal at <http://www.theverymany.com>.

[Text credit: [fasterdonuts.tumblr.com](http://fasterdonuts.tumblr.com), 13-Feb-2011]

## Triangulations

- Introduction
  - Definitions
  - Types of Triangulations
- Constrained Triangulations
- Computing Constrained Triangulations
  - Facts and State-of-the-Art
  - Polygon Triangulation via Ear-Clipping
- Triangulations in 3D
- Applications of Triangulations
  - Rendering
  - Triangulated Irregular Network
  - Visibility Determination
  - Topologically Consistent Watermarking

## Triangulations

- Introduction
  - Definitions
  - Types of Triangulations
- Constrained Triangulations
- Computing Constrained Triangulations
- Triangulations in 3D
- Applications of Triangulations



## Definition 99 (Simplex)

A  $k$ -simplex is the convex hull of  $k + 1$  affinely independent points in  $\mathbb{R}^d$ .

## Definition 99 (Simplex)

A  $k$ -simplex is the convex hull of  $k + 1$  affinely independent points in  $\mathbb{R}^d$ .

## Lemma 100

A  $k$ -simplex is a  $k$ -dimensional polytope in  $\mathbb{R}^d$ .

## Definition 99 (Simplex)

A  $k$ -simplex is the convex hull of  $k + 1$  affinely independent points in  $\mathbb{R}^d$ .

## Lemma 100

A  $k$ -simplex is a  $k$ -dimensional polytope in  $\mathbb{R}^d$ .

- Of course,  $d \geq k$ .
- Simplices form a generalization of triangles to  $d$  dimensions and represent the simplest-possible polytopes: point (0-simplex), segment (1-simplex), triangle (2-simplex), tetrahedron (3-simplex), pentachoron (4-simplex), etc.

## Definition 99 (Simplex)

A  $k$ -simplex is the convex hull of  $k + 1$  affinely independent points in  $\mathbb{R}^d$ .

## Lemma 100

A  $k$ -simplex is a  $k$ -dimensional polytope in  $\mathbb{R}^d$ .

- Of course,  $d \geq k$ .
- Simplices form a generalization of triangles to  $d$  dimensions and represent the simplest-possible polytopes: point (0-simplex), segment (1-simplex), triangle (2-simplex), tetrahedron (3-simplex), pentachoron (4-simplex), etc.
- The boundary of a simplex is given by lower-dimensional elements like vertices, edges, faces, etc.

## Definition 99 (Simplex)

A  $k$ -simplex is the convex hull of  $k + 1$  affinely independent points in  $\mathbb{R}^d$ .

## Lemma 100

A  $k$ -simplex is a  $k$ -dimensional polytope in  $\mathbb{R}^d$ .

- Of course,  $d \geq k$ .
- Simplices form a generalization of triangles to  $d$  dimensions and represent the simplest-possible polytopes: point (0-simplex), segment (1-simplex), triangle (2-simplex), tetrahedron (3-simplex), pentachoron (4-simplex), etc.
- The boundary of a simplex is given by lower-dimensional elements like vertices, edges, faces, etc.

## Warning

Note that the terminology is not consistent! In particular, the terms “face” and “facet” seem to take on several meanings across different authors.



## Definition 101 (Face)

A  $j$ -face of a  $k$ -simplex is a  $j$ -simplex formed by  $j + 1$  of the points that define the  $k$ -simplex.



## Definition 101 (Face)

A  $j$ -face of a  $k$ -simplex is a  $j$ -simplex formed by  $j + 1$  of the points that define the  $k$ -simplex.

- The empty set is seen as a (-1)-dimensional face common to all simplices.

## Definition 101 (Face)

A  $j$ -face of a  $k$ -simplex is a  $j$ -simplex formed by  $j + 1$  of the points that define the  $k$ -simplex.

- The empty set is seen as a (-1)-dimensional face common to all simplices.

## Lemma 102

Every  $k$ -simplex has exactly  $\binom{k+1}{j+1}$   $j$ -faces.

## Definition 101 (Face)

A  $j$ -face of a  $k$ -simplex is a  $j$ -simplex formed by  $j + 1$  of the points that define the  $k$ -simplex.

- The empty set is seen as a (-1)-dimensional face common to all simplices.

## Lemma 102

Every  $k$ -simplex has exactly  $\binom{k+1}{j+1}$   $j$ -faces.

- Common names for faces of a  $k$ -simplex:
  - 0-face: vertex
  - 1-face: edge
  - 2-face: face
  - 3-face: cell
  - $(k - 2)$ -face: ridge
  - $(k - 1)$ -face: facet



## Definition 103 (Triangulation in $\mathbb{R}^d$ )

A *triangulation* of a finite set  $S$  of points in  $\mathbb{R}^d$ , for  $d \geq 2$ , is a finite collection  $\mathcal{T}(S)$  of  $d$ -simplices such that

- ① the union of all simplices equals  $CH(S)$ ,
- ② every point of  $S$  is a vertex of at least one simplex,
- ③ any pair of these simplices intersects in a common face (which, possibly, is the  $(-1)$ -face).



## Definition 103 (Triangulation in $\mathbb{R}^d$ )

A *triangulation* of a finite set  $S$  of points in  $\mathbb{R}^d$ , for  $d \geq 2$ , is a finite collection  $\mathcal{T}(S)$  of  $d$ -simplices such that

- ① the union of all simplices equals  $CH(S)$ ,
  - ② every point of  $S$  is a vertex of at least one simplex,
  - ③ any pair of these simplices intersects in a common face (which, possibly, is the  $(-1)$ -face).
- 
- Sometimes, the first condition is relaxed and it is only demanded that all points of  $S$  form vertices of  $d$ -simplices, and that the boundary of the union of all simplices forms a  $(d - 1)$ -manifold, i.e., a simple curve in  $\mathbb{R}^2$  or a surface in  $\mathbb{R}^3$ .



## Definition 103 (Triangulation in $\mathbb{R}^d$ )

A *triangulation* of a finite set  $S$  of points in  $\mathbb{R}^d$ , for  $d \geq 2$ , is a finite collection  $\mathcal{T}(S)$  of  $d$ -simplices such that

- ① the union of all simplices equals  $CH(S)$ ,
  - ② every point of  $S$  is a vertex of at least one simplex,
  - ③ any pair of these simplices intersects in a common face (which, possibly, is the  $(-1)$ -face).
- 
- Sometimes, the first condition is relaxed and it is only demanded that all points of  $S$  form vertices of  $d$ -simplices, and that the boundary of the union of all simplices forms a  $(d - 1)$ -manifold, i.e., a simple curve in  $\mathbb{R}^2$  or a surface in  $\mathbb{R}^3$ .

## Definition 104 (Simplicial complex)

A *simplicial complex*  $\mathcal{K}$  is a set of simplices such that

- ① every face of a simplex of  $\mathcal{K}$  is also in  $\mathcal{K}$ ,
- ② the intersection of any pair of simplices  $\sigma_1, \sigma_2 \in \mathcal{K}$  forms a face of both  $\sigma_1$  and  $\sigma_2$ .



## What is a Good Triangulation?

- Let  $S$  be a set of  $n$  points in  $\mathbb{R}^2$ . Several options to demand additional properties for a triangulation of  $S$ .

## What is a Good Triangulation?

- Let  $S$  be a set of  $n$  points in  $\mathbb{R}^2$ . Several options to demand additional properties for a triangulation of  $S$ .

### Definition 105 (Locally Delaunay)

A triangulation  $\mathcal{T}(S)$  of  $S$  is *locally Delaunay* if for every pair of adjacent triangles  $\Delta(a, b, c)$  and  $\Delta(a, c, d)$  of  $\mathcal{T}(S)$  the Delaunay triangulation of  $a, b, c, d$  includes these two triangles.

## What is a Good Triangulation?

- Let  $S$  be a set of  $n$  points in  $\mathbb{R}^2$ . Several options to demand additional properties for a triangulation of  $S$ .

### Definition 105 (Locally Delaunay)

A triangulation  $\mathcal{T}(S)$  of  $S$  is *locally Delaunay* if for every pair of adjacent triangles  $\Delta(a, b, c)$  and  $\Delta(a, c, d)$  of  $\mathcal{T}(S)$  the Delaunay triangulation of  $a, b, c, d$  includes these two triangles.

### Lemma 106 (Fortune 1992)

A triangulation of  $S$  is a Delaunay triangulation of  $S$  if and only if it is locally Delaunay.



## What is a Good Triangulation?

- Let  $S$  be a set of  $n$  points in  $\mathbb{R}^2$ . Several options to demand additional properties for a triangulation of  $S$ .

### Definition 105 (Locally Delaunay)

A triangulation  $\mathcal{T}(S)$  of  $S$  is *locally Delaunay* if for every pair of adjacent triangles  $\Delta(a, b, c)$  and  $\Delta(a, c, d)$  of  $\mathcal{T}(S)$  the Delaunay triangulation of  $a, b, c, d$  includes these two triangles.

### Lemma 106 (Fortune 1992)

A triangulation of  $S$  is a Delaunay triangulation of  $S$  if and only if it is locally Delaunay.

### Lemma 107 (Sibson 1978)

The minimum internal angle of the triangles of  $\mathcal{DT}(S)$  is maximum over all triangulations.

## What is a Good Triangulation?

- Let  $S$  be a set of  $n$  points in  $\mathbb{R}^2$ . Several options to demand additional properties for a triangulation of  $S$ .

### Definition 105 (Locally Delaunay)

A triangulation  $\mathcal{T}(S)$  of  $S$  is *locally Delaunay* if for every pair of adjacent triangles  $\Delta(a, b, c)$  and  $\Delta(a, c, d)$  of  $\mathcal{T}(S)$  the Delaunay triangulation of  $a, b, c, d$  includes these two triangles.

### Lemma 106 (Fortune 1992)

A triangulation of  $S$  is a Delaunay triangulation of  $S$  if and only if it is locally Delaunay.

### Lemma 107 (Sibson 1978)

The minimum internal angle of the triangles of  $\mathcal{DT}(S)$  is maximum over all triangulations.

### Lemma 108 (Lambert 1994)

The (arithmetic) mean inradius of the triangles of  $\mathcal{DT}(S)$  is maximum over all triangulations.



# What is a Good Triangulation?

## Definition 109 (Hamiltonian triangulation)

A triangulation of  $S$  is a *Hamiltonian triangulation* if the dual graph of the triangulation admits a Hamiltonian path.



# What is a Good Triangulation?

## Definition 109 (Hamiltonian triangulation)

A triangulation of  $S$  is a *Hamiltonian triangulation* if the dual graph of the triangulation admits a Hamiltonian path.

## Lemma 110 (Arkin et al. 1996)

A Hamiltonian triangulation of  $S$  can be computed in  $O(n \log n)$  time.



# What is a Good Triangulation?

## Definition 109 (Hamiltonian triangulation)

A triangulation of  $S$  is a *Hamiltonian triangulation* if the dual graph of the triangulation admits a Hamiltonian path.

## Lemma 110 (Arkin et al. 1996)

A Hamiltonian triangulation of  $S$  can be computed in  $O(n \log n)$  time.

## Definition 111 (Minimum-weight triangulation)

A triangulation of  $S$  is a *minimum-weight triangulation* (MWT) if the sum of the lengths of the triangulation edges is minimum over all triangulations.

# What is a Good Triangulation?

## Definition 109 (Hamiltonian triangulation)

A triangulation of  $S$  is a *Hamiltonian triangulation* if the dual graph of the triangulation admits a Hamiltonian path.

## Lemma 110 (Arkin et al. 1996)

A Hamiltonian triangulation of  $S$  can be computed in  $O(n \log n)$  time.

## Definition 111 (Minimum-weight triangulation)

A triangulation of  $S$  is a *minimum-weight triangulation* (MWT) if the sum of the lengths of the triangulation edges is minimum over all triangulations.

## Theorem 112 (Mulzer&Rote 2006)

Computing a minimum-weight triangulation is  $\mathcal{NP}$ -hard.

## What is a Good Triangulation?

### Definition 109 (Hamiltonian triangulation)

A triangulation of  $S$  is a *Hamiltonian triangulation* if the dual graph of the triangulation admits a Hamiltonian path.

### Lemma 110 (Arkin et al. 1996)

A Hamiltonian triangulation of  $S$  can be computed in  $O(n \log n)$  time.

### Definition 111 (Minimum-weight triangulation)

A triangulation of  $S$  is a *minimum-weight triangulation* (MWT) if the sum of the lengths of the triangulation edges is minimum over all triangulations.

### Theorem 112 (Mulzer&Rote 2006)

Computing a minimum-weight triangulation is  $\mathcal{NP}$ -hard.

### Theorem 113 (Remy&Steger 2009)

For any  $\varepsilon > 0$ , a minimum-weight triangulation can be approximated with  $1 + \varepsilon$  as approximation ratio in quasi-polynomial time.



## Counting Triangulations

- No tight bounds are known for the minimum and the maximum number of different (straight-edge) triangulations that  $n$  points in 2D may admit.
- For 20 points, the best known minimum is 20 662 980, and the best known maximum is 918 462 742 512 [Aichholzer et alii, 2001–2003].

## Counting Triangulations

- No tight bounds are known for the minimum and the maximum number of different (straight-edge) triangulations that  $n$  points in 2D may admit.
- For 20 points, the best known minimum is 20 662 980, and the best known maximum is 918 462 742 512 [Aichholzer et alii, 2001–2003].

### Lemma 114 (Goldbach&Euler 1751, Lamé 1838)

If  $(n + 2)$  points are in convex position then the number of different triangulations is given by the  $n$ -th Catalan number  $C_n$ . (We have  $C_n = \frac{(2n)!}{(n+1)! \cdot n!}$ .)

# Counting Triangulations

- No tight bounds are known for the minimum and the maximum number of different (straight-edge) triangulations that  $n$  points in 2D may admit.
- For 20 points, the best known minimum is 20 662 980, and the best known maximum is 918 462 742 512 [Aichholzer et alii, 2001–2003].

## Lemma 114 (Goldbach&Euler 1751, Lamé 1838)

If  $(n + 2)$  points are in convex position then the number of different triangulations is given by the  $n$ -th Catalan number  $C_n$ . (We have  $C_n = \frac{(2n)!}{(n+1)! \cdot n!}$ .)

## Lemma 115 (Sharir&Sheffer 2009)

A set of  $n$  points in the plane admits at most  $30^n$  different triangulations.



# Counting Triangulations

- No tight bounds are known for the minimum and the maximum number of different (straight-edge) triangulations that  $n$  points in 2D may admit.
- For 20 points, the best known minimum is 20 662 980, and the best known maximum is 918 462 742 512 [Aichholzer et alii, 2001–2003].

## Lemma 114 (Goldbach&Euler 1751, Lamé 1838)

If  $(n + 2)$  points are in convex position then the number of different triangulations is given by the  $n$ -th Catalan number  $C_n$ . (We have  $C_n = \frac{(2n)!}{(n+1)! \cdot n!}$ .)

## Lemma 115 (Sharir&Sheffer 2009)

A set of  $n$  points in the plane admits at most  $30^n$  different triangulations.

## Lemma 116 (Dumitrescu et alii 2010)

There exist sets of  $n$  points in the plane which admit at least  $\Omega(8.65^n)$  different triangulations.



# Triangulations are Related via Edge Flips

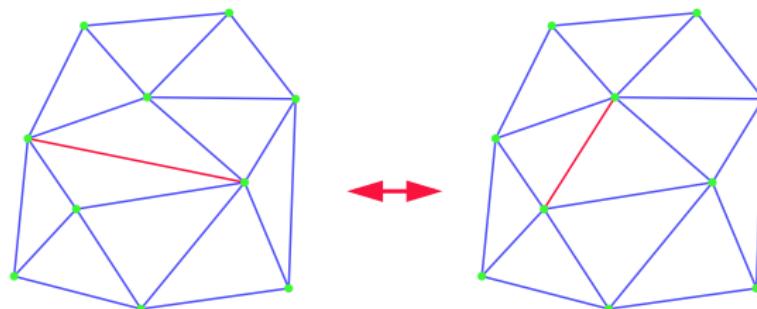
## Definition 117 (Edge flip, Dt.: Kantenaustausch)

An *edge flip* is a local operation on a triangulation that replaces one diagonal of a convex quadrilateral (formed by two neighboring triangles) with the other diagonal.

# Triangulations are Related via Edge Flips

## Definition 117 (Edge flip, Dt.: Kantenaustausch)

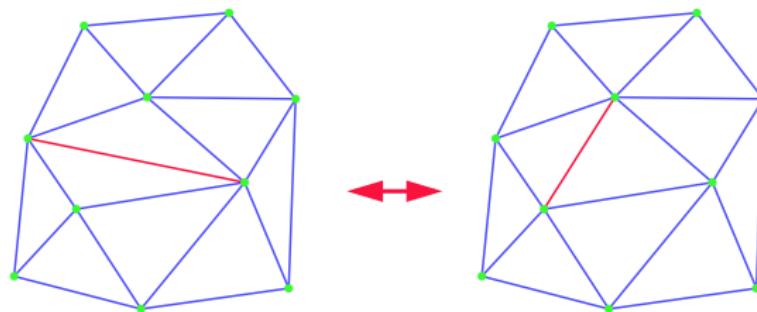
An *edge flip* is a local operation on a triangulation that replaces one diagonal of a convex quadrilateral (formed by two neighboring triangles) with the other diagonal.



# Triangulations are Related via Edge Flips

## Definition 117 (Edge flip, Dt.: Kantenaustausch)

An *edge flip* is a local operation on a triangulation that replaces one diagonal of a convex quadrilateral (formed by two neighboring triangles) with the other diagonal.



## Lemma 118 (Bern&Eppstein 1992)

$O(n^2)$  edge-flipping operations suffice to transform any triangulation of  $n$  points into a Delaunay triangulation.

## Triangulations

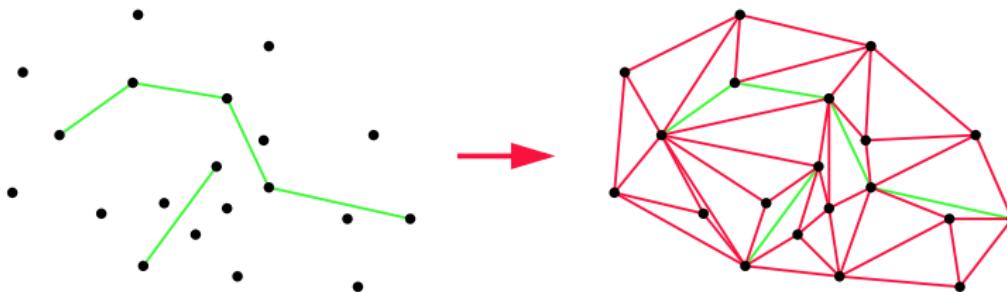
- Introduction
- **Constrained Triangulations**
- Computing Constrained Triangulations
- Triangulations in 3D
- Applications of Triangulations

# Constrained Triangulation

## Definition 119 (Constrained triangulation)

A triangulation  $\mathcal{T}$  forms a *constrained triangulation* of an admissible set  $S$  of vertices and line segments if

- ① the vertices of  $S$  are the nodes of  $\mathcal{T}$ , and
- ② all line segments of  $S$  are edges of  $\mathcal{T}$ .

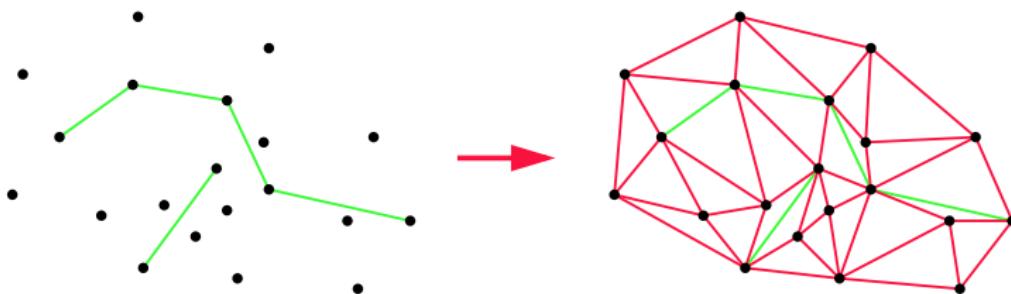


# Constrained Delaunay Triangulation

## Definition 120 (Constrained Delaunay triangulation)

A triangulation  $\mathcal{T}$  forms a *constrained Delaunay triangulation* (CDT) of an admissible set  $S$  of vertices and line segments if

- ①  $\mathcal{T}$  is a constrained triangulation of  $S$ , and
- ② no triangle  $\Delta$  of  $\mathcal{T}$  contains a vertex of  $S$  that is visible from  $\Delta$  in its circumcircle.

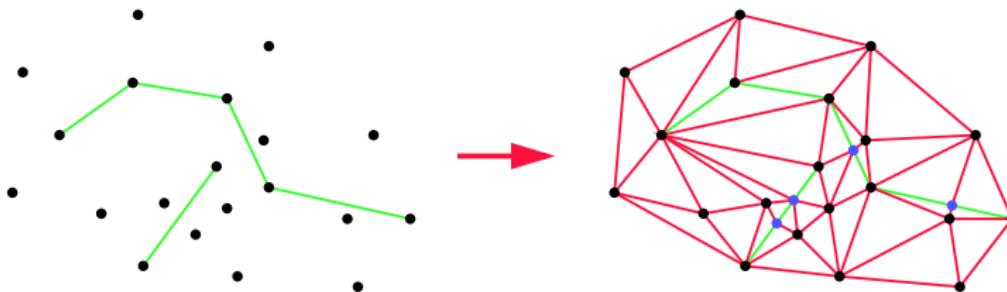


# Conforming Triangulation

## Definition 121 (Conforming triangulation)

A triangulation  $\mathcal{T}$  forms a *conforming triangulation* of an admissible set  $S$  of vertices and line segments if

- ① all vertices of  $S$  are nodes of  $\mathcal{T}$  (but Steiner points are allowed), and
- ② all line segments of  $S$  are represented by unions of edges of  $\mathcal{T}$ .



## Triangulations

- Introduction
- Constrained Triangulations
- Computing Constrained Triangulations
  - Facts and State-of-the-Art
  - Polygon Triangulation via Ear-Clipping
- Triangulations in 3D
- Applications of Triangulations

# Computing Constrained Triangulations: Polygons

## Definition 122 (Diagonal)

The line segment  $p_i p_j$  forms a *diagonal* of the polygon  $\mathcal{P} = [p_1, p_2, \dots, p_n]$  if  $p_i p_j$  lies completely in the interior of  $\mathcal{P}$ , except for  $p_i$  and  $p_j$ .



# Computing Constrained Triangulations: Polygons

## Definition 122 (Diagonal)

The line segment  $p_i p_j$  forms a *diagonal* of the polygon  $\mathcal{P} = [p_1, p_2, \dots, p_n]$  if  $p_i p_j$  lies completely in the interior of  $\mathcal{P}$ , except for  $p_i$  and  $p_j$ .

## Lemma 123

Every simple polygon with  $n \geq 4$  vertices contains a diagonal.

# Computing Constrained Triangulations: Polygons

## Definition 122 (Diagonal)

The line segment  $p_i p_j$  forms a *diagonal* of the polygon  $\mathcal{P} = [p_1, p_2, \dots, p_n]$  if  $p_i p_j$  lies completely in the interior of  $\mathcal{P}$ , except for  $p_i$  and  $p_j$ .

## Lemma 123

Every simple polygon with  $n \geq 4$  vertices contains a diagonal.

## Corollary 124

Every simple polygon with  $n$  vertices can be partitioned into  $n - 2$  triangles by inserting  $n - 3$  appropriate diagonals.

# Computing Constrained Triangulations: Polygons

## Definition 122 (Diagonal)

The line segment  $p_i p_j$  forms a *diagonal* of the polygon  $\mathcal{P} = [p_1, p_2, \dots, p_n]$  if  $p_i p_j$  lies completely in the interior of  $\mathcal{P}$ , except for  $p_i$  and  $p_j$ .

## Lemma 123

Every simple polygon with  $n \geq 4$  vertices contains a diagonal.

## Corollary 124

Every simple polygon with  $n$  vertices can be partitioned into  $n - 2$  triangles by inserting  $n - 3$  appropriate diagonals.

## Lemma 125

A regularization procedure can be used to partition a simple polygon in  $O(n \log n)$  into a set of monotone polygons.



# Computing Constrained Triangulations: Polygons

## Definition 122 (Diagonal)

The line segment  $p_i p_j$  forms a *diagonal* of the polygon  $\mathcal{P} = [p_1, p_2, \dots, p_n]$  if  $p_i p_j$  lies completely in the interior of  $\mathcal{P}$ , except for  $p_i$  and  $p_j$ .

## Lemma 123

Every simple polygon with  $n \geq 4$  vertices contains a diagonal.

## Corollary 124

Every simple polygon with  $n$  vertices can be partitioned into  $n - 2$  triangles by inserting  $n - 3$  appropriate diagonals.

## Lemma 125

A regularization procedure can be used to partition a simple polygon in  $O(n \log n)$  into a set of monotone polygons.

## Corollary 126

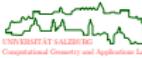
A simple polygon can be triangulated in  $O(n \log n)$  time.



# Computing Constrained Triangulations: Polygons

## Theorem 127 (Chew 1989)

A constrained Delaunay triangulation of an admissible set  $S$  can be computed in optimal  $O(n \log n)$  time.



# Computing Constrained Triangulations: Polygons

## Theorem 127 (Chew 1989)

A constrained Delaunay triangulation of an admissible set  $S$  can be computed in optimal  $O(n \log n)$  time.

## Theorem 128 (Chazelle 1991)

A simple polygon can be triangulated in optimal  $O(n)$  time.

# Computing Constrained Triangulations: Polygons

## Theorem 127 (Chew 1989)

A constrained Delaunay triangulation of an admissible set  $S$  can be computed in optimal  $O(n \log n)$  time.

## Theorem 128 (Chazelle 1991)

A simple polygon can be triangulated in optimal  $O(n)$  time.

## Corollary 129 (Chazelle 1991)

In  $O(n)$  time we can check whether a polygon is simple.

# Computing Constrained Triangulations: Polygons

## Theorem 127 (Chew 1989)

A constrained Delaunay triangulation of an admissible set  $S$  can be computed in optimal  $O(n \log n)$  time.

## Theorem 128 (Chazelle 1991)

A simple polygon can be triangulated in optimal  $O(n)$  time.

## Corollary 129 (Chazelle 1991)

In  $O(n)$  time we can check whether a polygon is simple.

## Theorem 130 (Chin&Wang 1998)

A constrained Delaunay triangulation of a simple polygon can be computed in optimal  $O(n)$  time.

# Computing Constrained Triangulations: Polygons

## Theorem 127 (Chew 1989)

A constrained Delaunay triangulation of an admissible set  $S$  can be computed in optimal  $O(n \log n)$  time.

## Theorem 128 (Chazelle 1991)

A simple polygon can be triangulated in optimal  $O(n)$  time.

## Corollary 129 (Chazelle 1991)

In  $O(n)$  time we can check whether a polygon is simple.

## Theorem 130 (Chin&Wang 1998)

A constrained Delaunay triangulation of a simple polygon can be computed in optimal  $O(n)$  time.

- The algorithms by Chazelle and Chin&Wang are far too complicated to be of practical relevance; randomized, expected linear-time algorithms and pretty good heuristics that achieve close-to-linear time in practice do exist.

# Polygon Triangulation via Ear-Clipping

## Definition 131 (Ear)

Three consecutive vertices  $(p_{i-1}, p_i, p_{i+1})$  of a simple polygon  $\mathcal{P}$  form an *ear* of  $\mathcal{P}$  if  $p_{i-1}p_{i+1}$  constitutes a diagonal of  $\mathcal{P}$ .

# Polygon Triangulation via Ear-Clipping

## Definition 131 (Ear)

Three consecutive vertices  $(p_{i-1}, p_i, p_{i+1})$  of a simple polygon  $\mathcal{P}$  form an **ear** of  $\mathcal{P}$  if  $p_{i-1}p_{i+1}$  constitutes a diagonal of  $\mathcal{P}$ .

## Lemma 132 (Meisters 1975)

Every simple polygon with four or more vertices has at least two non-overlapping ears.

## Definition 131 (Ear)

Three consecutive vertices  $(p_{i-1}, p_i, p_{i+1})$  of a simple polygon  $\mathcal{P}$  form an *ear* of  $\mathcal{P}$  if  $p_{i-1}p_{i+1}$  constitutes a diagonal of  $\mathcal{P}$ .

## Lemma 132 (Meisters 1975)

Every simple polygon with four or more vertices has at least two non-overlapping ears.

- Simple triangulation algorithm:
  - Find an ear of  $\mathcal{P}$  and clip it.
  - Repeat the ear clipping until only a triangle is left.
- An ear-clipping operation transforms an  $n$ -gon into an  $(n - 1)$ -gon.

# Polygon Triangulation via Ear-Clipping

## Definition 131 (Ear)

Three consecutive vertices  $(p_{i-1}, p_i, p_{i+1})$  of a simple polygon  $\mathcal{P}$  form an *ear* of  $\mathcal{P}$  if  $p_{i-1}p_{i+1}$  constitutes a diagonal of  $\mathcal{P}$ .

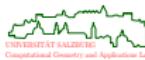
## Lemma 132 (Meisters 1975)

Every simple polygon with four or more vertices has at least two non-overlapping ears.

- Simple triangulation algorithm:
  - Find an ear of  $\mathcal{P}$  and clip it.
  - Repeat the ear clipping until only a triangle is left.
- An ear-clipping operation transforms an  $n$ -gon into an  $(n - 1)$ -gon.

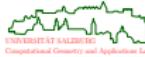
## Warning

Real-world data tends to exhibit all types of degeneracies and should not be assumed to be “simple” and clean.



## Animation of Ear Clipping

- Find an ear, and clip it. Keep clipping ears, until triangulation is finished.



# Polygon Triangulation via Ear-Clipping

- Complexity of naïve algorithm:  $O(n^3)$ .
  - $O(n)$  time to check validity of an ear.
  - $O(n)$  many checks to find next ear.
  - $O(n)$  many ears needed.

## Polygon Triangulation via Ear-Clipping

- Complexity of naïve algorithm:  $O(n^3)$ .
  - $O(n)$  time to check validity of an ear.
  - $O(n)$  many checks to find next ear.
  - $O(n)$  many ears needed.
- Observation: The clipping of one ear affects the earity of at most two other triples of vertices of  $\mathcal{P}$ .

# Polygon Triangulation via Ear-Clipping

- Complexity of naïve algorithm:  $O(n^3)$ .
  - $O(n)$  time to check validity of an ear.
  - $O(n)$  many checks to find next ear.
  - $O(n)$  many ears needed.
- Observation: The clipping of one ear affects the earity of at most two other triples of vertices of  $\mathcal{P}$ .
- Thus, the overall complexity is reduced to  $O(n^2)$ .

## Lemma 133

Ear clipping computes a triangulation of a simple  $n$ -gon in  $O(n^2)$  time.

## Lemma 134

Three consecutive vertices  $(p_{i-1}, p_i, p_{i+1})$  of  $\mathcal{P}$  form an ear of  $\mathcal{P}$  if

- ①  $p_i$  is a convex vertex,
- ② the triangle  $\Delta(p_{i-1}, p_i, p_{i+1})$  contains no reflex vertex of  $\mathcal{P}$ , except for  $p_{i-1}$  and  $p_{i+1}$  if they are reflex.



## Lemma 134

Three consecutive vertices  $(p_{i-1}, p_i, p_{i+1})$  of  $\mathcal{P}$  form an ear of  $\mathcal{P}$  if

- ①  $p_i$  is a convex vertex,
- ② the triangle  $\Delta(p_{i-1}, p_i, p_{i+1})$  contains no reflex vertex of  $\mathcal{P}$ , except for  $p_{i-1}$  and  $p_{i+1}$  if they are reflex.

## Corollary 135

Ear clipping computes a triangulation of a simple  $n$ -gon in  $O(n \cdot r)$  time, where  $r$  is the number of its reflex vertices.

# Polygon Triangulation via Ear-Clipping

## Lemma 134

Three consecutive vertices  $(p_{i-1}, p_i, p_{i+1})$  of  $\mathcal{P}$  form an ear of  $\mathcal{P}$  if

- ①  $p_i$  is a convex vertex,
- ② the triangle  $\Delta(p_{i-1}, p_i, p_{i+1})$  contains no reflex vertex of  $\mathcal{P}$ , except for  $p_{i-1}$  and  $p_{i+1}$  if they are reflex.

## Corollary 135

Ear clipping computes a triangulation of a simple  $n$ -gon in  $O(n \cdot r)$  time, where  $r$  is the number of its reflex vertices.

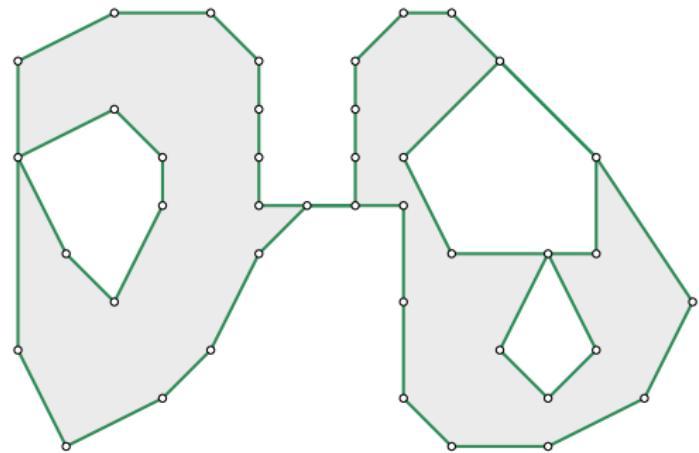
- [Held 2001]: A triangulation algorithm based on ear-clipping and geometric hashing can be engineered to run in near-linear time, beating implementations of theoretically better algorithms on thousands of synthetic and real-world data sets.  
→ “Fast Industrial-Strength Triangulation” (FIST).

## Fast Industrial-Strength Triangulation (FIST)

- Triangulates polygons with holes in 2D and 3D,
  - based on ear-clipping and
  - multi-level geometric hashing to speed up computation.

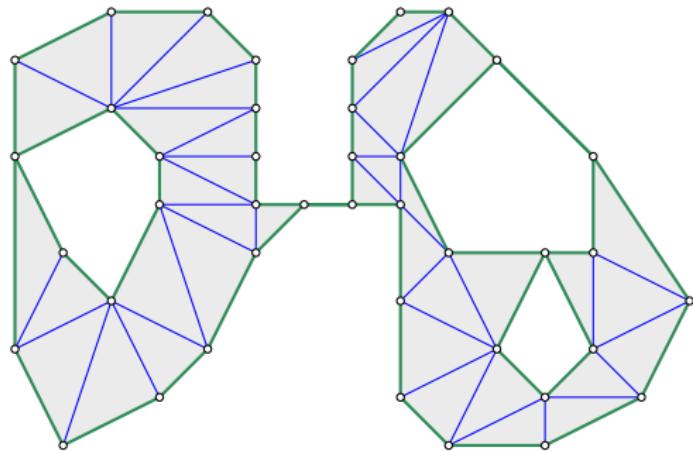
## Fast Industrial-Strength Triangulation (FIST)

- Triangulates polygons with holes in 2D and 3D,
  - based on ear-clipping and
  - multi-level geometric hashing to speed up computation.
- Handles
  - degenerate input,



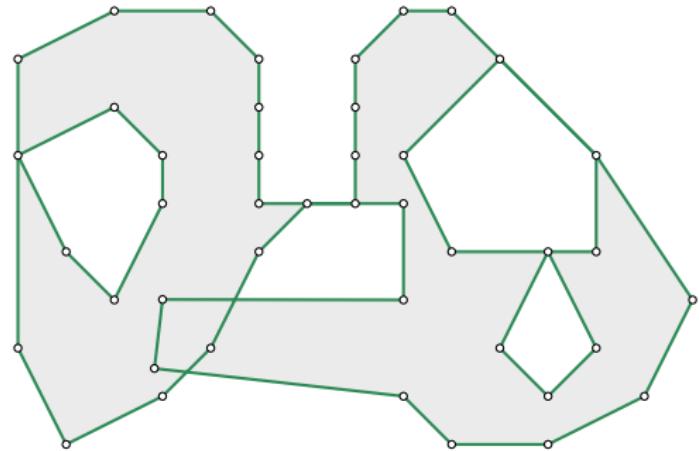
## Fast Industrial-Strength Triangulation (FIST)

- Triangulates polygons with holes in 2D and 3D,
  - based on ear-clipping and
  - multi-level geometric hashing to speed up computation.
- Handles
  - degenerate input,



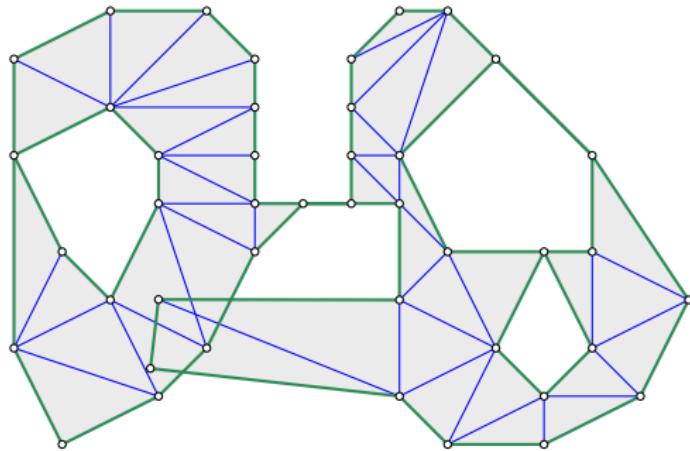
# Fast Industrial-Strength Triangulation (FIST)

- Triangulates polygons with holes in 2D and 3D,
  - based on ear-clipping and
  - multi-level geometric hashing to speed up computation.
- Handles
  - degenerate input,
  - self-overlapping input,



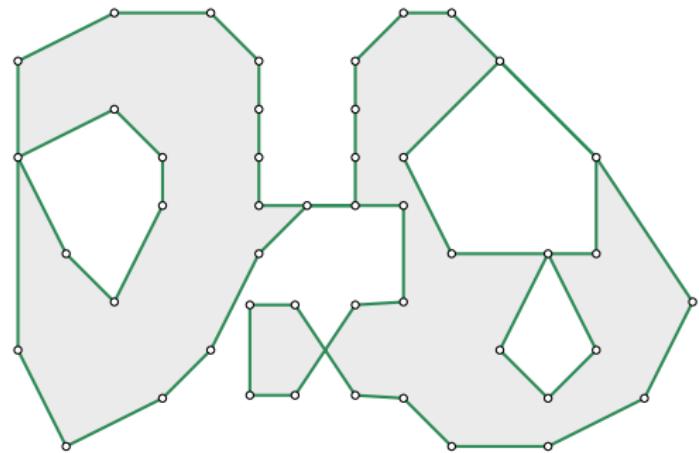
## Fast Industrial-Strength Triangulation (FIST)

- Triangulates polygons with holes in 2D and 3D,
  - based on ear-clipping and
  - multi-level geometric hashing to speed up computation.
- Handles
  - degenerate input,
  - self-overlapping input,



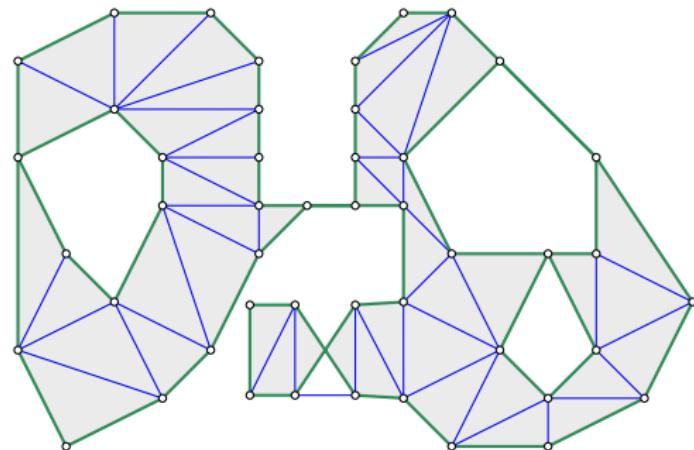
# Fast Industrial-Strength Triangulation (FIST)

- Triangulates polygons with holes in 2D and 3D,
  - based on ear-clipping and
  - multi-level geometric hashing to speed up computation.
- Handles
  - degenerate input,
  - self-overlapping input,
  - self-intersecting input.



## Fast Industrial-Strength Triangulation (FIST)

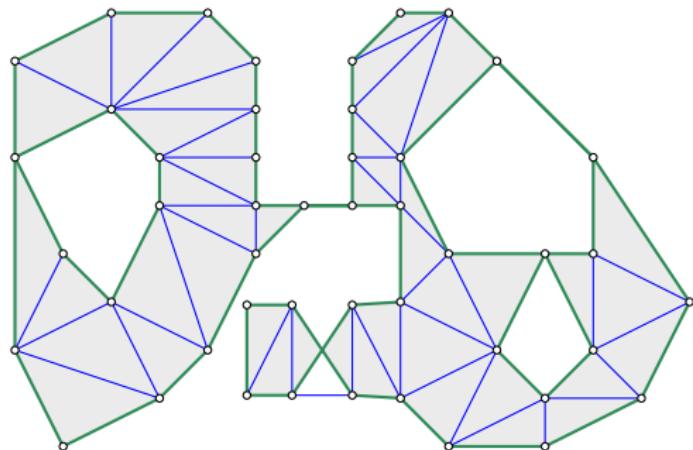
- Triangulates polygons with holes in 2D and 3D,
  - based on ear-clipping and
  - multi-level geometric hashing to speed up computation.
- Handles
  - degenerate input,
  - self-overlapping input,
  - self-intersecting input.



## Fast Industrial-Strength Triangulation (FIST)

- Triangulates polygons with holes in 2D and 3D,
  - based on ear-clipping and
  - multi-level geometric hashing to speed up computation.

- Handles
  - degenerate input,
  - self-overlapping input,
  - self-intersecting input.



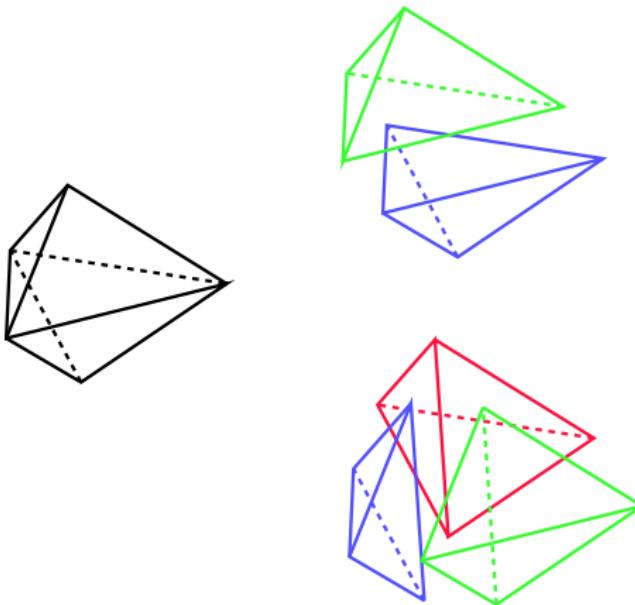
- No Delaunay triangulation, but heuristics to generate “decent” triangles.
- Typical applications in industry: triangulation of (very) large GIS datasets, triangulation of “planar” faces of 3D models.

## Triangulations

- Introduction
- Constrained Triangulations
- Computing Constrained Triangulations
- **Triangulations in 3D**
- Applications of Triangulations

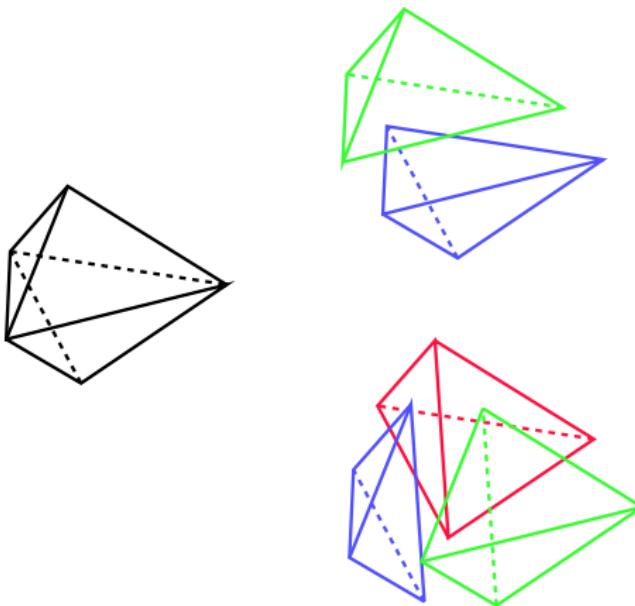
## Caveats for 3D Triangulations: Number of Tetrahedra

- The number of tetrahedra may vary.



## Caveats for 3D Triangulations: Number of Tetrahedra

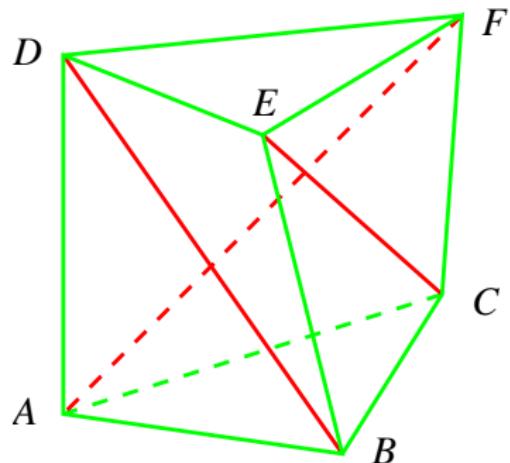
- The number of tetrahedra may vary.



- Up to  $O(n^2)$  many tetrahedra may occur for  $n$  input points.

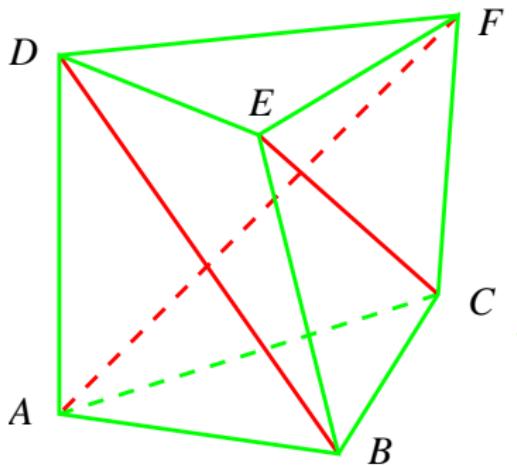
## Caveats for 3D Triangulations: Schönhardt Polyhedron

- Not every polyhedron can be triangulated: The triangle  $\Delta(D, E, F)$  of Schönhardt's polyhedron is rotated relative to  $\Delta(A, B, C)$ , causing the three red edges  $BD$ ,  $CE$  and  $AF$  to become reflex.



## Caveats for 3D Triangulations: Schönhardt Polyhedron

- Not every polyhedron can be triangulated: The triangle  $\Delta(D, E, F)$  of Schönhardt's polyhedron is rotated relative to  $\Delta(A, B, C)$ , causing the three red edges  $BD$ ,  $CE$  and  $AF$  to become reflex.



### Theorem 136 (Ruppert & Seidel 1992)

It is  $\mathcal{NP}$ -complete to determine whether a polyhedron requires Steiner points for triangulation. (This result holds even for star-shaped polyhedra which can trivially be triangulated with at most one Steiner point!)

## Caveats for 3D Triangulations: Many Steiner Points Required

### Theorem 137 (Chazelle 1984)

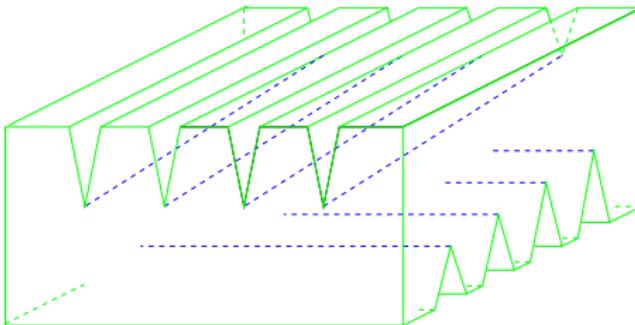
There exist polyhedra that require  $\Omega(n^2)$  Steiner points.



# Caveats for 3D Triangulations: Many Steiner Points Required

## Theorem 137 (Chazelle 1984)

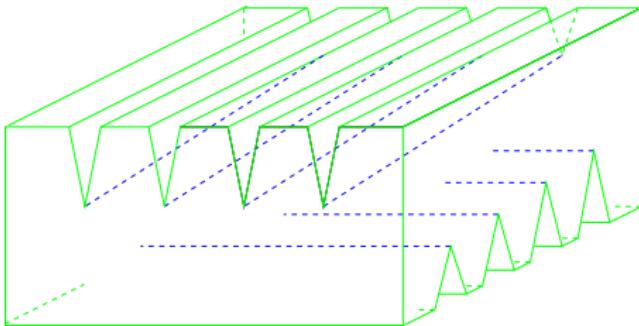
There exist polyhedra that require  $\Omega(n^2)$  Steiner points.



## Caveats for 3D Triangulations: Many Steiner Points Required

### Theorem 137 (Chazelle 1984)

There exist polyhedra that require  $\Omega(n^2)$  Steiner points.



### Theorem 138 (Chazelle&Palios 1990)

Every simple polyhedron with  $n$  vertices and  $r$  reflex edges can be triangulated using  $O(n + r^2)$  Steiner points and a total of  $O(n + r^2)$  tetrahedra.

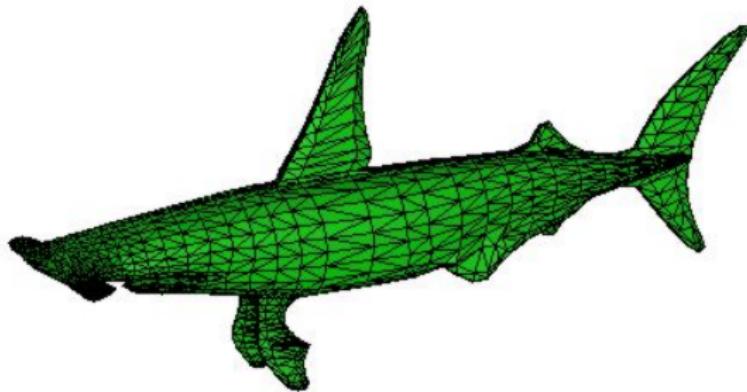
## Triangulations

- Introduction
- Constrained Triangulations
- Computing Constrained Triangulations
- Triangulations in 3D
- Applications of Triangulations
  - Rendering
  - Triangulated Irregular Network
  - Visibility Determination
  - Topologically Consistent Watermarking



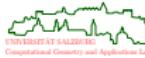
## Rendering

- Graphics hardware is best at handling triangles rather than more general geometric primitives. Thus, the surfaces of 3D models need to be triangulated.
- Note: 3D models used for graphics purposes tend to exhibit all types of degeneracies!



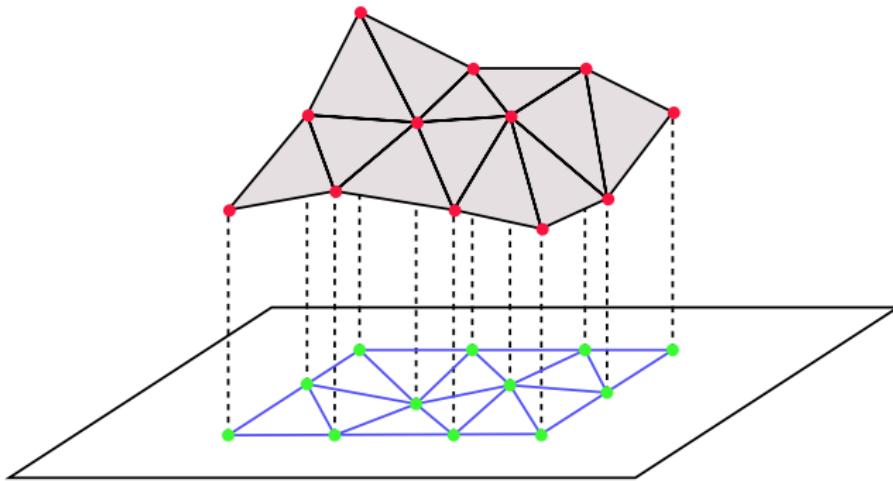
# Triangulated Irregular Network

- Given is a set of sample points on a terrain. How can we model the actual surface by interpolating those points?



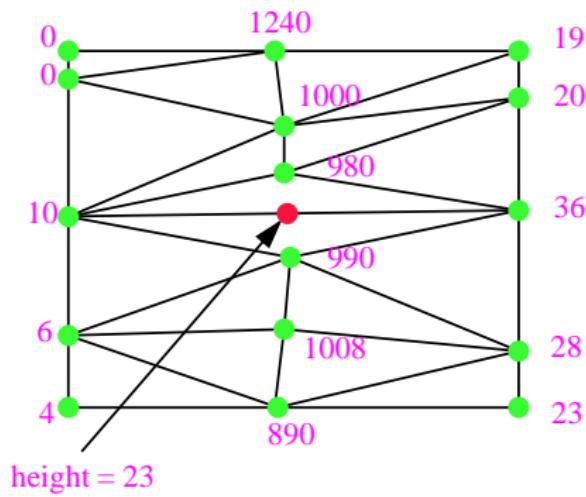
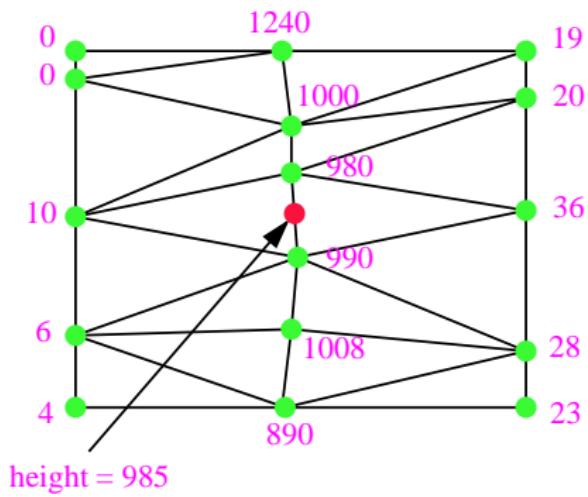
# Triangulated Irregular Network

- Given is a set of sample points on a terrain. How can we model the actual surface by interpolating those points?
- Natural approach: Project the points onto 2D (by discarding their z-coordinate), compute a triangulation  $\mathcal{T}$  of the projected points, and lift  $\mathcal{T}$  back to 3D.
- Known to the GIS community as TIN (triangulated irregular network).



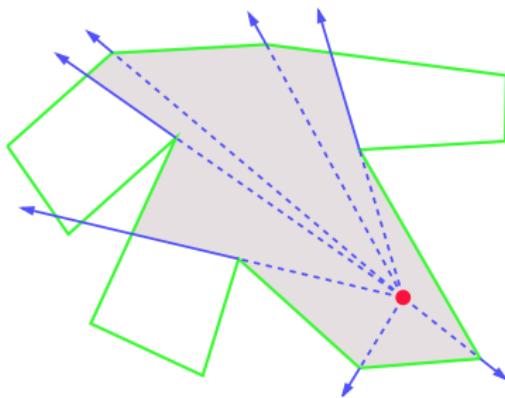
# Triangulated Irregular Network

- Note: The topology of the terrain depends heavily on the 2D triangulation!



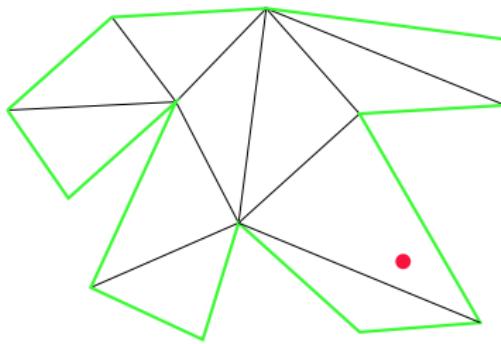
# Visibility Determination Within Triangulated Environments

- Which portion of the green polygon is visible from the red point?



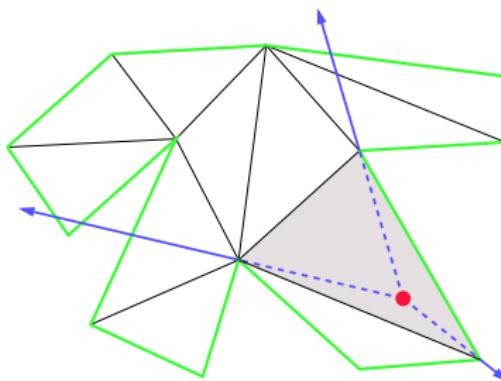
# Visibility Determination Within Triangulated Environments

- Which portion of the green polygon is visible from the red point?
- We start with an arbitrary triangulation of the polygon.



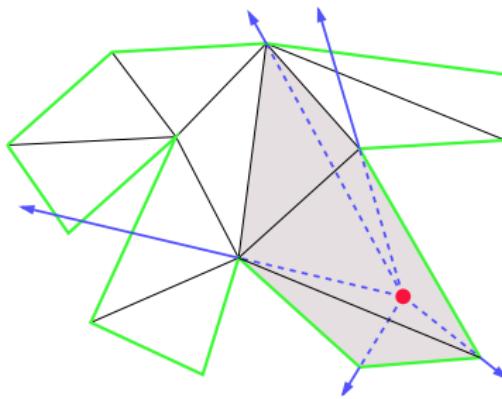
# Visibility Determination Within Triangulated Environments

- Which portion of the green polygon is visible from the red point?
- We start with an arbitrary triangulation of the polygon.
- The triangle that contains the red point is illuminated.



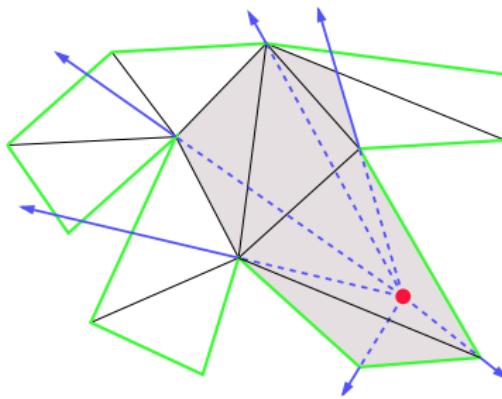
# Visibility Determination Within Triangulated Environments

- Which portion of the green polygon is visible from the red point?
- We start with an arbitrary triangulation of the polygon.
- The triangle that contains the red point is illuminated.
- Neighboring triangles are illuminated and new visibility rays generated.



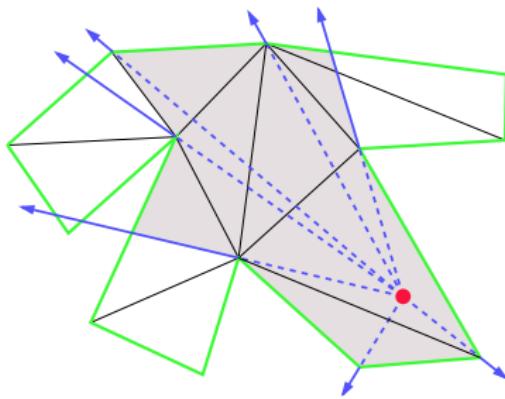
# Visibility Determination Within Triangulated Environments

- Which portion of the green polygon is visible from the red point?
- We start with an arbitrary triangulation of the polygon.
- The triangle that contains the red point is illuminated.
- Neighboring triangles are illuminated and new visibility rays generated.



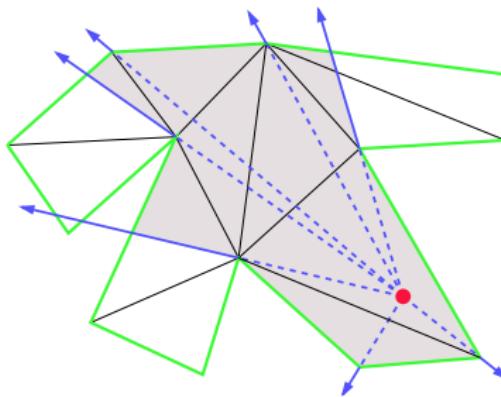
# Visibility Determination Within Triangulated Environments

- Which portion of the green polygon is visible from the red point?
- We start with an arbitrary triangulation of the polygon.
- The triangle that contains the red point is illuminated.
- Neighboring triangles are illuminated and new visibility rays generated.



## Visibility Determination Within Triangulated Environments

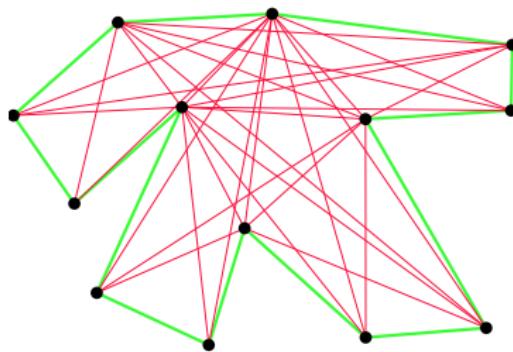
- Which portion of the green polygon is visible from the red point?
- We start with an arbitrary triangulation of the polygon.
- The triangle that contains the red point is illuminated.
- Neighboring triangles are illuminated and new visibility rays generated.
- All triangles which are at least partially visible have been traversed.



# Visibility Graph

## Definition 139 (Visibility graph, Dt.: Sichtbarkeitsgraph)

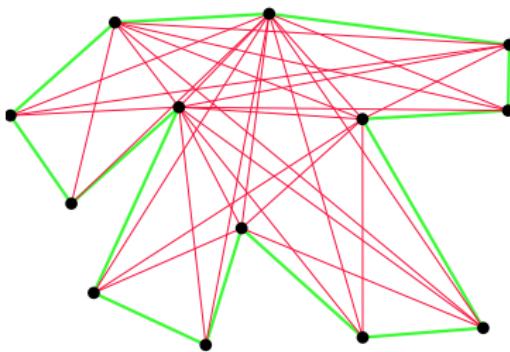
The *visibility graph* inside an  $n$ -gon  $\mathcal{P}$  consists of the vertices of  $\mathcal{P}$  as nodes connected by edges if the vertices can see each other.



## Visibility Graph

### Definition 139 (Visibility graph, Dt.: Sichtbarkeitsgraph)

The *visibility graph* inside an  $n$ -gon  $\mathcal{P}$  consists of the vertices of  $\mathcal{P}$  as nodes connected by edges if the vertices can see each other.

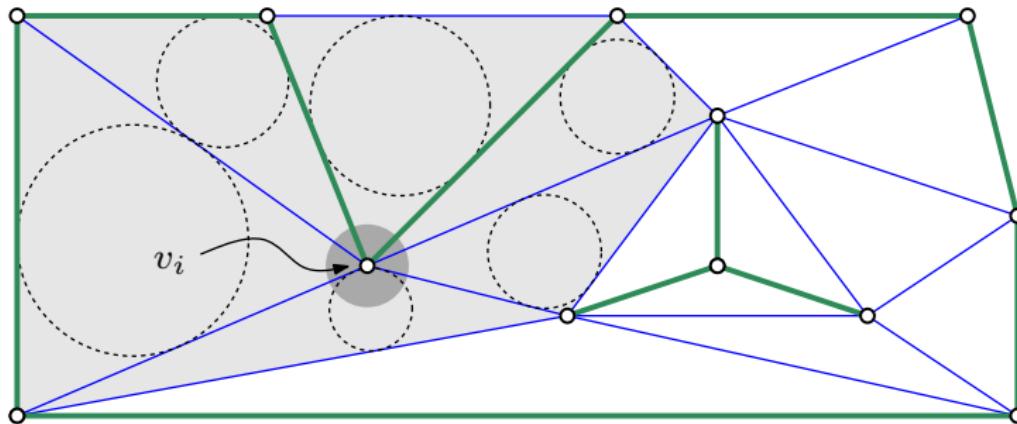


### Theorem 140 (Gosh&Mount 1991)

The full visibility graph inside an  $n$ -gon can be computed in time  $O(|E| + n \log n)$ , where  $|E|$  is the size of the visibility graph.

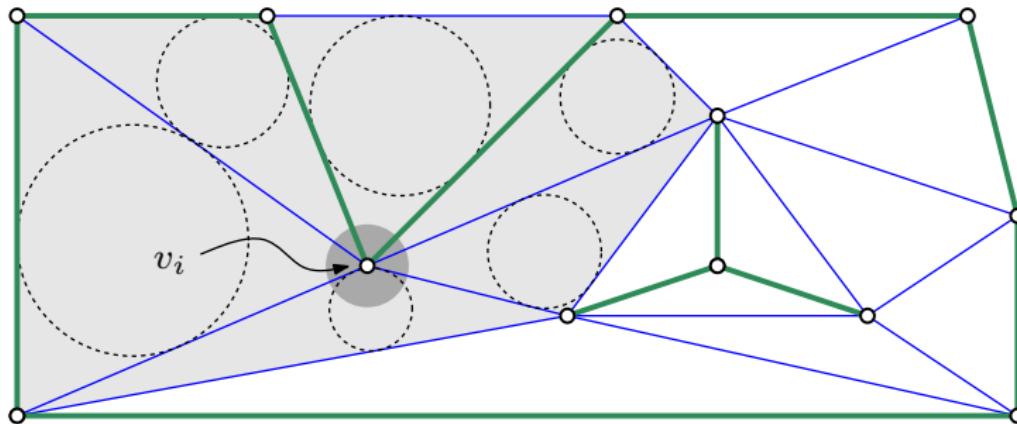
# Topology-Preserving Watermarking of Vector Graphics

- [Huber et alii 2013] compute for each vertex a disk-shaped maximum perturbation region (MPR), based on the radii of the inscribed circles of a constrained triangulation of the input.
- Perturbing the vertices within their MPRs causes the edges to stay within their hoses and allows to preserve the input topology.



# Topology-Preserving Watermarking of Vector Graphics

- [Huber et alii 2013] compute for each vertex a disk-shaped maximum perturbation region (MPR), based on the radii of the inscribed circles of a constrained triangulation of the input.
- Perturbing the vertices within their MPRs causes the edges to stay within their hoses and allows to preserve the input topology.
- This scheme can be extended to 3D.



## Robustness Issues

- Introduction to Robustness Problems
  - Computing with Floating-Point Arithmetic
  - Manifestations of Robustness Problems
  - Robustness and Stability
- Approaches to Achieving Robustness
  - Exact Arithmetic
  - Exact Geometric Computing
  - Symbolic Perturbation
- Improving the Reliability of FP-Code
  - Standard Tricks for Achieving Reliable FP-Code
  - Topology-Oriented Computation
  - Relaxation of Epsilon Thresholds
  - Desperate Mode
  - MPFR Library
- Industrial Applications
  - Industrial Requirements
  - Experimental Results for Triangulations
  - Experimental Results for Voronoi Diagrams
- Using FP-Arithmetic to Implement the Turing Test

## Robustness Issues

- Introduction to Robustness Problems
  - Computing with Floating-Point Arithmetic
  - Manifestations of Robustness Problems
  - Robustness and Stability
- Approaches to Achieving Robustness
- Improving the Reliability of FP-Code
- Industrial Applications
- Using FP-Arithmetic to Implement the Turing Test

## Floating-Point Errors

- Voronoi diagrams of points and line segments are easy to understand, but notorious for being difficult to implement reliably.
- Similar robustness problems arise for many algorithms of computational geometry.



## Floating-Point Errors

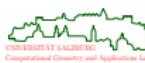
- Voronoi diagrams of points and line segments are easy to understand, but notorious for being difficult to implement reliably.
- Similar robustness problems arise for many algorithms of computational geometry.
- Computers employ floating-point (fp) arithmetic to perform real arithmetic.
- No matter how many bits are used, fp-arithmetic represents a number by a fixed-length binary mantissa and an exponent of fixed size.
- Thus, only a finite number of values within a finite sub-interval of  $\mathbb{R}$  can be represented accurately; all other values have to be rounded to the closest number that is representable.



## Floating-Point Errors

- Voronoi diagrams of points and line segments are easy to understand, but notorious for being difficult to implement reliably.
- Similar robustness problems arise for many algorithms of computational geometry.
- Computers employ floating-point (fp) arithmetic to perform real arithmetic.
- No matter how many bits are used, fp-arithmetic represents a number by a fixed-length binary mantissa and an exponent of fixed size.
- Thus, only a finite number of values within a finite sub-interval of  $\mathbb{R}$  can be represented accurately; all other values have to be rounded to the closest number that is representable.
- More generally, there are two sources of error for fp-computations: input error and round-off error.

**Input error:** It is well-known that  $\frac{1}{3}$  cannot be represented by a finite sum of powers of 10. Similarly, 0.1 cannot be represented by a finite sum of powers of 2!



## Floating-Point Errors

- Voronoi diagrams of points and line segments are easy to understand, but notorious for being difficult to implement reliably.
- Similar robustness problems arise for many algorithms of computational geometry.
- Computers employ floating-point (fp) arithmetic to perform real arithmetic.
- No matter how many bits are used, fp-arithmetic represents a number by a fixed-length binary mantissa and an exponent of fixed size.
- Thus, only a finite number of values within a finite sub-interval of  $\mathbb{R}$  can be represented accurately; all other values have to be rounded to the closest number that is representable.
- More generally, there are two sources of error for fp-computations: input error and round-off error.

**Input error:** It is well-known that  $\frac{1}{3}$  cannot be represented by a finite sum of powers of 10. Similarly, 0.1 cannot be represented by a finite sum of powers of 2!

**Round-off error:** It arises from rounding results of fp-computations during an algorithm. E.g.,  $\sqrt{2}$  cannot be represented exactly since  $\sqrt{2}$  is an irrational number.



## Machine Precision

- The round-off error is bounded in terms of the *machine precision*,  $\varepsilon$ , which is the smallest value satisfying

$$|fp(a \circ b) - (a \circ b)| \leq \varepsilon |a \circ b|$$

for all fp-numbers  $a, b$  and any of the four operations  $+, -, \cdot, /$  instead of  $\circ$ , for which  $a \circ b$  does not cause an underflow or an overflow.

## Machine Precision

- The round-off error is bounded in terms of the *machine precision*,  $\varepsilon$ , which is the smallest value satisfying

$$|fp(a \circ b) - (a \circ b)| \leq \varepsilon |a \circ b|$$

for all all fp-numbers  $a, b$  and any of the four operations  $+, -, \cdot, /$  instead of  $\circ$ , for which  $a \circ b$  does not cause an underflow or an overflow.

- On IEEE-754 machines,  $\varepsilon = 2^{-23} \approx 1.19 \cdot 10^{-7}$  for floats, and  $\varepsilon = 2^{-52} \approx 2.22 \cdot 10^{-16}$  for doubles.
- On some exotic platform,  $\varepsilon$  can be determined approximately by finding the smallest positive value  $x$  such that  $1 + x \neq 1$ .

## Machine Precision

- The round-off error is bounded in terms of the *machine precision*,  $\varepsilon$ , which is the smallest value satisfying

$$|fp(a \circ b) - (a \circ b)| \leq \varepsilon |a \circ b|$$

for all all fp-numbers  $a, b$  and any of the four operations  $+, -, \cdot, /$  instead of  $\circ$ , for which  $a \circ b$  does not cause an underflow or an overflow.

- On IEEE-754 machines,  $\varepsilon = 2^{-23} \approx 1.19 \cdot 10^{-7}$  for floats, and  $\varepsilon = 2^{-52} \approx 2.22 \cdot 10^{-16}$  for doubles.
- On some exotic platform,  $\varepsilon$  can be determined approximately by finding the smallest positive value  $x$  such that  $1 + x \neq 1$ .
- Note: Some compilers promote floats to doubles!

# Floating-Point Arithmetic and Compilers

- Be warned that the floating-point unit on x86 processors uses 80bit registers and operators, while standard “double” variables are stored in 64bit memory cells.



## Floating-Point Arithmetic and Compilers

- Be warned that the floating-point unit on x86 processors uses 80bit registers and operators, while standard “double” variables are stored in 64bit memory cells.
- Hence, rounding to a lower precision is necessary whenever a floating-point variable is transferred from register to memory.
- Optimizing compilers analyze code and keep variables within the registers whenever this makes sense, without storing intermediate results in memory.



- Be warned that the floating-point unit on x86 processors uses 80bit registers and operators, while standard “double” variables are stored in 64bit memory cells.
- Hence, rounding to a lower precision is necessary whenever a floating-point variable is transferred from register to memory.
- Optimizing compilers analyze code and keep variables within the registers whenever this makes sense, without storing intermediate results in memory.

## Warning

The result of fp-computations may depend on the compile-time options!

- Be warned that the floating-point unit on x86 processors uses 80bit registers and operators, while standard “double” variables are stored in 64bit memory cells.
- Hence, rounding to a lower precision is necessary whenever a floating-point variable is transferred from register to memory.
- Optimizing compilers analyze code and keep variables within the registers whenever this makes sense, without storing intermediate results in memory.

## Warning

The result of fp-computations may depend on the compile-time options!

- As a consequence of this “excess precision” of the register variables, on my PC,

$$\sum_{i=1}^{10000000} 0.001 = 1000.000000000009095 \quad \text{when } \text{gcc } -O2 \text{ was used, and}$$

$$\sum_{i=1}^{10000000} 0.001 = 999.9999999832650701 \quad \text{when } \text{gcc } -O0 \text{ was used.}$$

- Be warned that the floating-point unit on x86 processors uses 80bit registers and operators, while standard “double” variables are stored in 64bit memory cells.
- Hence, rounding to a lower precision is necessary whenever a floating-point variable is transferred from register to memory.
- Optimizing compilers analyze code and keep variables within the registers whenever this makes sense, without storing intermediate results in memory.

## Warning

The result of fp-computations may depend on the compile-time options!

- As a consequence of this “excess precision” of the register variables, on my PC,

$$\sum_{i=1}^{10000000} 0.001 = 1000.000000000009095 \quad \text{when } \text{gcc } -O2 \text{ was used, and}$$

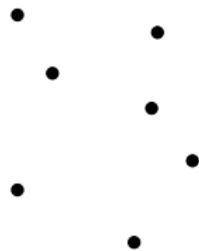
$$\sum_{i=1}^{10000000} 0.001 = 999.9999999832650701 \quad \text{when } \text{gcc } -O0 \text{ was used.}$$

- The `gcc` option `-ffloat-store` cures this problem, possibly at the expense of an increased cpu-time consumption.

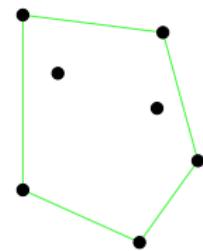


# Geometric Predicates

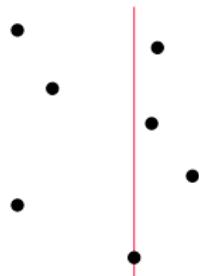
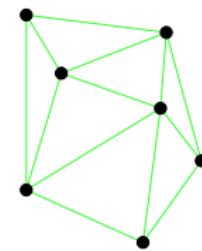
Sorting



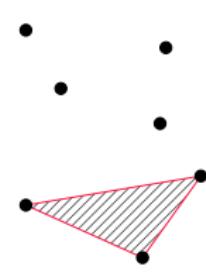
Convex Hull



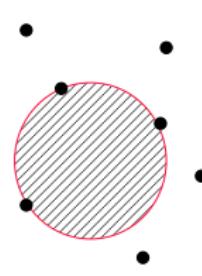
Delaunay Triangulation



coordinate comparison



sidedness test



incircle test

## Degeneracy Versus Numerical Precision

- Degeneracies are caused by the special position of two or more geometric objects. E.g., two line segments that overlap partially rather than being disjoint or intersecting in a point.



## Degeneracy Versus Numerical Precision

- Degeneracies are caused by the special position of two or more geometric objects. E.g., two line segments that overlap partially rather than being disjoint or intersecting in a point.
- Typically, a degeneracy occurs if a predicate evaluates to zero.
- The net result of degeneracy is a vastly increased number of so-called *special cases*.

## Degeneracy Versus Numerical Precision

- Degeneracies are caused by the special position of two or more geometric objects. E.g., two line segments that overlap partially rather than being disjoint or intersecting in a point.
- Typically, a degeneracy occurs if a predicate evaluates to zero.
- The net result of degeneracy is a vastly increased number of so-called *special cases*.
- Note: Degeneracies may be intentional, and real-world data should be assumed to be degenerate!

## Degeneracy Versus Numerical Precision

- Degeneracies are caused by the special position of two or more geometric objects. E.g., two line segments that overlap partially rather than being disjoint or intersecting in a point.
- Typically, a degeneracy occurs if a predicate evaluates to zero.
- The net result of degeneracy is a vastly increased number of so-called *special cases*.
- Note: Degeneracies may be intentional, and real-world data should be assumed to be degenerate!
- The mere fact that a degeneracy cannot be classified reliably on an fp-arithmetic complicates matters significantly.
- Obvious problems:
  - Is it a special case or simply a numerical inaccuracy?
  - How shall we handle all special cases correctly?
  - How can we even be sure that all special cases have been modeled?



## Epsilon Thresholds

- Topological decisions are based on the results of floating-point (fp) computations, which are prone to round-off errors.
- Threshold-based comparison:

$$(a =_{\varepsilon} b) := (|a - b| \leq \varepsilon),$$

for some positive value of  $\varepsilon$ .



## Epsilon Thresholds

- Topological decisions are based on the results of floating-point (fp) computations, which are prone to round-off errors.
- Threshold-based comparison:

$$(a =_{\varepsilon} b) := (|a - b| \leq \varepsilon),$$

for some positive value of  $\varepsilon$ .

- Note:  $|a - b| \leq \varepsilon$  rather than  $|a - b| < \varepsilon$ !



## Epsilon Thresholds

- Topological decisions are based on the results of floating-point (fp) computations, which are prone to round-off errors.
- Threshold-based comparison:

$$(a =_{\varepsilon} b) := (|a - b| \leq \varepsilon),$$

for some positive value of  $\varepsilon$ .

- Note:  $|a - b| \leq \varepsilon$  rather than  $|a - b| < \varepsilon$ !
- Threshold-based comparisons are not transitive:  $a =_{\varepsilon} b$  and  $b =_{\varepsilon} c$  need not imply  $a =_{\varepsilon} c$ .



## Epsilon Thresholds

- Topological decisions are based on the results of floating-point (fp) computations, which are prone to round-off errors.
- Threshold-based comparison:

$$(a =_{\varepsilon} b) := (|a - b| \leq \varepsilon),$$

for some positive value of  $\varepsilon$ .

- Note:  $|a - b| \leq \varepsilon$  rather than  $|a - b| < \varepsilon$ !
- Threshold-based comparisons are not transitive:  $a =_{\varepsilon} b$  and  $b =_{\varepsilon} c$  need not imply  $a =_{\varepsilon} c$ .
- This gap between theory and practice has important and severe consequences for the actual coding practice when implementing geometric algorithms:
  - ➊ The correctness proof of the mathematical algorithm does not extend to the program, and the program can fail on seemingly appropriate input data.



## Epsilon Thresholds

- Topological decisions are based on the results of floating-point (fp) computations, which are prone to round-off errors.
- Threshold-based comparison:

$$(a =_{\varepsilon} b) := (|a - b| \leq \varepsilon),$$

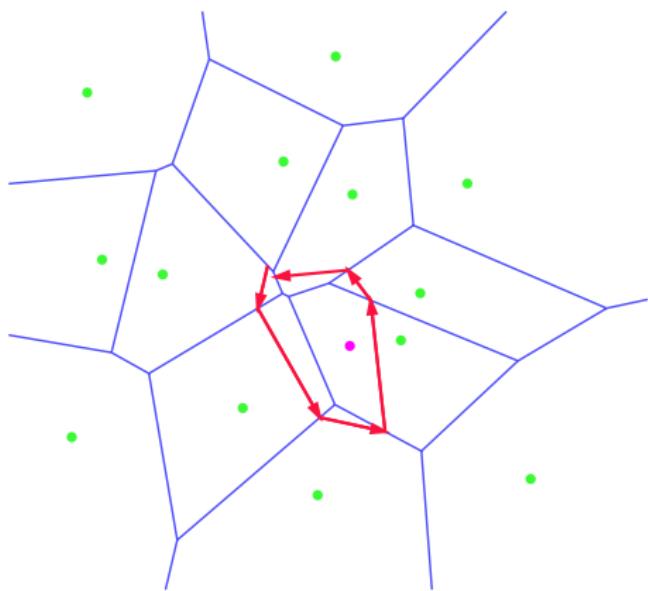
for some positive value of  $\varepsilon$ .

- Note:  $|a - b| \leq \varepsilon$  rather than  $|a - b| < \varepsilon$ !
- Threshold-based comparisons are not transitive:  $a =_{\varepsilon} b$  and  $b =_{\varepsilon} c$  need not imply  $a =_{\varepsilon} c$ .
- This gap between theory and practice has important and severe consequences for the actual coding practice when implementing geometric algorithms:
  - The correctness proof of the mathematical algorithm does not extend to the program, and the program can fail on seemingly appropriate input data.
  - Local consistency need not imply global consistency.



## Sample Robustness Problem: Lack of Global Consistency

- Local consistency need not imply global consistency.



## Sample Robustness Problem: Problematic Intersections

- The quartic equation  $x^4 + 4x^3 + 6x^2 + 4x + 1 = 0$  has the quadruple root  $x = -1$ .

## Sample Robustness Problem: Problematic Intersections

- The quartic equation  $x^4 + 4x^3 + 6x^2 + 4x + 1 = 0$  has the quadruple root  $x = -1$ .
- Changing the coefficient of  $x$  to 4.00000001 drastically affects the solution: Now we get  $x = -1.01002496875 \dots$  and  $x = -0.99002503124 \dots$  as the only real roots of the equation.

## Sample Robustness Problem: Problematic Intersections

- The quartic equation  $x^4 + 4x^3 + 6x^2 + 4x + 1 = 0$  has the quadruple root  $x = -1$ .
- Changing the coefficient of  $x$  to 4.00000001 drastically affects the solution: Now we get  $x = -1.01002496875 \dots$  and  $x = -0.99002503124 \dots$  as the only real roots of the equation.
- Similarly, the linear system

$$x + 2y = 3$$

$$0.48x + 0.99y = 1.47$$

has the exact solution  $x = 1, y = 1$ .

## Sample Robustness Problem: Problematic Intersections

- The quartic equation  $x^4 + 4x^3 + 6x^2 + 4x + 1 = 0$  has the quadruple root  $x = -1$ .
- Changing the coefficient of  $x$  to 4.00000001 drastically affects the solution: Now we get  $x = -1.01002496875 \dots$  and  $x = -0.99002503124 \dots$  as the only real roots of the equation.
- Similarly, the linear system

$$x + 2y = 3$$

$$0.48x + 0.99y = 1.47$$

has the exact solution  $x = 1, y = 1$ .

- However, the system

$$x + 2y = 3$$

$$0.49x + 0.99y = 1.47$$

has the exact solution  $x = 3, y = 0$ .

## Sample Robustness Problem: Problematic Intersections

- The quartic equation  $x^4 + 4x^3 + 6x^2 + 4x + 1 = 0$  has the quadruple root  $x = -1$ .
- Changing the coefficient of  $x$  to 4.00000001 drastically affects the solution: Now we get  $x = -1.01002496875 \dots$  and  $x = -0.99002503124 \dots$  as the only real roots of the equation.
- Similarly, the linear system

$$x + 2y = 3$$

$$0.48x + 0.99y = 1.47$$

has the exact solution  $x = 1, y = 1$ .

- However, the system

$$x + 2y = 3$$

$$0.49x + 0.99y = 1.47$$

has the exact solution  $x = 3, y = 0$ .

- Note that the old solution,  $x = 1, y = 1$ , also “nearly” fulfills this linear system.

## Sample Robustness Problem: Incorrect Orientation Predicate

- [Kettner et alii 2006] study the standard determinant-based orientation predicate on IEEE 754 fp-arithmetic to check the sidedness of  $(p_x + x \cdot u, p_y + y \cdot u)$  relative to two points  $q, r$ , for  $0 \leq x, y \leq 255$  and with  $u := 2^{-53}$ :

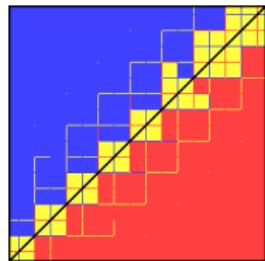
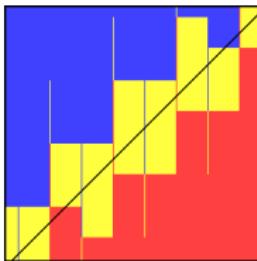
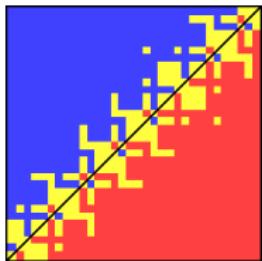
$$\text{sign} \det \begin{pmatrix} 1 & p_x + x \cdot u & p_y + y \cdot u \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix} \left\{ \begin{array}{c} > \\ = \\ < \end{array} \right\} 0 ?$$

## Sample Robustness Problem: Incorrect Orientation Predicate

- [Kettner et alii 2006] study the standard determinant-based orientation predicate on IEEE 754 fp-arithmetic to check the sidedness of  $(p_x + x \cdot u, p_y + y \cdot u)$  relative to two points  $q, r$ , for  $0 \leq x, y \leq 255$  and with  $u := 2^{-53}$ :

$$\text{sign det} \left( \begin{array}{ccc} 1 & p_x + x \cdot u & p_y + y \cdot u \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{array} \right) \left\{ \begin{array}{c} > \\ = \\ < \end{array} \right\} 0 ?$$

- The resulting  $256 \times 256$  array of signs (as a function of  $x, y$ ) is color-coded: A yellow (red, blue) pixel indicates collinear (negative, positive, resp.) orientation.
- The black line indicates the line through  $q$  and  $r$ .
- Note the sign inversions!

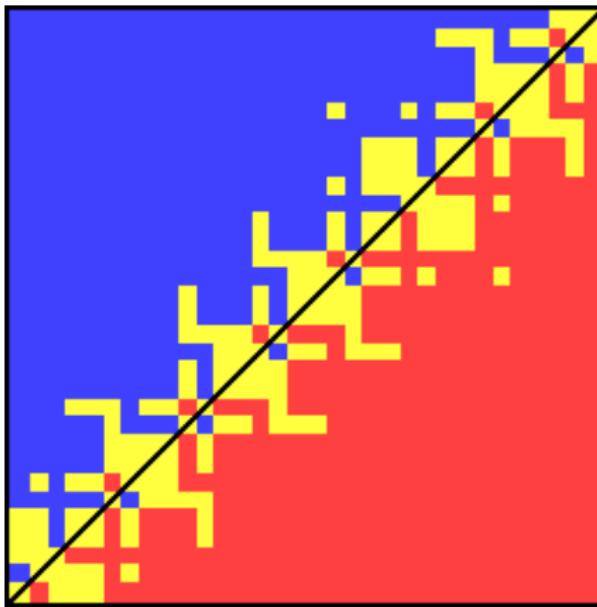


[Image credit: [www.mpi-inf.mpg.de/~kettner/proj/NonRobust/](http://www.mpi-inf.mpg.de/~kettner/proj/NonRobust/)] 

## Sample Robustness Problem: Incorrect Orientation Predicate

- [Kettner et alii 2006]: A yellow (red, blue) pixel indicates collinear (negative, positive, resp.) orientation.

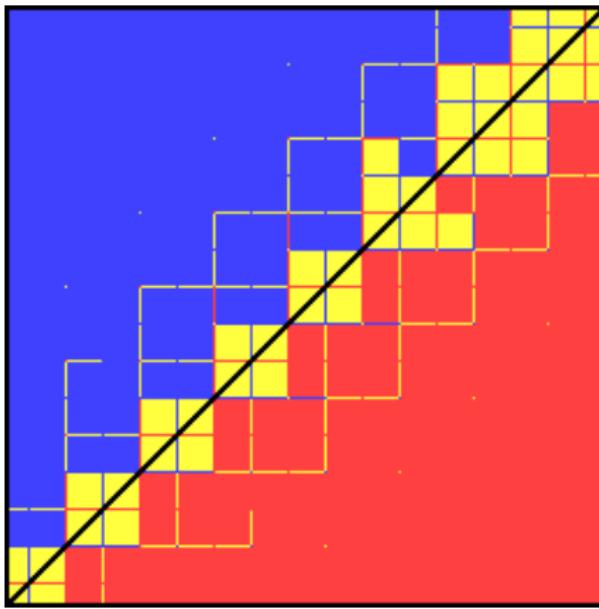
$$p := \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \quad q := \begin{pmatrix} 12 \\ 12 \end{pmatrix} \quad r := \begin{pmatrix} 24 \\ 24 \end{pmatrix}$$



## Sample Robustness Problem: Incorrect Orientation Predicate

- [Kettner et alii 2006]: A yellow (red, blue) pixel indicates collinear (negative, positive, resp.) orientation.

$$p := \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \quad q := \begin{pmatrix} 8.8000000000000007 \\ 8.8000000000000007 \end{pmatrix} \quad r := \begin{pmatrix} 12.1 \\ 12.1 \end{pmatrix}$$



## Robustness and Stability

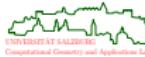
- Steve Fortune [1989]:
  - A geometric code implementation is called *robust* if it never fails, and if there exists a perturbation of the input such that the code's output is correct with respect to this perturbed input.

## Robustness and Stability

- Steve Fortune [1989]:
  - A geometric code implementation is called *robust* if it never fails, and if there exists a perturbation of the input such that the code's output is correct with respect to this perturbed input.
  - If small perturbations suffice then the code is called *stable*.

# Robustness and Stability

- Steve Fortune [1989]:
  - A geometric code implementation is called *robust* if it never fails, and if there exists a perturbation of the input such that the code's output is correct with respect to this perturbed input.
  - If small perturbations suffice then the code is called *stable*.
- Robustness implies that a code conforms to its specifications.
- The term *reliable* is often used as a synonym for robust or stable.



## Robustness Issues

- Introduction to Robustness Problems
- Approaches to Achieving Robustness
  - Exact Arithmetic
  - Exact Geometric Computing
  - Symbolic Perturbation
- Improving the Reliability of FP-Code
- Industrial Applications
- Using FP-Arithmetic to Implement the Turing Test

# Approaches to Improving Robustness

- Several approaches have been proposed in recent years:
  - Sophisticated tolerancing.
  - Error analysis.
  - Exact arithmetic (on integers, rationals, or even algebraic numbers).
  - Exact geometric computing.
  - Floating-point filters.
  - Symbolic perturbation.

# Approaches to Improving Robustness

- Several approaches have been proposed in recent years:
  - Sophisticated tolerancing.
  - Error analysis.
  - Exact arithmetic (on integers, rationals, or even algebraic numbers).
  - Exact geometric computing.
  - Floating-point filters.
  - Symbolic perturbation.
- No agreement on the best approach . . .
- All methods have shortcomings, typically also limited applicability — or they suffer from inefficiency!

## Exact Arithmetic

- Input is assumed to be exact.
- Compute the numerical value of every predicate exactly, based on exact number types.
- Exact computation is possible if all numerical values are algebraic. (This is the case for most current problems in computational geometry.)



## Exact Arithmetic

- Input is assumed to be exact.
- Compute the numerical value of every predicate exactly, based on exact number types.
- Exact computation is possible if all numerical values are algebraic. (This is the case for most current problems in computational geometry.)
- Note: We may no longer assume that each arithmetic operation takes constant time!
- Note: Constructors complicate the situation significantly!

# Exact Number Types

**Bigint:** arbitrary-precision integers.

- Usually based on Karatsuba-Offman multiplication, with  $\Theta(b^{\log_2 3}) \equiv \Theta(b^{1.58})$  complexity for multiplying  $b$ -bit numbers.
- See, e.g., GNU's GMP library <http://gmplib.org/>.

# Exact Number Types

**Bigint:** arbitrary-precision integers.

- Usually based on Karatsuba-Offman multiplication, with  $\Theta(b^{\log_2 3}) \equiv \Theta(b^{1.58})$  complexity for multiplying  $b$ -bit numbers.
- See, e.g., GNU's GMP library <http://gmplib.org/>.

**Bigrational:** quotients of bigints.

- Standard rational arithmetic.
- It is important to reduce fractions frequently.
- Euclid's algorithm can be used for finding common factors.
- See, e.g., GMP's `mpq_t` number type.

# Exact Number Types

**Bigint:** arbitrary-precision integers.

- Usually based on Karatsuba-Offman multiplication, with  $\Theta(b^{\log_2 3}) \equiv \Theta(b^{1.58})$  complexity for multiplying  $b$ -bit numbers.
- See, e.g., GNU's GMP library <http://gmplib.org/>.

**Bigrational:** quotients of bigints.

- Standard rational arithmetic.
- It is important to reduce fractions frequently.
- Euclid's algorithm can be used for finding common factors.
- See, e.g., GMP's `mpq_t` number type.

**Algebraic numbers:** roots of polynomials with bigint coefficients.

- Complexity of deciding equality grows with the number of operators allowed.
- [Caviness 1967] proved undecidability of functional equivalence for moderately complicated terms.

# Exact Number Types

**Bigint:** arbitrary-precision integers.

- Usually based on Karatsuba-Offman multiplication, with  $\Theta(b^{\log_2 3}) \equiv \Theta(b^{1.58})$  complexity for multiplying  $b$ -bit numbers.
- See, e.g., GNU's GMP library <http://gmplib.org/>.

**Bigrational:** quotients of bigints.

- Standard rational arithmetic.
- It is important to reduce fractions frequently.
- Euclid's algorithm can be used for finding common factors.
- See, e.g., GMP's `mpq_t` number type.

**Algebraic numbers:** roots of polynomials with bigint coefficients.

- Complexity of deciding equality grows with the number of operators allowed.
- [Caviness 1967] proved undecidability of functional equivalence for moderately complicated terms.

**Homogeneous coordinates:** not exactly a number type.

- Can often be used to avoid divisions.
- It is important to reduce fractions frequently.
- Most predicates can be expressed directly in terms of homogeneous coordinates.

# Exact Geometric Computing

- Do we really need exact arithmetic for solving geometric problems?



# Exact Geometric Computing

- Do we really need exact arithmetic for solving geometric problems?
- Not quite! → Exact geometric computing (EGC)
- Input is assumed to be exact.
- Basic idea: Make sure that the sign of every predicate is evaluated correctly.

# Exact Geometric Computing

- Do we really need exact arithmetic for solving geometric problems?
- Not quite! → Exact geometric computing (EGC)
- Input is assumed to be exact.
- Basic idea: Make sure that the sign of every predicate is evaluated correctly.
- Error analysis is needed to come up with *root bounds*: A positive number  $r$  forms a root bound for a predicate  $p$  if  $|p(x)| > r$  guarantees that the evaluation of  $p$  for  $x$  yields the correct sign.

# Exact Geometric Computing

- Do we really need exact arithmetic for solving geometric problems?
- Not quite! → Exact geometric computing (EGC)
- Input is assumed to be exact.
- Basic idea: Make sure that the sign of every predicate is evaluated correctly.
- Error analysis is needed to come up with *root bounds*: A positive number  $r$  forms a root bound for a predicate  $p$  if  $|p(x)| > r$  guarantees that the evaluation of  $p$  for  $x$  yields the correct sign.
- Main problems:
  - Arbitrary-precision arithmetic is needed.
  - Tight root bounds are difficult to obtain.
  - Constructors may cause the bit-length to grow tremendously.

- Do we really need exact arithmetic for solving geometric problems?
- Not quite! → Exact geometric computing (EGC)
- Input is assumed to be exact.
- Basic idea: Make sure that the sign of every predicate is evaluated correctly.
- Error analysis is needed to come up with *root bounds*: A positive number  $r$  forms a root bound for a predicate  $p$  if  $|p(x)| > r$  guarantees that the evaluation of  $p$  for  $x$  yields the correct sign.
- Main problems:
  - Arbitrary-precision arithmetic is needed.
  - Tight root bounds are difficult to obtain.
  - Constructors may cause the bit-length to grow tremendously.
- The use of fp-filters may help to keep the increase in cpu-time consumption a bit more moderate:
  - Work with fp-arithmetic and check whether it gives the right answer.
  - Resort to exact arithmetic if the answer is incorrect.
  - Great example: Shewchuk's fine (and efficient!) CDT code "Triangle" [Shewchuk 1996].

- Do we really need exact arithmetic for solving geometric problems?
- Not quite! → Exact geometric computing (EGC)
- Input is assumed to be exact.
- Basic idea: Make sure that the sign of every predicate is evaluated correctly.
- Error analysis is needed to come up with *root bounds*: A positive number  $r$  forms a root bound for a predicate  $p$  if  $|p(x)| > r$  guarantees that the evaluation of  $p$  for  $x$  yields the correct sign.
- Main problems:
  - Arbitrary-precision arithmetic is needed.
  - Tight root bounds are difficult to obtain.
  - Constructors may cause the bit-length to grow tremendously.
- The use of fp-filters may help to keep the increase in cpu-time consumption a bit more moderate:
  - Work with fp-arithmetic and check whether it gives the right answer.
  - Resort to exact arithmetic if the answer is incorrect.
  - Great example: Shewchuk's fine (and efficient!) CDT code "Triangle" [Shewchuk 1996].
- Software libraries that provide support for EGC: Core, CGAL, and Mörig's RealAlgebraic data type.

## Symbolic Perturbation

- Aka: Simulation of simplicity [Edelsbrunner&Mücke 1990].
- Basic idea: Perturb input such that no degeneracy occurs.

## Symbolic Perturbation

- Aka: Simulation of simplicity [Edelsbrunner&Mücke 1990].
- Basic idea: Perturb input such that no degeneracy occurs.
- Roughly, each coordinate  $x$  is replaced by  $x + f(\varepsilon)$ , where  $\varepsilon$  is unknown but very small and the perturbation function  $f$  is simple, e.g. a polynomial.
- Symbolic perturbation transforms the result of a numerical predicate into a polynomial in  $\varepsilon$ , whose sign is given by the sign of the first non-zero coefficient.
- For several important types of predicates perturbation functions can be chosen such that all degeneracies are resolved.

## Symbolic Perturbation

- Aka: Simulation of simplicity [Edelsbrunner&Mücke 1990].
- Basic idea: Perturb input such that no degeneracy occurs.
- Roughly, each coordinate  $x$  is replaced by  $x + f(\varepsilon)$ , where  $\varepsilon$  is unknown but very small and the perturbation function  $f$  is simple, e.g. a polynomial.
- Symbolic perturbation transforms the result of a numerical predicate into a polynomial in  $\varepsilon$ , whose sign is given by the sign of the first non-zero coefficient.
- For several important types of predicates perturbation functions can be chosen such that all degeneracies are resolved.
- Main problems:
  - Exact arithmetic is required for evaluating the predicates.
  - Constructors are often disallowed.
  - The output computed does not correspond to the actual input.
  - Intentional degeneracies in the input are resolved, too!

## Robustness Issues

- Introduction to Robustness Problems
- Approaches to Achieving Robustness
- Improving the Reliability of FP-Code
  - Standard Tricks for Achieving Reliable FP-Code
  - Topology-Oriented Computation
  - Relaxation of Epsilon Thresholds
  - Desperate Mode
  - MPFR Library
- Industrial Applications
- Using FP-Arithmetic to Implement the Turing Test

## Standard Tricks for Achieving Reliable FP-Code

- All fp-computations need to be consistent.

## Standard Tricks for Achieving Reliable FP-Code

- All fp-computations need to be consistent.
- For instance, when computing  $3 \times 3$  determinants, the following identities are a must even on an fp-arithmetic:

$$\begin{aligned}\det(p, q, r) &= \det(q, r, p) = \det(r, p, q) \\ &= -\det(q, p, r) = -\det(p, r, q) = -\det(r, q, p).\end{aligned}$$

## Standard Tricks for Achieving Reliable FP-Code

- All fp-computations need to be consistent.
- For instance, when computing  $3 \times 3$  determinants, the following identities are a must even on an fp-arithmetic:

$$\begin{aligned}\det(p, q, r) &= \det(q, r, p) = \det(r, p, q) \\ &= -\det(q, p, r) = -\det(p, r, q) = -\det(r, q, p).\end{aligned}$$

- Recall that algebraically equivalent terms need not be equally reliable on an fp-arithmetic.

## Standard Tricks for Achieving Reliable FP-Code

- All fp-computations need to be consistent.
- For instance, when computing  $3 \times 3$  determinants, the following identities are a must even on an fp-arithmetic:

$$\begin{aligned}\det(p, q, r) &= \det(q, r, p) = \det(r, p, q) \\ &= -\det(q, p, r) = -\det(p, r, q) = -\det(r, q, p).\end{aligned}$$

- Recall that algebraically equivalent terms need not be equally reliable on an fp-arithmetic.
- For instance, consider the quadratic equation  $x^2 + px + q = 0$  and compare

$$\left\{ \begin{array}{l} x_1 = -\frac{1}{2}(p + \sqrt{p^2 - 4q}) \\ x_2 = -\frac{1}{2}(p - \sqrt{p^2 - 4q}) \end{array} \right\} \quad \text{to} \quad \left\{ \begin{array}{l} x_1 = -\frac{1}{2}(p + \text{sign}(p)\sqrt{p^2 - 4q}) \\ x_2 = q/x_1 \end{array} \right\}$$

if  $|q|$  is small.



## Standard Tricks for Achieving Reliable FP-Code

- All fp-computations need to be consistent.
- For instance, when computing  $3 \times 3$  determinants, the following identities are a must even on an fp-arithmetic:

$$\begin{aligned}\det(p, q, r) &= \det(q, r, p) = \det(r, p, q) \\ &= -\det(q, p, r) = -\det(p, r, q) = -\det(r, q, p).\end{aligned}$$

- Recall that algebraically equivalent terms need not be equally reliable on an fp-arithmetic.
- For instance, consider the quadratic equation  $x^2 + px + q = 0$  and compare
$$\left\{ \begin{array}{l} x_1 = -\frac{1}{2}(p + \sqrt{p^2 - 4q}) \\ x_2 = -\frac{1}{2}(p - \sqrt{p^2 - 4q}) \end{array} \right\} \quad \text{to} \quad \left\{ \begin{array}{l} x_1 = -\frac{1}{2}(p + \text{sign}(p)\sqrt{p^2 - 4q}) \\ x_2 = q/x_1 \end{array} \right\}$$
if  $|q|$  is small.
- Epsilon-based comparisons need to be relative to the absolute values of the numbers to be compared, or the input has to be scaled (by performing shifts!) to fit into the unit square/cube prior to the actual computation.

## Standard Tricks for Achieving Reliable FP-Code

- All fp-computations need to be consistent.
- For instance, when computing  $3 \times 3$  determinants, the following identities are a must even on an fp-arithmetic:

$$\begin{aligned}\det(p, q, r) &= \det(q, r, p) = \det(r, p, q) \\ &= -\det(q, p, r) = -\det(p, r, q) = -\det(r, q, p).\end{aligned}$$

- Recall that algebraically equivalent terms need not be equally reliable on an fp-arithmetic.
- For instance, consider the quadratic equation  $x^2 + px + q = 0$  and compare

$$\left\{ \begin{array}{l} x_1 = -\frac{1}{2}(p + \sqrt{p^2 - 4q}) \\ x_2 = -\frac{1}{2}(p - \sqrt{p^2 - 4q}) \end{array} \right\} \quad \text{to} \quad \left\{ \begin{array}{l} x_1 = -\frac{1}{2}(p + \text{sign}(p)\sqrt{p^2 - 4q}) \\ x_2 = q/x_1 \end{array} \right\}$$

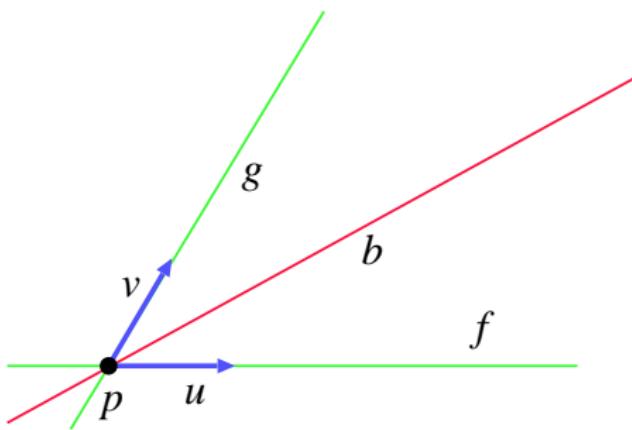
if  $|q|$  is small.

- Epsilon-based comparisons need to be relative to the absolute values of the numbers to be compared, or the input has to be scaled (by performing shifts!) to fit into the unit square/cube prior to the actual computation.
- Use iterations as back-up for analytical solutions to equations. If at all possible, use methods that bracket the solution sought!



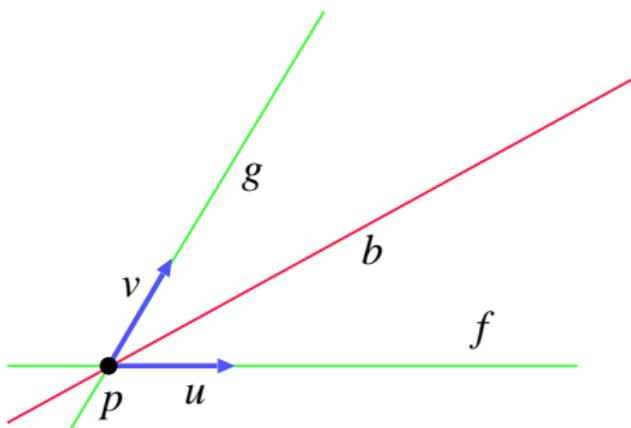
## Case Study: Computing Voronoi Bisectors

- Algorithm for computing a bisector  $b$  between two lines  $f$  and  $g$  which are not parallel:
  - Compute their point of intersection:  $p$ .
  - Compute unit direction vectors  $u$  and  $v$  of  $f, g$ .
  - Then a parameterization of  $b$  is given by  $p + t \cdot (u + v)$ .



## Case Study: Computing Voronoi Bisectors

- Algorithm for computing a bisector  $b$  between two lines  $f$  and  $g$  which are not parallel:
  - Compute their point of intersection:  $p$ .
  - Compute unit direction vectors  $u$  and  $v$  of  $f, g$ .
  - Then a parameterization of  $b$  is given by  $p + t \cdot (u + v)$ .
- This “natural” approach to computing  $b$  becomes completely infeasible if  $f$  and  $g$  are nearly parallel. (In that case the computation of  $p$  will become *very* unreliable!)



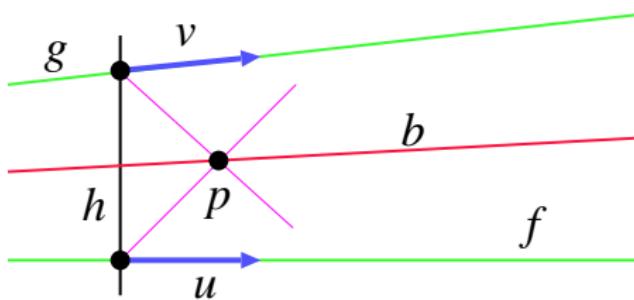
## Case Study: Computing Voronoi Bisectors

- Algorithm for computing a bisector  $b$  between two lines  $f$  and  $g$  which are roughly parallel:



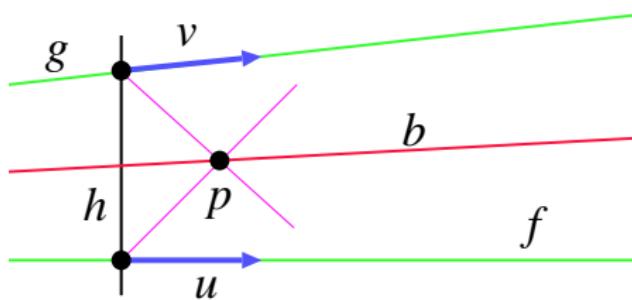
## Case Study: Computing Voronoi Bisectors

- Algorithm for computing a bisector  $b$  between two lines  $f$  and  $g$  which are roughly parallel:
  - Compute a line  $h$  that is normal on  $f$ .



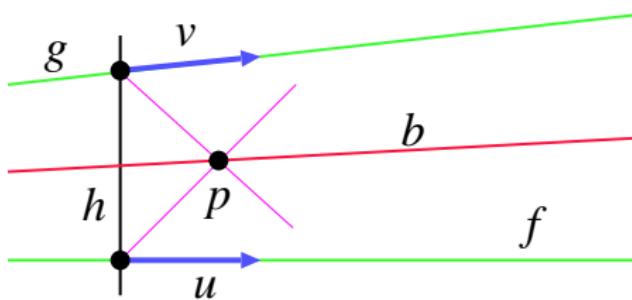
## Case Study: Computing Voronoi Bisectors

- Algorithm for computing a bisector  $b$  between two lines  $f$  and  $g$  which are roughly parallel:
  - Compute a line  $h$  that is normal on  $f$ .
  - Compute the bisector  $b_1$  between  $h$  and  $f$ , and the bisector  $b_2$  between  $h$  and  $g$ .



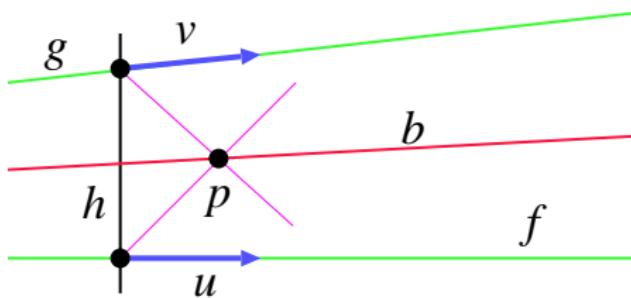
## Case Study: Computing Voronoi Bisectors

- Algorithm for computing a bisector  $b$  between two lines  $f$  and  $g$  which are roughly parallel:
  - Compute a line  $h$  that is normal on  $f$ .
  - Compute the bisector  $b_1$  between  $h$  and  $f$ , and the bisector  $b_2$  between  $h$  and  $g$ .
  - Compute their point of intersection:  $p$ .



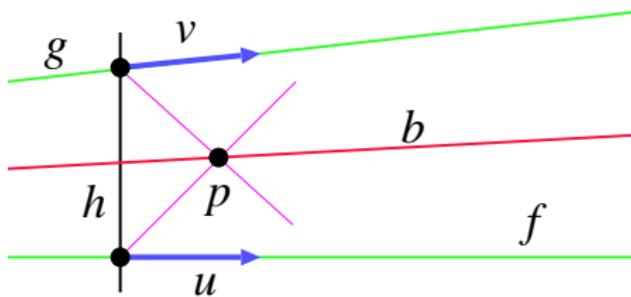
## Case Study: Computing Voronoi Bisectors

- Algorithm for computing a bisector  $b$  between two lines  $f$  and  $g$  which are roughly parallel:
  - Compute a line  $h$  that is normal on  $f$ .
  - Compute the bisector  $b_1$  between  $h$  and  $f$ , and the bisector  $b_2$  between  $h$  and  $g$ .
  - Compute their point of intersection:  $p$ .
  - Compute unit direction vectors  $u$  and  $v$  of  $f, g$ .
  - Then a parameterization of  $b$  is given by  $p + t \cdot (u + v)$ .



## Case Study: Computing Voronoi Bisectors

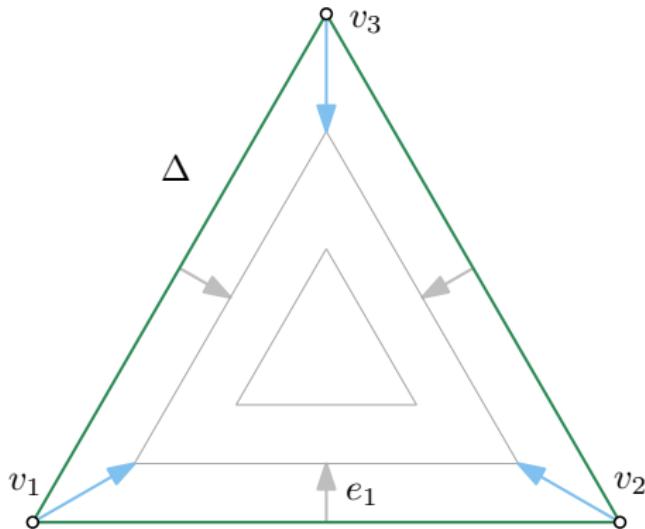
- Algorithm for computing a bisector  $b$  between two lines  $f$  and  $g$  which are roughly parallel:
  - Compute a line  $h$  that is normal on  $f$ .
  - Compute the bisector  $b_1$  between  $h$  and  $f$ , and the bisector  $b_2$  between  $h$  and  $g$ .
  - Compute their point of intersection:  $p$ .
  - Compute unit direction vectors  $u$  and  $v$  of  $f, g$ .
  - Then a parameterization of  $b$  is given by  $p + t \cdot (u + v)$ .



- Note: All intersections are defined by pairs of lines that are roughly perpendicular.

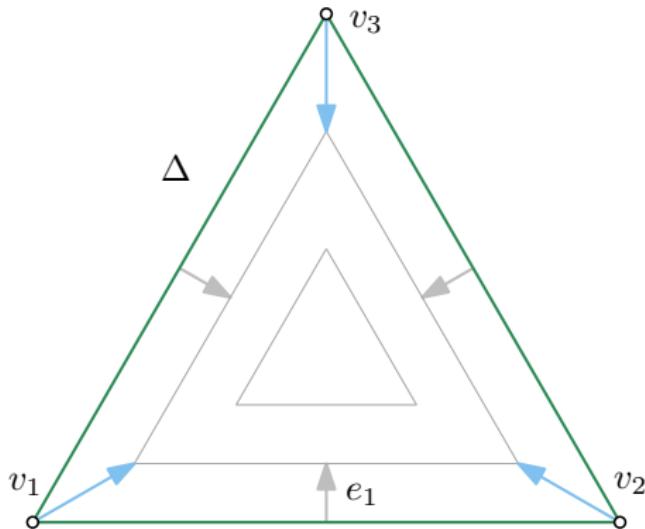
## Case Study: Collapse Time of Kinetic Triangle

- Suppose that all three vertices  $v_1, v_2, v_3$  of a triangle  $\Delta$  move along paths (modeled as functions of time), and we are interested in knowing when the triangle collapses. (That is, when it has zero area.)



## Case Study: Collapse Time of Kinetic Triangle

- Suppose that all three vertices  $v_1, v_2, v_3$  of a triangle  $\Delta$  move along paths (modeled as functions of time), and we are interested in knowing when the triangle collapses. (That is, when it has zero area.)



- How can we determine the time(s) of collapse?

## Case Study: Collapse Time of Kinetic Triangle

- Which option is better?
  - ① Check the signed area of the triangle.

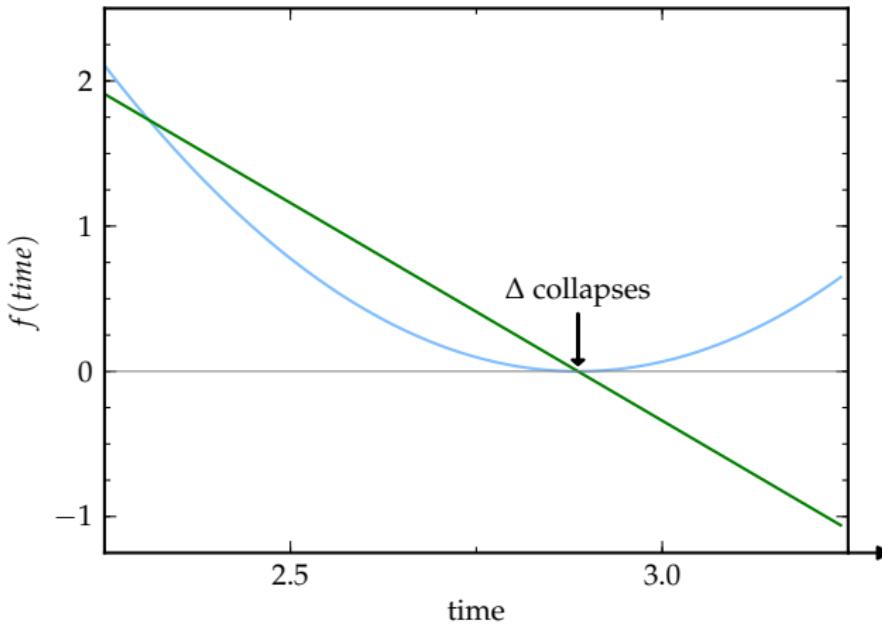
## Case Study: Collapse Time of Kinetic Triangle

- Which option is better?
  - ① Check the signed area of the triangle.
  - ② Check the signed distance of a kinetic vertex to its opposite edge.



## Case Study: Collapse Time of Kinetic Triangle

- Which option is better?
  - ① Check the signed area of the triangle.
  - ② Check the signed distance of a kinetic vertex to its opposite edge.



# Topology-Oriented Computation

- First used by Sugihara et alii [1992, 2000].
- Basic idea:
  - ① Define topological criteria that the output has to meet.



# Topology-Oriented Computation

- First used by Sugihara et alii [1992, 2000].
- Basic idea:
  - ① Define topological criteria that the output has to meet.
  - ② Use fp-computations to choose among different topological set-ups if two or more set-ups fulfill all topological criteria.

## Topology-Oriented Computation

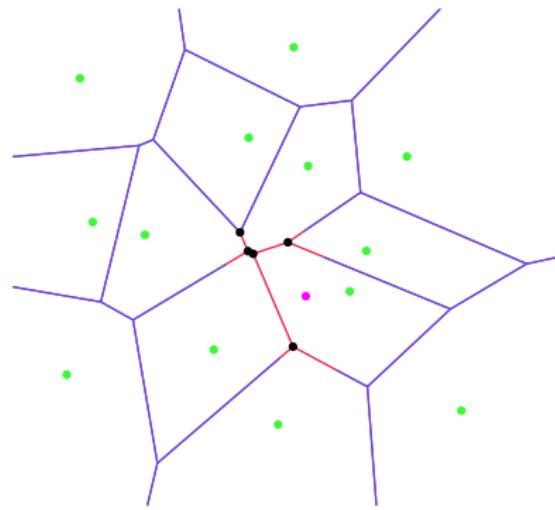
- First used by Sugihara et alii [1992, 2000].
- Basic idea:
  - ① Define topological criteria that the output has to meet.
  - ② Use fp-computations to choose among different topological set-ups if two or more set-ups fulfill all topological criteria.
- Main problem: For many problems it is difficult to formulate meaningful topological criteria.

## Topology-Oriented Computation

- First used by Sugihara et alii [1992, 2000].
- Basic idea:
  - ① Define topological criteria that the output has to meet.
  - ② Use fp-computations to choose among different topological set-ups if two or more set-ups fulfill all topological criteria.
- Main problem: For many problems it is difficult to formulate meaningful topological criteria.
- Sample application: When inserting a new site into a Voronoi diagram (during an incremental construction), the portion of the old Voronoi diagram to be deleted forms a tree.

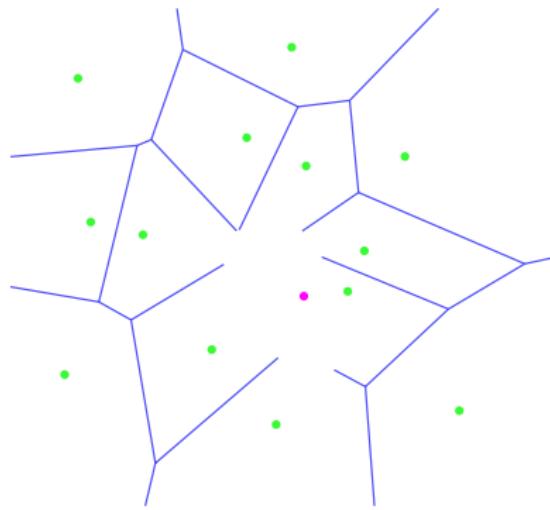
## Topology-Oriented Computation

- The portion of the Voronoi diagram to be deleted forms a tree.
- We start at a Voronoi node and scan the Voronoi diagram for nodes to be deleted, making sure that no cycle occurs.



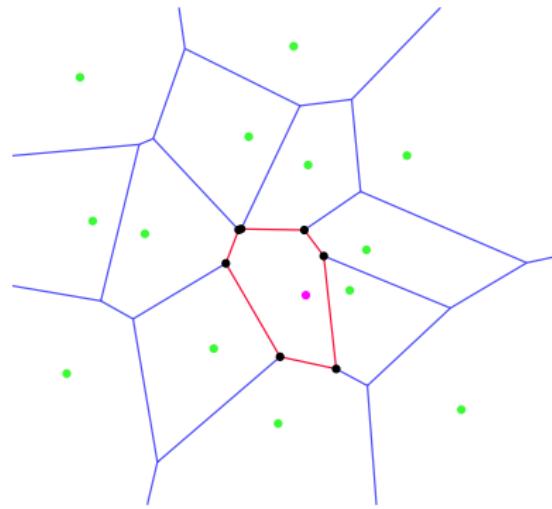
## Topology-Oriented Computation

- The portion of the Voronoi diagram to be deleted forms a tree.
- We start at a Voronoi node and scan the Voronoi diagram for nodes to be deleted, making sure that no cycle occurs.
- All Voronoi nodes found by this scan are deleted, and their incident bisectors are shortened appropriately.



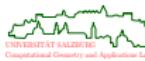
## Topology-Oriented Computation

- The portion of the Voronoi diagram to be deleted forms a tree.
- We start at a Voronoi node and scan the Voronoi diagram for nodes to be deleted, making sure that no cycle occurs.
- All Voronoi nodes found by this scan are deleted, and their incident bisectors are shortened appropriately.
- All that remains to be done is to compute the new Voronoi nodes, which form the corners of the new Voronoi polygon, and to link them in cyclic order.



## Relaxation of Epsilon Thresholds

- Common practice: Use epsilon thresholds for comparisons with respect to zero.
- The test  $x = 0?$  is replaced by the test  $|x| \leq \varepsilon?$ , for some positive value of  $\varepsilon$ .
- Just what is an appropriate value for  $\varepsilon$ ?



## Relaxation of Epsilon Thresholds

- Common practice: Use epsilon thresholds for comparisons with respect to zero.
- The test  $x = 0?$  is replaced by the test  $|x| \leq \varepsilon?$ , for some positive value of  $\varepsilon$ .
- Just what is an appropriate value for  $\varepsilon$ ?
- Personal opinion: Asking the user of a geometric code to choose an appropriate precision threshold comes close to asking for witch-craft.
- To make the situation worse, experience tells me that one fixed precision threshold will rarely suffice to handle all sorts of different input data.



## Relaxation of Epsilon Thresholds

- Common practice: Use epsilon thresholds for comparisons with respect to zero.
- The test  $x = 0?$  is replaced by the test  $|x| \leq \varepsilon?$ , for some positive value of  $\varepsilon$ .
- Just what is an appropriate value for  $\varepsilon$ ?
- Personal opinion: Asking the user of a geometric code to choose an appropriate precision threshold comes close to asking for witch-craft.
- To make the situation worse, experience tells me that one fixed precision threshold will rarely suffice to handle all sorts of different input data.
- Alternate approach:
  - A natural lower bound  $\varepsilon_{min}$  for a suitable threshold is given by the floating-point precision of the machine used.



## Relaxation of Epsilon Thresholds

- Common practice: Use epsilon thresholds for comparisons with respect to zero.
- The test  $x = 0?$  is replaced by the test  $|x| \leq \varepsilon?$ , for some positive value of  $\varepsilon$ .
- Just what is an appropriate value for  $\varepsilon$ ?
- Personal opinion: Asking the user of a geometric code to choose an appropriate precision threshold comes close to asking for witch-craft.
- To make the situation worse, experience tells me that one fixed precision threshold will rarely suffice to handle all sorts of different input data.
- Alternate approach:
  - A natural lower bound  $\varepsilon_{min}$  for a suitable threshold is given by the floating-point precision of the machine used.
  - The user specifies the maximum distance that two points (out of the unit square) may be apart in order to allow the code to treat them as one point. This yields an upper bound  $\varepsilon_{max}$ .

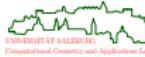
## Relaxation of Epsilon Thresholds

- Common practice: Use epsilon thresholds for comparisons with respect to zero.
- The test  $x = 0?$  is replaced by the test  $|x| \leq \varepsilon?$ , for some positive value of  $\varepsilon$ .
- Just what is an appropriate value for  $\varepsilon$ ?
- Personal opinion: Asking the user of a geometric code to choose an appropriate precision threshold comes close to asking for witch-craft.
- To make the situation worse, experience tells me that one fixed precision threshold will rarely suffice to handle all sorts of different input data.
- Alternate approach:
  - A natural lower bound  $\varepsilon_{min}$  for a suitable threshold is given by the floating-point precision of the machine used.
  - The user specifies the maximum distance that two points (out of the unit square) may be apart in order to allow the code to treat them as one point. This yields an upper bound  $\varepsilon_{max}$ .
  - The code varies  $\varepsilon$  at its own discretion, always attempting to succeed with the smallest  $\varepsilon$  possible, i.e., with maximal precision.

# Relaxation of Epsilon Thresholds

## Algorithm Typical Computational Unit

```
1.    $\varepsilon = \varepsilon_{min}$ ;                                (* set epsilon to maximum precision *)
2. repeat
3.    $x = \text{ComputeData}(\varepsilon)$ ;                (* compute some data *)
4.   success = CheckConditions( $x, \varepsilon$ ); (* check topological/numerical conditions *)
5.   if ( not success ) then
6.      $\varepsilon = 10 \cdot \varepsilon$ ;                      (* relaxation of epsilon threshold *)
7.     reset data structures appropriately;
8.   until ( success OR  $\varepsilon > \varepsilon_{max}$  );
9.    $\varepsilon = \varepsilon_{min}$ ;                            (* make sure to reset epsilon *)
10.  if ( not success ) then
11.    illegal = CheckInput();                  (* check locally for soundness of input *)
12.    if ( illegal ) then
13.      clean data locally;                  (* fix the problem in the input data *)
14.      restart computation from scratch;
15.    else
16.       $x = \text{DesperateMode}()$ ;          (* time to hope for the best *)
```



## Desperate Mode

- What shall we do if computing  $x$  would require an invalid operation, e.g., a division by zero or taking the square root of a negative number?
- After all,  $x$  may be needed for subsequent computations! E.g., if a new Voronoi node is to be inserted into the Voronoi diagram then the code may subsequently need to access the coordinates of this node.



## Desperate Mode

- What shall we do if computing  $x$  would require an invalid operation, e.g., a division by zero or taking the square root of a negative number?
- After all,  $x$  may be needed for subsequent computations! E.g., if a new Voronoi node is to be inserted into the Voronoi diagram then the code may subsequently need to access the coordinates of this node.
- Desperate mode: Replace “correct” or “optimum” by “best possible”.



## Desperate Mode

- What shall we do if computing  $x$  would require an invalid operation, e.g., a division by zero or taking the square root of a negative number?
- After all,  $x$  may be needed for subsequent computations! E.g., if a new Voronoi node is to be inserted into the Voronoi diagram then the code may subsequently need to access the coordinates of this node.
- Desperate mode: Replace “correct” or “optimum” by “best possible”.
  - Do not use any operation which is not defined for all fp-numbers.



## Desperate Mode

- What shall we do if computing  $x$  would require an invalid operation, e.g., a division by zero or taking the square root of a negative number?
- After all,  $x$  may be needed for subsequent computations! E.g., if a new Voronoi node is to be inserted into the Voronoi diagram then the code may subsequently need to access the coordinates of this node.
- Desperate mode: Replace “correct” or “optimum” by “best possible”.
  - Do not use any operation which is not defined for all fp-numbers.
  - Any violation of a topological condition is “cured” by forcing its validity. E.g., a cycle of Voronoi nodes can be broken by inserting a new dummy node on one of the edges of the cycle.



## Desperate Mode

- What shall we do if computing  $x$  would require an invalid operation, e.g., a division by zero or taking the square root of a negative number?
- After all,  $x$  may be needed for subsequent computations! E.g., if a new Voronoi node is to be inserted into the Voronoi diagram then the code may subsequently need to access the coordinates of this node.
- Desperate mode: Replace “correct” or “optimum” by “best possible”.
  - Do not use any operation which is not defined for all fp-numbers.
  - Any violation of a topological condition is “cured” by forcing its validity. E.g., a cycle of Voronoi nodes can be broken by inserting a new dummy node on one of the edges of the cycle.
  - If numerical data does not fulfill all numerical conditions then accept the data whichever meets most of them.

## Desperate Mode

- What shall we do if computing  $x$  would require an invalid operation, e.g., a division by zero or taking the square root of a negative number?
- After all,  $x$  may be needed for subsequent computations! E.g., if a new Voronoi node is to be inserted into the Voronoi diagram then the code may subsequently need to access the coordinates of this node.
- Desperate mode: Replace “correct” or “optimum” by “best possible”.
  - Do not use any operation which is not defined for all fp-numbers.
  - Any violation of a topological condition is “cured” by forcing its validity. E.g., a cycle of Voronoi nodes can be broken by inserting a new dummy node on one of the edges of the cycle.
  - If numerical data does not fulfill all numerical conditions then accept the data whichever meets most of them.
- Net result: A code that resorts to desperate mode will never get stuck, crash or loop.

## Desperate Mode

- What shall we do if computing  $x$  would require an invalid operation, e.g., a division by zero or taking the square root of a negative number?
- After all,  $x$  may be needed for subsequent computations! E.g., if a new Voronoi node is to be inserted into the Voronoi diagram then the code may subsequently need to access the coordinates of this node.
- Desperate mode: Replace “correct” or “optimum” by “best possible”.
  - Do not use any operation which is not defined for all fp-numbers.
  - Any violation of a topological condition is “cured” by forcing its validity. E.g., a cycle of Voronoi nodes can be broken by inserting a new dummy node on one of the edges of the cycle.
  - If numerical data does not fulfill all numerical conditions then accept the data whichever meets most of them.
- Net result: A code that resorts to desperate mode will never get stuck, crash or loop.
- Whether or not the output still is of practical use despite of desperate mode depends on the application and the type of problem that caused desperate mode.



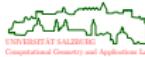
## Desperate Mode

- What shall we do if computing  $x$  would require an invalid operation, e.g., a division by zero or taking the square root of a negative number?
- After all,  $x$  may be needed for subsequent computations! E.g., if a new Voronoi node is to be inserted into the Voronoi diagram then the code may subsequently need to access the coordinates of this node.
- Desperate mode: Replace “correct” or “optimum” by “best possible”.
  - Do not use any operation which is not defined for all fp-numbers.
  - Any violation of a topological condition is “cured” by forcing its validity. E.g., a cycle of Voronoi nodes can be broken by inserting a new dummy node on one of the edges of the cycle.
  - If numerical data does not fulfill all numerical conditions then accept the data whichever meets most of them.
- Net result: A code that resorts to desperate mode will never get stuck, crash or loop.
- Whether or not the output still is of practical use despite of desperate mode depends on the application and the type of problem that caused desperate mode.
- Personal experience: Desperate mode works remarkably well for “incremental” algorithms.



## Desperate Mode: Caveats

- The relaxation of epsilon thresholds does not come for free: Repeatedly computing some data will increase the cpu-time consumption.



## Desperate Mode: Caveats

- The relaxation of epsilon thresholds does not come for free: Repeatedly computing some data will increase the cpu-time consumption.
- Always attempt to perform a computation as it would be performed normally, irrelevant of whether or not desperate mode was used once before.

## Desperate Mode: Caveats

- The relaxation of epsilon thresholds does not come for free: Repeatedly computing some data will increase the cpu-time consumption.
- Always attempt to perform a computation as it would be performed normally, irrelevant of whether or not desperate mode was used once before.
- Keep track of how often your code resorts to an actual relaxation: Frequent use of epsilon relaxation is a hint that the numerical reliability of your code is less than ideal.



## Desperate Mode: Caveats

- The relaxation of epsilon thresholds does not come for free: Repeatedly computing some data will increase the cpu-time consumption.
- Always attempt to perform a computation as it would be performed normally, irrelevant of whether or not desperate mode was used once before.
- Keep track of how often your code resorts to an actual relaxation: Frequent use of epsilon relaxation is a hint that the numerical reliability of your code is less than ideal.

### Warning

The availability of such a multi-phase recovery process can easily conceal genuine algorithmic flaws or bugs!



## Desperate Mode: Caveats

- The relaxation of epsilon thresholds does not come for free: Repeatedly computing some data will increase the cpu-time consumption.
- Always attempt to perform a computation as it would be performed normally, irrelevant of whether or not desperate mode was used once before.
- Keep track of how often your code resorts to an actual relaxation: Frequent use of epsilon relaxation is a hint that the numerical reliability of your code is less than ideal.

### Warning

The availability of such a multi-phase recovery process can easily conceal genuine algorithmic flaws or bugs!

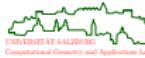
### Important advice

Always test your code with desperate mode being disabled! Robustness without desperate mode is a must for *all* tests on *your* test data!



## Adding an MPFR Backend

- GNU's MPFR library ([www.mpfr.org](http://www.mpfr.org)) is a C library for multiple-precision floating-point computations.



## Adding an MPFR Backend

- GNU's MPFR library ([www.mpfr.org](http://www.mpfr.org)) is a C library for multiple-precision floating-point computations.
- Canonical adaptions:
  - Precision threshold  $\text{EPS}$  needs to depend on MPFR precision.
  - [Held&Huber 2013] use a heuristic formula:

$$\text{EPS} := \epsilon_{\text{fp}} \cdot 2^{-100 \cdot (\sqrt{\text{prec}/53} - 1)},$$

where  $\epsilon_{\text{fp}}$  is the former machine-precision  $\text{EPS}$ .

## Adding an MPFR Backend

- GNU's MPFR library ([www.mpfr.org](http://www.mpfr.org)) is a C library for multiple-precision floating-point computations.
- Canonical adaptions:
  - Precision threshold `EPS` needs to depend on MPFR precision.
  - [Held&Huber 2013] use a heuristic formula:

$$\text{EPS} := \epsilon_{\text{fp}} \cdot 2^{-100 \cdot (\sqrt{\text{prec}/53} - 1)},$$

where  $\epsilon_{\text{fp}}$  is the former machine-precision `EPS`.

- Subtle problem encountered: `mpfr_set_default_prec` does not change existing variables.
  - Global variables are not adjusted.



## Robustness Issues

- Introduction to Robustness Problems
- Approaches to Achieving Robustness
- Improving the Reliability of FP-Code
- Industrial Applications
  - Industrial Requirements
  - Experimental Results for Triangulations
  - Experimental Results for Voronoi Diagrams
- Using FP-Arithmetic to Implement the Turing Test

## Datasets from Industry

- Real-world data often means no quality at all:
  - raster-to-vector conversions,
  - data cleaned up manually or “visually”,
  - data preprocessed by some dubious program of unknown origin,
  - data comes from “an important customer” or from “God knows where”,
  - brute-force simplifications of complex datasets,
  - brute-force approximations of free-form curves.



## Datasets from Industry

- Real-world data often means no quality at all:
  - raster-to-vector conversions,
  - data cleaned up manually or “visually”,
  - data preprocessed by some dubious program of unknown origin,
  - data comes from “an important customer” or from “God knows where”,
  - brute-force simplifications of complex datasets,
  - brute-force approximations of free-form curves.
- As a consequence:
  - all sorts of degeneracies.
  - self-intersections.
  - tiny gaps in supposedly closed contours.



# Datasets from Industry

- Real-world data often means no quality at all:
  - raster-to-vector conversions,
  - data cleaned up manually or “visually”,
  - data preprocessed by some dubious program of unknown origin,
  - data comes from “an important customer” or from “God knows where”,
  - brute-force simplifications of complex datasets,
  - brute-force approximations of free-form curves.
- As a consequence:
  - all sorts of degeneracies.
  - self-intersections.
  - tiny gaps in supposedly closed contours.

## Advice

Be prepared for troubles — general position must not be assumed!

# Datasets from Industry

- Real-world data often means no quality at all:
  - raster-to-vector conversions,
  - data cleaned up manually or “visually”,
  - data preprocessed by some dubious program of unknown origin,
  - data comes from “an important customer” or from “God knows where”,
  - brute-force simplifications of complex datasets,
  - brute-force approximations of free-form curves.
- As a consequence:
  - all sorts of degeneracies.
  - self-intersections.
  - tiny gaps in supposedly closed contours.

## Advice

Be prepared for troubles — general position must not be assumed!

- Data sizes vary substantially from a few thousand segments/arcs in a machining application to a few million segments in a GIS application.



# Industrial Requirements

## Efficiency requirements:

- Efficiency requirements also vary substantially from real-time map generation on a smart phone to minutes of CPU time allowed on some high-end machine.
- In general, linear space complexity and a close-to-linear time complexity is expected.

# Industrial Requirements

## Efficiency requirements:

- Efficiency requirements also vary substantially from real-time map generation on a smart phone to minutes of CPU time allowed on some high-end machine.
- In general, linear space complexity and a close-to-linear time complexity is expected.

## Parallelization requirements:

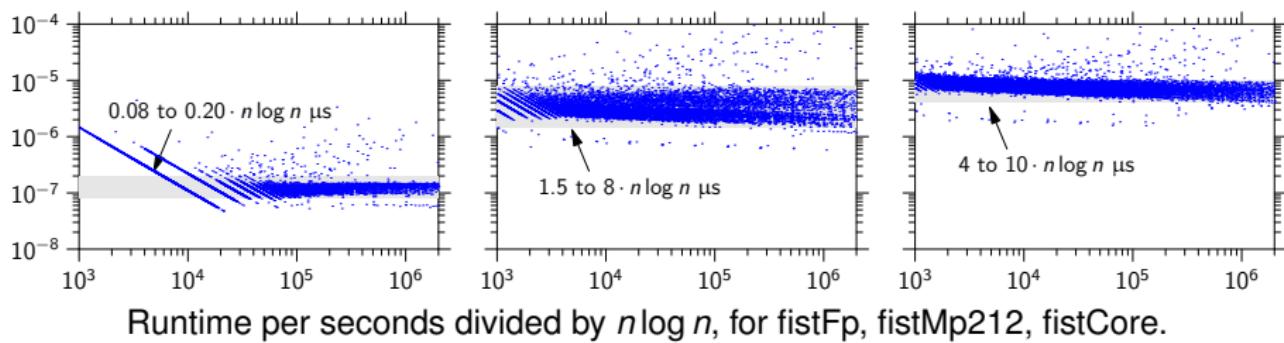
- Absolutely no inquiry concerning GPU-based codes so far.
- Only moderate interest in multi-core computing and multi-threaded implementations.

## Experimental Results for Triangulations

- 21 175 polygons with and without holes.
- Six arithmetic configurations:
  - fistFp, fistShew, fistCore, fistMp{53, 212, 1000}

# Experimental Results for Triangulations

- 21 175 polygons with and without holes.
- Six arithmetic configurations:
  - fistFp, fistShew, fistCore, fistMp{53, 212, 1000}
- Conclusion:
  - Shewchuck's predicates have negligible impact on speed.
  - fistMP\* about 24 times slower than fistFp.
  - fistCore about 60 times slower than fistFp.



## Experimental Results for Triangulations

[Held&Mann 2011]: Correctness of inexact configurations?

- Verification code:
  - Bentley-Ottmann algorithm, implemented with exact `mpq_t` from GMP.



## Experimental Results for Triangulations

[Held&Mann 2011]: Correctness of inexact configurations?

- Verification code:
  - Bentley-Ottmann algorithm, implemented with exact `mpq_t` from GMP.
- **Scenario 1:**
  - We interpret `0 . 1` in input files as  $\frac{1}{10}$ .
  - 4.9% of all results faulty, uniformly across all non-CORE configurations.
    - Including `fistShew`.
  - But: Error only visible at huge zoom factors.



# Experimental Results for Triangulations

[Held&Mann 2011]: Correctness of inexact configurations?

- Verification code:
  - Bentley-Ottmann algorithm, implemented with exact `mpq_t` from GMP.
- **Scenario 1:**
  - We interpret `0.1` in input files as  $\frac{1}{10}$ .
  - 4.9% of all results faulty, uniformly across all non-CORE configurations.
    - Including fistShew.
  - But: Error only visible at huge zoom factors.
- **Scenario 2:**
  - We take `0.1` as closest fp-number using `atof()`.
  - No errors found!

# Experimental Results for Triangulations

[Held&Mann 2011]: Correctness of inexact configurations?

- Verification code:
  - Bentley-Ottmann algorithm, implemented with exact `mpq_t` from GMP.
- **Scenario 1:**
  - We interpret `0.1` in input files as  $\frac{1}{10}$ .
  - 4.9% of all results faulty, uniformly across all non-CORE configurations.
    - Including `fistShew`.
  - But: Error only visible at huge zoom factors.
- **Scenario 2:**
  - We take `0.1` as closest fp-number using `atof()`.
  - No errors found!

## Conclusion

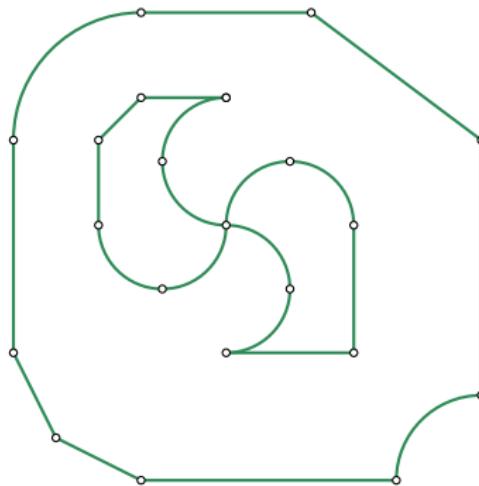
Non-exactness need not be a practical issue in pure fp-applications.

## Software for Computing Voronoi Diagrams: CGAL

- Segment-Delaunay-Graph implemented within CGAL by [Karavelas 2004].
- Input:
  - Points and straight-line segments;
  - Input sites may intersect arbitrarily;
  - No support (yet?) for circular arcs.
- Incremental construction.
- Complexity:
  - $O((n + m) \log^2 n)$  expected time,  
where  $n$  is the number of sites and  $m$  is the number of intersections.
  - $O((n + m) \log n)$  in practice.
- CPU-time consumption is claimed to be mostly insensitive to the number type used.

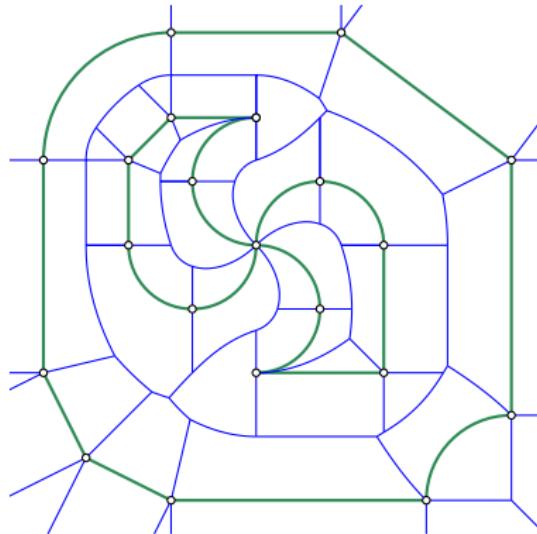
## Software for Computing Voronoi Diagrams: VRONI and ARCVRONI

- [Held 2001, Held&Huber 2009]: Computes Voronoi diagrams
  - of points, straight-line segments and circular arcs,
  - based on randomized incremental insertion and a topology-oriented approach.



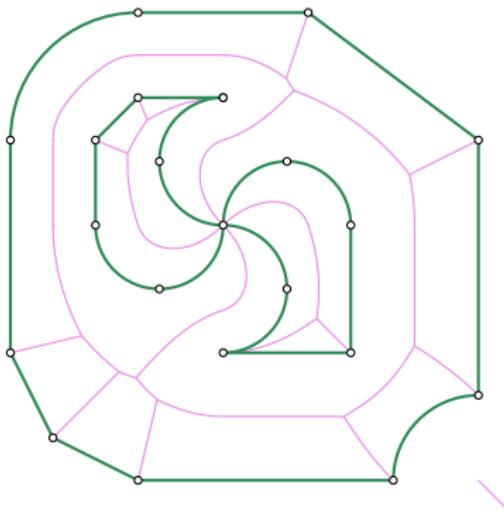
## Software for Computing Voronoi Diagrams: VRONI and ARCVRONI

- [Held 2001, Held&Huber 2009]: Computes Voronoi diagrams
  - of points, straight-line segments and circular arcs,
  - based on randomized incremental insertion and a topology-oriented approach.



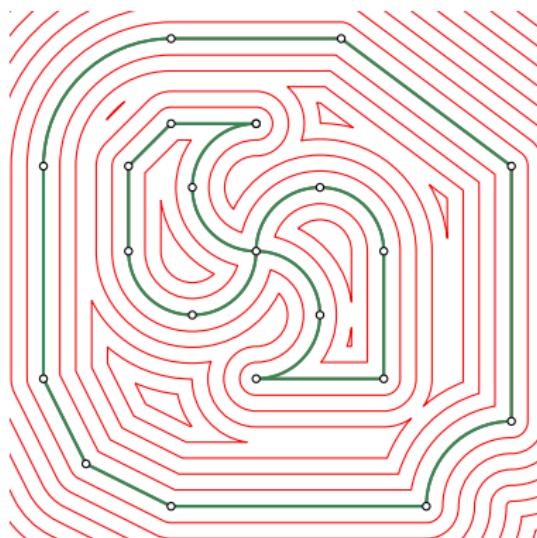
# Software for Computing Voronoi Diagrams: VRONI and ARCVRONI

- [Held 2001, Held&Huber 2009]: Computes Voronoi diagrams
  - of points, straight-line segments and circular arcs,
  - based on randomized incremental insertion and a topology-oriented approach.
- Also computes
  - (weighted) medial axis,



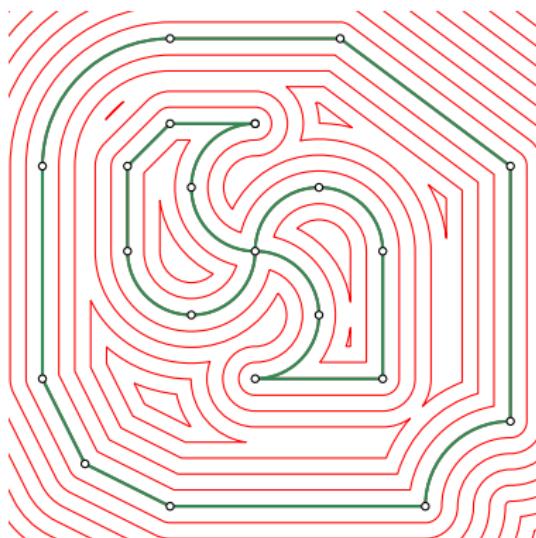
# Software for Computing Voronoi Diagrams: VRONI and ARCVRONI

- [Held 2001, Held&Huber 2009]: Computes Voronoi diagrams
  - of points, straight-line segments and circular arcs,
  - based on randomized incremental insertion and a topology-oriented approach.
- Also computes
  - (weighted) medial axis,
  - offset curves, and
  - maximum-inscribed circle.



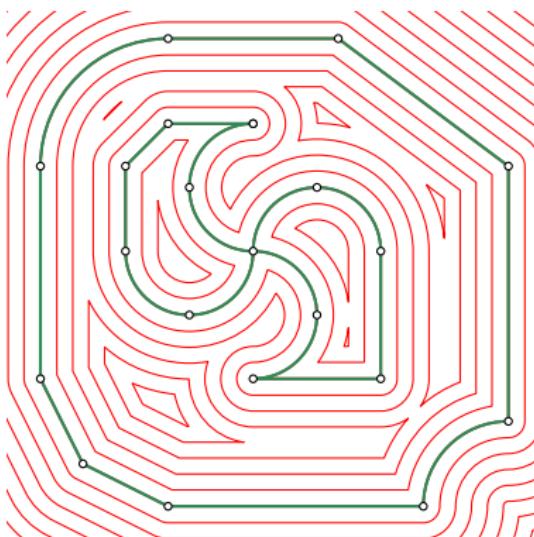
## Software for Computing Voronoi Diagrams: VRONI and ARCVRONI

- [Held 2001, Held&Huber 2009]: Computes Voronoi diagrams
  - of points, straight-line segments and circular arcs,
  - based on randomized incremental insertion and a topology-oriented approach.
- Also computes
  - (weighted) medial axis,
  - offset curves, and
  - maximum-inscribed circle.
- Complexity:
  - $O((n + m) \log n)$  expected time, where  $n$  is the number of sites and  $m$  is the number of intersections.



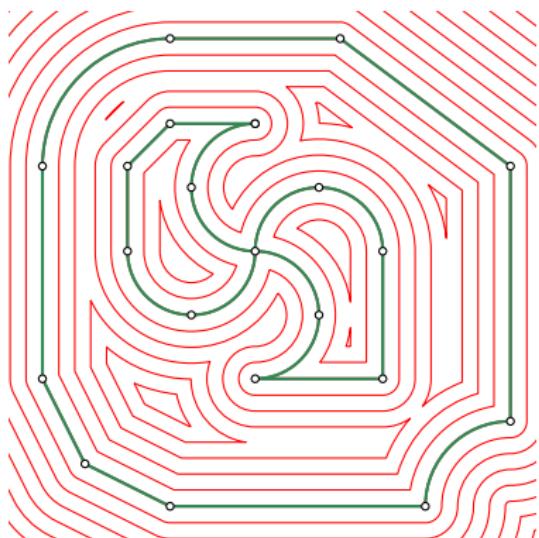
# Software for Computing Voronoi Diagrams: VRONI and ARCVRONI

- [Held 2001, Held&Huber 2009]: Computes Voronoi diagrams
  - of points, straight-line segments and circular arcs,
  - based on randomized incremental insertion and a topology-oriented approach.
- Also computes
  - (weighted) medial axis,
  - offset curves, and
  - maximum-inscribed circle.
- Complexity:
  - $O((n + m) \log n)$  expected time, where  $n$  is the number of sites and  $m$  is the number of intersections.
- Based on standard IEEE 754 floating-point arithmetic, with epsilon relaxation and desperate mode.



# Software for Computing Voronoi Diagrams: VRONI and ARCVRONI

- [Held 2001, Held&Huber 2009]: Computes Voronoi diagrams
  - of points, straight-line segments and circular arcs,
  - based on randomized incremental insertion and a topology-oriented approach.
- Also computes
  - (weighted) medial axis,
  - offset curves, and
  - maximum-inscribed circle.
- Complexity:
  - $O((n + m) \log n)$  expected time, where  $n$  is the number of sites and  $m$  is the number of intersections.
- Based on standard IEEE 754 floating-point arithmetic, with epsilon relaxation and desperate mode.
- Typical applications in industry: generation of tool paths (e.g., for machining or sintering), generation of buffers in GIS applications.



# Experimental Results for Voronoi Diagrams

## Codes tested:

- CGAL 3.8 (cgvdEx, cgvdFp):
  - CORE::Expr as predicate kernel.
  - Segment\_Delaunay\_graph\_filtered\_traits\_2 template parameter to the underlying segment Delaunay graph class.
  - Graphics disabled, input stream-lined, own timing routine added.
  - Compiled with `g++ -O2`.
- VRONI 6.0 (vroniFp, vroniMp{53, 212, 1000}):
  - Also compiled with `g++ -O2`.

# Experimental Results for Voronoi Diagrams

## Codes tested:

- CGAL 3.8 (cgvdEx, cgvdFp):
  - CORE::Expr as predicate kernel.
  - Segment\_Delaunay\_graph\_filtered\_traits\_2 template parameter to the underlying segment Delaunay graph class.
  - Graphics disabled, input stream-lined, own timing routine added.
  - Compiled with `g++ -O2`.
- VRONI 6.0 (vroniFp, vroniMp{53, 212, 1000}):
  - Also compiled with `g++ -O2`.

## Test platform:

- 3.33GHz i7 CPU X 980; 24GB RAM; 64bit Ubuntu.
- All timings given in CPU microseconds.

# Experimental Results for Voronoi Diagrams

## Codes tested:

- CGAL 3.8 (cgvdEx, cgvdFp):
  - CORE::Expr as predicate kernel.
  - Segment\_Delaunay\_graph\_filtered\_traits\_2 template parameter to the underlying segment Delaunay graph class.
  - Graphics disabled, input stream-lined, own timing routine added.
  - Compiled with `g++ -O2`.
- VRONI 6.0 (vroniFp, vroniMp{53, 212, 1000}):
  - Also compiled with `g++ -O2`.

## Test platform:

- 3.33GHz i7 CPU X 980; 24GB RAM; 64bit Ubuntu.
- All timings given in CPU microseconds.

## Test data:

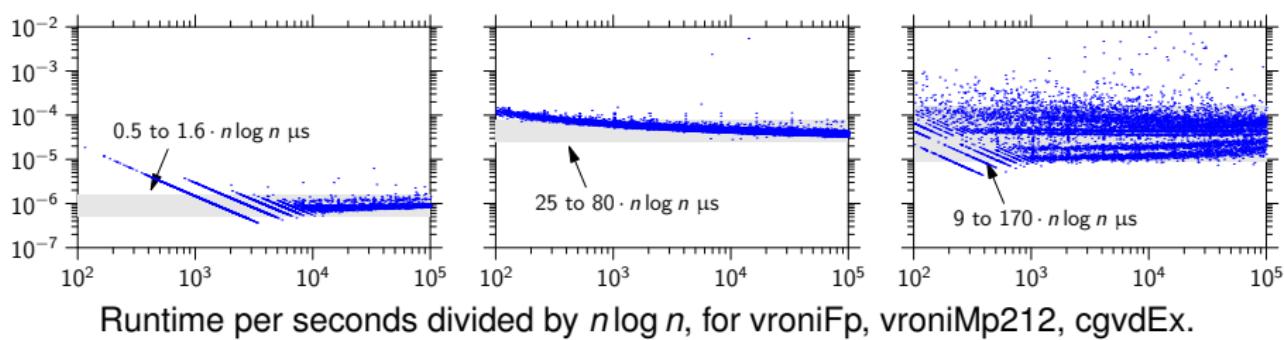
- Synthetic and real-world data; circular arcs approximated by polygonal chains.
- 18 787 data sets tested, with  $200 \leq \#(\text{segs}) \leq 100\,000$ .



# Experimental Results for Voronoi Diagrams

- Conclusion:

- vroniFp about 50–70 times faster than vroniMp\*.
- vroniFp about 50–80 times faster than cgvd\*.
- cgvdFp only 1.5 times faster than cgvdEx.
  - Crashed on 937 datasets due to fp-exception.
- On average, cgvdEx slightly faster than vroniMp\*.
  - cgvdEx timings vary by a factor of 20.
  - A few cgvdEx results were numerically clearly wrong.



## Experimental Results for Voronoi Diagrams

- [Held&Mann 2011]: Implemented a verifier based on GMP's `mpq_t` data type.
- Brute-force all-pairs distance computations used.
- Hence, verification was limited to small inputs with up to 2 000 segments.

## Experimental Results for Voronoi Diagrams

- [Held&Mann 2011]: Implemented a verifier based on GMP's `mpq_t` data type.
- Brute-force all-pairs distance computations used.
- Hence, verification was limited to small inputs with up to 2 000 segments.

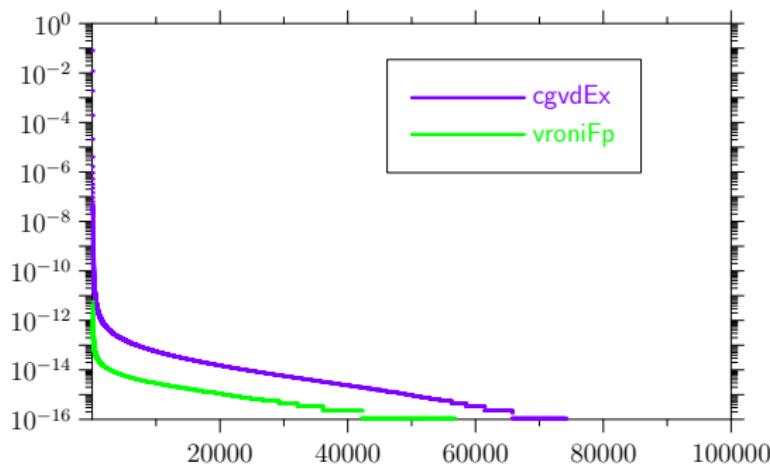
**Deviation:** Difference in the distances of a node to its defining sites.

**Violation:** Another site is closer to a node than defining sites.



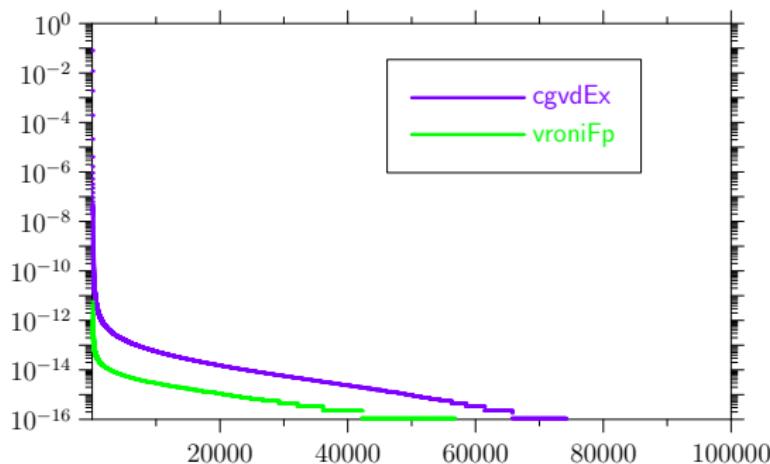
# Experimental Results for Voronoi Diagrams

- Deviation: Difference in the distances of a node to its defining sites.



# Experimental Results for Voronoi Diagrams

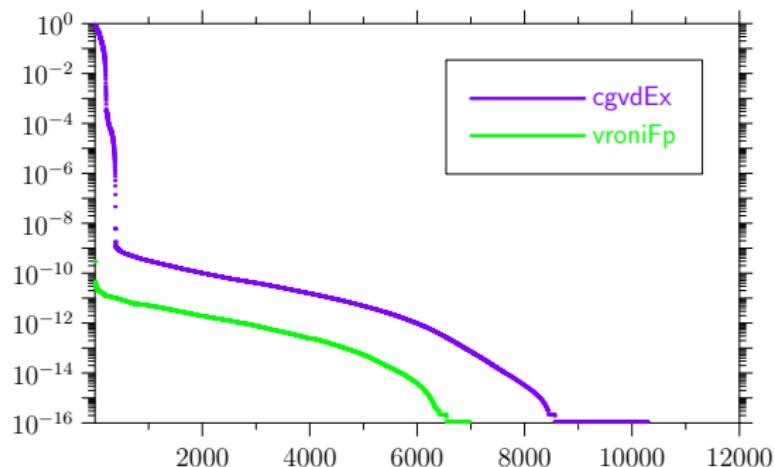
- Deviation: Difference in the distances of a node to its defining sites.



	VRONI	CGAL
maximum:	$7.25 \cdot 10^{-8}$	$8.04 \cdot 10^{-1}$
median:	$2.22 \cdot 10^{-16}$	$3.10 \cdot 10^{-15}$

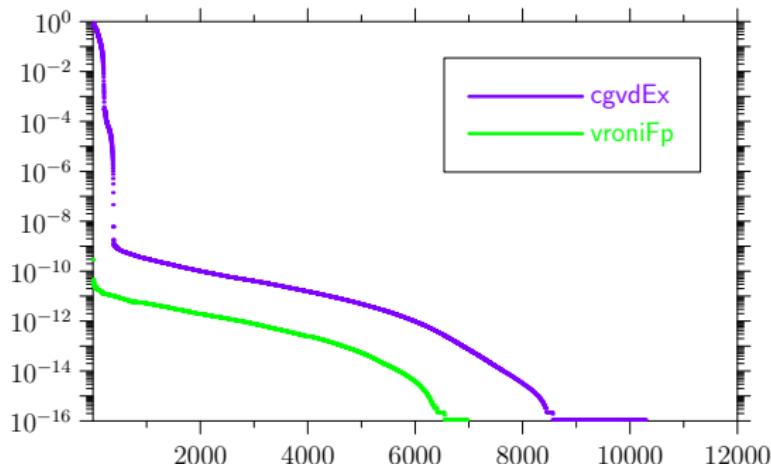
# Experimental Results for Voronoi Diagrams

- Violation: Another site is closer to a node than defining sites.



# Experimental Results for Voronoi Diagrams

- Violation: Another site is closer to a node than defining sites.



	VRONI	CGAL
maximum:	$1.75 \cdot 10^{-6}$	$9.14 \cdot 10^{-1}$
median:	$3.87 \cdot 10^{-14}$	$3.94 \cdot 10^{-12}$

## Robustness Issues

- Introduction to Robustness Problems
- Approaches to Achieving Robustness
- Improving the Reliability of FP-Code
- Industrial Applications
- Using FP-Arithmetic to Implement the Turing Test

# A Turing Test Based on FP-Arithmetic: Are You Human?

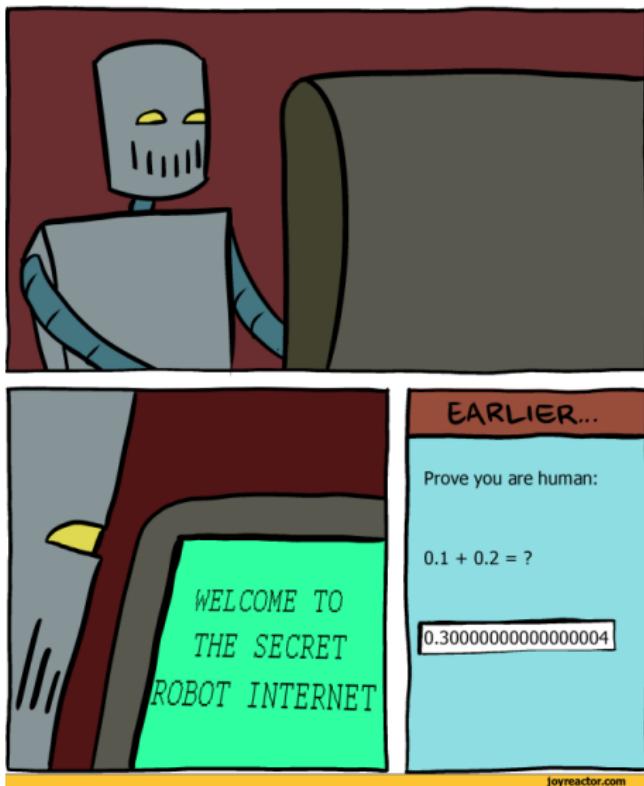


Image credit: <http://joyreactor.com/post/818128>

# The End!

I hope that you enjoyed this course, and I wish you all the best for your future studies.

