DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

TRIANGULATIONS

PETR FELKEL

FEL CTU PRAGUE

felkel@fel.cvut.cz

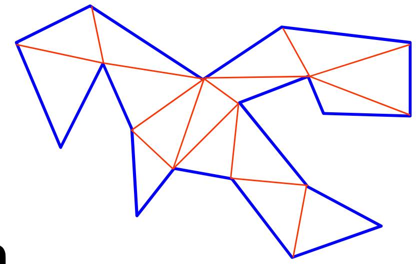
https://cw.felk.cvut.cz/doku.php/courses/a4m39vg/start

Based on [Berg] and [Mount]

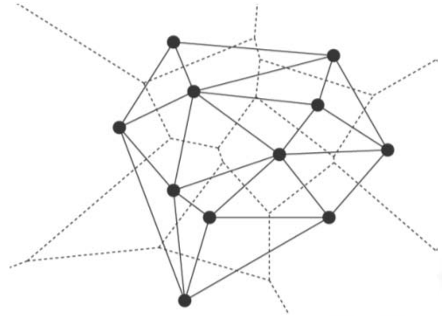
Version from 21.11.2021

Talk overview

- Polygon triangulation
  - Monotone polygon triangulation
  - Monotonization of non-monotone polygon

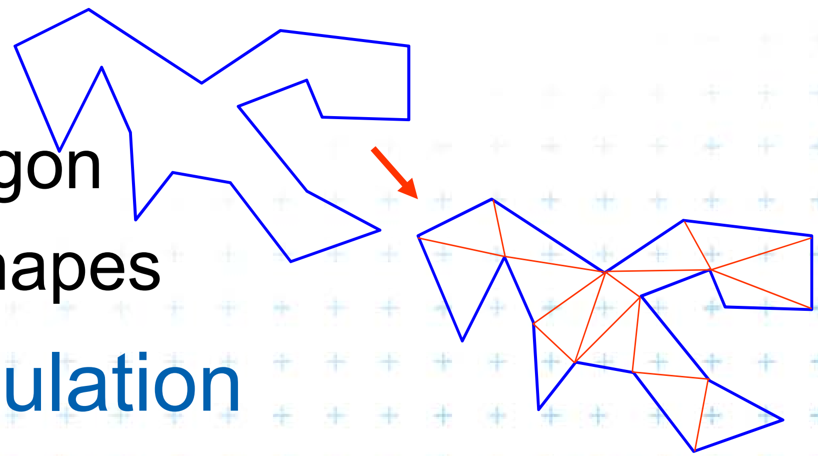


- Delaunay triangulation (DT) of points
  - Input: set of 2D points
  - Properties
  - Incremental Algorithm
  - Relation of DT in 2D and lower envelope (CH) in 3D
    and
    relation of VD in 2D to upper envelope in 3D



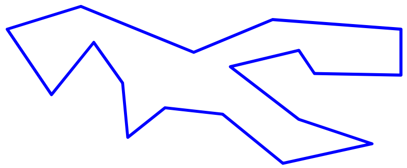
Polygon triangulation problem

- Triangulation (in general)
  = subdividing a spatial domain into simplices
- Application
  – decomposition of complex shapes into simpler shapes
  – art gallery problem (how many cameras and where)
- We will discuss
  – Triangulation of a simple polygon
  – without demand on triangle shapes



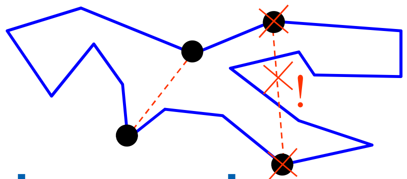
- Complexity of polygon triangulation
  – O(n) alg. exists [Chazelle91], but it is too complicated
  – practical algorithms run in O(n log n)

Terminology



Simple polygon

= region enclosed by a closed polygonal chain that does not intersect itself



Visible points

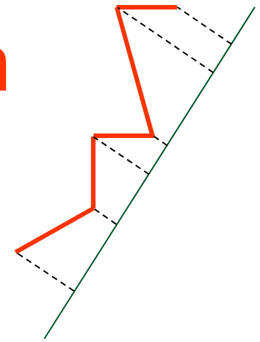
= two points on the boundary are visible if the interior of the line segment joining them lies entirely in the interior of the polygon

Diagonal

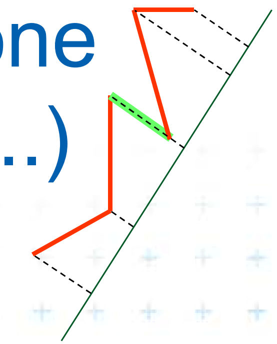
= line segment joining any pair of visible vertices

Terminology

- A polygonal chain C is strictly monotone with respect to line L, if any line orthogonal to L intersects C in at most one *point*



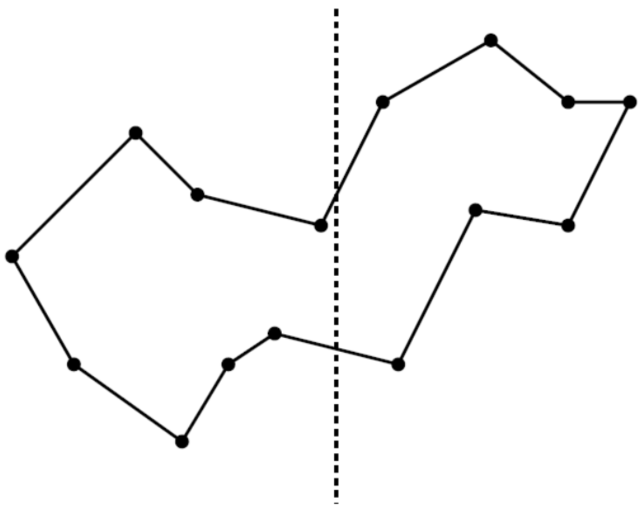
- A chain C is monotone with respect to line L, if any line orthogonal to L intersects C in at most one *connected component* (point, line segment,...)



- Polygon P is monotone with respect to line L, if its boundary (bnd(P), ∂P) can be split into two chains, each of which is monotone with respect to L

Terminology

- Horizontally monotone polygon
  = monotone with respect to *x*-axis
  - Can be tested in $O(n)$
  - Find leftmost and rightmost point in $O(n)$
  - Split boundary to upper and lower chain
  - Walk left to right, verifying that x-coord are non-decreasing

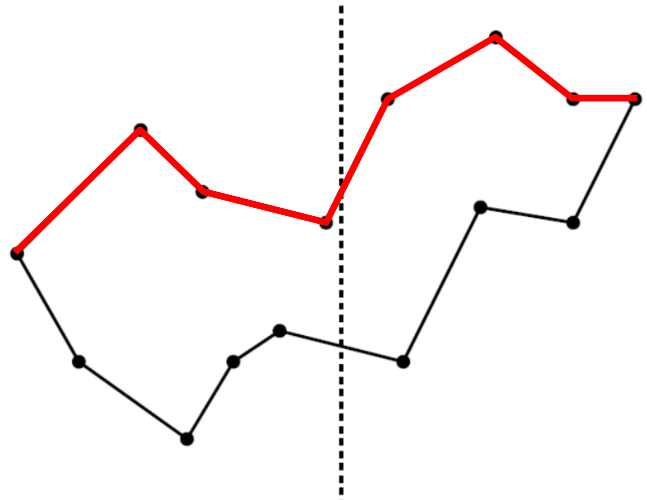


x–monotone polygon

[Mount]

Terminology

- Horizontally monotone polygon
  = monotone with respect to *x*-axis
  – Can be tested in $O(n)$
  – Find leftmost and rightmost point in $O(n)$
  – Split boundary to upper and lower chain
  – Walk left to right, verifying that x-coord are non-decreasing

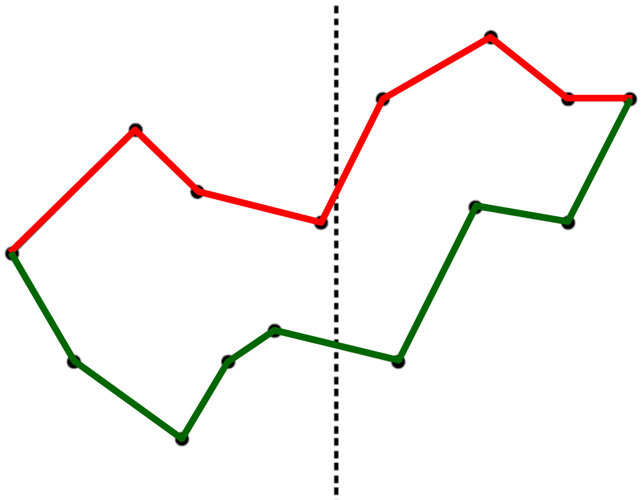


x–monotone polygon

[Mount]

Terminology

- Horizontally monotone polygon
  = monotone with respect to *x*-axis
  – Can be tested in $O(n)$
  – Find leftmost and rightmost point in $O(n)$
  – Split boundary to upper and lower chain
  – Walk left to right, verifying that x-coord are non-decreasing



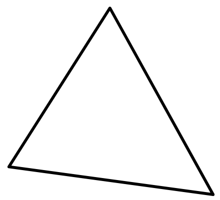
x–monotone polygon

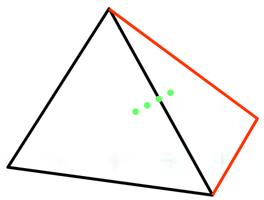
[Mount]

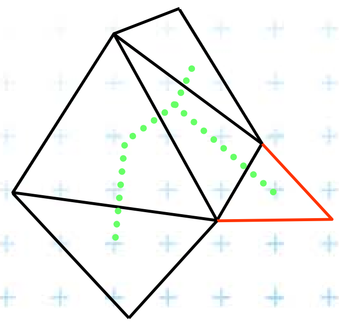
Terminology

- Every simple polygon can be triangulated
- Simple polygon with n vertices consists of
  - exactly $n - 2$ triangles
  - exactly $n - 3$ diagonals
  - Each diagonal is added once
    $\Rightarrow O(n)$ sweep line algorithm exist

Proof by induction







$n = 3 \Rightarrow 0$ diagonal

$n = 4 \Rightarrow 1$ diagonal
$n - 3$

$n := n + 1 \Rightarrow n + 1 - 3$ diagonals
$n + 1 = 7 \Rightarrow 4$ diagonals)

Simple polygon triangulation

- Simple polygon can be triangulated in 2 steps:
  1. Partition the polygon into x-monotone pieces
  2. Triangulate all monotone pieces

(we will discuss the steps in the reversed order)

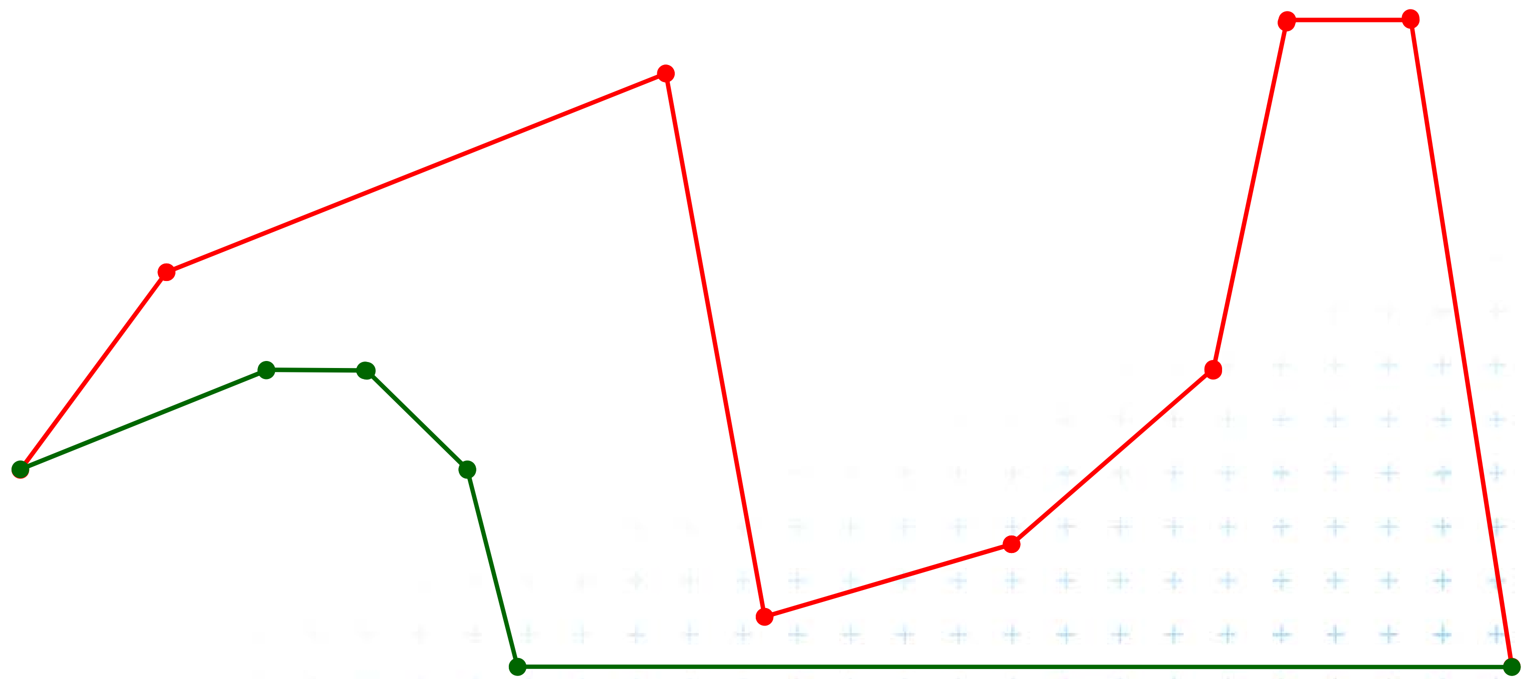
Simple polygon triangulation

- Simple polygon can be triangulated in 2 steps:
  1. Partition the polygon into x-monotone pieces
  2. Triangulate all monotone pieces

(we will discuss the steps in the reversed order)

x- monotone polygon triangulation principle

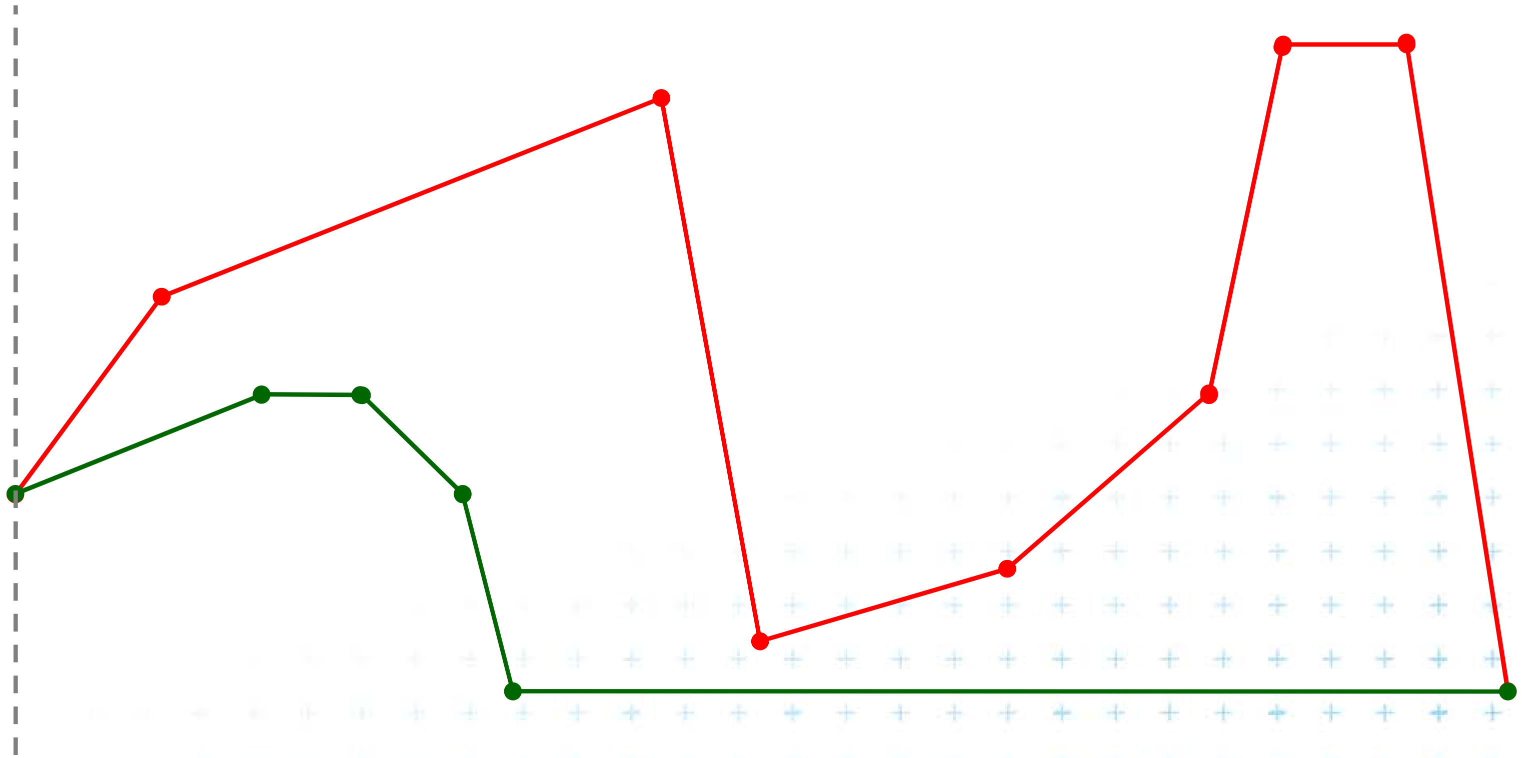
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

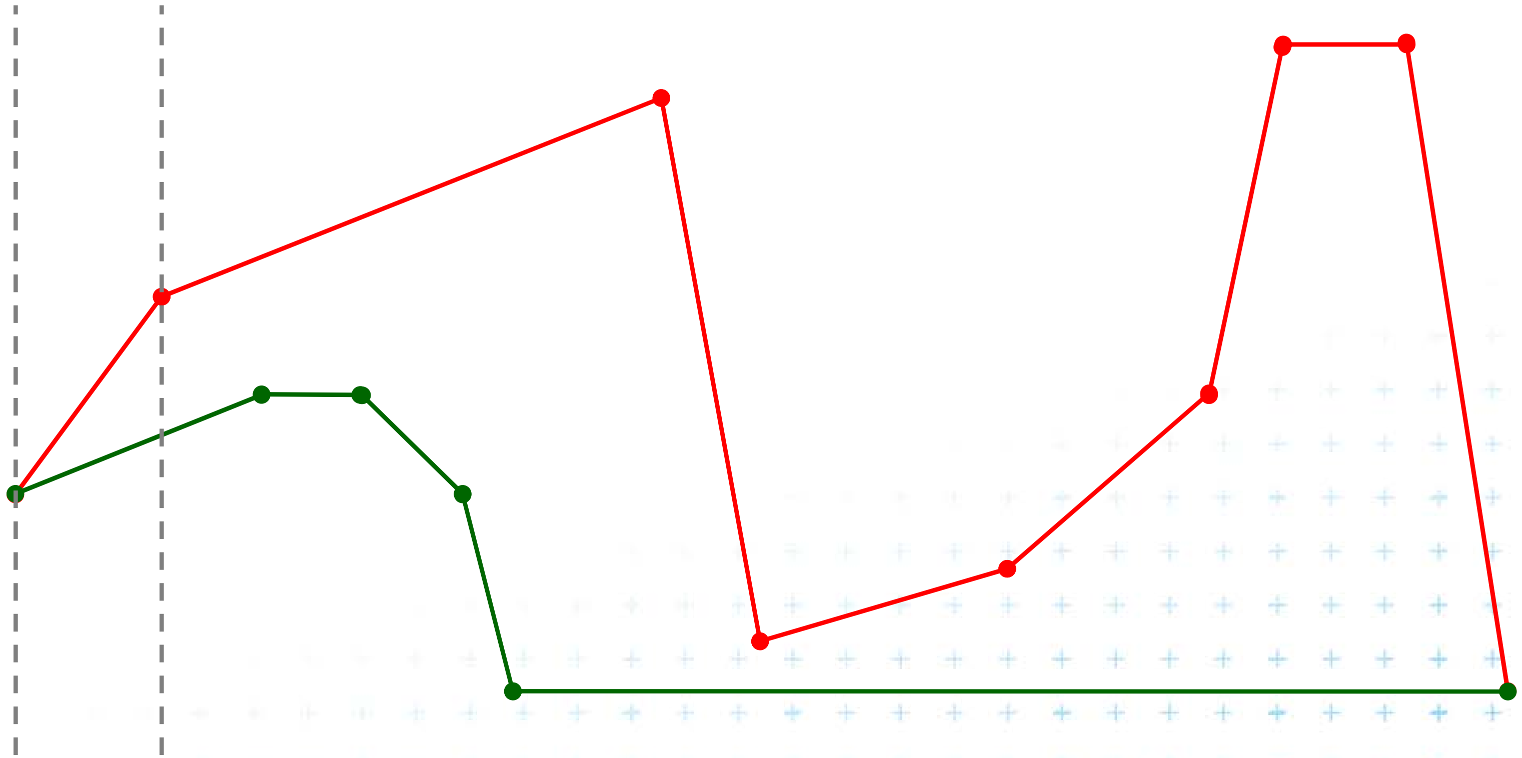
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

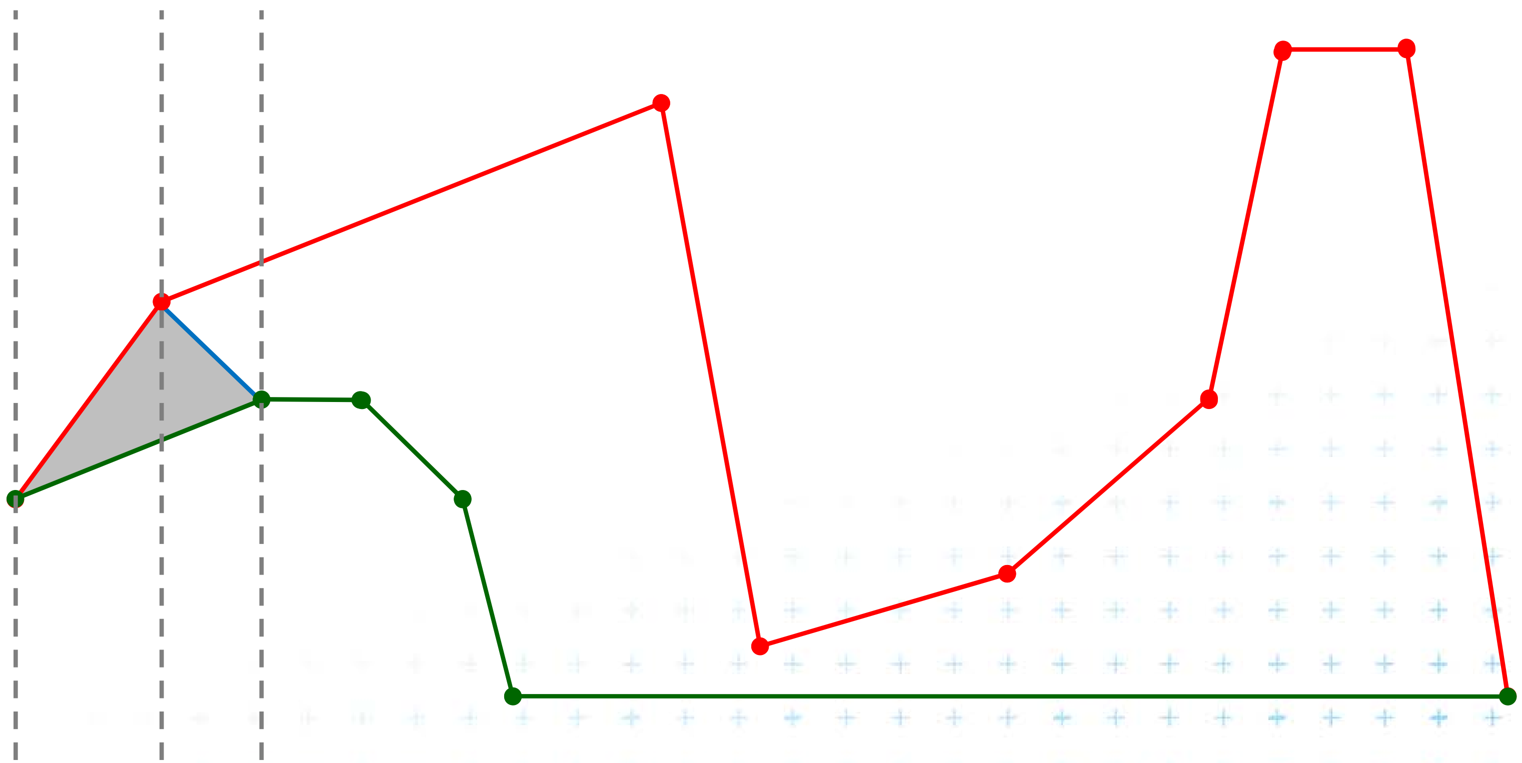
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

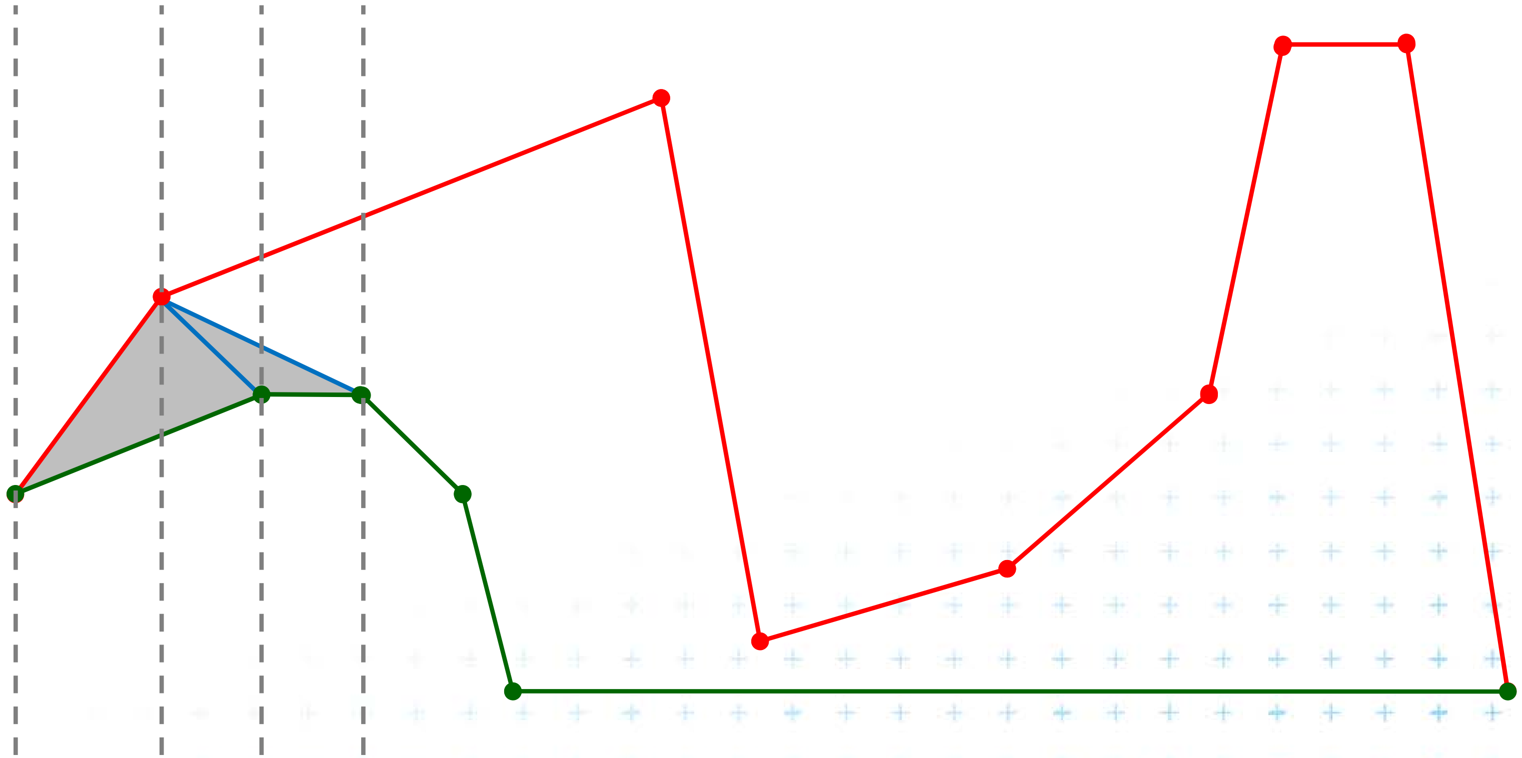
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

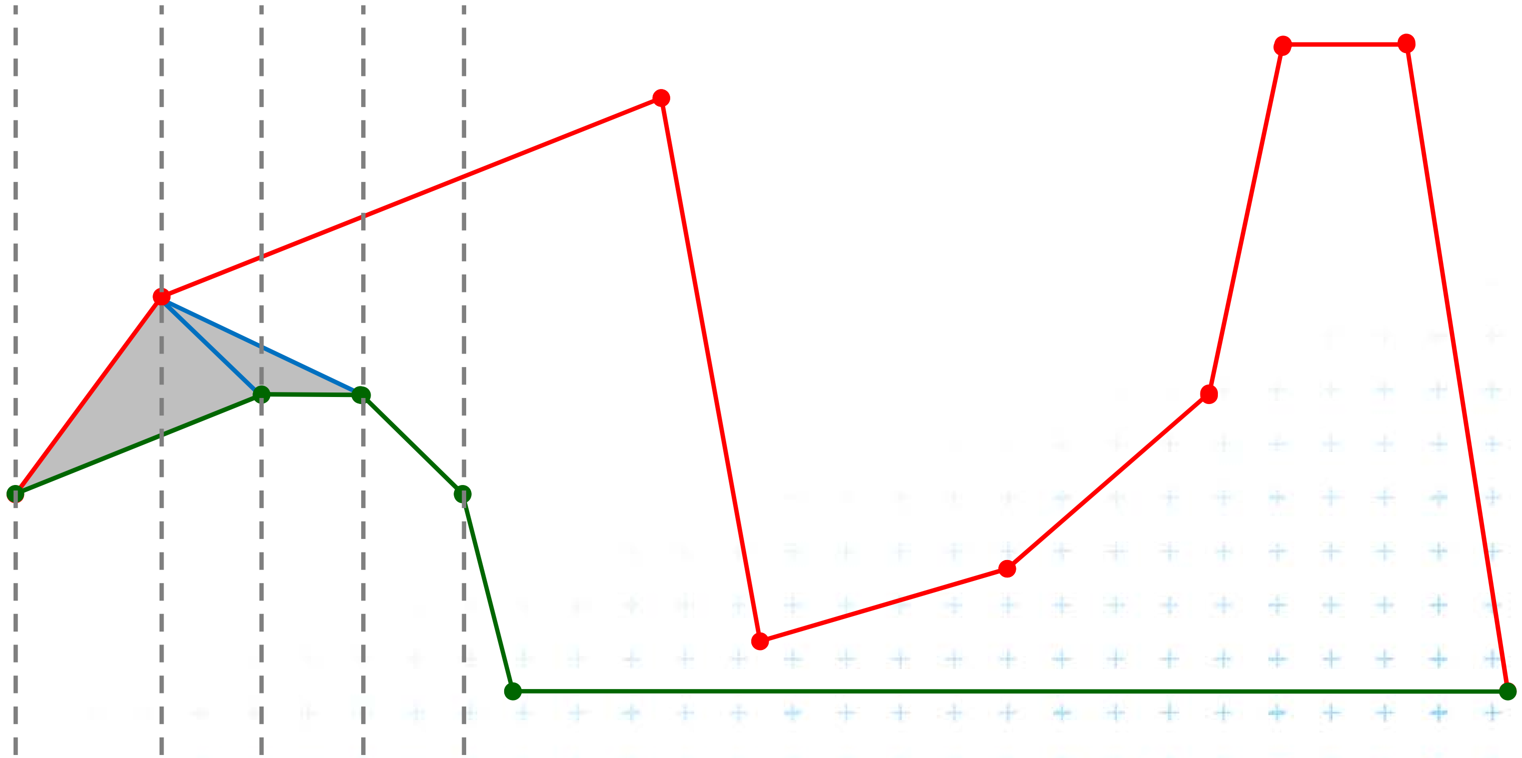
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

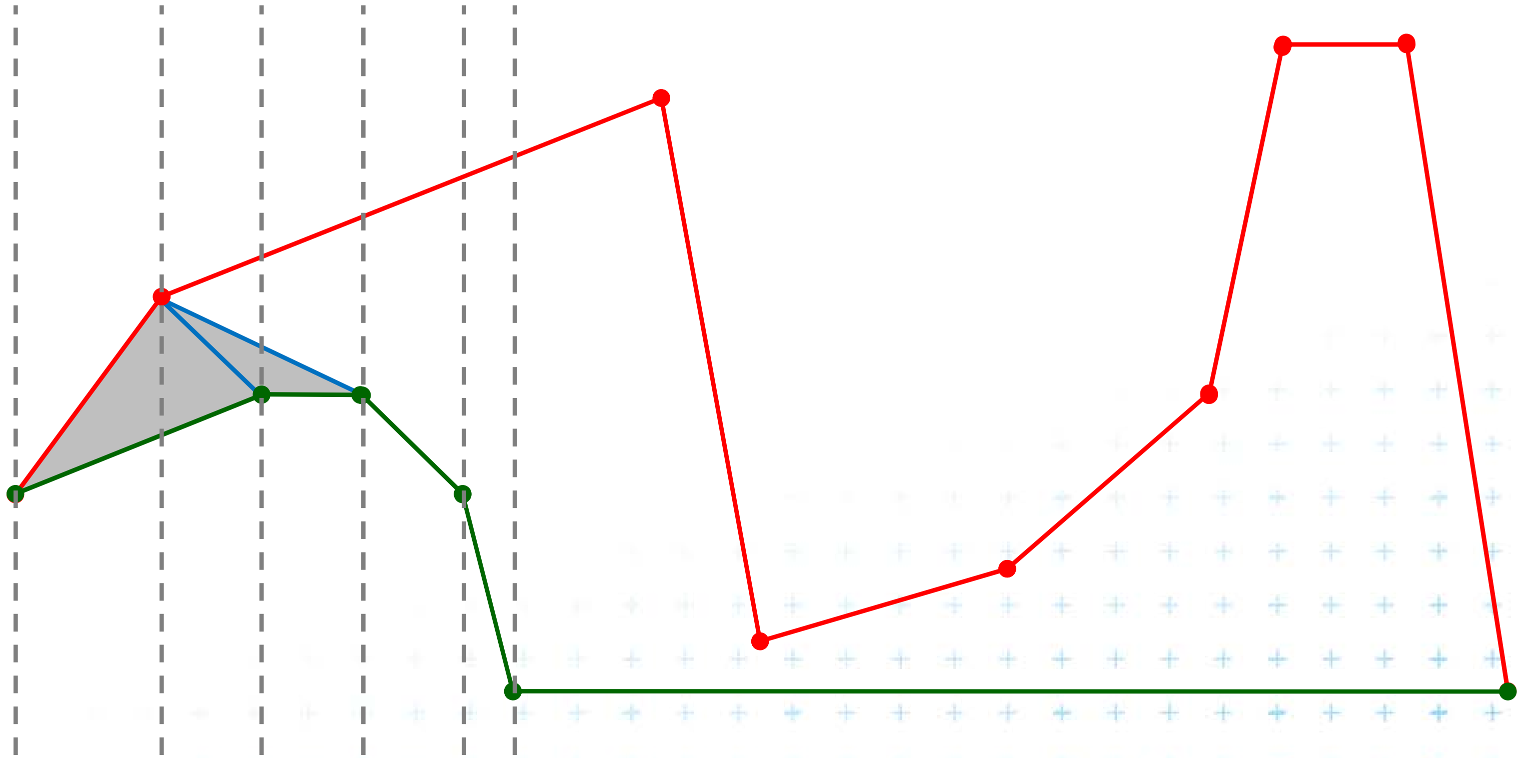
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

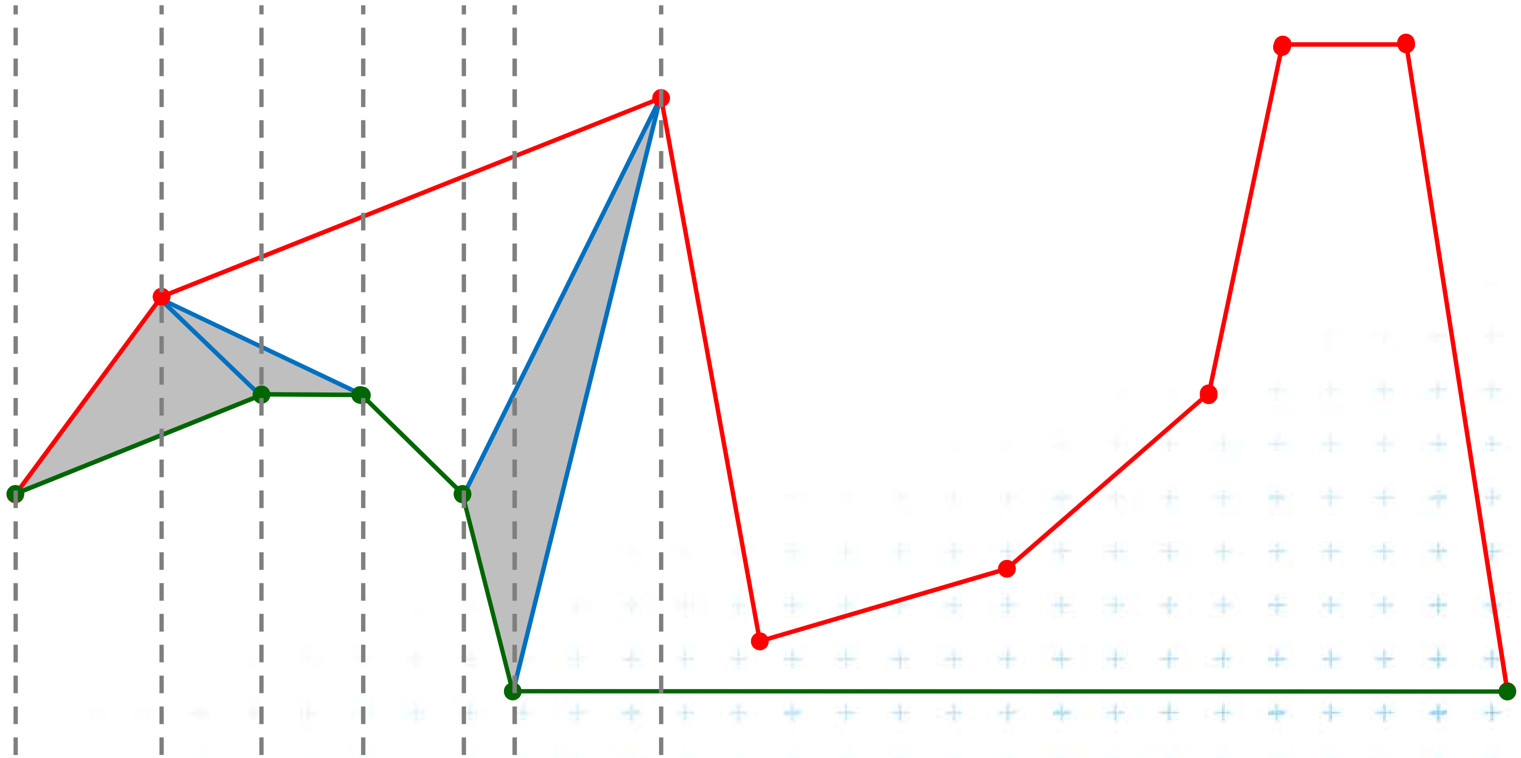
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

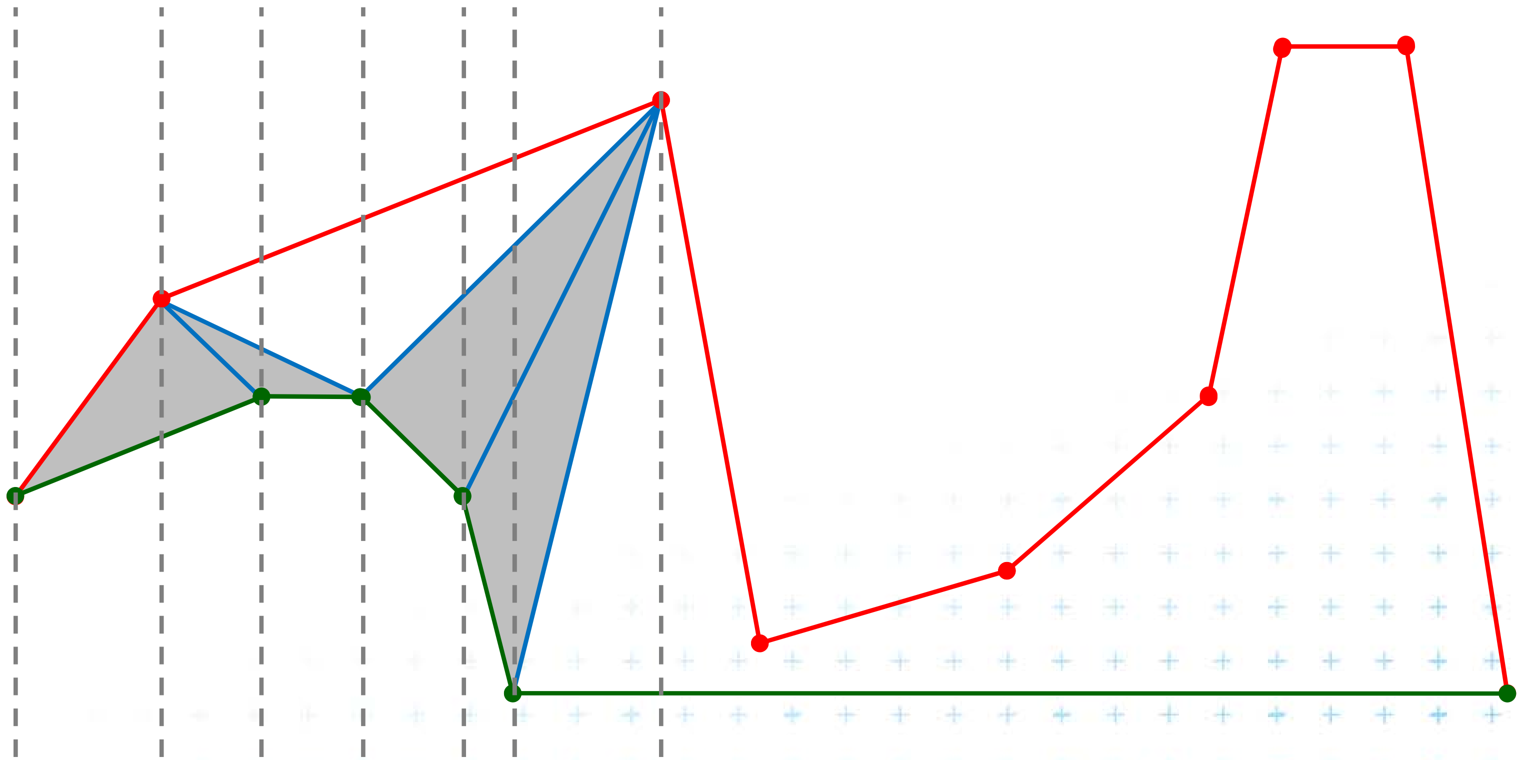
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

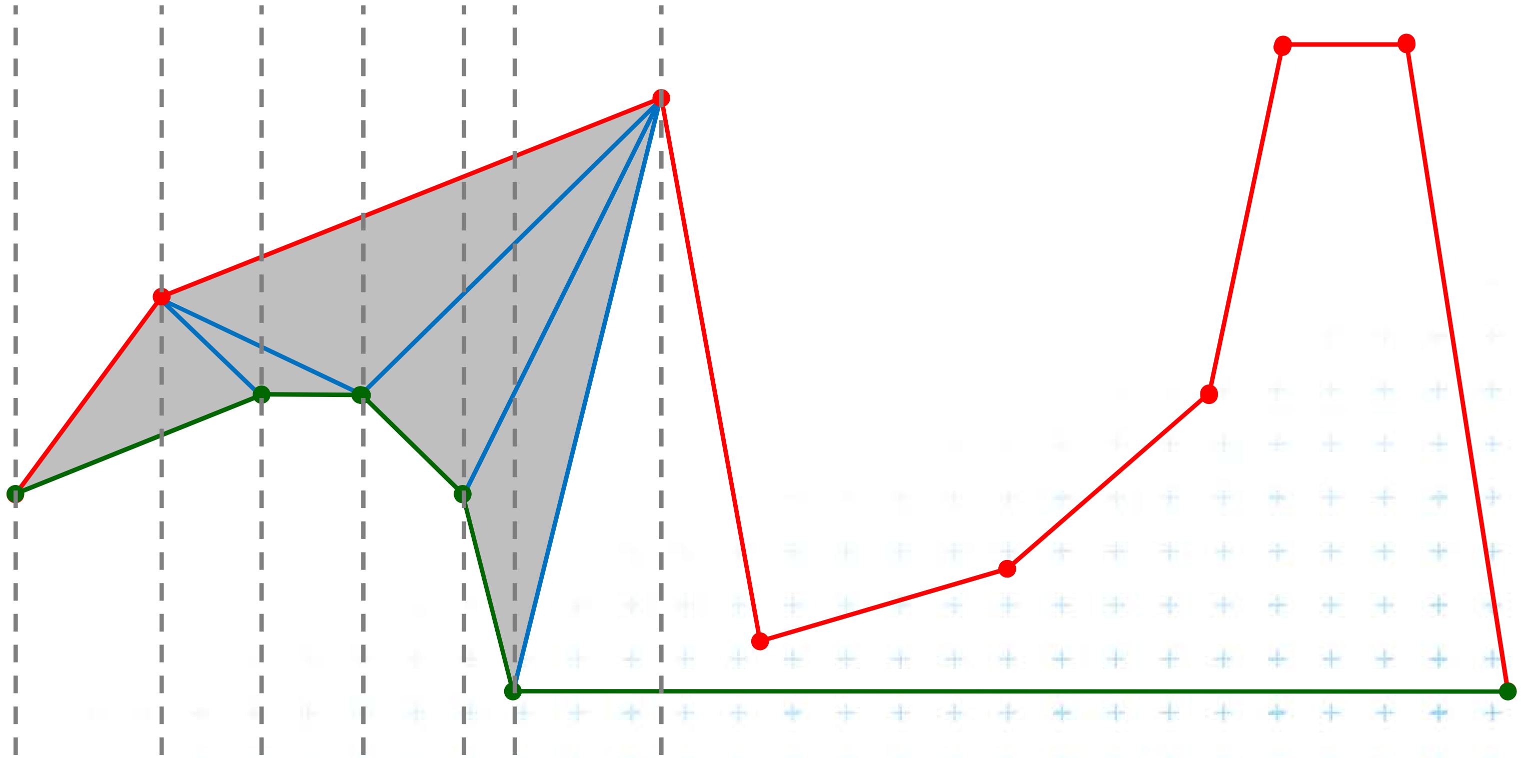
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

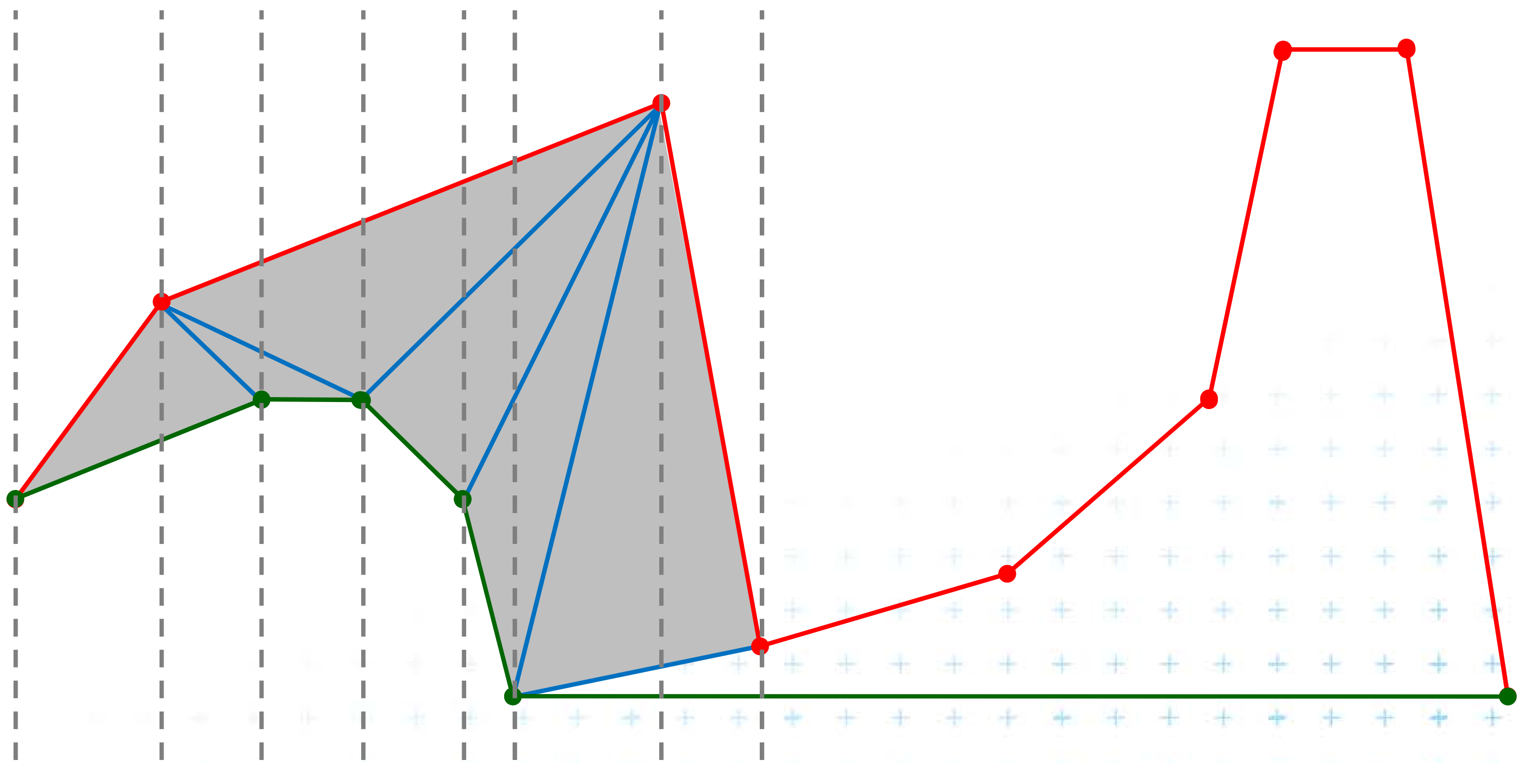
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

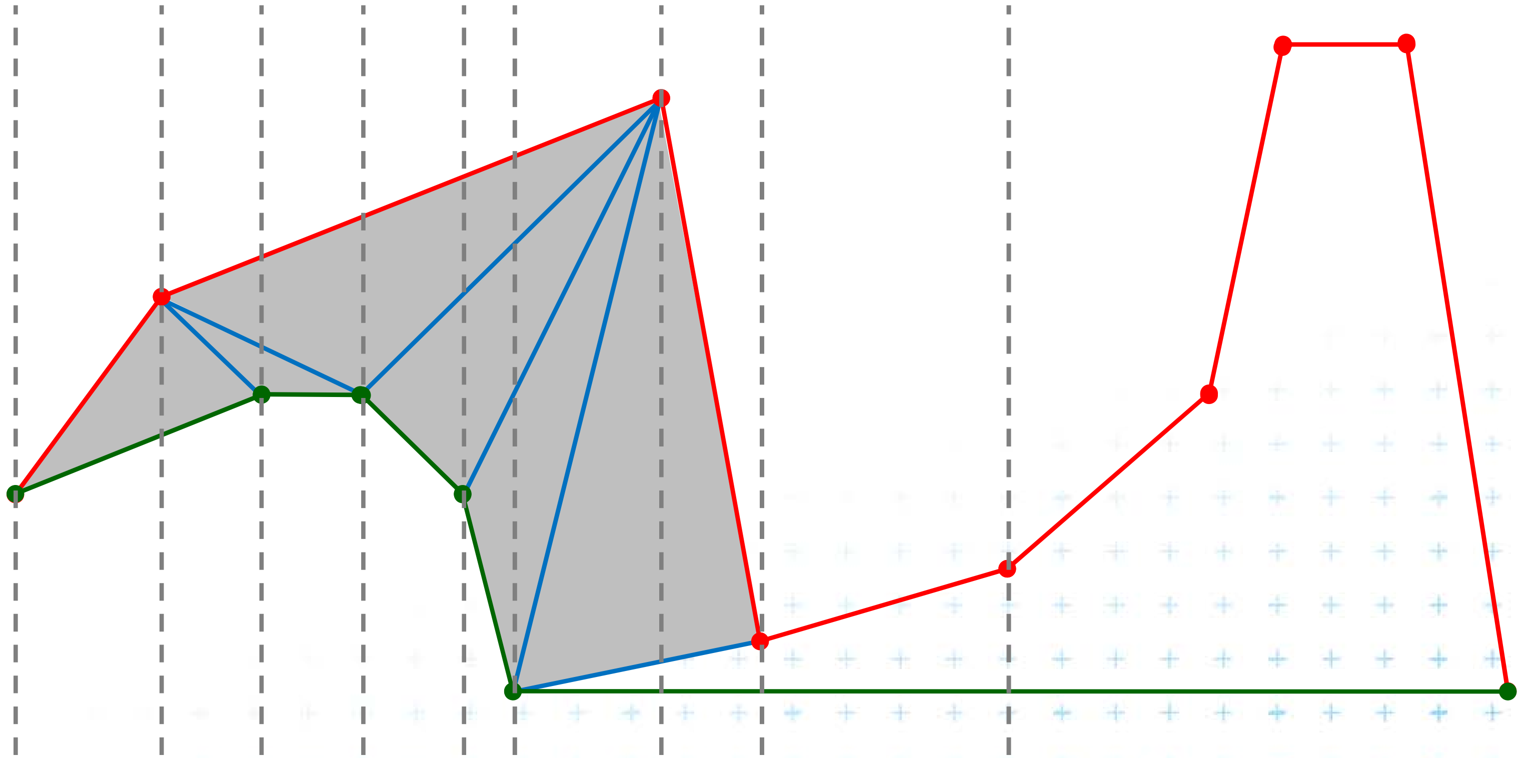
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

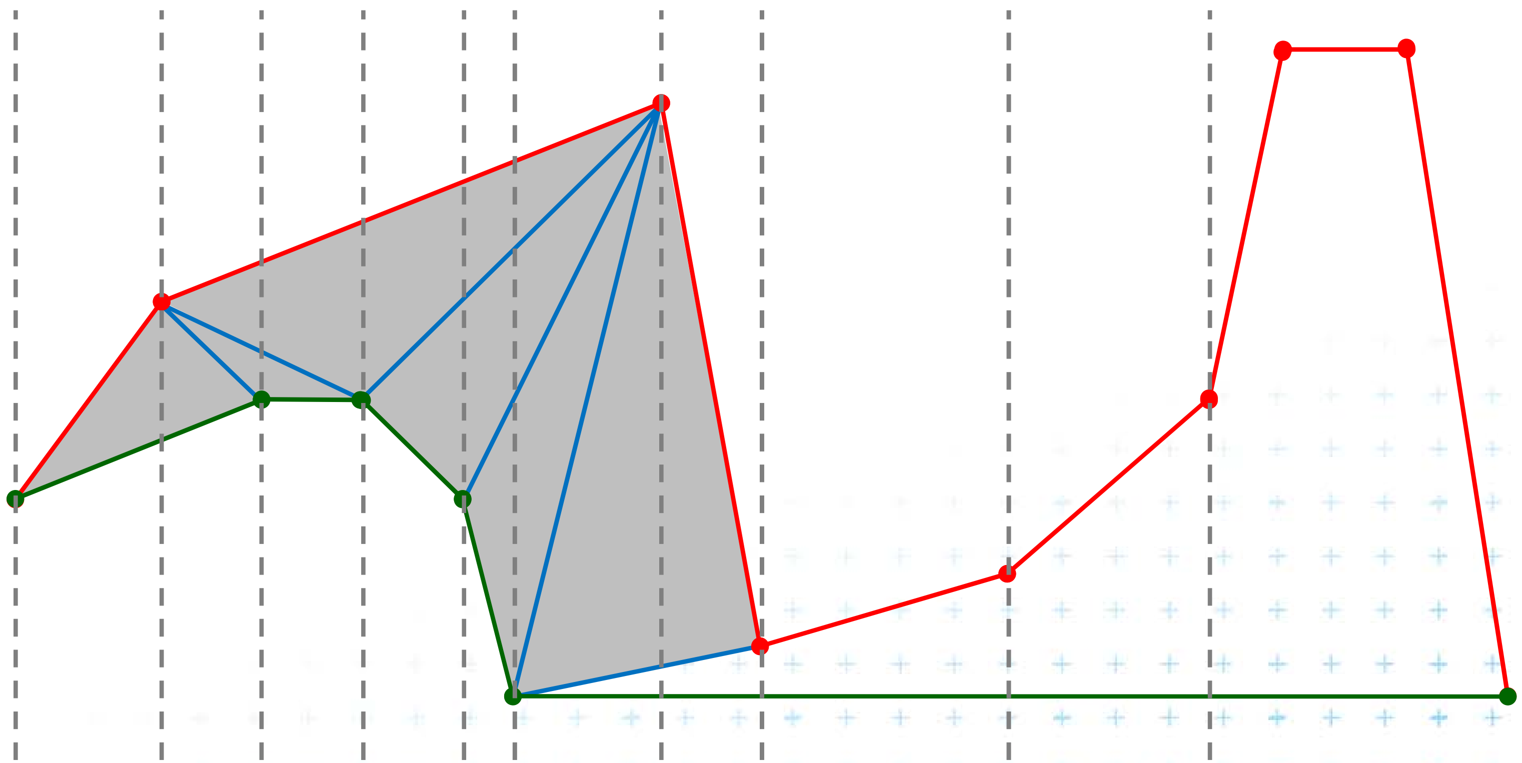
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

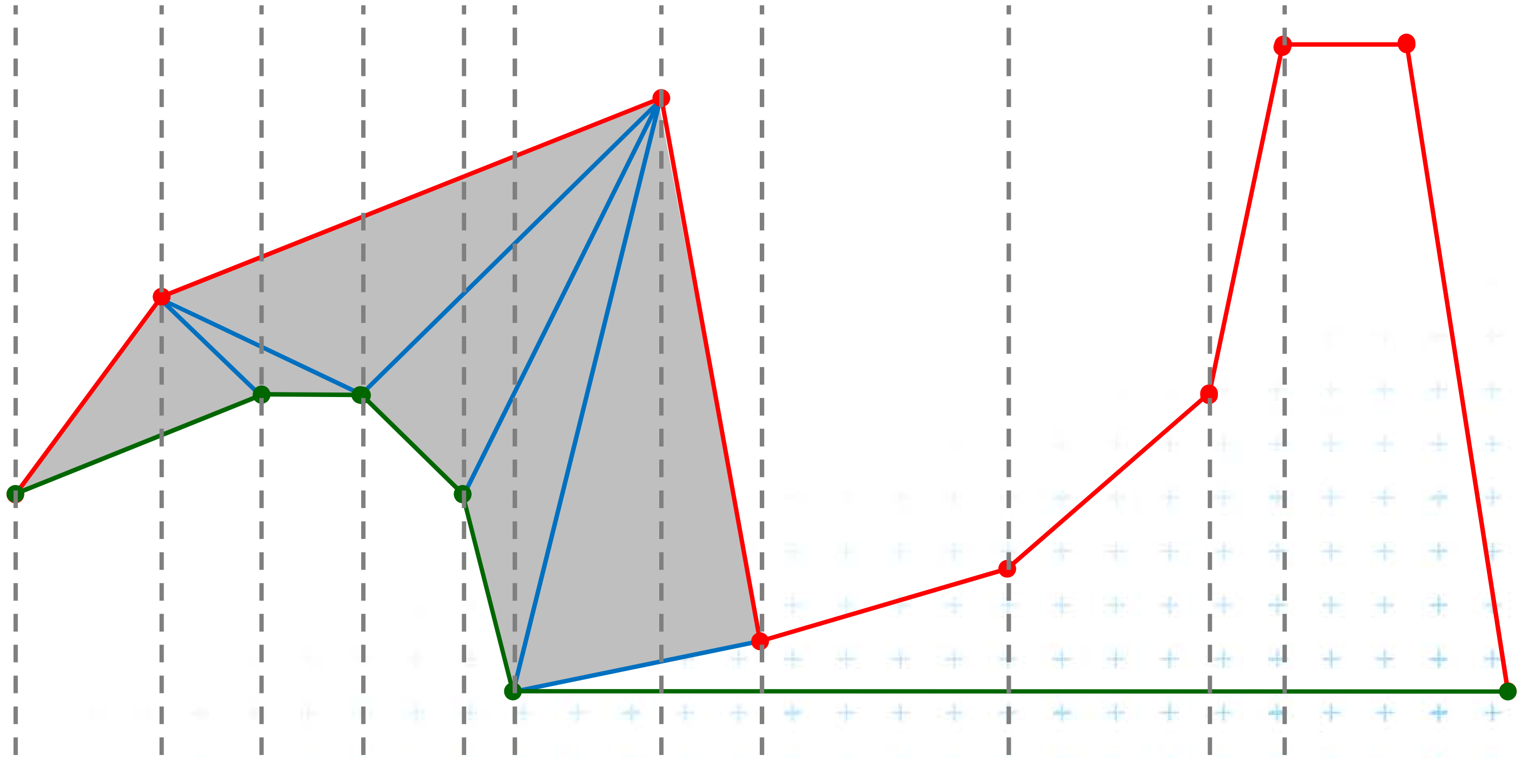
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

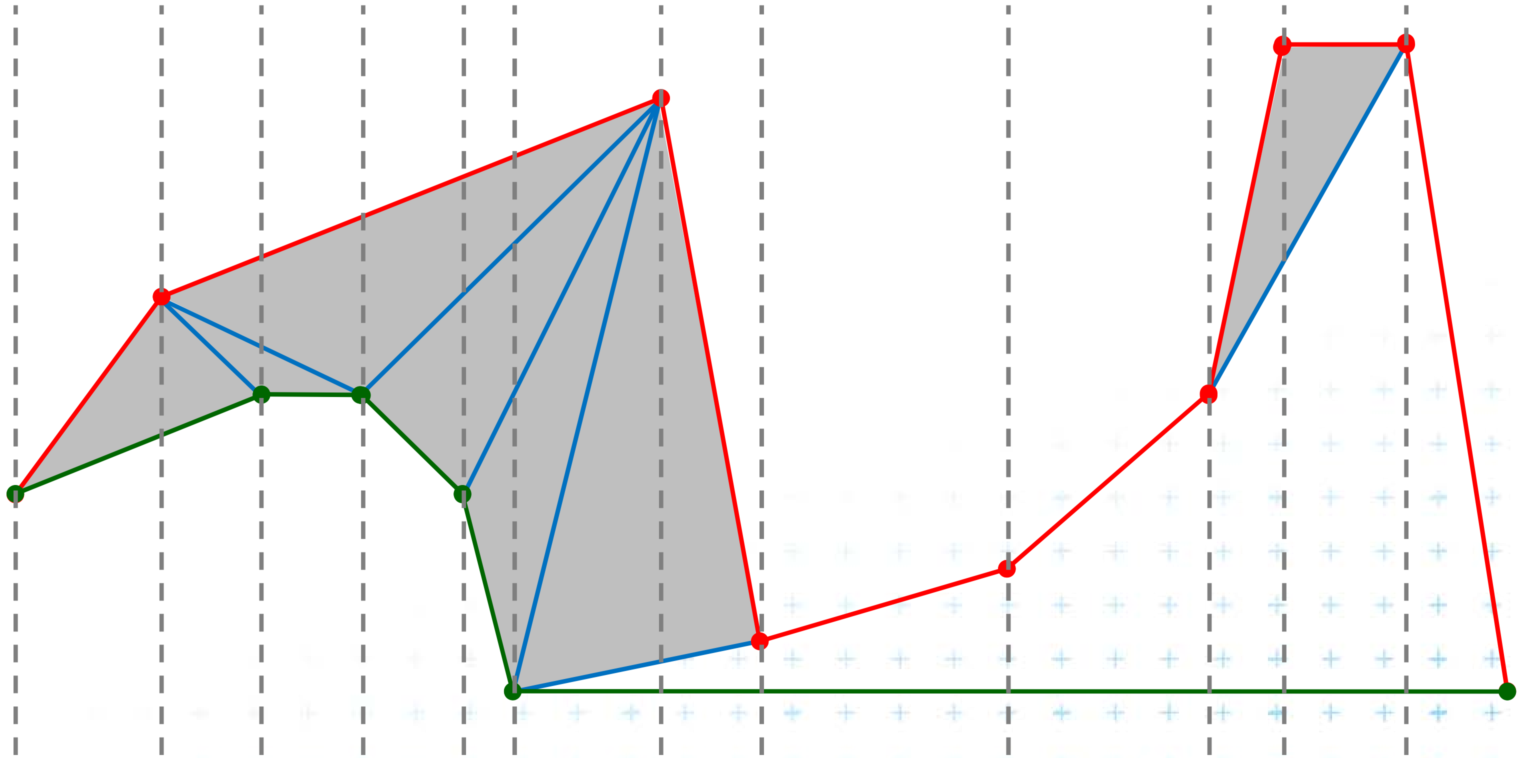
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

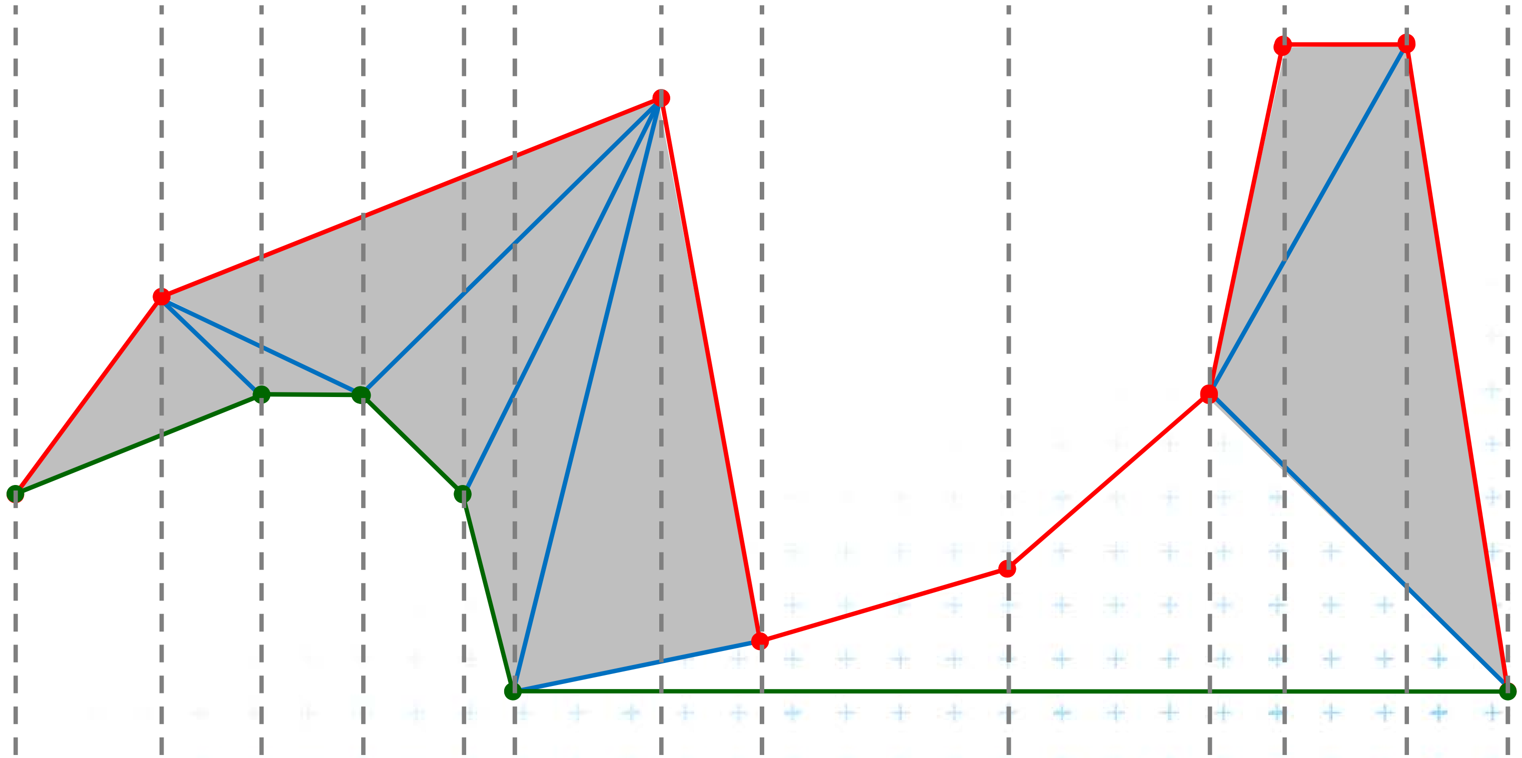
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

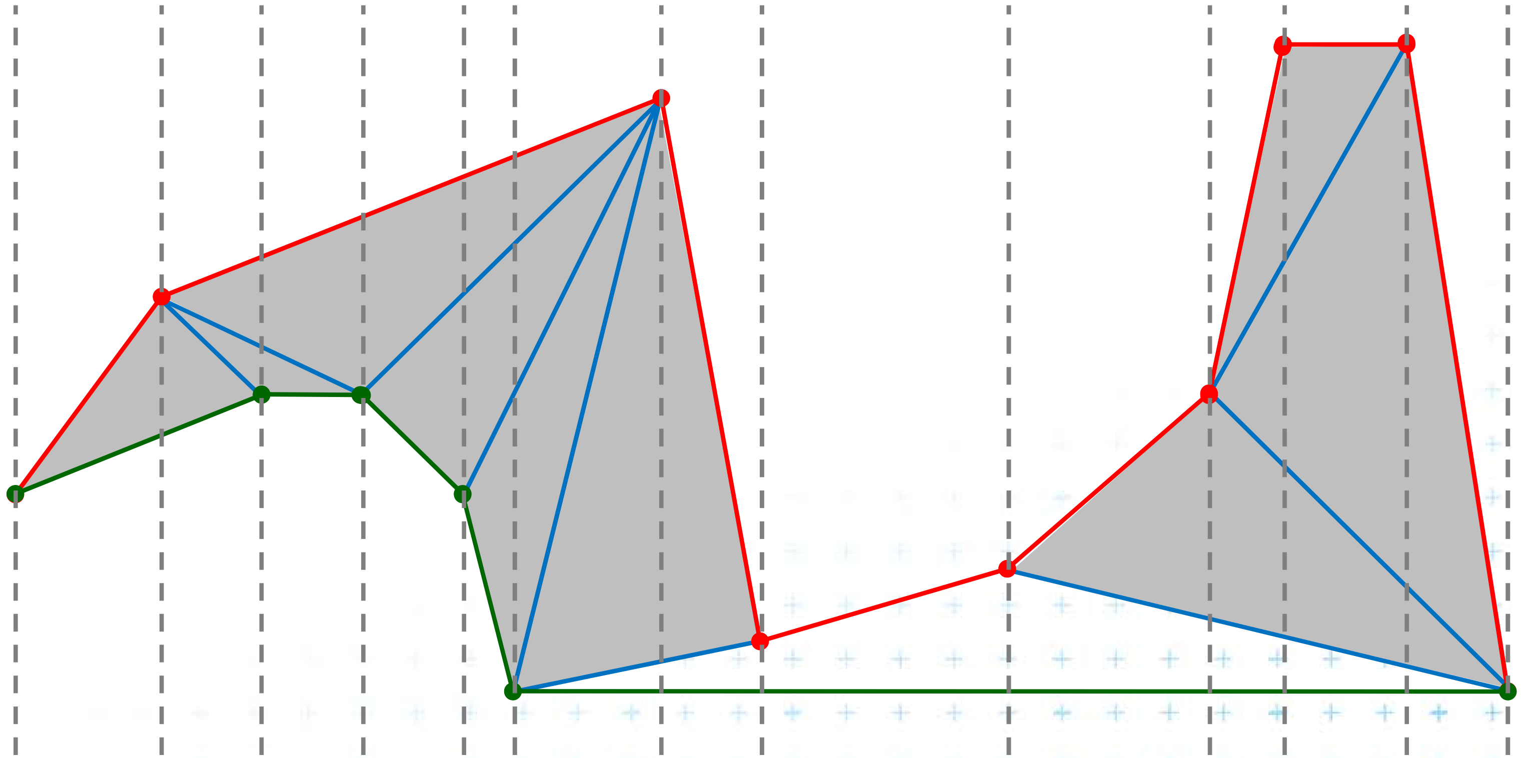
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

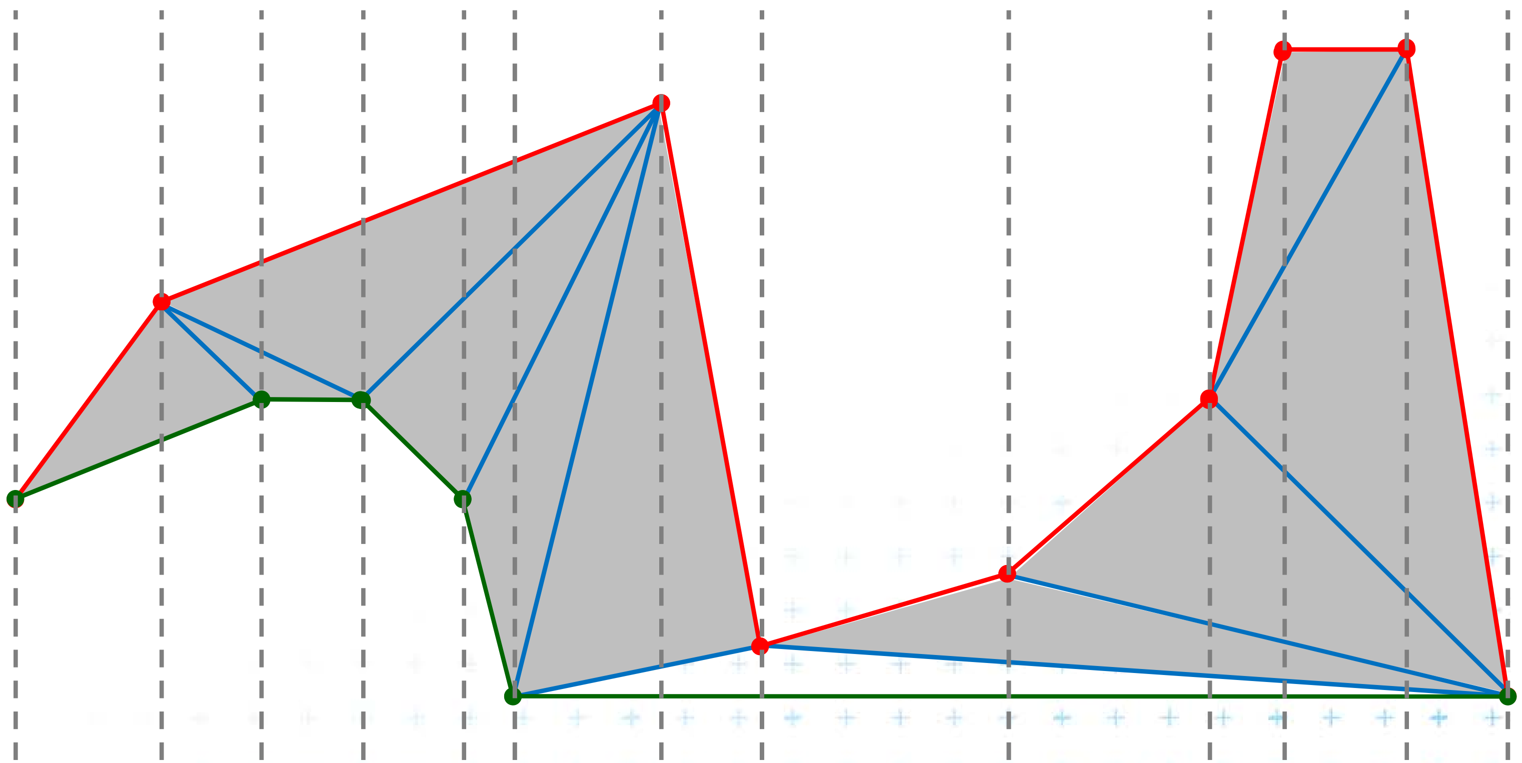
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

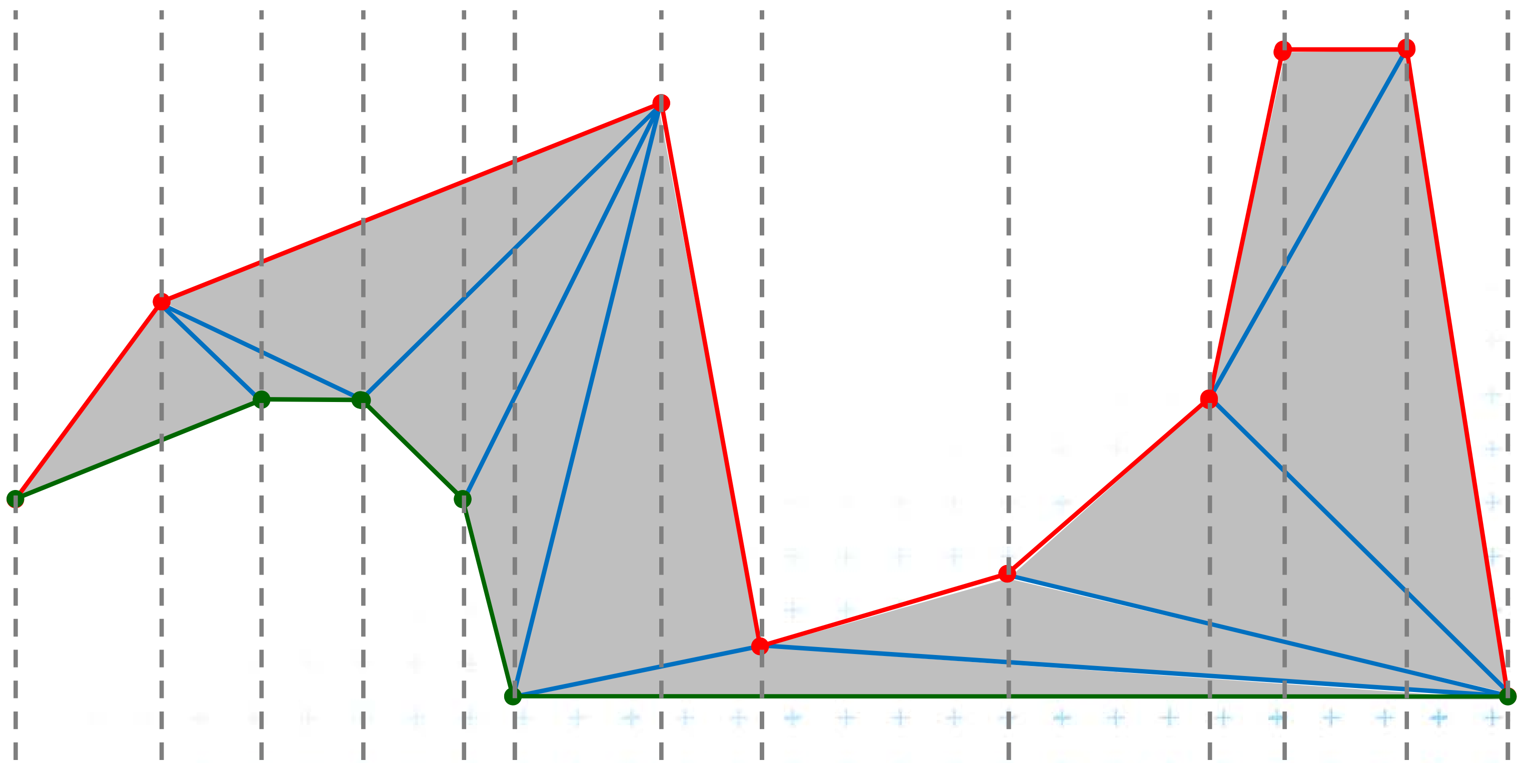
- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)



[Mount]

x- monotone polygon triangulation principle

- Sweep left to right - in $O(n)$ steps
- Triangulate everything you can by adding diagonals between visible points (left from the sweep line)

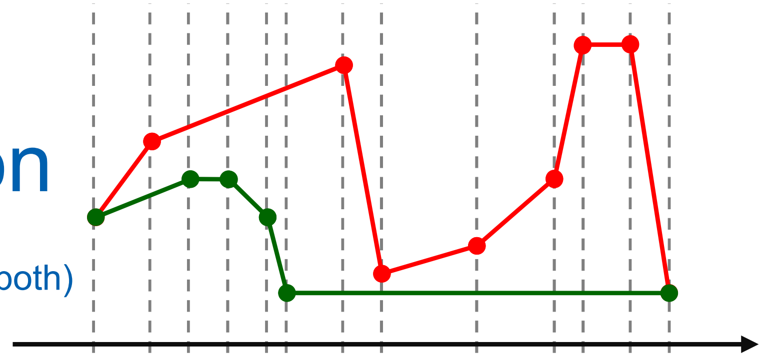


[Mount]

Event queue

Sweep line event queue

- x-sorted vertices of the polygon with lower/upper flag (2-bits, extremes to both)



Construction – $O(n)$

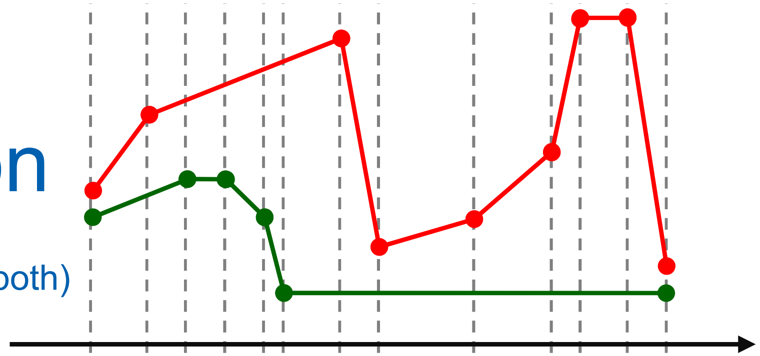
- Find min x and max x
- Extract lower and upper chain (between min and max x)

  Both are sorted in increasing order of their x-coords
- Merge chains in $O(n)$ keeping lower/upper flag

Event queue

Sweep line event queue

- x-sorted vertices of the polygon with lower/upper flag (2-bits, extremes to both)



Construction – $O(n)$

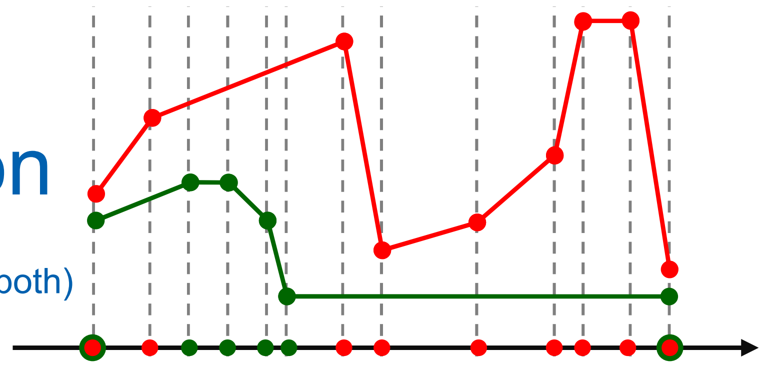
- Find min x and max x
- Extract lower and upper chain (between min and max x)

  Both are sorted in increasing order of their x-coords
- Merge chains in $O(n)$ keeping lower/upper flag

Event queue

Sweep line event queue

- x-sorted vertices of the polygon with lower/upper flag (2-bits, extremes to both)



Construction – $O(n)$

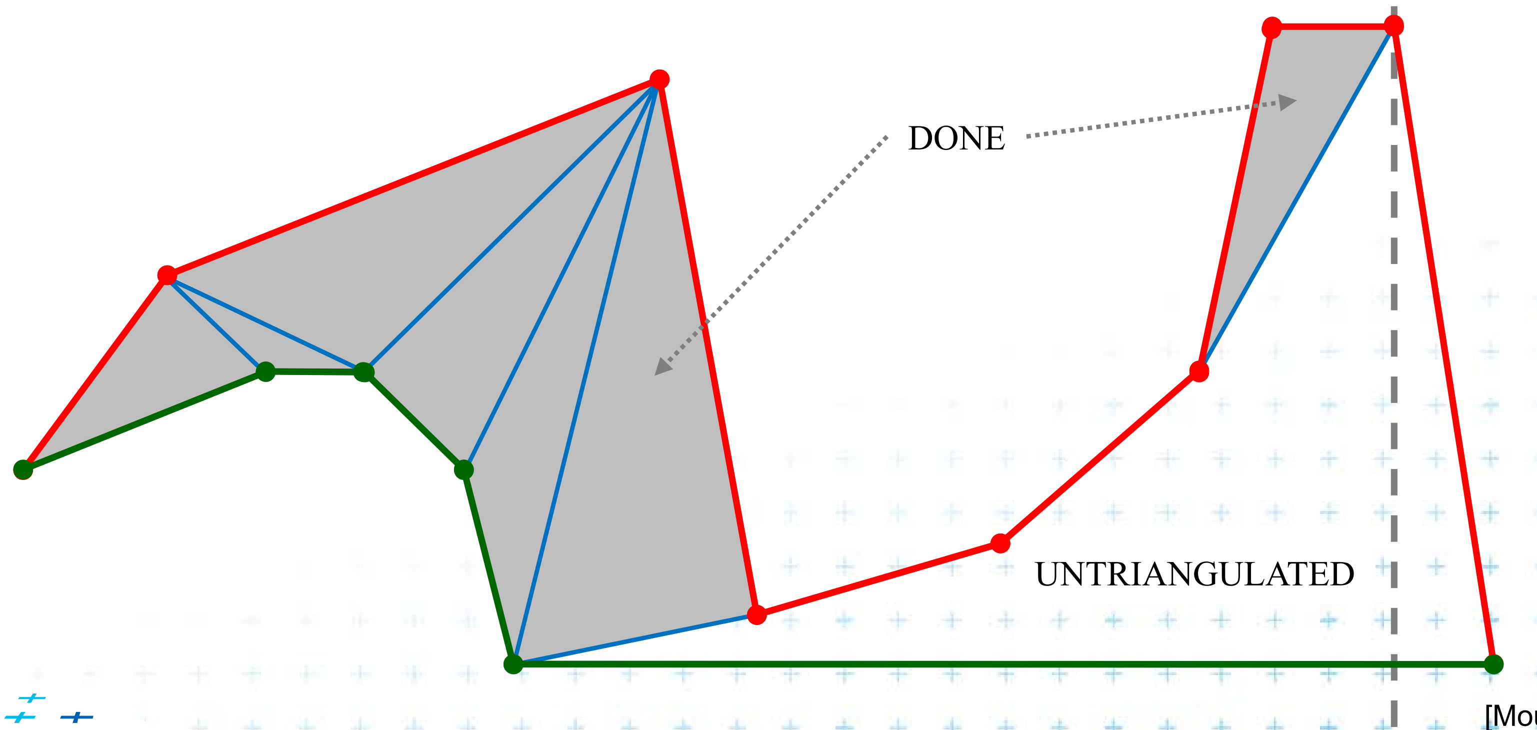
- Find min x and max x
- Extract lower and upper chain (between min and max x)

  Both are sorted in increasing order of their x-coords
- Merge chains in $O(n)$ keeping lower/upper flag

Regions on the left from the sweep line

a) triangulated – points were visible – DONE

b) untriangulated – points were not visible
– characterized by an invariant
(= a condition that is true after each step)



[Mount]

Reflex vertex and reflex chain

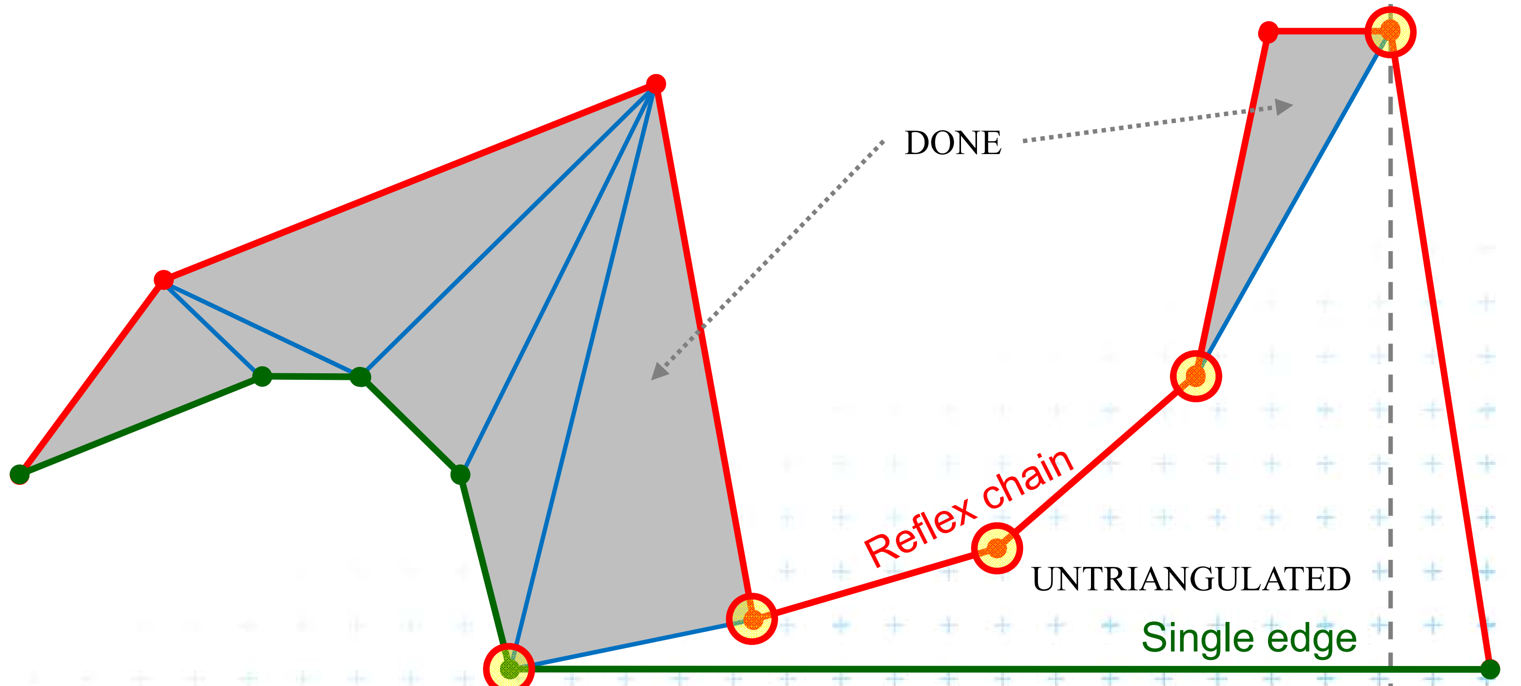
Untriangulated region is bounded by a reflex chain

= a sequence of reflex vertices along the not-triangulated part of the polygon

– in the alg. is stored in stack

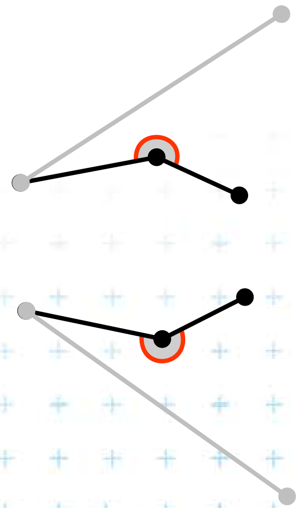
Reflex vertex

interior angle $\geq \pi$



u = reflex chain start point

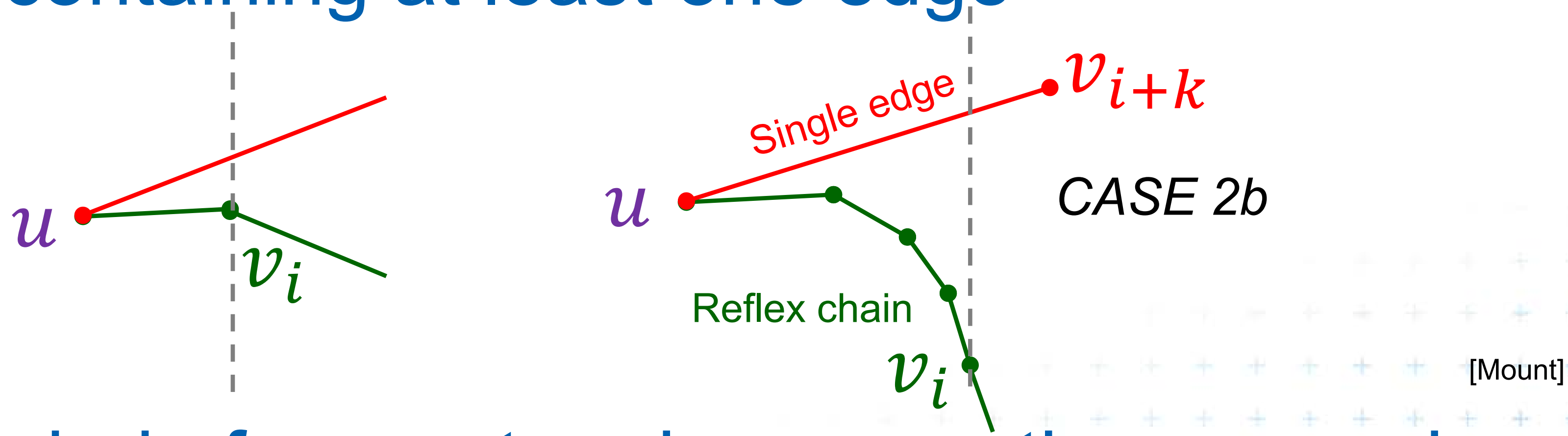
[Mount]



Main invariant of untriangulated region left from SL

i starts from 1, first vertex is v_1

- Let v_i, $i \geq 2$ be the vertex just being processed
- The untriangulated region left of v_i consists of two x-monotone chains (upper and lower) each containing at least one edge

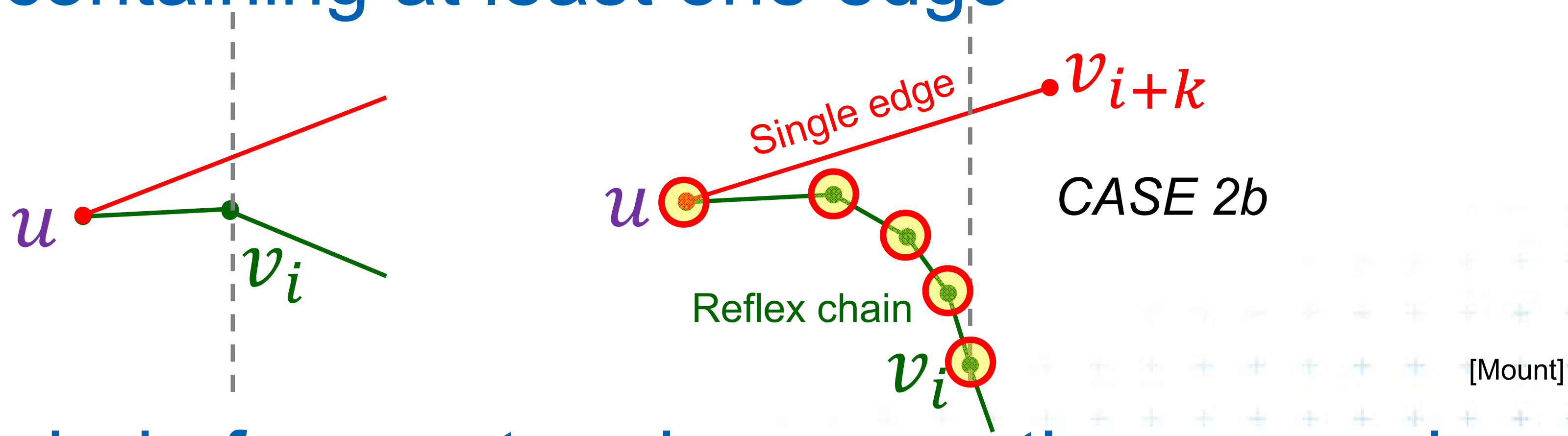


- If the chain from v_i to u has more than one edge
  - these edges form a reflex chain
  - the other chain consist of single edge from u to vertex v_{i+k} right of v_i

Main invariant of untriangulated region left from SL

i starts from 1, first vertex is v_1

- Let v_i, $i \geq 2$ be the vertex just being processed
- The untriangulated region left of v_i consists of two x-monotone chains (upper and lower) each containing at least one edge

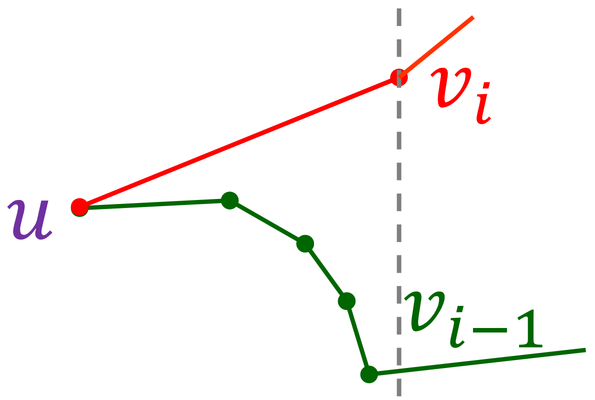


- If the chain from v_i to u has more than one edge
  – these edges form a reflex chain
  – the other chain consist of single edge from u to vertex v_{i+k} right of v_i

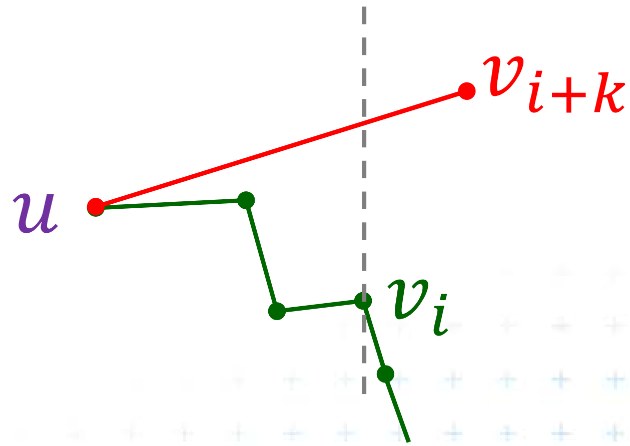
The remaining regions are triangulated

- Elsewhere, it would have been triangulated in this step

CASE 1

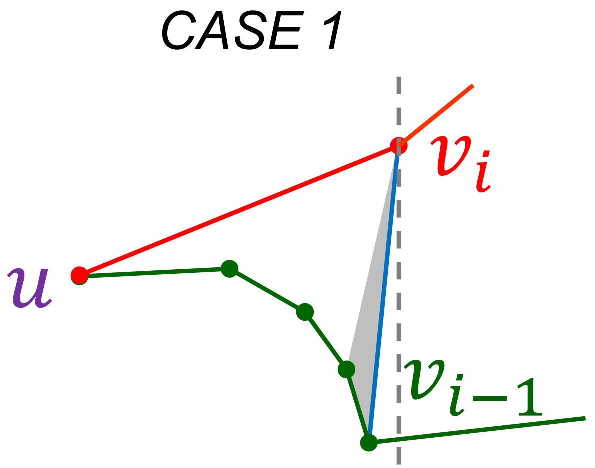


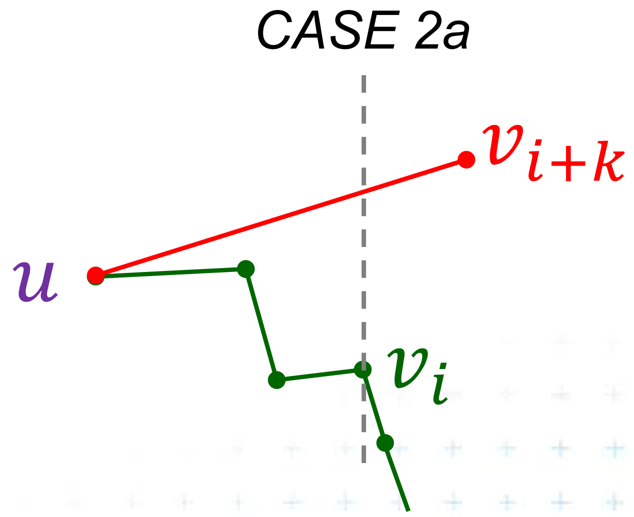
CASE 2a



The remaining regions are triangulated

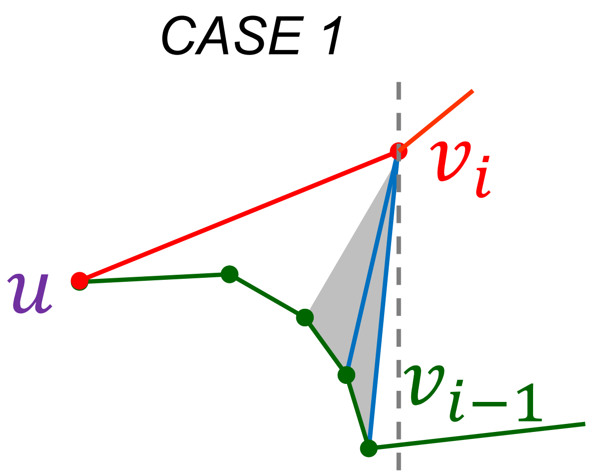
- Elsewhere, it would have been triangulated in this step

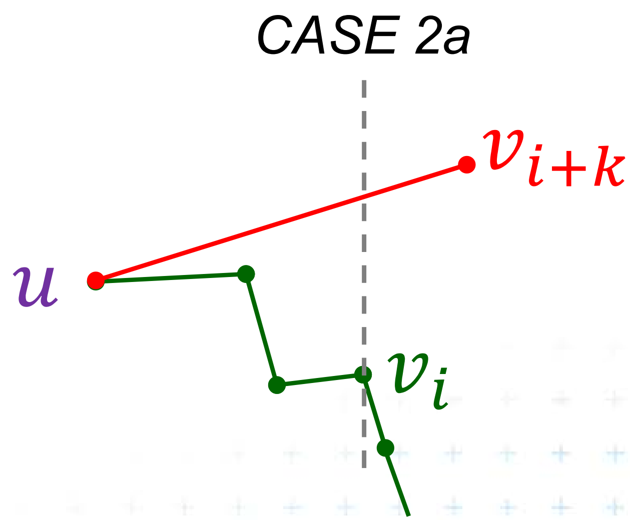




The remaining regions are triangulated

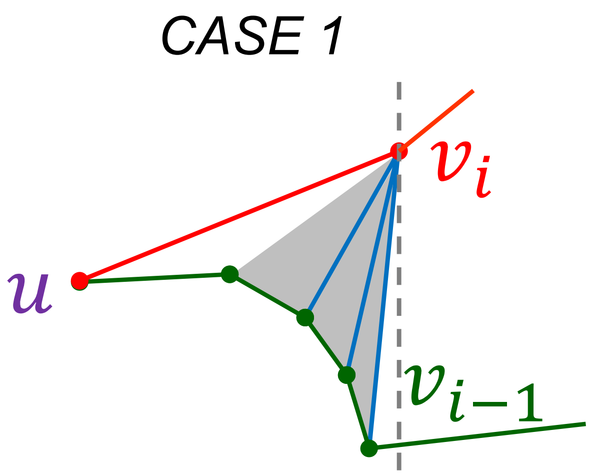
- Elsewhere, it would have been triangulated in this step

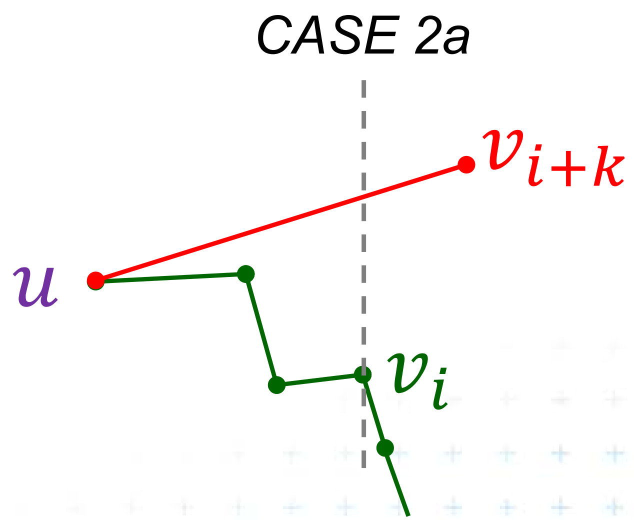




The remaining regions are triangulated

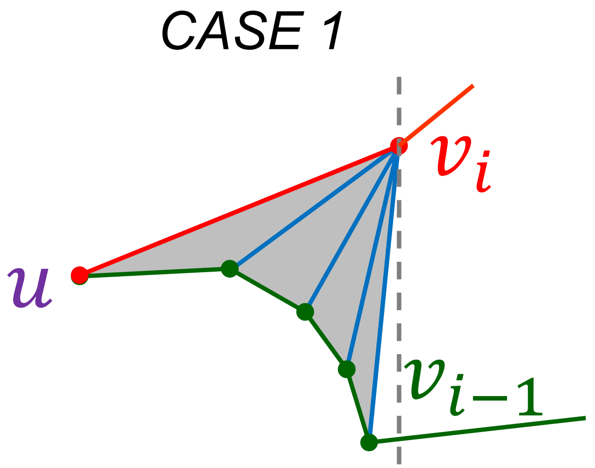
- Elsewhere, it would have been triangulated in this step

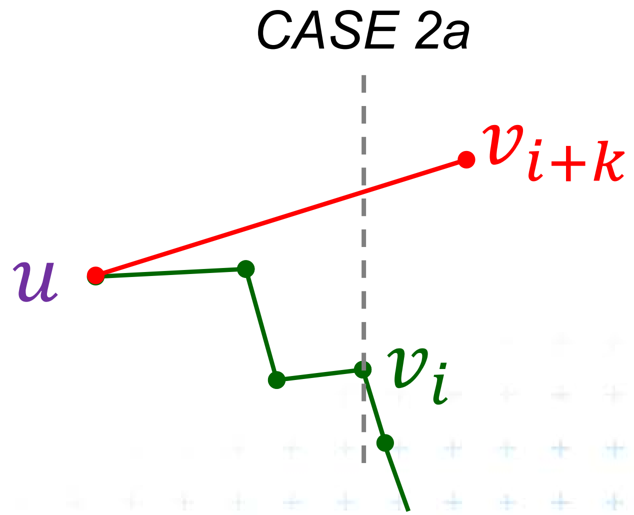




The remaining regions are triangulated

- Elsewhere, it would have been triangulated in this step

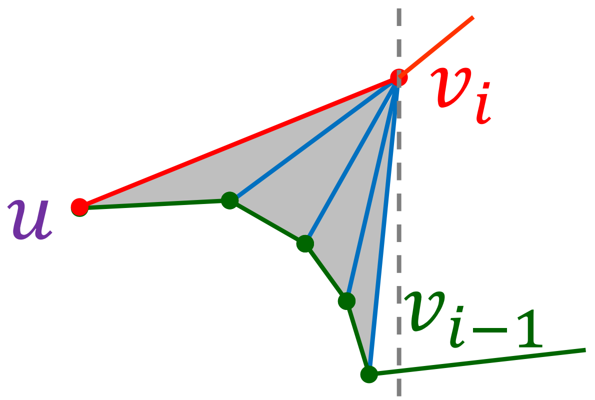




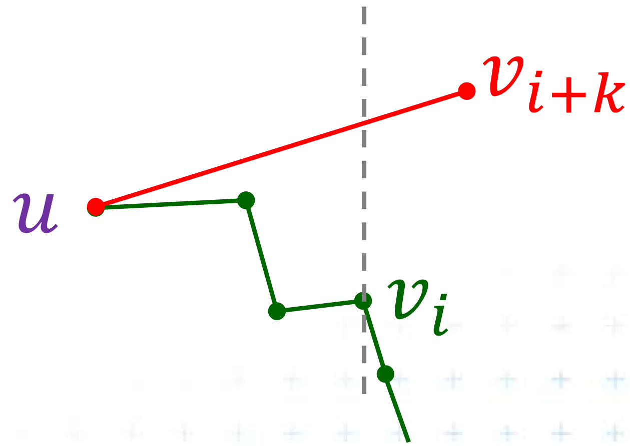
The remaining regions are **triangulated**

- Elsewhere, it would have been triangulated in this step

CASE 1

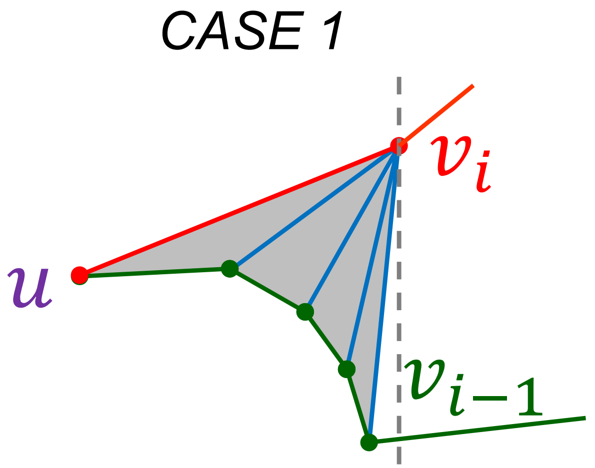


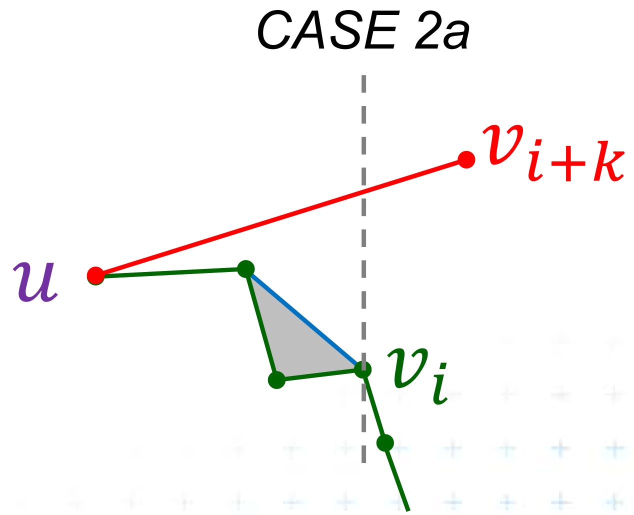
CASE 2a



The remaining regions are triangulated

- Elsewhere, it would have been triangulated in this step

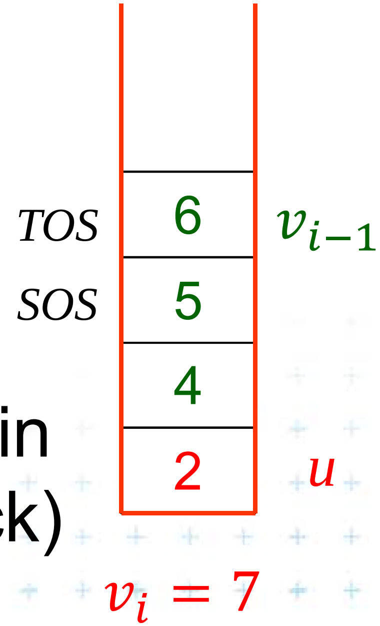




Triangulation algorithm

Data structures

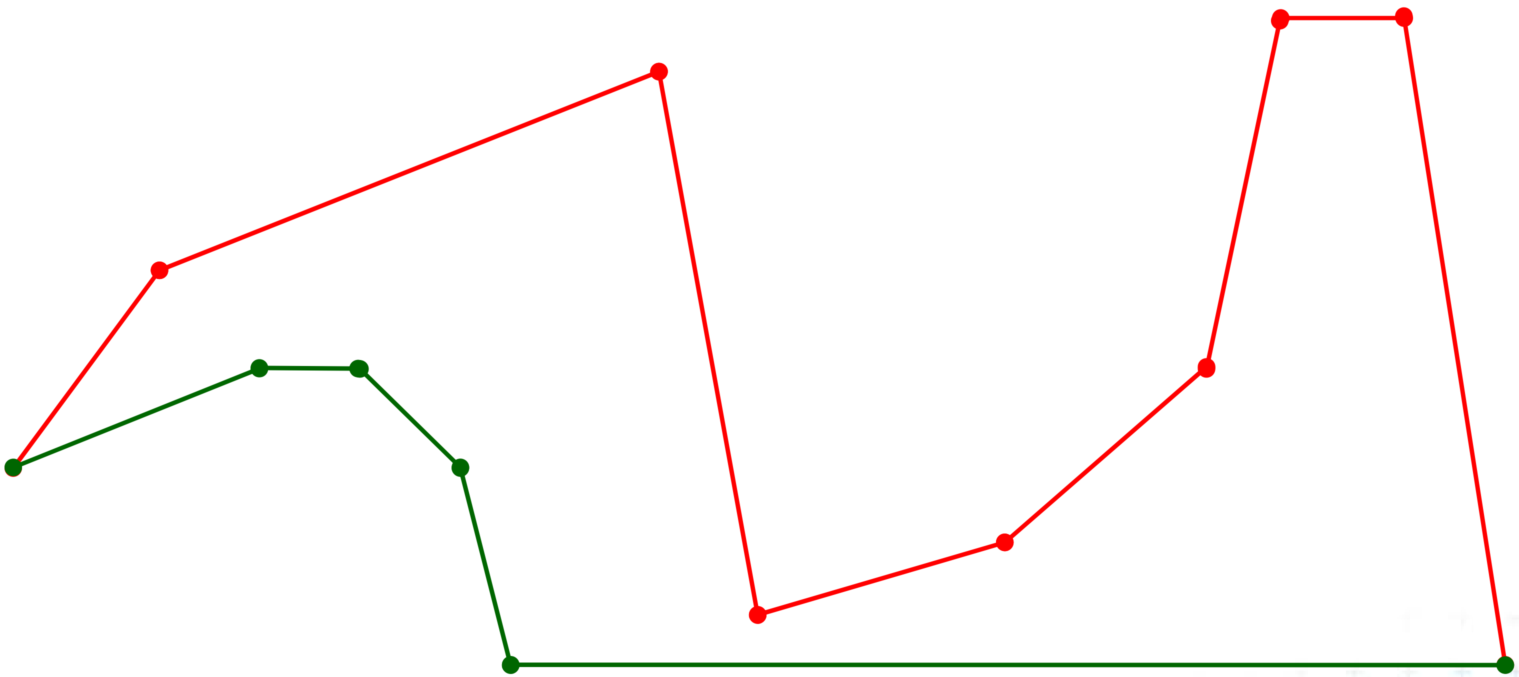
- Event queue with merged upper and lower chain
- Status
  - Current vertex v_i (sweep line position i)
  - Reflex vertices chain in the stack
  - Upper/lower chain flag
    all vertices except u are from the same chain
    u is from the opposite chain (bottom of stack)



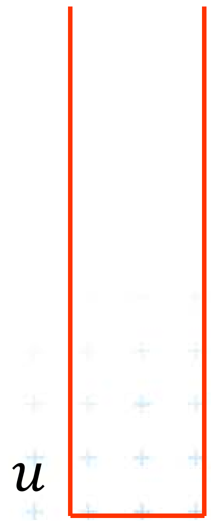
Orientation test

– reflex(TOS, SOS, v_i)

Monotone polygon triangulation algorithm

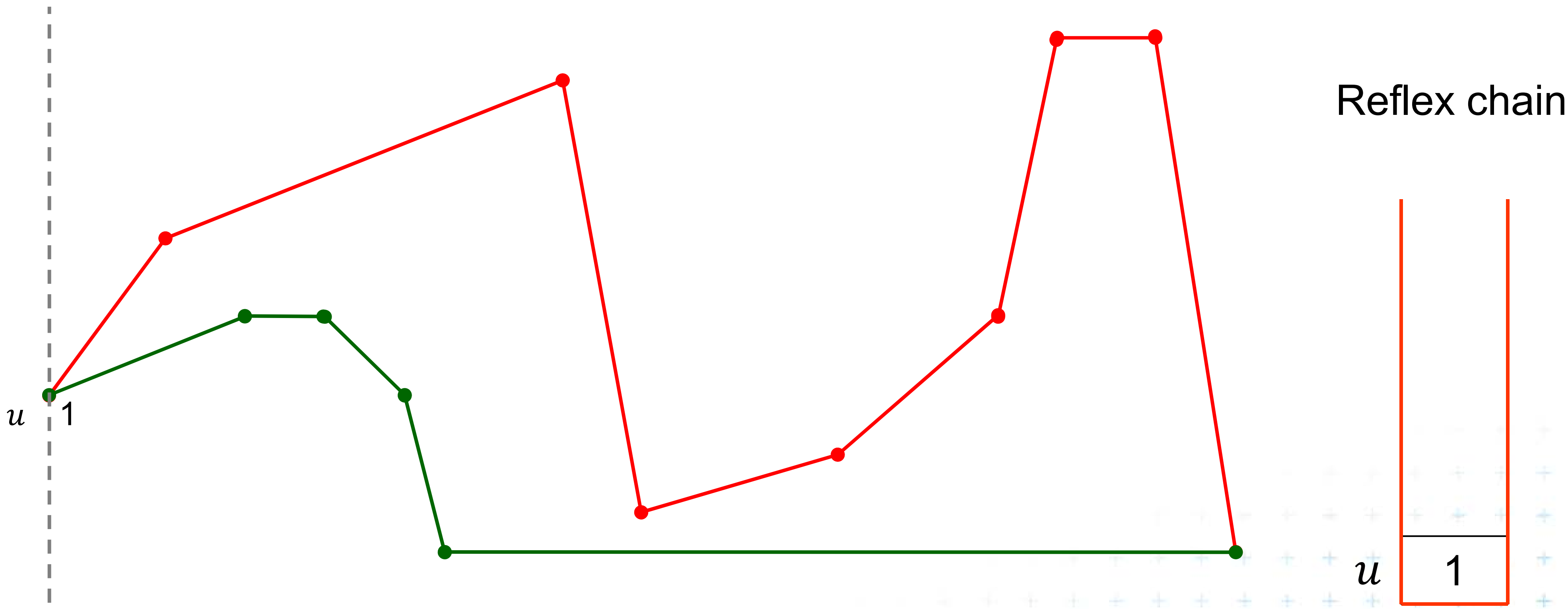


Reflex chain



[Mount]

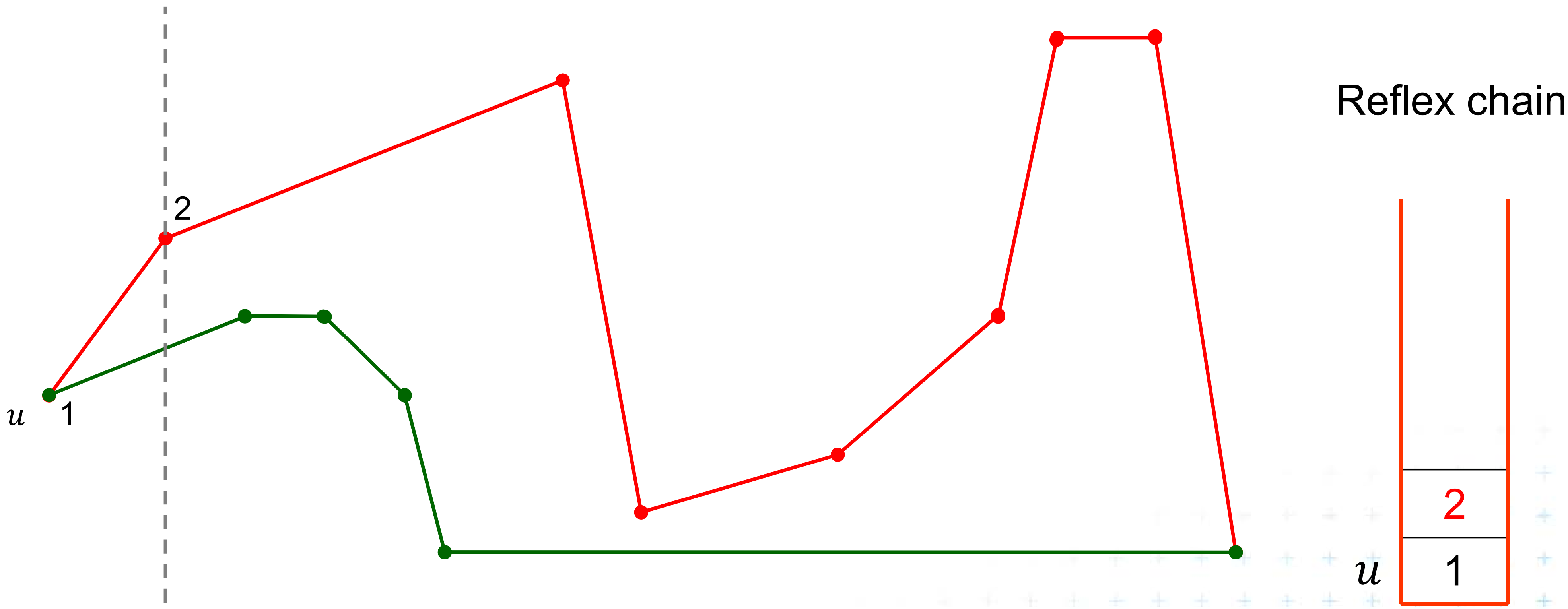
Monotone polygon triangulation algorithm



Start – set reflex chain start u (bottom of stack)

[Mount]

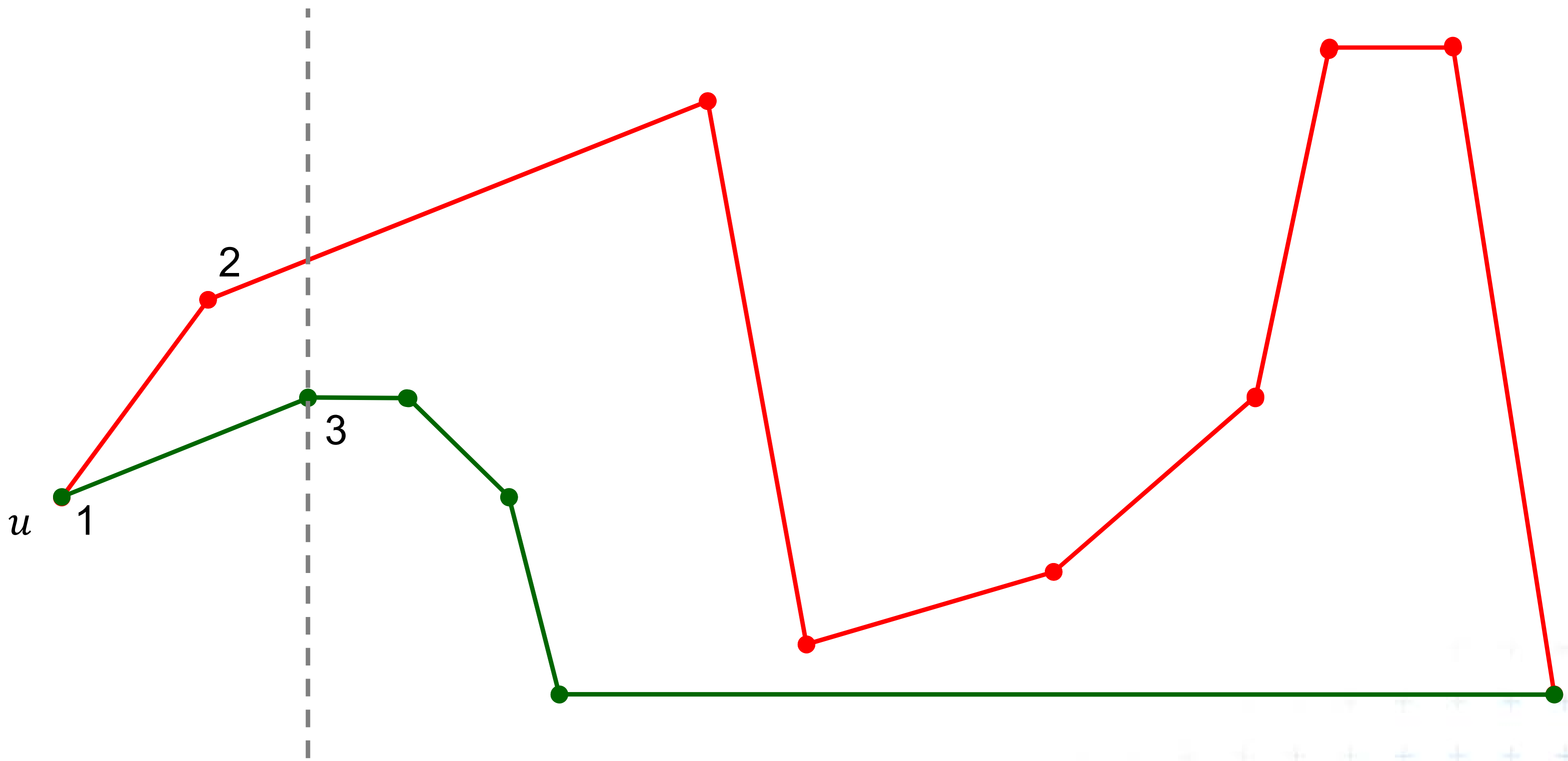
Monotone polygon triangulation algorithm

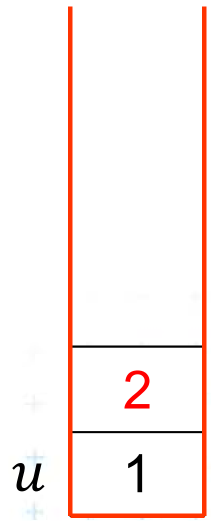


Start – set trivial reflex chain end (top of stack)

[Mount]

Monotone polygon triangulation algorithm

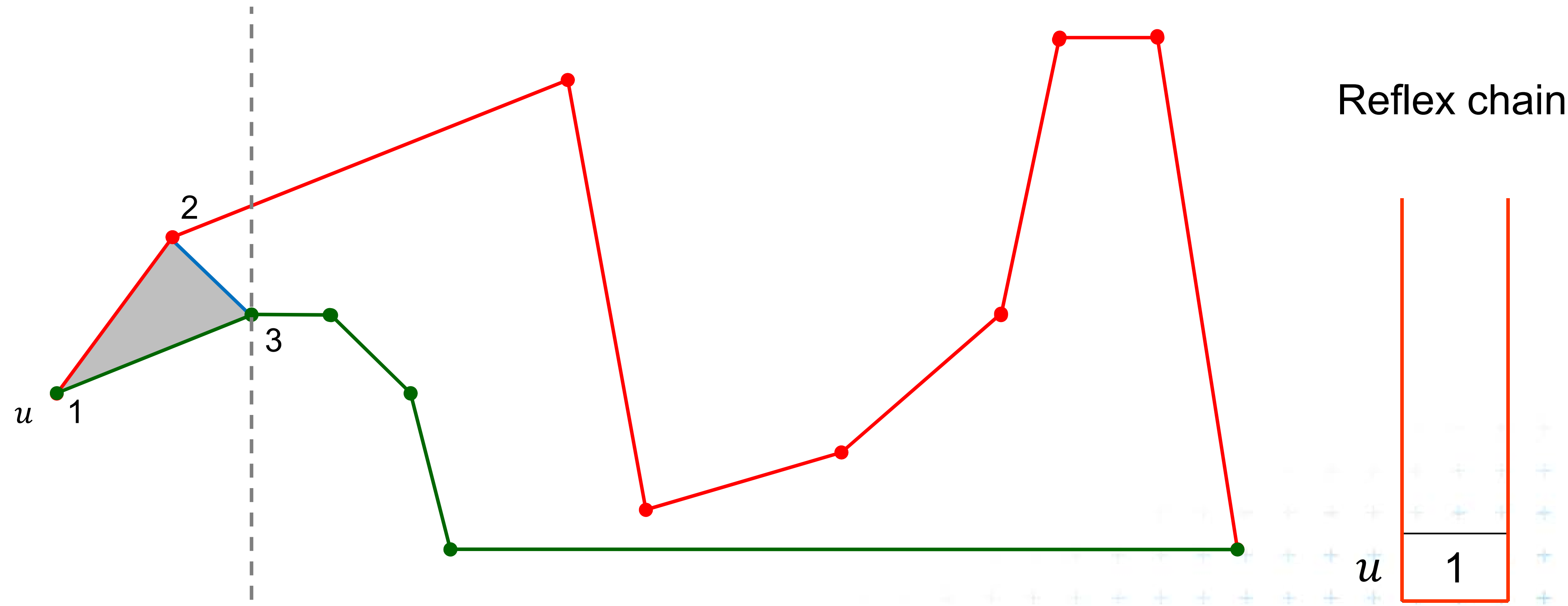




Case 1 – point v_i on opposite chain from v_{i-1}

[Mount]

Monotone polygon triangulation algorithm

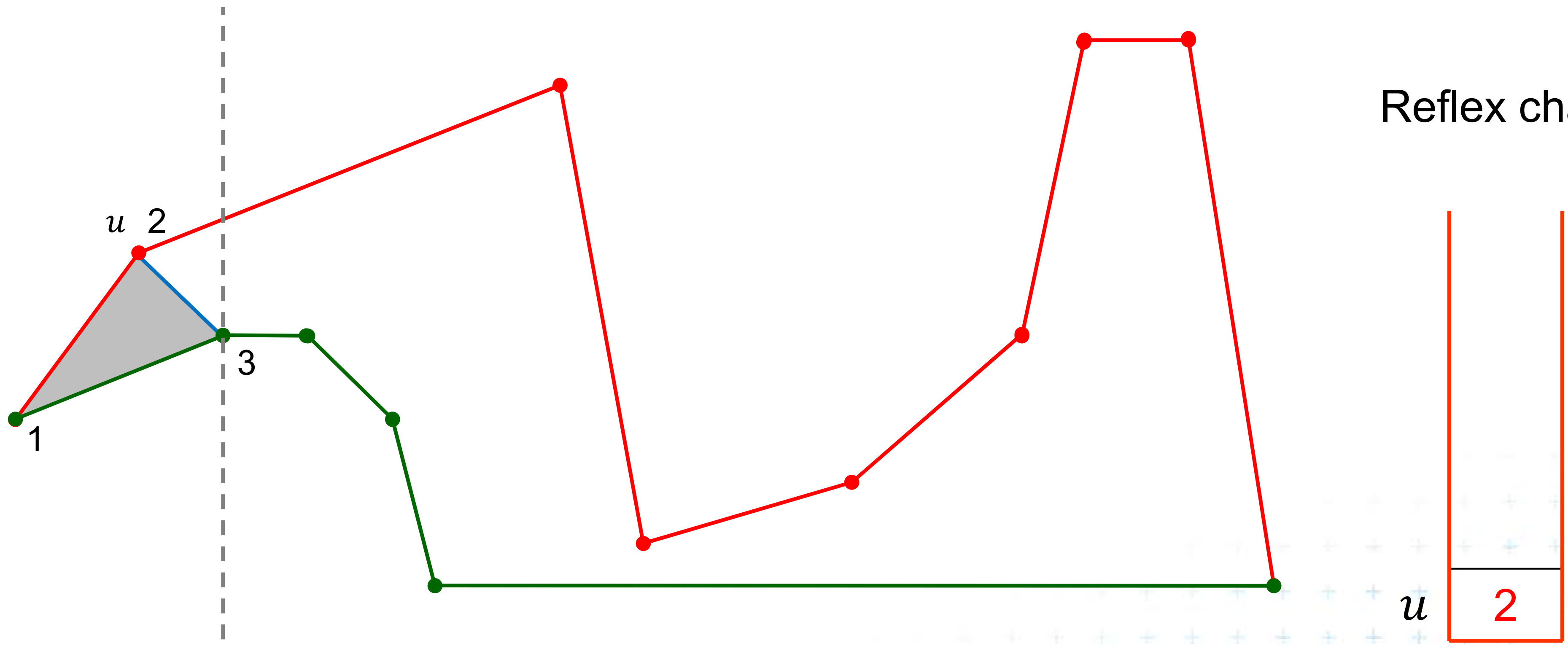


Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm



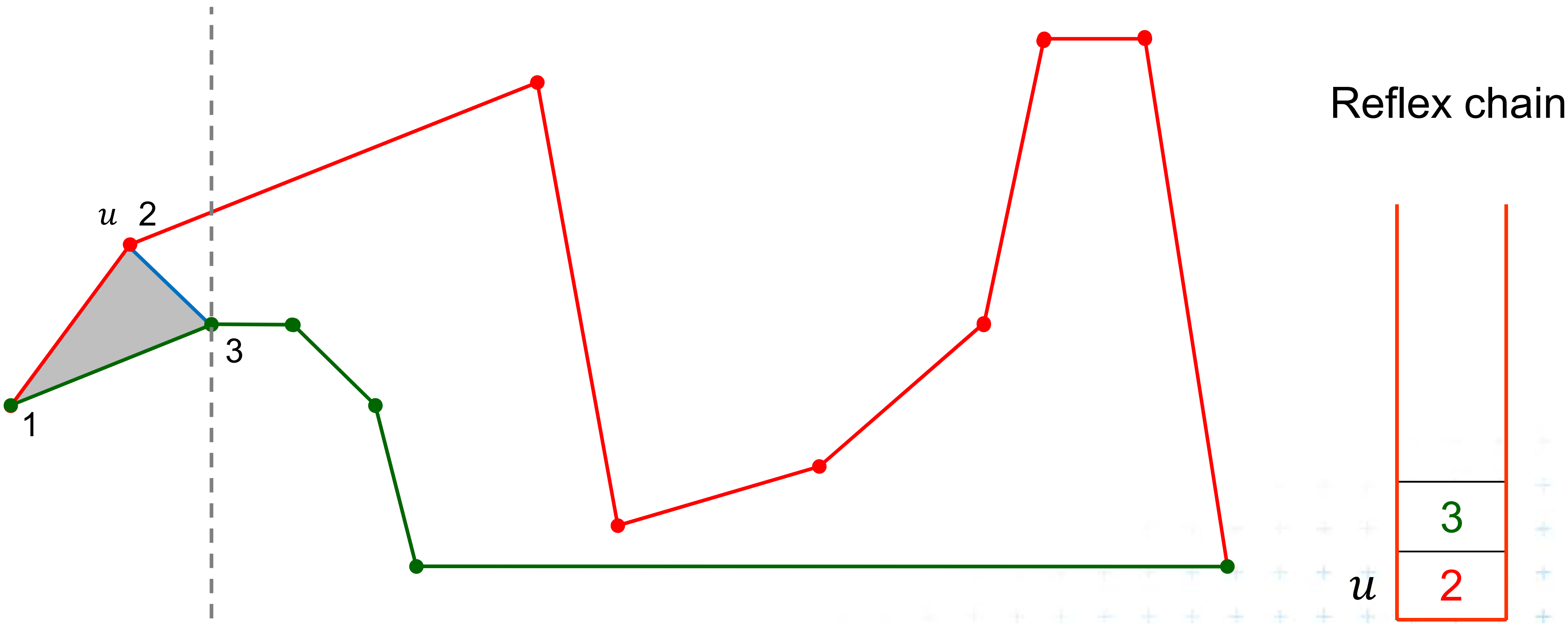
Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

Set trivial reflex chain $v_i v_{i-1}$: New u: pop(), push(v_{i-1}), push(v_i)

[Mount]

Monotone polygon triangulation algorithm



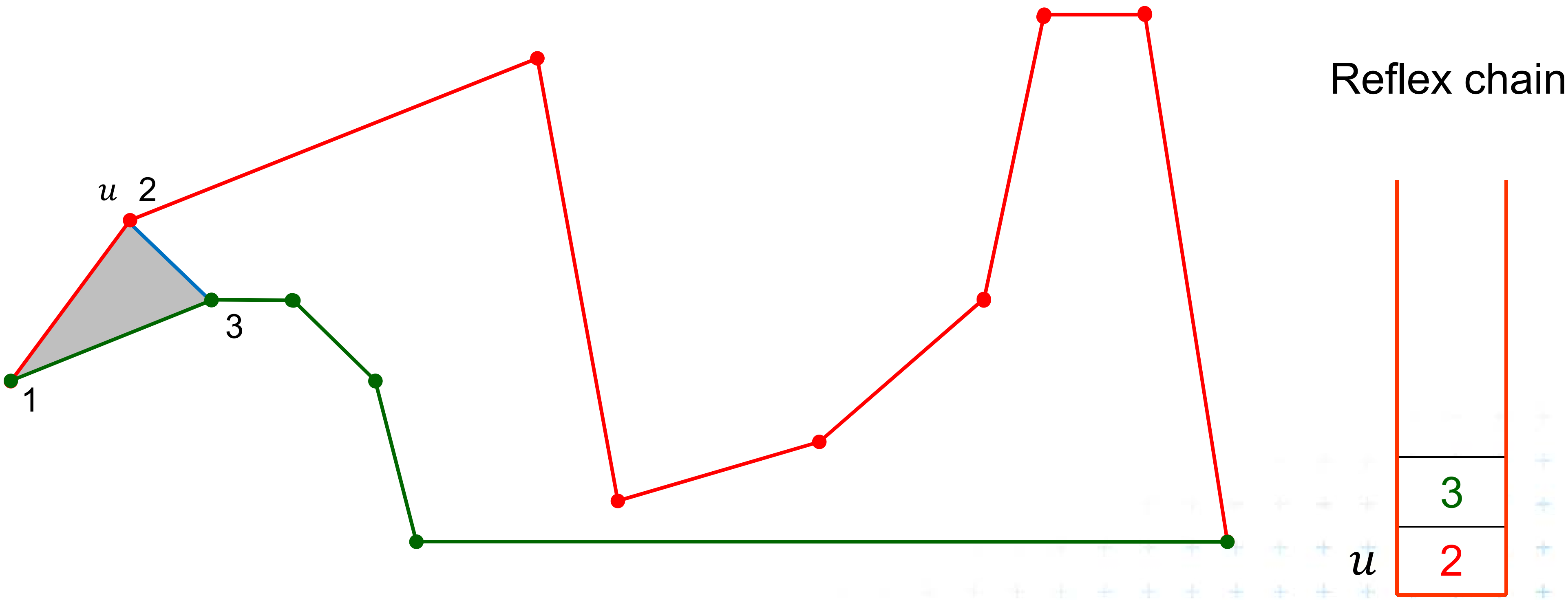
Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

Set trivial reflex chain $v_i v_{i-1}$: New u: pop(), push(v_{i-1}), push(v_i)

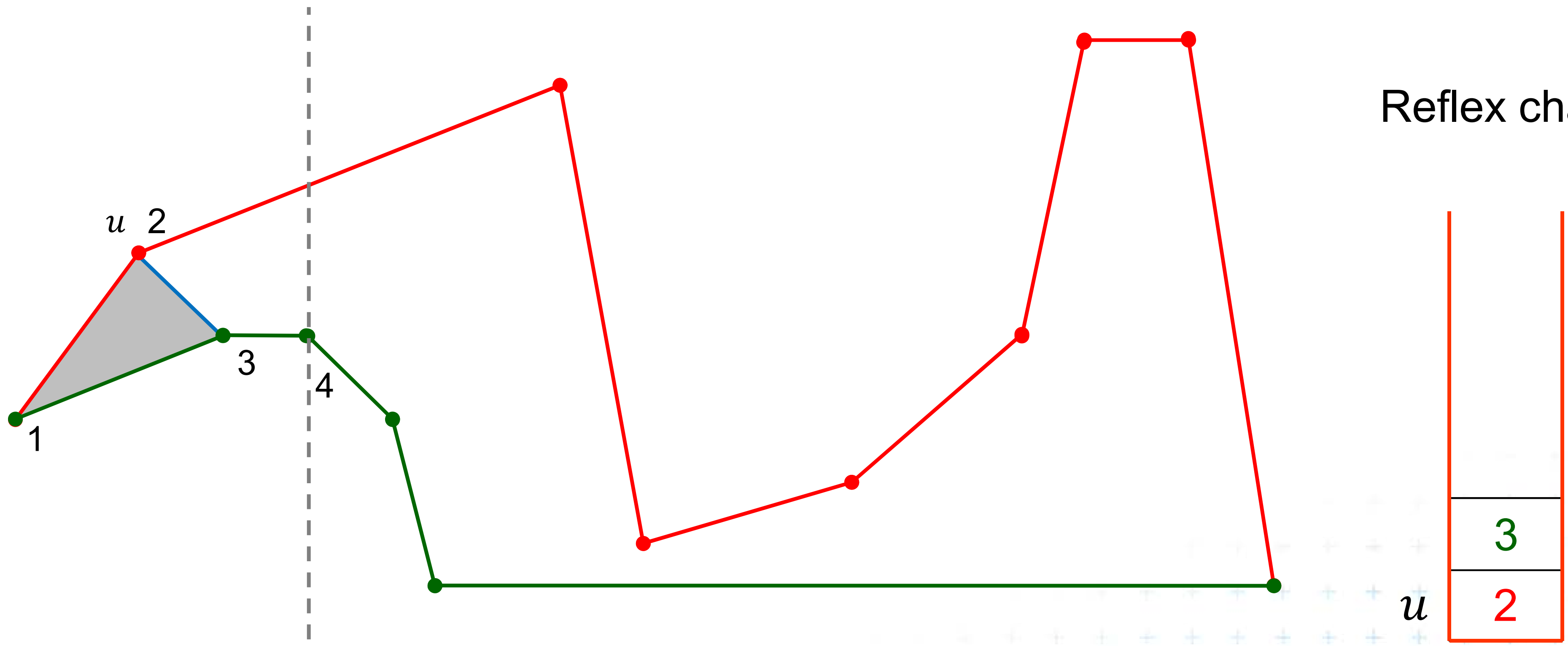
[Mount]

Monotone polygon triangulation algorithm



[Mount]

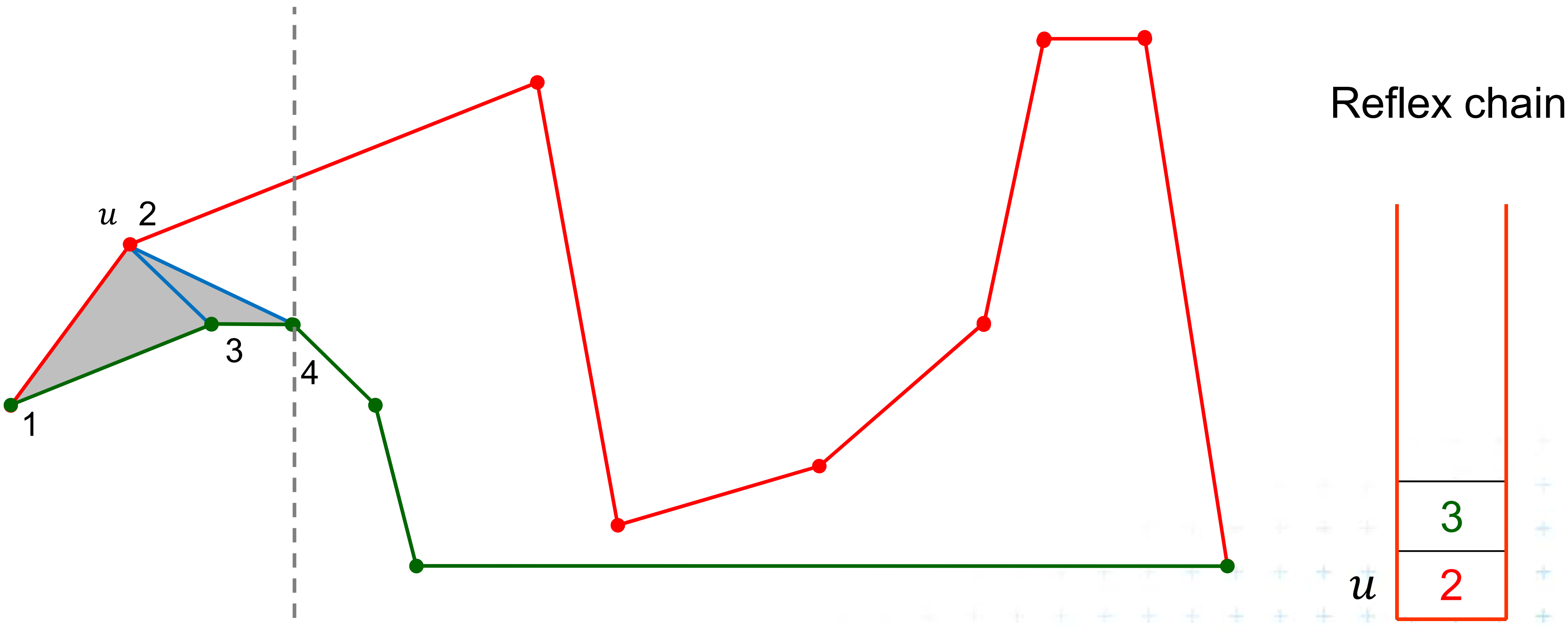
Monotone polygon triangulation algorithm



Case 2a – point v_i on the same chain as non-reflex v_{i-1}

[Mount]

Monotone polygon triangulation algorithm

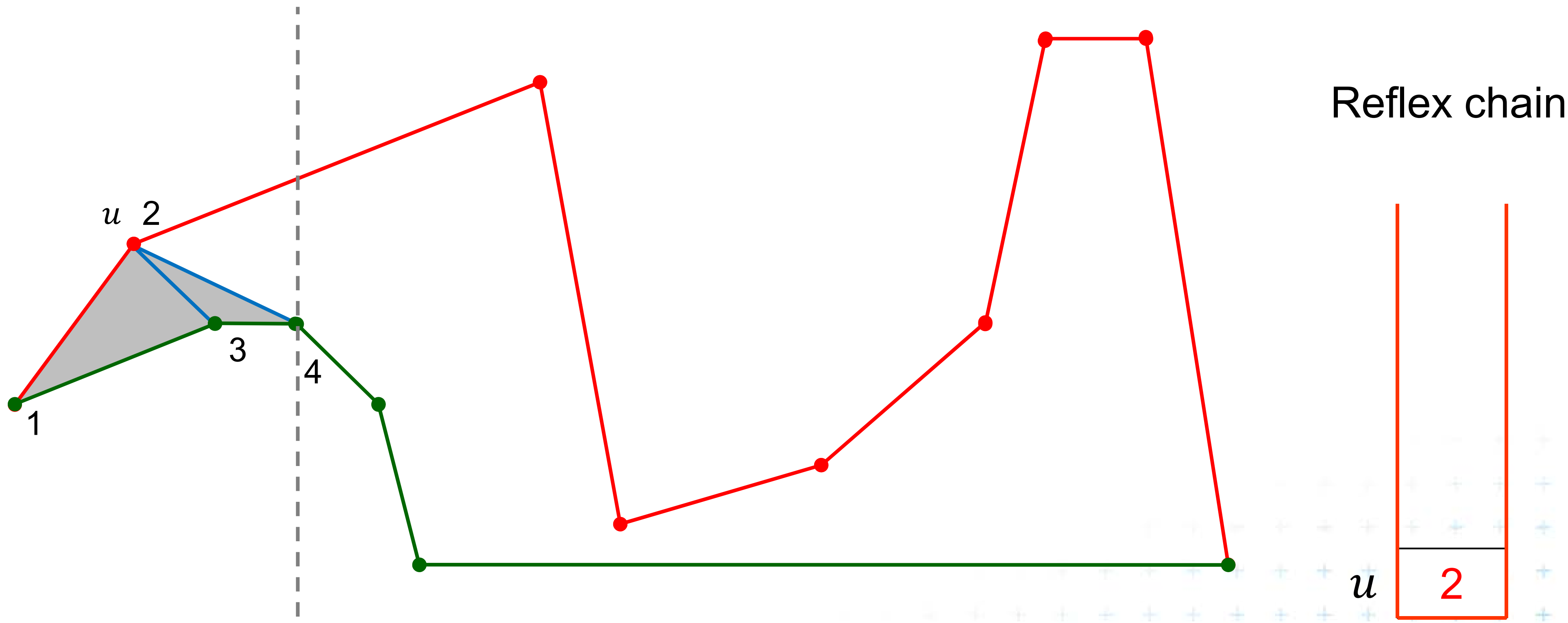


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to visible points on reflex chain – pop()

[Mount]

Monotone polygon triangulation algorithm

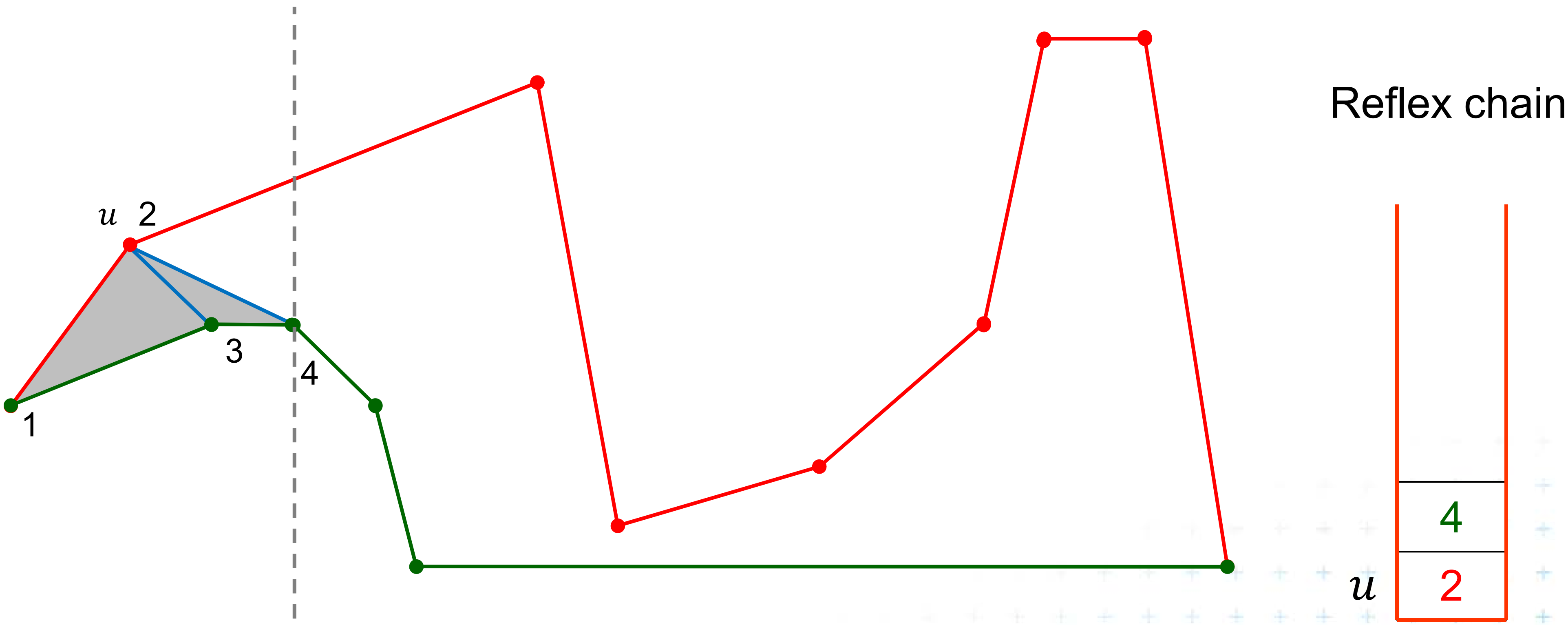


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to visible points on reflex chain – pop()

[Mount]

Monotone polygon triangulation algorithm



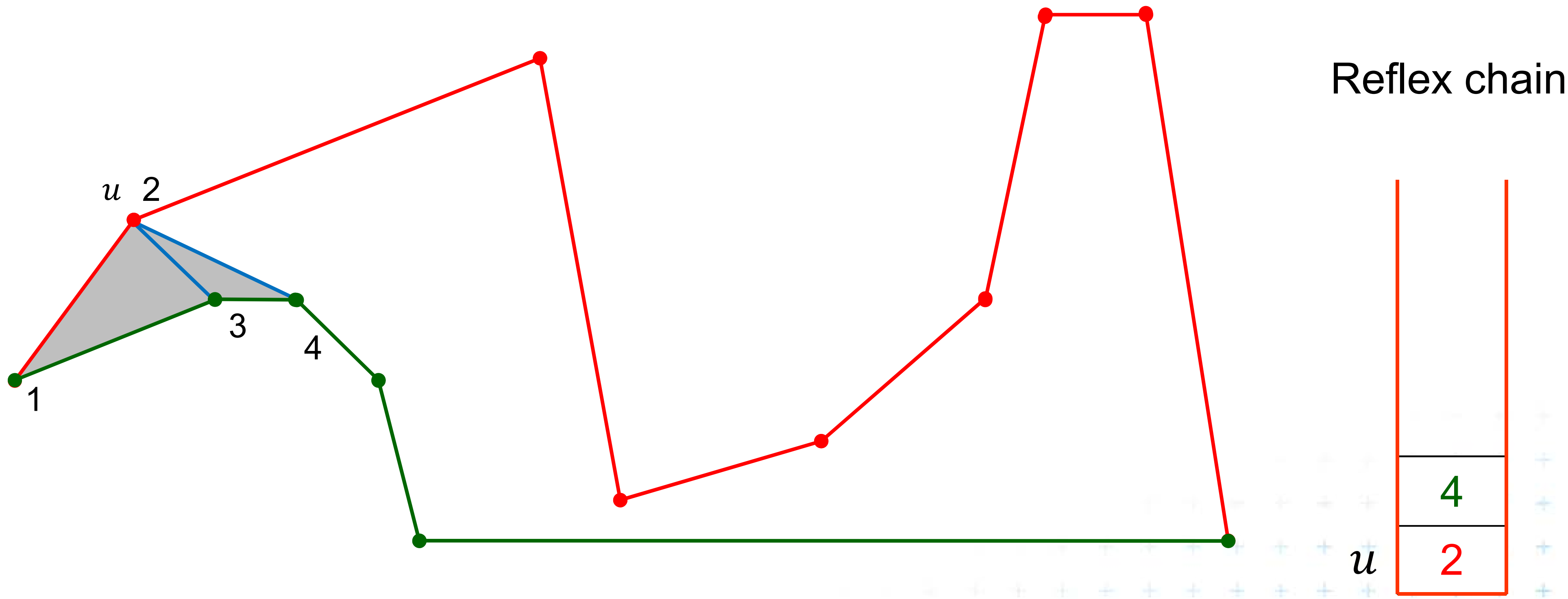
Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to visible points on reflex chain – pop()

Leave the last visible. Add v_i to reflex chain stack – push(v_i)

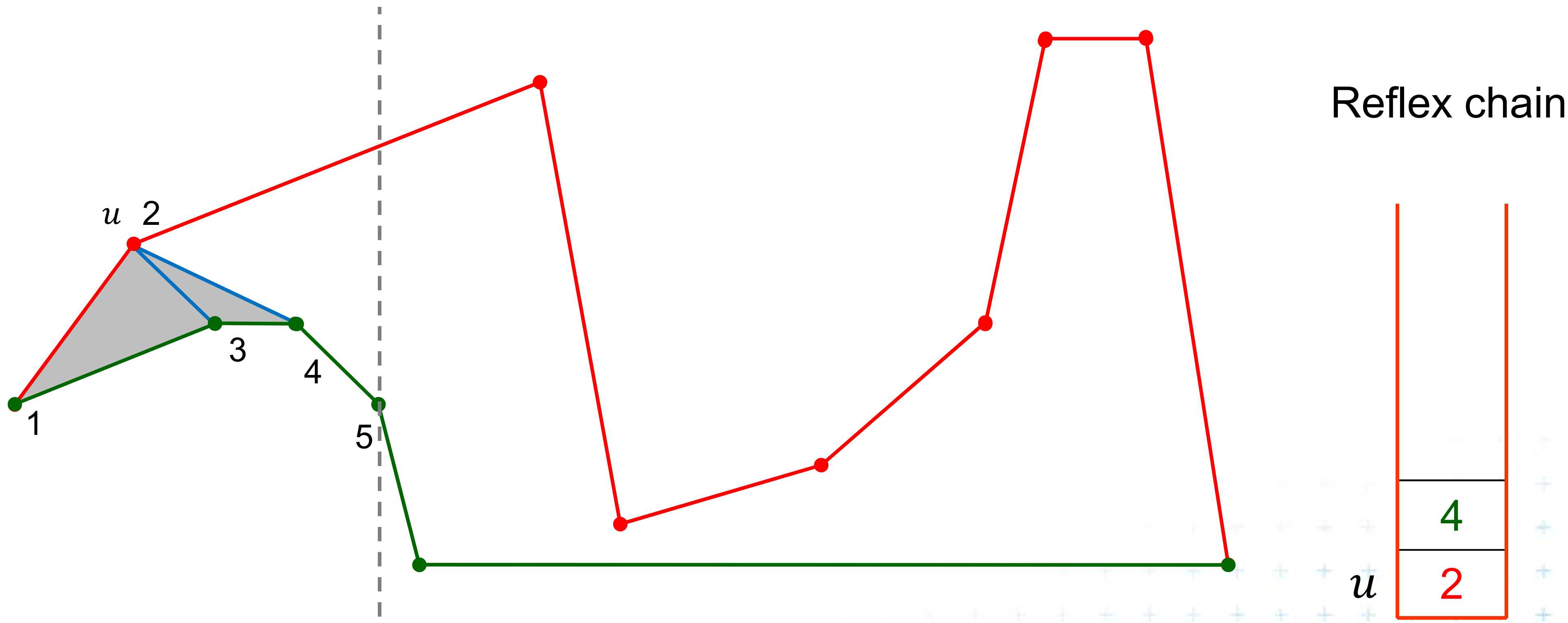
[Mount]

Monotone polygon triangulation algorithm



[Mount]

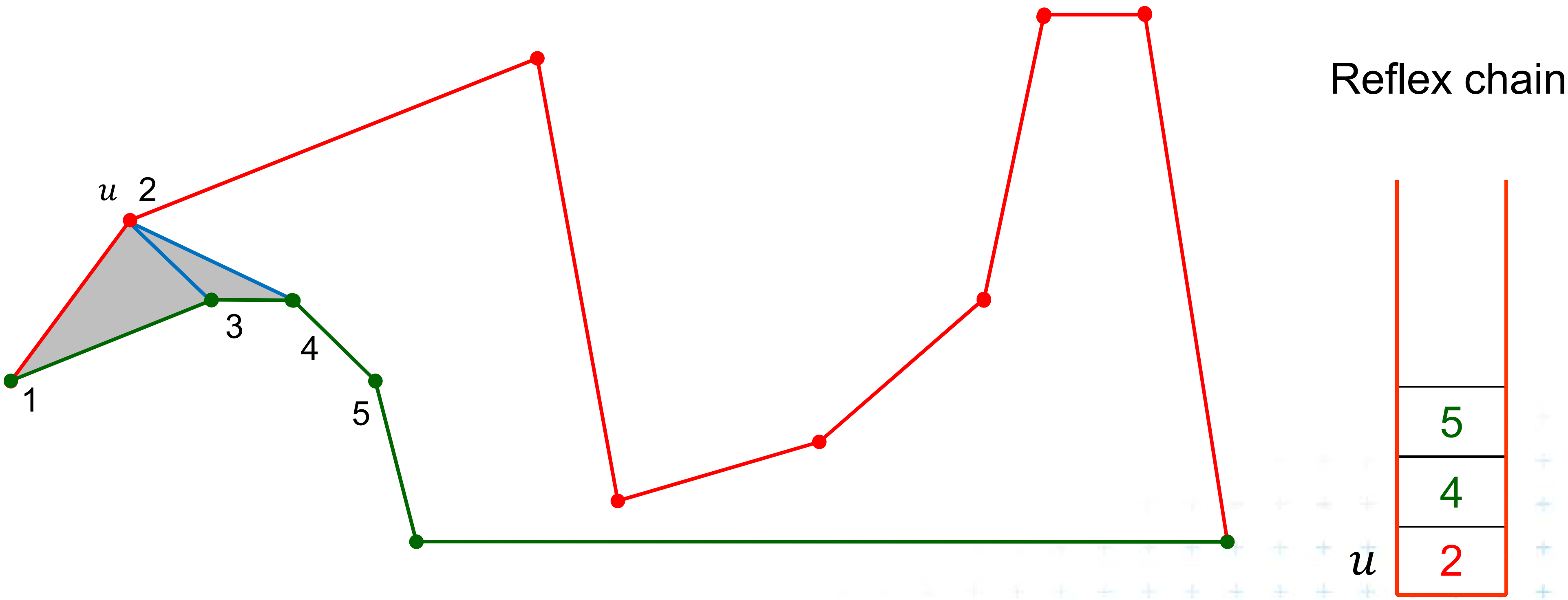
Monotone polygon triangulation algorithm



Case 2b – point v_i on the same chain as reflex v_{i-1}

[Mount]

Monotone polygon triangulation algorithm

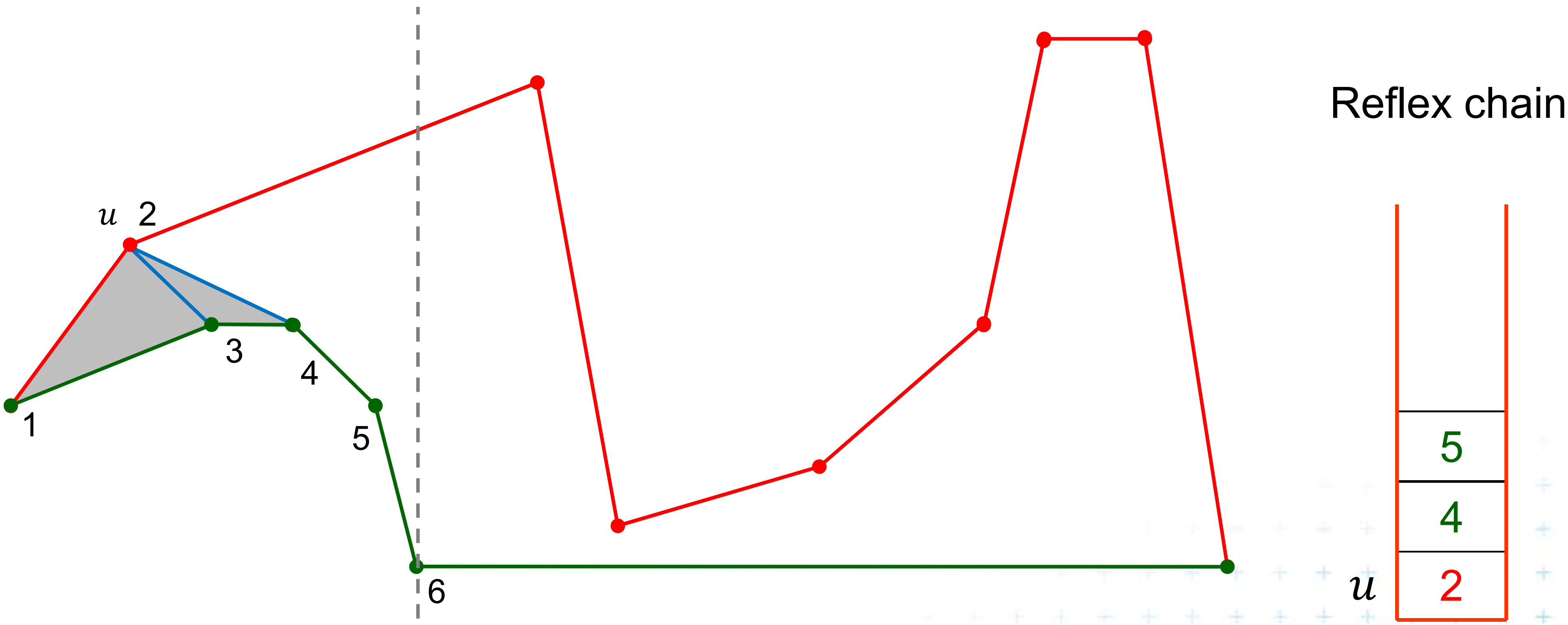


Case 2b – point v_i on the same chain as reflex v_{i-1}

Push point v_i to the reflex chain stack

[Mount]

Monotone polygon triangulation algorithm

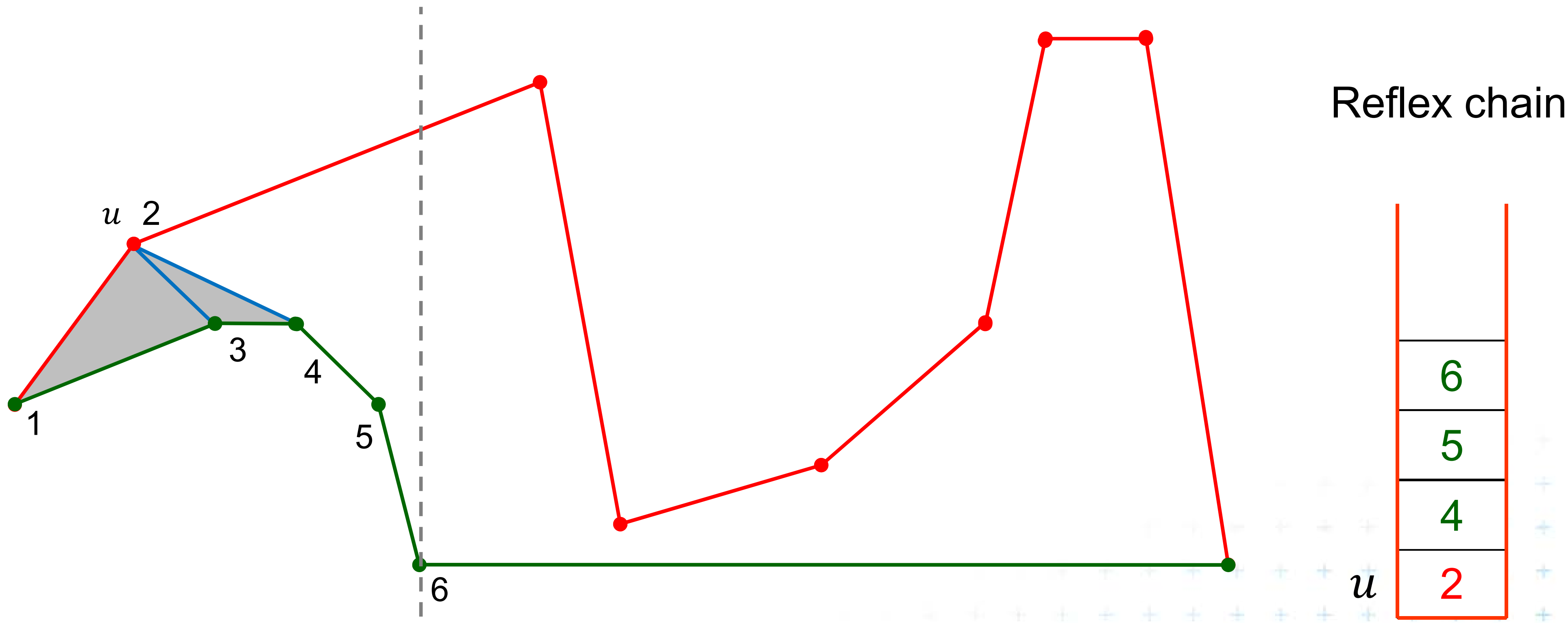


Case 2b – point v_i on the same chain as reflex v_{i-1}

Push point v_i to the reflex chain stack

[Mount]

Monotone polygon triangulation algorithm

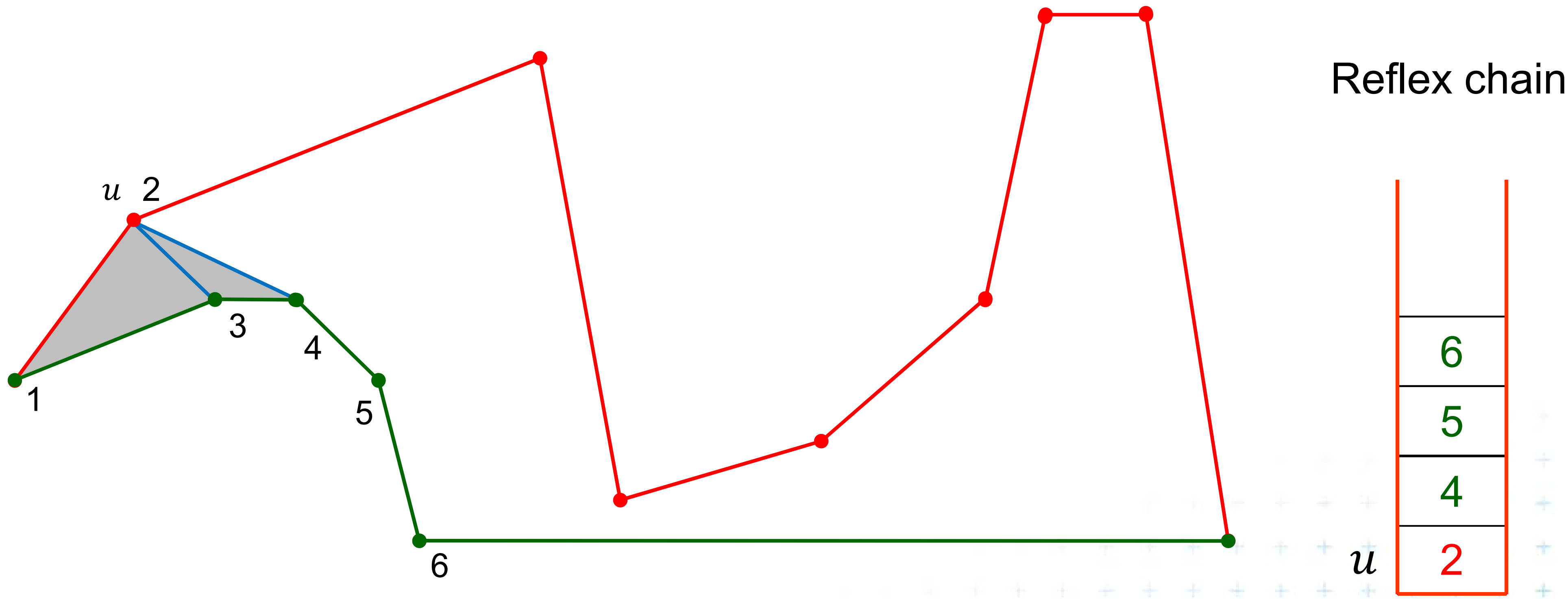


Case 2b – point v_i on the same chain as reflex v_{i-1}

Push point v_i to the reflex chain stack

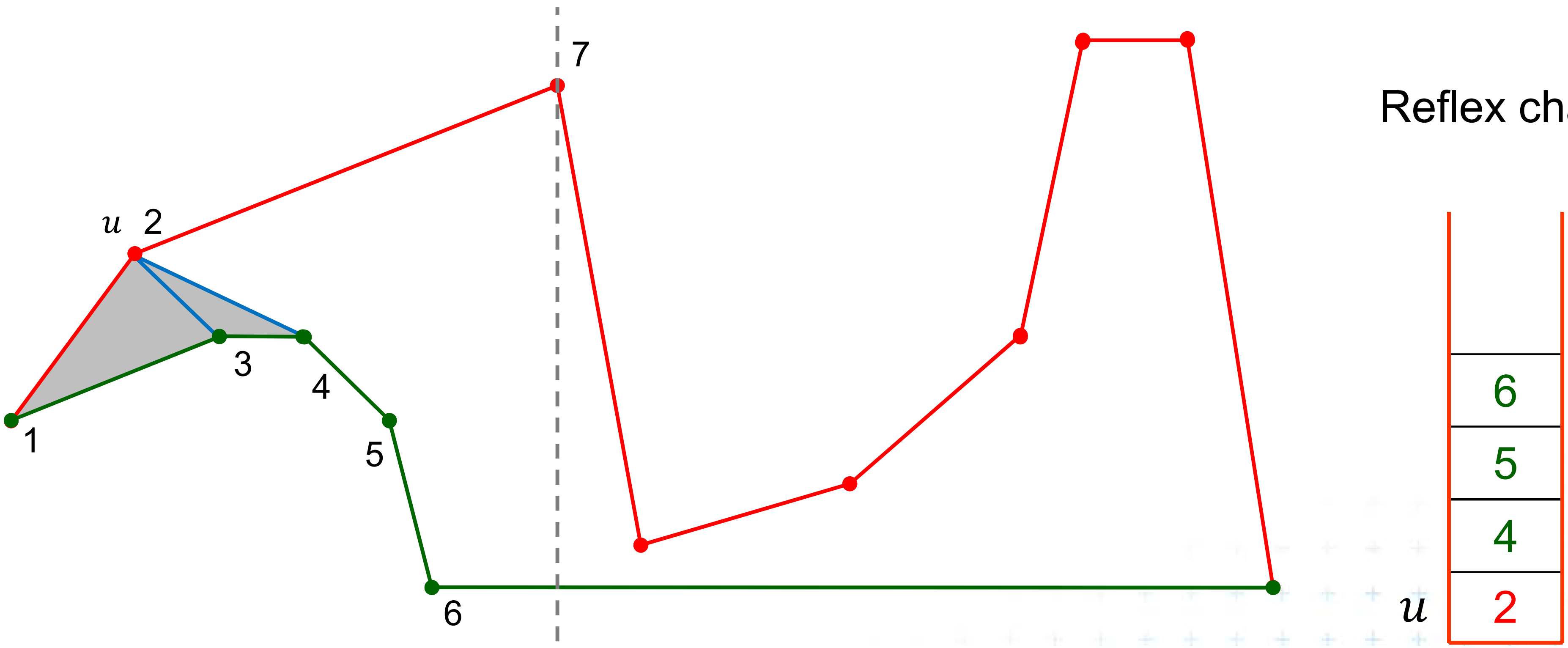
[Mount]

Monotone polygon triangulation algorithm



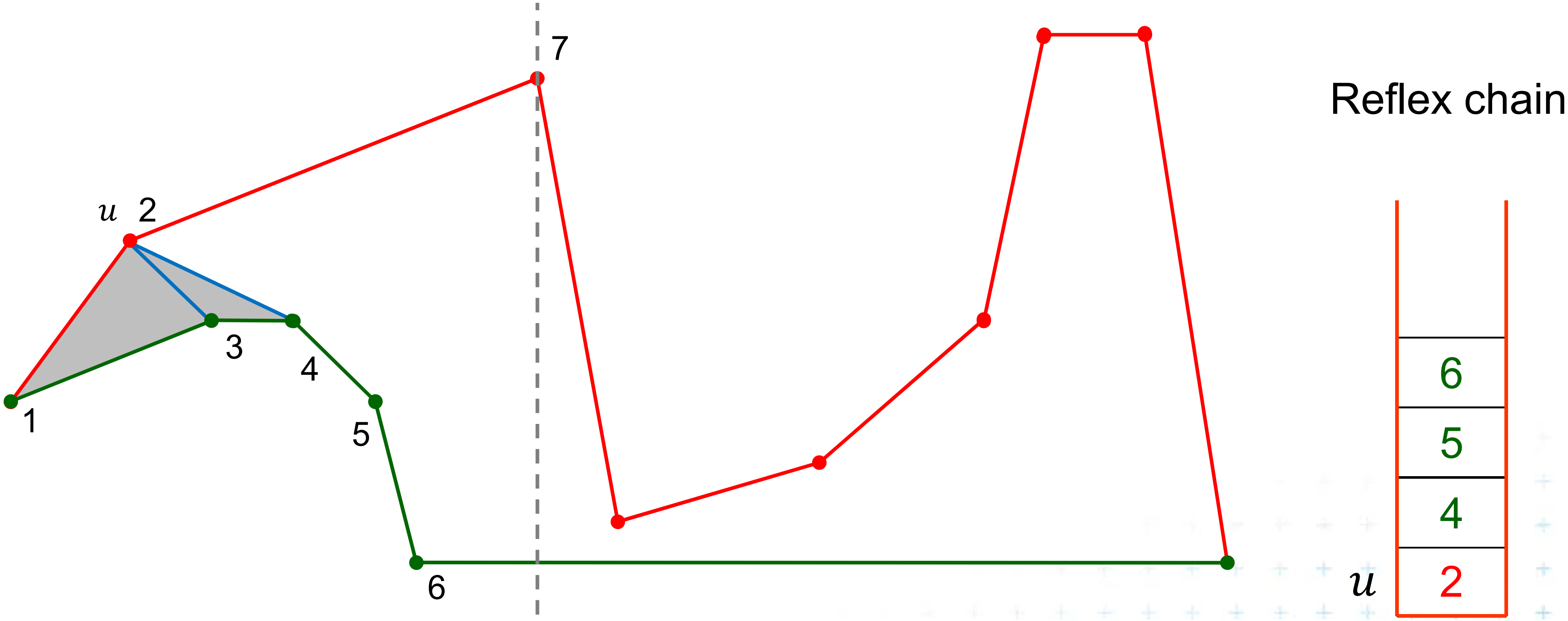
[Mount]

Monotone polygon triangulation algorithm



[Mount]

Monotone polygon triangulation algorithm

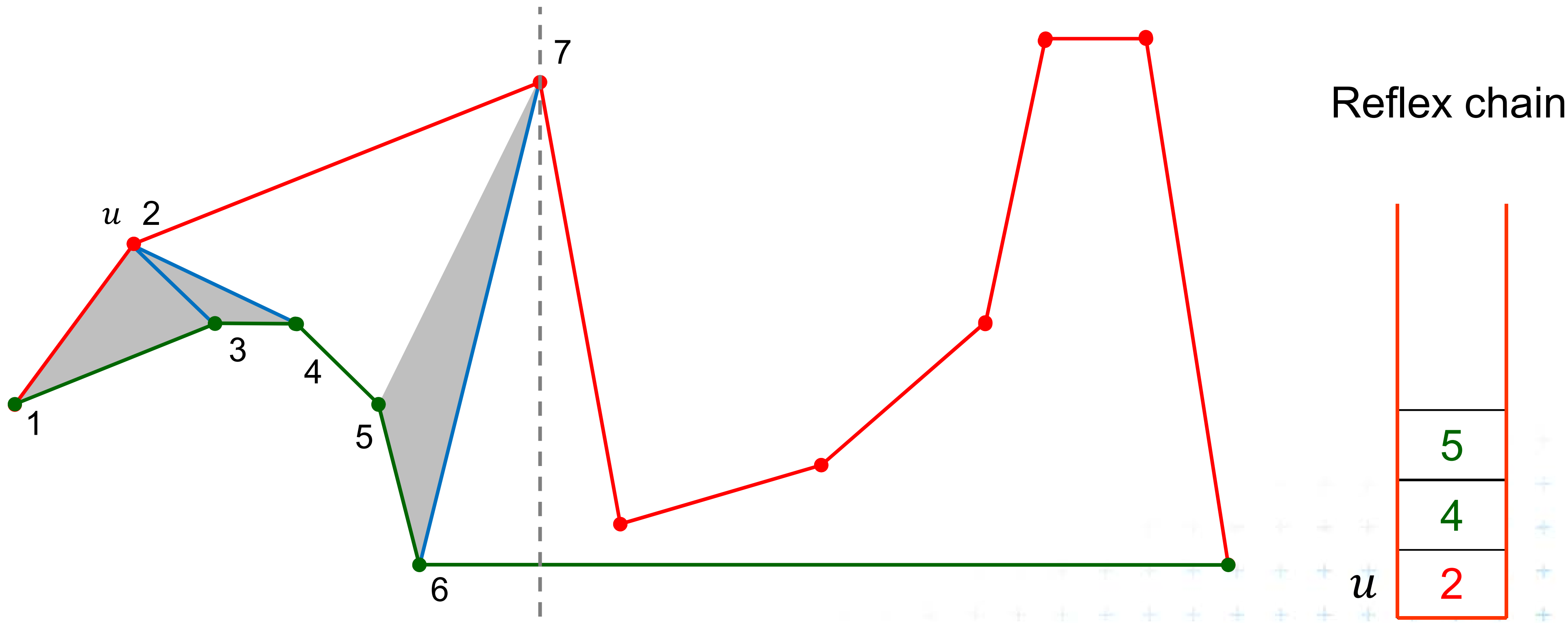


Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm

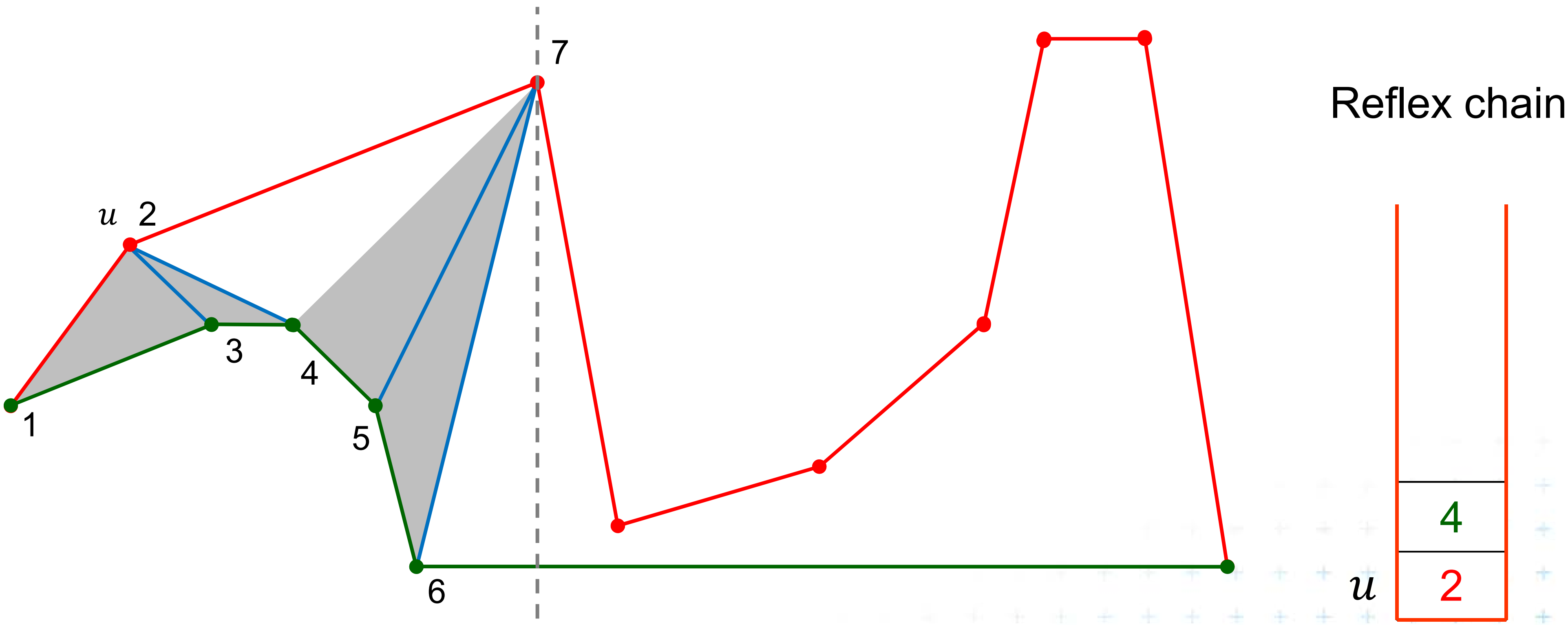


Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm

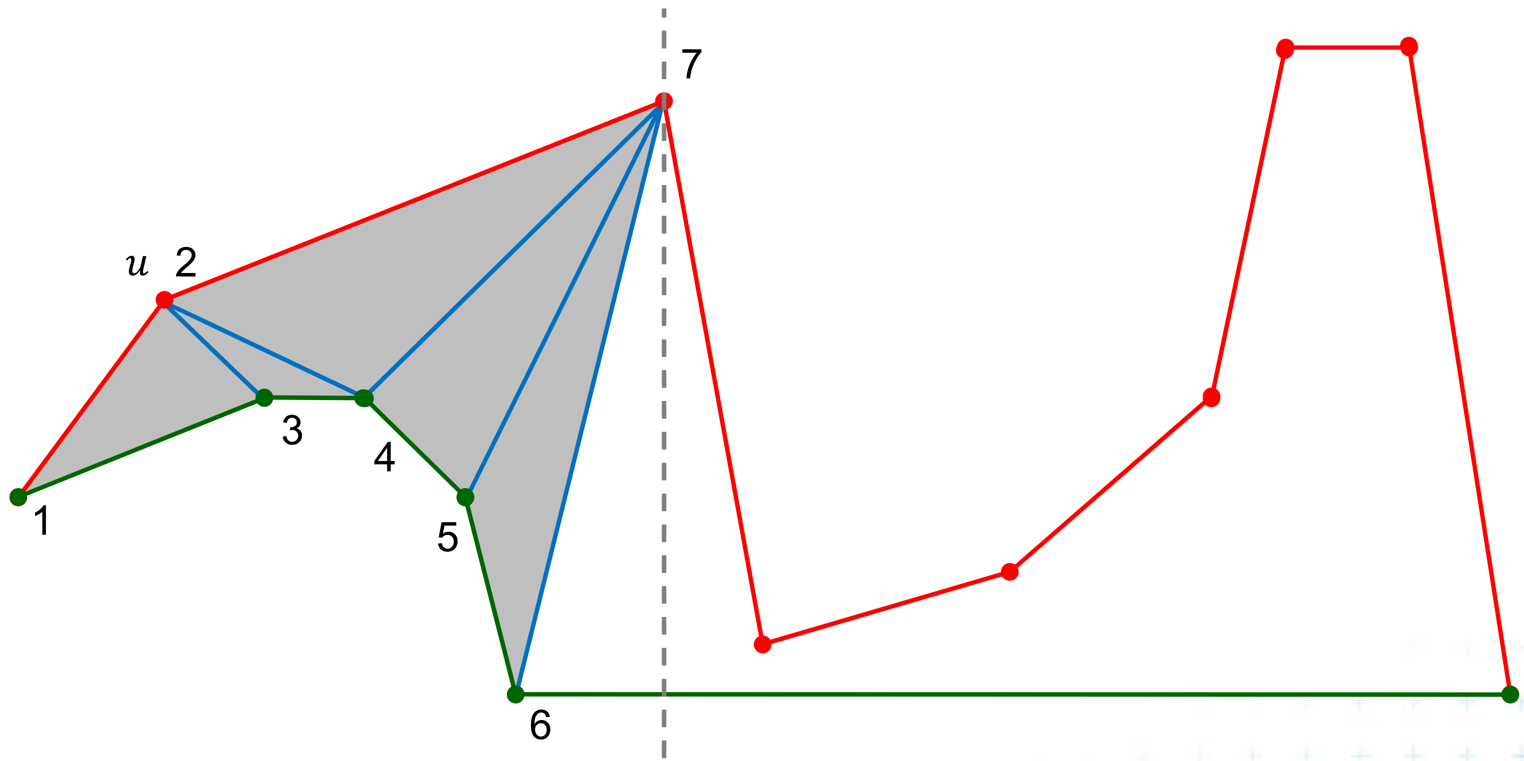


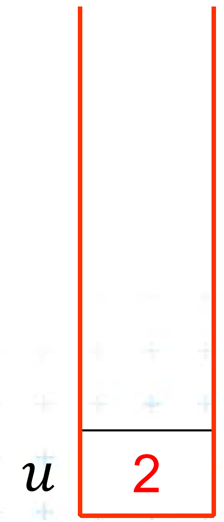
Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm



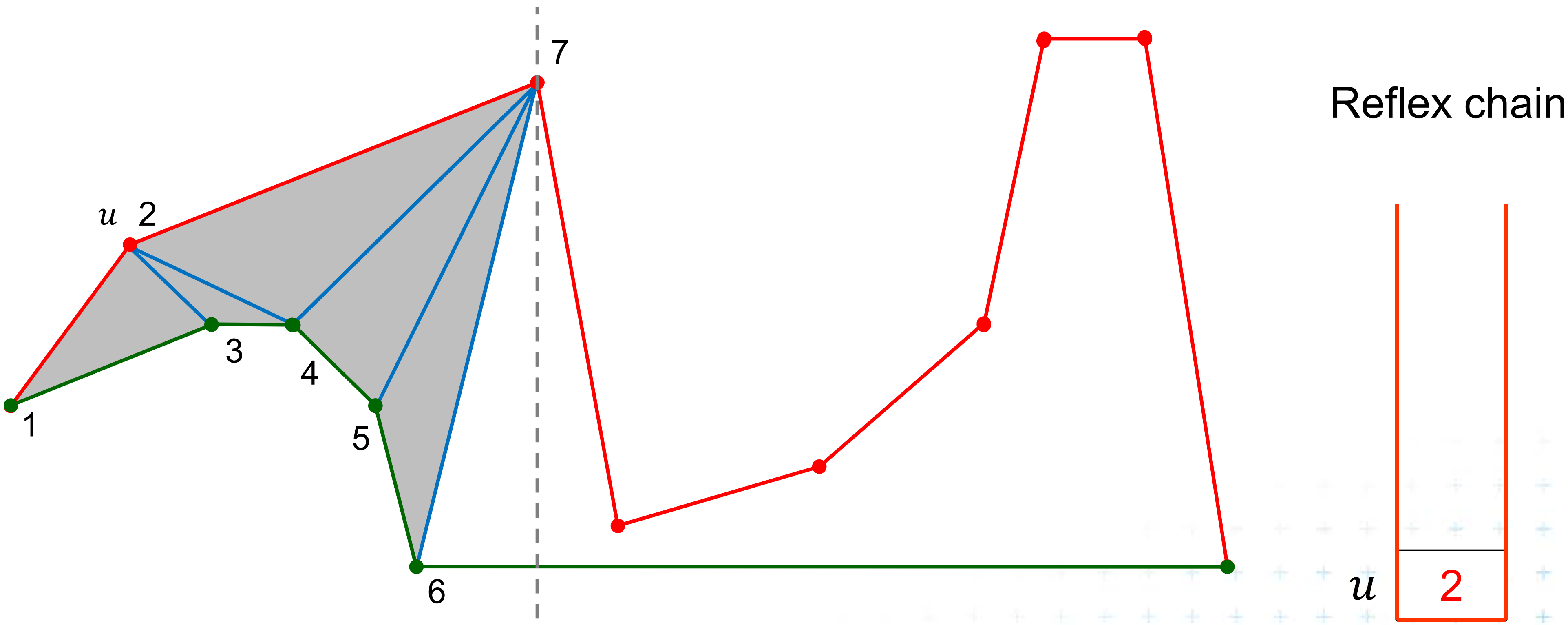


Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm



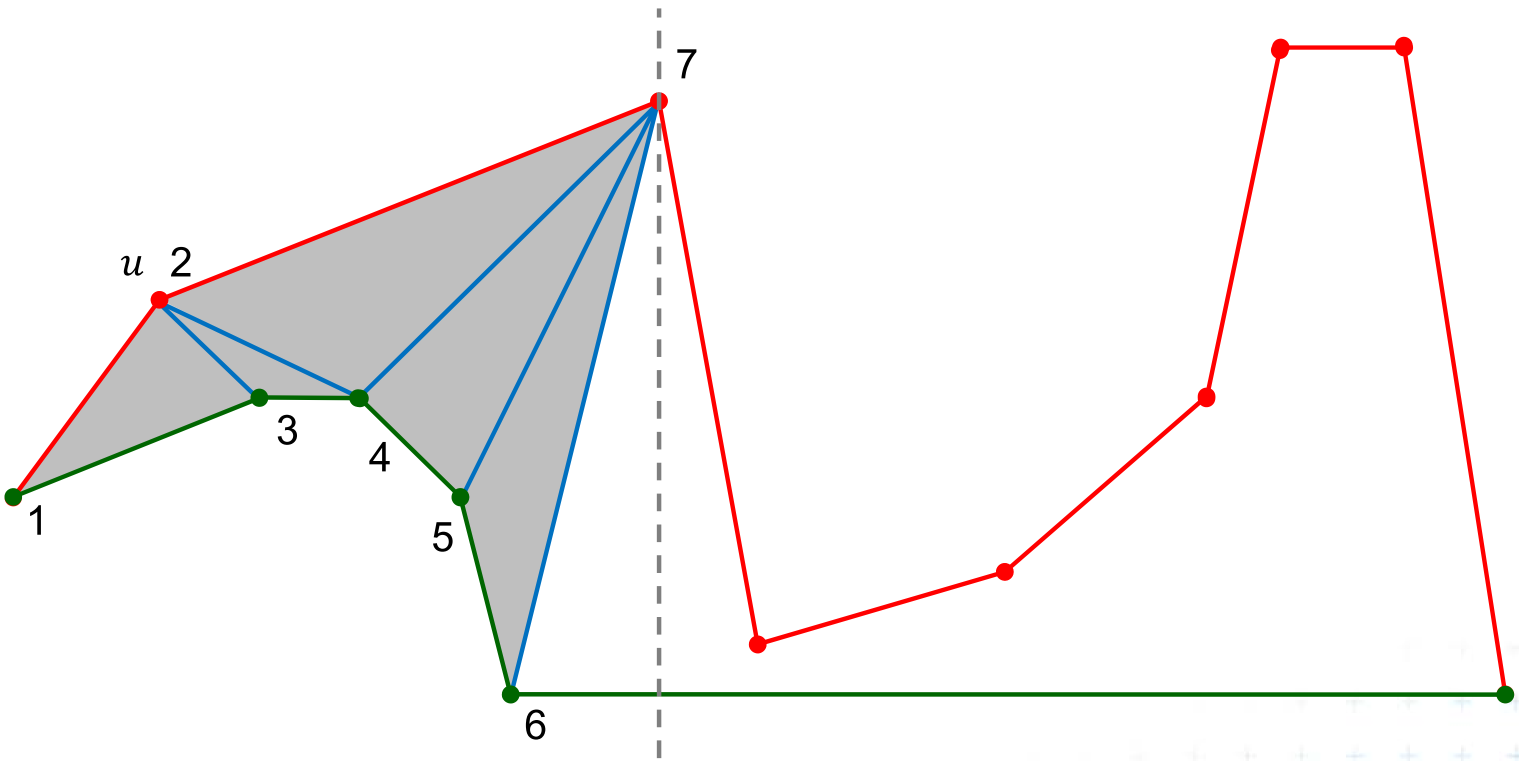
Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

Set trivial reflex chain $v_i v_{i-1}$: New u: pop(), push(v_{i-1}), push(v_i)

[Mount]

Monotone polygon triangulation algorithm



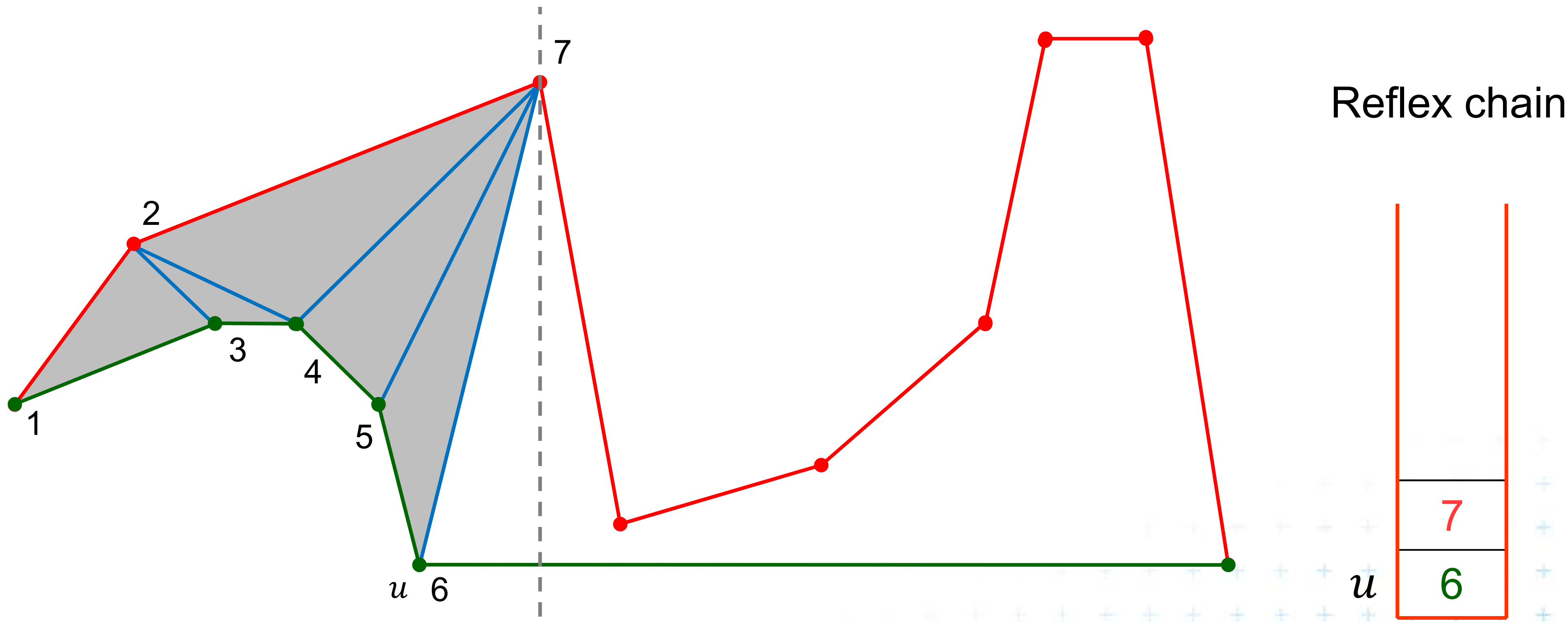
Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

Set trivial reflex chain $v_i v_{i-1}$: New u: pop(), push(v_{i-1}), push(v_i)

[Mount]

Monotone polygon triangulation algorithm



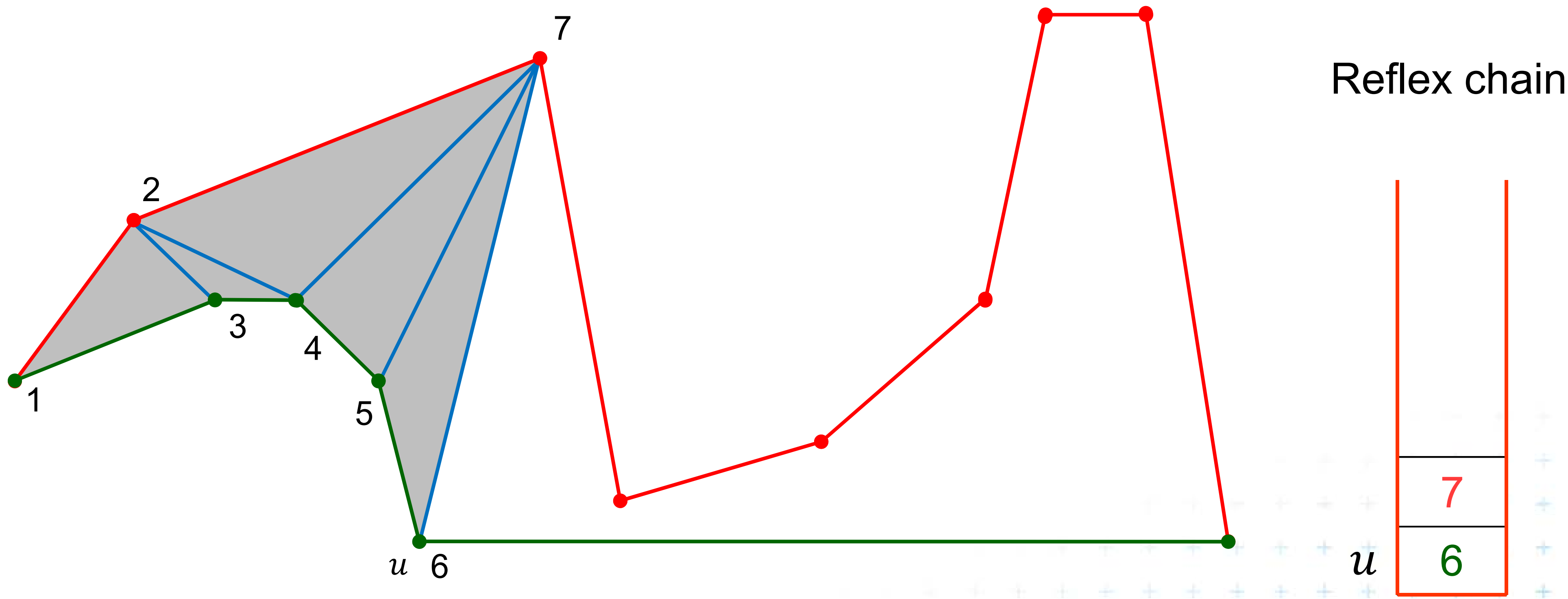
Case 1 – point v_i on opposite chain from v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

Set trivial reflex chain $v_i v_{i-1}$: New u: pop(), push(v_{i-1}), push(v_i)

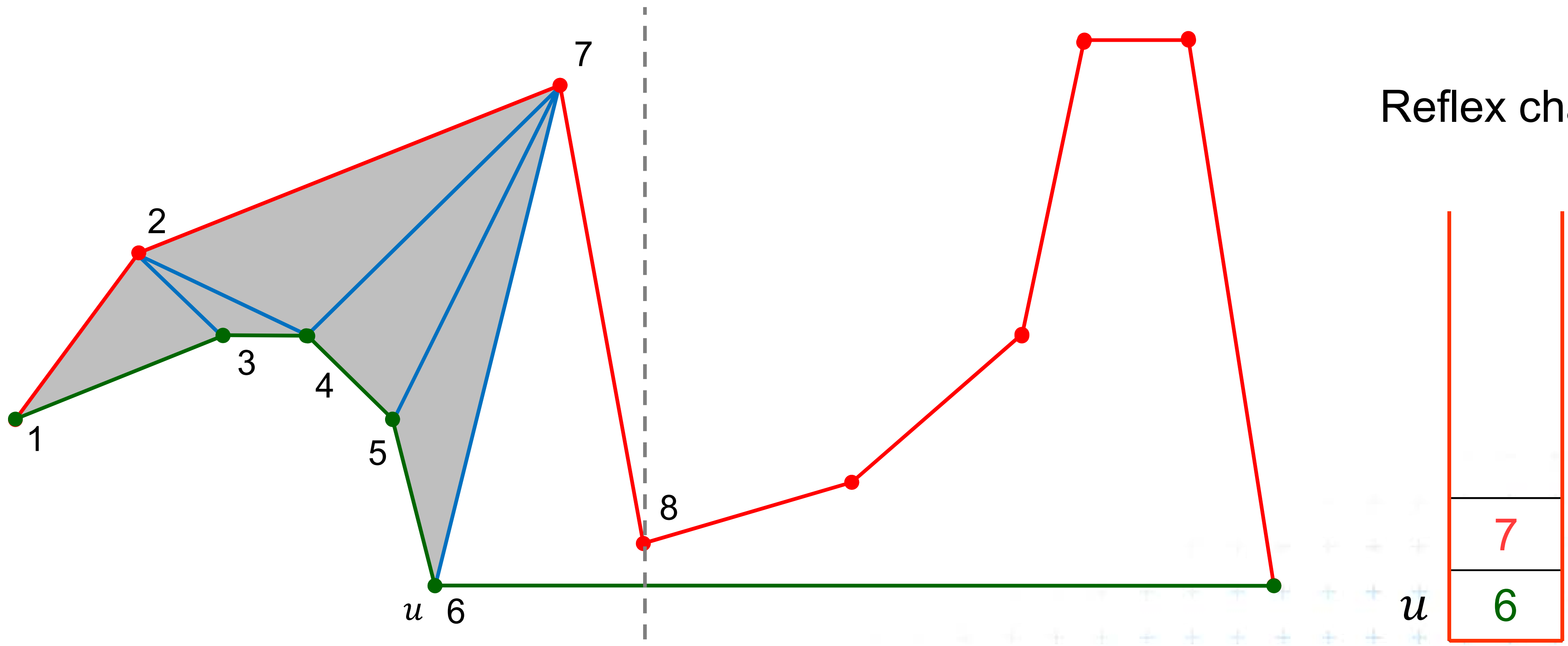
[Mount]

Monotone polygon triangulation algorithm



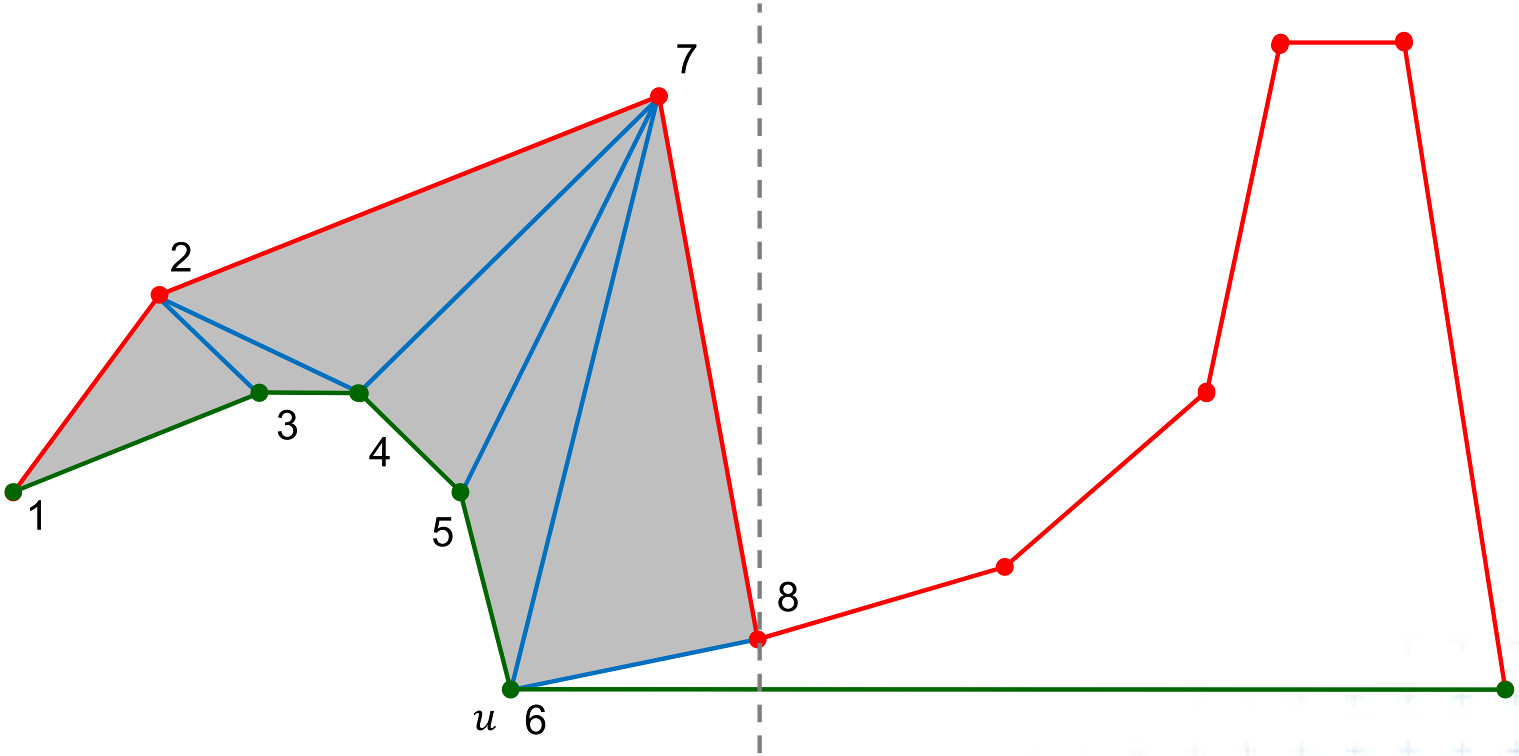
[Mount]

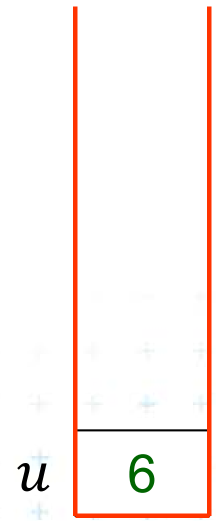
Monotone polygon triangulation algorithm



[Mount]

Monotone polygon triangulation algorithm



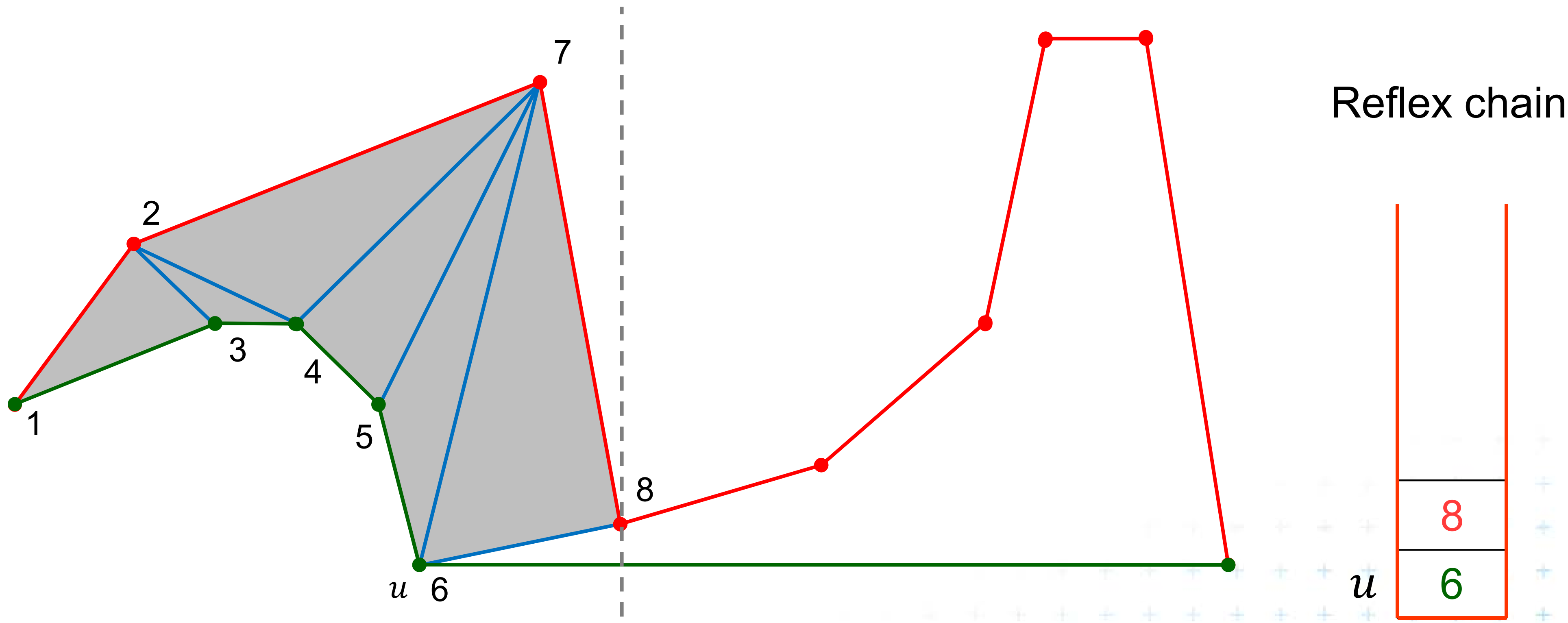


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to visible points on reflex chain – pop()

[Mount]

Monotone polygon triangulation algorithm



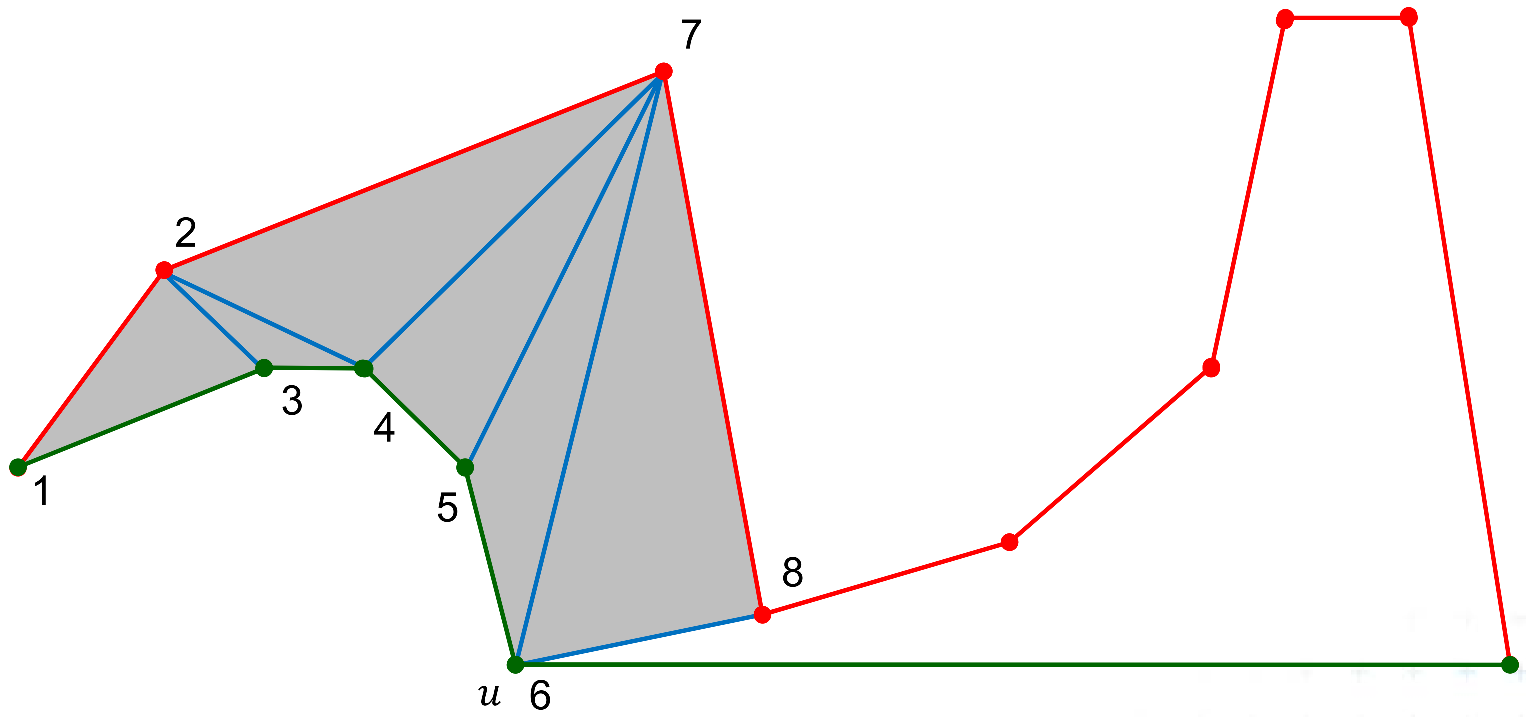
Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to visible points on reflex chain – pop()

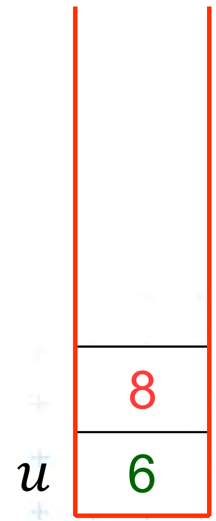
Leave the last visible. Add v_i to reflex chain stack – push(v_i)

[Mount]

Monotone polygon triangulation algorithm

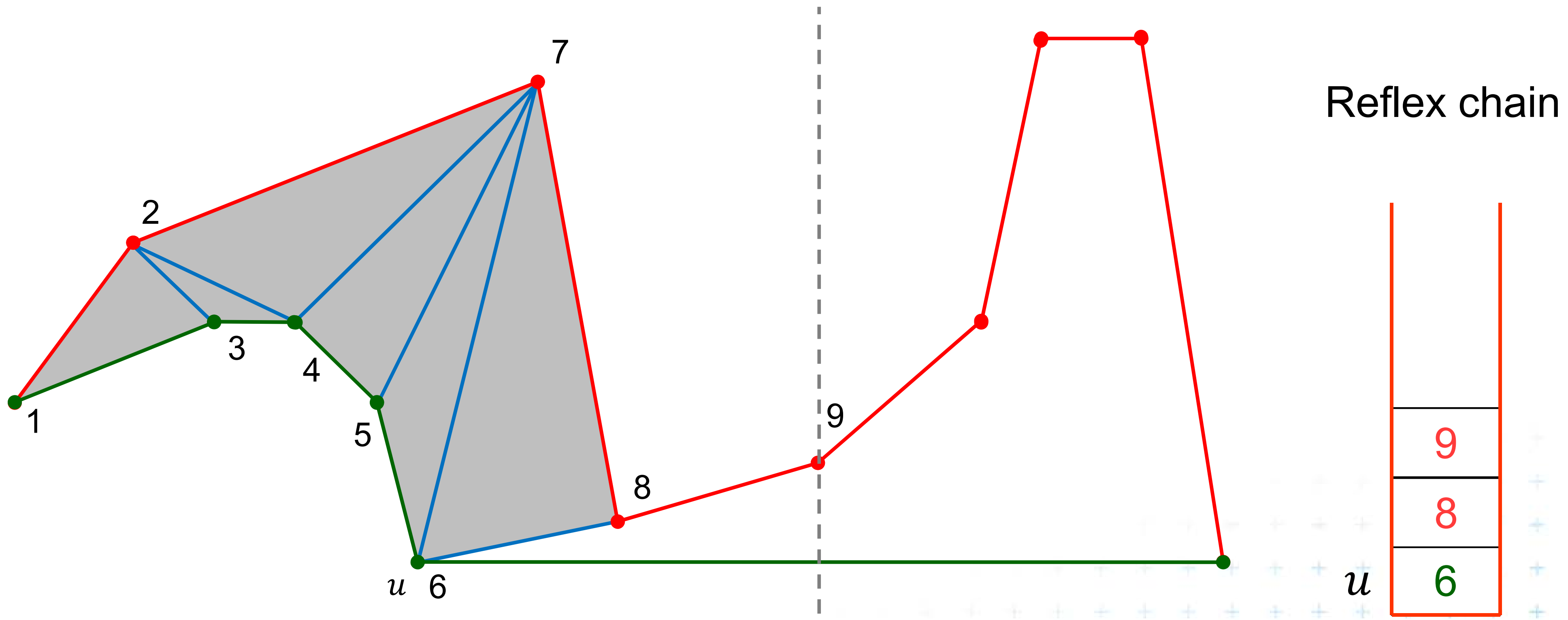


Reflex chain



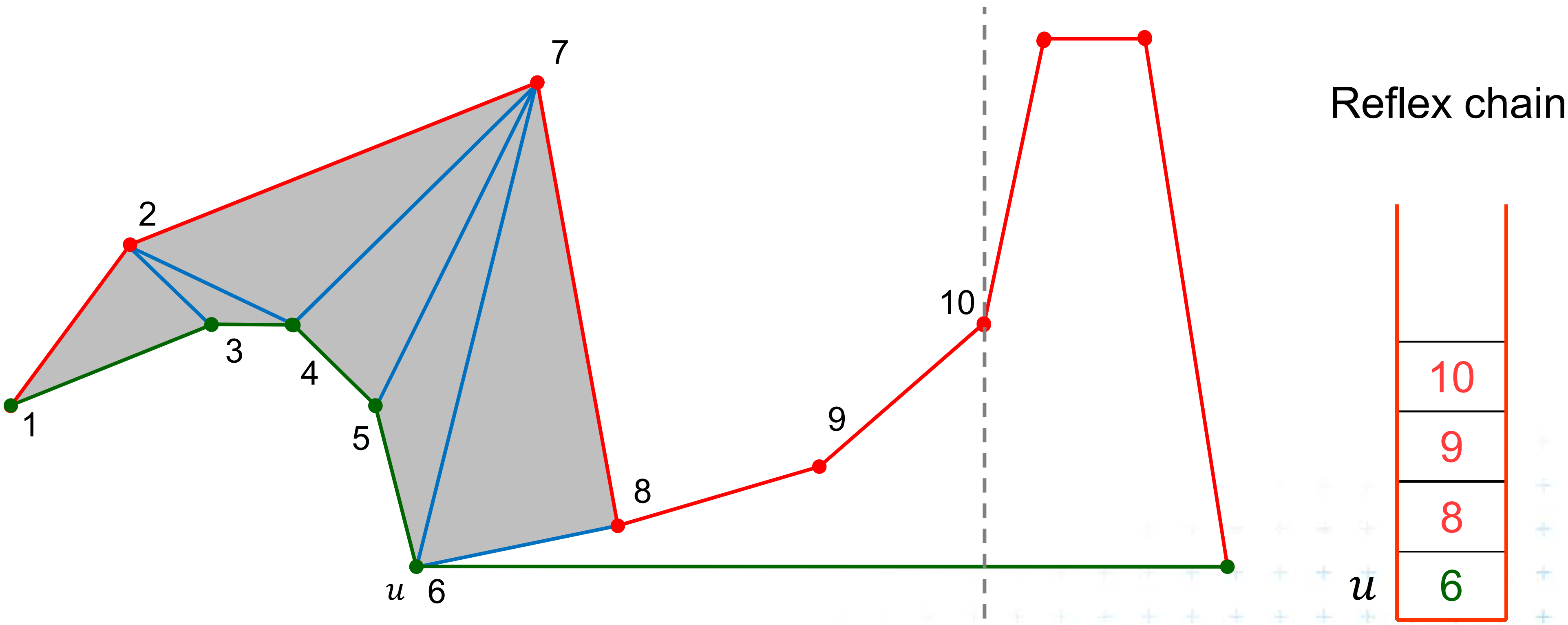
[Mount]

Monotone polygon triangulation algorithm



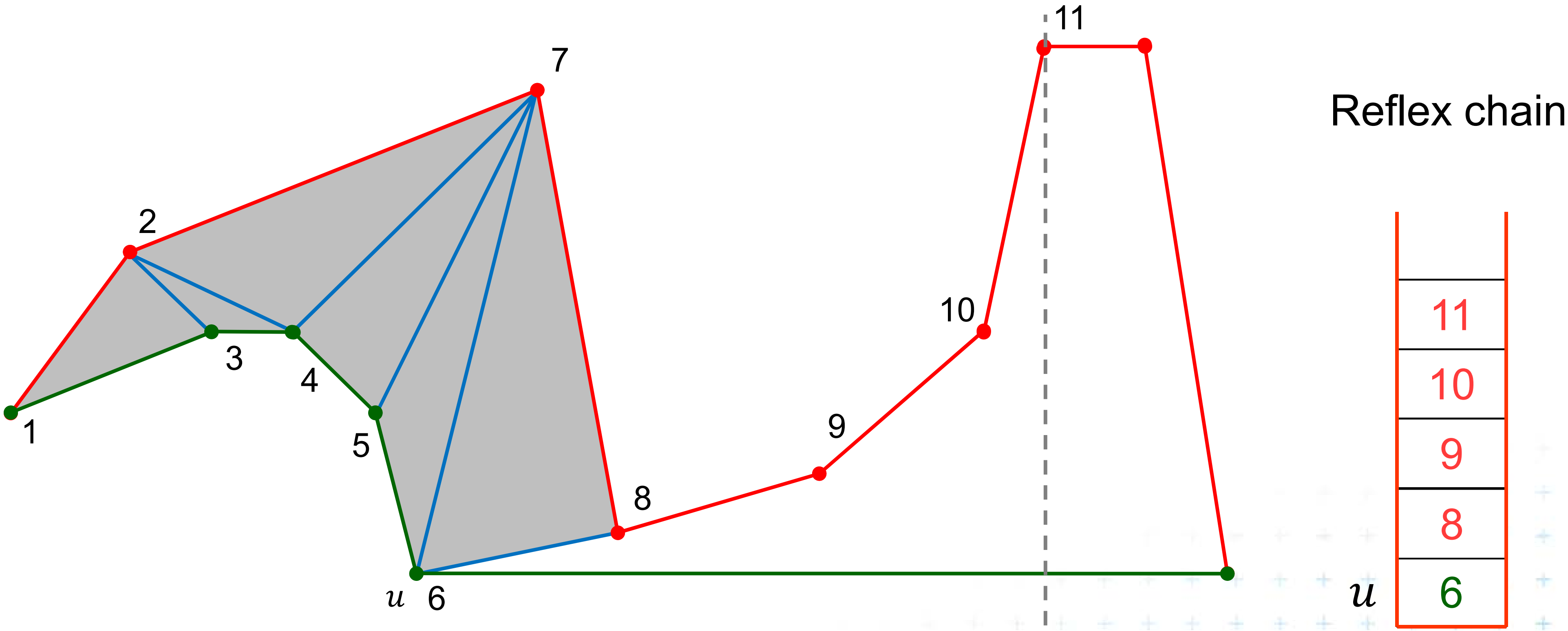
[Mount]

Monotone polygon triangulation algorithm



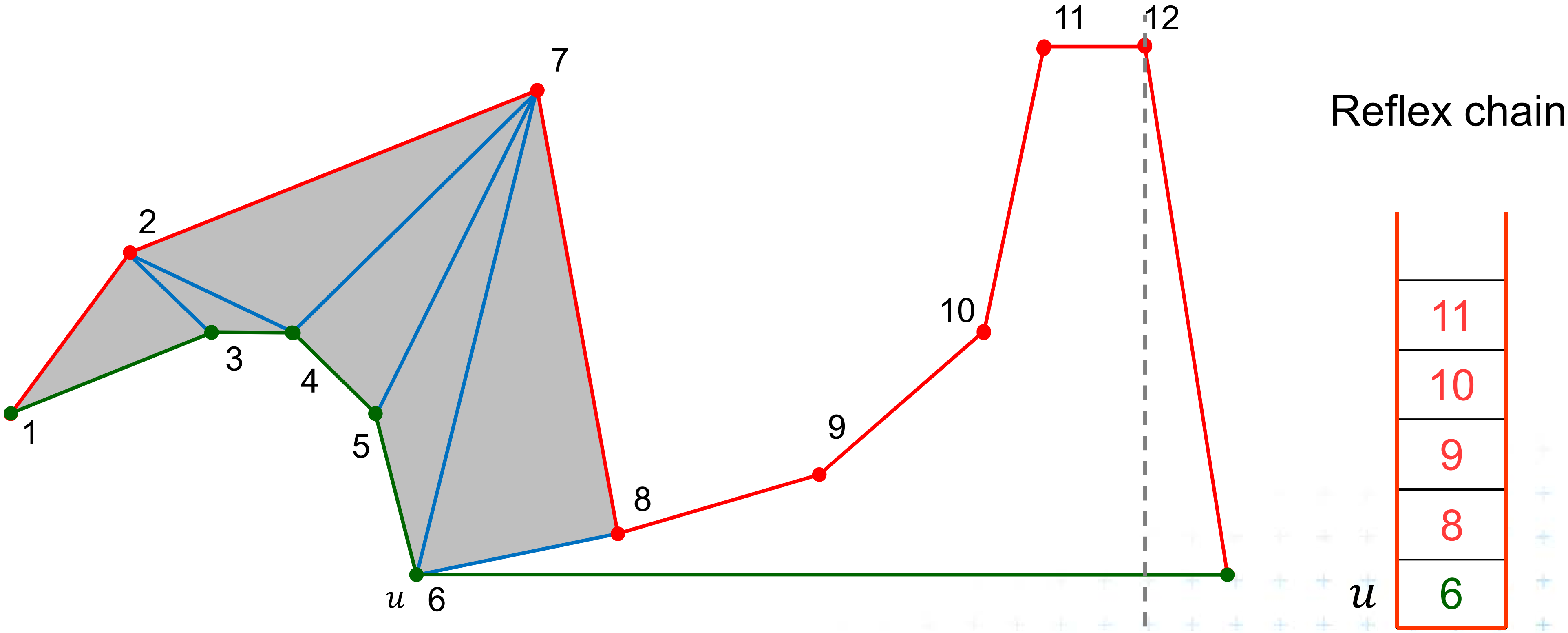
[Mount]

Monotone polygon triangulation algorithm



[Mount]

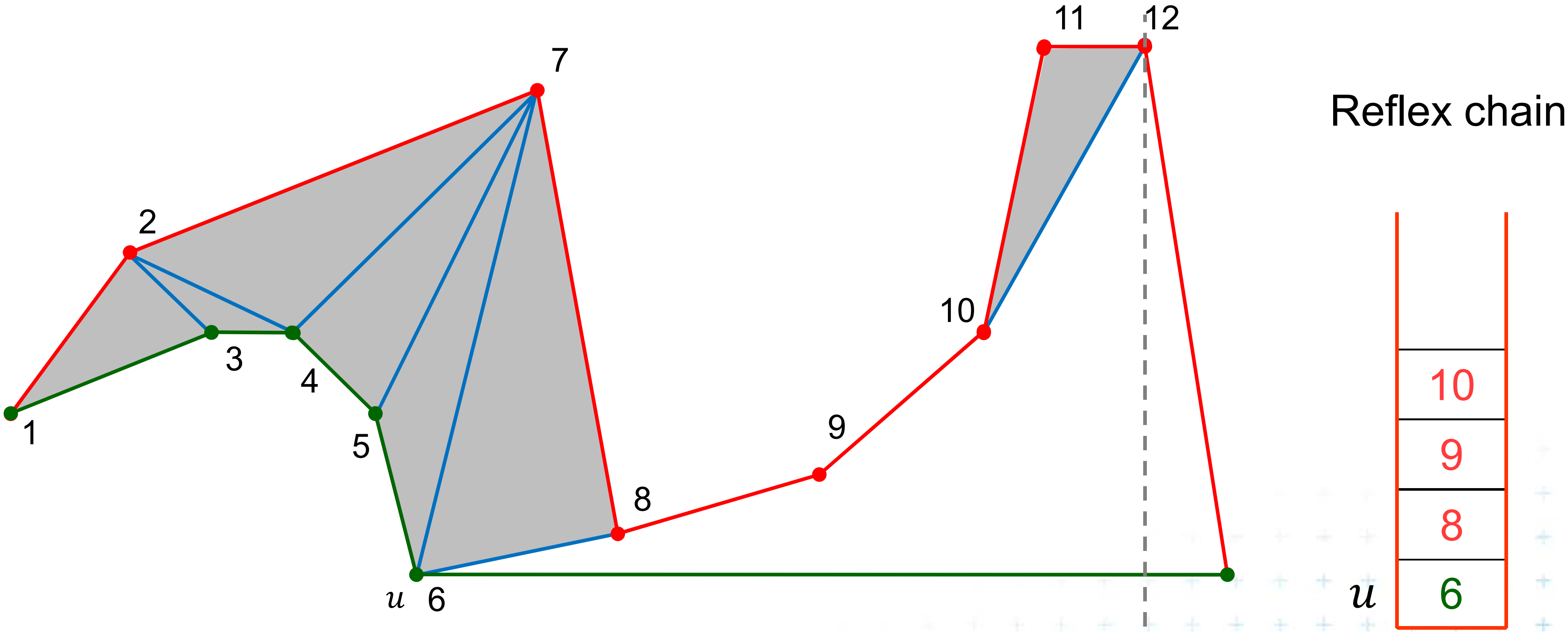
Monotone polygon triangulation algorithm



Case 2a – point v_i on the same chain as non-reflex v_{i-1}

[Mount]

Monotone polygon triangulation algorithm

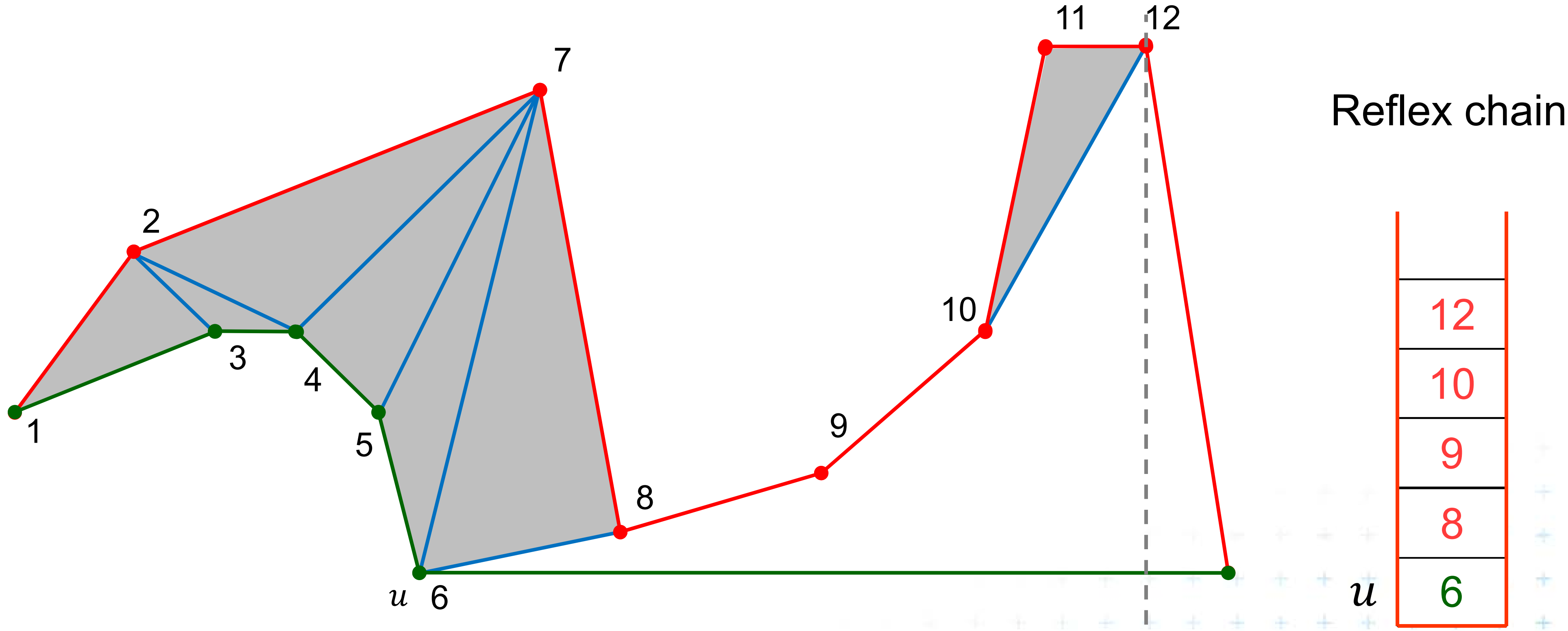


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to visible points on reflex chain – pop()

[Mount]

Monotone polygon triangulation algorithm



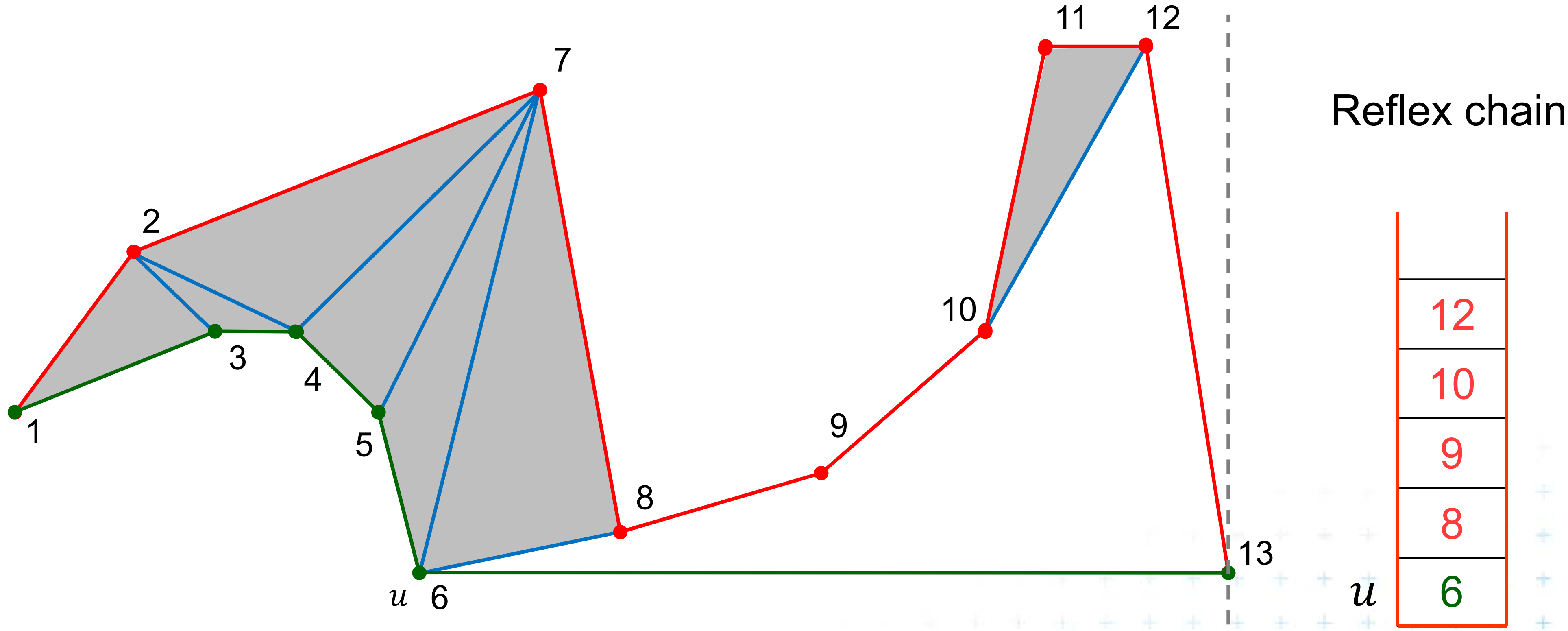
Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to visible points on reflex chain – pop()

Leave the last visible. Add v_i to reflex chain stack – push(v_i)

[Mount]

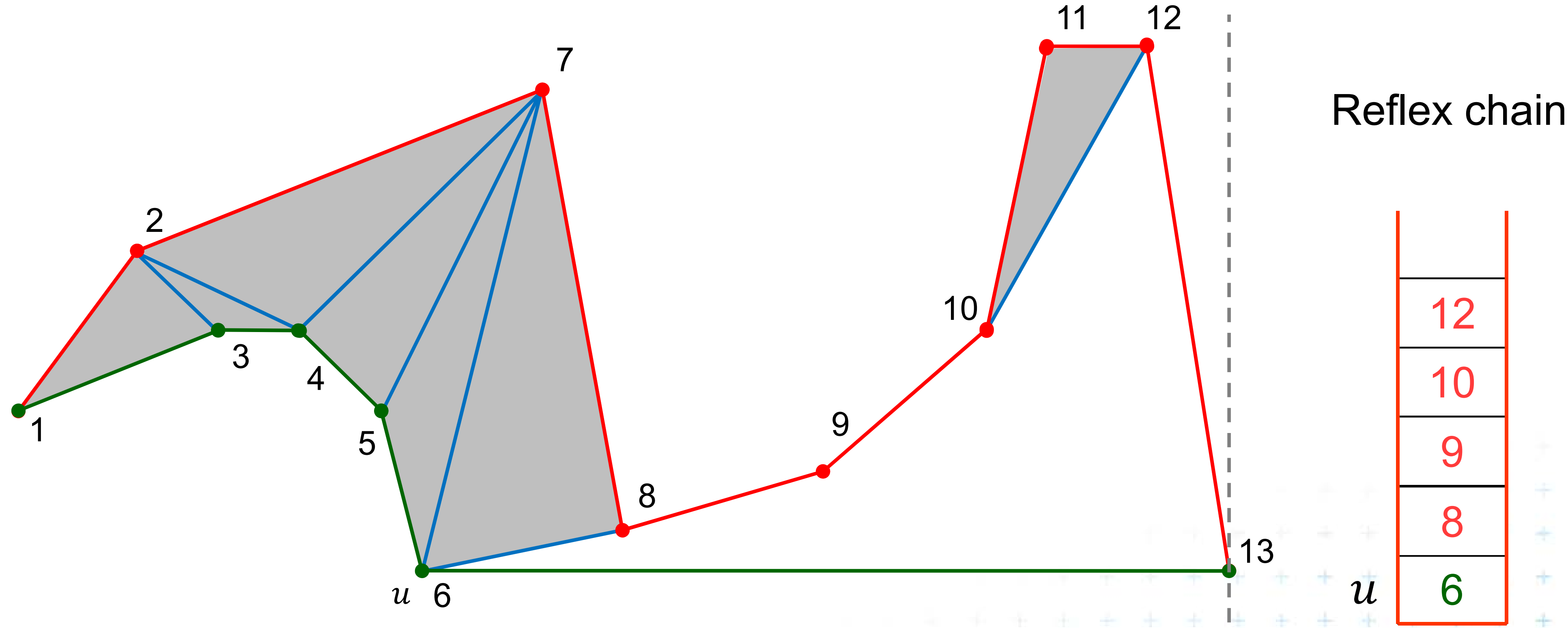
Monotone polygon triangulation algorithm



Case 2a – point v_i on the same chain as non-reflex v_{i-1}

[Mount]

Monotone polygon triangulation algorithm

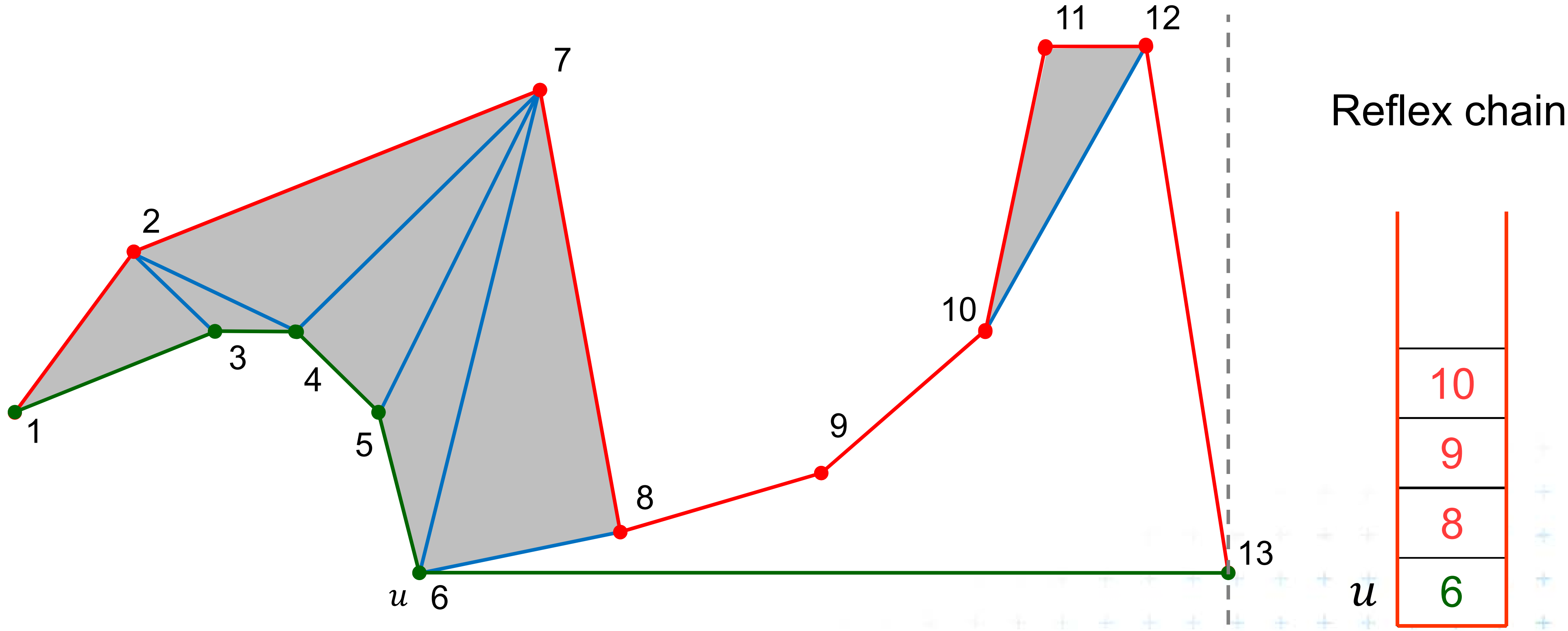


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm

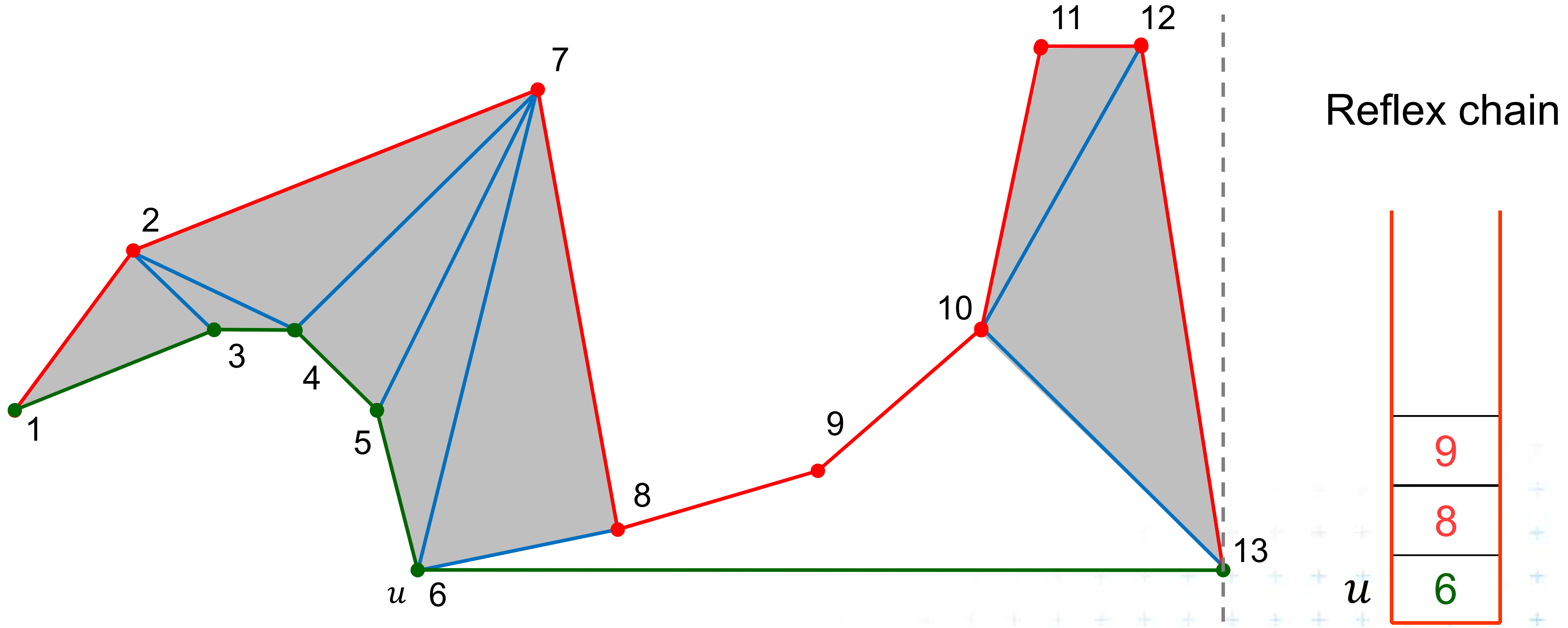


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm

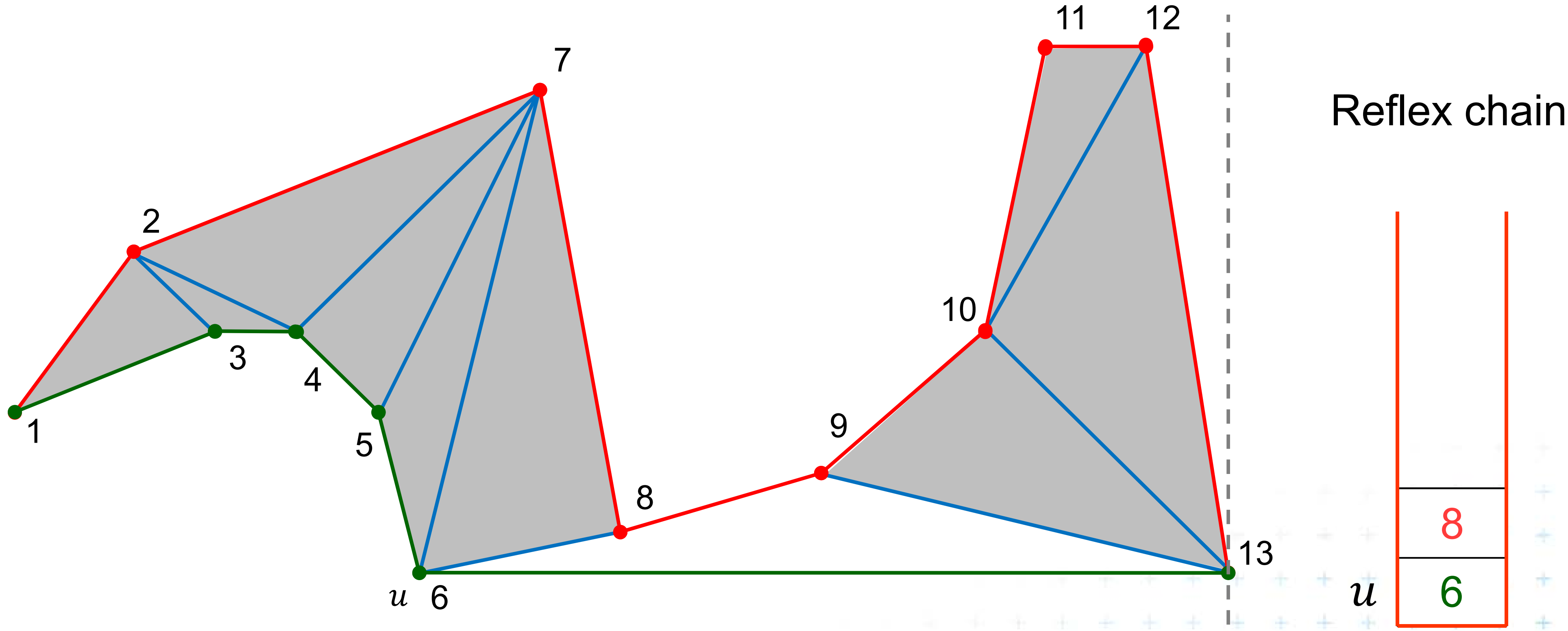


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm

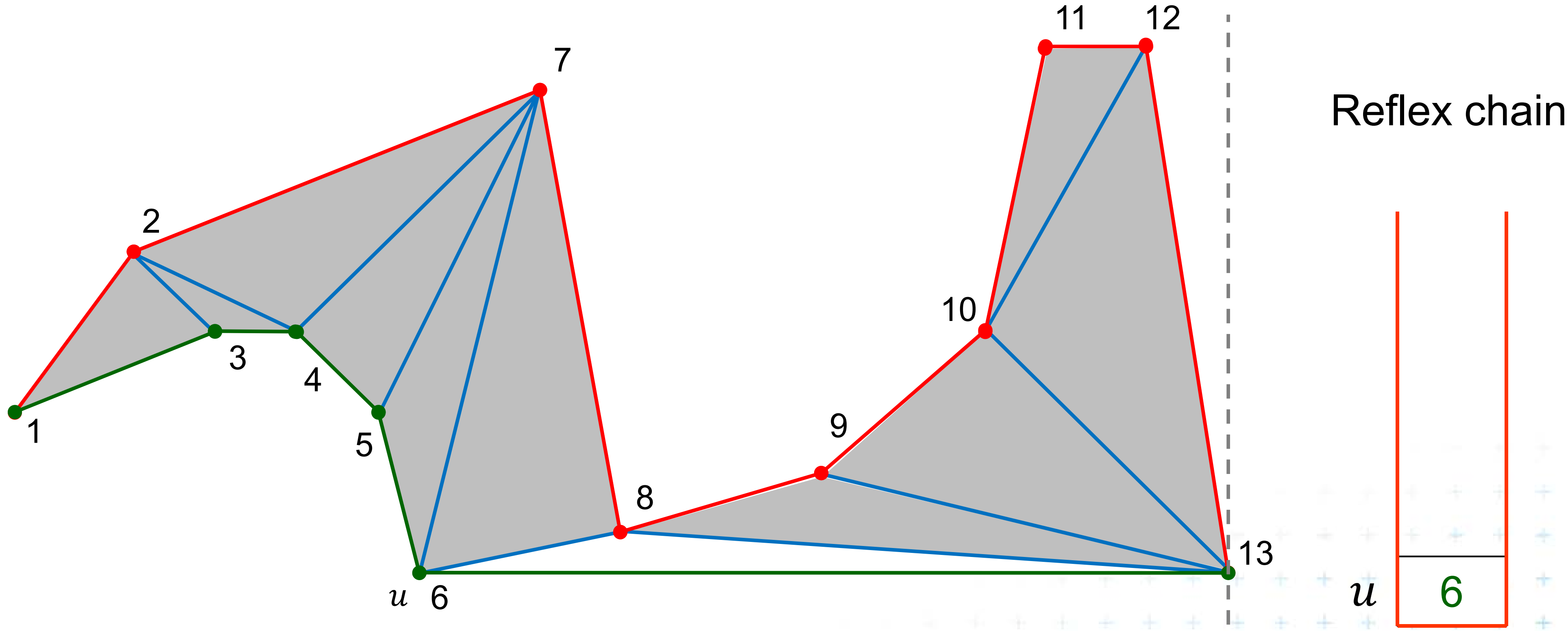


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm

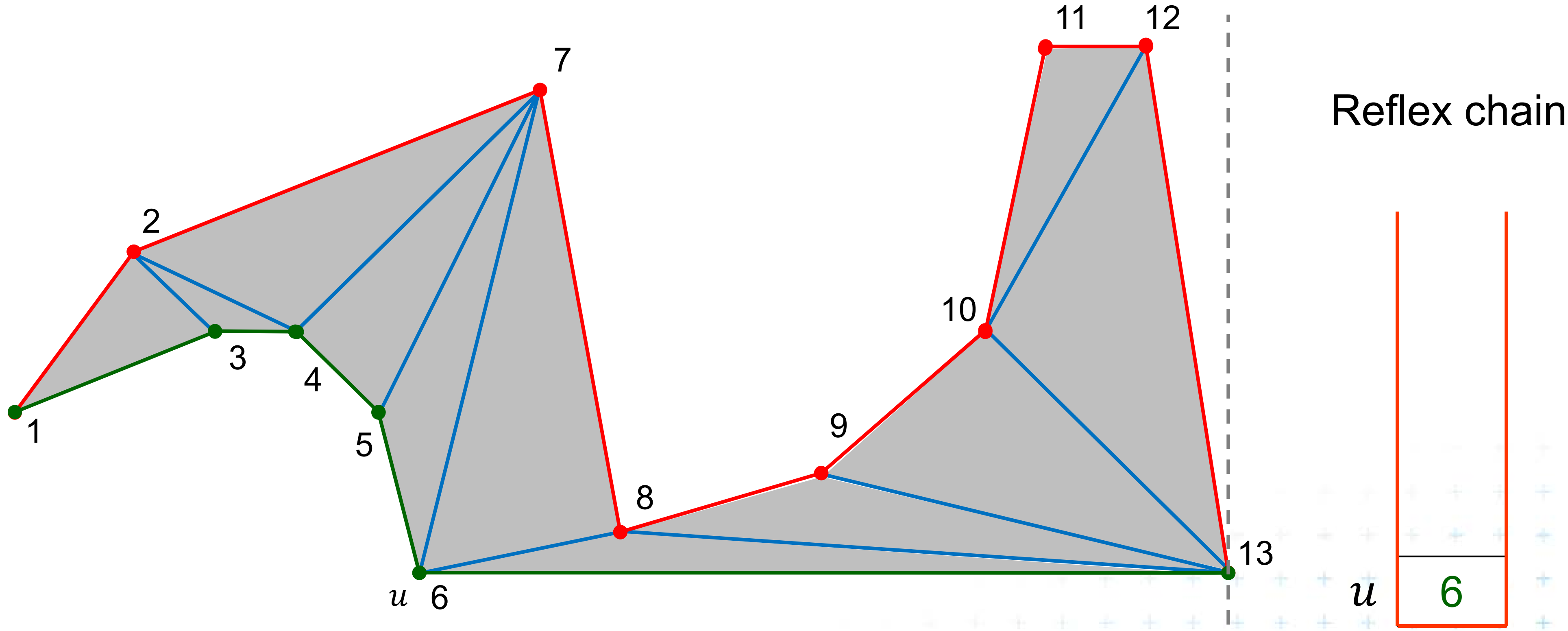


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm

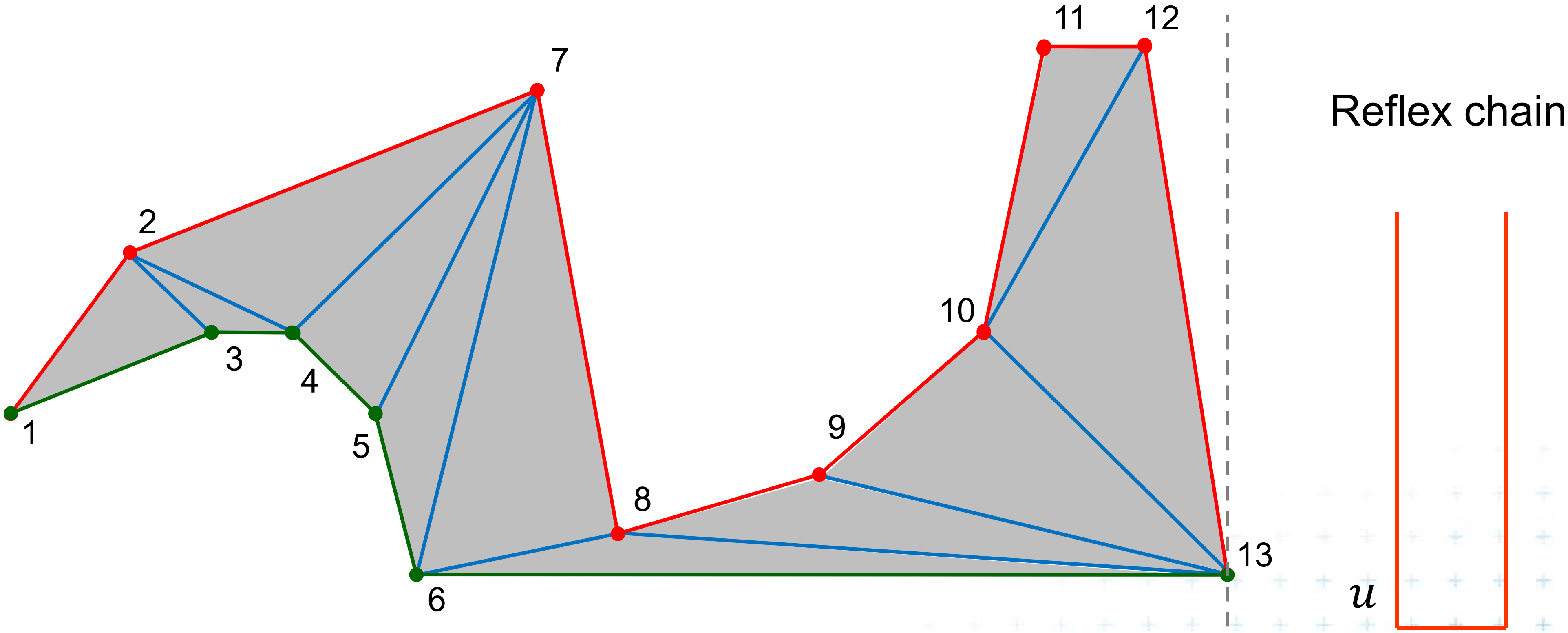


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm

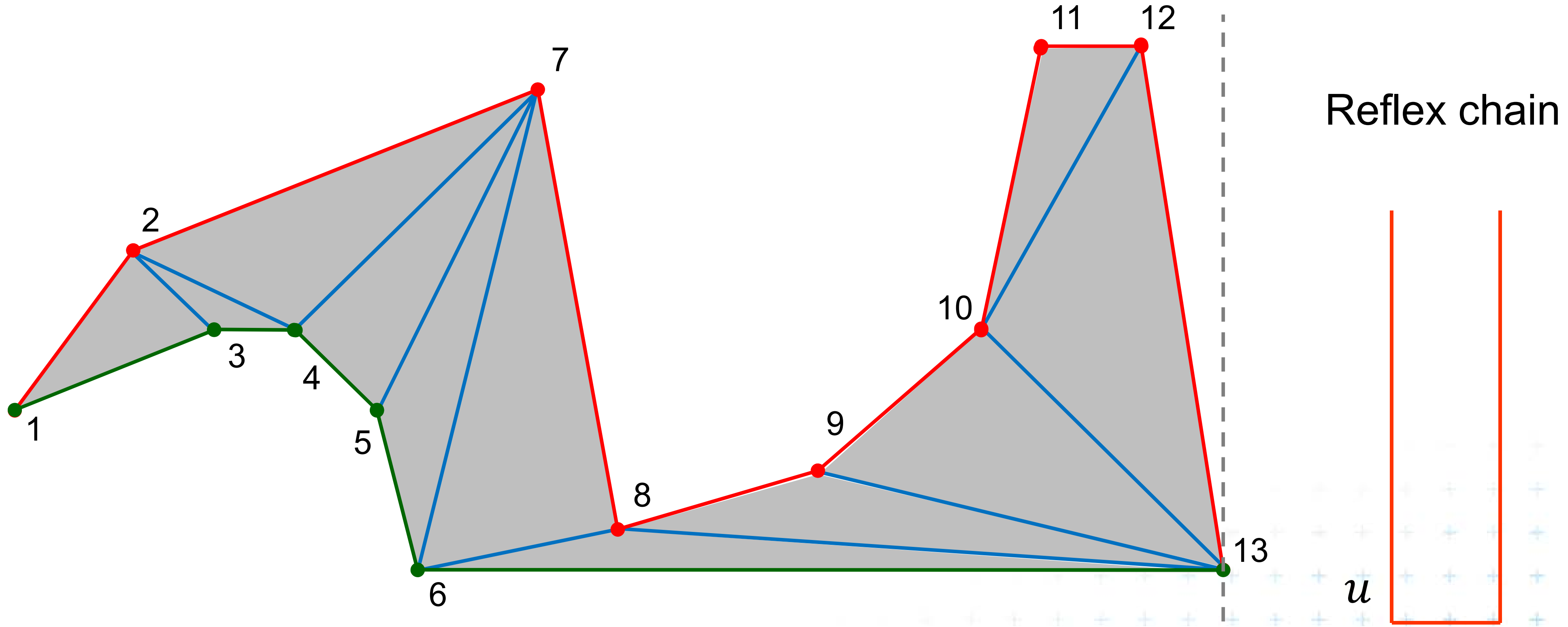


Case 2a – point v_i on the same chain as non-reflex v_{i-1}

Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm



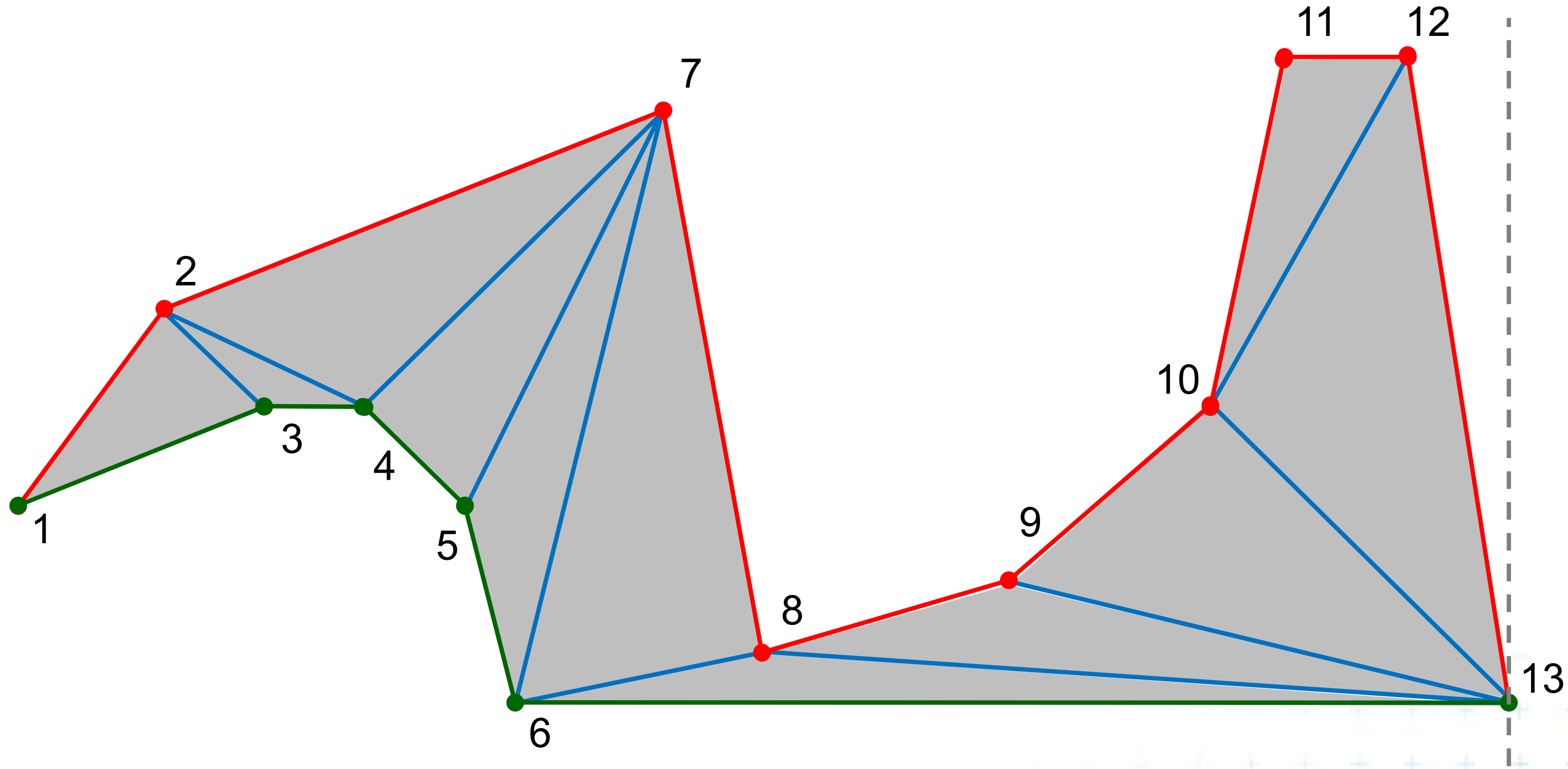
Case 1 – point v_i on opposite chain from v_{i-1}

Would do the same from 13

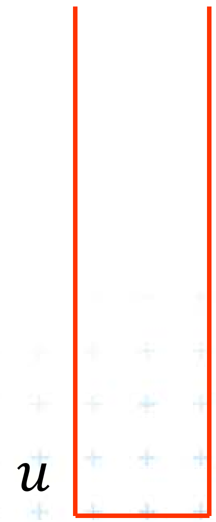
Add diagonal(s) from v_i to all points on reflex chain in stack – pop()

[Mount]

Monotone polygon triangulation algorithm



Reflex chain

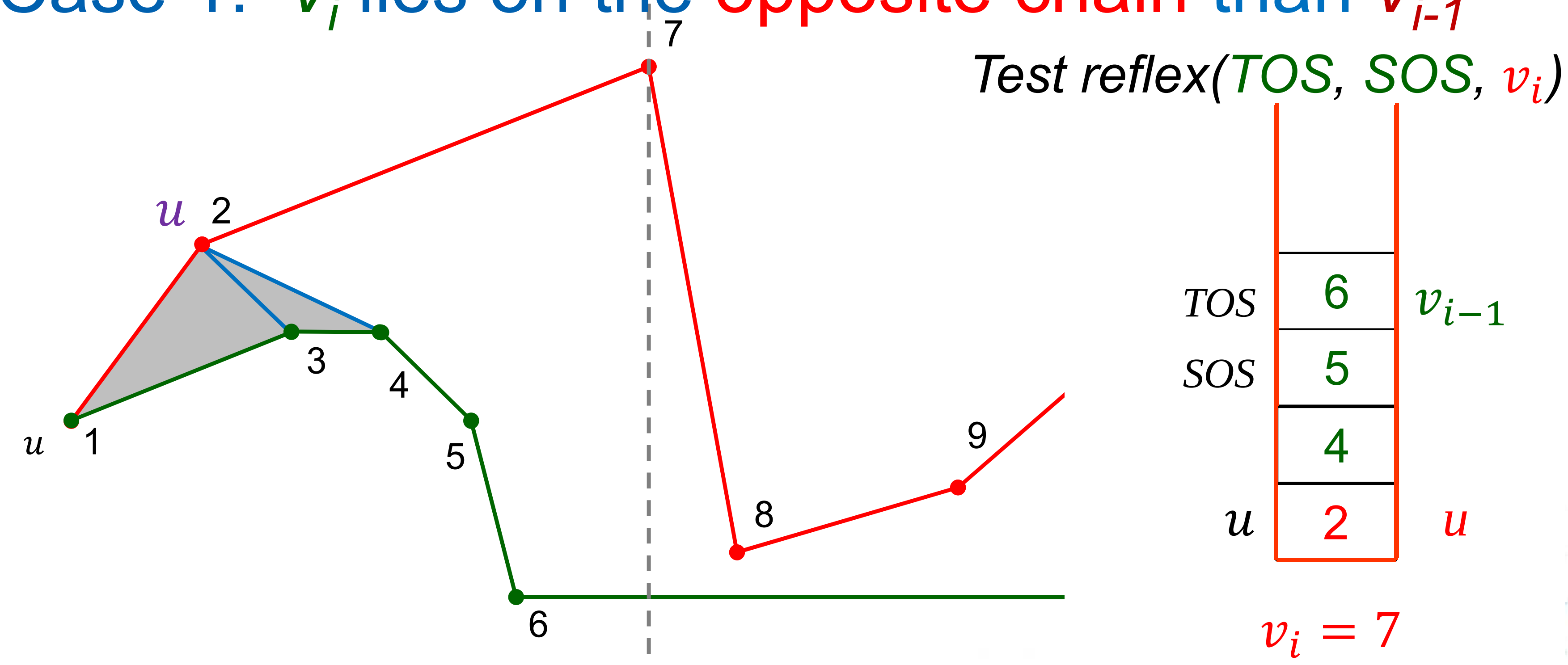


The end

[Mount]

Monotone polygon triangulation algorithm

Case 1: v_i lies on the opposite chain than v_{i-1}

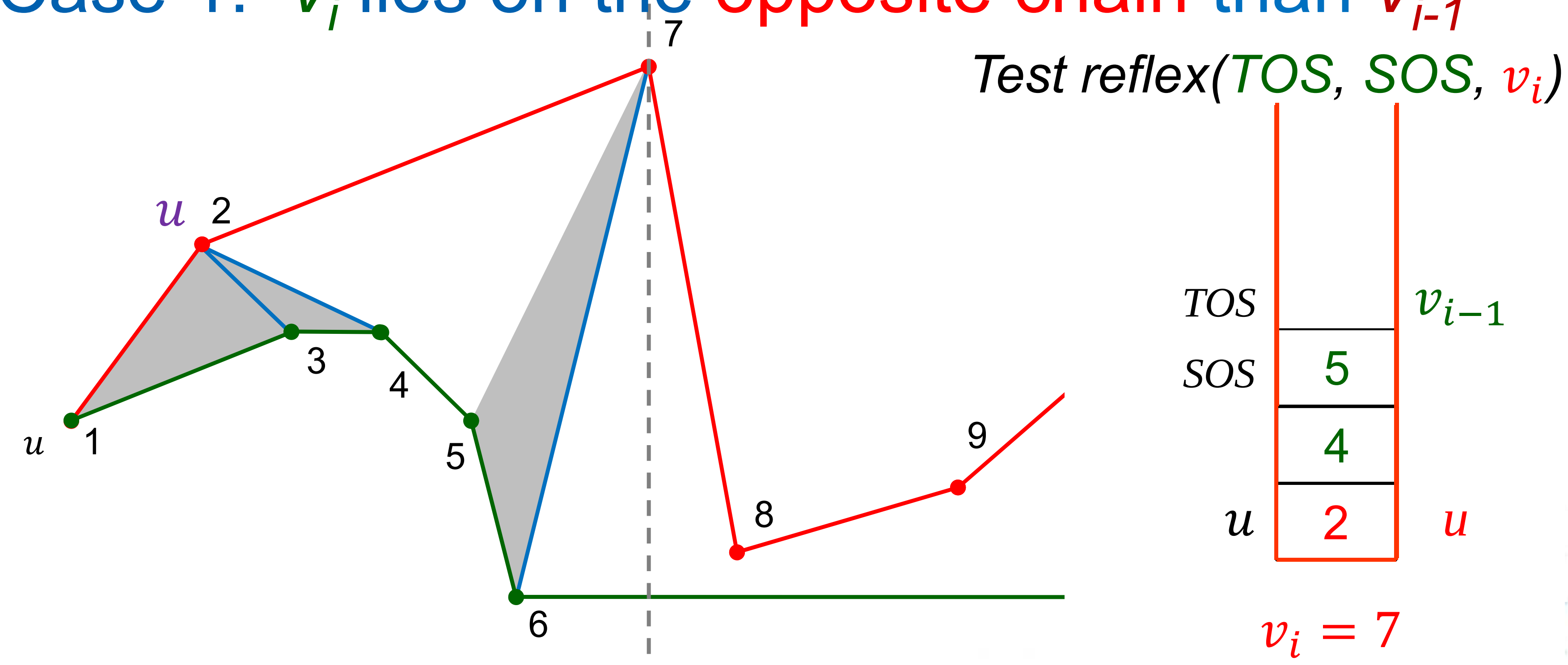


- Left vertex of the last added opposite diagonal is u
- Vertices between u and v_i are waiting in the stack

[Mount]

Monotone polygon triangulation algorithm

Case 1: v_i lies on the opposite chain than v_{i-1}

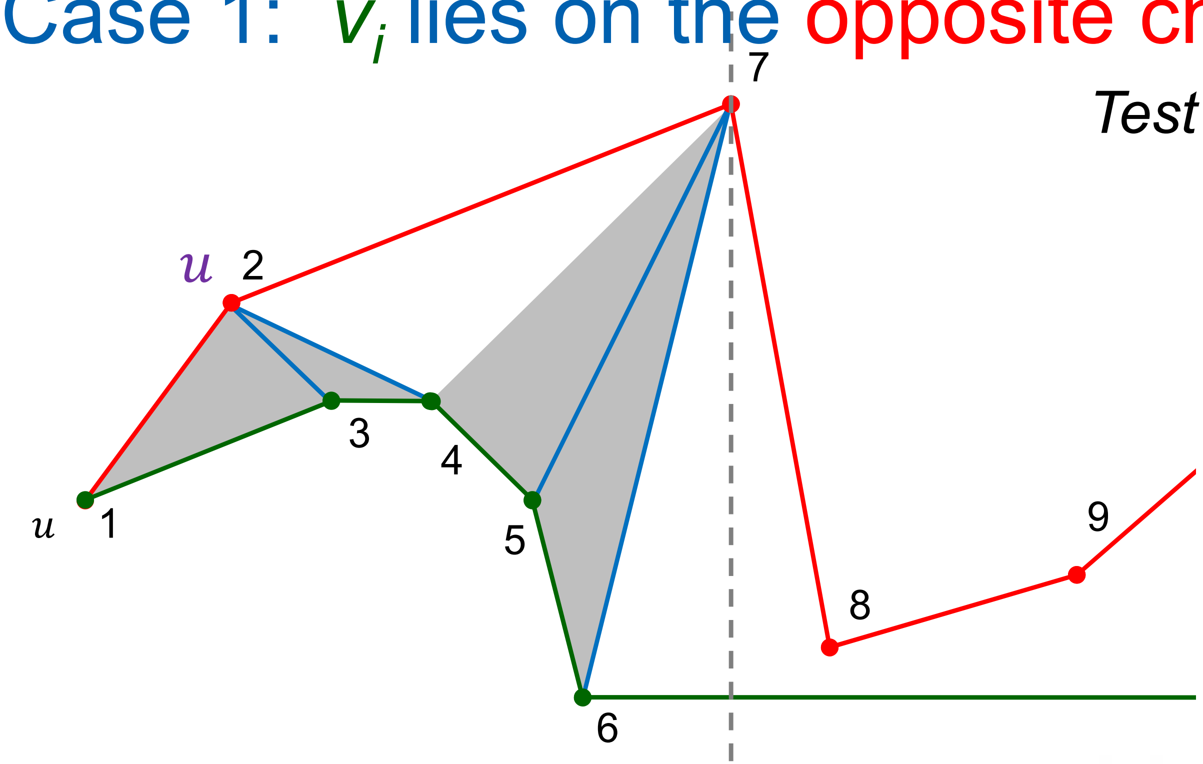


- Left vertex of the last added opposite diagonal is u
- Vertices between u and v_i are waiting in the stack

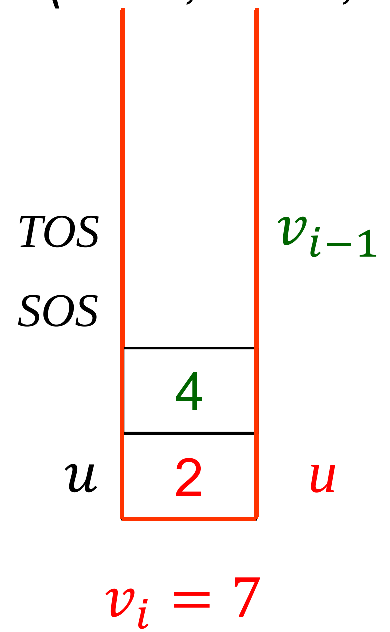
[Mount]

Monotone polygon triangulation algorithm

Case 1: v_i lies on the opposite chain than v_{i-1}



Test reflex(TOS, SOS, v_i)

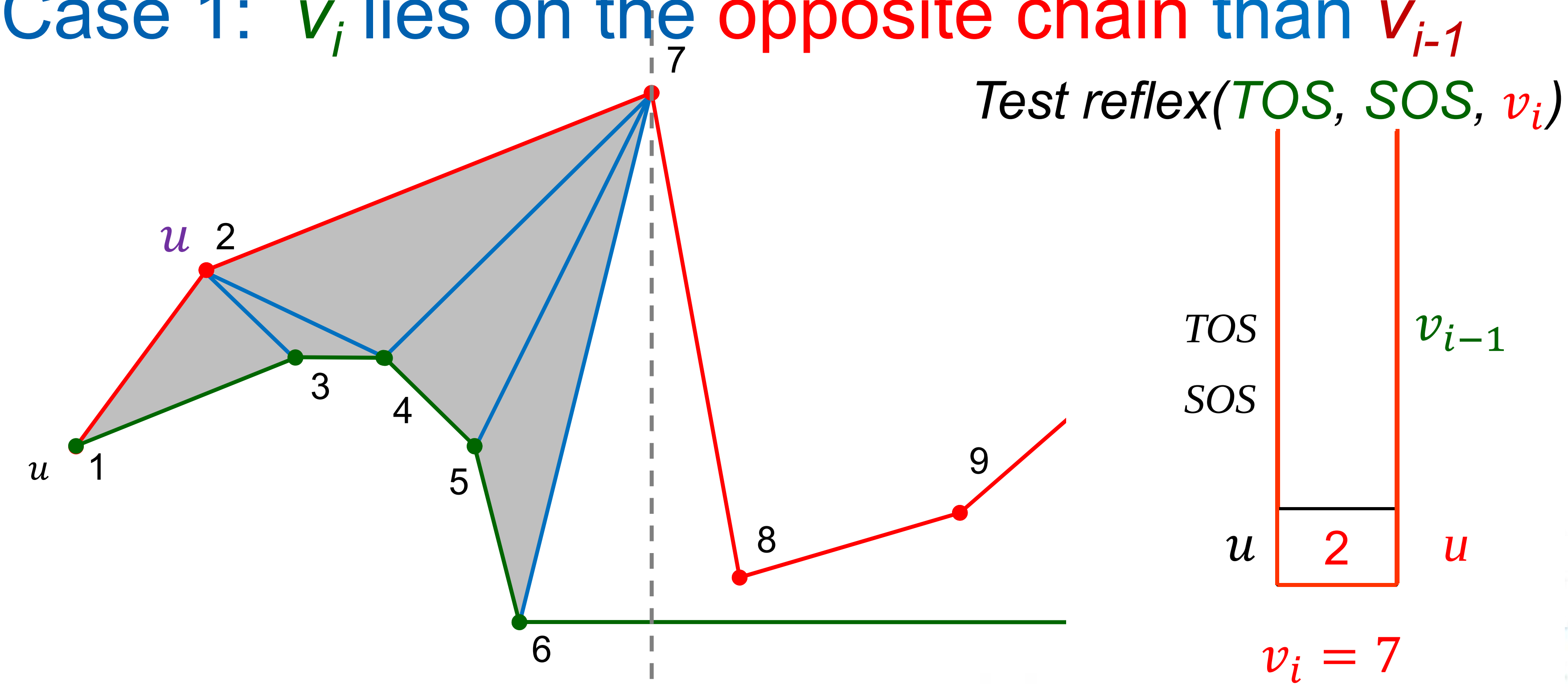


- Left vertex of the last added opposite diagonal is u
- Vertices between u and v_i are waiting in the stack

[Mount]

Monotone polygon triangulation algorithm

Case 1: v_i lies on the opposite chain than v_{i-1}

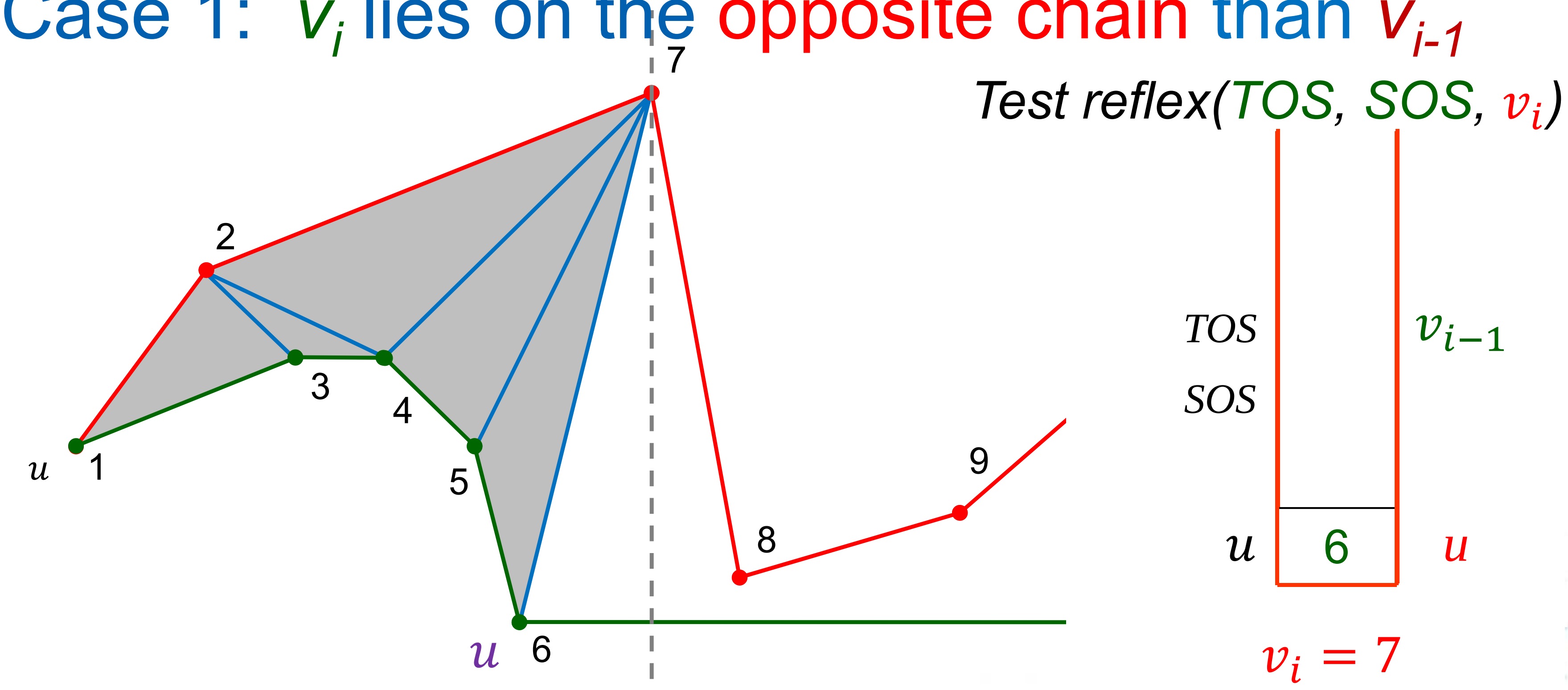


- Left vertex of the last added opposite diagonal is u
- Vertices between u and v_i are waiting in the stack

[Mount]

Monotone polygon triangulation algorithm

Case 1: v_i lies on the opposite chain than v_{i-1}



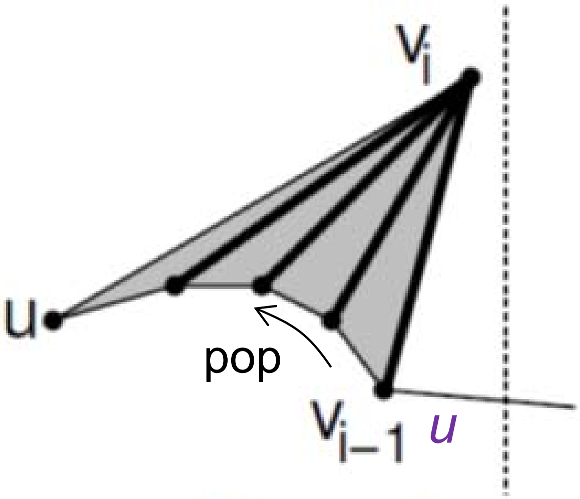
- Left vertex of the last added opposite diagonal is u
- Vertices between u and v_i are waiting in the stack

[Mount]

Triangulation cases for v_i (vertex being just processed)

Case 1: v_i lies on the opposite chain than v_{i-1}

- Add diagonals from next(u) to v_{i-1} (empty the stack-pop)
- Set u = v_{i-1}. Last diagonal (invariant) is $v_{i-1}v_i$
- push u = v_{i-1} and v_i to stack



Case 1

u updated

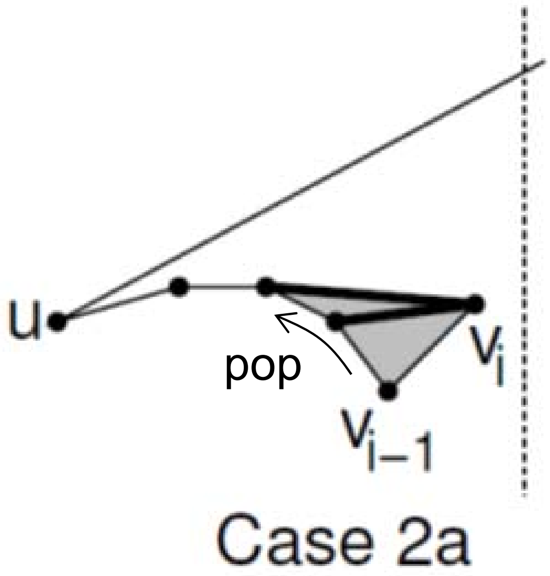
[Mount]

~~*u* unchanged~~

Triangulation cases for v_i (vertex being just processed)

Case 2a: v_i is on the same chain as v_{i-1}

- walk back, adding diagonals joining v_i to prior vertices until the angle becomes > 180° or u is reached – pop
- push v_i to stack



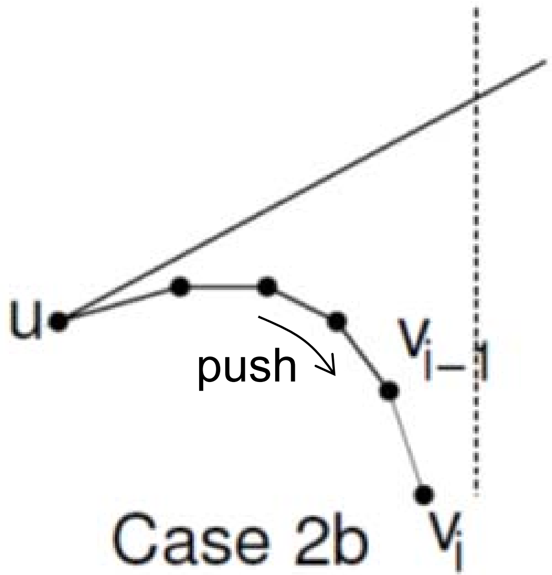
u unchanged

[Mount]

Triangulation cases for v_i (vertex being just processed)

Case 2b: v_i is on the same chain as v_{i-1}

– push v_i to stack



u unchanged

[Mount]

Analysis

Polygon with n vertices has $n - 3$ diagonals

$\Rightarrow O(n)$ total time

Algorithm

sorted list of vertices through merging - $O(n)$

stack operations – max n times $O(1)$ - $O(n)$

orientation test - v_i and top two entries

- $O(1)$ per diagonal - $O(n)$

(add diagonal or push)

Simple polygon triangulation

- Simple polygon can be triangulated in 2 steps:
  1. Partition the polygon into x-monotone pieces
  2. Triangulate all monotone pieces

(we discuss the steps in the reversed order)

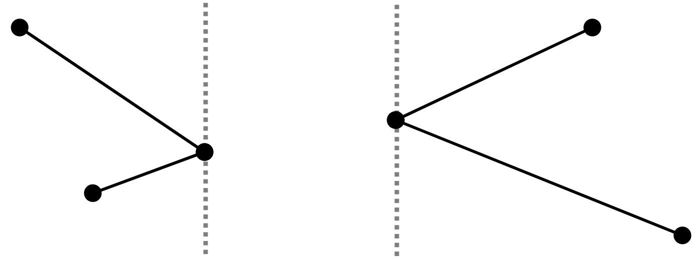
Simple polygon triangulation

- Simple polygon can be triangulated in 2 steps:
  1. Partition the polygon into x-monotone pieces
  2. Triangulate all monotone pieces

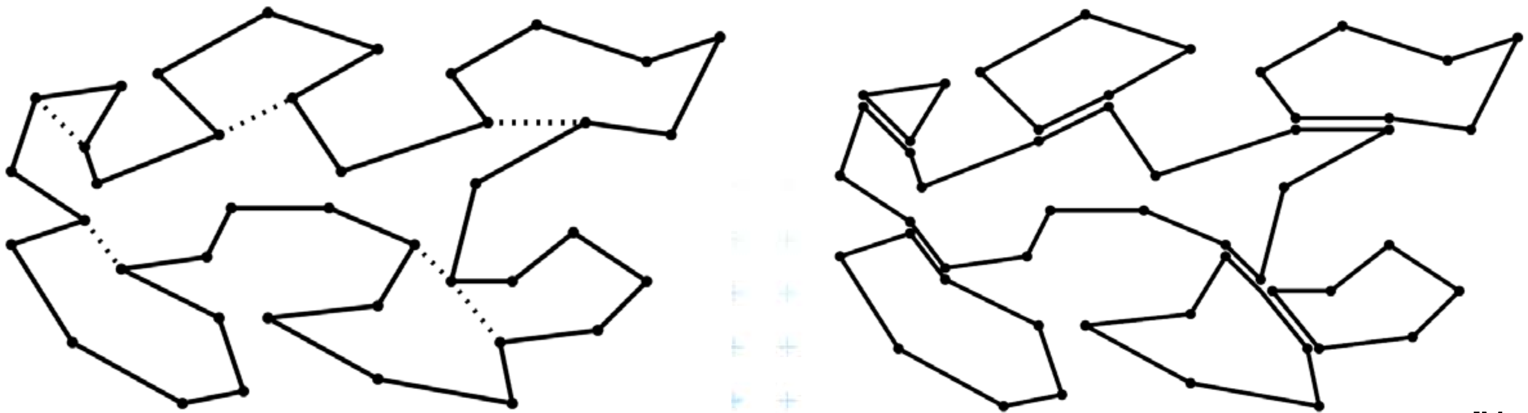
  (we discuss the steps in the reversed order)

1. Polygon subdivision into monotone pieces

- X-monotonicity breaks the polygon in vertices with edges directed both left or both right (inner angle > 180º)



- The monotone polygons parts are separated by the splitting diagonals (joining vertex and helper)



Splitting diagonals

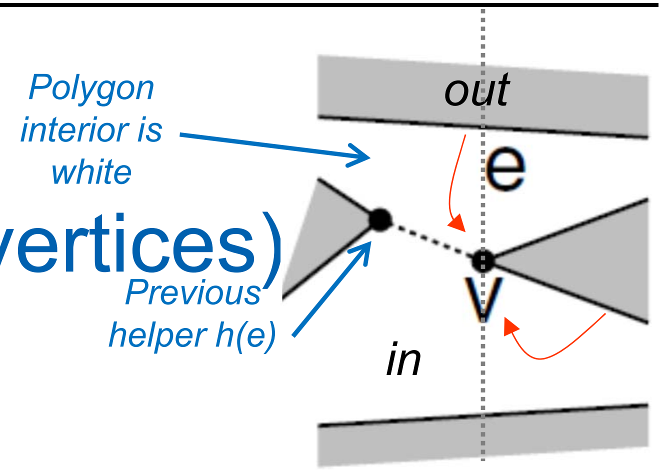
Monotone decomposition

[Mount]

Sweep line algorithm

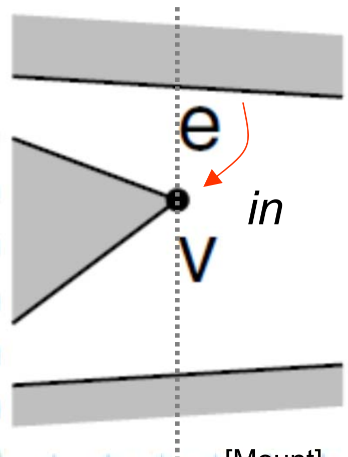
Sweep from left to right

Add diagonals (from split to merge vertices)



In split vertex

- Add diagonal as we reach it



[Mount]

In merge vertex

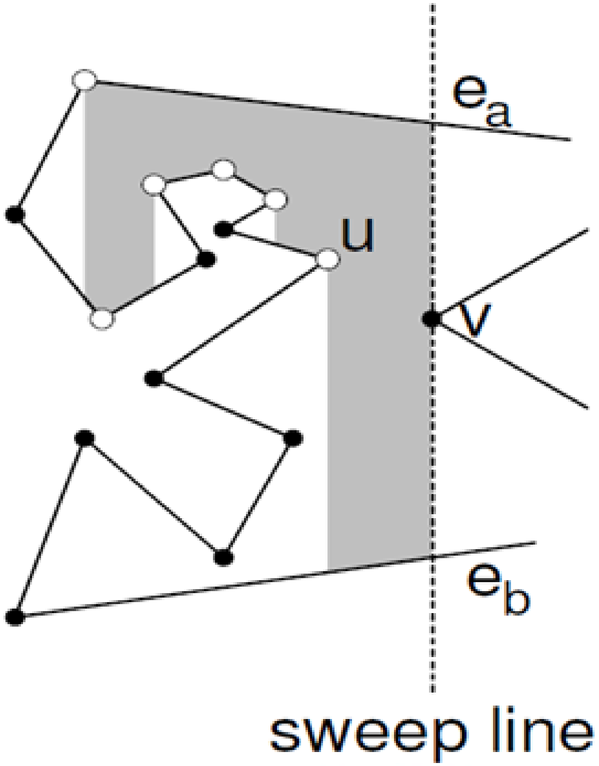
- Take a note about v into helper(e)
- Will be connected later

[Mount]

Data structures for subdivision

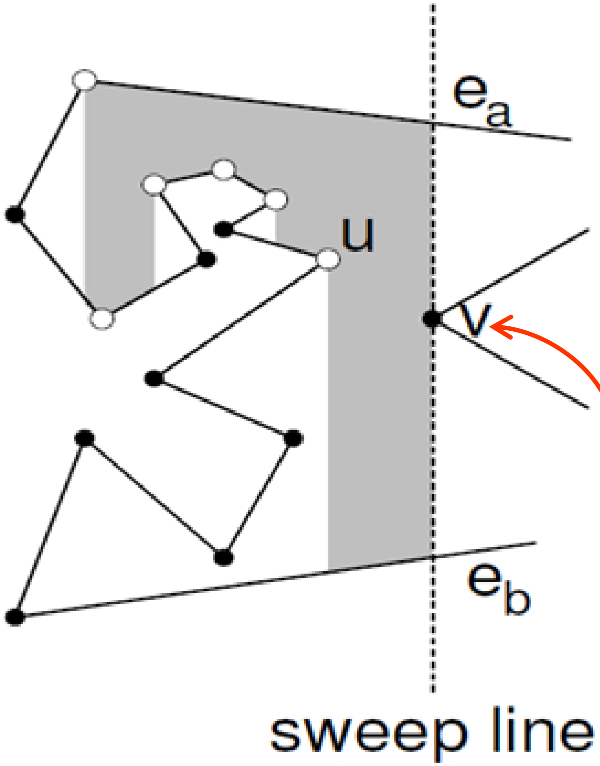
- Events
  - Endpoints of edges, known from the beginning
  - Can be stored in sorted list – no priority queue
- Sweep status
  - List of edges intersecting the sweep line (top to bottom)
  - Stored in O(log n) time dictionary (such as balanced tree)
- Event processing
  - Six event types based on local structure of edges around vertex *v*

Adding a diagonal



[Mount]

Adding a diagonal

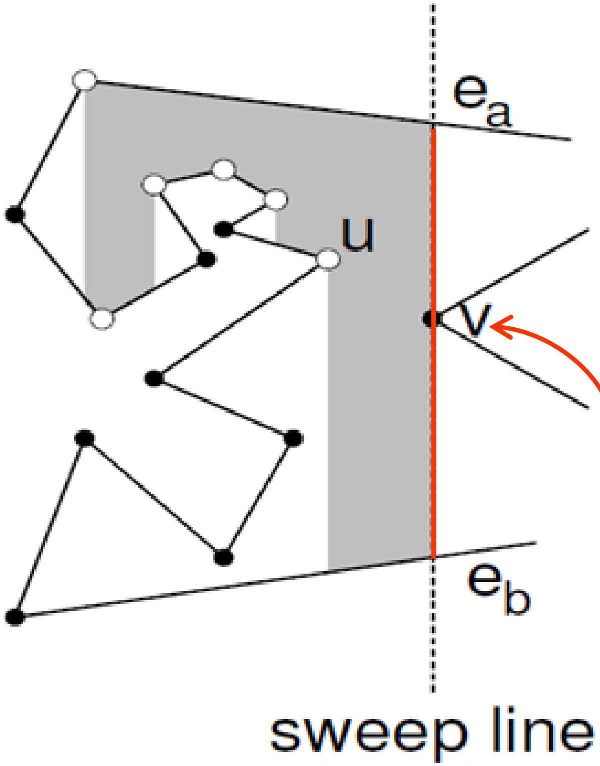


v = current vertex
(sweep line stop)

[Mount]

Adding a diagonal

Find edges e_a & e_b (above and below v) the SL status

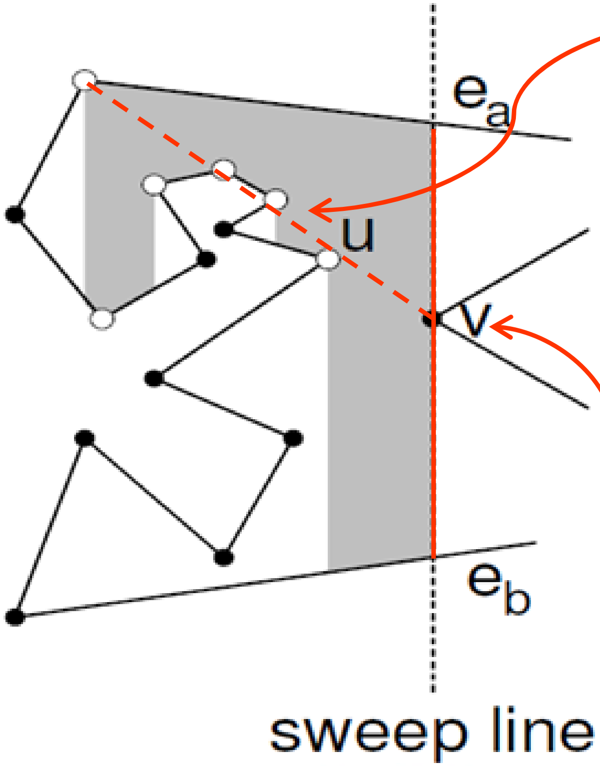


v = current vertex
(sweep line stop)

[Mount]

Adding a diagonal

Find edges e_a & e_b (above and below v) the SL status



Segment from v to left point of e_a is not a diagonal

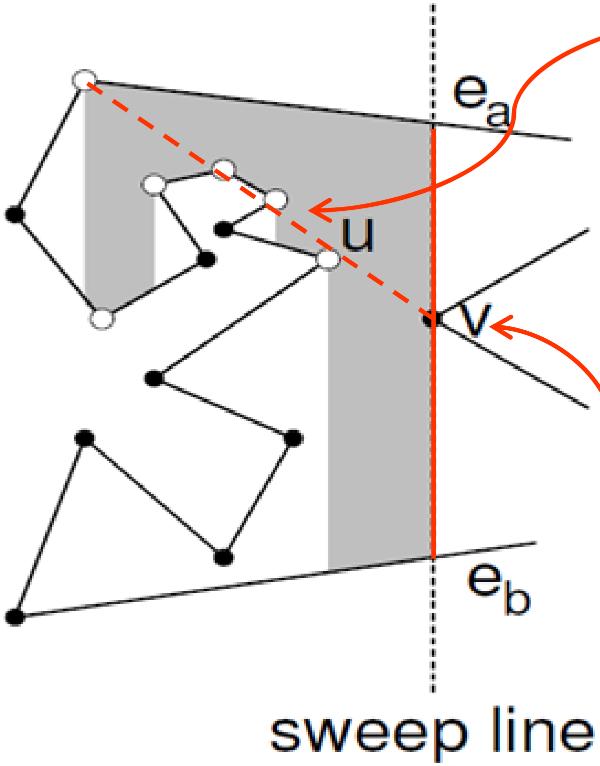
v = current vertex (sweep line stop)

[Mount]

Adding a diagonal

Find edges e_a & e_b (above and below v) the SL status

Use the rightmost visible vertex from edge e_a



Segment from v to left point of e_a is not a diagonal

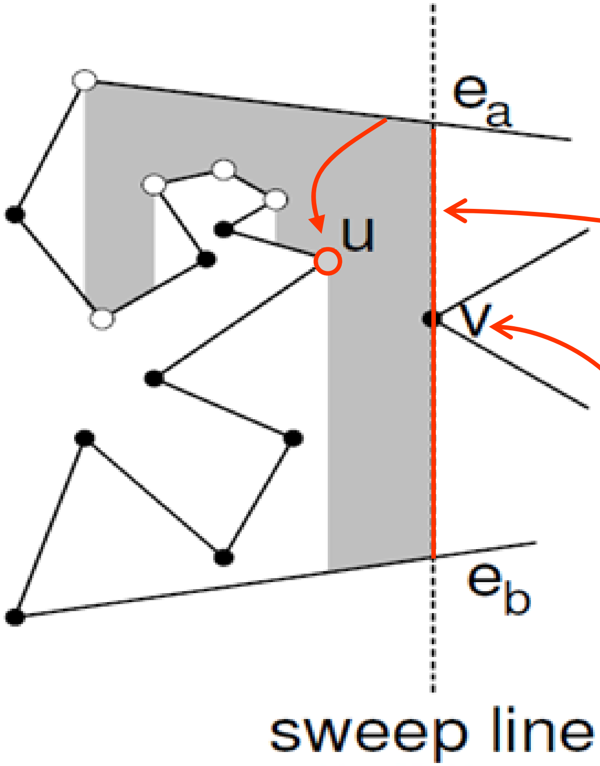
v = current vertex (sweep line stop)

[Mount]

Helper – definition

helper(e_a)
= the rightmost vertically visible processed vertex u - *on or* below edge e_a on polygonal chain between edges e_a & e_b

is visible to every point along the sweep line between e_a & e_b



- vertices visible from e_a
- u = helper(e_a) the rightmost of

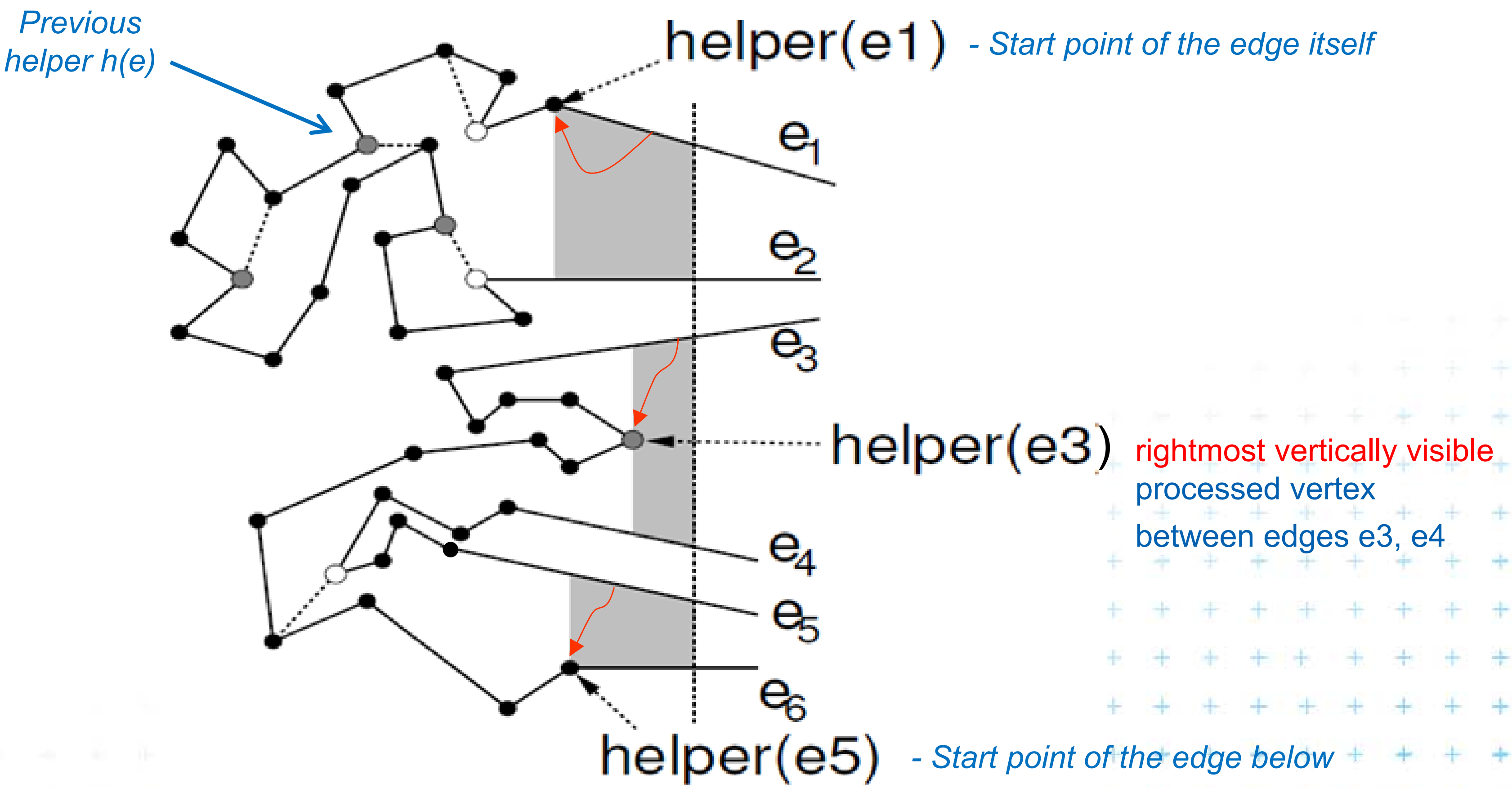
all these vertices see u = helper(e_a)

v = current vertex (sweep line stop)

[Mount]

Helper variants

helper(e_a)
is defined only for edges intersected by the sweep line



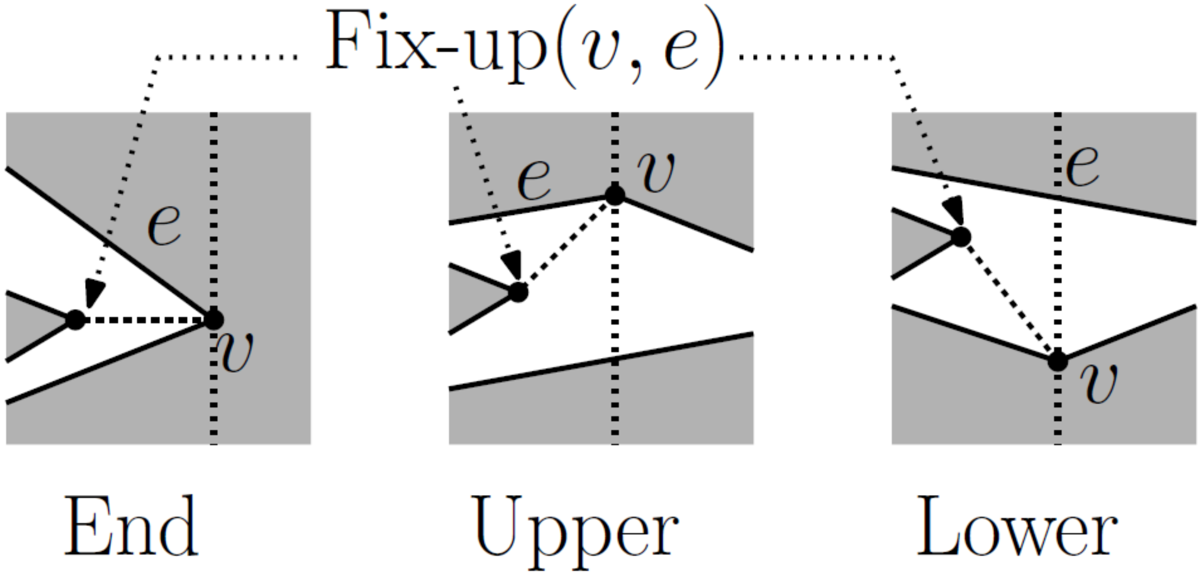
[Mount]

Fix-up function

Fix-up(v, e)

- Gets vertex v and edge e above or incident to v
- if(helper(e) is merge vertex)

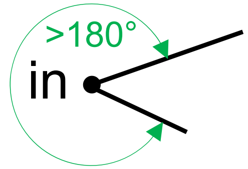
  add diagonal from v to helper(e)

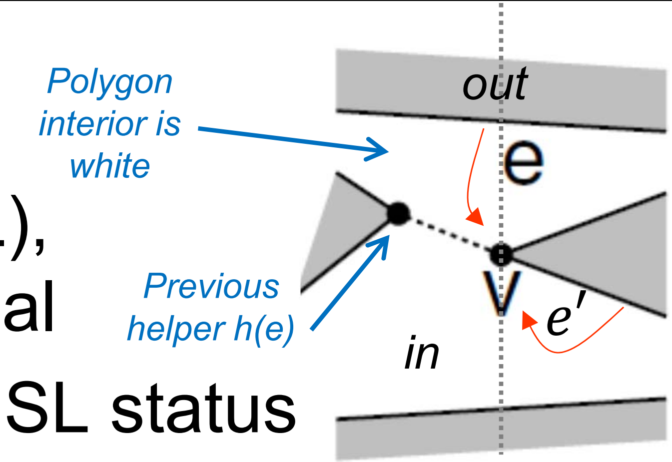


Six event types of vertex v

1. Split vertex

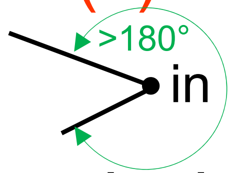
splits the polygon

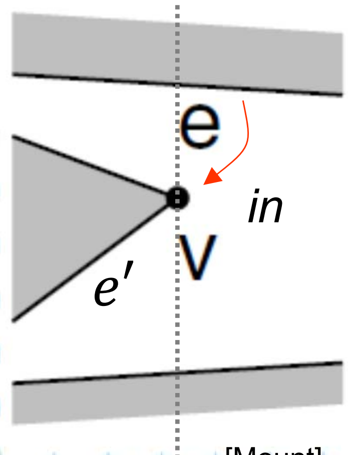




- Find edge e above v (along the SL), connect v with helper(e) by diagonal
- Add 2 new edges starting in *v* into SL status mark lower of them as e'
- Set new helper(e) = helper(e') = v

2. Merge vertex



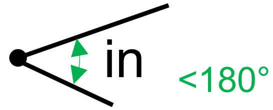


[Mount]

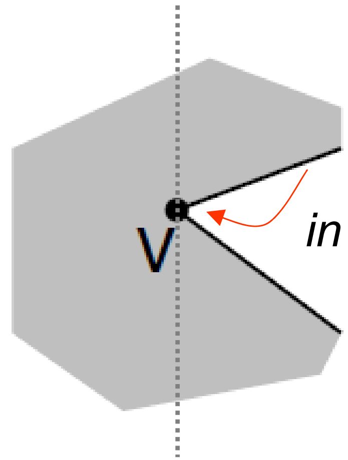
- Find two edges incident with v in SL status
- Delete both from SL status, the lower is e'
- Let e is edge immediately above v
- Make helper(e) = v
- Fix-up(v, e) and Fix-up(v, e')

Six event types of vertex *v*

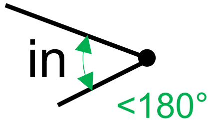
3. Start vertex



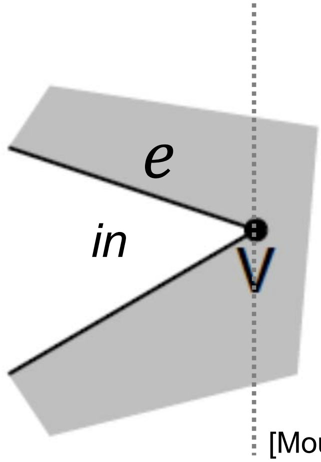
- Both incident edges lie right from v
- But interior angle <180°
- Insert both edges to SL status
- Set helper(upper edge) $= v$



4. End vertex



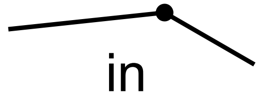
- Both incident edges lie left from v, e is the upper. Fix-up(v, e)
- Delete both edges from SL status
- No helper set – we are out of the polygon

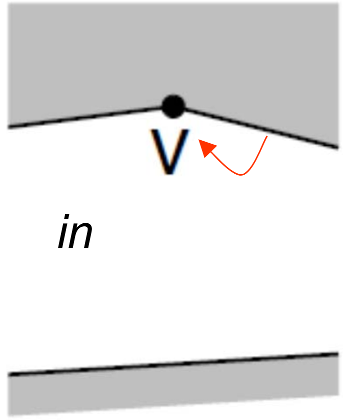


[Mount]

Six event types of vertex *v*

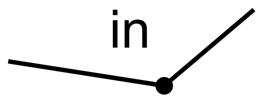
5. Upper chain-vertex

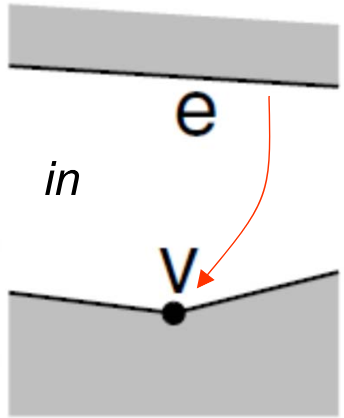




- one side is to the left, one side to the right, interior is below, Fix-up(v, e)
- replace the left edge with the right edge in the SL status
- Make v helper of the new (upper) edge

6. Lower chain-vertex





- one side is to the left, one side to the right, interior is above
- replace the left edge with the right edge in the SL status
- Make v helper of the edge e *above*, Fix-up(v, e)

[Mount]

Polygon subdivision complexity

- Simple polygon with *n* vertices can be partitioned into x-monotone polygons in
  - $O(n \log n)$ time sort
  - $O(n \log n)$ time   (n steps of SL, $\log n$ search each)
  - $O(n)$ storage

- Complete simple polygon triangulation
  - $O(n \log n)$ time for partitioning into monotone polygons
  - $O(n)$ time for triangulation
  - $O(n)$ storage

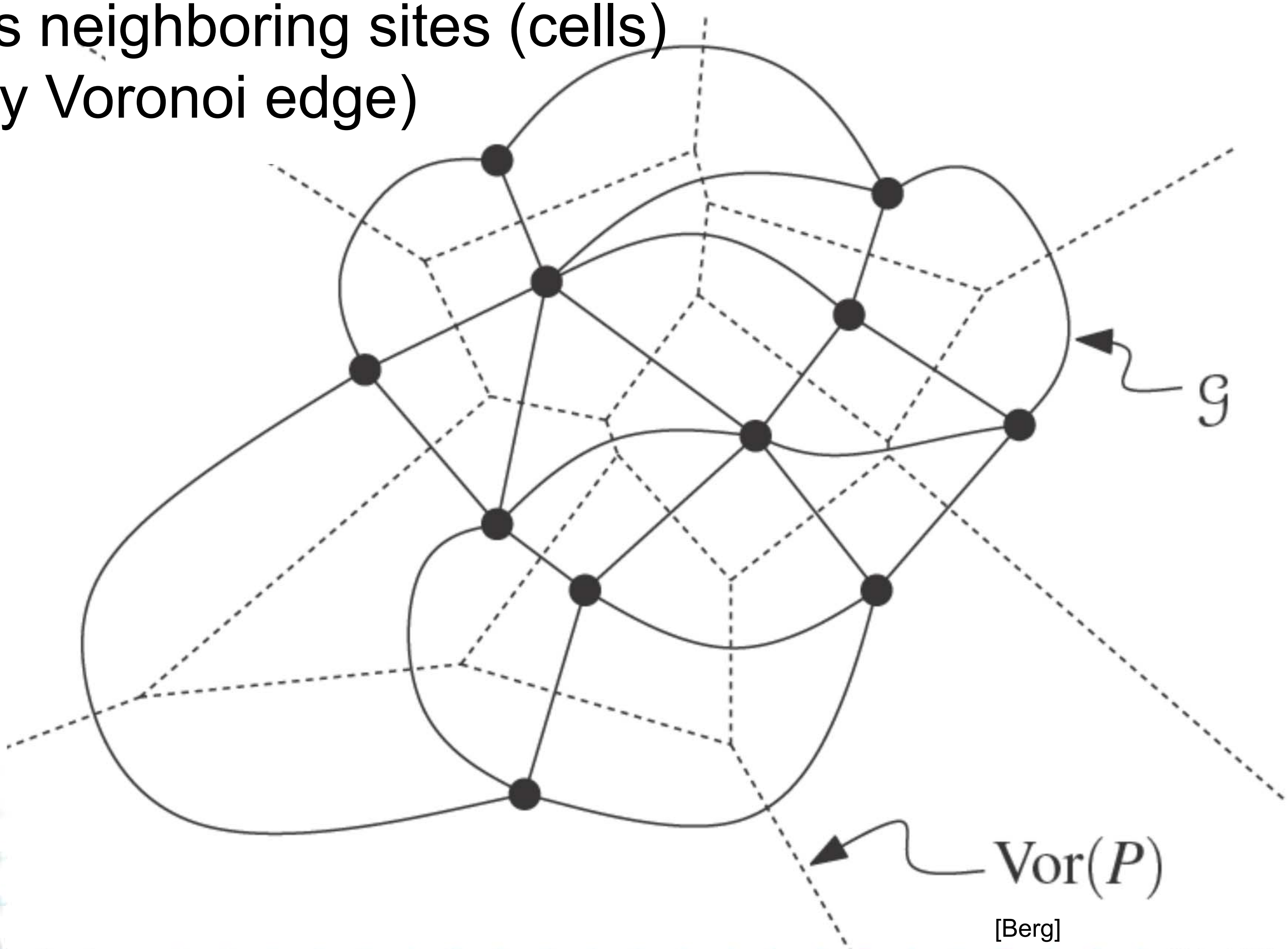
Delone triangulation

(Delaunay - de Launay)

Dual graph G for a Voronoi diagram

Graph G: Node for each Voronoi-diagram site p ~ VD cell $V(p)$

Arc connects neighboring sites (cells)
(arc for every Voronoi edge)



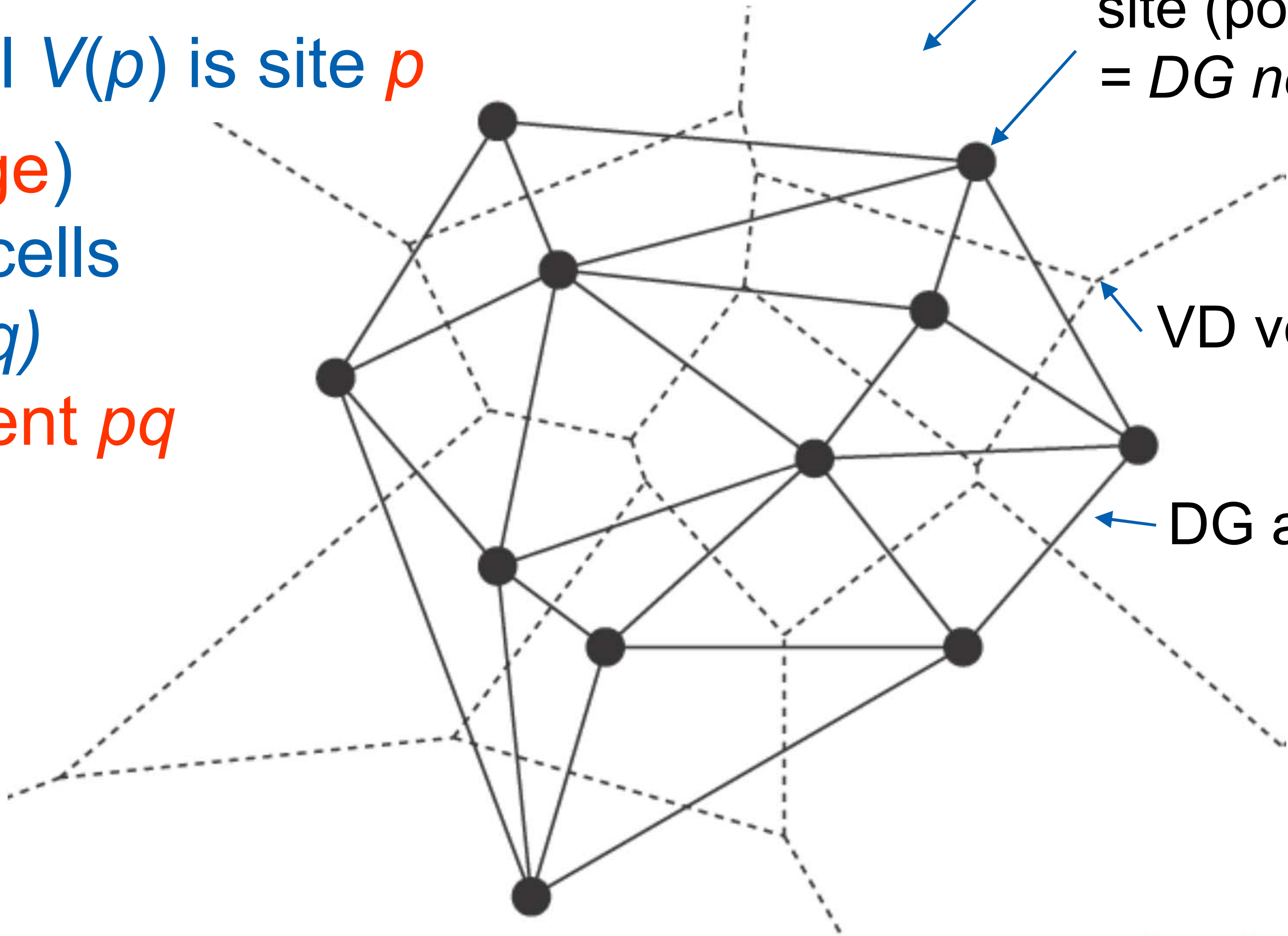
[Berg]

Delone graph *DG*(*P*)

[*Борис Николаевич Делоне*]

= straight line embedding of *G*
(straight-line dual of Voronoi diagram)

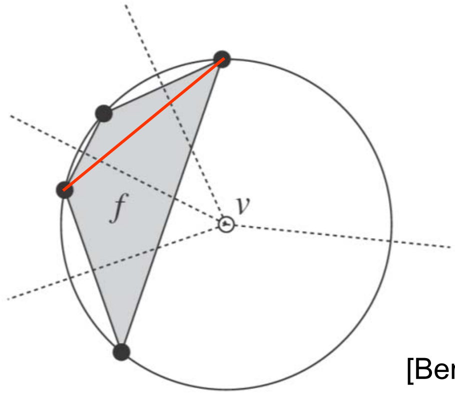
- Node for cell *V*(*p*) is site *p*
- Arc (DG edge) connecting cells *V(p) and V(q)* is the segment *pq*



[Berg]

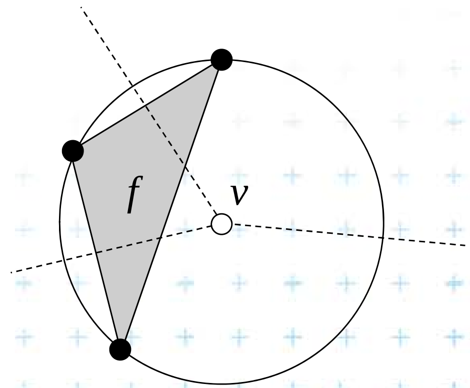
Delaunay graph and Delaunay triangulation

- Delone *graph DG(P)* has convex polygonal faces (with number of vertices ≥3, equal to the degree of Voronoi vertex)
  - Triangulate faces with more vertices *DG(P)* sites not in general position such triangulation is not unique



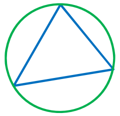
[Berg]

- Delone *triangulation DT*(*P*) = Delone graph for sites in general position
  - No four sites on a circle
  - Faces are triangles (Voronoi vertices have degree = 3)
  - DT is unique



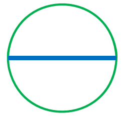
Delone triangulation properties 1/2

Circumcircle property



- The circumcircle of any triangle in DT is empty (no sites)
  Proof: It's center is the Voronoi vertex
- Three points *a,b,c* are vertices of the same face of *DG*(*P*) **iff** circle through *a,b,c* contains no point of *P* in its interior

Empty circle property and legal edge



- Two points *a,b* form an edge of *DG*(*P*) – it is a legal edge **iff** ∃ closed disc with *a,b* on its boundary that contains no other point of *P* in its interior … disc minimal diameter = dist(a,b)

Closest pair property

- The closest pair of points in *P* are neighbors in *DT*(*P*)

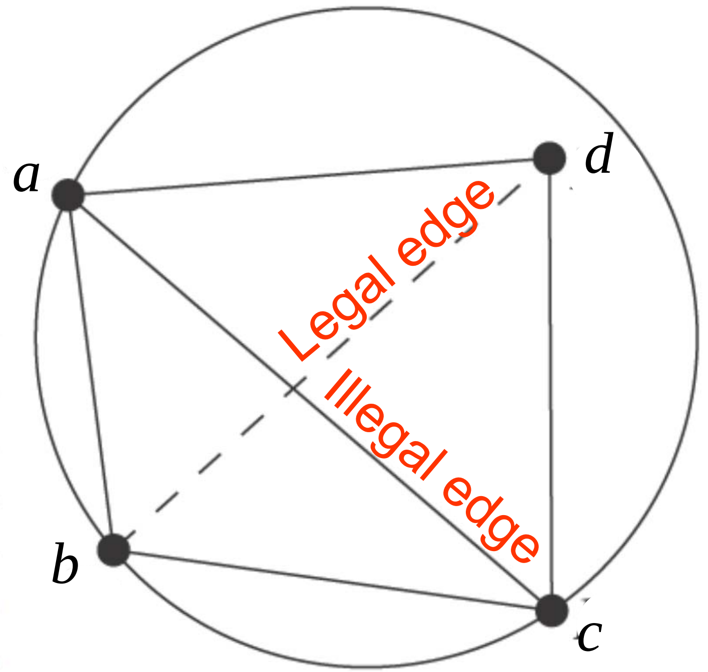
Delone triangulation properties 2/2

- DT edges do not intersect
- Triangulation *T* is legal, **iff** *T* is a Delone triangulation (i.e., if it does not contain illegal edges)
- Edge in DT that was legal before may become illegal if one of the triangles incident to it changes

Non-convex quad has only one diagonal

- In convex quadrilateral *abcd* (*abcd* do not lie on common circle) exactly one of *ac, bd* is an illegal edge and the other edge is legal

= principle of edge flip operation

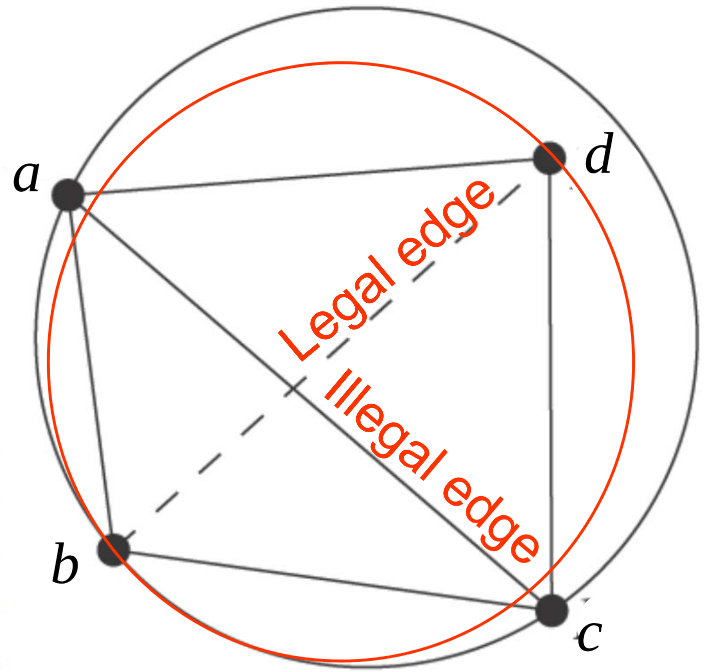


[Berg]

Delone triangulation properties 2/2

- DT edges do not intersect
- Triangulation *T* is legal, **iff** *T* is a Delone triangulation (i.e., if it does not contain illegal edges)
- Edge in DT that was legal before may become illegal if one of the triangles incident to it changes
- In convex quadrilateral *abcd* (*abcd* do not lie on common circle) exactly one of *ac, bd* is an illegal edge and the other edge is legal = principle of edge flip operation

Non-convex quad has only one diagonal



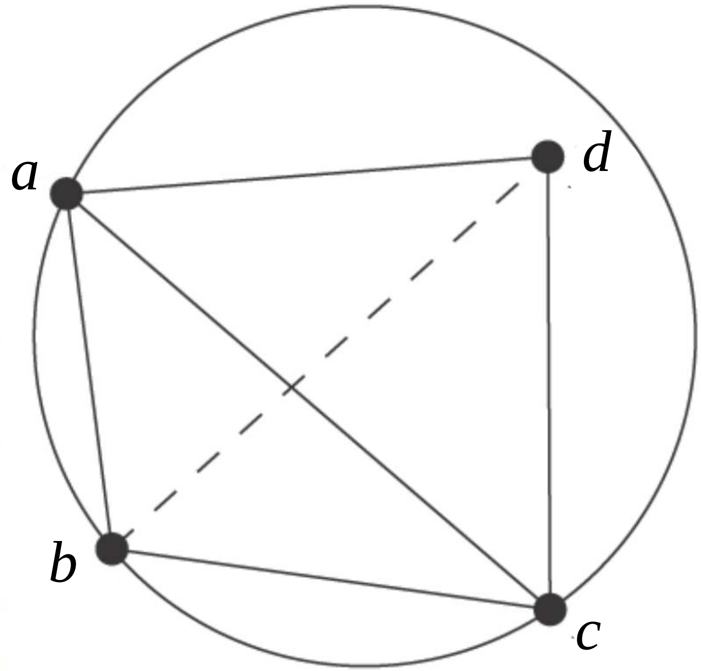
[Berg]

Edge flip operation

Edge flip   flips illegal edge → legal edge

= a local operation, that increases the angle vector

- Given two adjacent triangles $\triangle abc$ and $\triangle cda$ such that their union forms a convex quadrilateral, the edge flip operation replaces the diagonal ac with bd.



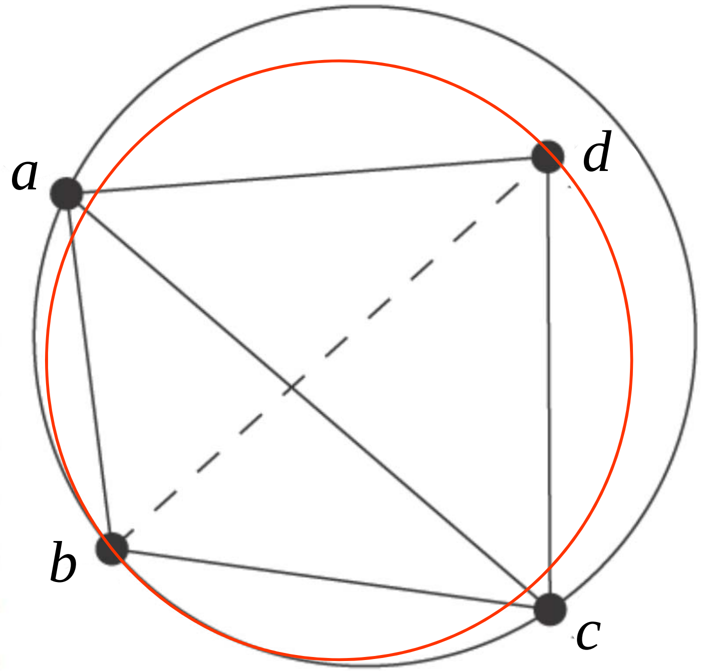
[Berg]

Edge flip operation

Edge flip flips illegal edge → legal edge

= a local operation, that increases the angle vector

- Given two adjacent triangles $\triangle abc$ and $\triangle cda$ such that their union forms a convex quadrilateral, the edge flip operation replaces the diagonal *ac* with *bd*.



[Berg]

Delone triangulation

- Let *T* be a triangulation with *m* triangles (and 3*m angles*)
- Angle-vector
  = non-decreasing ordered sequence $(\alpha_1, \alpha_2, \ldots, \alpha_{3m})$ inner angles of triangles, $\alpha_i \leq \alpha_j$, for $i < j$
- In the plane, Delaunay triangulation has the lexicographically largest angle sequence
  - It maximizes the minimal angle (the first angle in angle-vector)
  - It maximizes the second minimal angle, …
  - It maximizes all angles
  - It is an angle sequence optimal triangulation

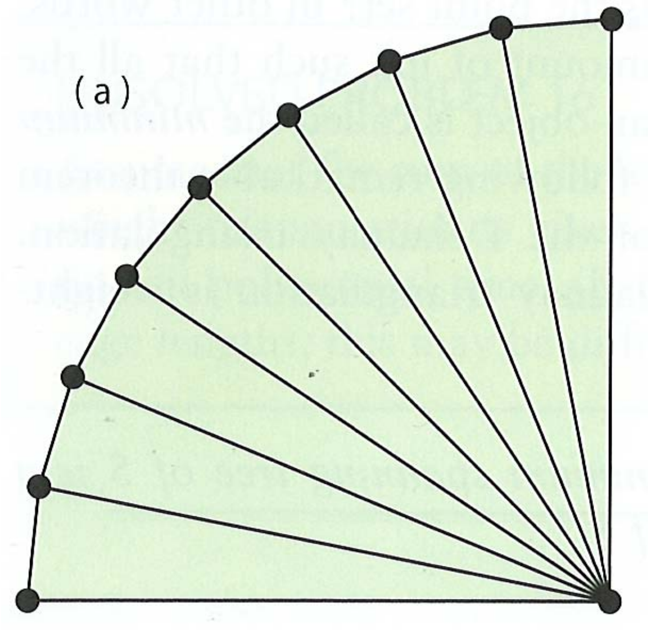
Delone triangulation

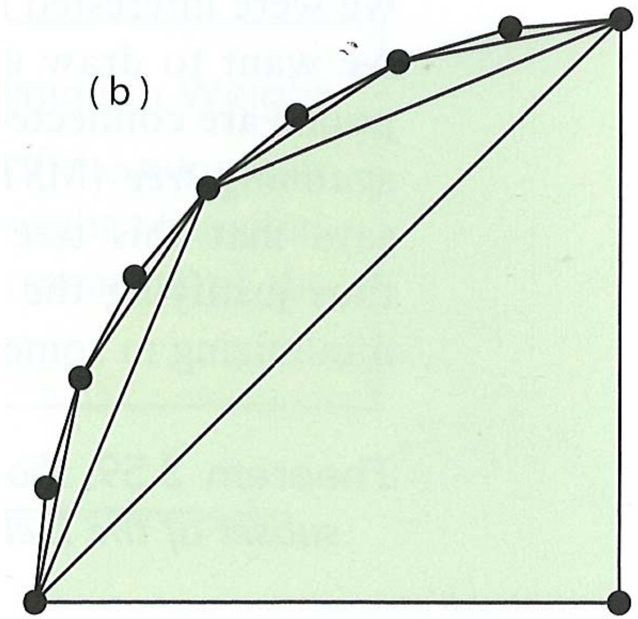
- It maximizes the minimal angle
  - The smallest angle in the DT is at least as large as the smallest angle in any other triangulation.
- Minimum spanning tree is a subset of DT min. kostra
- However, the Delaunay triangulation
  - does not necessarily minimize the maximum angle.
  - does not necessarily minimize the length of the edges.

DT and minimal weight triangulation

32 points on unit circle + center, $\frac{1}{4}$ shown

Connect every 2, 4, 8 + center





Delone triangulation

Minimum weight triangulation
(minimum sum of edge lengths)

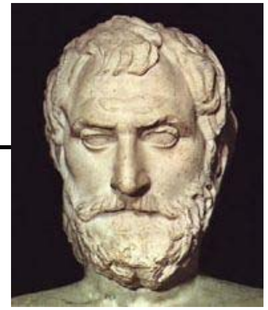
Total weight close to $2\pi + 32$

Total weight far less than $8\pi + 4 > 4x$ around

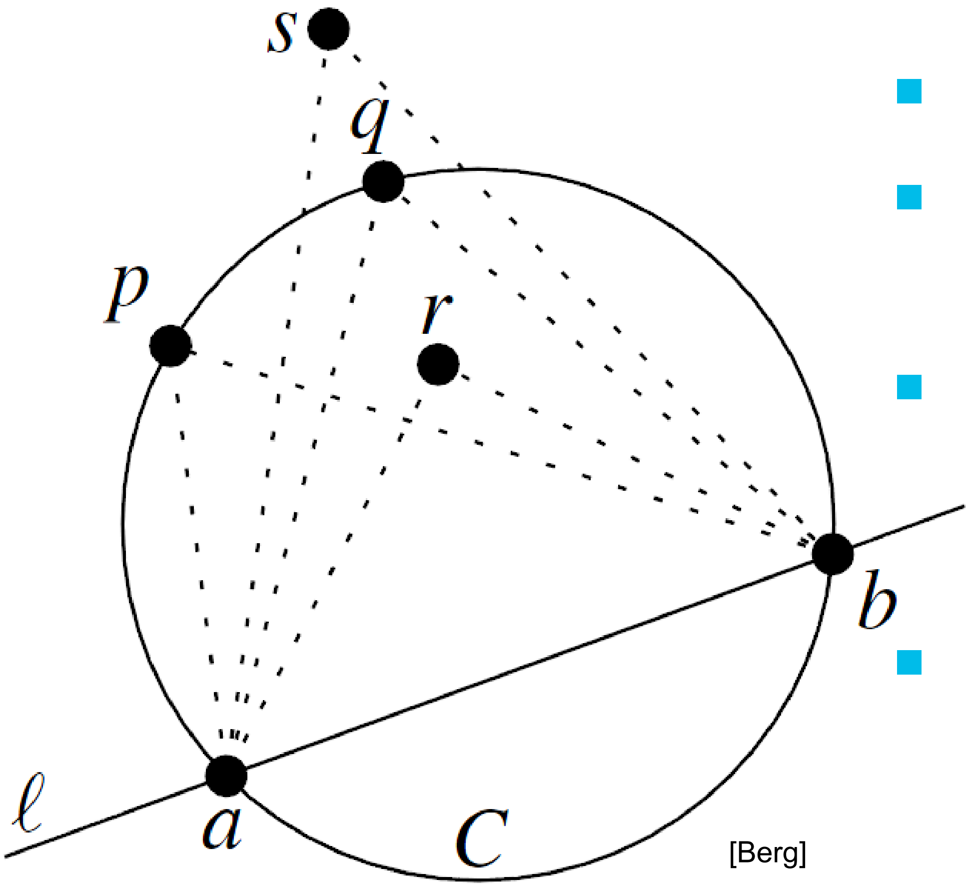
$$2\pi + 32 \approx 38 \quad > \quad 29 \approx 8\pi + 4$$

[Devadoss]

Thales's theorem (624-546 BC)



Respective Central Angle Theorem



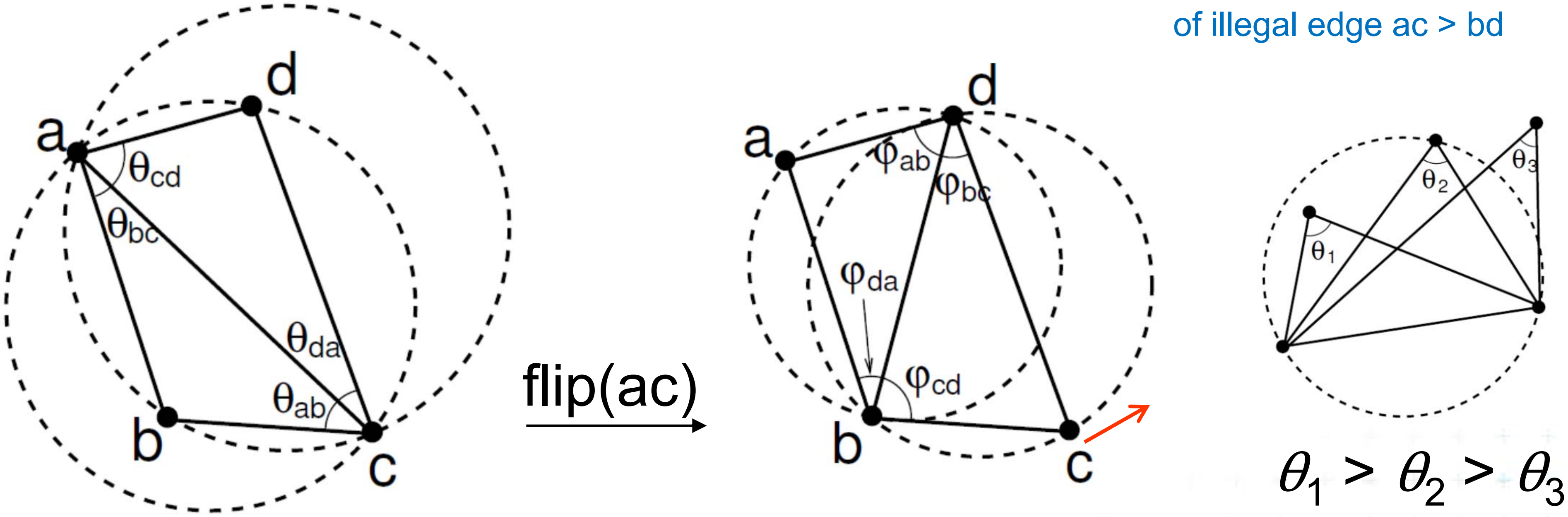
- Let C = circle,
- l =line intersecting C in points a, b
- p, q, r, s = points on the same side of l
  p,q on C , r is *in*, s is out
- Then for the angles holds:
  $\sphericalangle arb > \sphericalangle apb = \sphericalangle aqb > \sphericalangle asb$

http://www.mathopenref.com/arccentralangletheorem.html

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\varphi_{ab} > \theta_{ab}$ $\varphi_{bc} > \theta_{bc}$ $\varphi_{cd} > \theta_{cd}$ $\varphi_{da} > \theta_{da}$

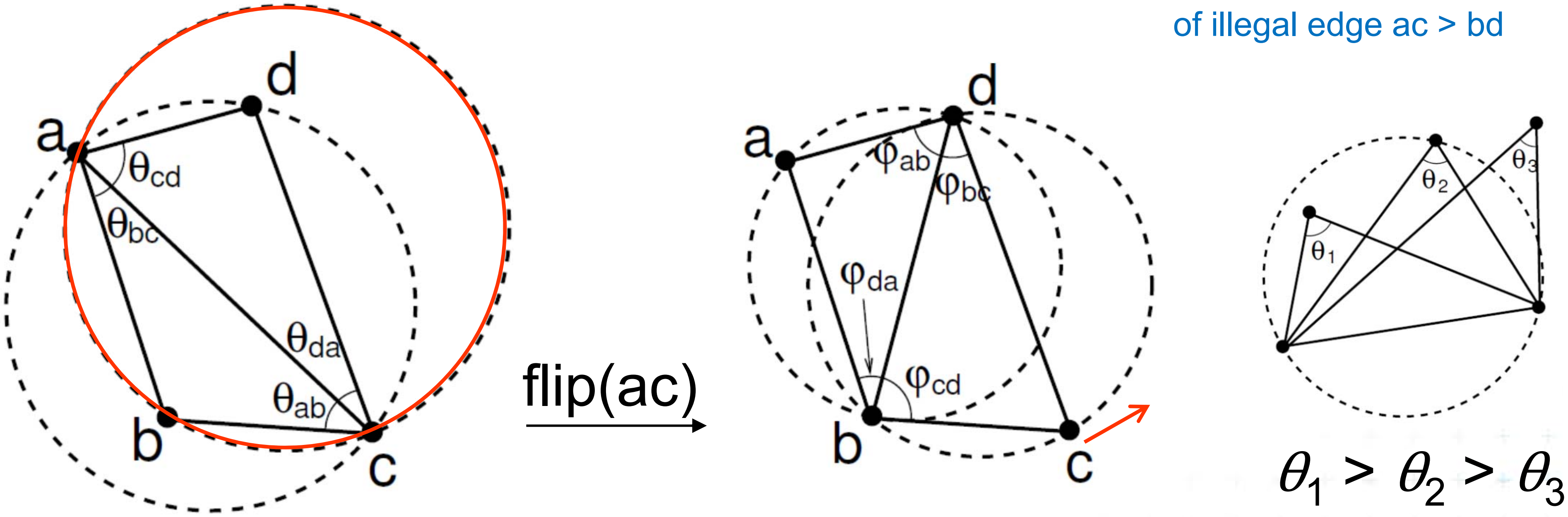
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\varphi_{ab} > \theta_{ab}$ $\varphi_{bc} > \theta_{bc}$ $\varphi_{cd} > \theta_{cd}$ $\varphi_{da} > \theta_{da}$

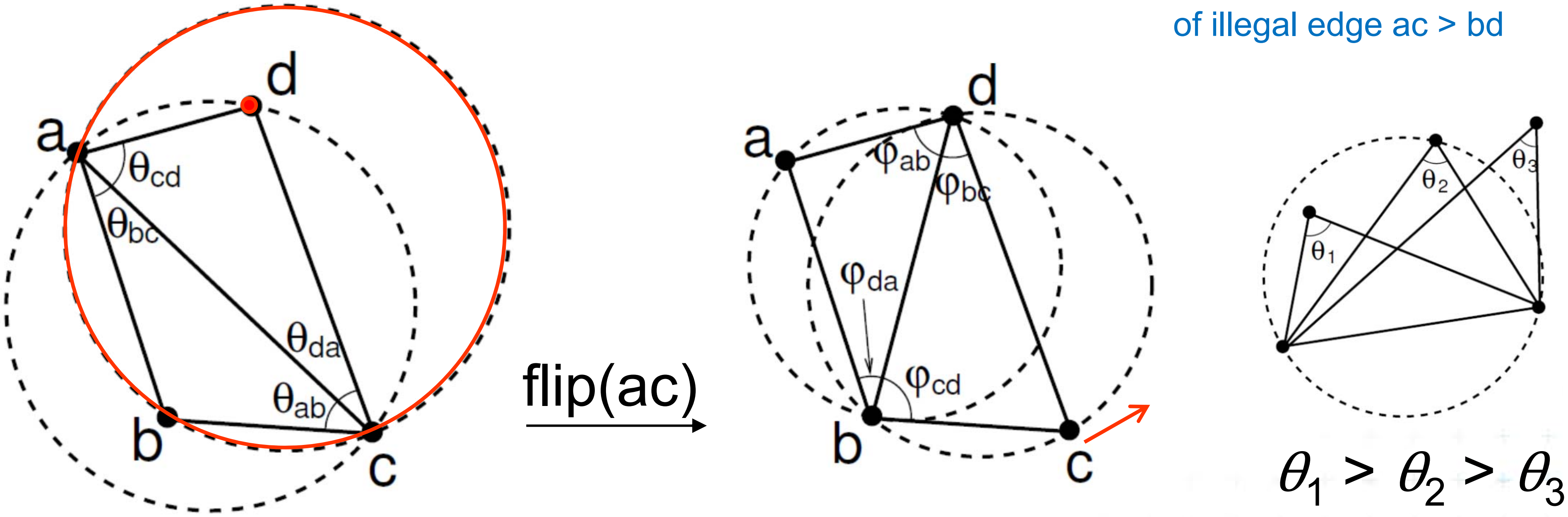
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\varphi_{ab} > \theta_{ab}$ $\varphi_{bc} > \theta_{bc}$ $\varphi_{cd} > \theta_{cd}$ $\varphi_{da} > \theta_{da}$

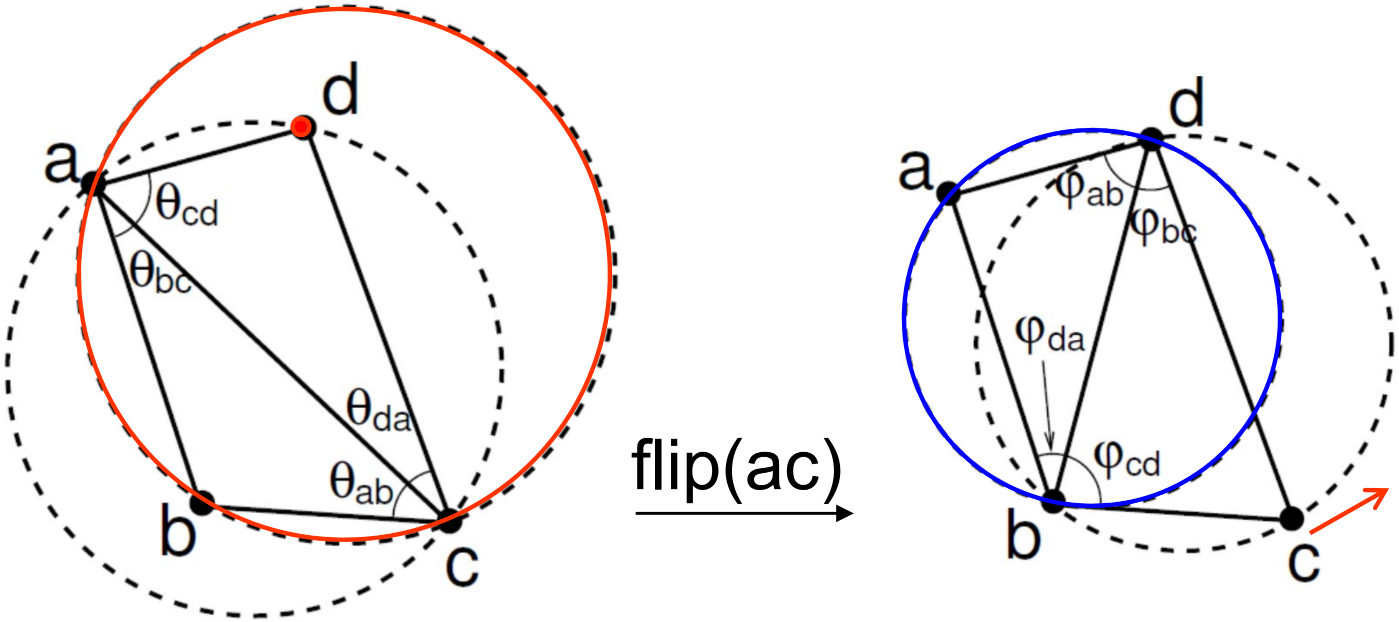
=> After limited number of edge flips

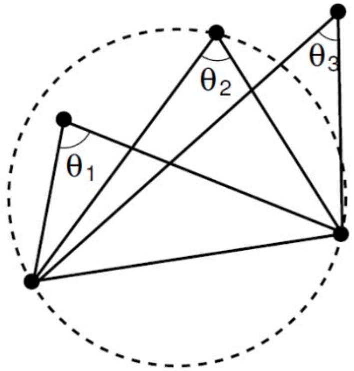
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\quad \varphi_{ab} > \theta_{ab} \quad \varphi_{bc} > \theta_{bc} \quad \varphi_{cd} > \theta_{cd} \quad \varphi_{da} > \theta_{da}$

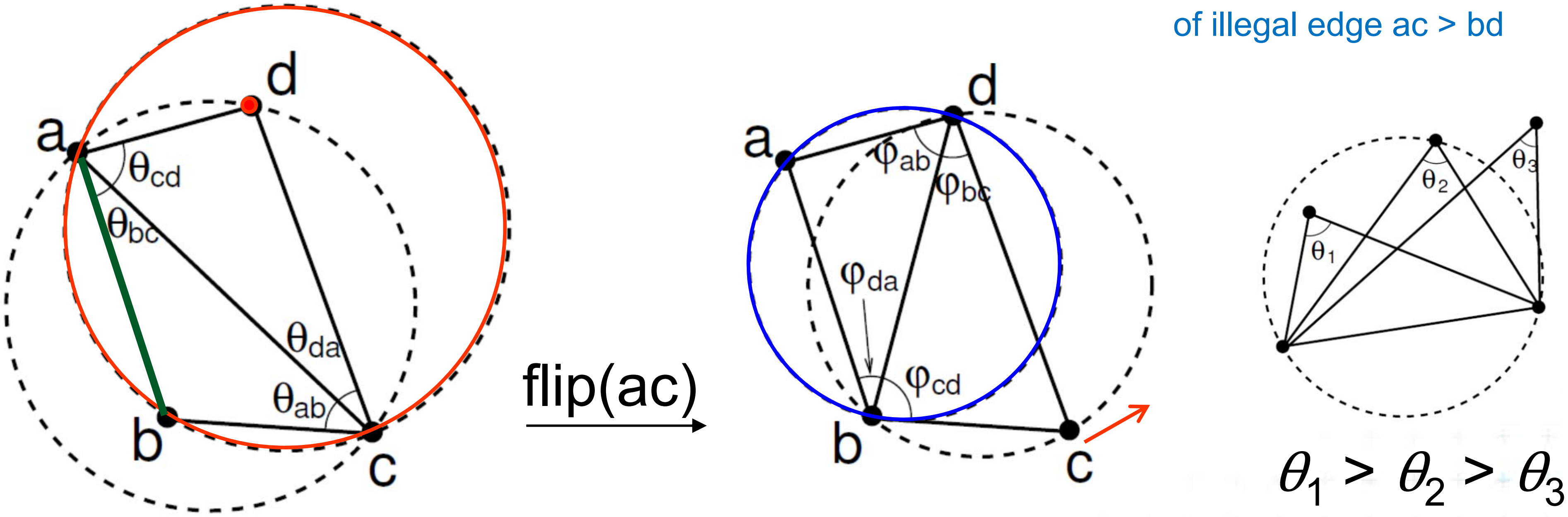
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac| \quad \varphi_{ab} > \theta_{ab} \quad \varphi_{bc} > \theta_{bc} \quad \varphi_{cd} > \theta_{cd} \quad \varphi_{da} > \theta_{da}$

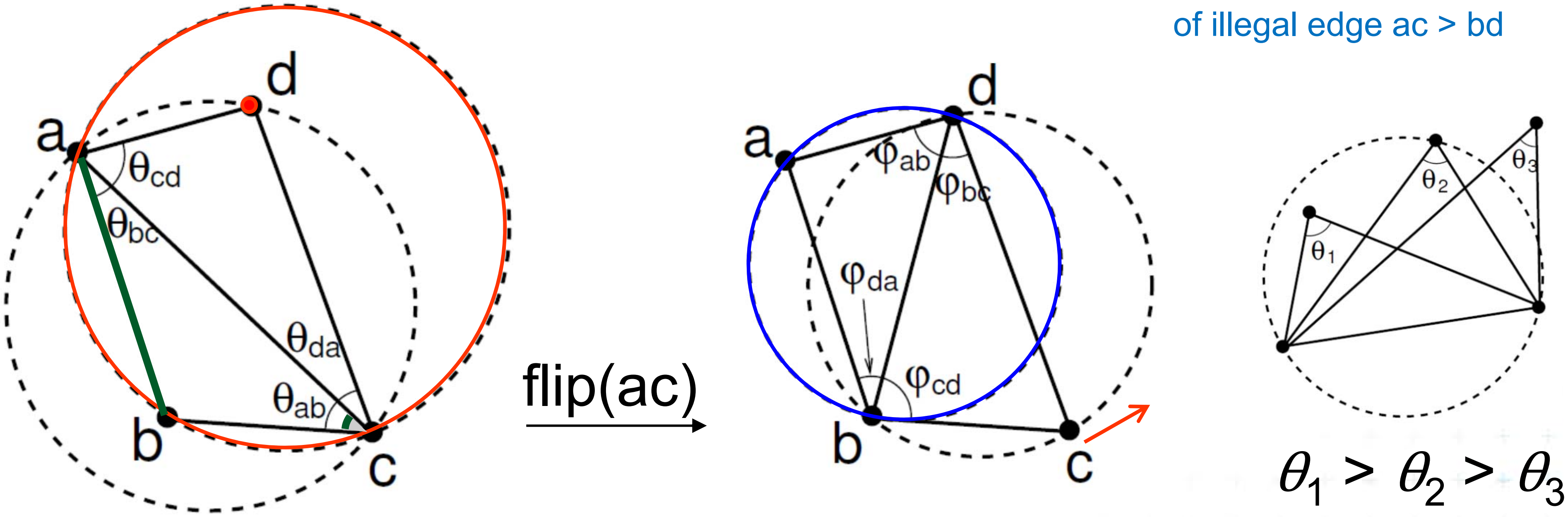
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\varphi_{ab} > \theta_{ab}$ $\varphi_{bc} > \theta_{bc}$ $\varphi_{cd} > \theta_{cd}$ $\varphi_{da} > \theta_{da}$

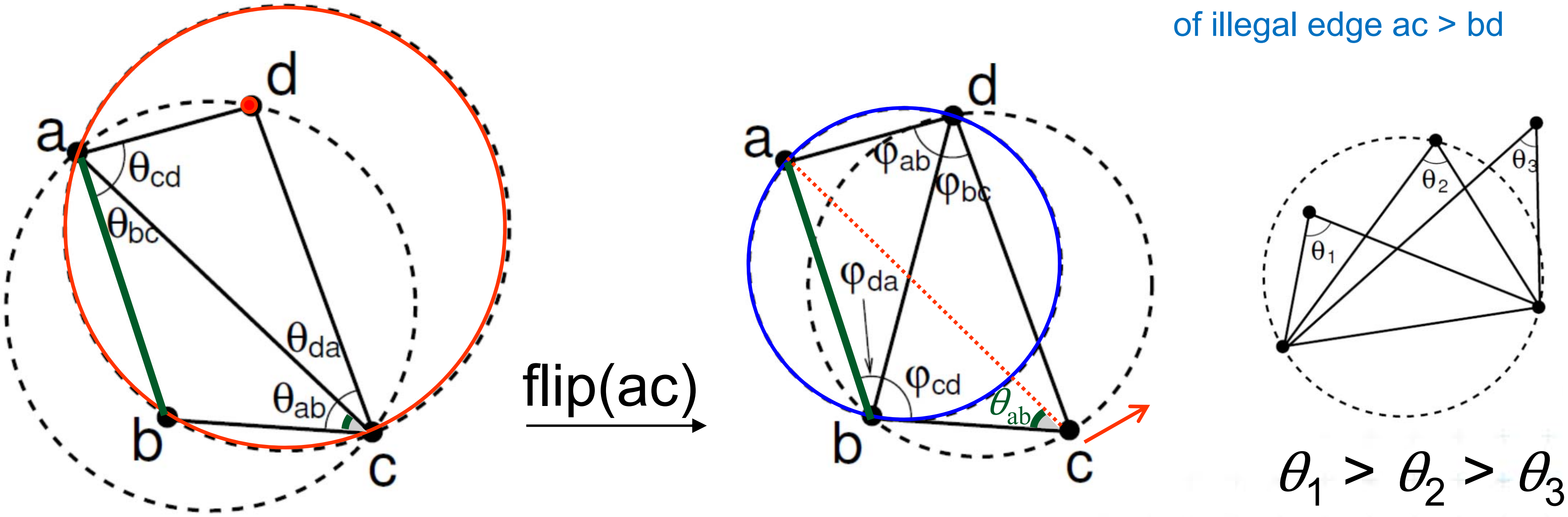
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac| \quad \varphi_{ab} > \theta_{ab} \quad \varphi_{bc} > \theta_{bc} \quad \varphi_{cd} > \theta_{cd} \quad \varphi_{da} > \theta_{da}$

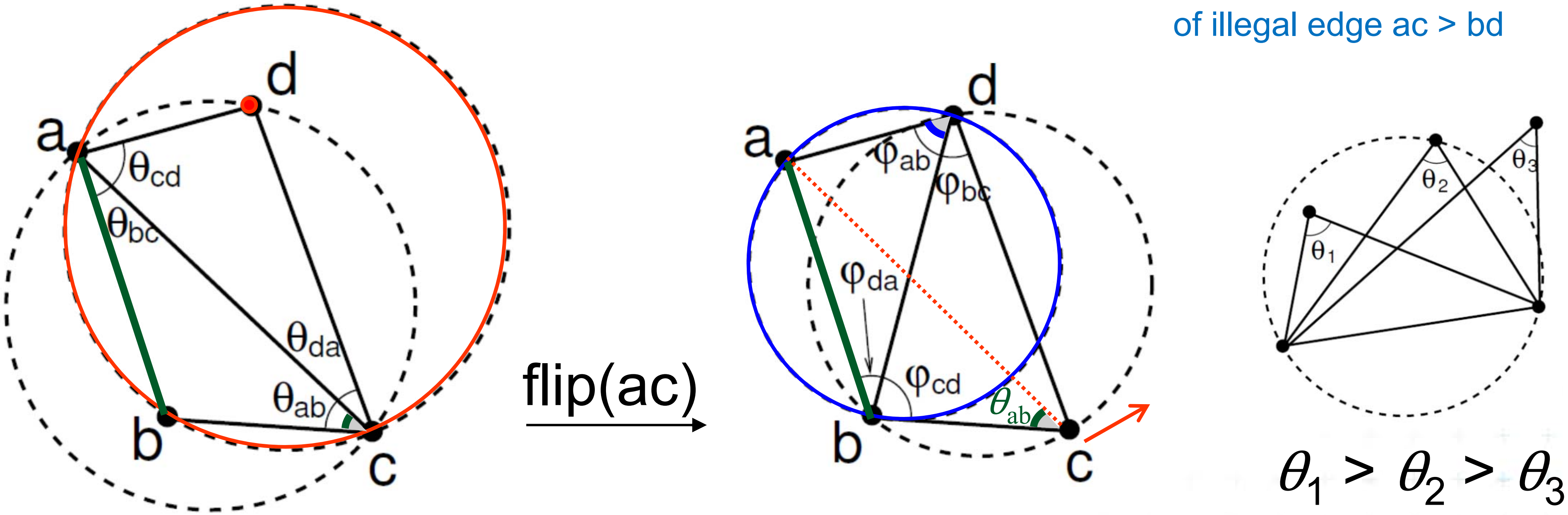
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\quad \varphi_{ab} > \theta_{ab} \quad \varphi_{bc} > \theta_{bc} \quad \varphi_{cd} > \theta_{cd} \quad \varphi_{da} > \theta_{da}$

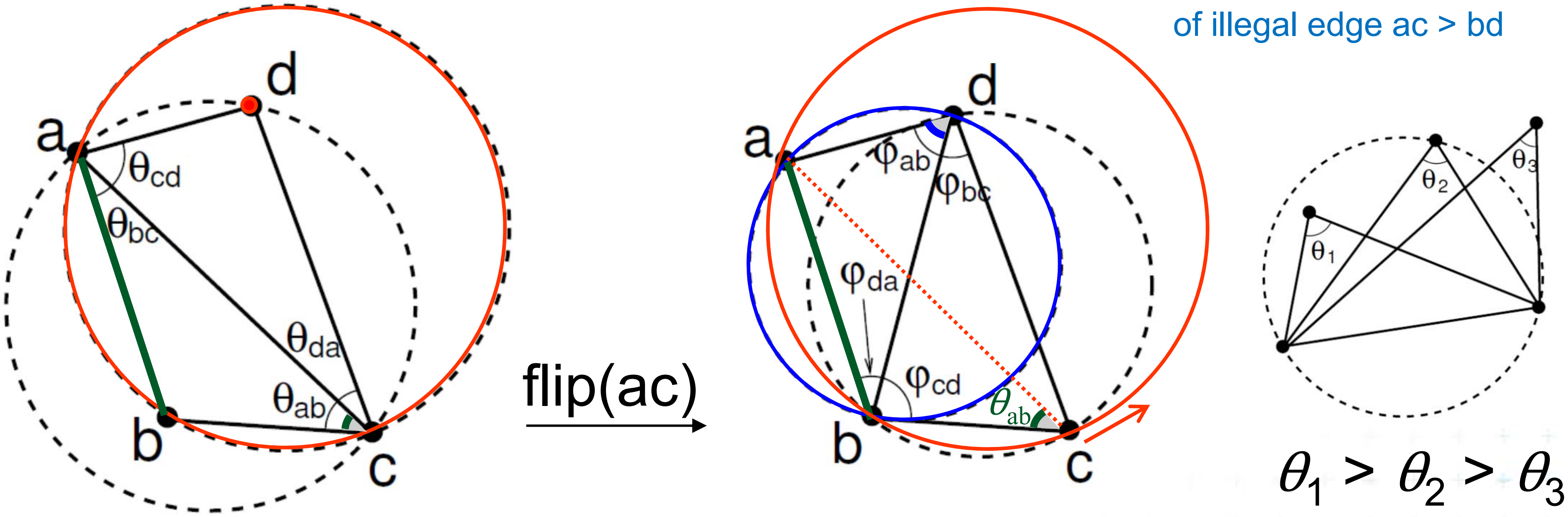
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\quad \varphi_{ab} > \theta_{ab}$ $\quad \varphi_{bc} > \theta_{bc}$ $\quad \varphi_{cd} > \theta_{cd}$ $\quad \varphi_{da} > \theta_{da}$

=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

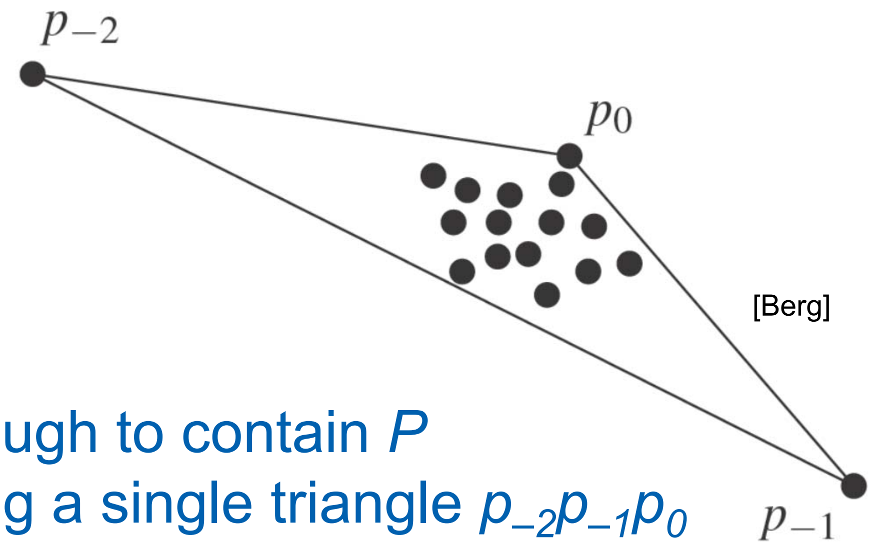
Incremental DT algorithm

Incremental algorithm principle

1. Create a large triangle containing all points (to avoid problems with unbounded cells)
   – must be larger than the largest circle through 3 points
   – will be discarded at the end
2. Insert the points in random order
   – Find triangle with inserted point *p*
   – Add edges to its vertices (these new edges are correct)
   – Check correctness of the old edges (triangles) “around *p*” and legalize (flip) potentially illegal edges
3. Discard the large triangle and incident edges

Incremental algorithm in detail

DelaunayTriangulation(*P*)
Input: Set P of n points in the plane
Output: A Delaunay triangulation T of P



1. Let p_{-2}, p_{-1}, p_0 form a triangle large enough to contain P
2. Initialize T as the triangulation consisting a single triangle $p_{-2}p_{-1}p_0$
3. Compute random permutation $p_1, p_2, \dots, p_n$ of $P \setminus \{p_0\}$
4. **for** $r = 1$ **to** n **do**
5. $\quad T = \text{Insert}(p_r, T)$
6. Discard p_{-1}, p_{-2} with all incident edges from T
7. **return** T

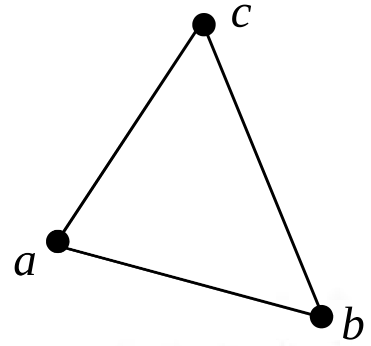
Incremental algorithm – insertion of a point

Insert(***p*, *T*** **)**

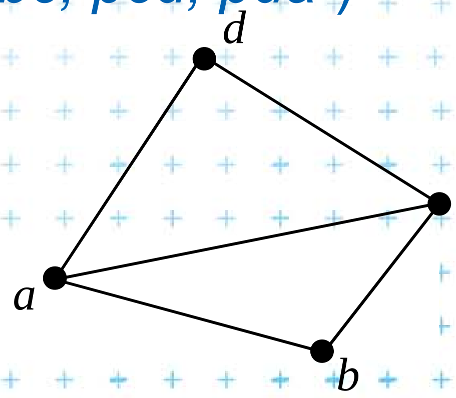
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

1. Find a triangle *abc* $\in$ *T* containing *p*
2. **if** *p* lies in the interior of *abc* **then**
3. Insert edges *pa, pb, pc* into triangulation *T* (splitting *abc* into 3 triangles *pab, pbc, pca*)
4. LegalizeEdge(*p, ab, T*)
5. LegalizeEdge(*p, bc, T*)
6. LegalizeEdge(*p, ca, T*)
7. **else** // *p* lies on the edge of *abc*, say *ac,* point *d* is right from edge *ac*
8. Remove *ac* and insert edges *pa, pb, pc, pd* into triangulation *T* (splitting *abc* and *abd* into 4 triangles *pab, pbc, pcd, pda*)
9. LegalizeEdge(*p, ab, T*)
10. LegalizeEdge(*p, bc, T*)
11. LegalizeEdge(*p, cd, T*)
12. LegalizeEdge(*p, da, T*)
13. **return** *T*



[Berg]



[Berg]

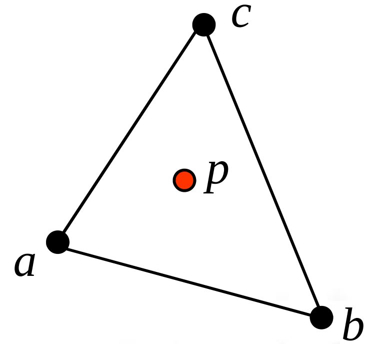
Incremental algorithm – insertion of a point

Insert(***p*, *T*** **)**

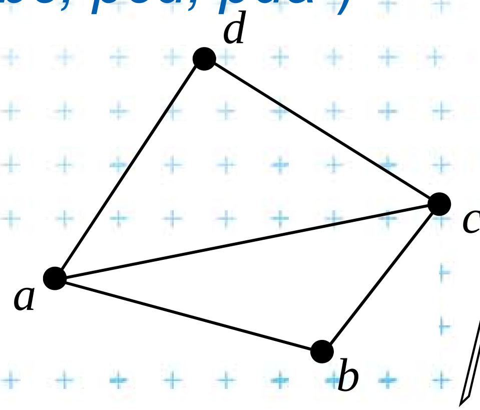
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

1. Find a triangle *abc* $\in$ *T* containing *p*
2. **if** *p* lies in the interior of *abc* **then**
3. Insert edges *pa, pb, pc* into triangulation *T* (splitting *abc* into 3 triangles *pab, pbc, pca*)
4. LegalizeEdge(*p, ab, T*)
5. LegalizeEdge(*p, bc, T*)
6. LegalizeEdge(*p, ca, T*)
7. **else** // *p* lies on the edge of *abc*, say *ac,* point *d* is right from edge *ac*
8. Remove *ac* and insert edges *pa, pb, pc, pd* into triangulation *T* (splitting *abc* and *abd* into 4 triangles *pab, pbc, pcd, pda*)
9. LegalizeEdge(*p, ab, T*)
10. LegalizeEdge(*p, bc, T*)
11. LegalizeEdge(*p, cd, T*)
12. LegalizeEdge(*p, da, T*)
13. **return** *T*



[Berg]



[Berg]

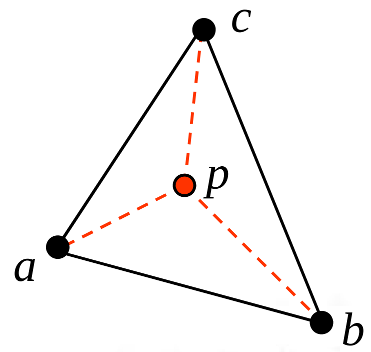
Incremental algorithm – insertion of a point

Insert(***p*****,** ***T*** **)**

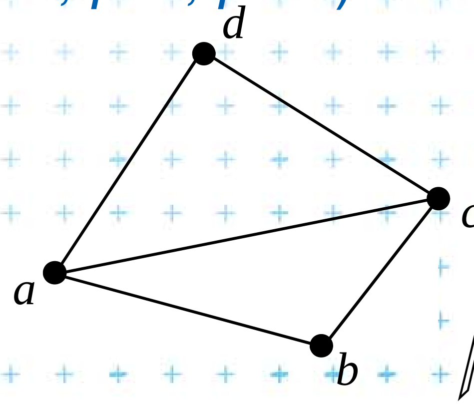
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

1. Find a triangle *abc* $\in$ *T* containing *p*
2. **if** *p* lies in the interior of *abc* **then**
3. Insert edges *pa, pb, pc* into triangulation *T* (splitting *abc* into 3 triangles *pab, pbc, pca*)
4. LegalizeEdge(*p, ab, T*)
5. LegalizeEdge(*p, bc, T*)
6. LegalizeEdge(*p, ca, T*)
7. **else** // *p* lies on the edge of *abc*, say *ac,* point *d* is right from edge *ac*
8. Remove *ac* and insert edges *pa, pb, pc, pd* into triangulation *T* (splitting *abc* and *abd* into 4 triangles *pab, pbc, pcd, pda*)
9. LegalizeEdge(*p, ab, T*)
10. LegalizeEdge(*p, bc, T*)
11. LegalizeEdge(*p, cd, T*)
12. LegalizeEdge(*p, da, T*)
13. **return** *T*



[Berg]



[Berg]

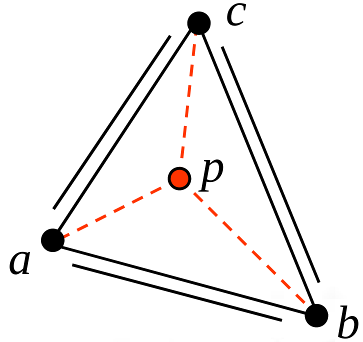
Incremental algorithm – insertion of a point

Insert(***p*, *T*** **)**

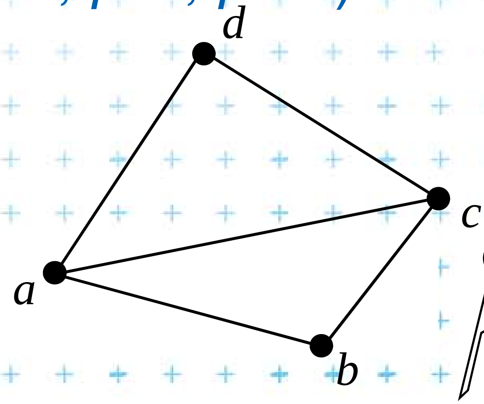
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

1. Find a triangle *abc* ∈ *T* containing *p*
2. **if** *p* lies in the interior of *abc* **then**
3. Insert edges *pa, pb, pc* into triangulation *T* (splitting *abc* into 3 triangles *pab, pbc, pca*)
4. LegalizeEdge(*p, ab, T*)
5. LegalizeEdge(*p, bc, T*)
6. LegalizeEdge(*p, ca, T*)
7. **else** // *p* lies on the edge of *abc*, say *ac,* point *d* is right from edge *ac*
8. Remove *ac* and insert edges *pa, pb, pc, pd* into triangulation *T* (splitting *abc* and *abd* into 4 triangles *pab, pbc, pcd, pda*)
9. LegalizeEdge(*p, ab, T*)
10. LegalizeEdge(*p, bc, T*)
11. LegalizeEdge(*p, cd, T*)
12. LegalizeEdge(*p, da, T*)
13. **return** *T*



[Berg]



[Berg]

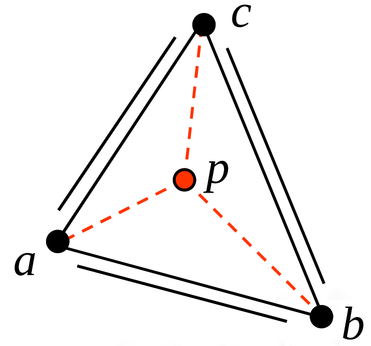
Incremental algorithm – insertion of a point

Insert(*p*, *T*)

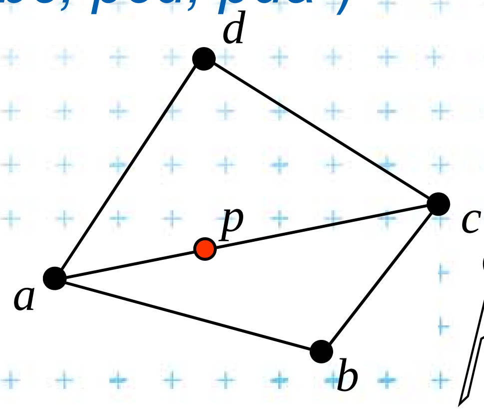
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

1. Find a triangle *abc* $\in$ *T* containing *p*
2. **if** *p* lies in the interior of *abc* **then**
3. Insert edges *pa, pb, pc* into triangulation *T* (splitting *abc* into 3 triangles *pab, pbc, pca*)
4. LegalizeEdge(*p, ab, T*)
5. LegalizeEdge(*p, bc, T*)
6. LegalizeEdge(*p, ca, T*)
7. **else** // *p* lies on the edge of *abc*, say *ac,* point *d* is right from edge *ac*
8. Remove *ac* and insert edges *pa, pb, pc, pd* into triangulation *T* (splitting *abc* and *abd* into 4 triangles *pab, pbc, pcd, pda*)
9. LegalizeEdge(*p, ab, T*)
10. LegalizeEdge(*p, bc, T*)
11. LegalizeEdge(*p, cd, T*)
12. LegalizeEdge(*p, da, T*)
13. **return** *T*



[Berg]



[Berg]

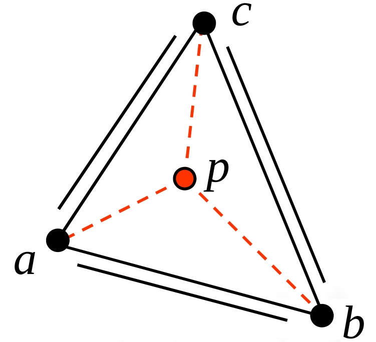
Incremental algorithm – insertion of a point

Insert(***p*, *T*** **)**

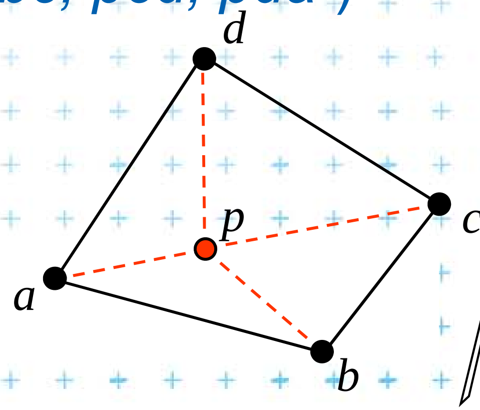
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

1. Find a triangle *abc* ∈ *T* containing *p*
2. **if** *p* lies in the interior of *abc* **then**
3. Insert edges *pa, pb, pc* into triangulation *T* (splitting *abc* into 3 triangles *pab, pbc, pca*)
4. LegalizeEdge(*p, ab, T*)
5. LegalizeEdge(*p, bc, T*)
6. LegalizeEdge(*p, ca, T*)
7. **else** // *p* lies on the edge of *abc*, say *ac,* point *d* is right from edge *ac*
8. Remove *ac* and insert edges *pa, pb, pc, pd* into triangulation *T* (splitting *abc* and *abd* into 4 triangles *pab, pbc, pcd, pda*)
9. LegalizeEdge(*p, ab, T*)
10. LegalizeEdge(*p, bc, T*)
11. LegalizeEdge(*p, cd, T*)
12. LegalizeEdge(*p, da, T*)
13. **return** *T*



[Berg]



[Berg]

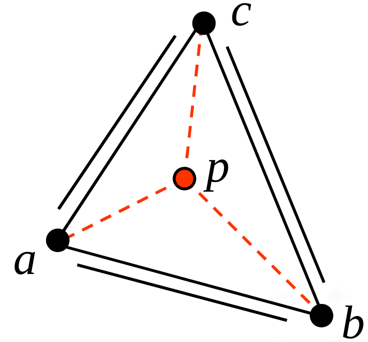
Incremental algorithm – insertion of a point

Insert(***p*****,** ***T*** **)**

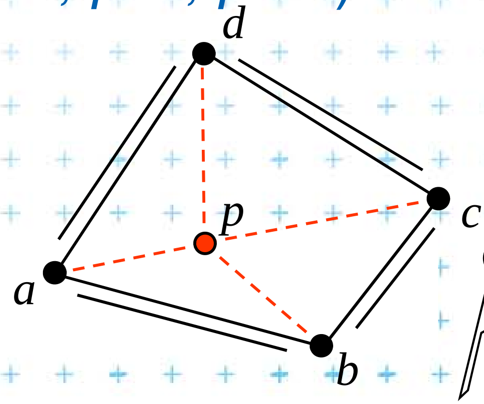
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

1. Find a triangle *abc* $\in$ *T* containing *p*
2. **if** *p* lies in the interior of *abc* **then**
3. Insert edges *pa, pb, pc* into triangulation *T* (splitting *abc* into 3 triangles *pab, pbc, pca*)
4. LegalizeEdge(*p, ab, T*)
5. LegalizeEdge(*p, bc, T*)
6. LegalizeEdge(*p, ca, T*)
7. **else** // *p* lies on the edge of *abc*, say *ac,* point *d* is right from edge *ac*
8. Remove *ac* and insert edges *pa, pb, pc, pd* into triangulation *T* (splitting *abc* and *abd* into 4 triangles *pab, pbc, pcd, pda*)
9. LegalizeEdge(*p, ab, T*)
10. LegalizeEdge(*p, bc, T*)
11. LegalizeEdge(*p, cd, T*)
12. LegalizeEdge(*p, da, T*)
13. **return** *T*



[Berg]



[Berg]

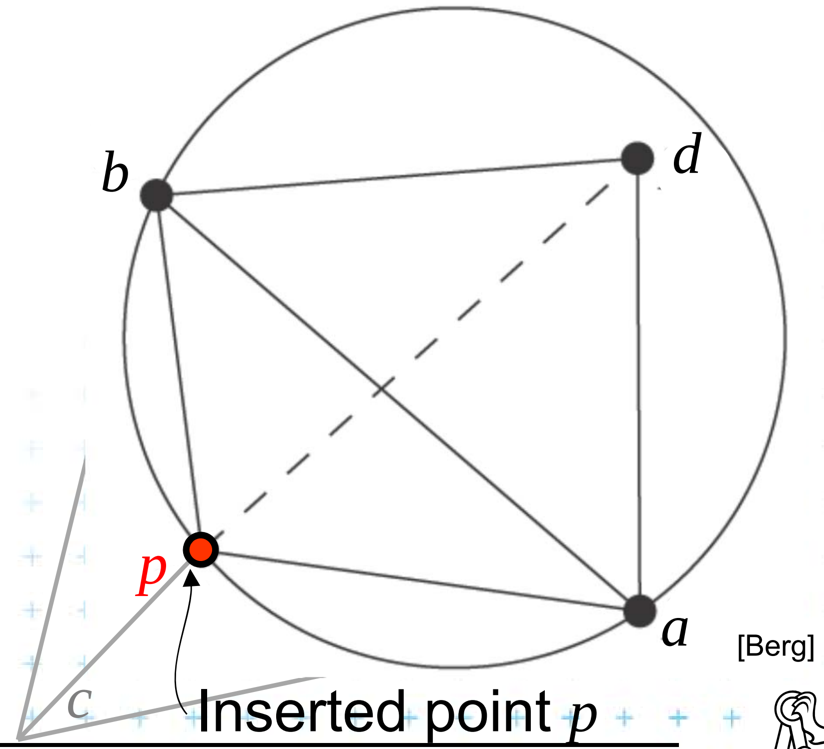
Incremental algorithm – edge legalization

LegalizeEdge(*p*, *ab*, *T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of $p \cup T$

1. **if**(*ab* is edge on the exterior face) **return**
2. let *d* be the vertex to the right of edge *ab*
3. if(inCircle(*p, a, b, d*)) // *d* is in the circle around *pab* => *d* is illegal
4. Flip edge *ab* for *pd*
5. LegalizeEdge(*p, ad, T*)
6. LegalizeEdge(*p, db, T*)



DCGI

Incremental algorithm – edge legalization

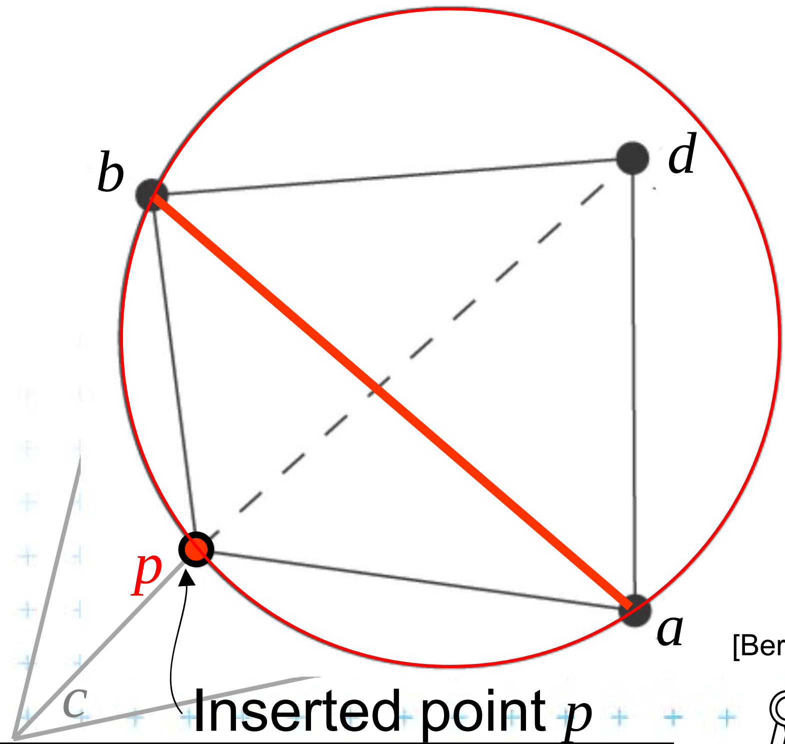
LegalizeEdge(***p, ab, T*** **)**

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of $p \cup T$

1. **if**(*ab* is edge on the exterior face) **return**
2. let *d* be the vertex to the right of edge *ab*
3. if(inCircle(*p, a, b, d*)) // *d* is in the circle around *pab* => *d* is illegal
4. Flip edge *ab* for *pd*
5. LegalizeEdge(*p, ad, T*)
6. LegalizeEdge(*p, db, T*)

Insertion of *p* may make edges *ab*, *bc* & *ca* illegal
(circle around *pab* will contain point *d*)



[Berg]

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

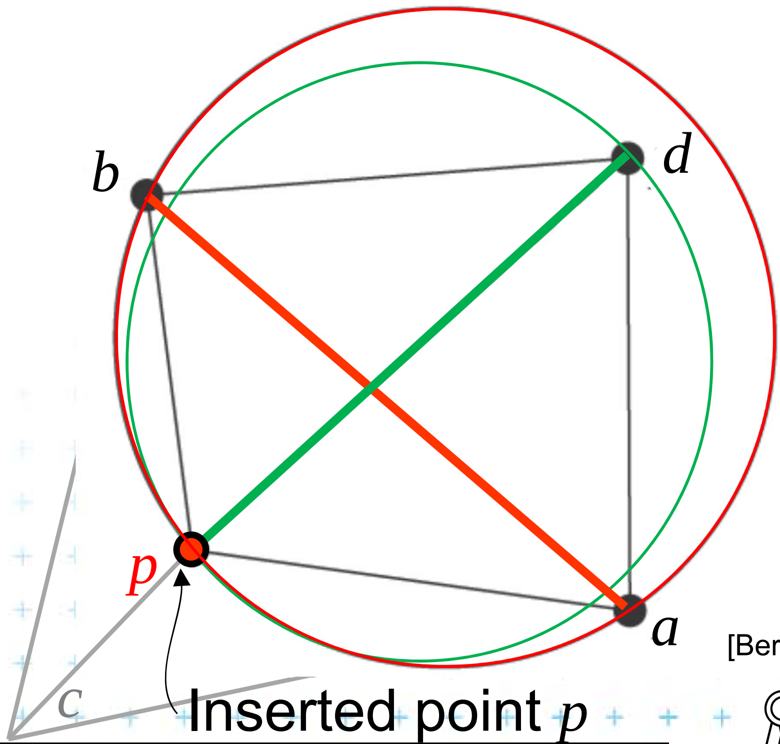
Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of *p* ∪*T*

1. **if**(*ab* is edge on the exterior face) **return**
2. let *d* be the vertex to the right of edge *ab*
3. if(inCircle(*p, a, b, d*)) // *d* is in the circle around *pab* => *d* is illegal
4. Flip edge *ab* for *pd*
5. LegalizeEdge(*p, ad, T*)
6. LegalizeEdge(*p, db, T*)

Insertion of *p* may make edges *ab, bc* & *ca* illegal (circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal (the circumcircles of the resulting triangles *pdb,* and *pad* will bee empty)



[Berg]

DCGI

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

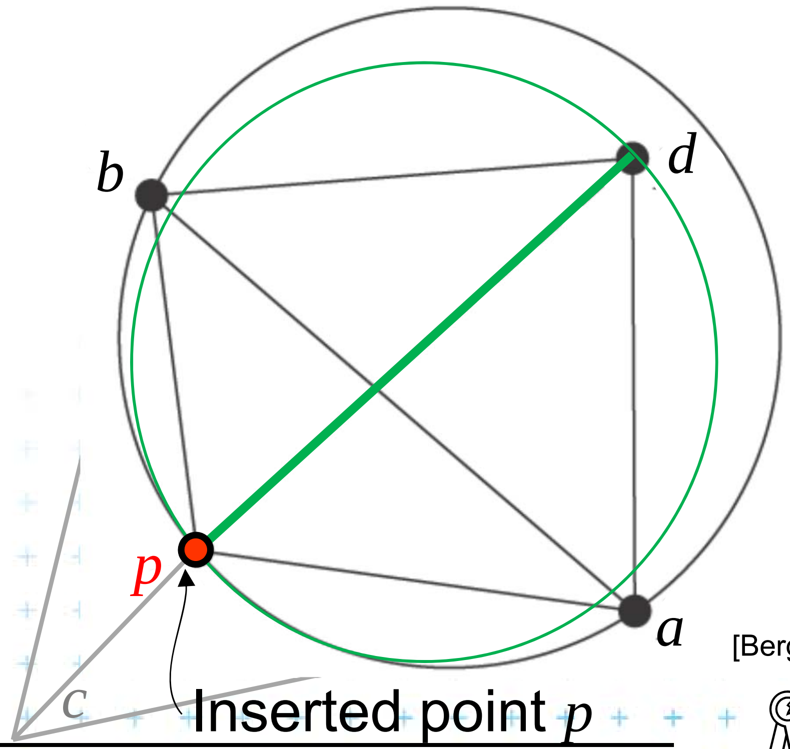
Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of $p \cup T$

1. **if**(*ab* is edge on the exterior face) **return**
2. let *d* be the vertex to the right of edge *ab*
3. if(inCircle(*p, a, b, d*)) // *d* is in the circle around *pab* => *d* is illegal
4. Flip edge *ab* for *pd*
5. LegalizeEdge(*p, ad, T*)
6. LegalizeEdge(*p, db, T*)

Insertion of *p* may make edges *ab, bc* & *ca* illegal (circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal (the circumcircles of the resulting triangles *pdb,* and *pad* will bee empty)



[Berg]

DCGI

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of $p \cup T$

if(*ab* is edge on the exterior face) **return**

let *d* be the vertex to the right of edge *ab*

if(inCircle(*p, a, b, d*)) // *d* is in the circle around *pab* => *d* is illegal

  Flip edge *ab* for *pd*

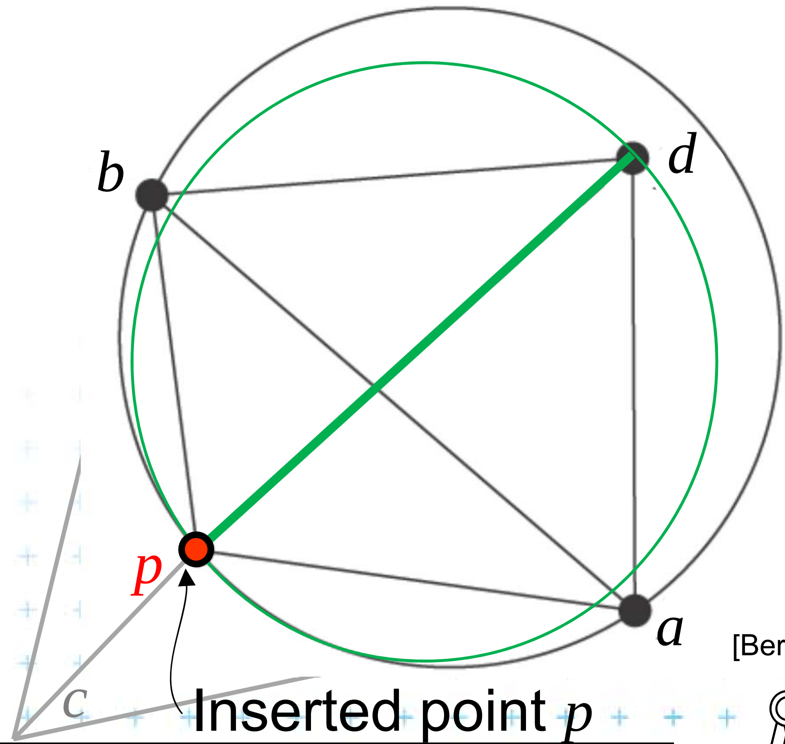
  LegalizeEdge(*p, ad, T*)

  LegalizeEdge(*p, db, T*)

Insertion of *p* may make edges *ab, bc* & *ca* illegal (circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal (the circumcircles of the resulting triangles *pdb,* and *pad* will bee empty)

We must check and possibly flip edges *ad, db*



[Berg]

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

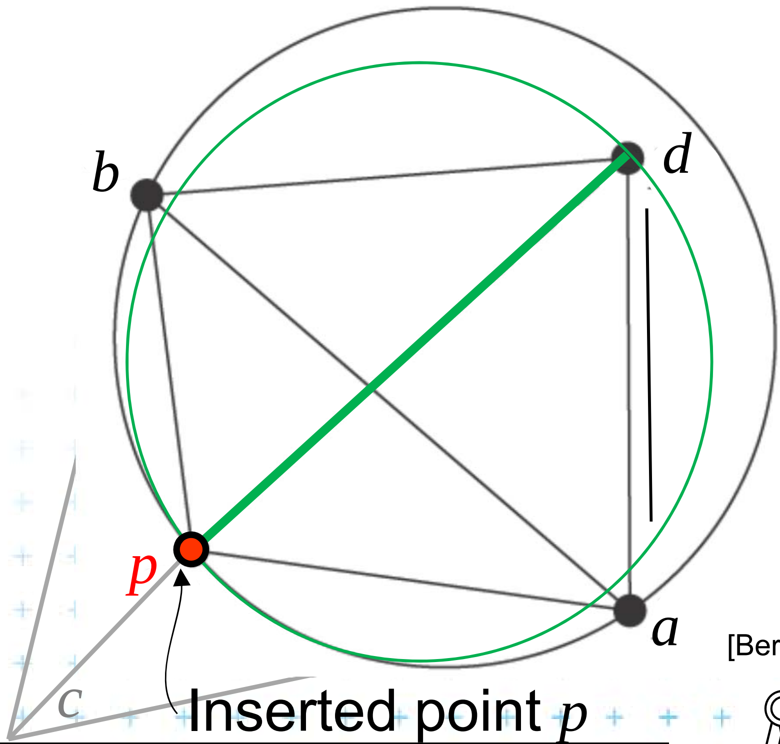
Output: Delaunay triangulation of $p \cup T$

1. **if**(*ab* is edge on the exterior face) **return**
2. let *d* be the vertex to the right of edge *ab*
3. if(inCircle(*p, a, b, d*)) // *d* is in the circle around *pab* => *d* is illegal
4. Flip edge *ab* for *pd*
5. LegalizeEdge(*p, ad, T*)
6. LegalizeEdge(*p, db, T*)

Insertion of *p* may make edges *ab, bc* & *ca* illegal
(circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal
(the circumcircles of the resulting triangles
pdb, and *pad* will bee empty)

We must check and possibly flip edges *ad, db*



[Berg]

DCGI

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

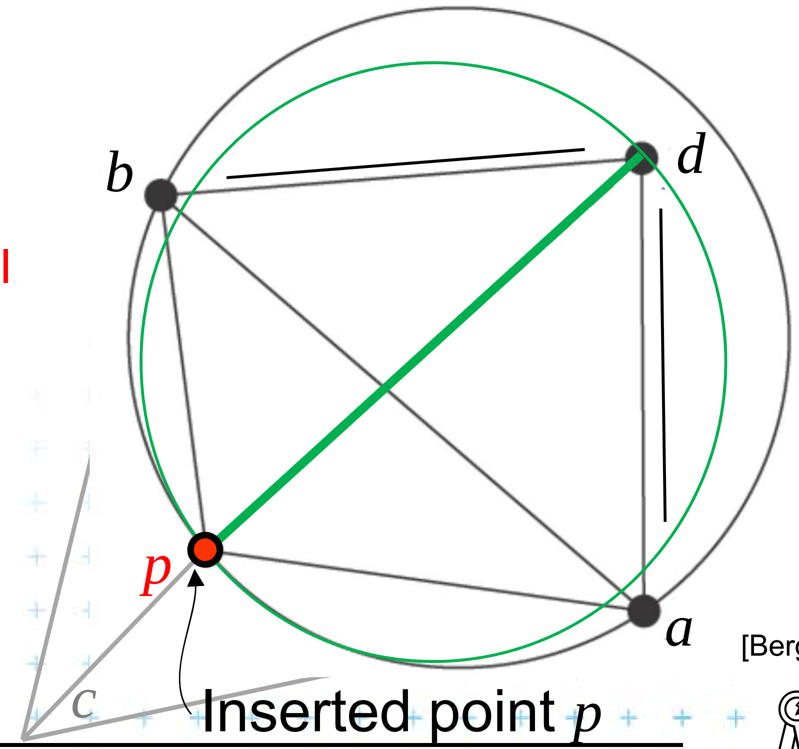
Output: Delaunay triangulation of $p \cup T$

1. **if**(*ab* is edge on the exterior face) **return**
2. let *d* be the vertex to the right of edge *ab*
3. if(inCircle(*p, a, b, d*)) // *d* is in the circle around *pab* => *d* is illegal
4. Flip edge *ab* for *pd*
5. LegalizeEdge(*p, ad, T*)
6. LegalizeEdge(*p, db, T*)

Insertion of *p* may make edges *ab, bc* & *ca* illegal
(circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal
(the circumcircles of the resulting triangles
pdb, and *pad* will bee empty)

We must check and possibly flip edges *ad, db*



[Berg]

DCGI

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of *p* ∪*T*

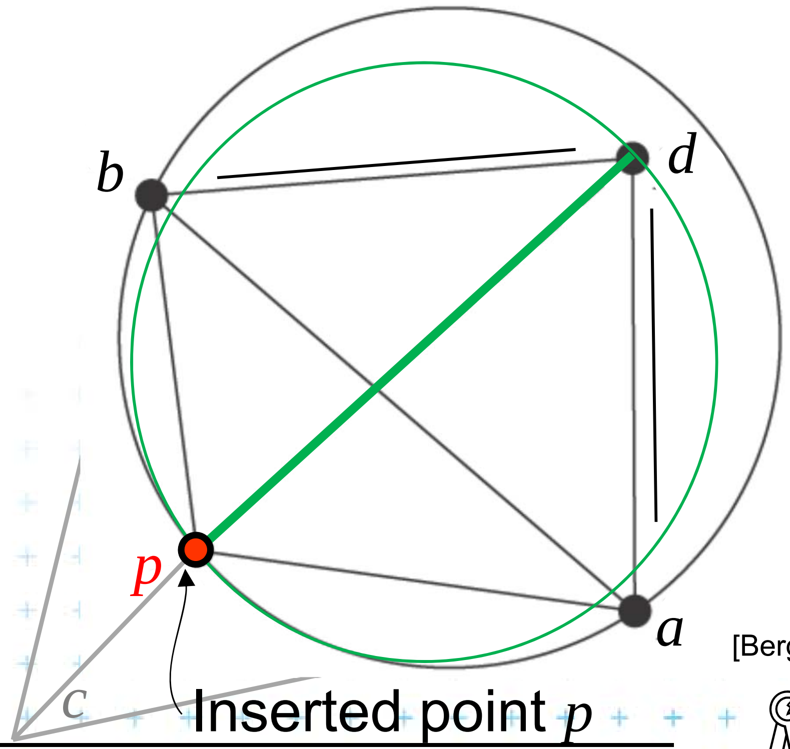
1. **if**(*ab* is edge on the exterior face) **return**
2. let *d* be the vertex to the right of edge *ab*
3. if(inCircle(*p, a, b, d*)) // *d* is in the circle around *pab* => *d* is illegal
4. Flip edge *ab* for *pd*
5. LegalizeEdge(*p, ad, T*)
6. LegalizeEdge(*p, db, T*)

Insertion of *p* may make edges *ab, bc* & *ca* illegal (circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal (the circumcircles of the resulting triangles *pdb,* and *pad* will bee empty)

We must check and possibly flip edges *ad, db*

(We must check and possibly flip edges *bc* & *ca* of the triangle *abc* - lines 5,6 in Insert(p, T))

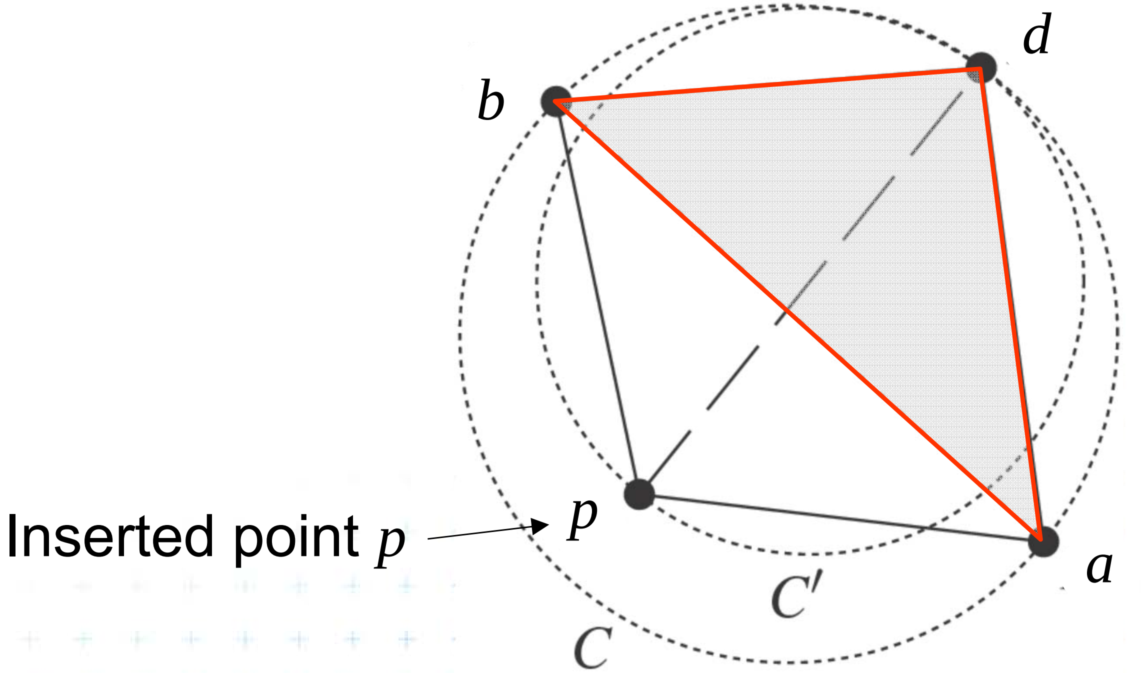


[Berg]

DCGI

Correctness of edge flip of illegal edge

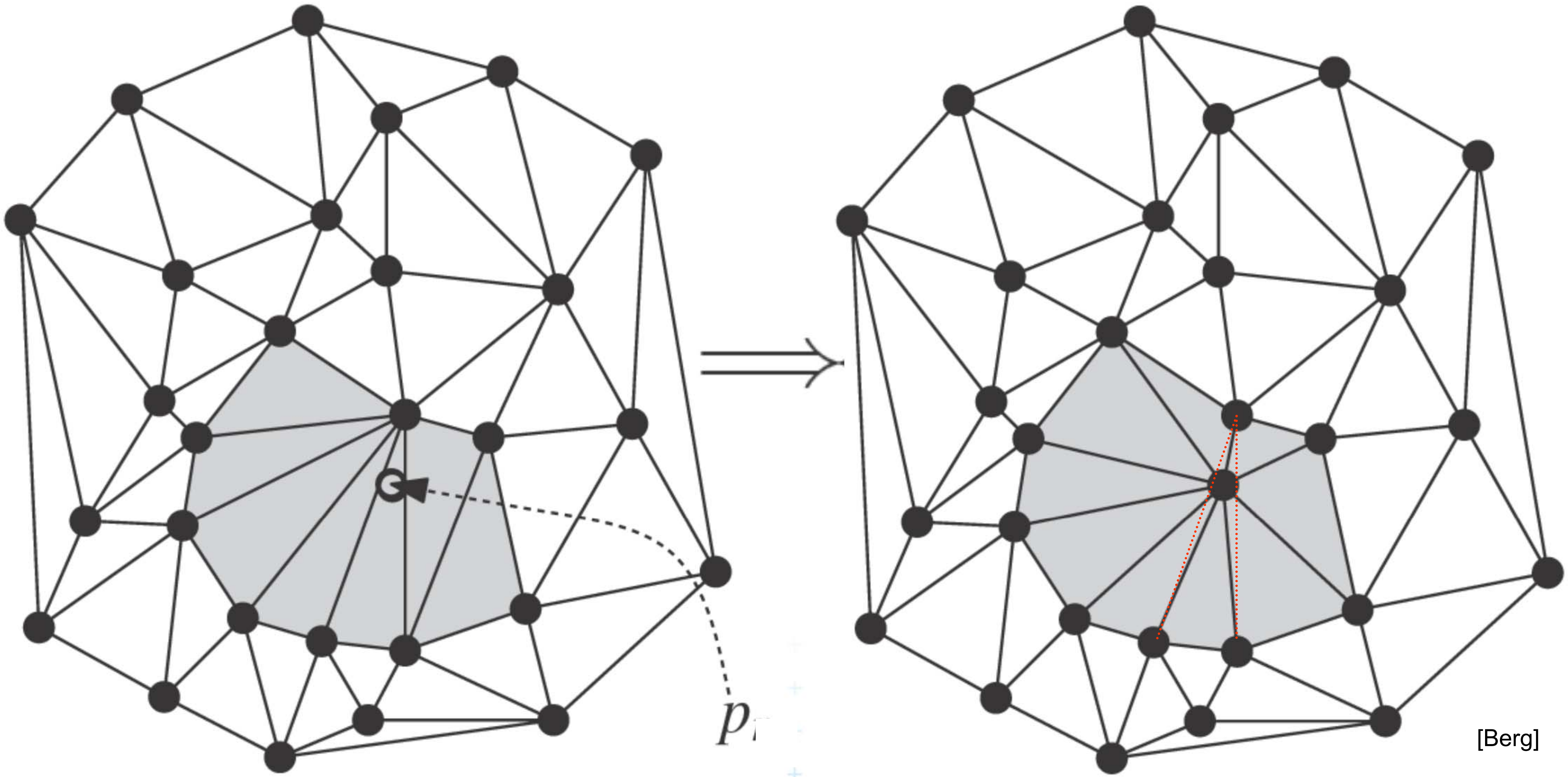
- Assume point p is in C (it violates DT criteria for adb)
- adb was a triangle of DT => C was an empty circle
- Create circle C' trough point p, C' is inscribed to C, $C' \subset C$
  => C' is also an empty circle $(a, b \notin C)$
  => new edge *pd* is also a Delone edge



Contradiction edge ab cannot be Delone edge

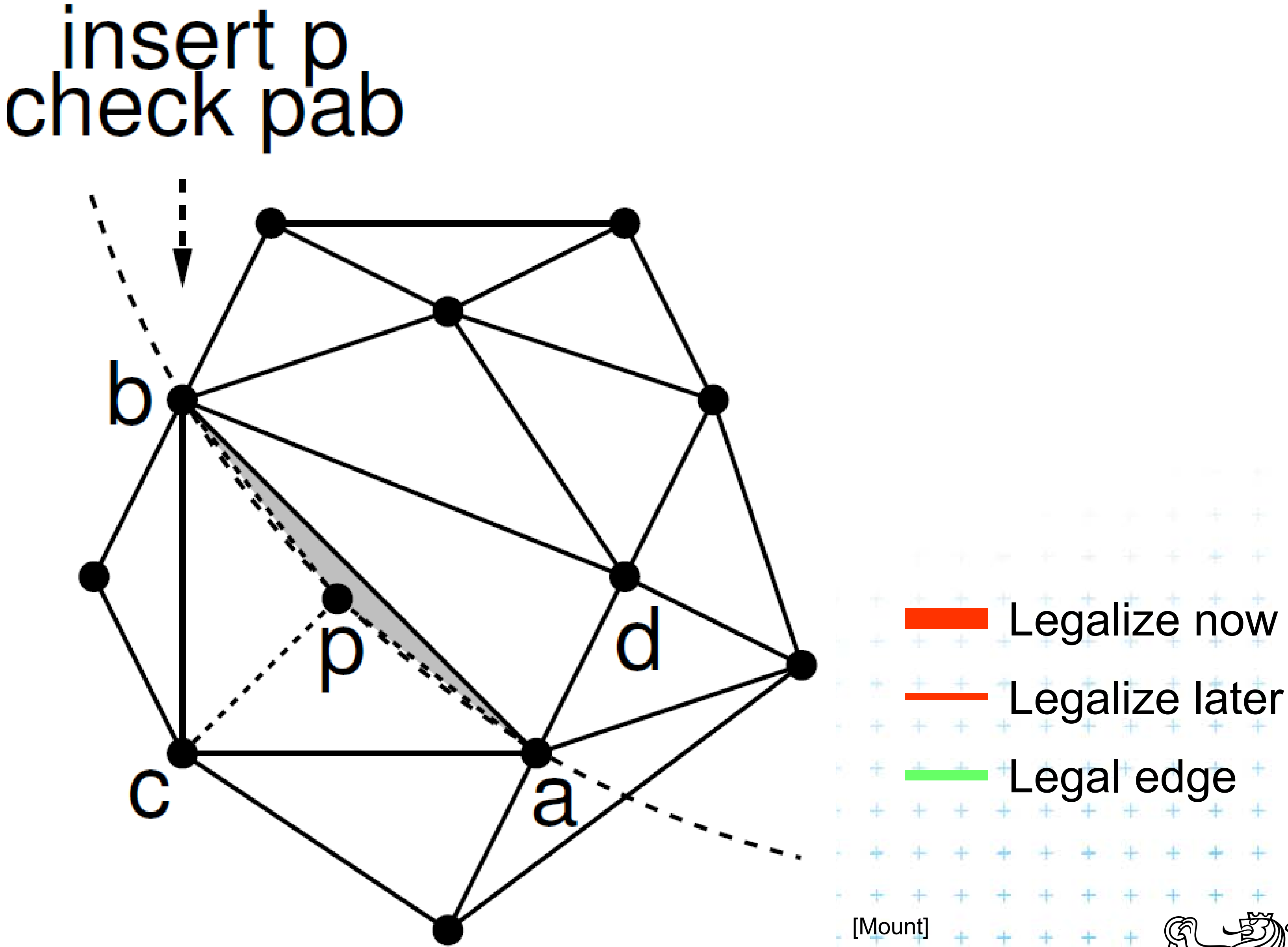
[Berg]

DT- point insert and mesh legalization



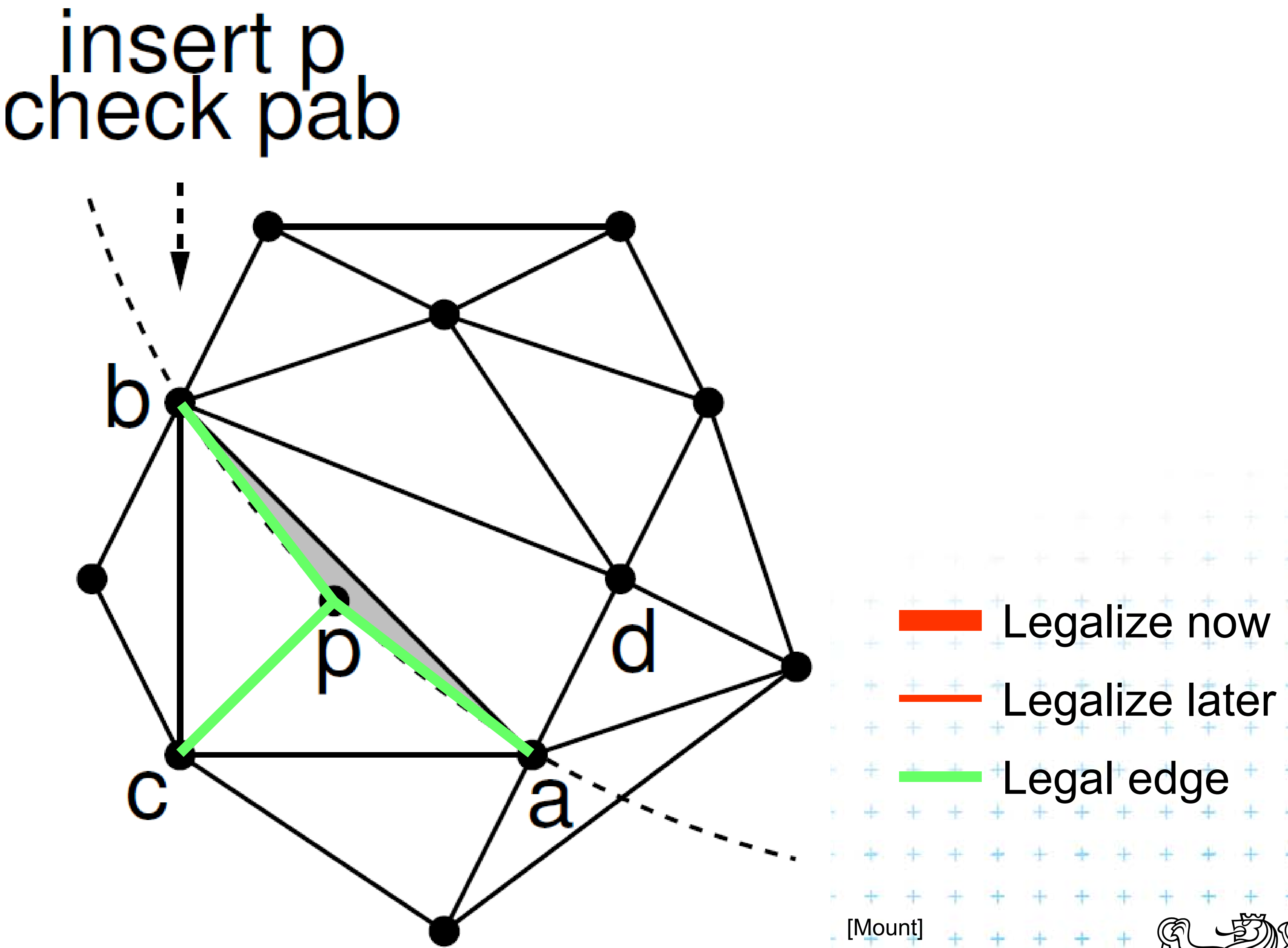
Every new edge created due to insertion of *p* will be incident to *p*

Delaunay triangulation – other point insert



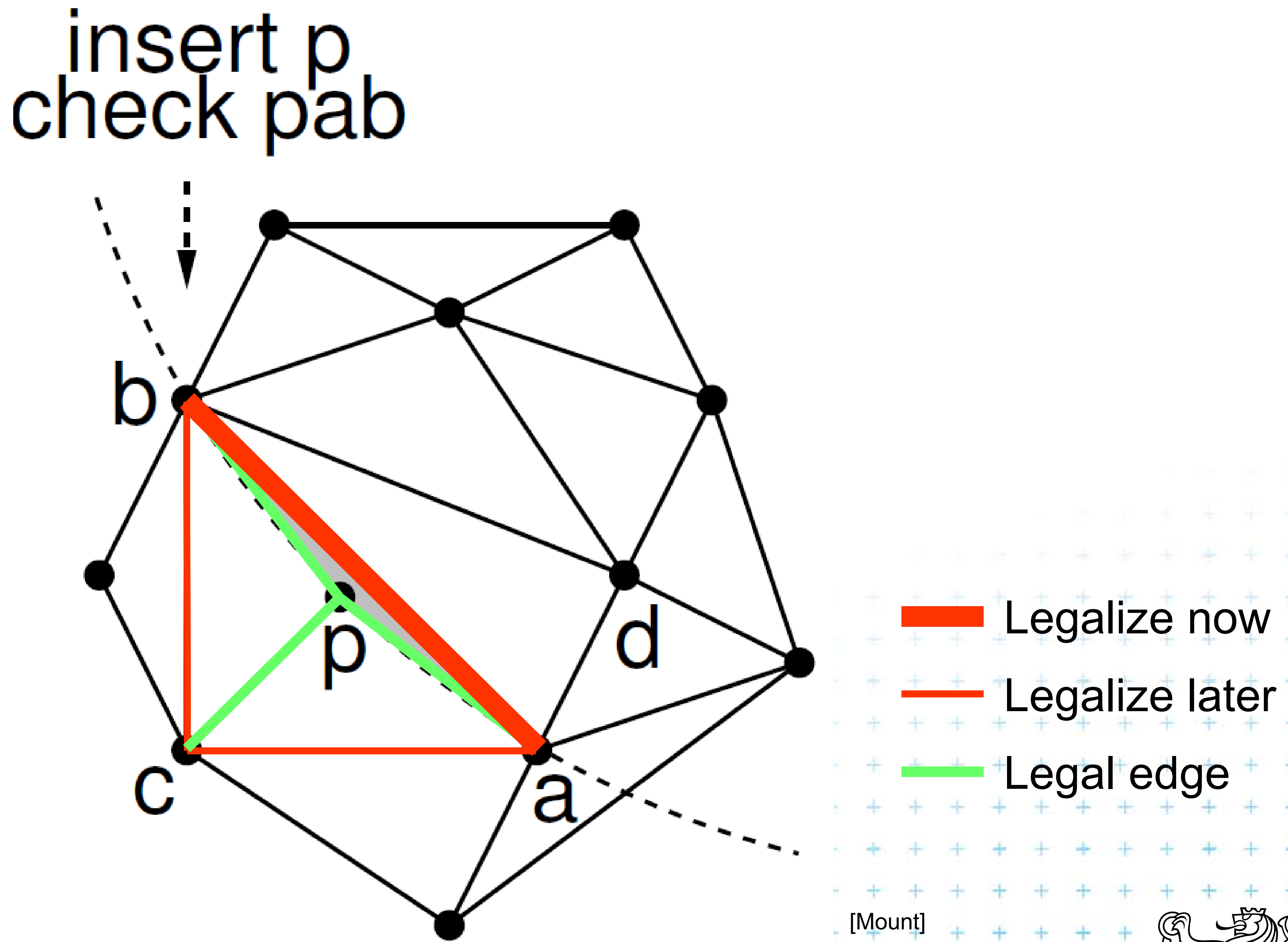
[Mount]

Delaunay triangulation – other point insert



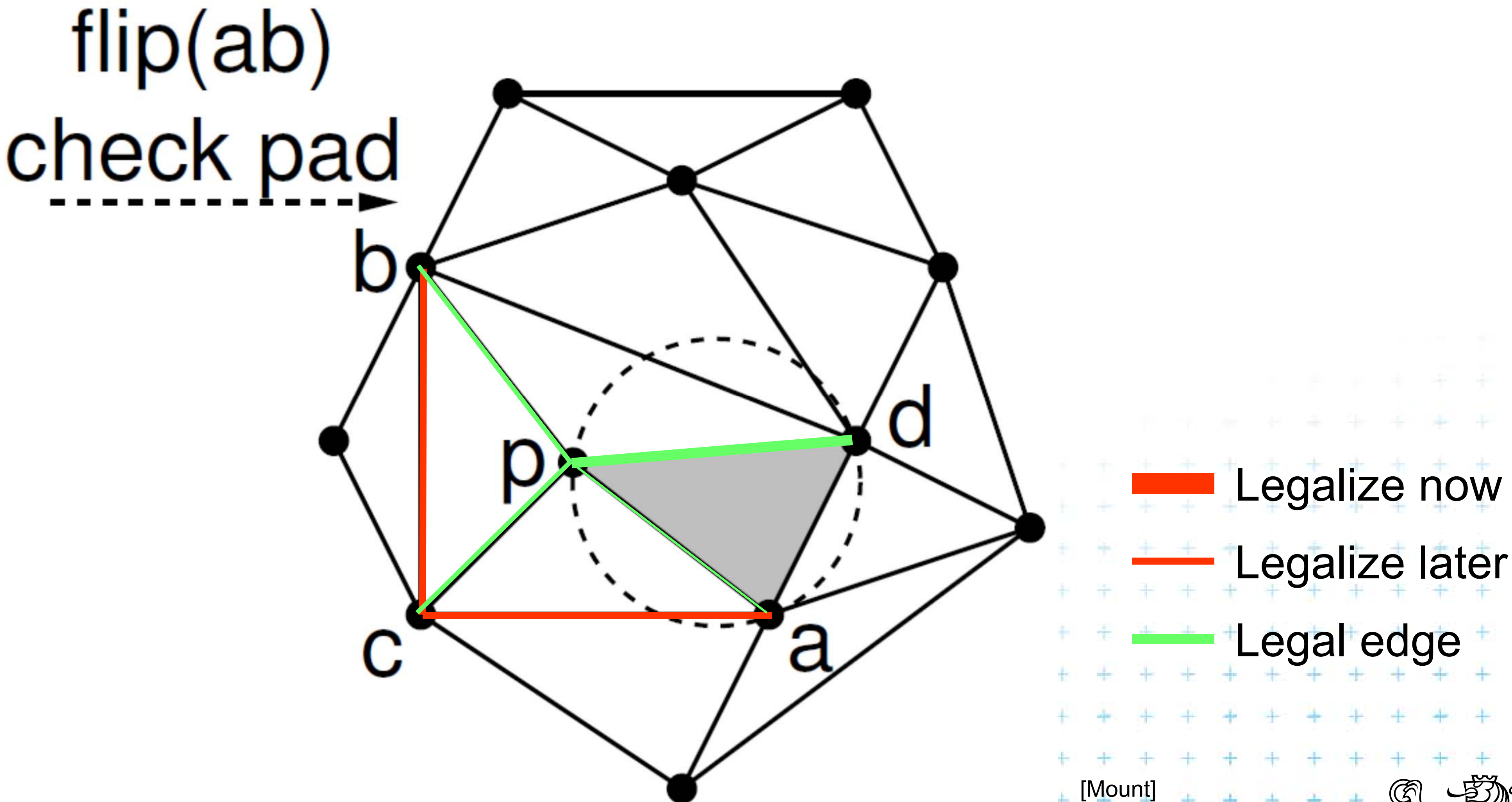
[Mount]

Delaunay triangulation – other point insert



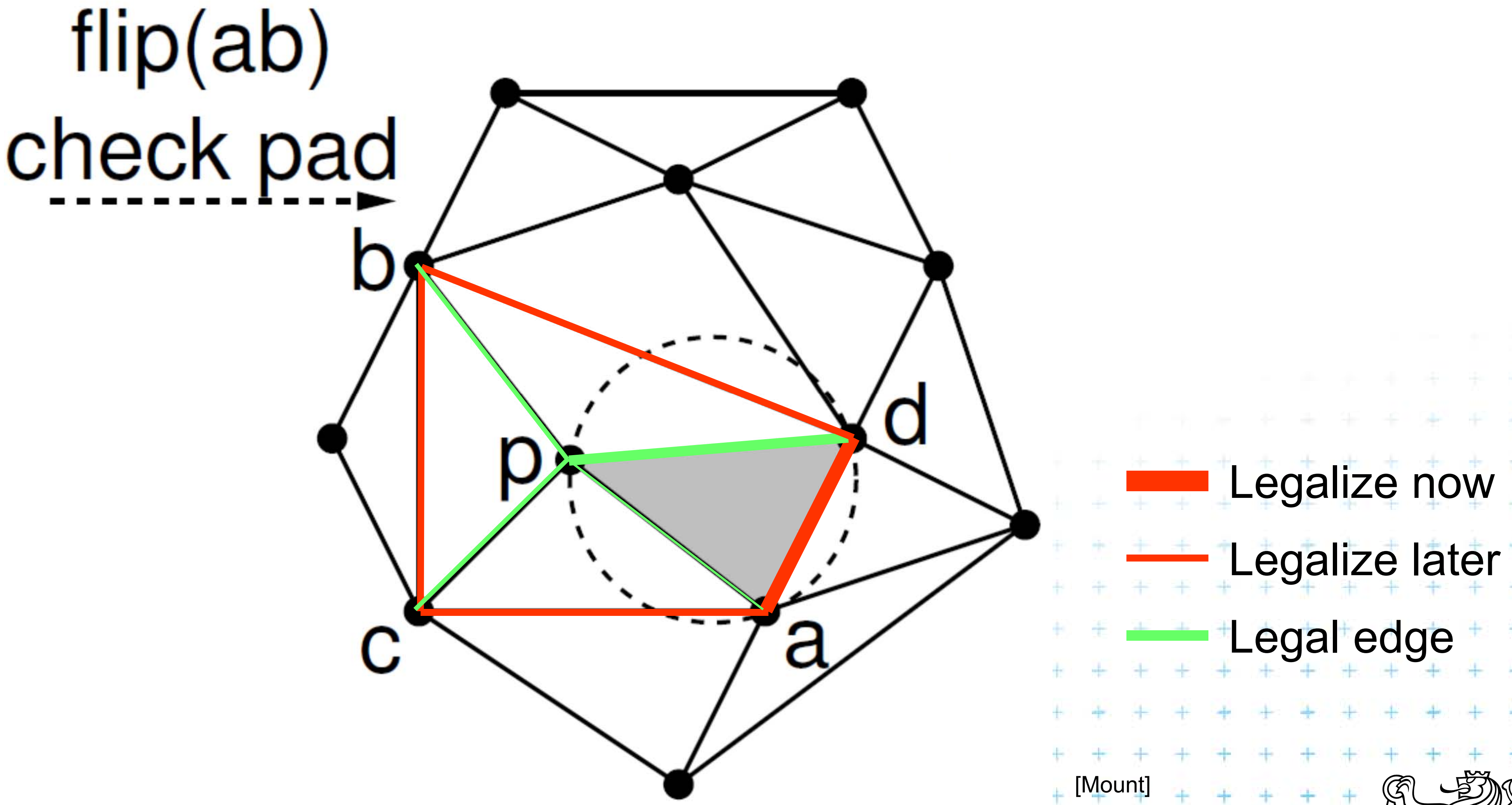
[Mount]

Delaunay triangulation – other point insert



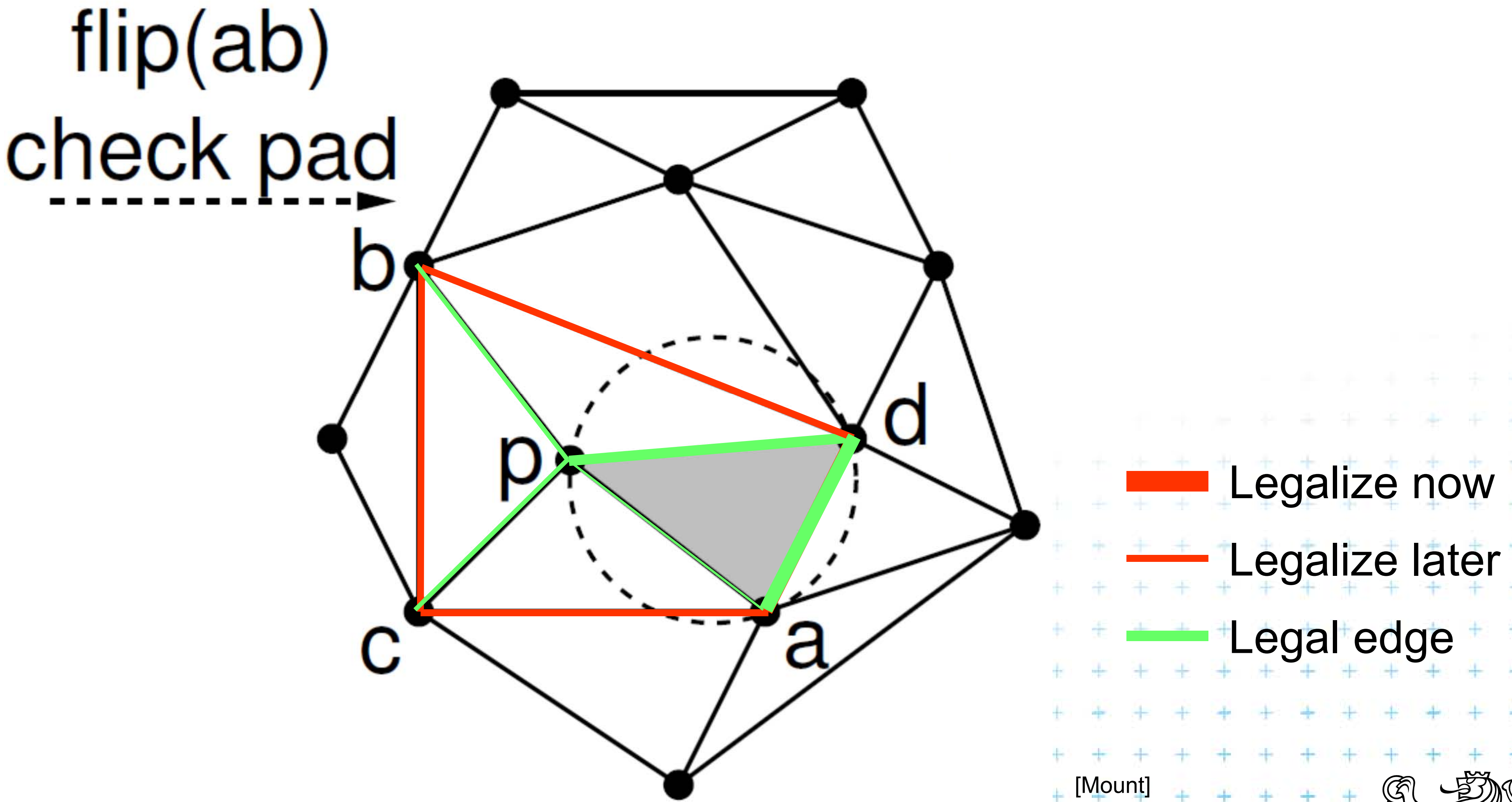
[Mount]

Delaunay triangulation – other point insert



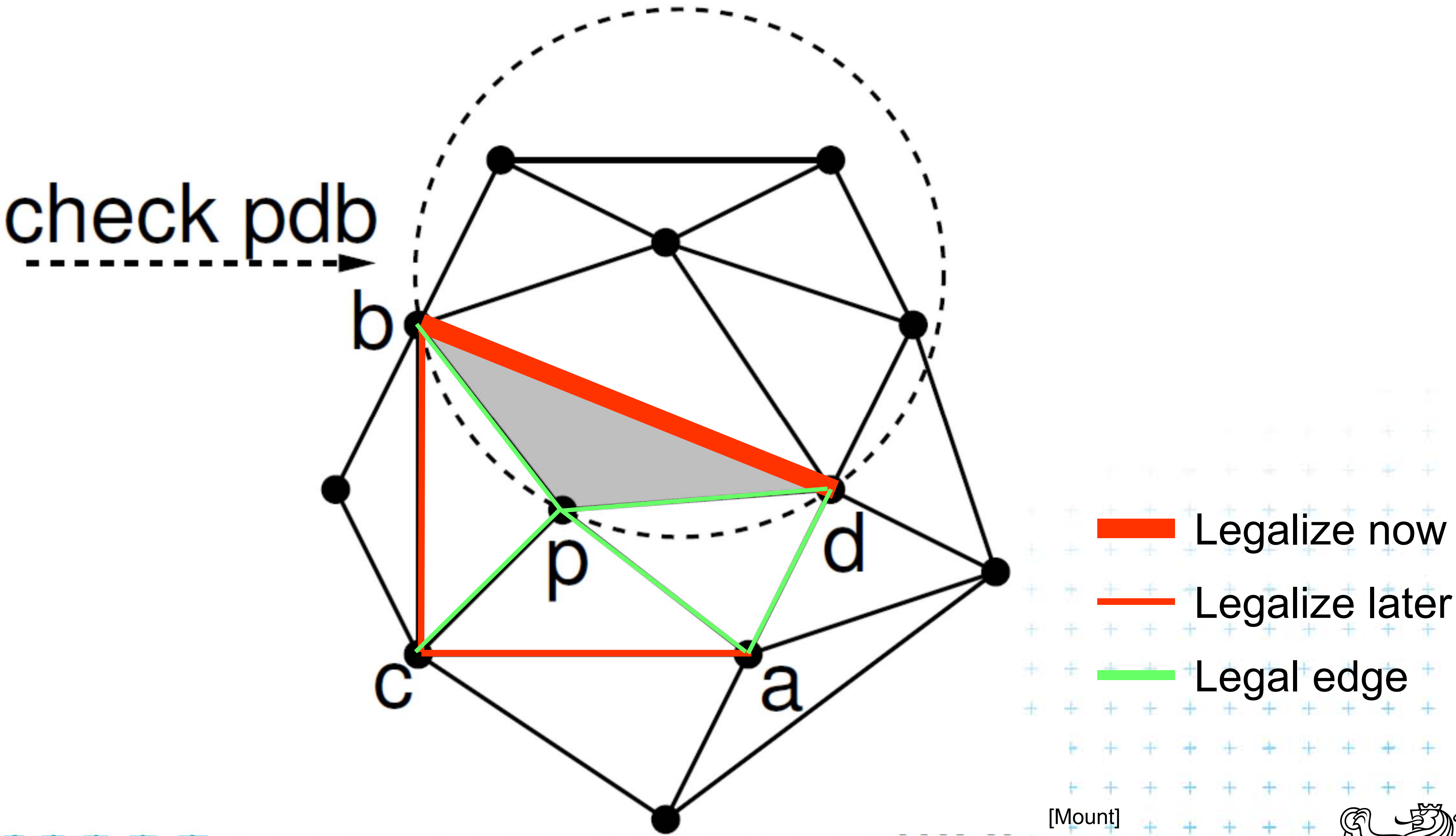
[Mount]

Delaunay triangulation – other point insert



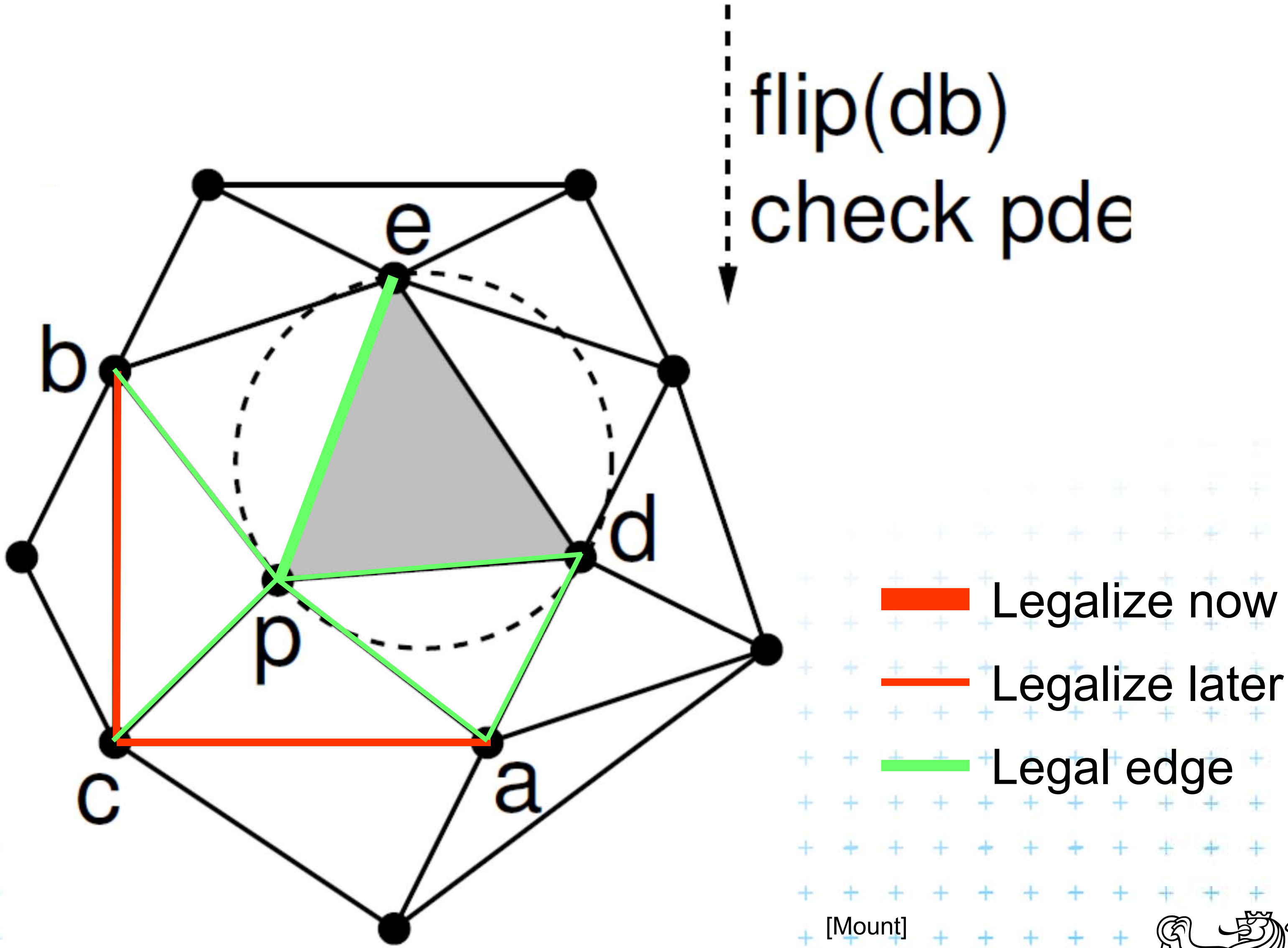
[Mount]

Delaunay triangulation – other point insert



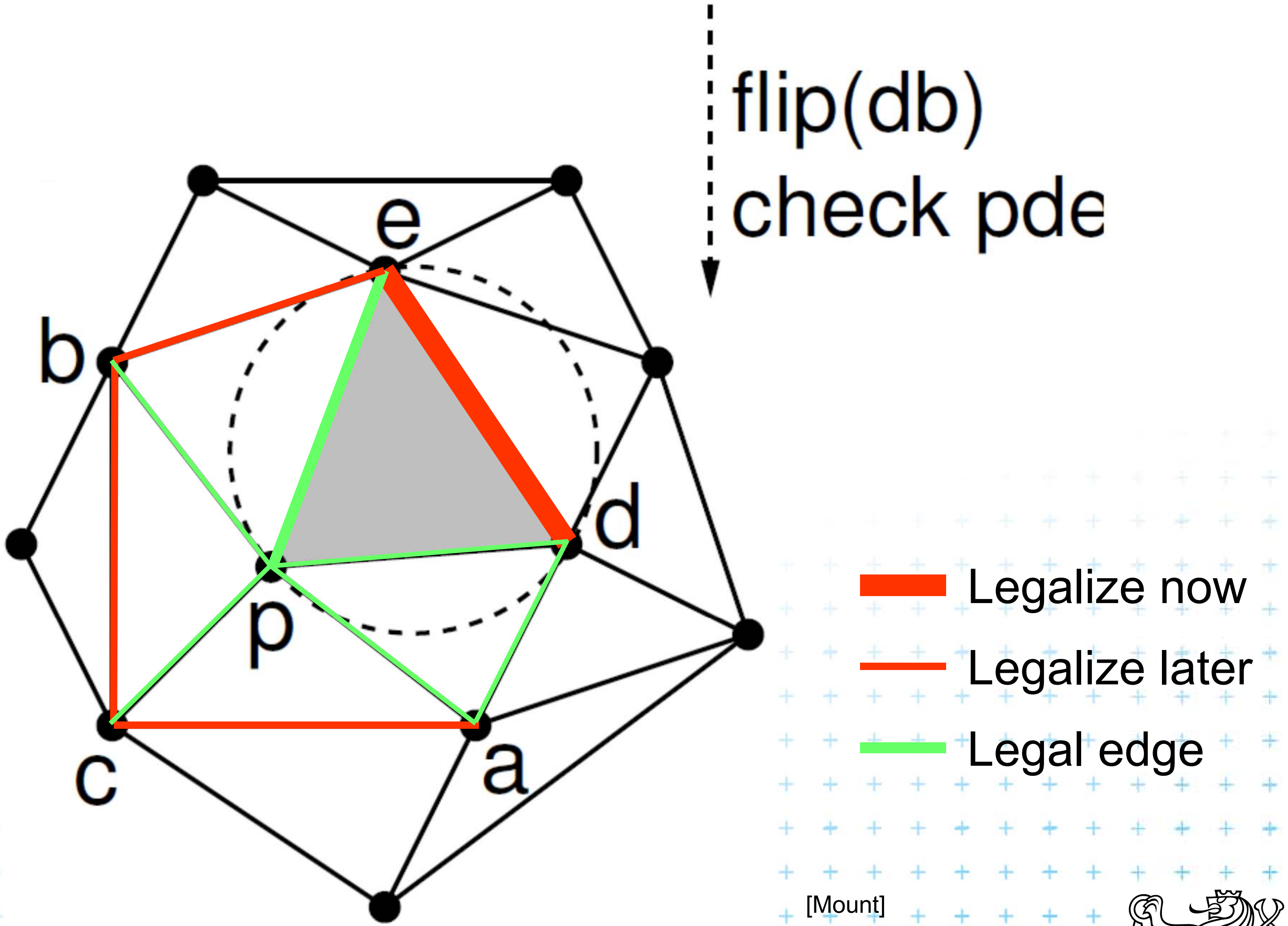
[Mount]

Delaunay triangulation – other point insert



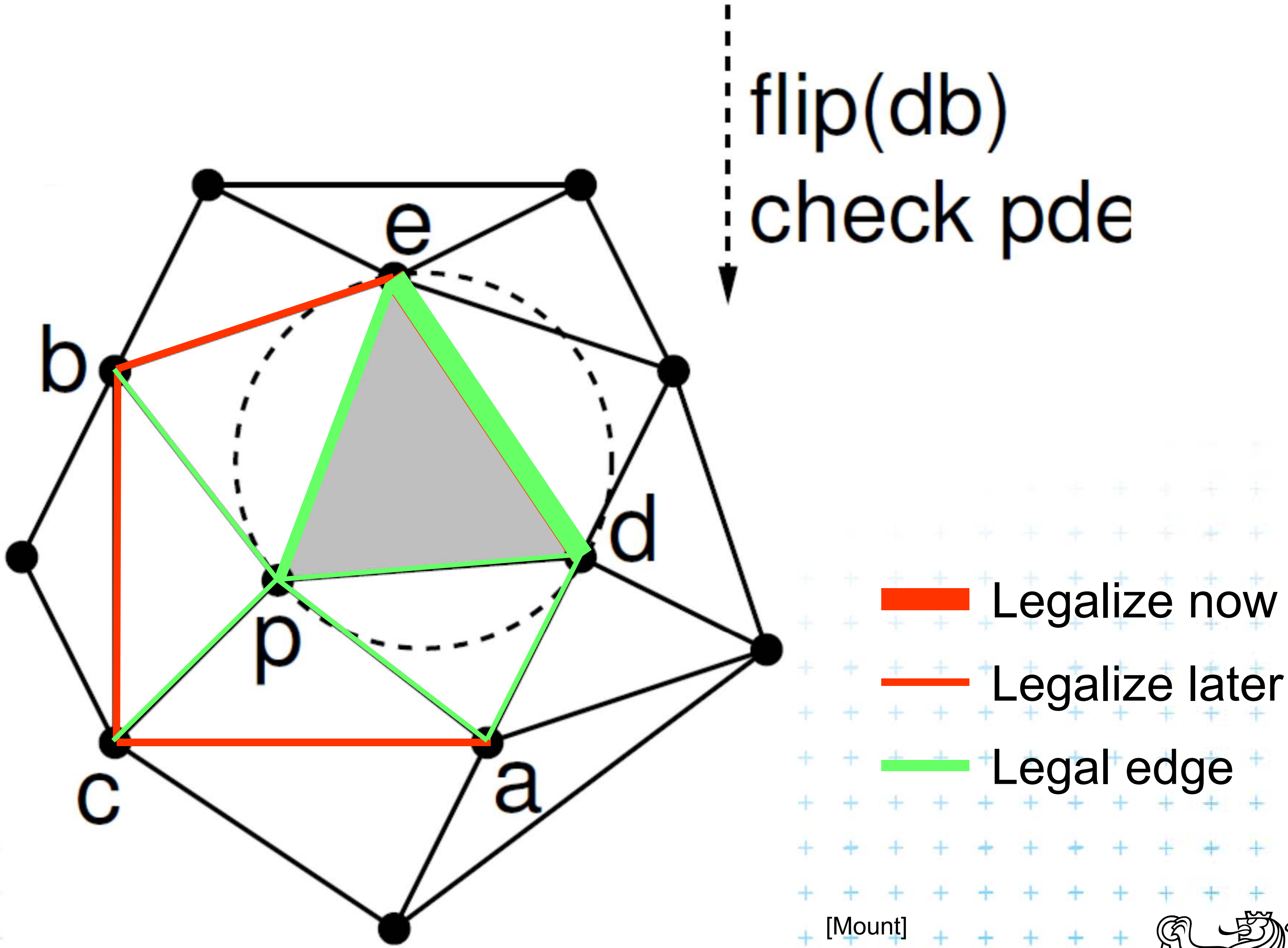
[Mount]

Delaunay triangulation – other point insert



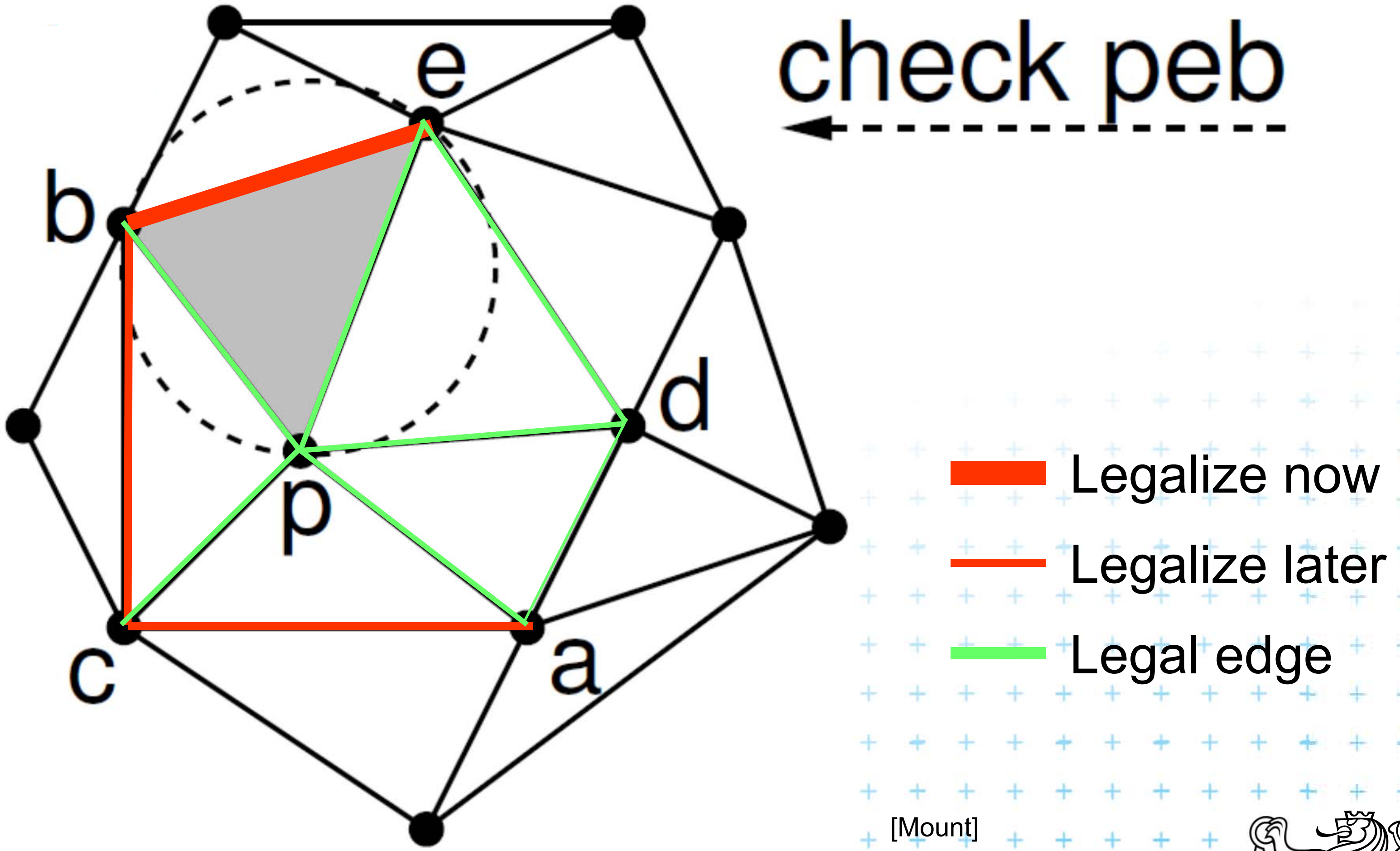
[Mount]

Delaunay triangulation – other point insert



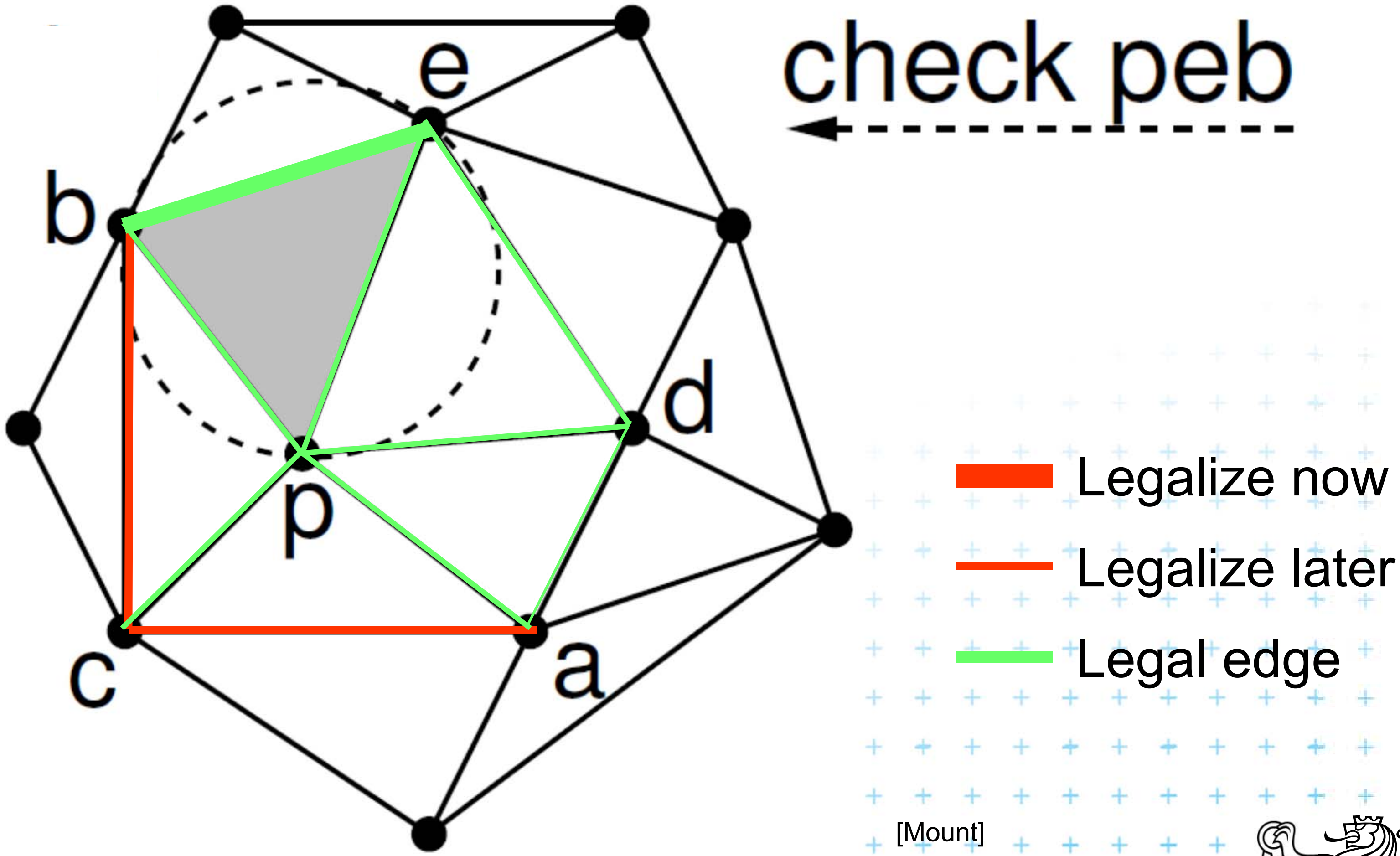
[Mount]

Delaunay triangulation – other point insert



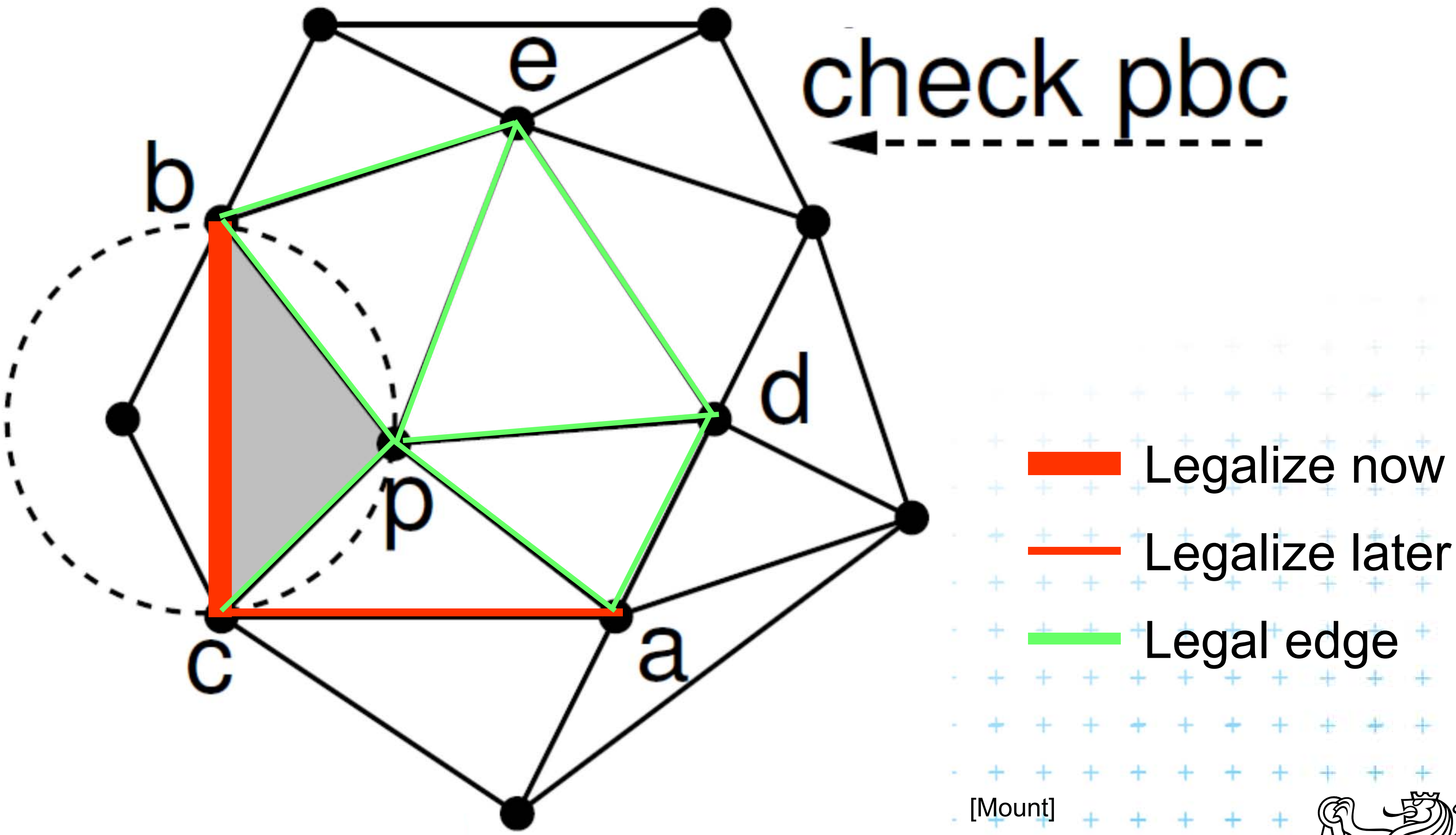
[Mount]

Delaunay triangulation – other point insert



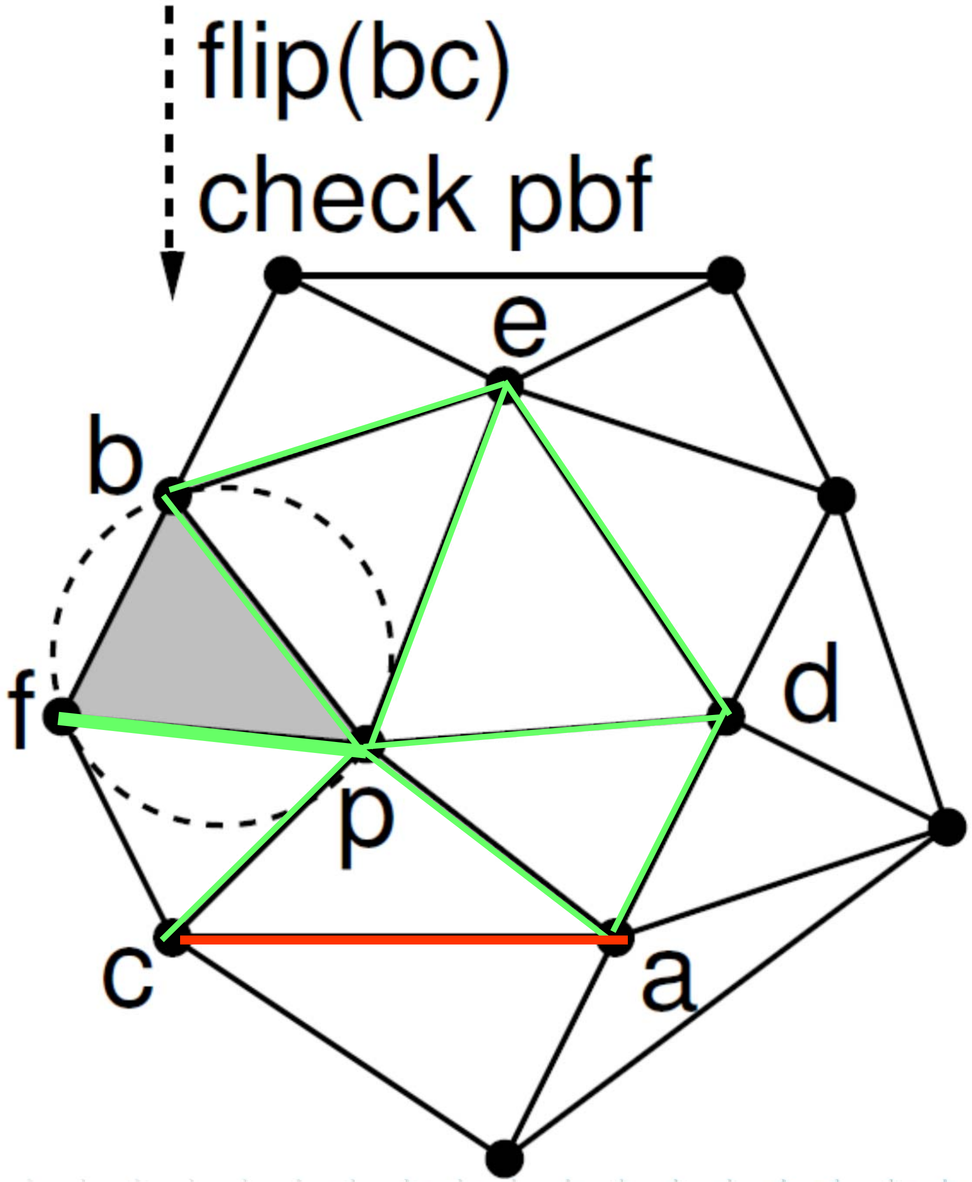
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



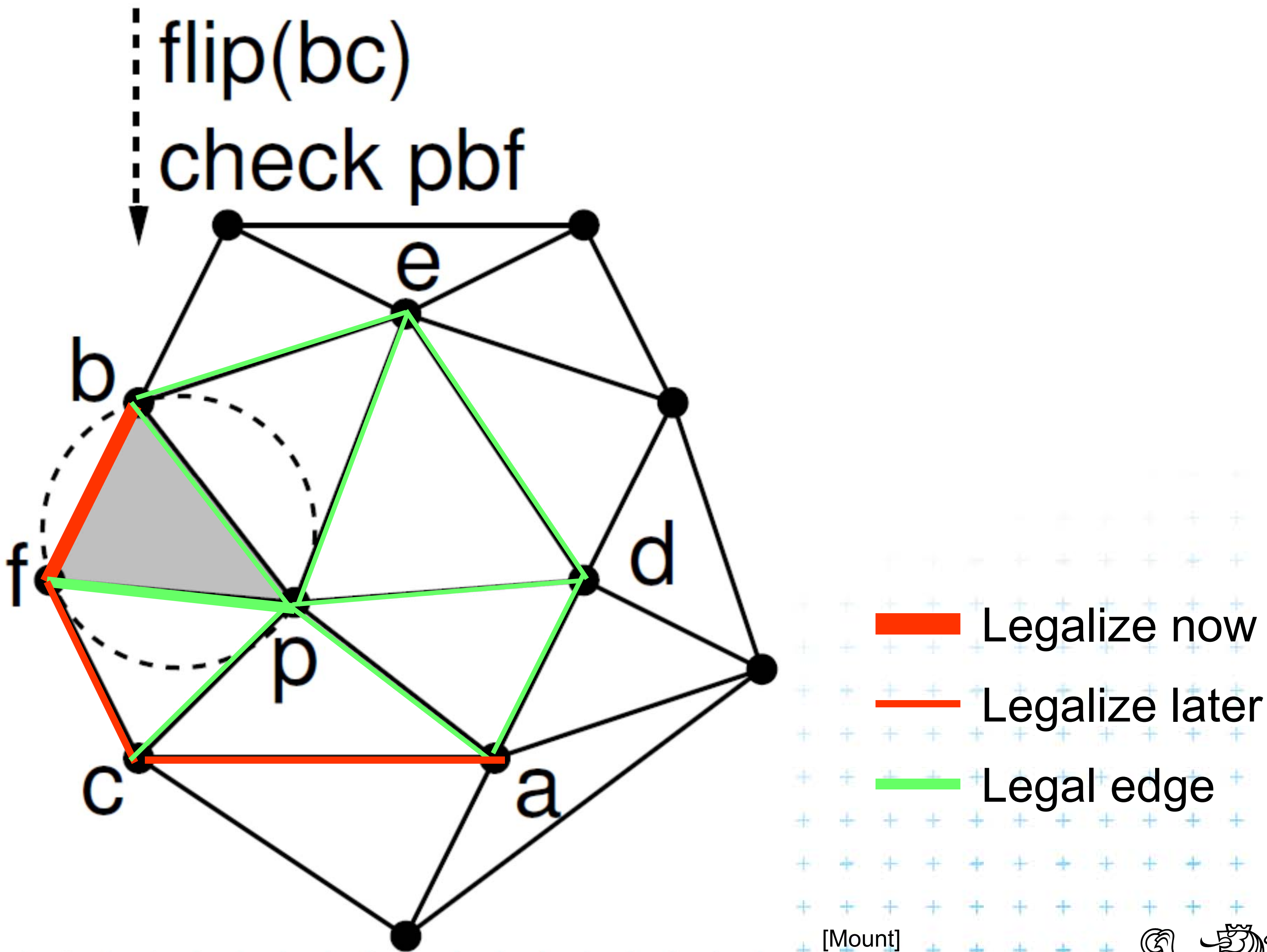
Legalize now

Legalize later

Legal edge

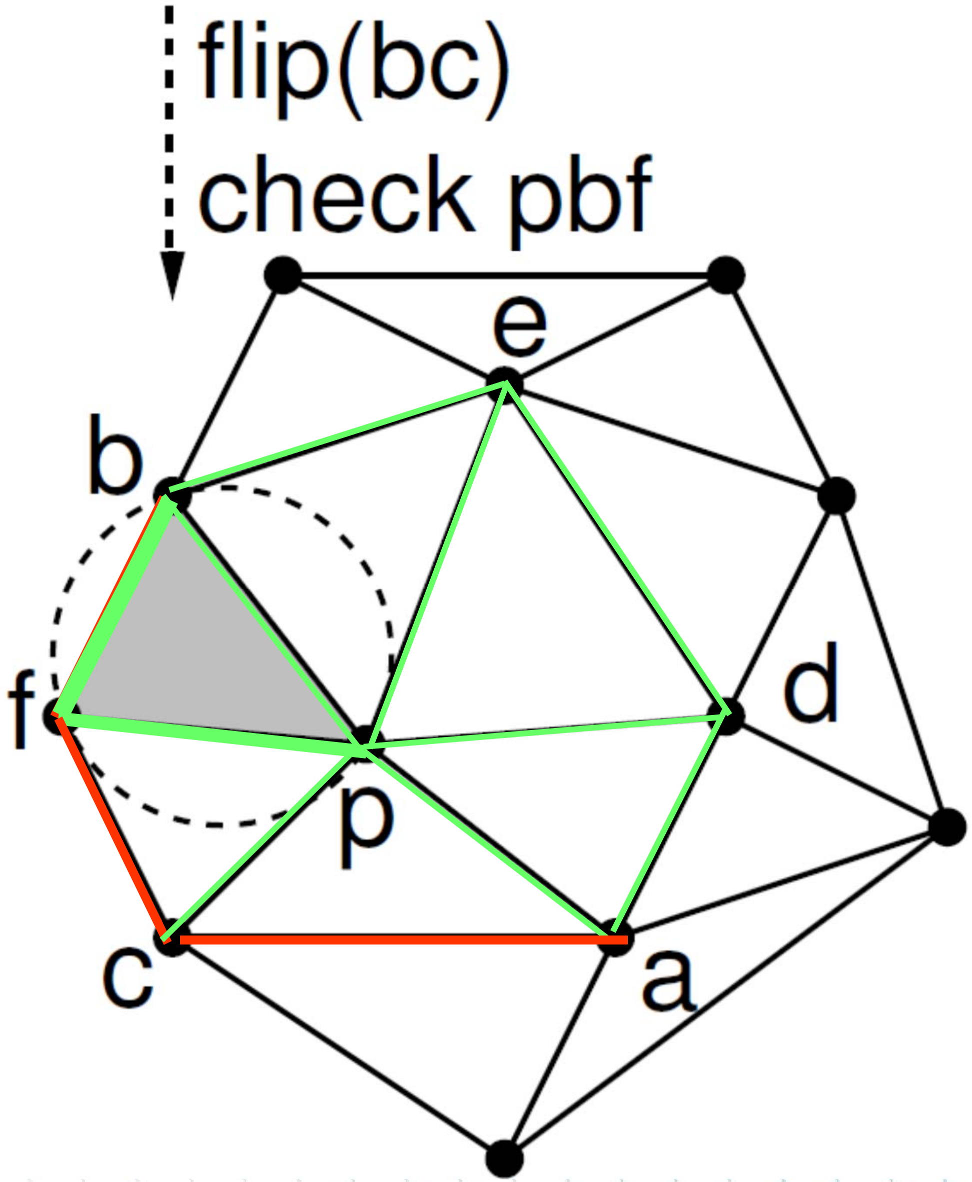
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



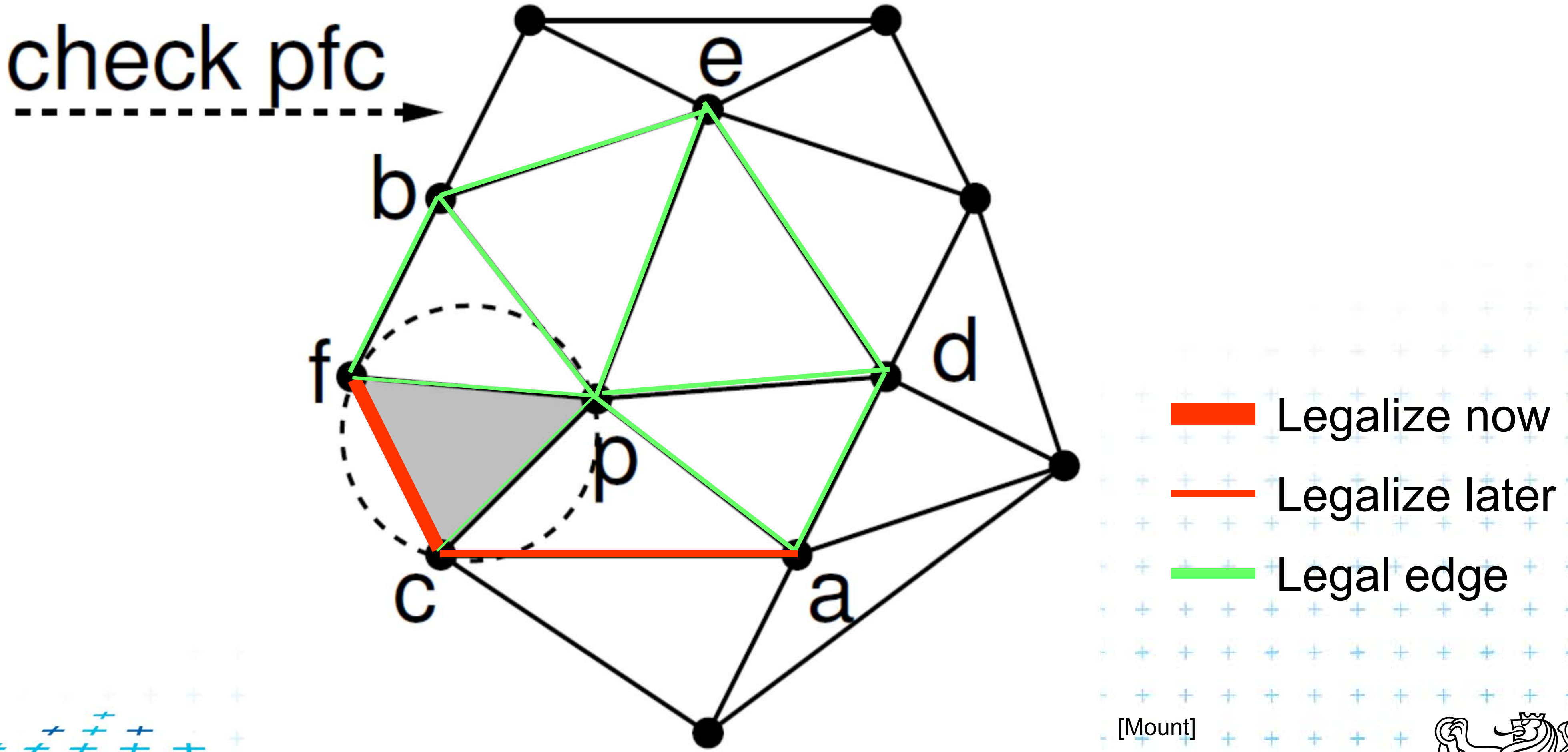
Legalize now

Legalize later

Legal edge

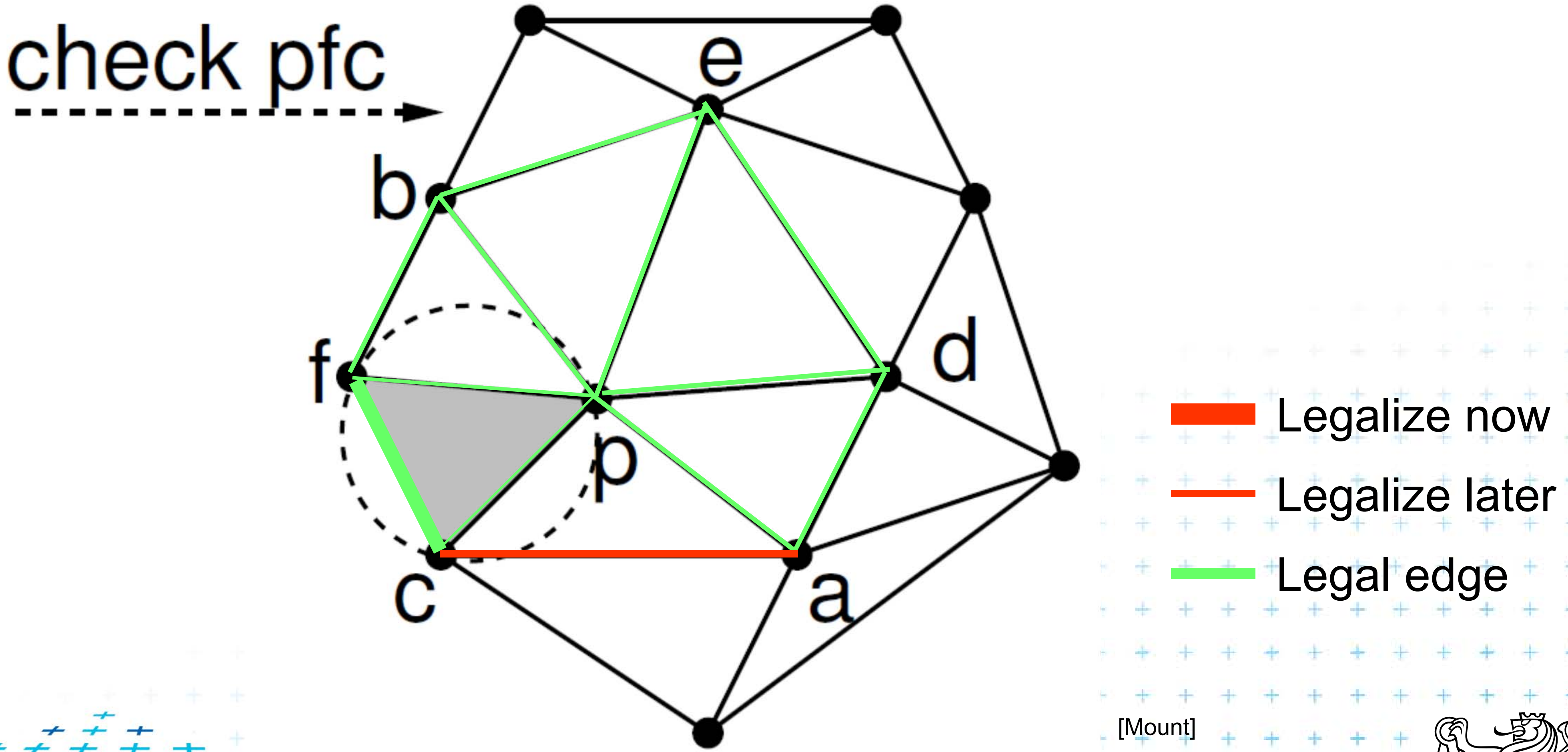
[Mount]

Delaunay triangulation – other point insert



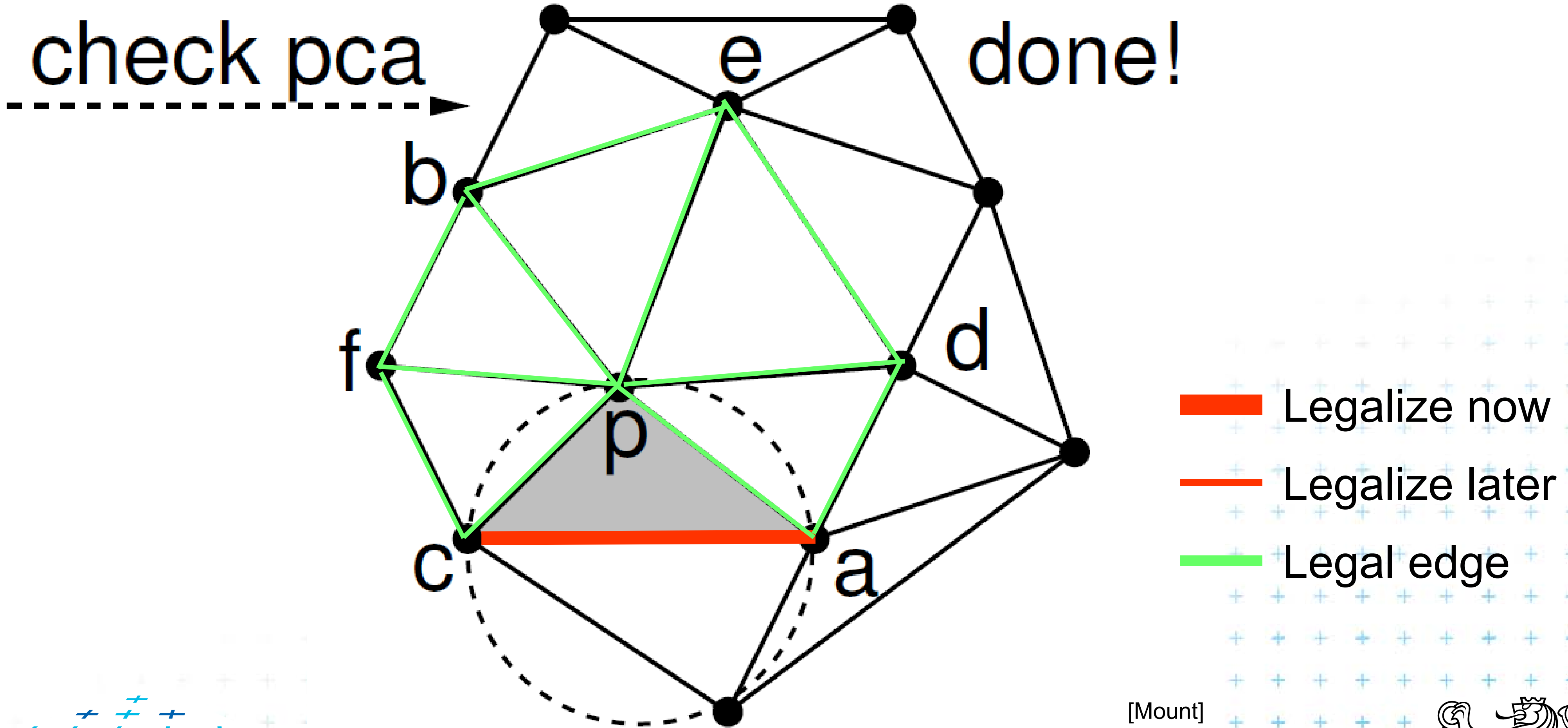
[Mount]

Delaunay triangulation – other point insert



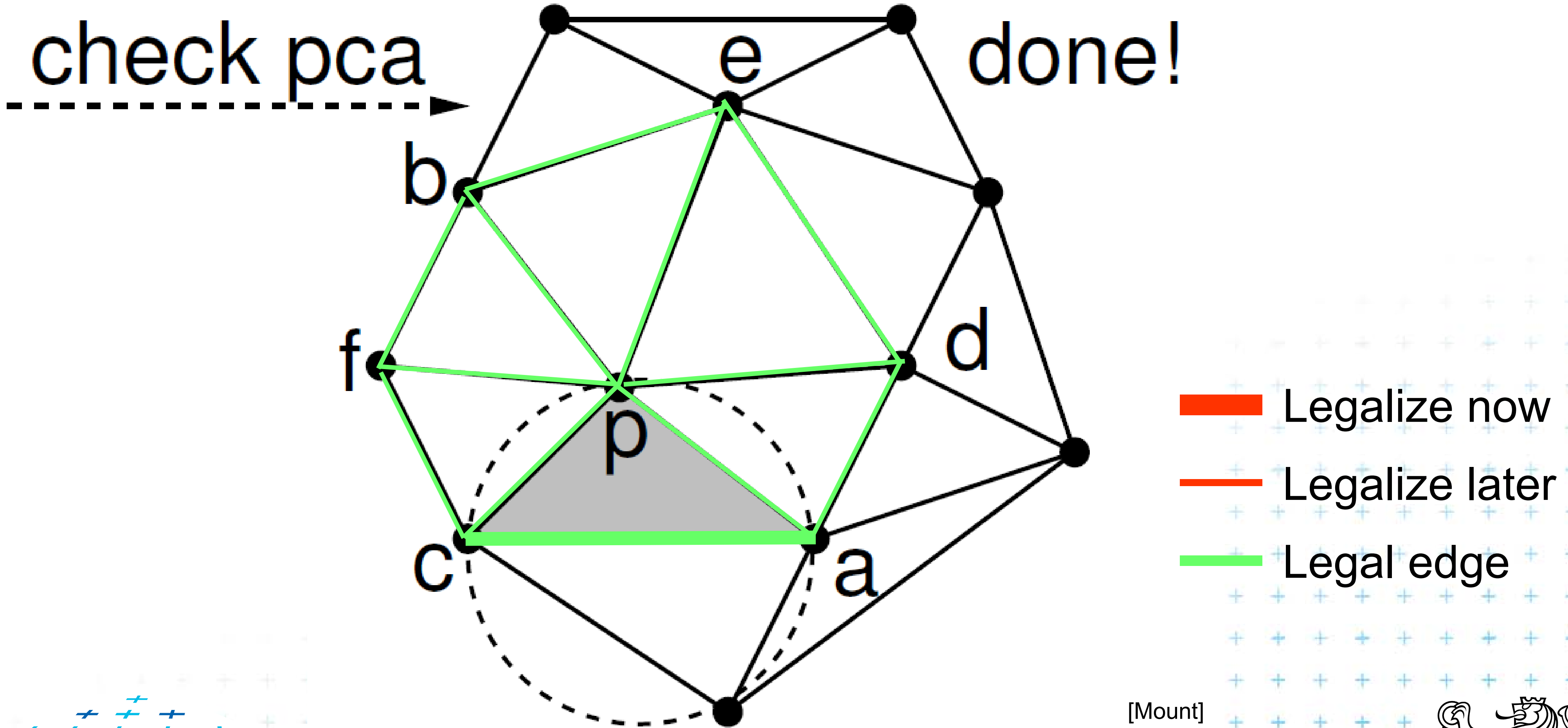
[Mount]

Delaunay triangulation – other point insert



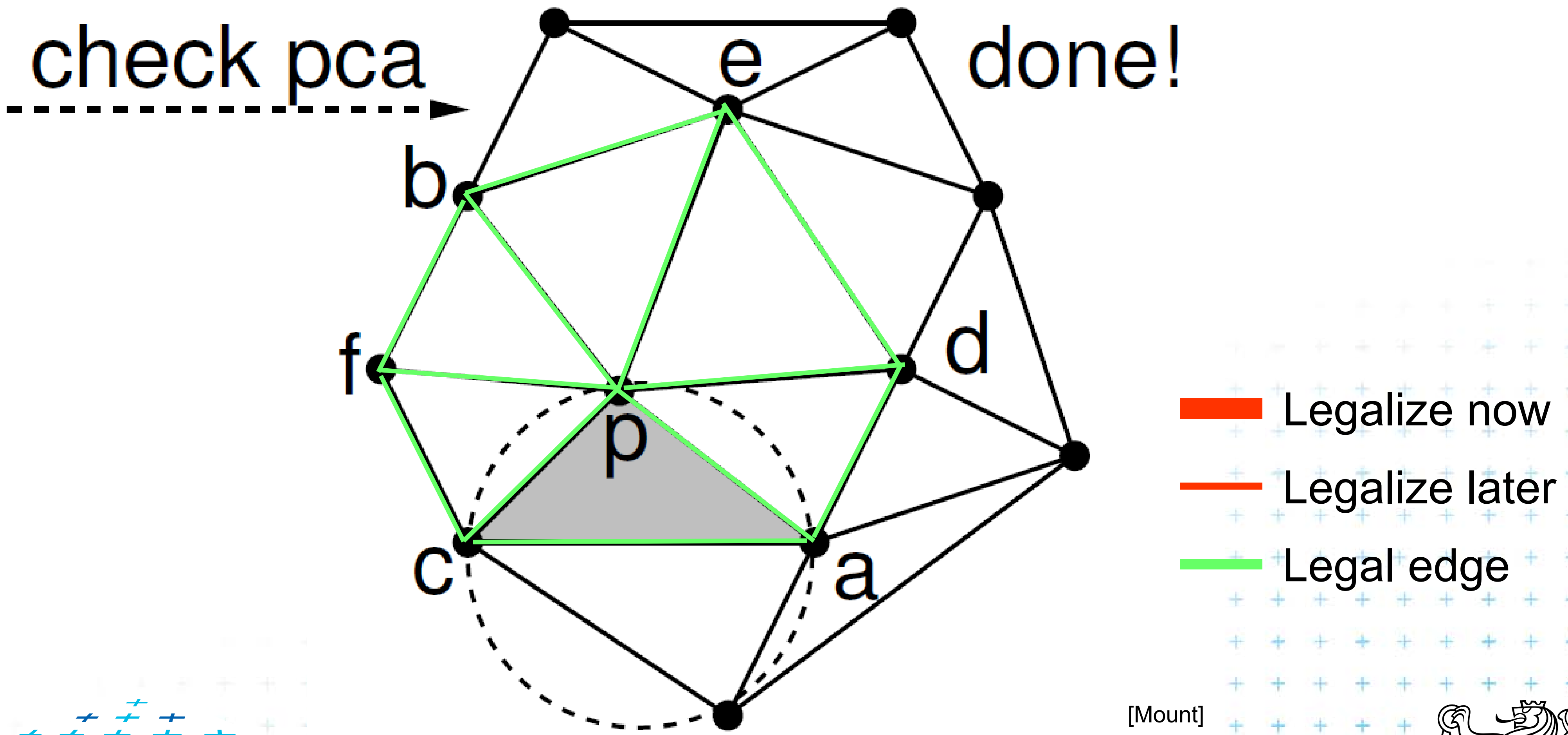
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



[Mount]

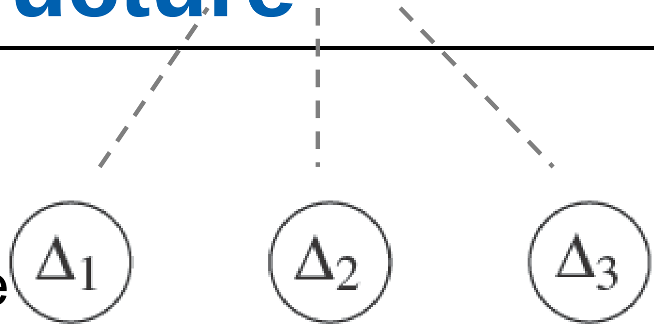
Correctness of the algorithm

- Every new edge (created due to insertion of *p*)
  - is incident to *p*
  - must be legal
    => no need to test them
- Edge can only become illegal if one of its incident triangle changes
  - Algorithm tests any edge that may become illegal
    => the algorithm is correct
- Every edge flip makes the angle-vector larger
  => algorithm can never get into infinite loop

Point location data structure

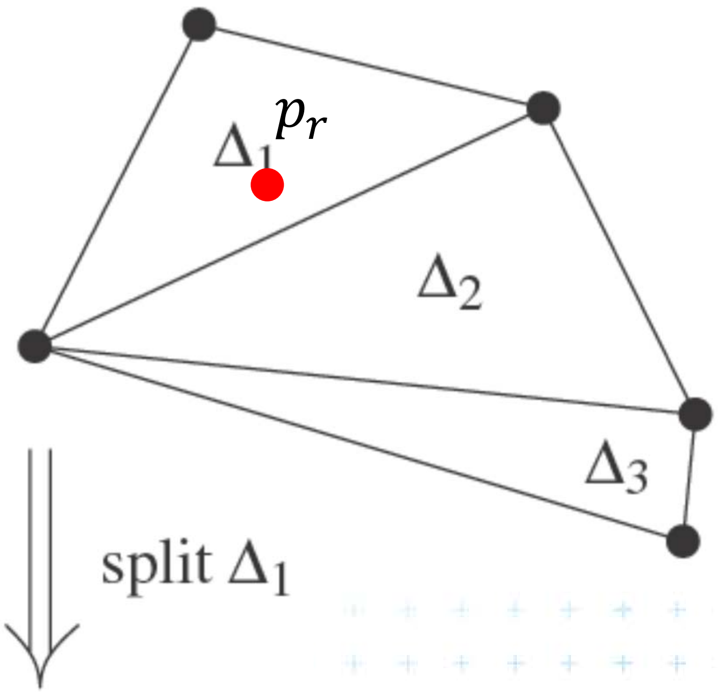
- For finding a triangle $abc \in T$ containing p
  - Leaves for active (current) triangles
  - Internal nodes for destroyed triangles
  - Links to new triangles
- Search p: start in root (initial triangle)
  - In each inner node of T:
    - Check all children (max three)
    - Descend to child containing p

Point location data structure



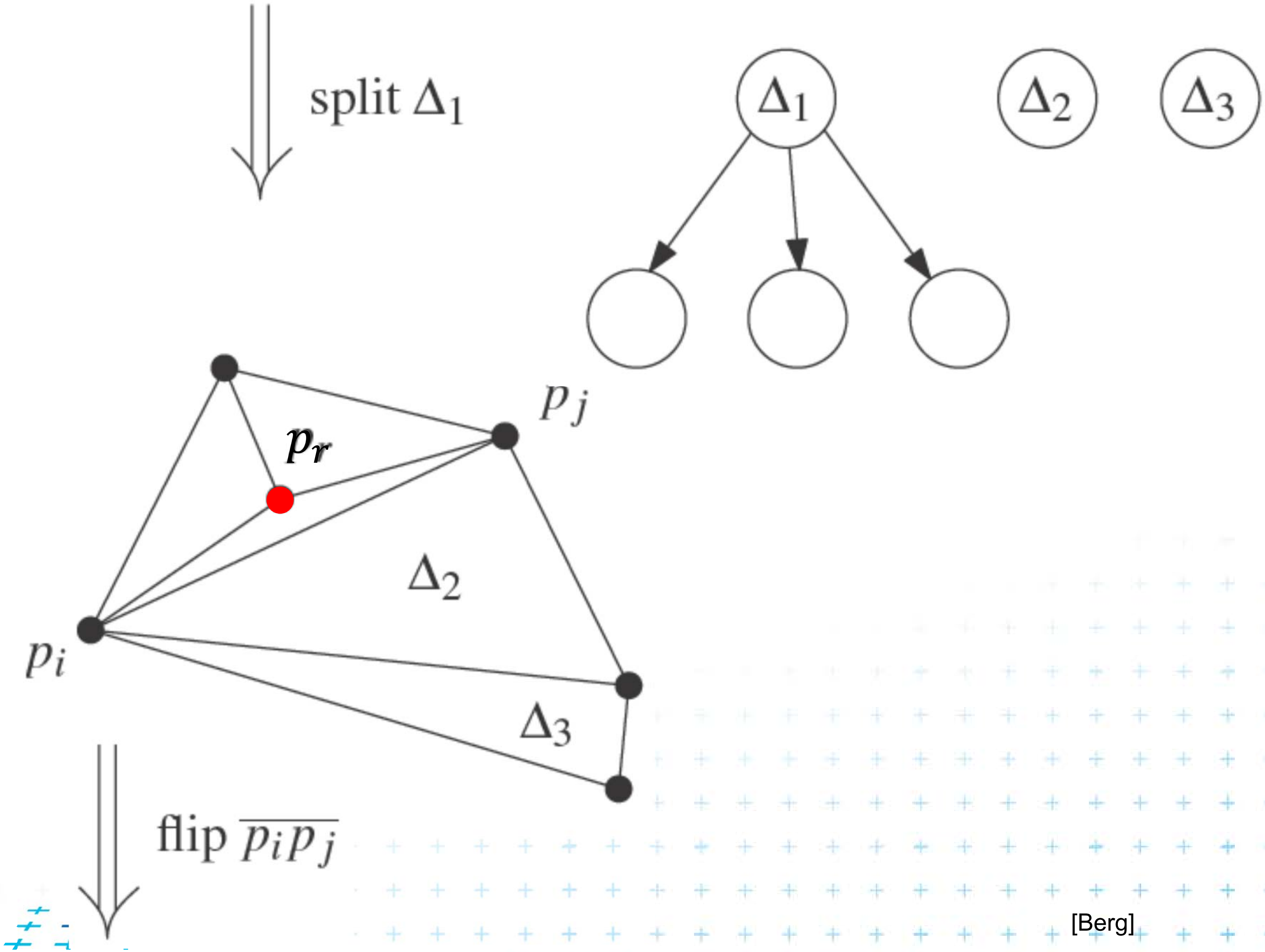
Simplified
- it should also contain the root node of the large triangle

New point p_r inserted to tr. 1



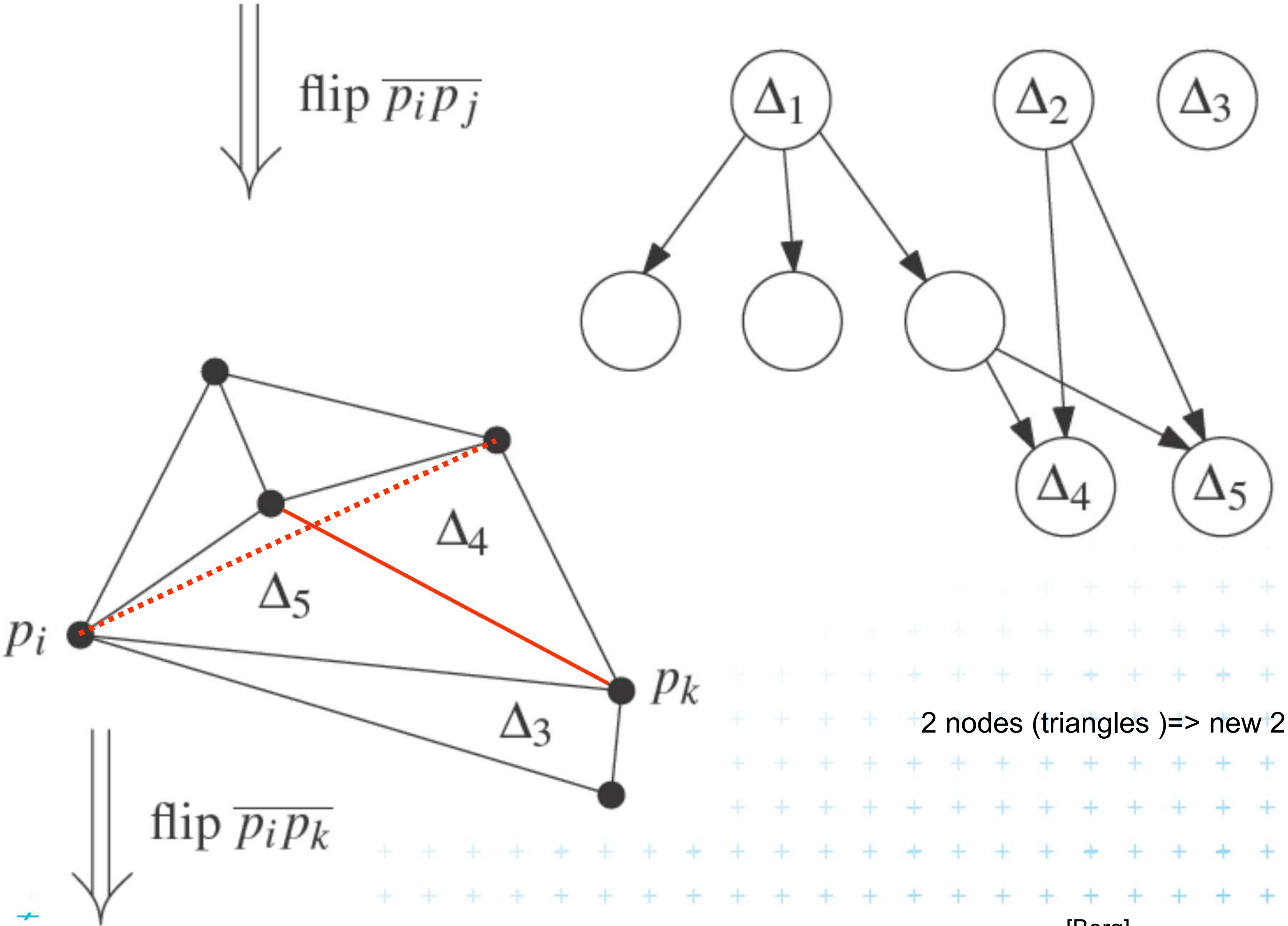
[Berg]

Point location data structure



[Berg]

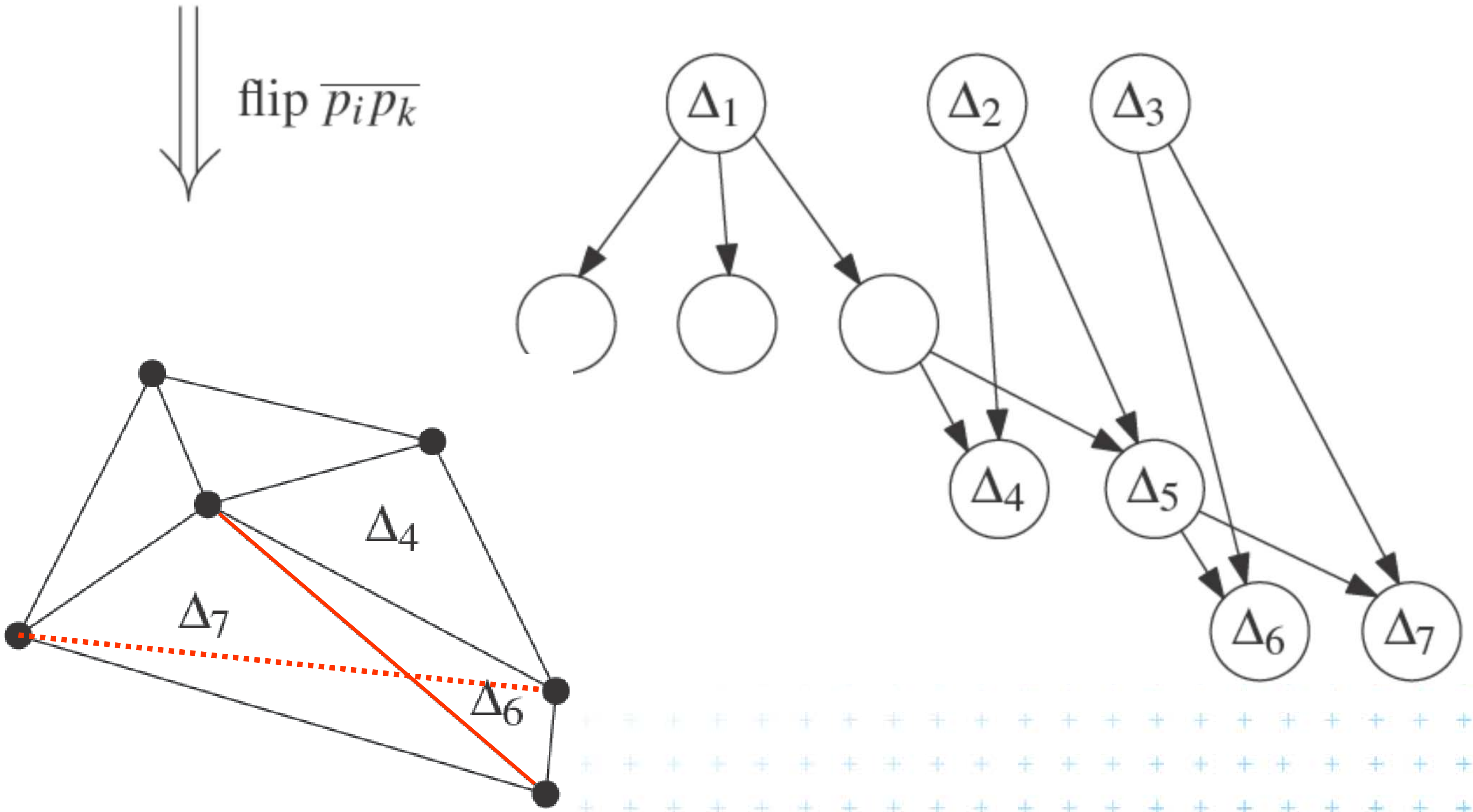
Point location data structure



2 nodes (triangles)=> new 2 nodes

[Berg]

Point location data structure

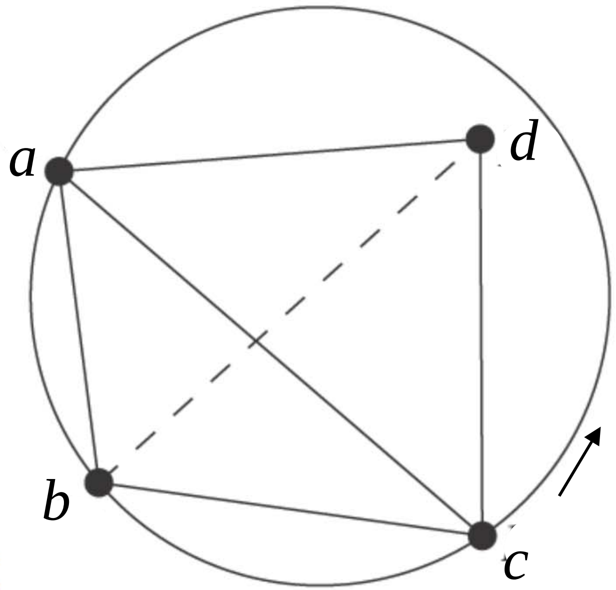


[Berg]

InCircle test

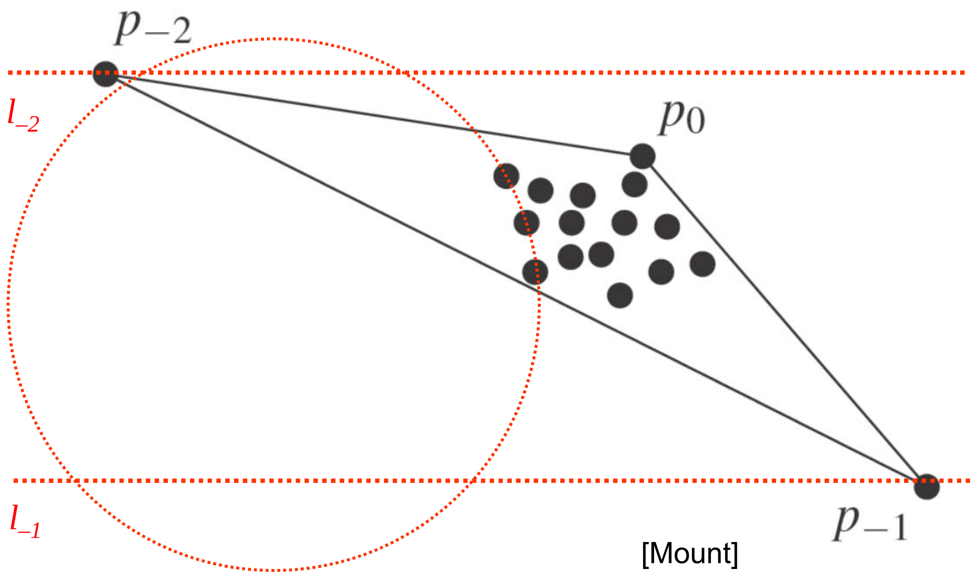
- *a,b,c* are counterclockwise in the plane
- Test, if *d* lies to the left of the oriented circle through *a,b,c*

$$\mathrm{inCircle}(a,b,c,d) = \det \begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{pmatrix} > 0$$



[Mount]

Creation of the initial triangle



Idea: For given points set *P:*

- Initial triangle $p_{-2}p_{-1}p_0$
  - Must contain all points of *P*
  - Must not be (none of its points) in any circle defined by non-collinear points of *P*
- l_{-2} = horizontal line above *P*
- l_{-1} = horizontal line below *P*
- p_{-2} = lies on l_{-2} as far left that p_{-2} lies outside every circle
- p_{-1} = lies on l_{-1} as far right that p_{-1} lies outside every circle defined by 3 non-collinear points of *P*

Replaced by symbolical tests with this triangle

=> p_{-1} and p_{-2} always out

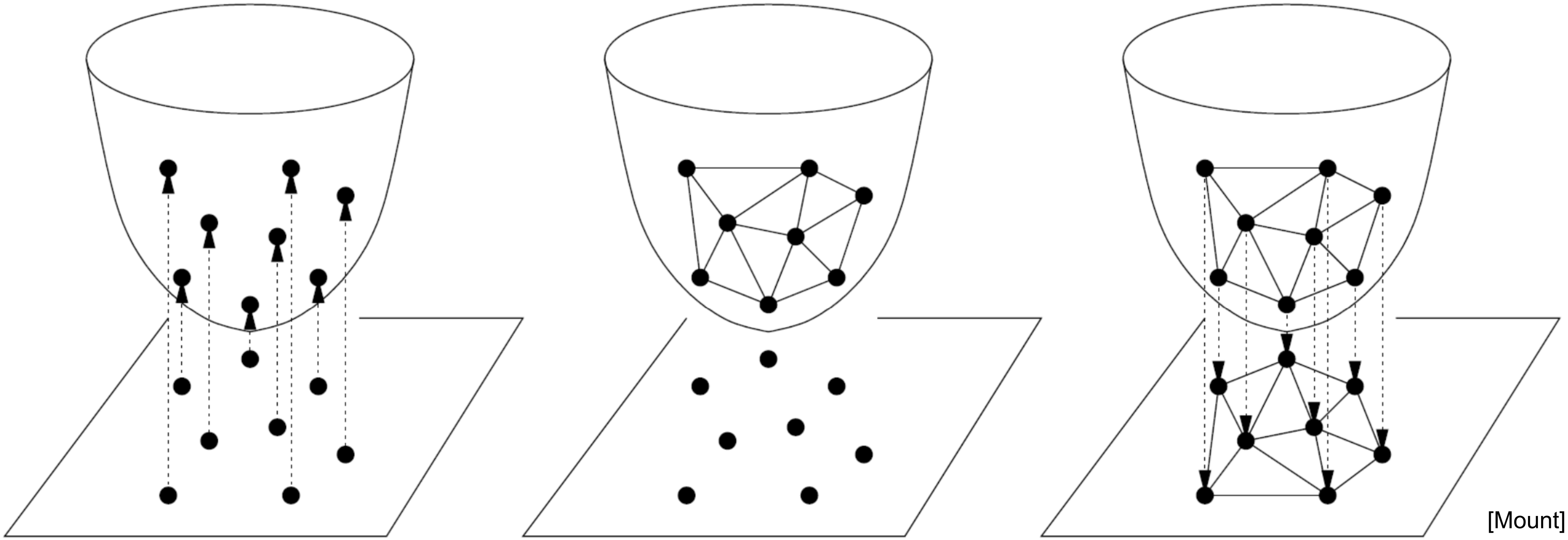
Complexity of incremental DT algorithm

- Delaunay triangulation of a pointset P in the plane can be computed in
  - $O(n \log n)$ expected time
  - using $O(n)$ storage
- For details see [Berg, Section 9.4]

  Idea
  - expected number of created triangles is $9n + 1$
  - expected search $O(\log n)$ in the search structure done n times for n inserted points

Delaunay triangulations and Convex hulls

- Delaunay triangulation in R^d can be computed as part of the convex hull in R^{d+1} (lower CH)
- 2D: Connection is the paraboloid: $z = x^2 + y^2$



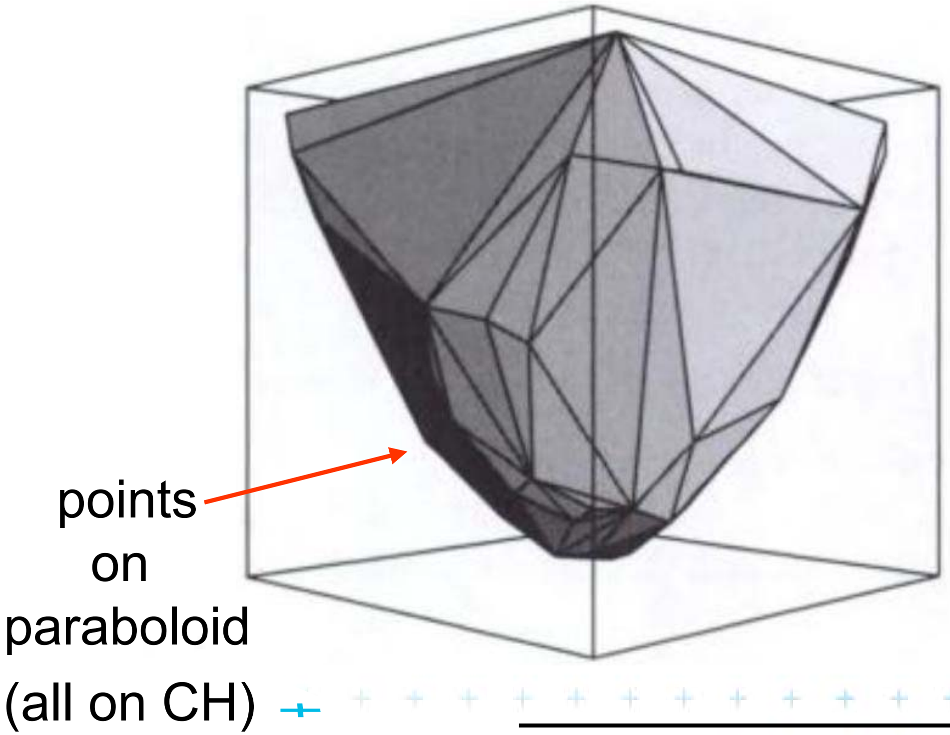
Project onto paraboloid. Compute convex hull. Project hull faces back to plane.

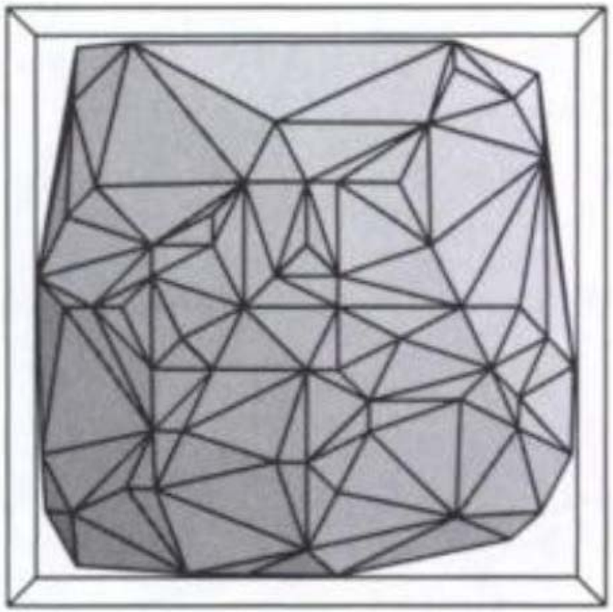
Vertical projection of points to paraboloid

- Vertical projection of 2D point to paraboloid in 3D

$$(x, y) \to (x, y, x^2 + y^2)$$

- Lower convex hull – forms Delone triangulation
  = portion of CH visible from $z = -\infty$





[Rourke]

Relation between CH and DT

Delaunay condition (2D)

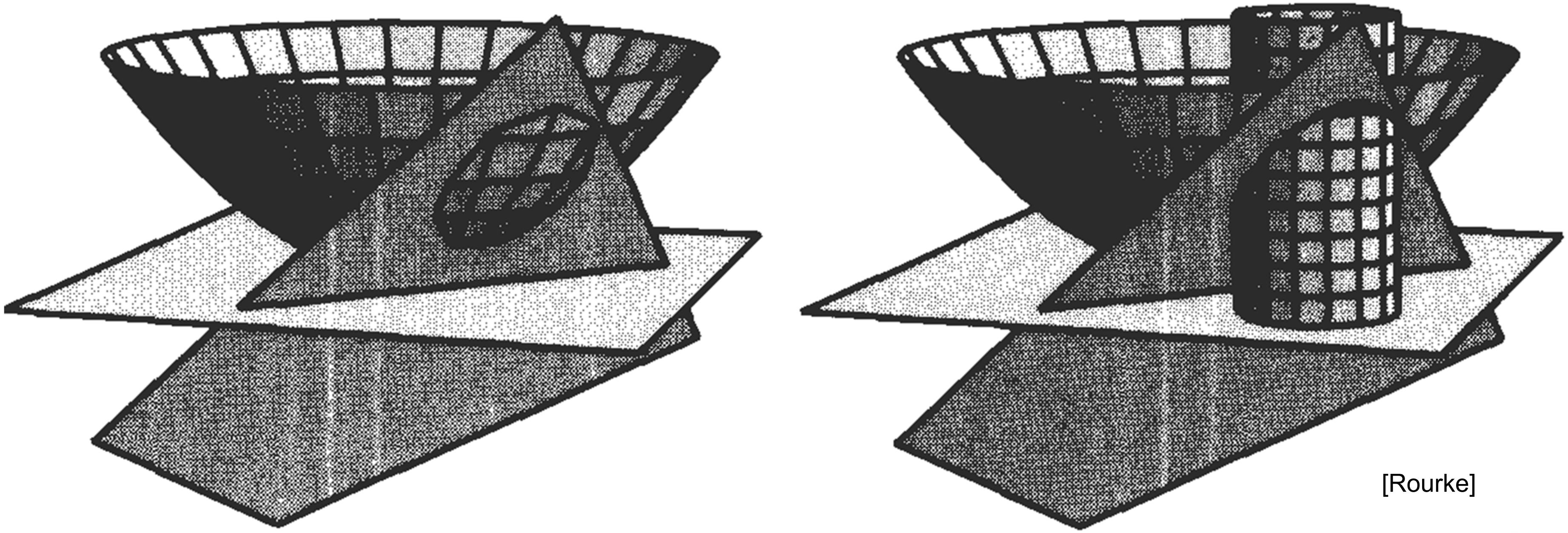
Points $p,q,r \in S$ form a Delone triangle **iff** the circumcircle of p,q,r is empty (contains no point)

Convex hull condition (3D)

Points $p',q',r' \in S'$ form a face of $CH(S')$ **iff** the plane passing through p',q',r' is supporting S'

- all other points lie to one side of the plane
- plane passing through p',q',r' is
  a supporting hyperplane of the convex hull CH(S')

Relation between CH and DT



[Rourke]

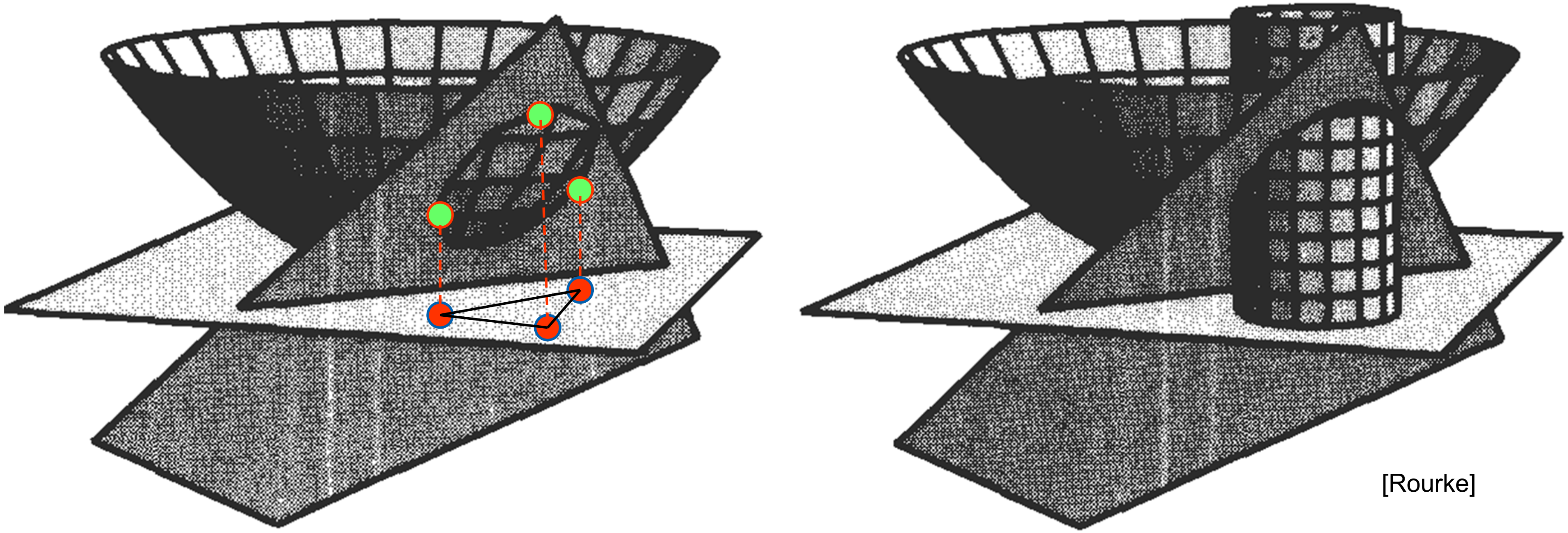
4 distinct points p, q, r, s in the plane, and

p', q', r', s' be their projections onto the paraboloid $z = x^2 + y^2$

The point s lies within the circumcircle of pqr **iff** s' lies on the lower side of the secant plane passing through p', q', r'

- Point s' cannot belong to CH, as the secant plane must be a supporting plane

Relation between CH and DT



[Rourke]

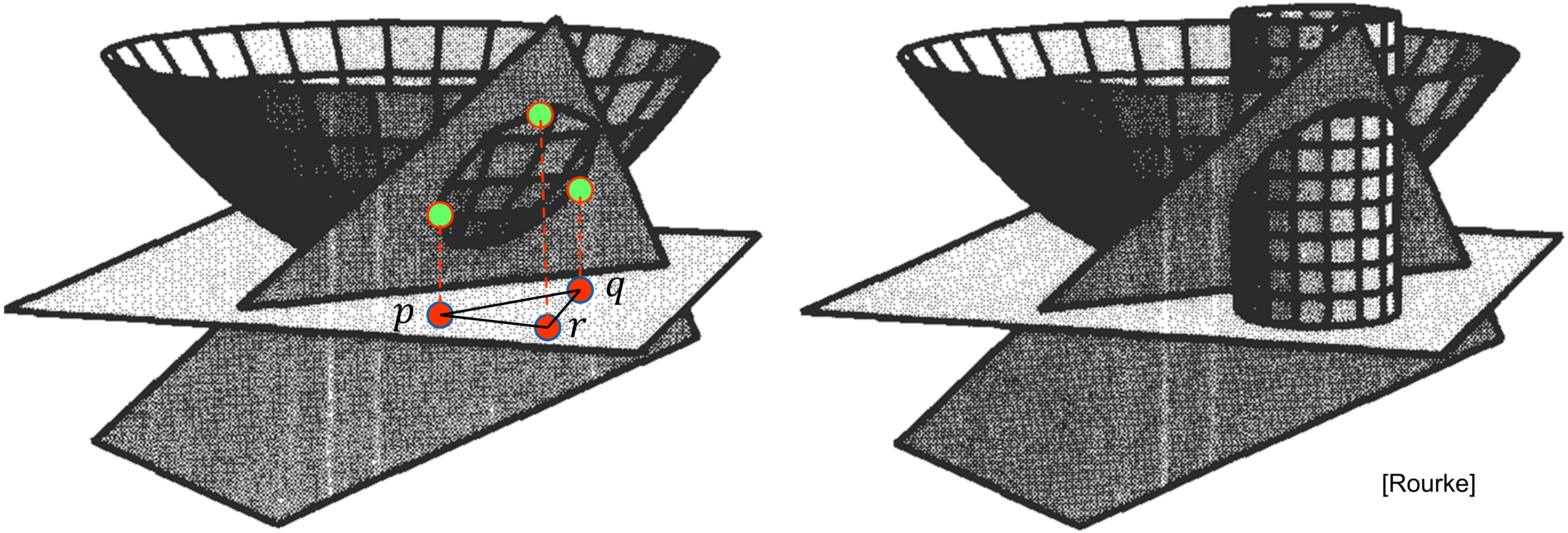
4 distinct points p, q, r, s in the plane, and

p', q', r', s' be their projections onto the paraboloid $z = x^2 + y^2$

The point s lies within the circumcircle of pqr **iff** s' lies on the lower side of the secant plane passing through p', q', r'

- Point s' cannot belong to CH, as the secant plane must be a supporting plane

Relation between CH and DT



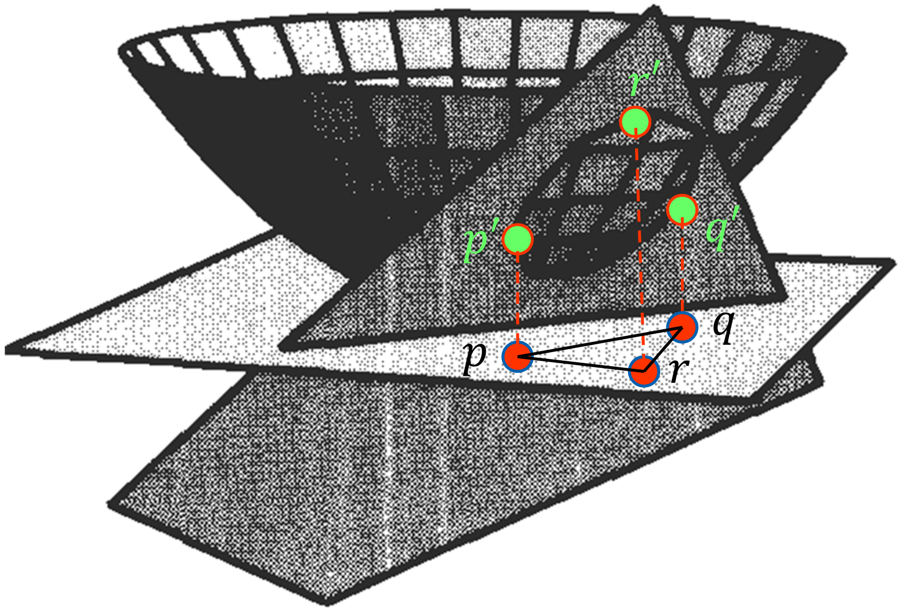
4 distinct points p, q, r, s in the plane, and

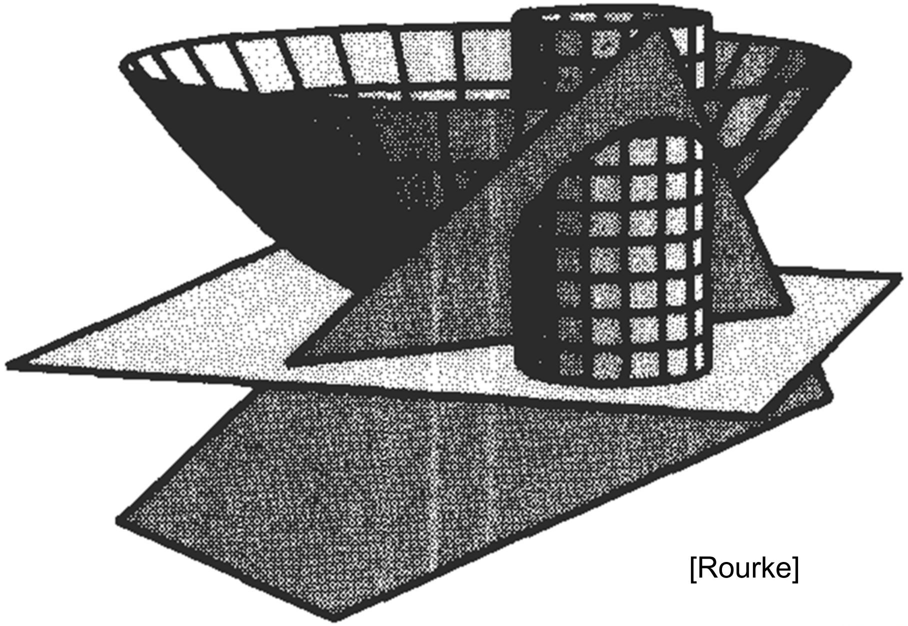
p', q', r', s' be their projections onto the paraboloid $z = x^2 + y^2$

The point s lies within the circumcircle of pqr **iff** s' lies on the lower side of the secant plane passing through p', q', r'

- Point s' cannot belong to CH, as the secant plane must be a supporting plane

Relation between CH and DT





[Rourke]

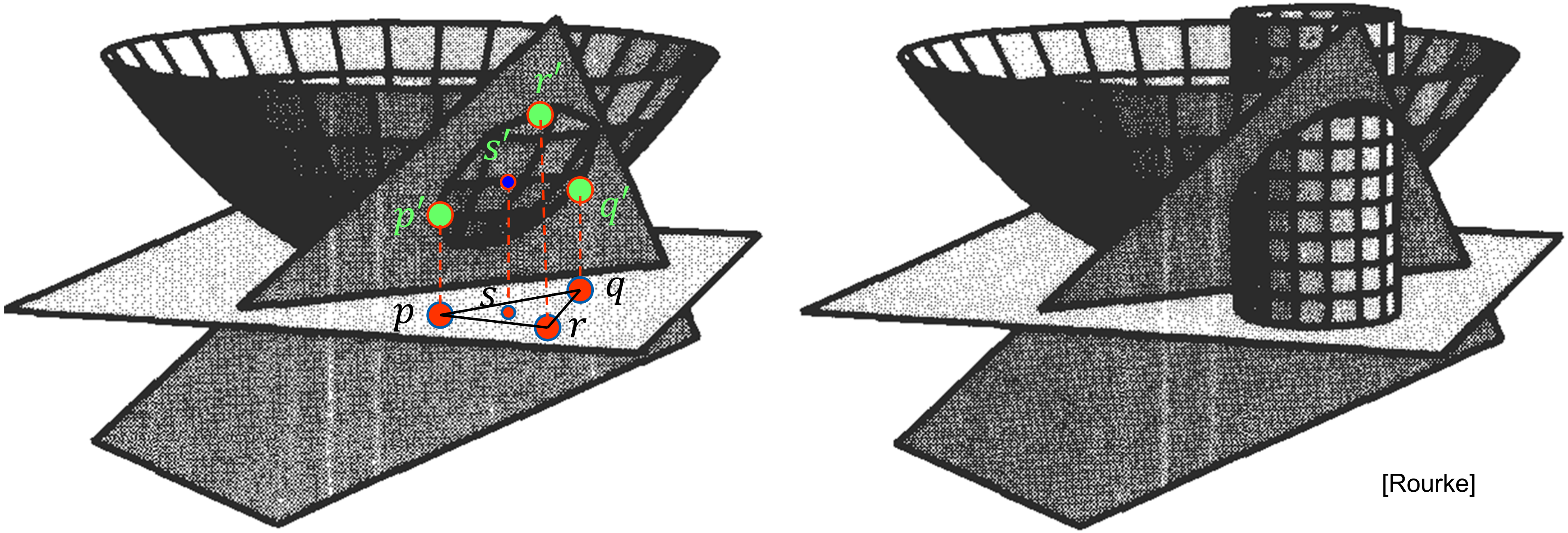
4 distinct points p, q, r, s in the plane, and

p', q', r', s' be their projections onto the paraboloid $z = x^2 + y^2$

The point s lies within the circumcircle of pqr **iff** s' lies on the lower side of the secant plane passing through p', q', r'

- Point s' cannot belong to CH, as the secant plane must be a supporting plane

Relation between CH and DT



[Rourke]

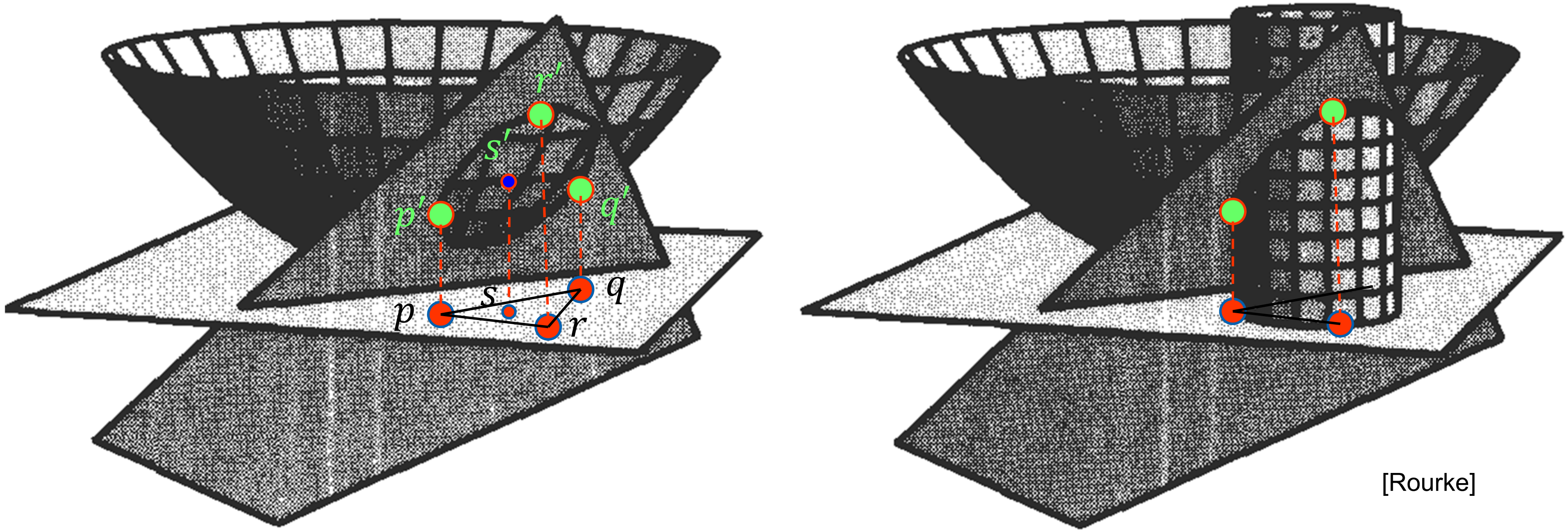
4 distinct points p, q, r, s in the plane, and

p', q', r', s' be their projections onto the paraboloid $z = x^2 + y^2$

The point s lies within the circumcircle of pqr **iff** s' lies on the lower side of the secant plane passing through p', q', r'

- Point s' cannot belong to CH, as the secant plane must be a supporting plane

Relation between CH and DT



[Rourke]

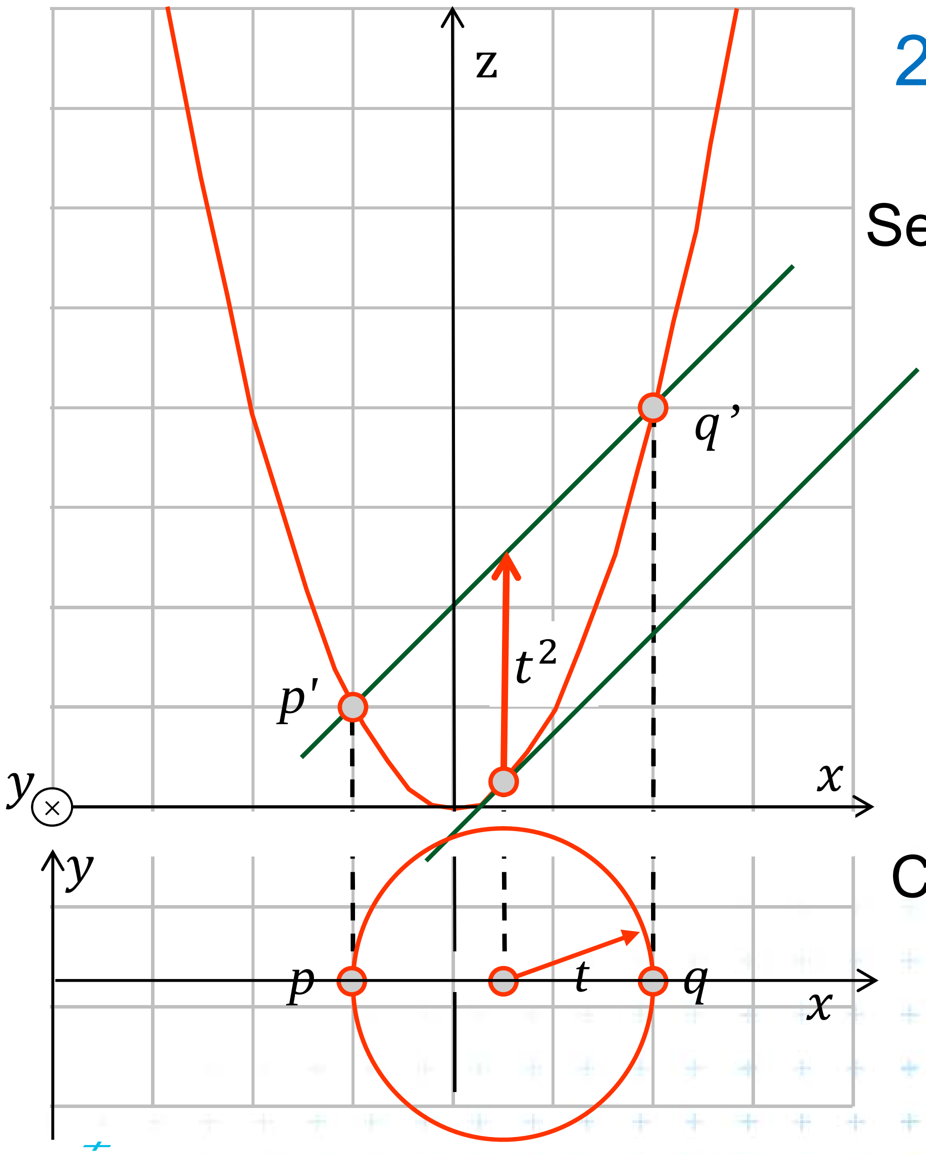
4 distinct points p, q, r, s in the plane, and

p', q', r', s' be their projections onto the paraboloid $z = x^2 + y^2$

The point s lies within the circumcircle of pqr **iff** s' lies on the lower side of the secant plane passing through p', q', r'

- Point s' cannot belong to CH, as the secant plane must be a supporting plane

Tangent and secant planes



2D cross section of the paraboloid

Secant plane

Tangent plane

Circle in xy plane with radius t

Tangent plane to paraboloid

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x$$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point $\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$
- Evaluates to

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point $\frac{\partial z}{\partial x} = 2x$ $\frac{\partial z}{\partial y} = 2y$
- Evaluates to $2a$ and $2b$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point $\frac{\partial z}{\partial x} = 2x$ $\frac{\partial z}{\partial y} = 2y$
- Evaluates to $2a$ and $2b$
- Plane:

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point $\frac{\partial z}{\partial x} = 2x \quad \frac{\partial z}{\partial y} = 2y$
- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point $\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$
- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$?

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$?

point

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point $\frac{\partial z}{\partial x} = 2x \quad \frac{\partial z}{\partial y} = 2y$
- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$?

point $a^2 + b^2 = 2a.a + 2b.b + \gamma$

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point $\frac{\partial z}{\partial x} = 2x \quad \frac{\partial z}{\partial y} = 2y$
- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$?

point $a^2 + b^2 = 2a.a + 2b.b + \gamma$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$?

point $a^2 + b^2 = 2a.a + 2b.b + \gamma$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$?

point $a^2 + b^2 = 2a.a + 2b.b + \gamma$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$?

$$\gamma = -(a^2 + b^2)$$

point $a^2 + b^2 = 2a.a + 2b.b + \gamma$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point $\frac{\partial z}{\partial x} = 2x \quad \frac{\partial z}{\partial y} = 2y$
- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$? $\quad \gamma = -(a^2 + b^2)$

point $a^2 + b^2 = 2a.a + 2b.b + \gamma$

- Tangent plane through point $(a, b, a^2 + b^2)$

[Mount]

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$? $\qquad \gamma = -(a^2 + b^2)$

point $a^2 + b^2 = 2a.a + 2b.b + \gamma$

- Tangent plane through point $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

[Mount]

Plane intersecting the paraboloid **(secant plane)**

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- (project to 2D)

- This is a circle projected to 2D with center (a, b):

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$
$z = 2ax + 2by - (a^2 + b^2)$

- Shift this plane t^2 upwards

- (project to 2D)

- This is a circle projected to 2D with center (a, b):

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- Shift this plane t^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

- (project to 2D)

- This is a circle projected to 2D with center (a, b):

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- Shift this plane t^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

$$z = 2ax + 2by - (a^2 + b^2) + t^2$$

- (project to 2D)

- This is a circle projected to 2D with center (a, b):

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- Shift this plane t^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

$$z = 2ax + 2by - (a^2 + b^2) + t^2$$

- Eliminate z (project to 2D)

- This is a circle projected to 2D with center (a, b):

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- Shift this plane t^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

$$z = 2ax + 2by - (a^2 + b^2) + t^2$$

- Eliminate z (project to 2D) $z = x^2 + y^2$

- This is a circle projected to 2D with center (a, b):

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- Shift this plane t^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

$$z = 2ax + 2by - (a^2 + b^2) + t^2$$

- Eliminate z (project to 2D) $z = x^2 + y^2$

$$x^2 + y^2 = 2ax + 2by - (a^2 + b^2) + t^2$$

- This is a circle projected to 2D with center (a, b):

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- Shift this plane t^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

$$z = 2ax + 2by - (a^2 + b^2) + t^2$$

- Eliminate z (project to 2D) $z = x^2 + y^2$

$$x^2 + y^2 = 2ax + 2by - (a^2 + b^2) + t^2$$

- This is a circle projected to 2D with center (a, b):

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- Shift this plane t^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

$$z = 2ax + 2by - (a^2 + b^2) + t^2$$

- Eliminate z (project to 2D) $z = x^2 + y^2$

$$x^2 + y^2 = 2ax + 2by - (a^2 + b^2) + t^2$$

- This is a circle projected to 2D with center (a, b):

$$(x - a)^2 + (y - b)^2 = t^2$$

[Mount]

Plane intersecting the paraboloid (secant plane)

Non-vertical tangent plane through $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

- Shift this plane t^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

$$z = 2ax + 2by - (a^2 + b^2) + t^2$$

- Eliminate z (project to 2D) $z = x^2 + y^2$

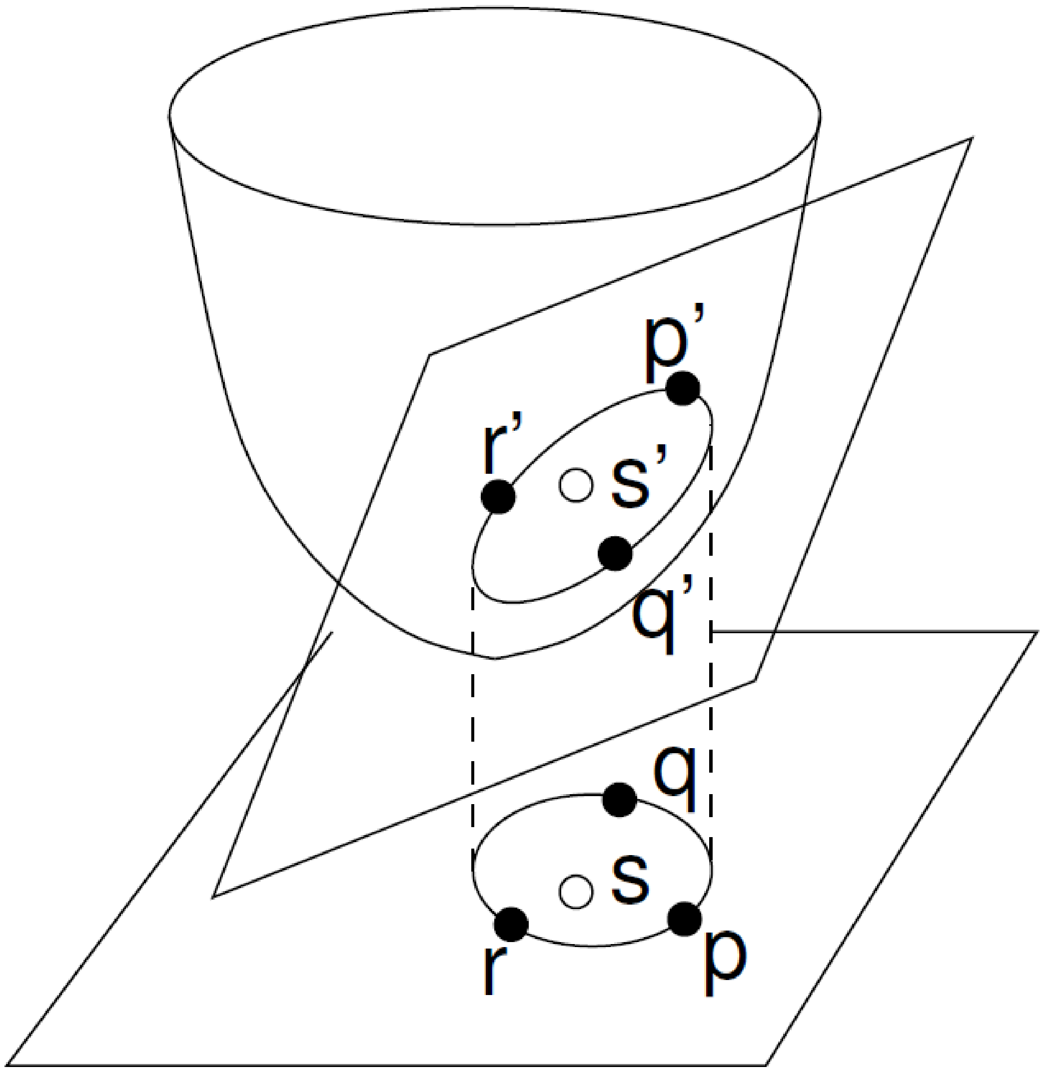
$$x^2 + y^2 = 2ax + 2by - (a^2 + b^2) + t^2$$

- This is a circle projected to 2D with center (a, b):

$(x - a)^2 + (y - b)^2 = t^2$ and radius t

[Mount]

Secant plane defined by three points



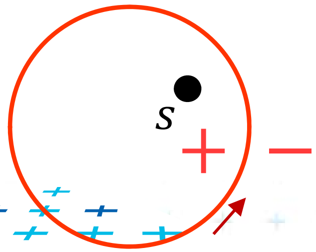
DT in 2D → CH in 3D

[Mount]

Test inCircle – meaning in 3D

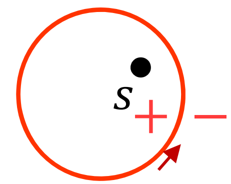
- Points *p,q,r* are counterclockwise in the plane
- Test, if *s* lies in the circumcircle of △*pqr* is equal to
  - = test, weather *s'* lies within a lower half space of the plane passing through *p',q',r'* (3D)
  - = test, if quadruple *p',q',r',s'* is positively oriented (3D)
  - = test, if *s lies* to the left of the oriented circle through *pqr* (2D)

$$\mathrm{in}(p,q,r,s) \;=\; \det\begin{pmatrix} p_x & p_y & p_x^2+p_y^2 & 1 \\ q_x & q_y & q_x^2+q_y^2 & 1 \\ r_x & r_y & r_x^2+r_y^2 & 1 \\ s_x & s_y & s_x^2+s_y^2 & 1 \end{pmatrix} > 0.$$



[Mount]

Delaunay triangulation and inCircle test



- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

inCircle(…) > 0

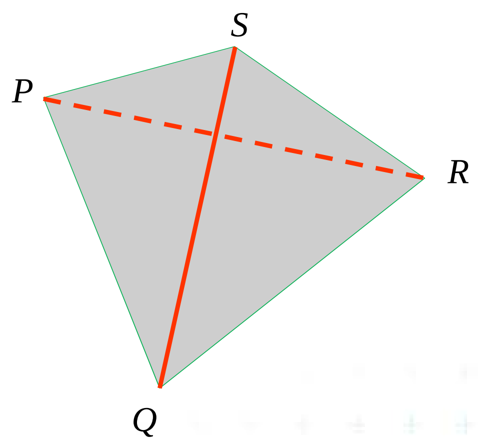
P - - - - - - - - R

Invalid diagonal

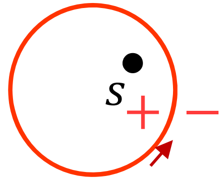
inCircle(…) < 0

S ———————— Q

Valid diagonal



Delaunay triangulation and inCircle test



- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

inCircle(…) > 0

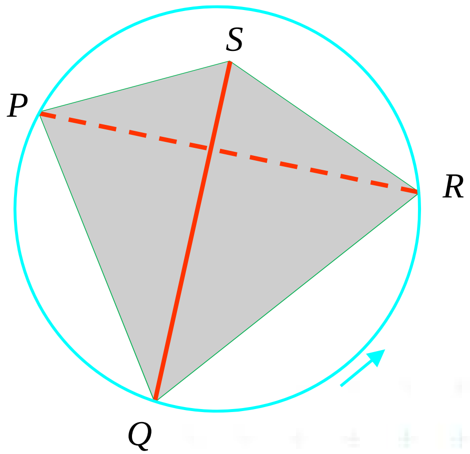
P - - - - - - - - - R

Invalid diagonal

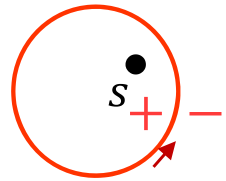
inCircle(…) < 0

S ———— Q

Valid diagonal



Delaunay triangulation and inCircle test



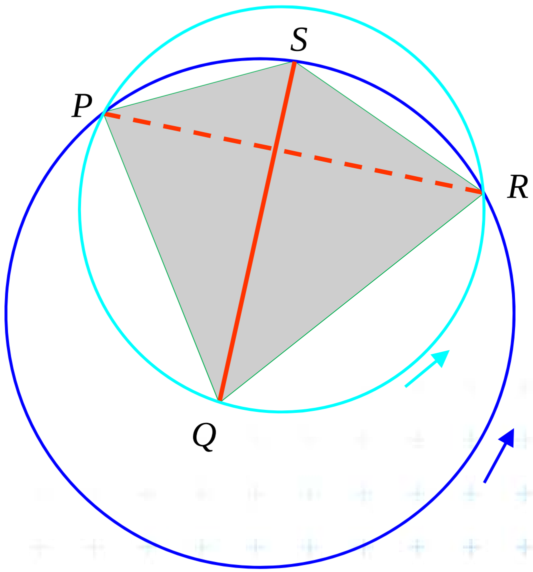
- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

inCircle(…) > 0

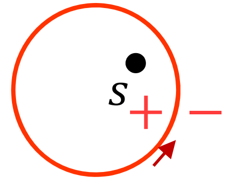
Invalid diagonal

inCircle(…) < 0

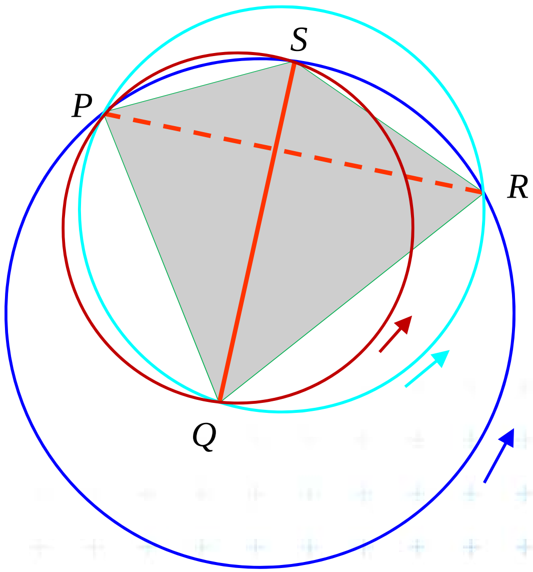
Valid diagonal



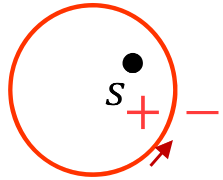
Delaunay triangulation and inCircle test



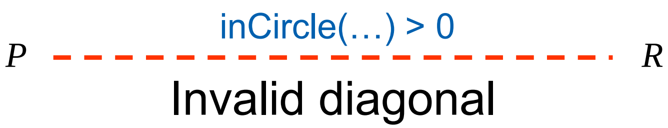
- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

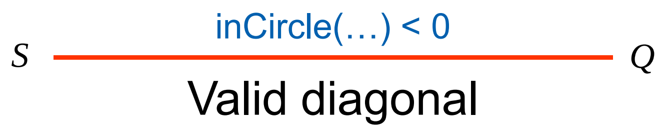


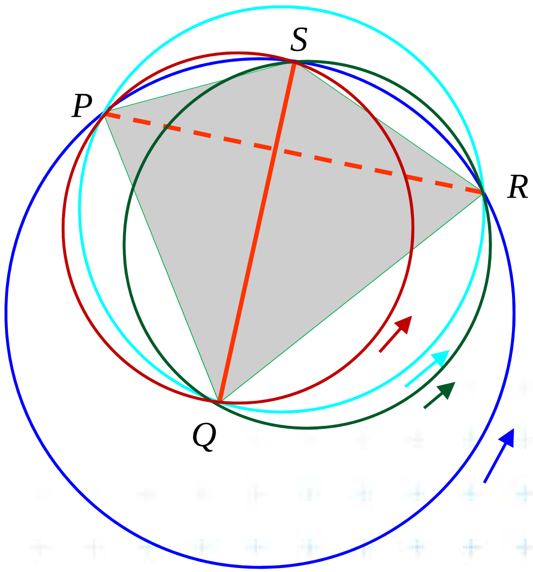
Delaunay triangulation and inCircle test



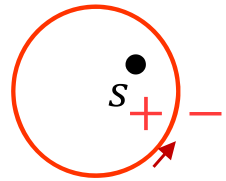
- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)







Delaunay triangulation and inCircle test



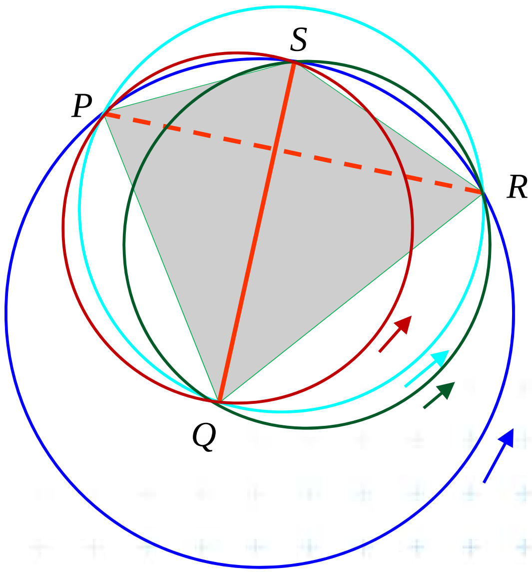
- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

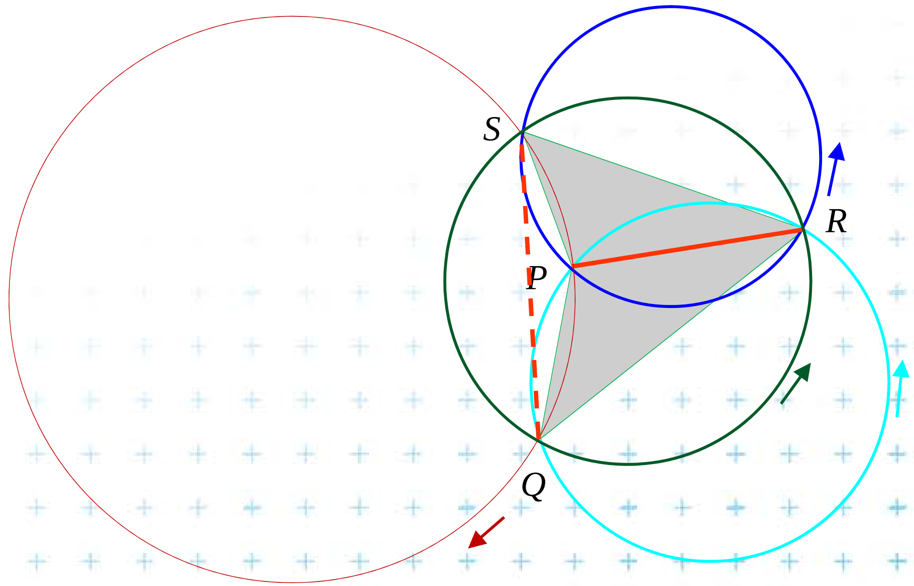
inCircle(…) > 0

Invalid diagonal

inCircle(…) < 0

Valid diagonal

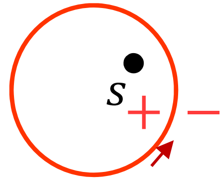


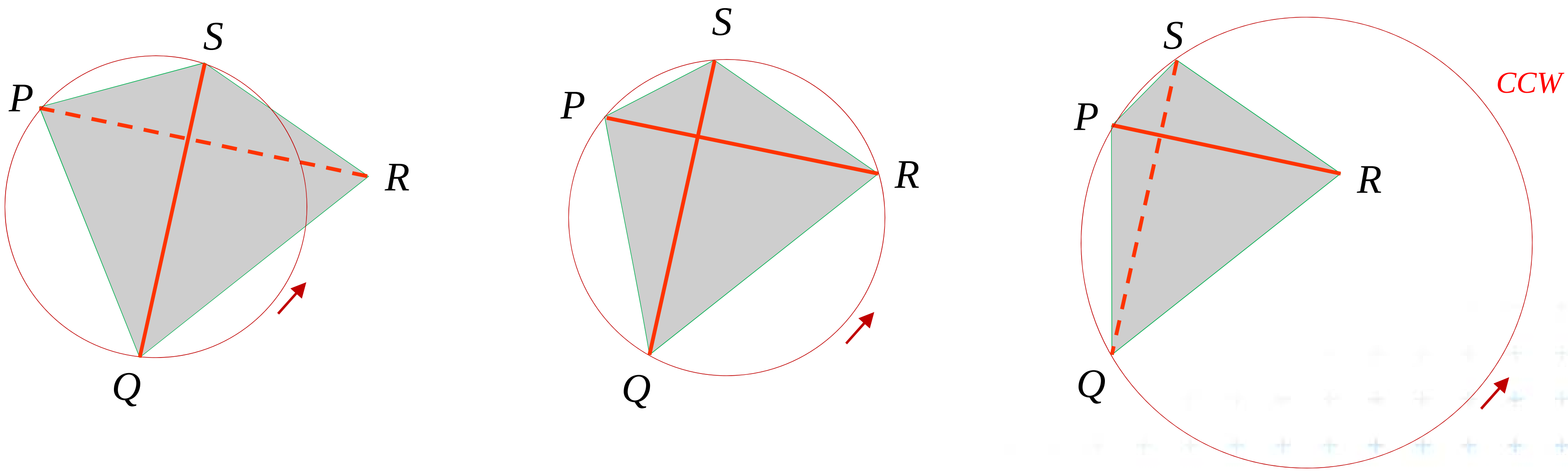


inCircle test detail

Point *P* moves right toward point *R*

We test position of *R* in relation to oriented circle (*P*,*Q*,*S*)





inCircle(P,Q,S,R) < 0
R is right (out)
diagonal QS is valid

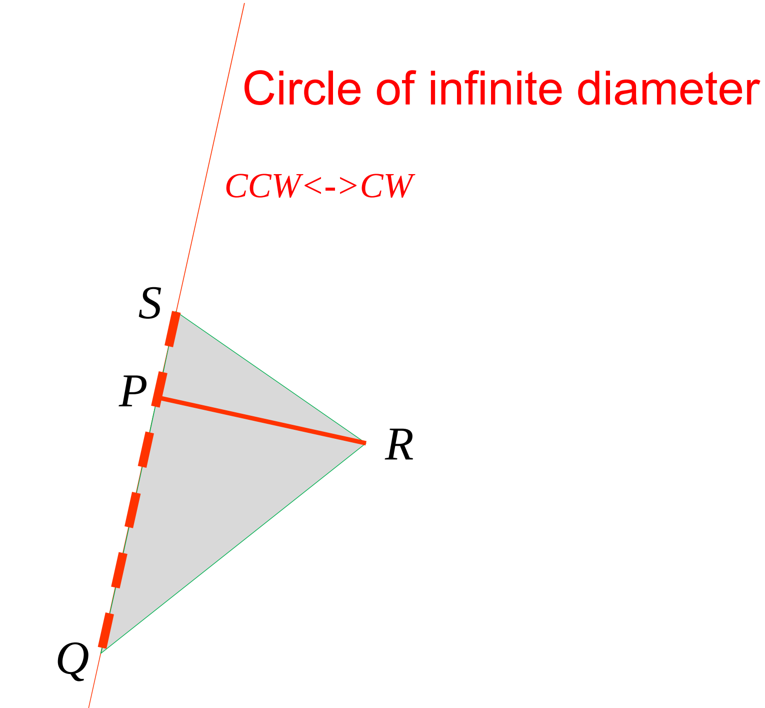
inCircle(P,Q,S,R) = 0
R is on the circle
both QS and PR are valid

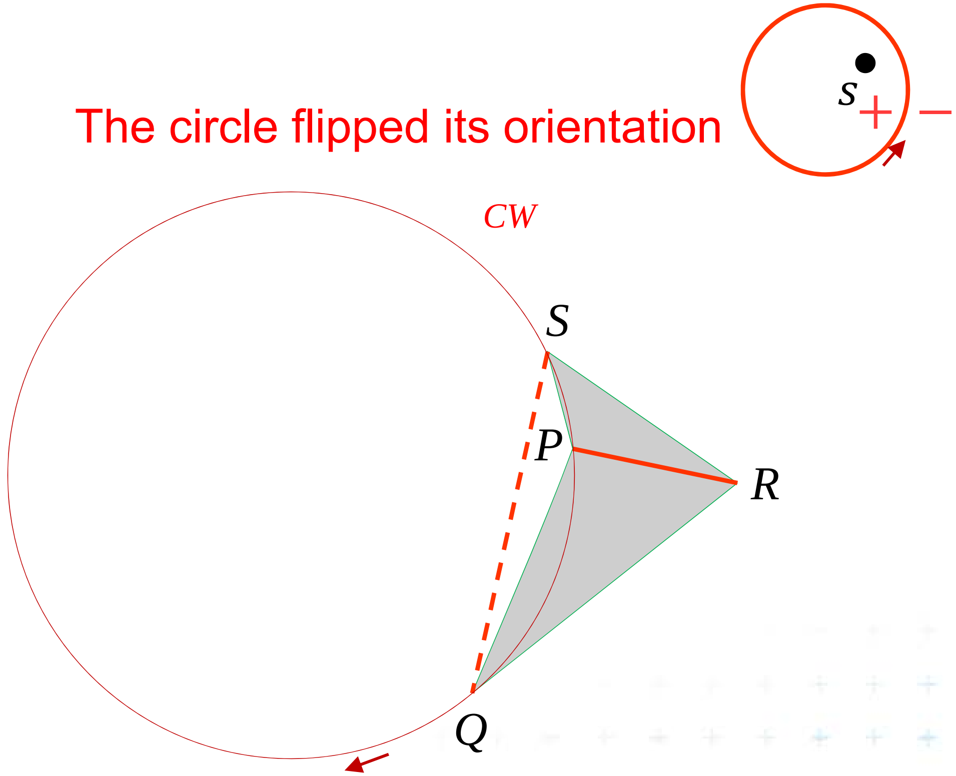
inCircle(P,Q,S,R) > 0
R is left (in)
QS is invalid

Invalid diagonal

Valid diagonal

inCircle test detail





inCircle(P,Q,S,R) > 0

R is left

QS is invalid

inCircle(P,Q,S,R) > 0

R is left

QS is invalid

Invalid diagonal

Valid diagonal

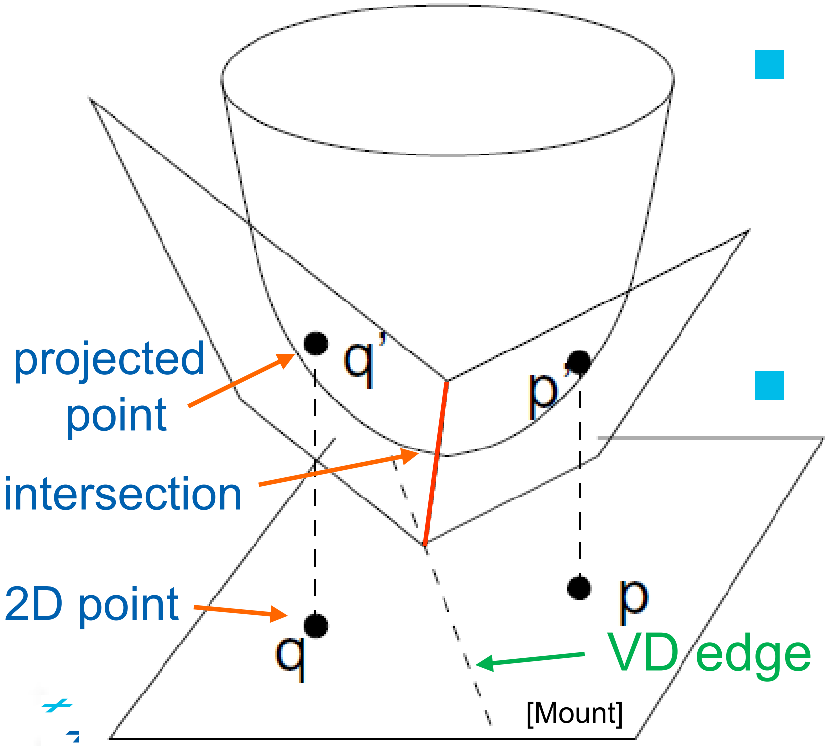
An the Voronoi diagram?

- VD and DT are dual structures
- Points and lines in the plane
  are dual to
  points and planes in 3D space
- VD of points in the plane
  can be transformed to
  intersection of halfspaces in 3D space

Voronoi diagram as upper envelope in R^d+1

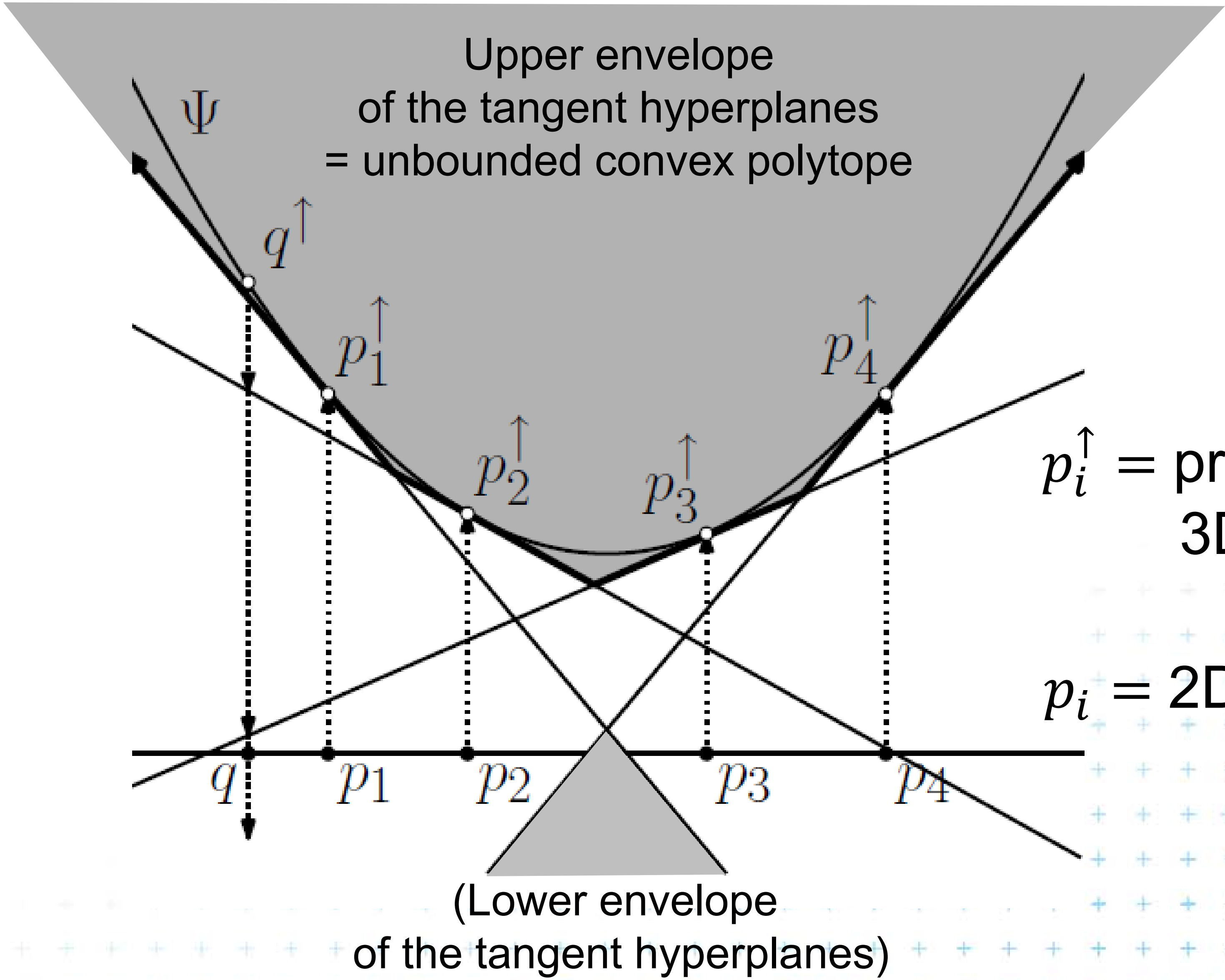
- For each point $p = (a, b)$ a tangent plane $H(p)$ to the paraboloid is $z = 2ax + 2by - (a^2 + b^2)$
- $H^+(p)$ is the set of points above this tangent plane

$$H^+(p) = \{(x, y, z) \mid z \geq 2ax + 2by - (a^2 + b^2)\}$$



- VD of points in the plane can be computed as intersection of halfspaces $H^+(p_i)$ in 3D
- This intersection of halfspaces = unbounded convex polyhedron = upper envelope of halfspaces $H^+(p_i)$

Upper envelope of planes (a 2D cross section)



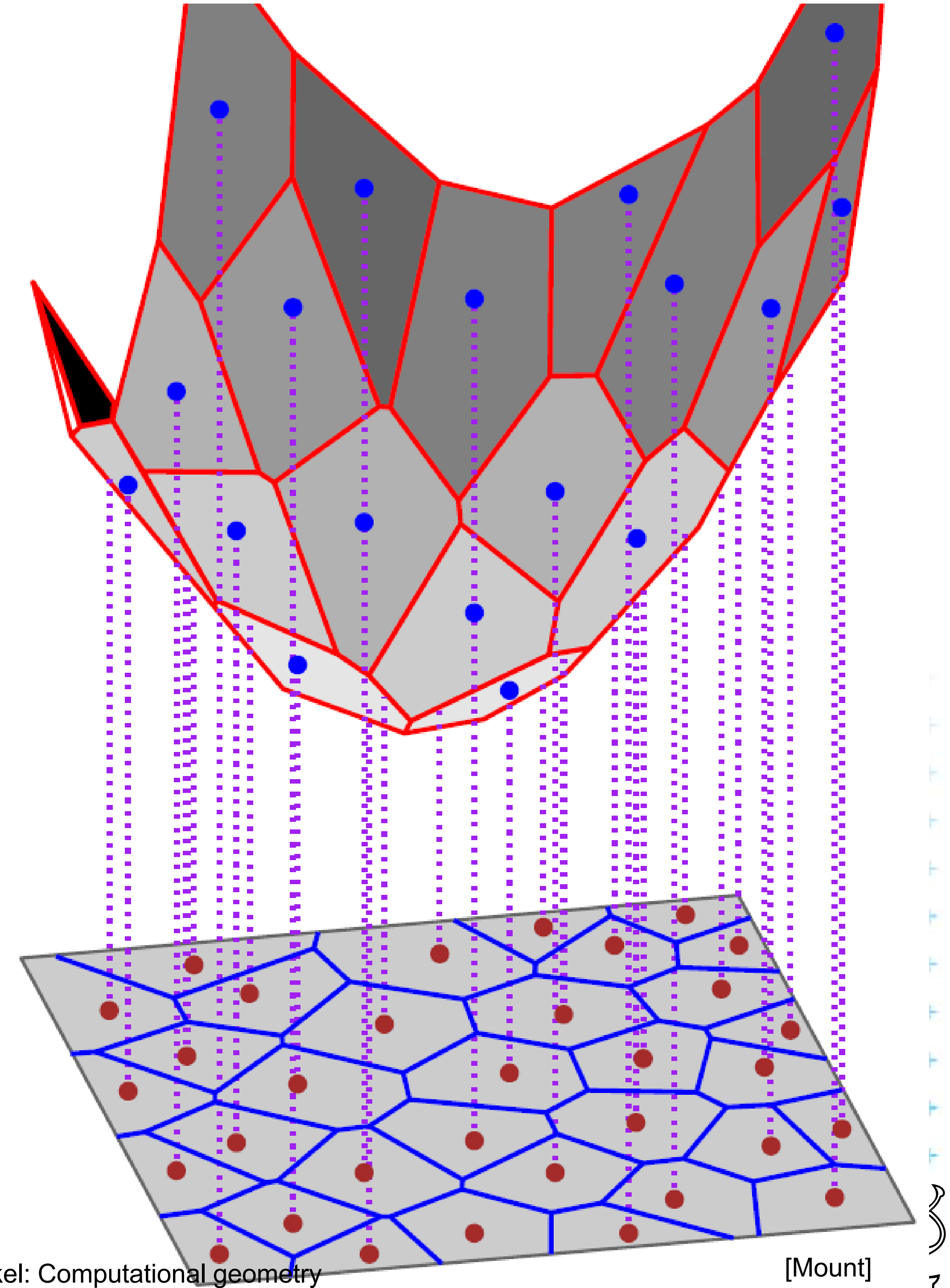
$p_i^{\uparrow}$ = projected 3D points

p_i = 2D points

[Mount]

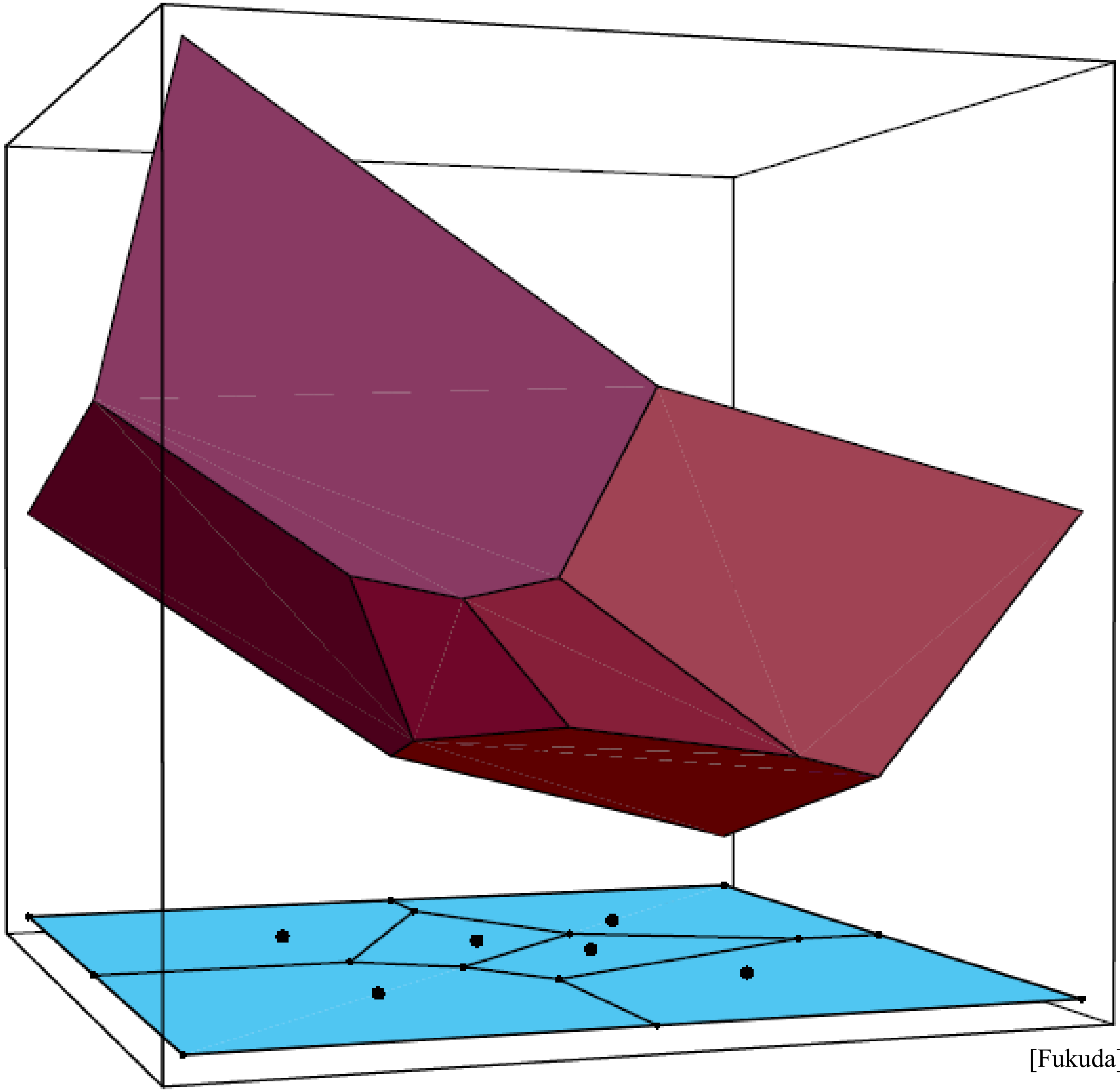
Projection to 2D

- Upper envelope of tangent hyperplanes (through sites projected upwards to the cone)
- Projected to 2D gives Voronoi diagram



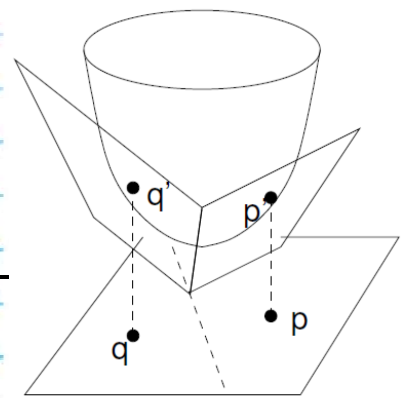
[Mount]

Voronoi diagram as upper envelope in 3D



[Fukuda]

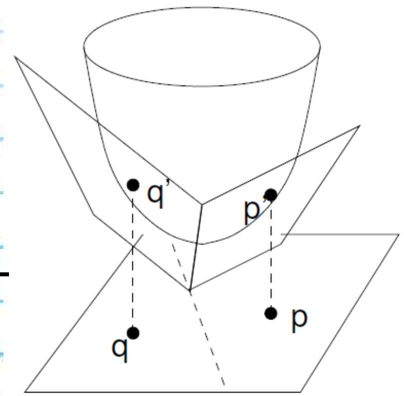
Derivation of projected Voronoi edge



[Mount]

Derivation of projected Voronoi edge

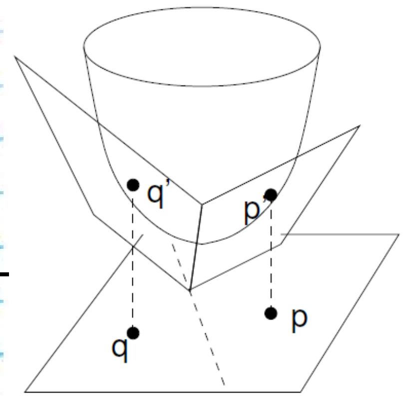
- 2 points: $p = (a, b)$ and in the plane



[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

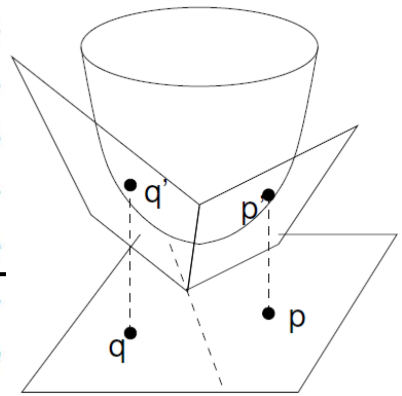


[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes
  to paraboloid

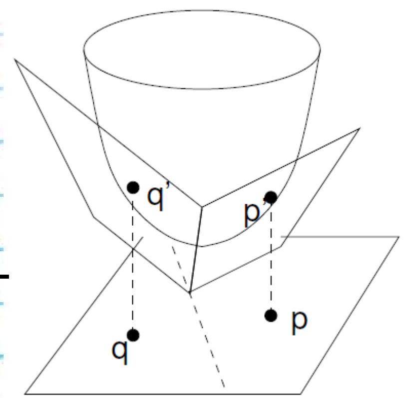


[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes $z = 2ax + 2by - (a^2 + b^2)$
  to paraboloid



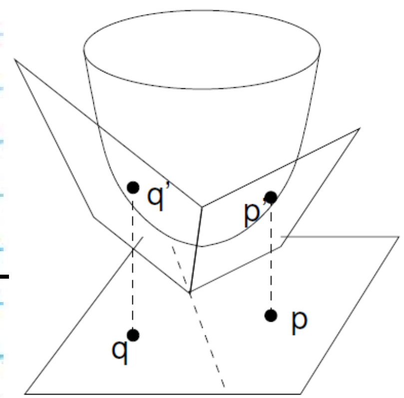
[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes to paraboloid
  $$z = 2ax + 2by - (a^2 + b^2)$$
  $$z = 2cx + 2dy - (c^2 + d^2)$$

[Mount]

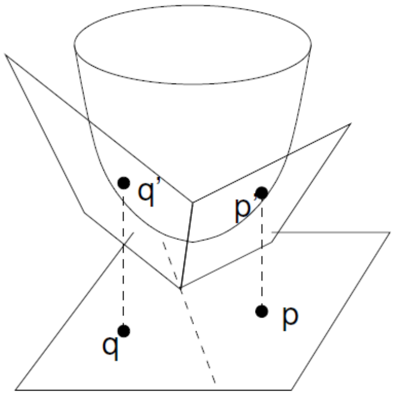


Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes to paraboloid
  $$z = 2ax + 2by - (a^2 + b^2)$$
  $$z = 2cx + 2dy - (c^2 + d^2)$$

- Intersect the planes, project onto xy (eliminate *z*)



[Mount]

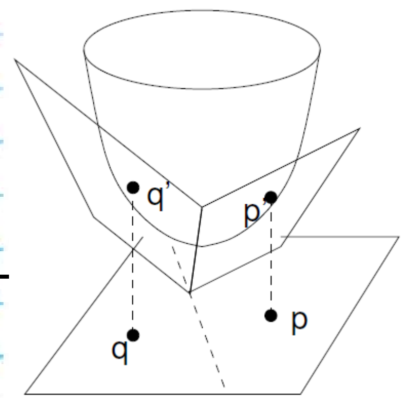
Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes to paraboloid

$$z = 2ax + 2by - (a^2 + b^2)$$
$$\underline{z = 2cx + 2dy - (c^2 + d^2) \qquad /(-)}$$

- Intersect the planes, project onto xy (eliminate *z*)



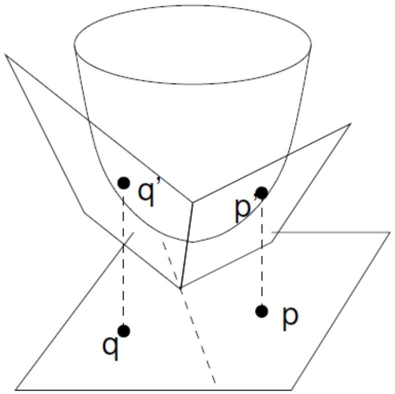
[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes to paraboloid
  $$z = 2ax + 2by - (a^2 + b^2)$$
  $$z = 2cx + 2dy - (c^2 + d^2) \qquad /(-)$$

- Intersect the planes, project onto xy (eliminate *z*)
  $$x(2a - 2c) + y(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$$



[Mount]

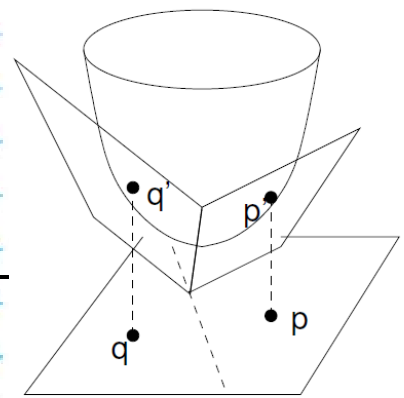
Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes to paraboloid
  $$z = 2ax + 2by - (a^2 + b^2)$$
  $$z = 2cx + 2dy - (c^2 + d^2) \qquad /(-)$$

- Intersect the planes, project onto xy (eliminate z)
  $$x(2a - 2c) + y(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$$

- This line passes through midpoint between p and q



[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes to paraboloid

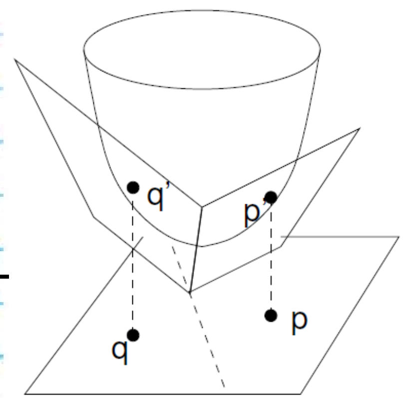
  $$z = 2ax + 2by - (a^2 + b^2)$$
  $$z = 2cx + 2dy - (c^2 + d^2) \quad /(-)$$

- Intersect the planes, project onto xy (eliminate z)

  $$x(2a - 2c) + y(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$$

- This line passes through midpoint between p and q

  $$\frac{a+c}{2}(2a - 2c) + \frac{b+d}{2}(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$$



[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes to paraboloid

$$z = 2ax + 2by - (a^2 + b^2)$$
$$z = 2cx + 2dy - (c^2 + d^2) \qquad /(-)$$

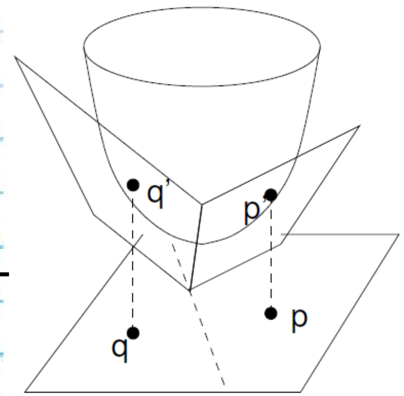
- Intersect the planes, project onto xy (eliminate *z*)

$$x(2a - 2c) + y(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$$

- This line passes through midpoint between p and q

$$\frac{a+c}{2}(2a - 2c) + \frac{b+d}{2}(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$$

- It is perpendicular bisector with slope



[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  2 tangent planes to paraboloid

$$z = 2ax + 2by - (a^2 + b^2)$$
$$z = 2cx + 2dy - (c^2 + d^2) \quad /(-)$$

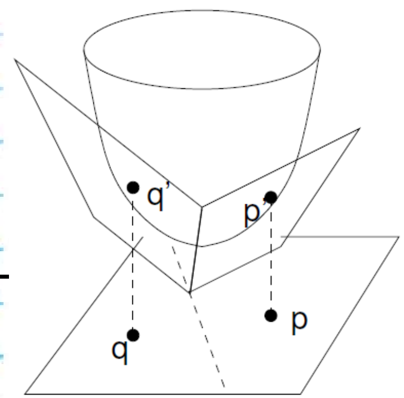
- Intersect the planes, project onto xy (eliminate z)

$$x(\underline{2a - 2c}) + y(\underline{2b - 2d}) = (a^2 - c^2) + (b^2 - d^2)$$

- This line passes through midpoint between p and q

$$\frac{a+c}{2}(2a - 2c) + \frac{b+d}{2}(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$$

- It is perpendicular bisector with slope



[Mount]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

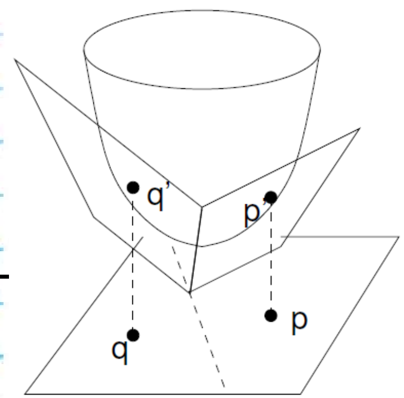
  2 tangent planes to paraboloid
  $$z = 2ax + 2by - (a^2 + b^2)$$
  $$z = 2cx + 2dy - (c^2 + d^2) \quad /(-)$$

- Intersect the planes, project onto xy (eliminate z)
  $$x(\underline{2a - 2c}) + y(\underline{2b - 2d}) = (a^2 - c^2) + (b^2 - d^2)$$

- This line passes through midpoint between p and q
  $$\frac{a+c}{2}(2a - 2c) + \frac{b+d}{2}(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$$

- It is perpendicular bisector with slope
  $$-(a - c)/(b - d)$$

[Mount]



References

[Berg] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: Computational Geometry: Algorithms and Applications, Springer-Verlag, 3rd rev. ed. 2008. 386 pages, 370 fig. ISBN: 978-3-540-77973-5, Chapters 3 and 9, http://www.cs.uu.nl/geobook/

[Mount] Mount, D.: *Computational Geometry Lecture Notes for Fall 2016*, University of Maryland, Lectures 6, 12, 13, 16, and 22. http://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf

[Rourke] Joseph O´Rourke: .: Computational Geometry in C, Cambridge University Press, 1993, ISBN 0-521- 44592-2 http://maven.smith.edu/~orourke/books/compgeom.html

[Fukuda] Komei Fukuda: Frequently Asked Questions in Polyhedral Computation. Version June 18, 2004 http://www.ifor.math.ethz.ch/~fukuda/polyfaq/polyfaq.html