

Computational Geometry

[csci 3250]

Laura Toma

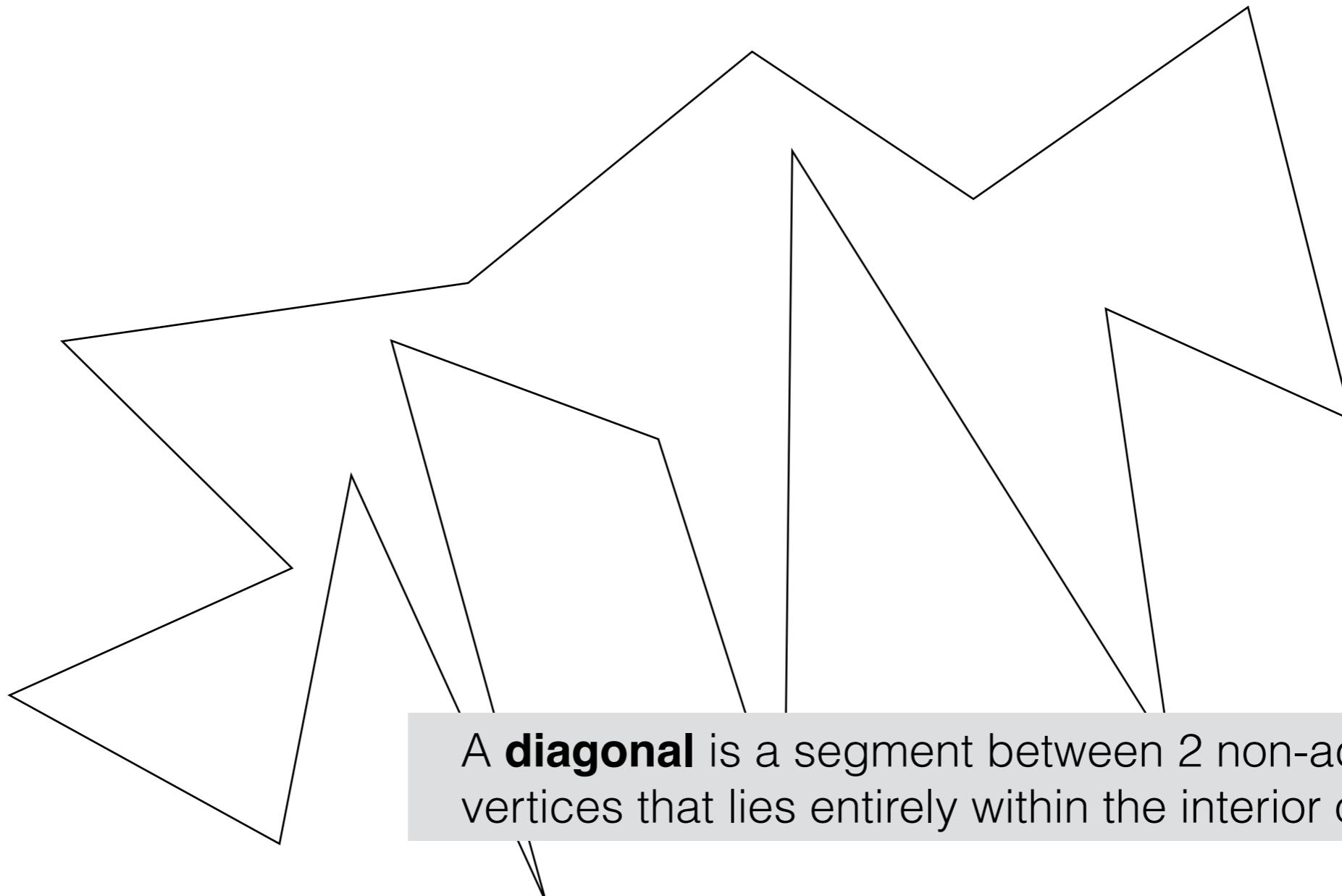
Bowdoin College

Polygon Triangulation

Polygon Triangulation: Definition

Polygon P

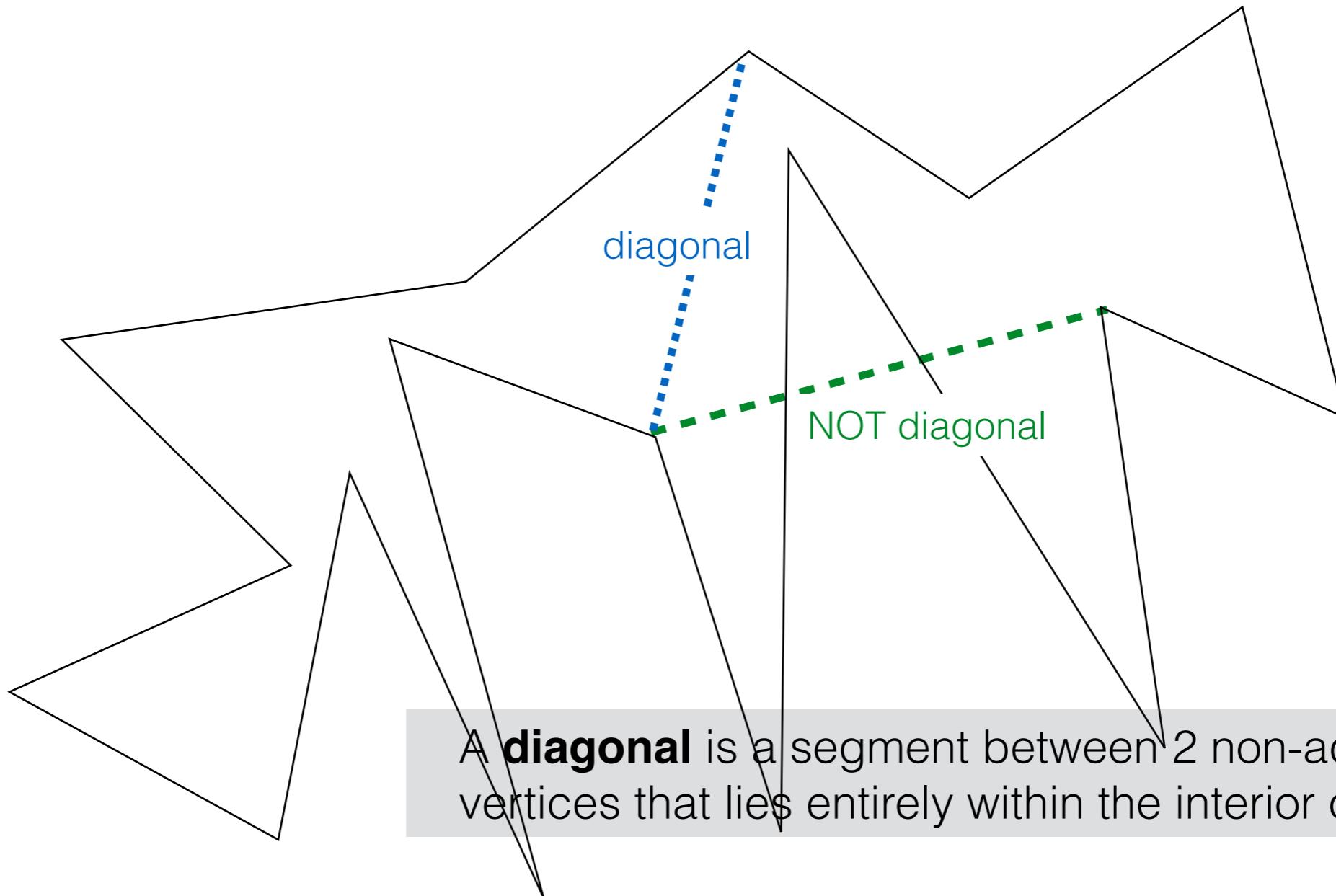
Triangulation of P: a partition of P into triangles using a set of diagonals.



Polygon Triangulation: Definition

Polygon P

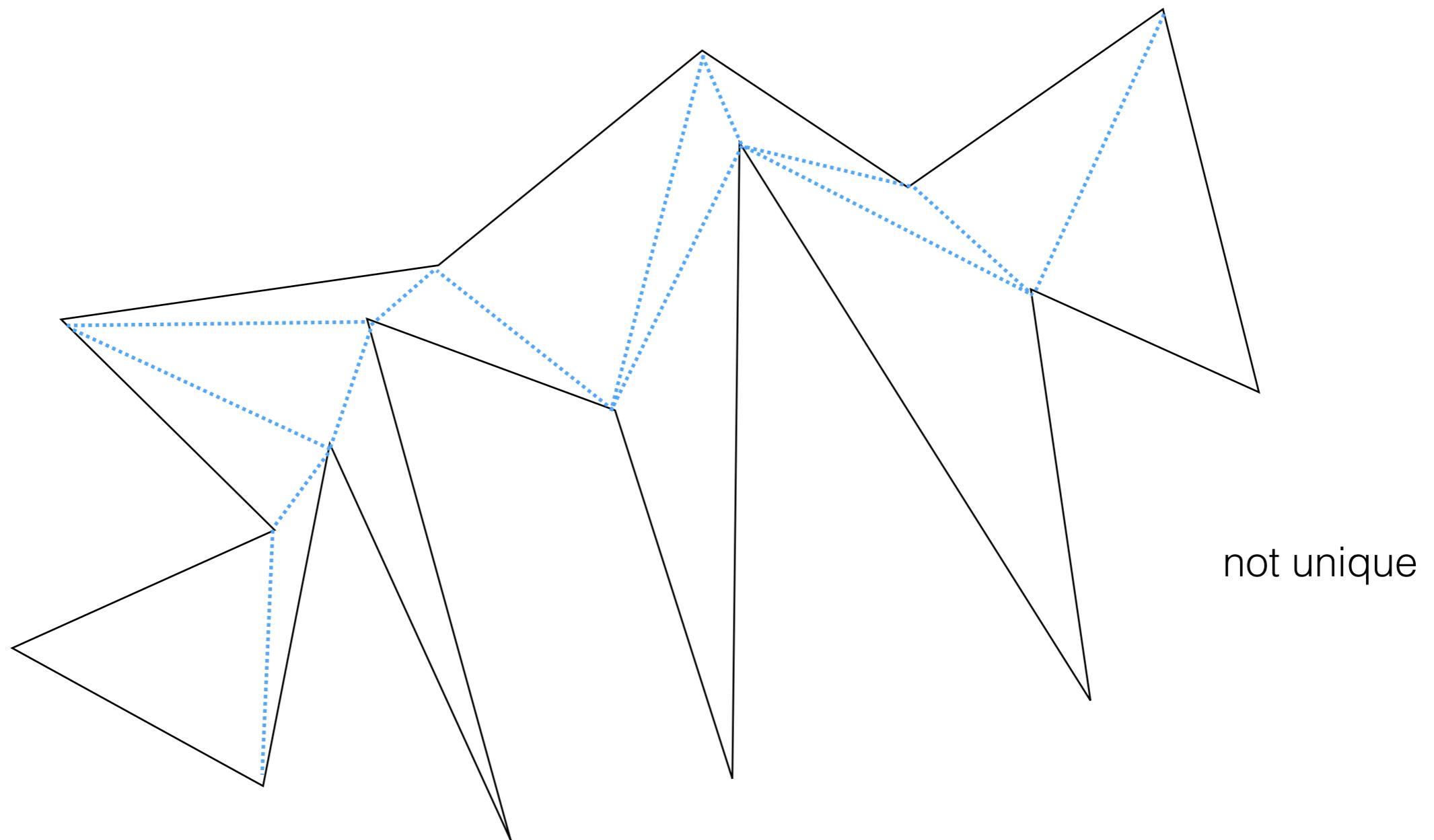
Triangulation of P: a partition of P into triangles using a set of diagonals.



Polygon Triangulation: Definition

Polygon P

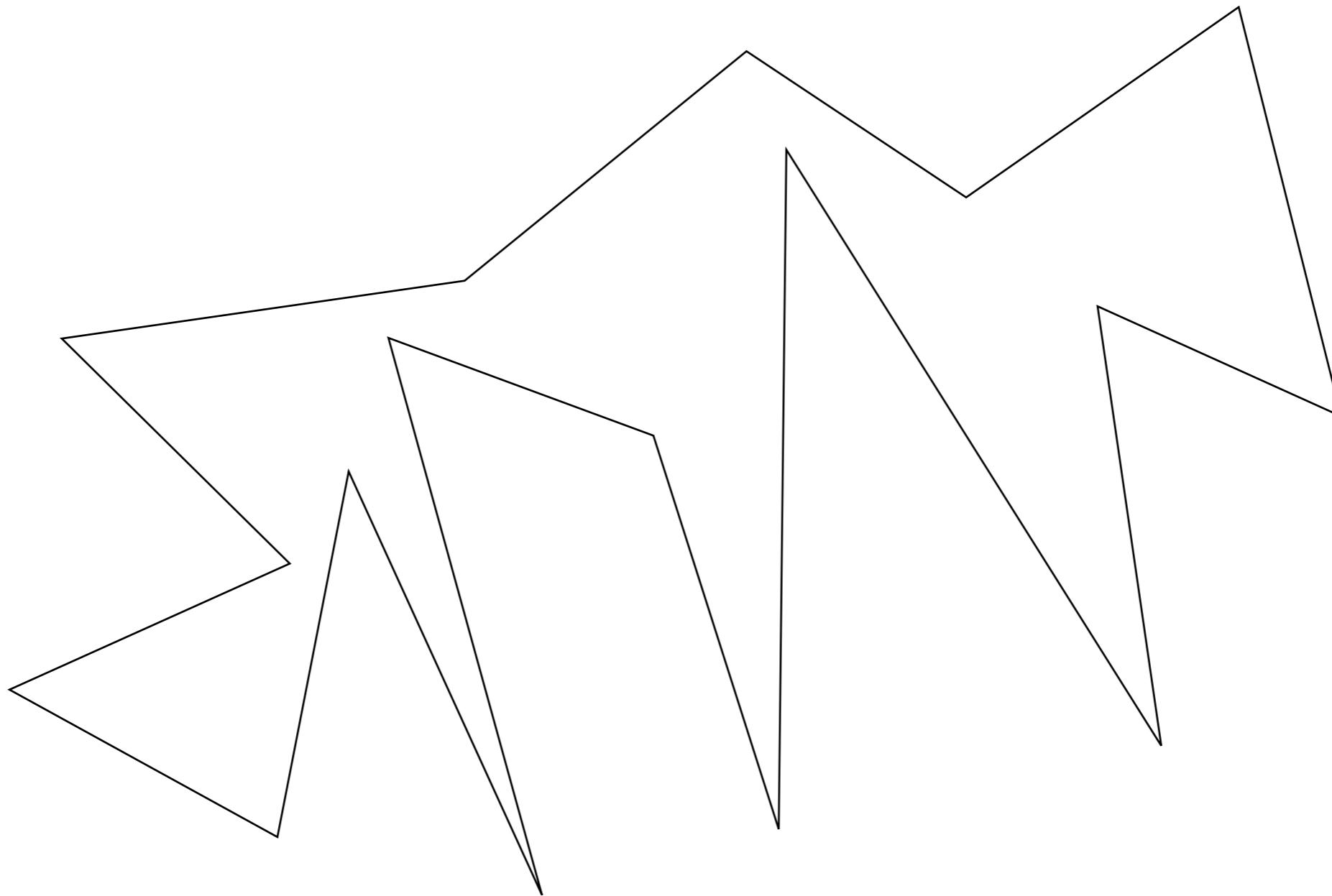
Triangulation of P: a partition of P into triangles using a set of diagonals.



Polygon Triangulation: The problem

Given a polygon P, triangulate it.

(output a set of diagonals that partition the polygon into triangles).

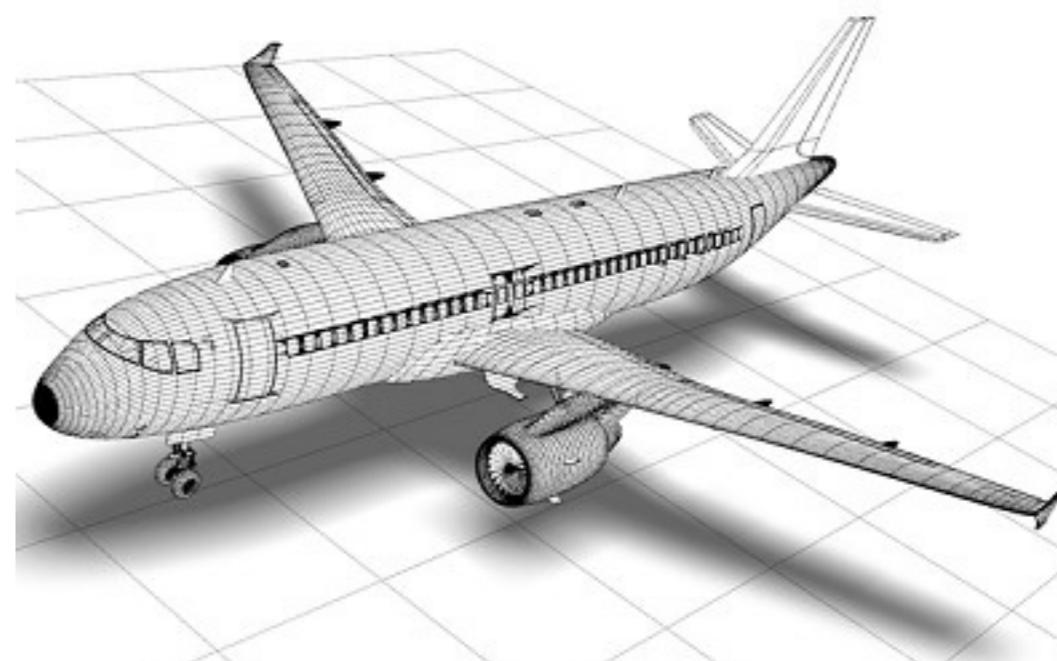
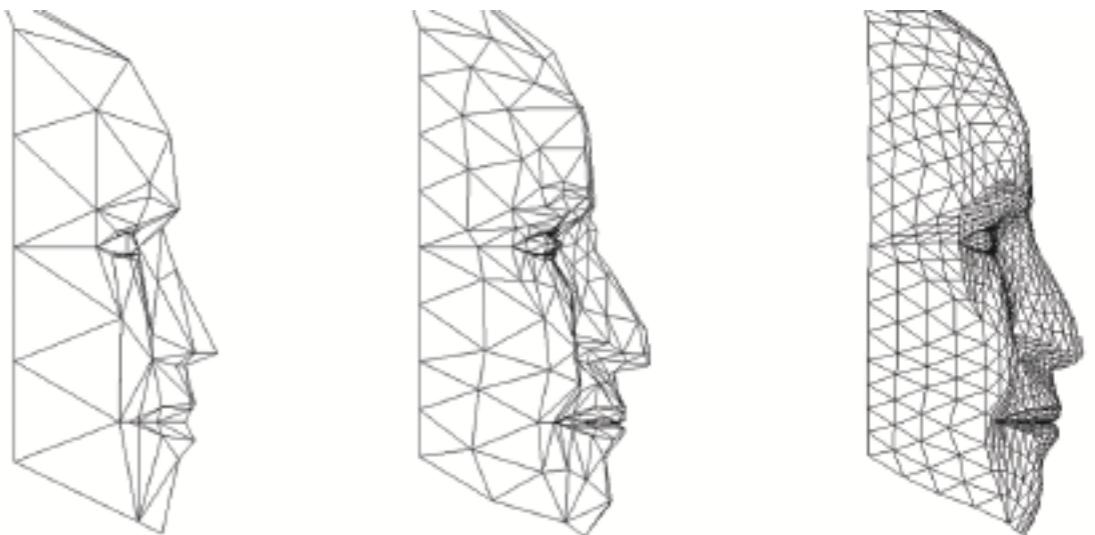
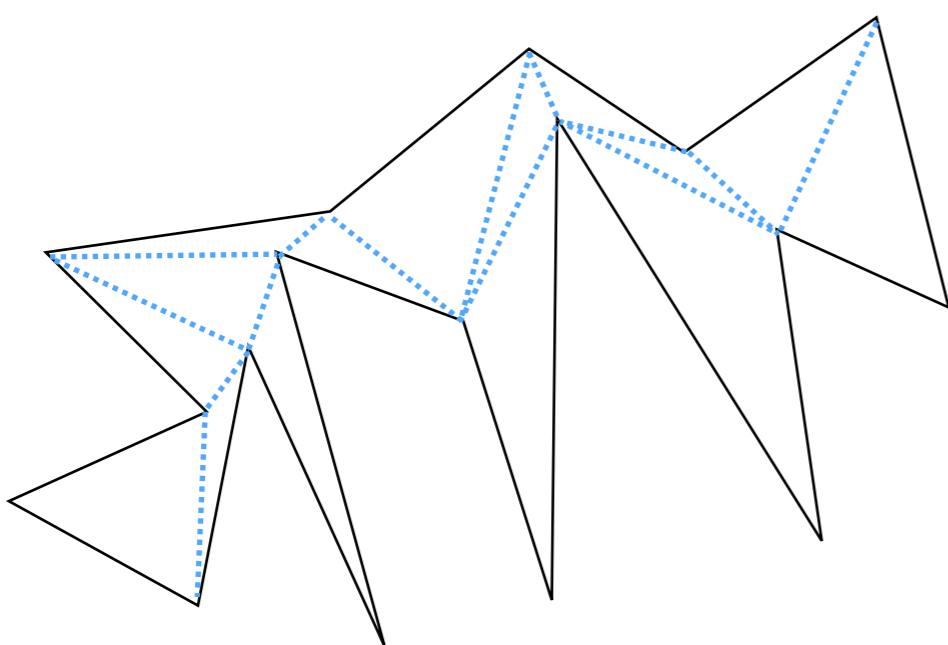


Motivation

Partitioning into simpler shapes: technique for dealing with complexity

2D: polygon triangulation

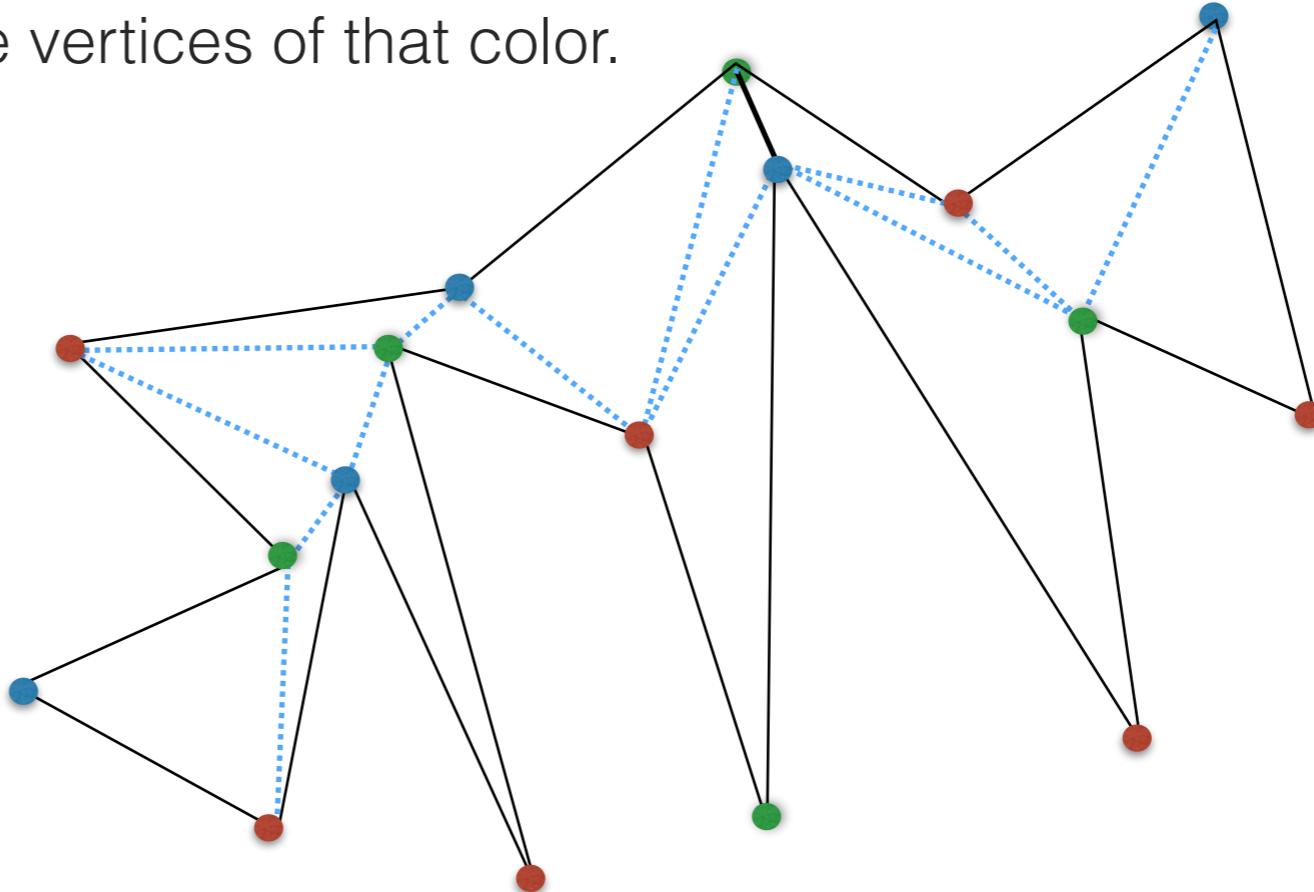
3D: meshing



Motivation

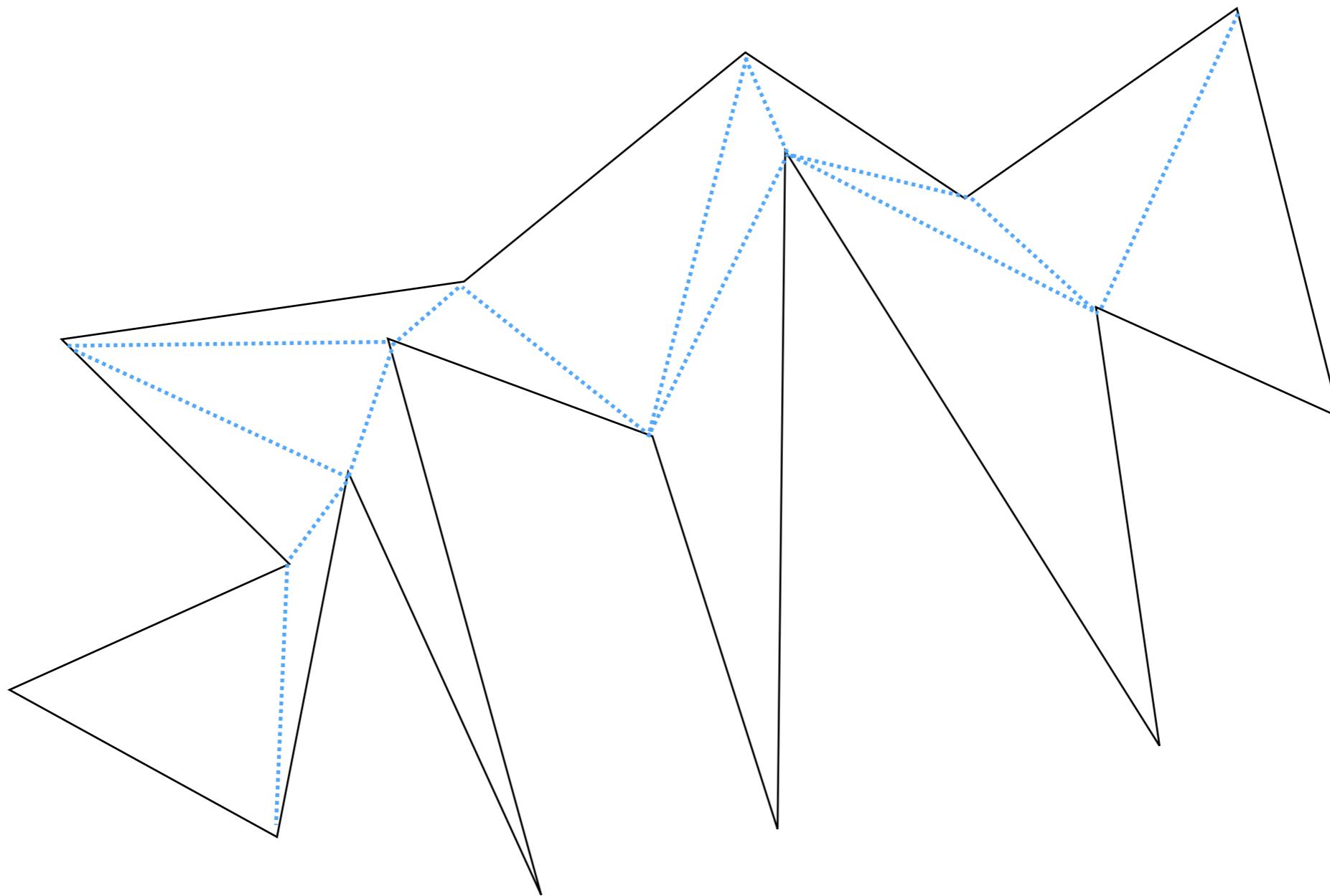
Art gallery , Fisk's proof

1. Any simple polygon can be triangulated.
2. Any triangulated simple polygon can be 3-colored.
3. Placing the guards at all the vertices assigned to one color guarantees the polygon is covered.
4. There must exist a color that's used at most $n/3$ times. Pick that color and place guards at the vertices of that color.



Does a triangulation always exist?

YES. The key to proving this is that any polygon ($n > 3$) has a diagonal.

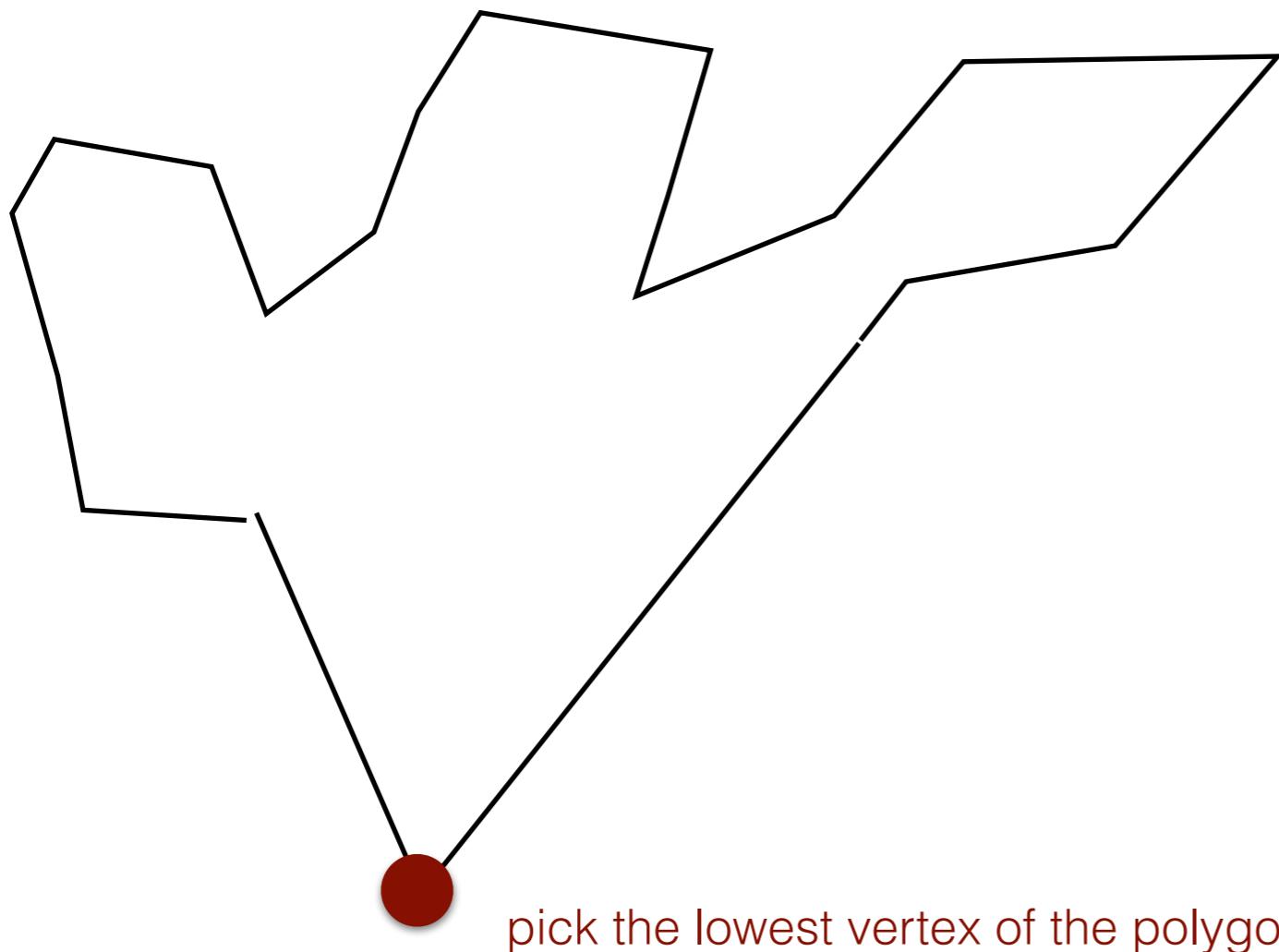


Known Results

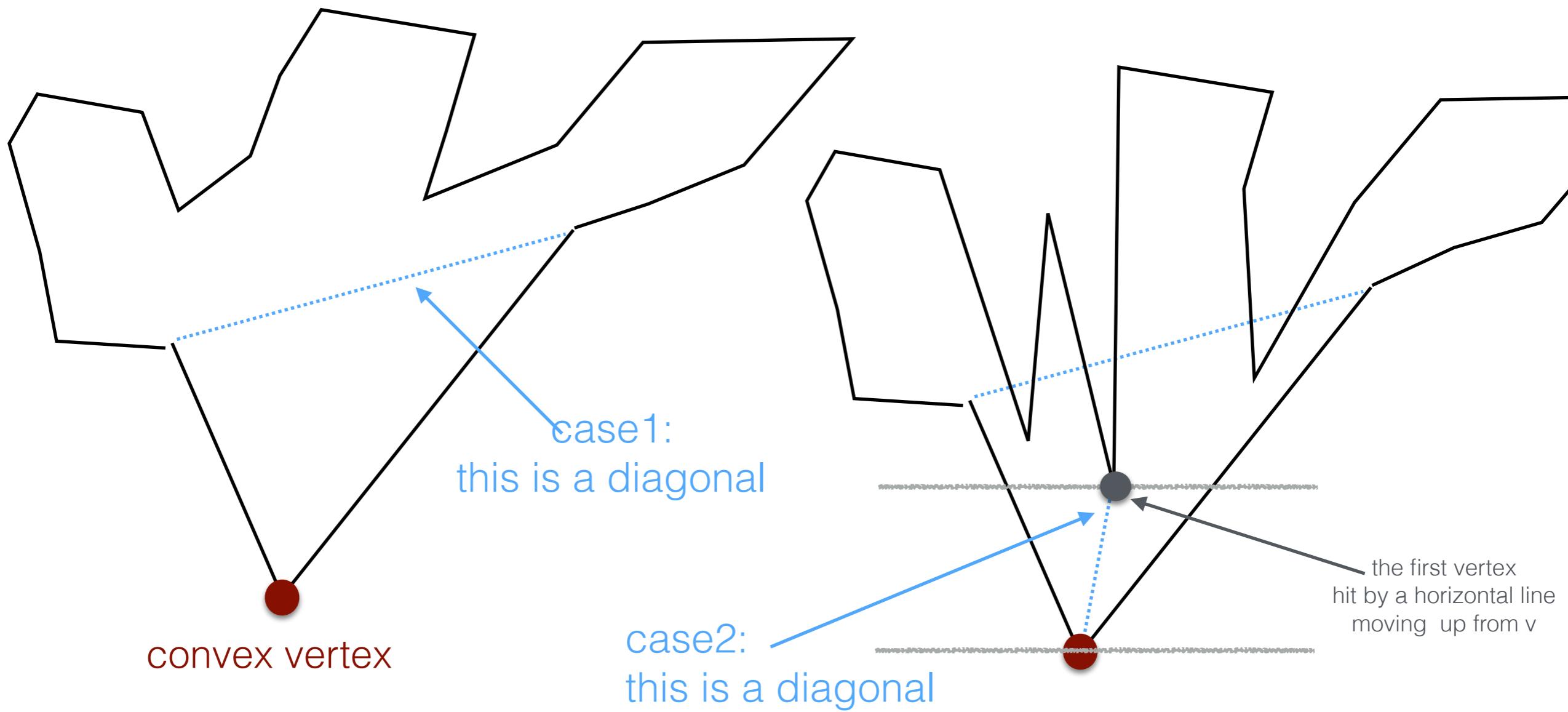
- Theorem 1: Any simple polygon must have a convex vertex (angle <180).
- Theorem 2: Any simple polygon with $n>3$ vertices contains (at least) a diagonal.
- Theorem 3: Any polygon can be triangulated by adding diagonals.
- Theorem 4: Any triangulation of a polygon of n vertices has $n-2$ triangles and $n-3$ diagonals.
- Theorem 5: Any simple polygon has at least two ears.

Theorem 1: Any simple polygon contains at least one **convex** vertex

the angle is <180



Theorem 2: Any simple polygon contains at least one diagonal.

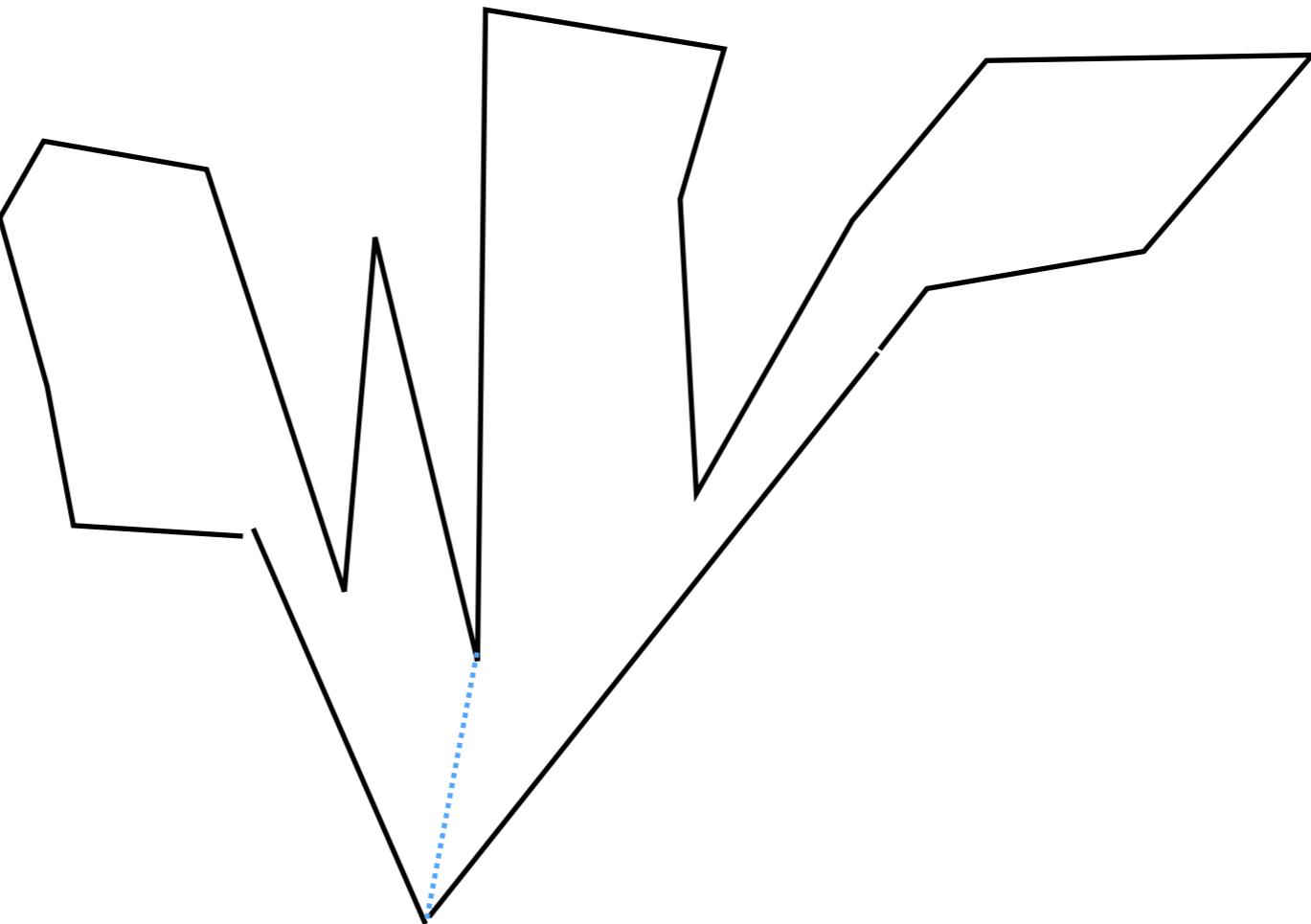


Theorem 3: Any polygon can be triangulated by adding diagonals

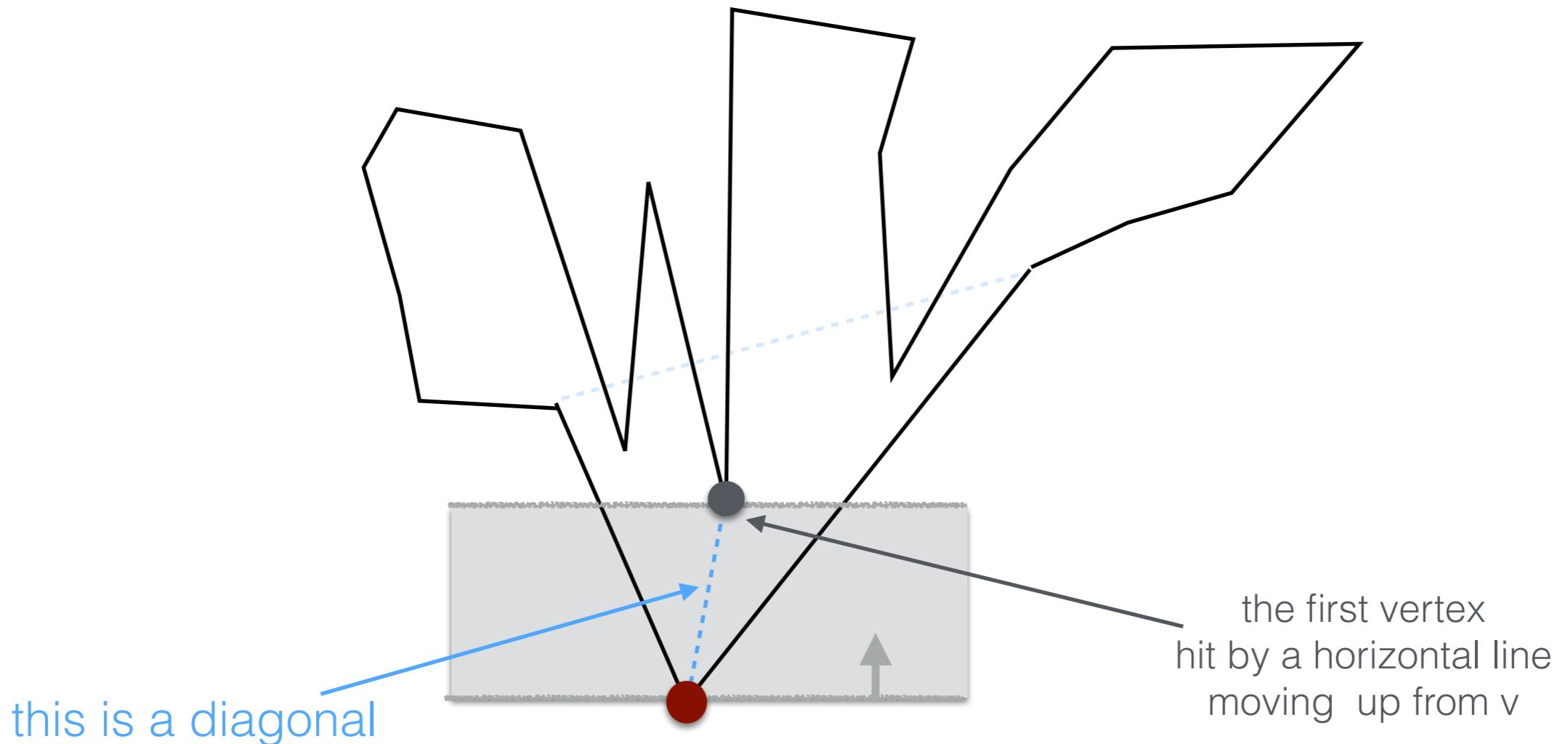
Proof: By induction on the size of the polygon

if $n=3$, holds trivially

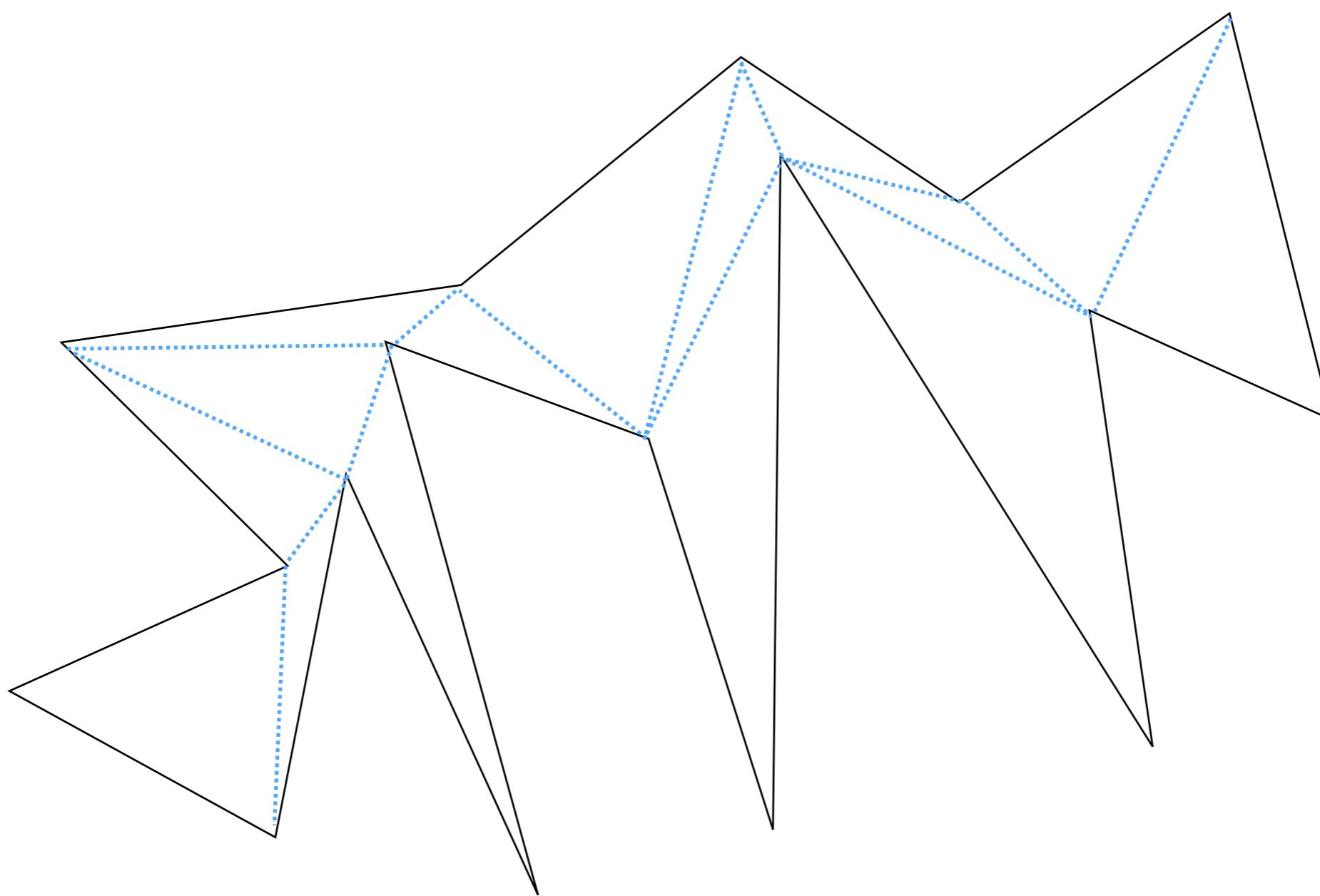
Assume it holds for any $k < n$. Partition P into two polygons with a diagonal. Each one has $< n$ vertices, and can be triangulated by ind. hyp.



Theorem 2: Any simple polygon contains at least one diagonal.



We want to triangulate a polygon



Naive triangulation by recursively finding diagonals

- Assume P is a polygon given as a vector of points (in ccw order along boundary)
- Idea:
 - Find a diagonal
 - Recurse
- Let's come up with an algorithm to determine if two vertices of P form a diagonal: $\text{isDiagonal}(p_i, p_j)$

- Let P be a polygon given as a vector of points
- ab is diagonal if it does not intersect the edges of P and is interior to P

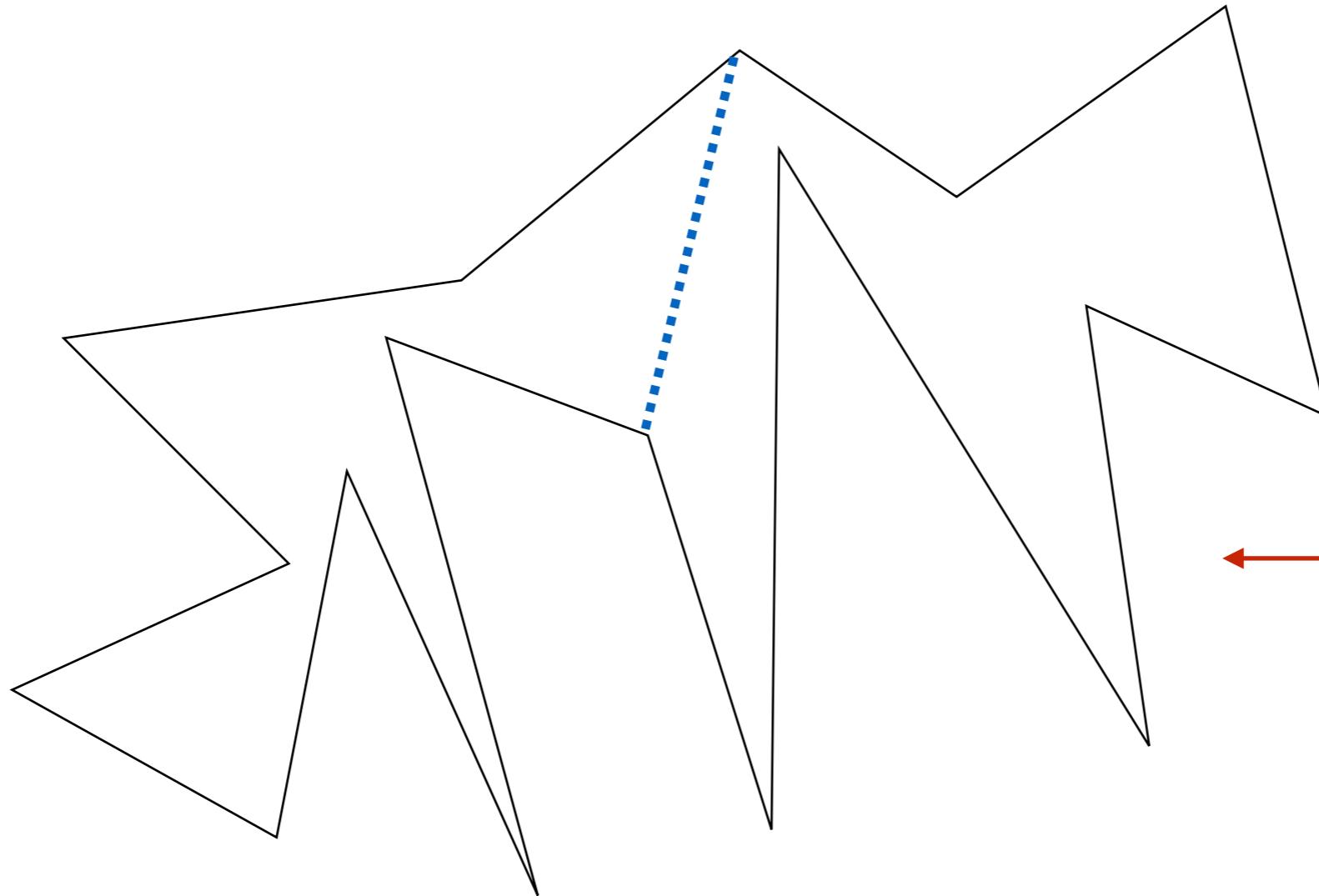
`isDiagonal(a,b):`

`//does any of the edges intersect ab?`

- for $i=0, i < n, i++:$
 - if $p_i p_{(i+1) \bmod n}$ intersects ab: return False
- ← actually: intersection at vertices is ok for the edges adjacent to a and b

`//if we got here, we know that ab intersects no edge. The only thing left to check is whether it's inside or outside P`

- return True if inside, False if outside



actually: intersection at
vertices is ok for the edges
adjacent to a and b

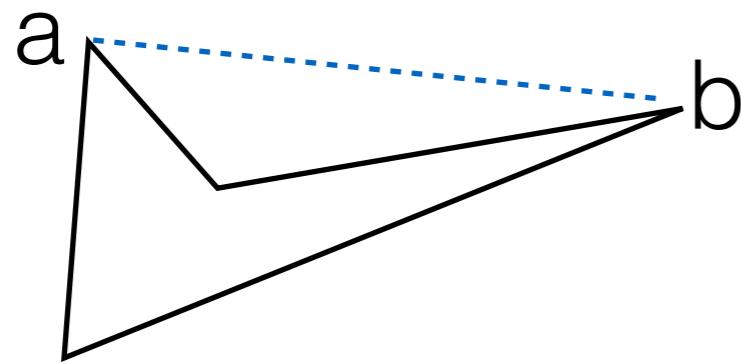
- Let P be a polygon given by a vector of vertices P

```
isDiagonal(a,b):
```

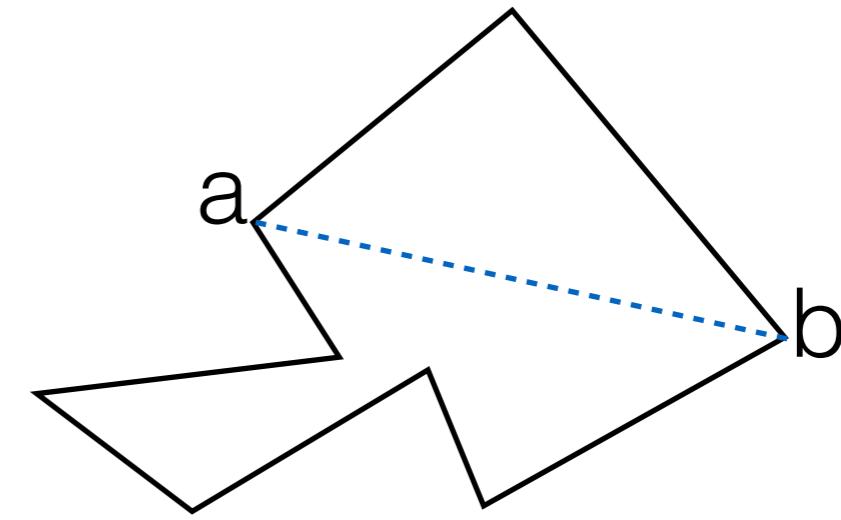
```
//does any of the edges intersect ab?
```

- for i=0; i< n; i++
 - if $a_i = p_i$ and $a_{(i+1)mod\ n} = p_{(i+1)mod\ n}$ and $b_i = p_i$ and $b_{(i+1)mod\ n} = p_{(i+1)mod\ n}$ and $\text{intersect}(a, b, p_i, p_{(i+1)mod\ n})$:
return False
- //if we got here, we know that ab intersects no edge. The only thing left to check is whether it's inside or outside P
- return True if inside, False if outside

- So ab does not intersect any edges. Is ab interior or exterior?

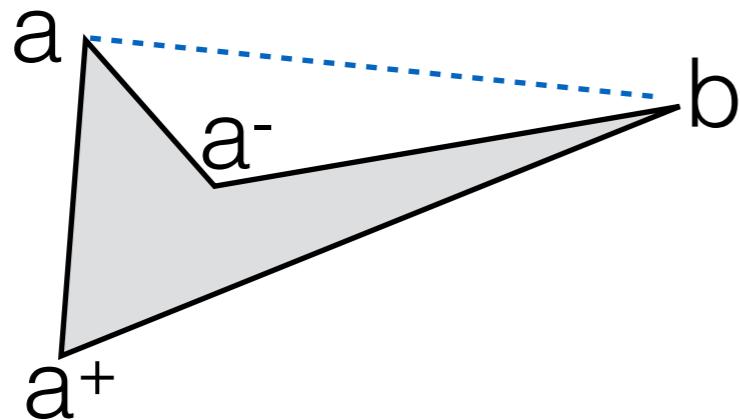


not a diagonal



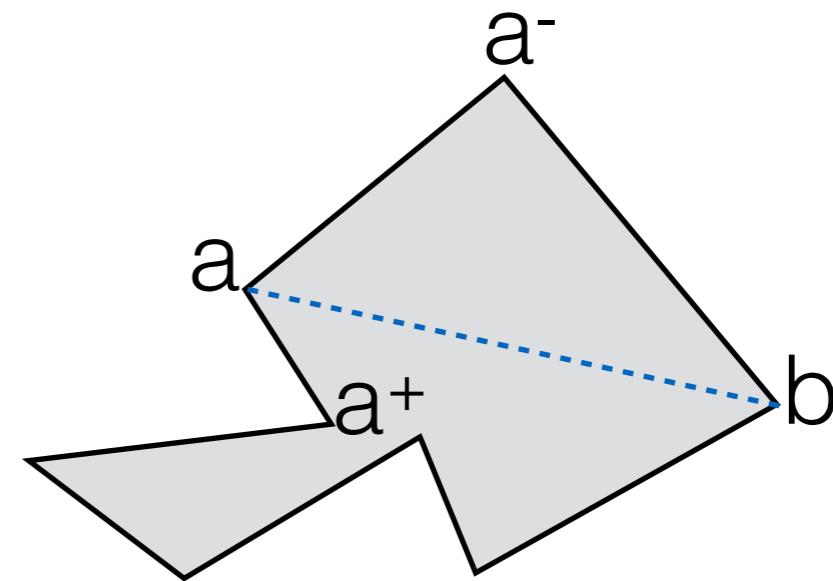
diagonal

- So ab does not intersect any edges. Is ab interior or exterior?



not a diagonal

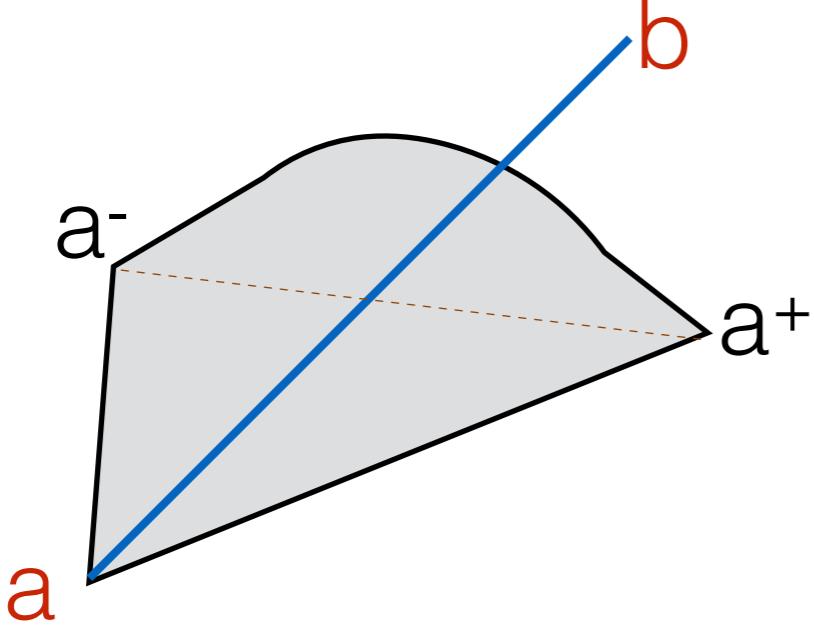
ab outside a^-, a, a^+



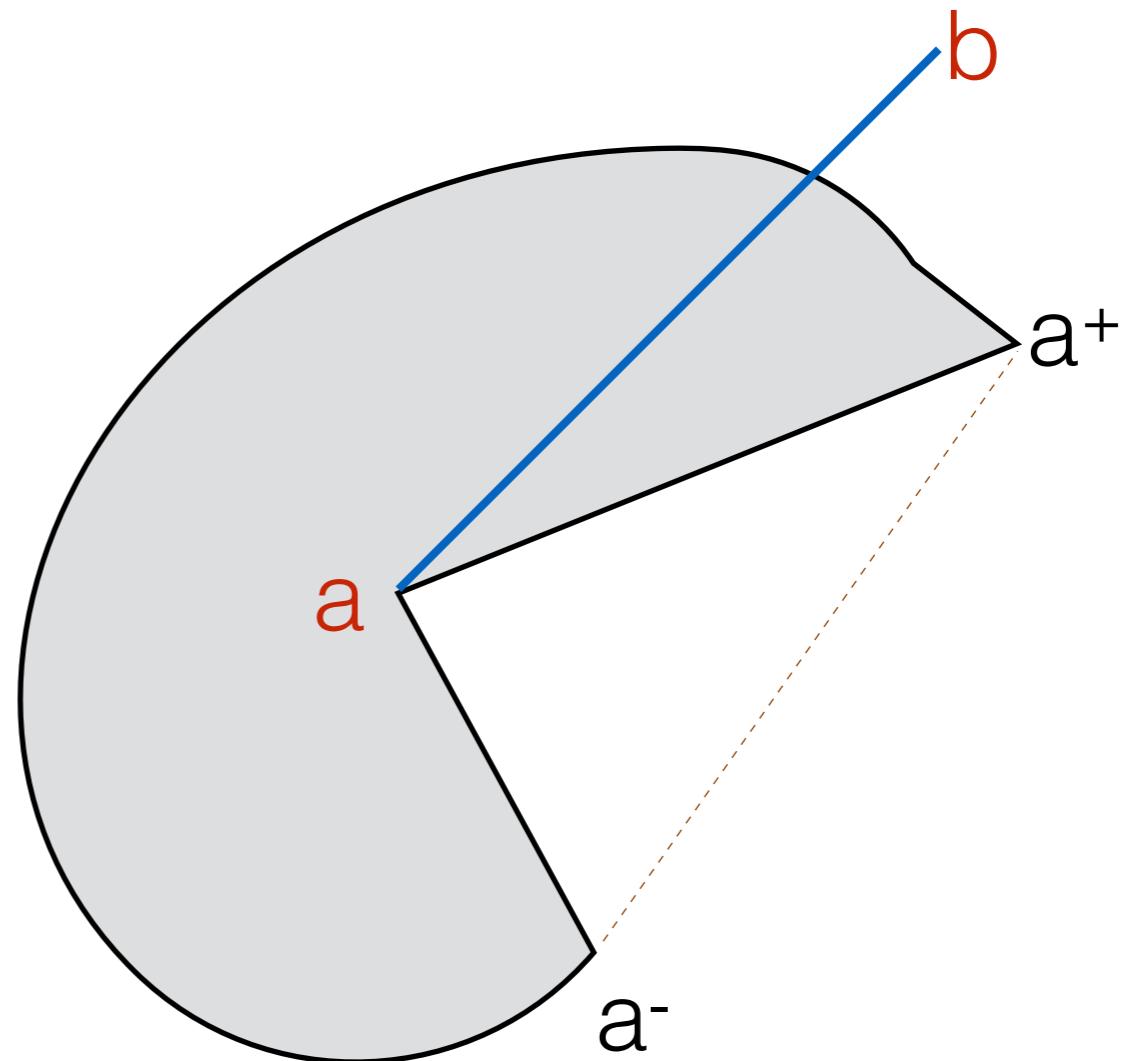
diagonal

ab is inside the cone formed by a^-, a, a^+

- $\text{InCone}(a, b)$: return True if ab is in the cone determined by a^- , a , a^+



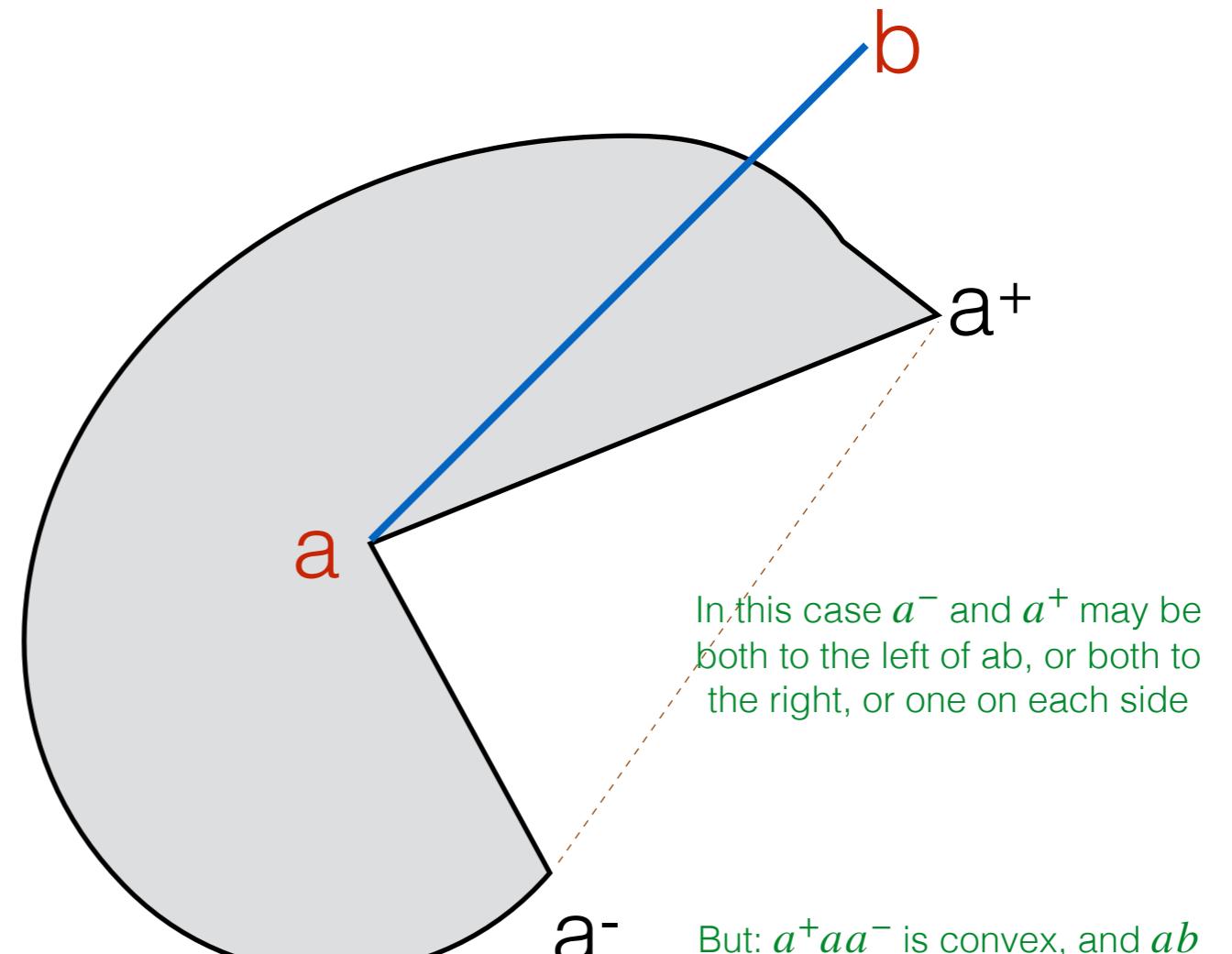
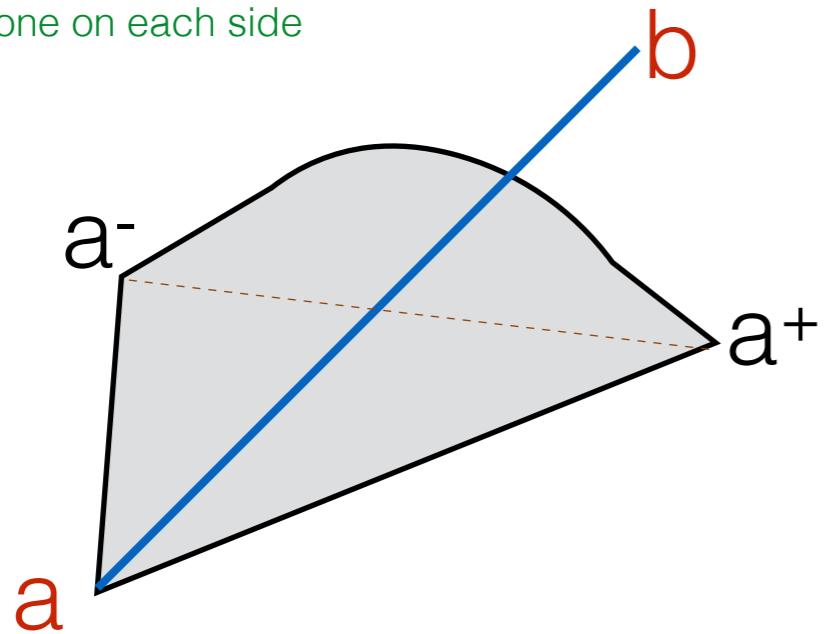
True



True

- $\text{InCone}(a, b)$: return True if ab is in the cone determined by a^-, a, a^+

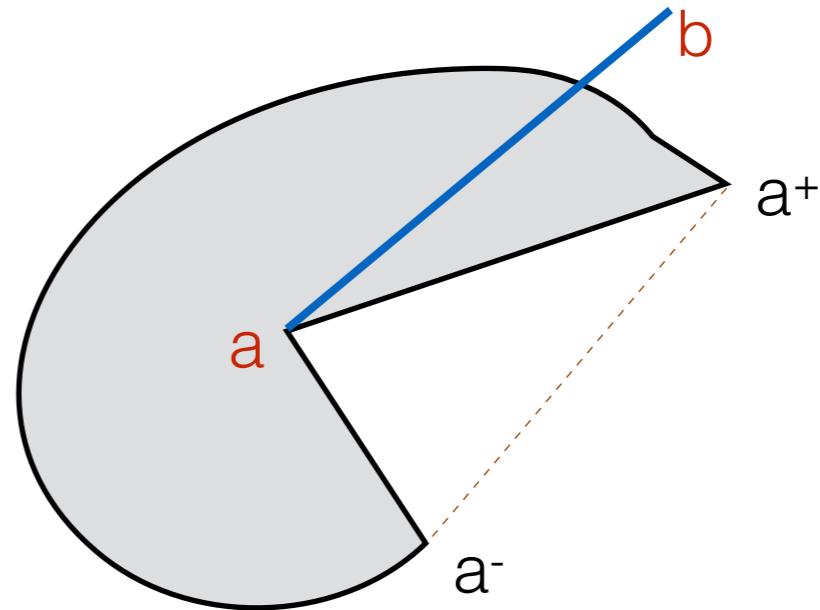
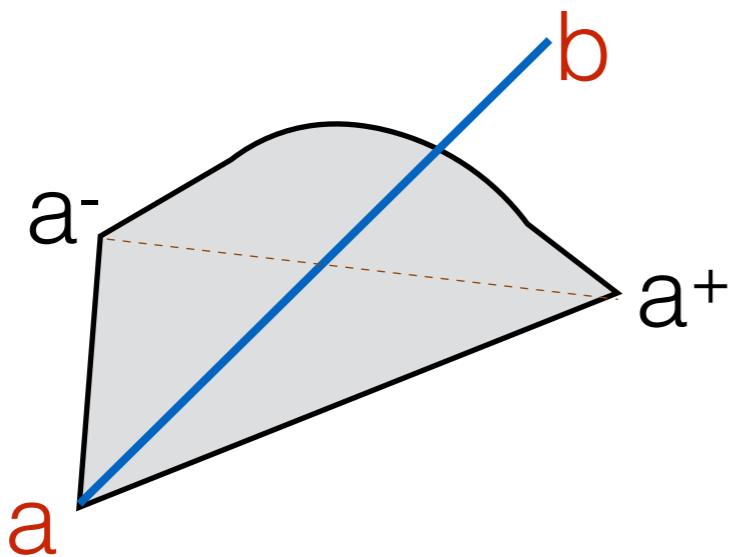
In this case a^- and a^+ must be one on each side



In this case a^- and a^+ may be both to the left of ab , or both to the right, or one on each side

But: a^+aa^- is convex, and ab is internal to a^-aa^+ if it is not internal to the convex a^+aa^-

- InCone(a , b): return True if ab is in the cone determined by a_-, a, a_+



// a, b must be points in P

- InCone(point2d a, b)
 - $a^- =$ point before a
 - $a^+ =$ point after a
- //if a is a convex vertex
- if $\text{LeftOn}(a^-, a, a^+)$: return $\text{Left}(a, b, a^-) \&\& \text{Left}(b, a, a^+)$
- Note: strict $\text{Left}()$ to exclude
ab collinear overlap with the cone
- //else its a reflex vertex
- return $!(\text{LeftOn}(a, b, a^+) \&\& \text{LeftOn}(b, a, a^-))$

Putting it all together: Is ab a diagonal?

//a, b must be points in P

```
isDiagonal(a,b):
```

//does any of the edges intersect ab?

- for i=0; i< n; i++

O(n) • if $a_1 = p_i$ and $a_2 = p_{(i+1)mod\ n}$ and $b_1 = p_i$ and $b_2 = p_{(i+1)mod\ n}$ and $\text{intersect}(a, b, p_i, p_{(i+1)mod\ n})$:
return False

//if we got here, we know that ab intersects no edge properly. The only thing left to check is whether it's inside or outside P

O(1) • return $\text{inCone}(a, b) \&\& \text{InCone}(b, a)$

Overall: O(n) time

In conclusion,
we can test if ab is a diagonal in $O(n)$ time

Straightforward way to find a diagonal:

- for $i=0, i < n, i++$
 - for $j=i+1, j < n, j++$
 - check if $p_i p_j$ is diagonal

Naive triangulation by recursively finding diagonals

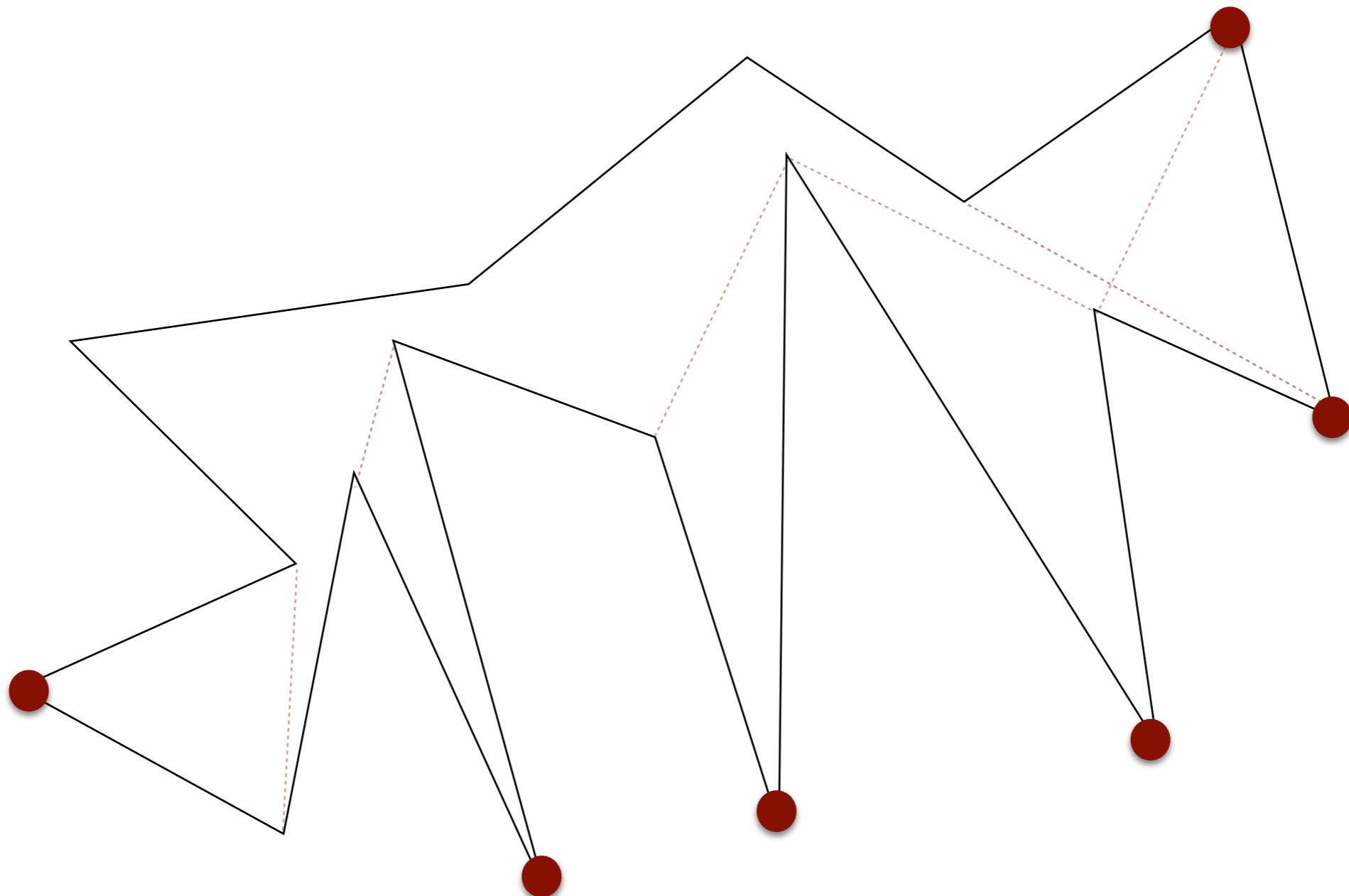
- **Algorithm 1:** Triangulation by finding diagonals
 - Idea: Check all pairs of vertices to find one which is a diagonal; recurse.
 - Analysis:
 - checking all vertices: $O(n^2)$ candidates for diagonals, checking each takes $O(n)$, overall $O(n^3)$
 - recurse, worst case on a problem of size $n-1$
 - overall $O(n^4)$
- **Algorithm 2:** Triangulation by *smartly* finding diagonals
 - A diagonal can be found in $O(n)$ time (using the proof that a diagonal exists)
 - Idea: Find a diagonal, output it, recurse.
 - $O(n^2)$

Algorithm 3: Triangulation by finding ears

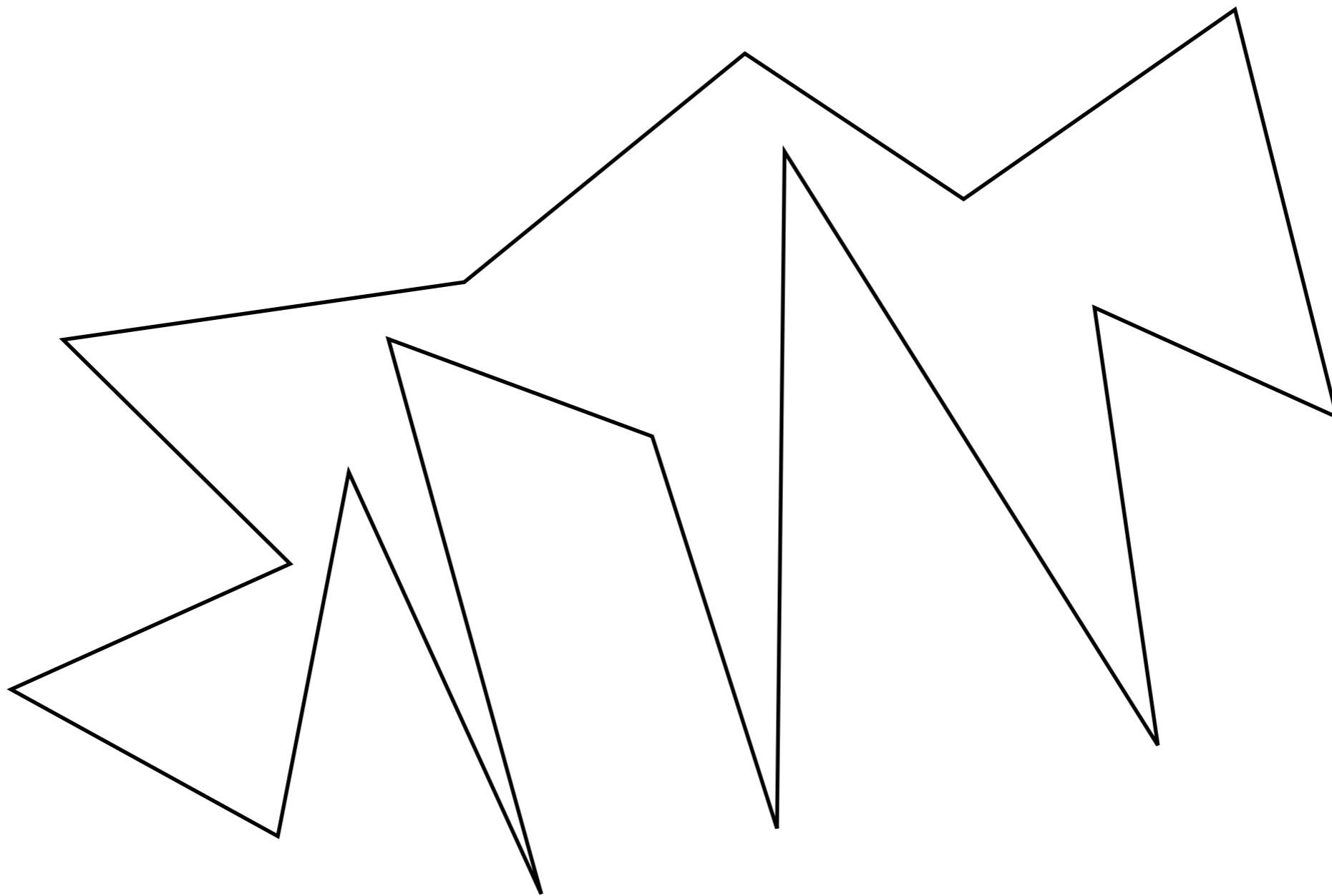


A bat-eared fox peeks from the grass in Hwange National Park, Zimbabwe.
PHOTOGRAPH BY ROY TOFT, NAT GEO IMAGE COLLECTION

A vertex p of P is called **ear** if p^-p^+ is a diagonal

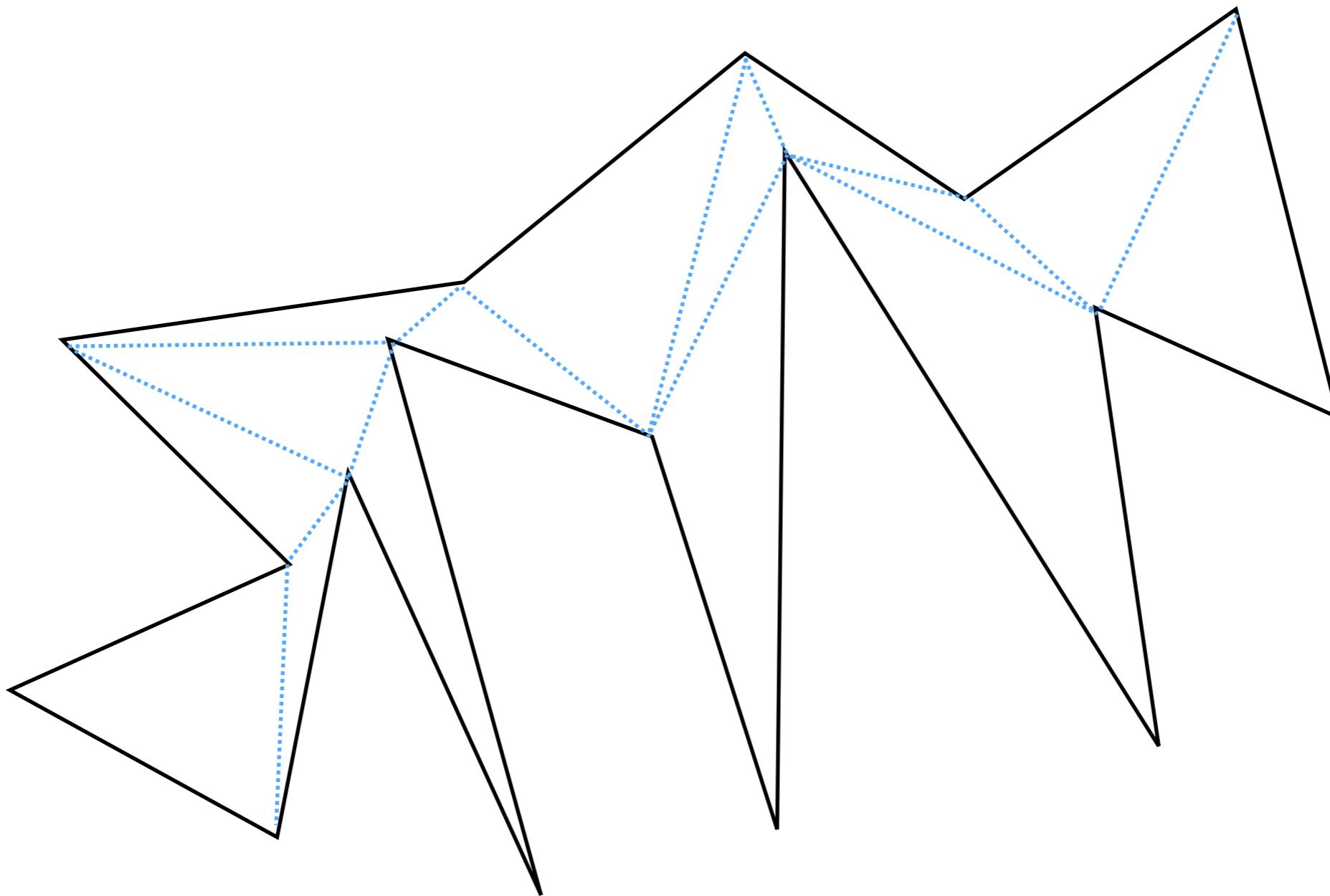


Theorem: Any simple polygon has at least two ears.



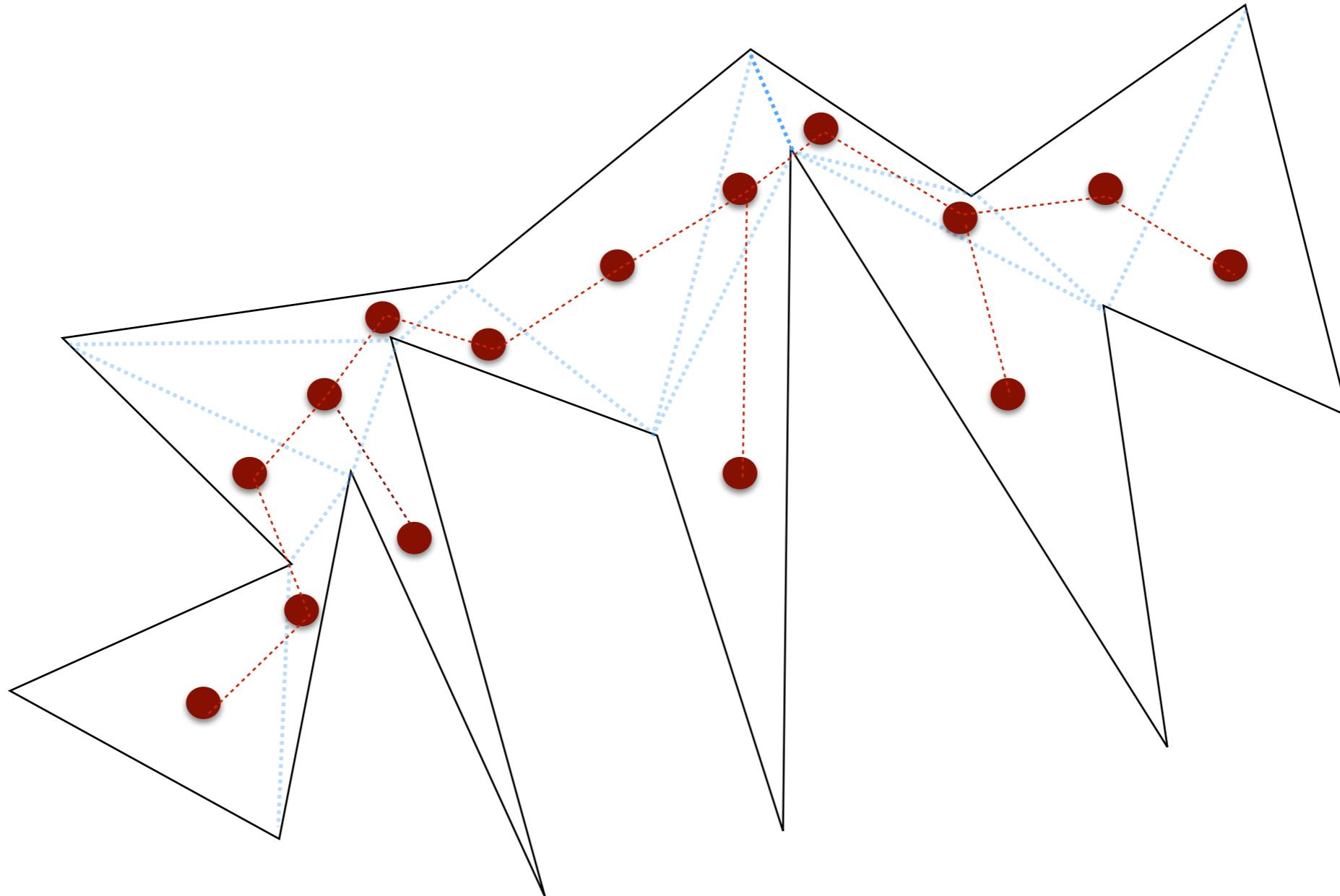
Theorem: Any simple polygon has at least two ears.

Proof: Triangulate P.



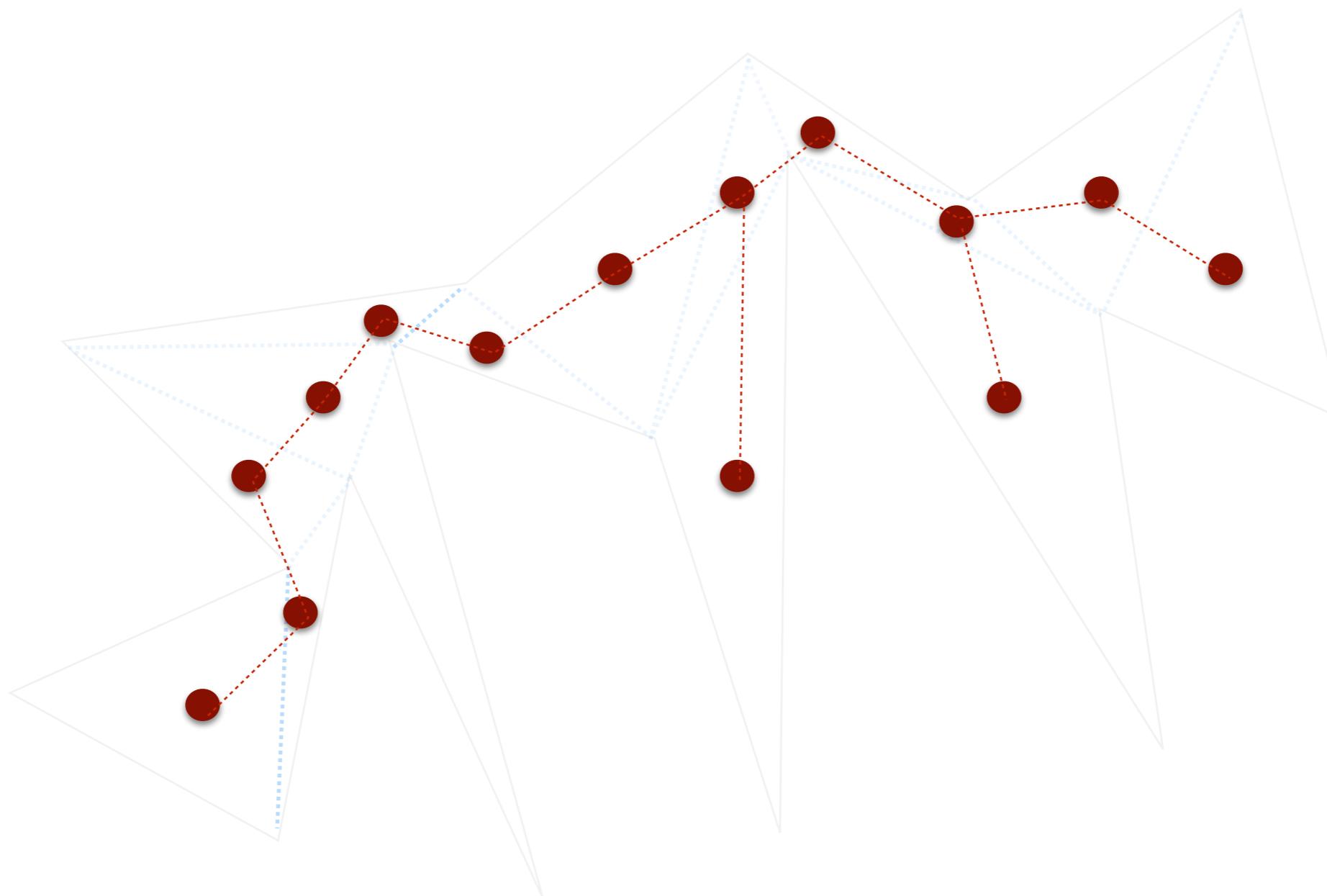
Theorem: Any simple polygon has at least two ears.

Proof: Triangulate P. Consider the dual graph.



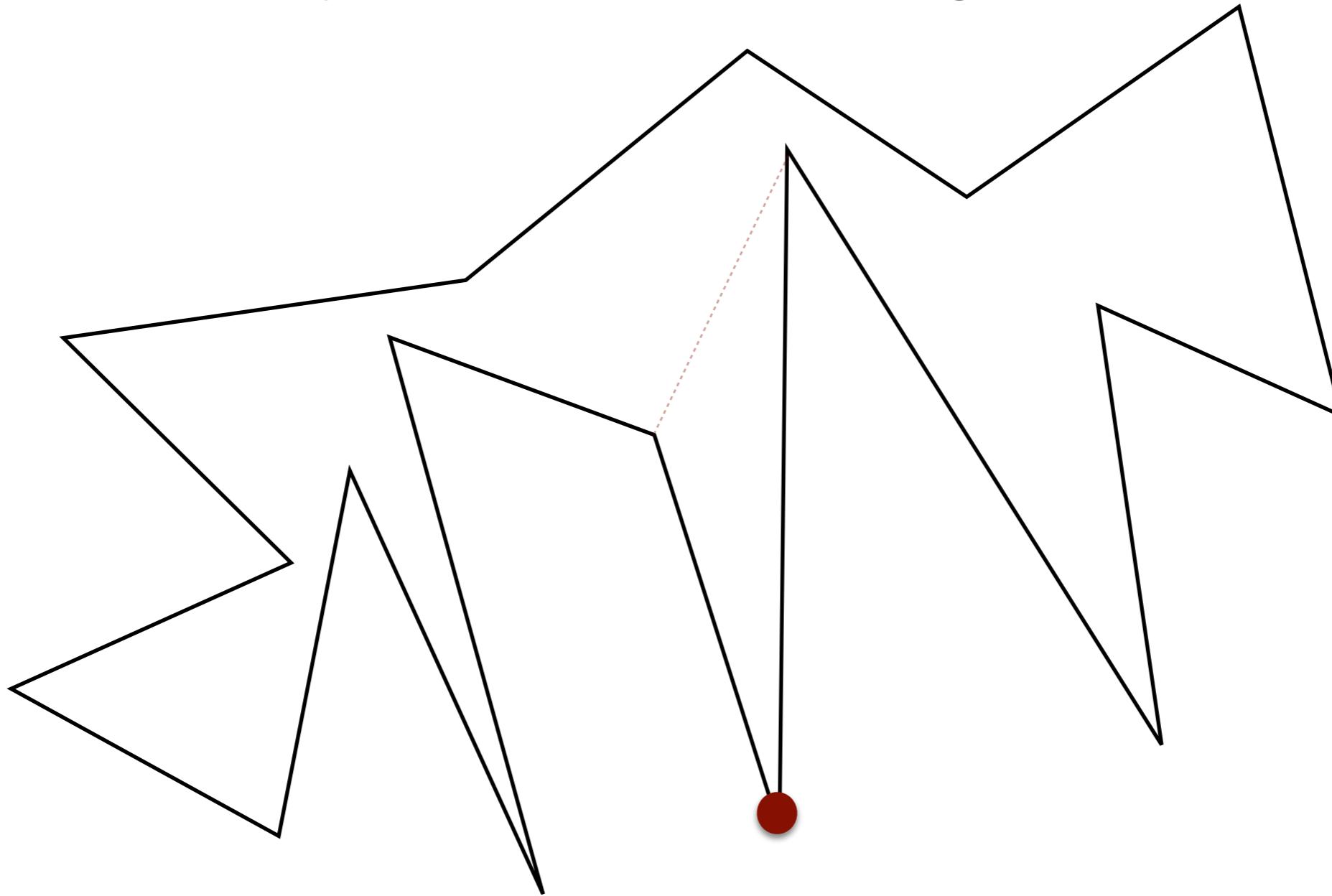
Theorem: Any simple polygon has at least two ears.

Proof: Triangulate P. Consider the dual graph. The dual graph is a tree. Any tree has at least two leaves. A leaf => ear



Algorithm 3: Triangulation by finding ears

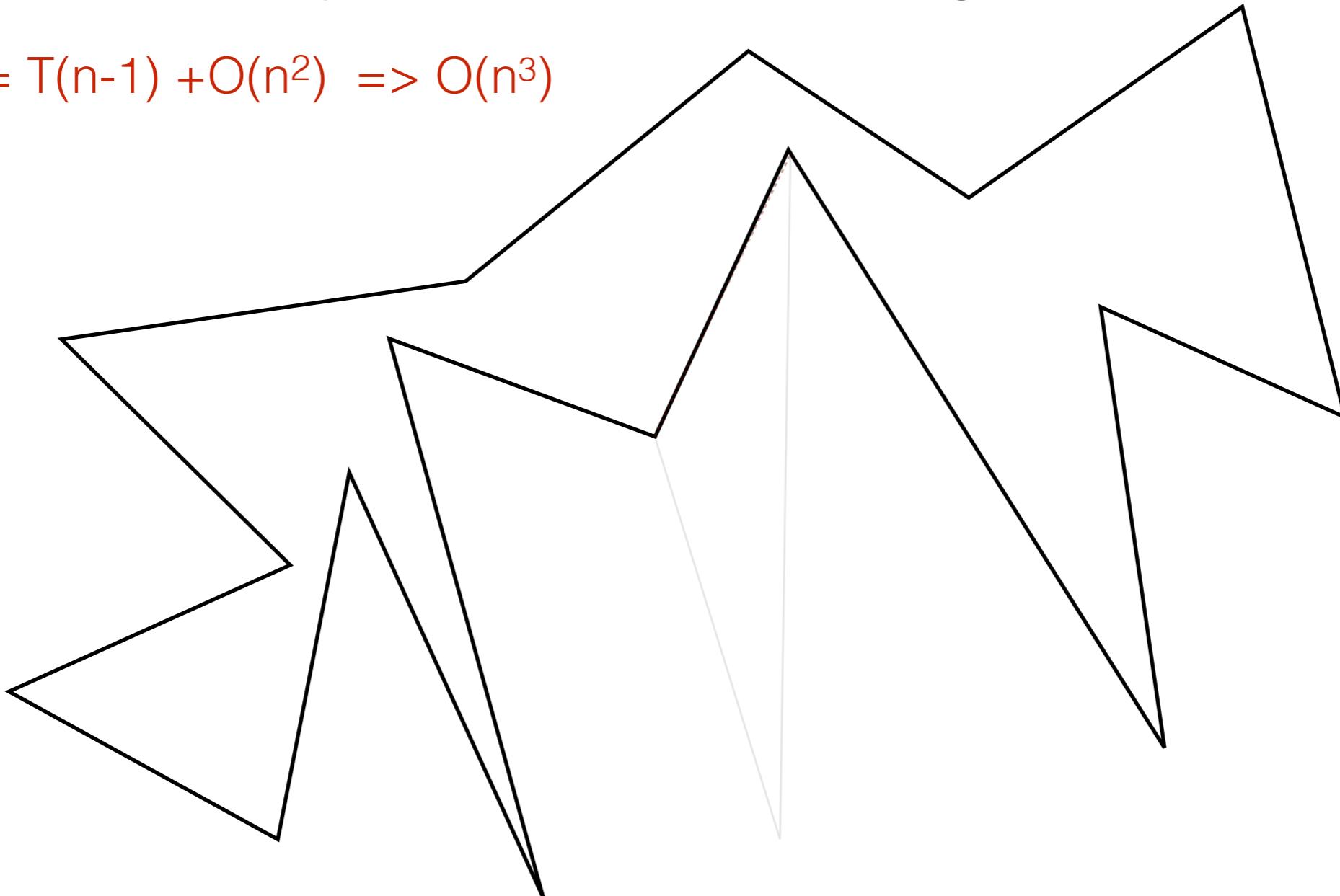
- Traverse P and for each point p, determine if it's an ear
- When find a ear p: recurse on the remaining P



Algorithm 3: Triangulation by finding ears

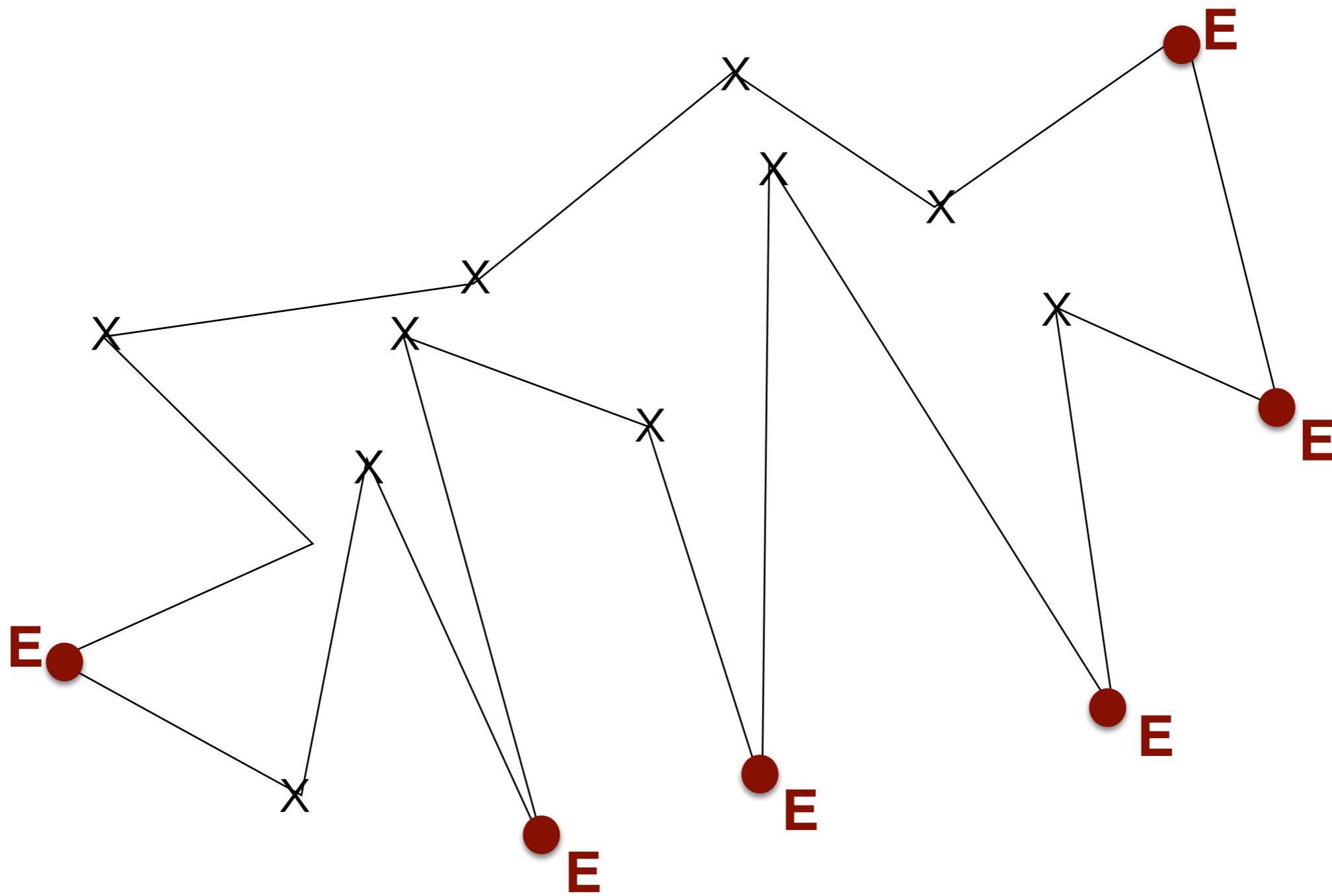
- $O(n)$
- Traverse P and for each point p, determine if it's an ear
 - When find a ear p: recurse on the remaining P

$$T(n) = T(n-1) + O(n^2) \Rightarrow O(n^3)$$



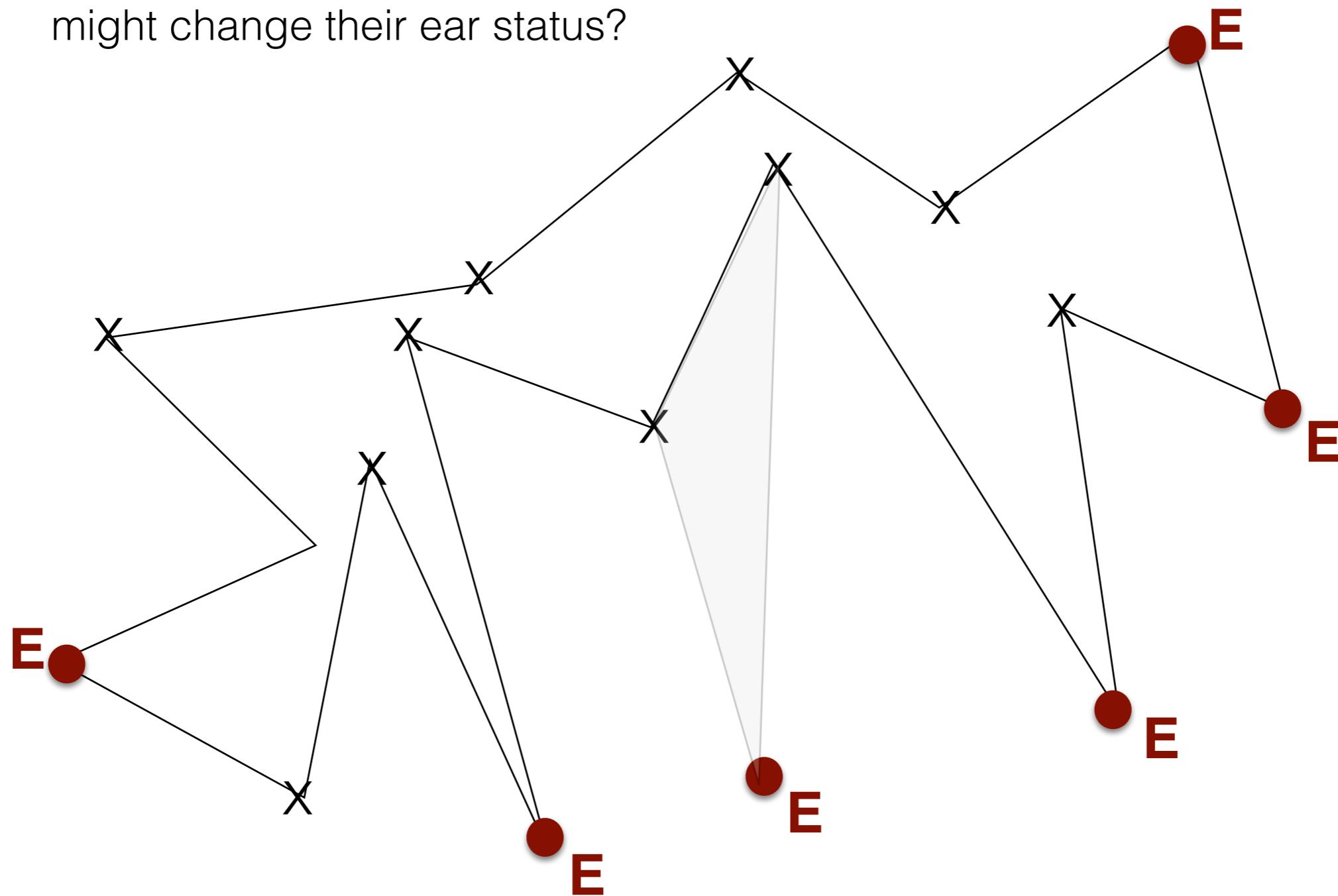
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time



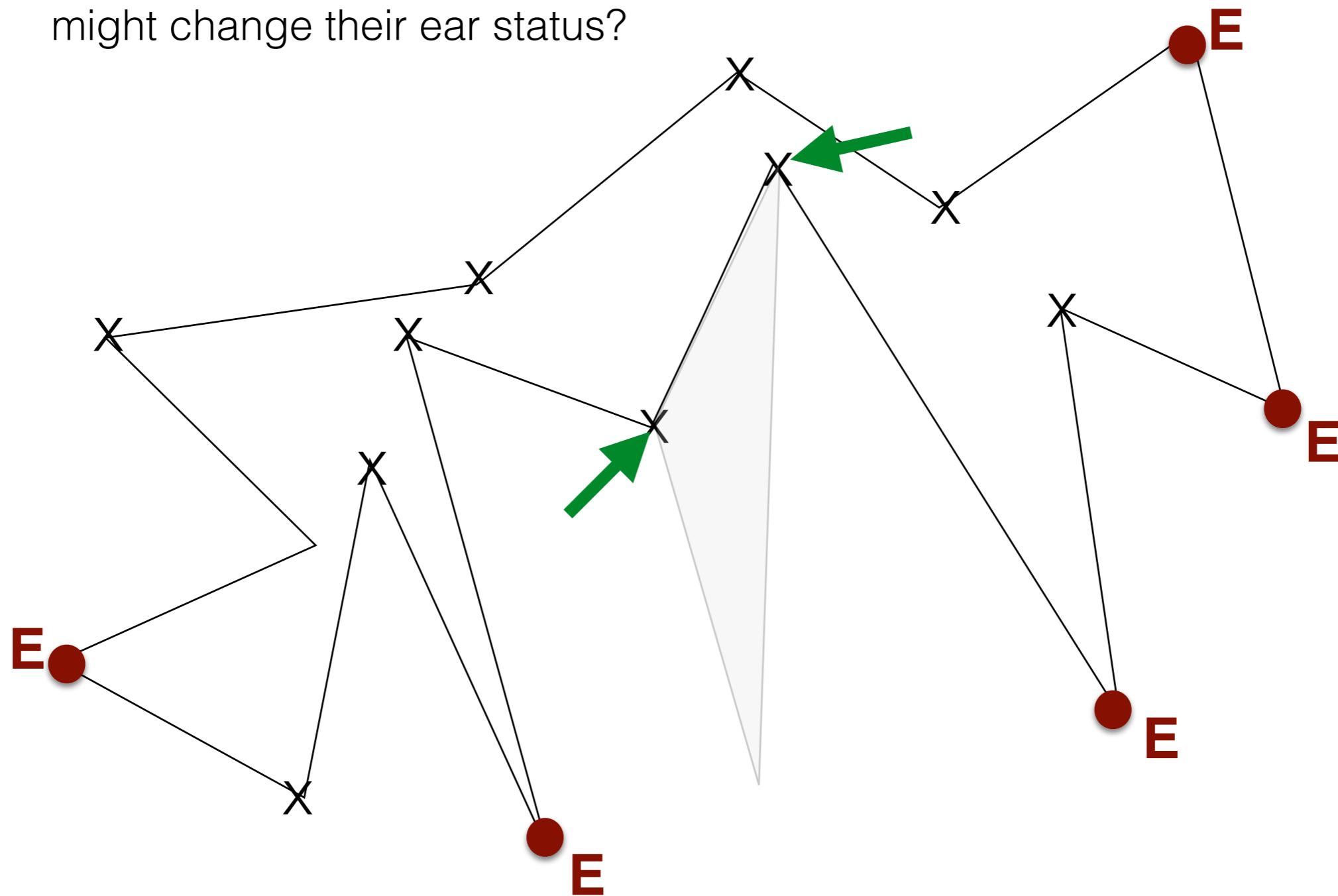
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
 - When you remove an ear tip from the polygon, which vertices might change their ear status?



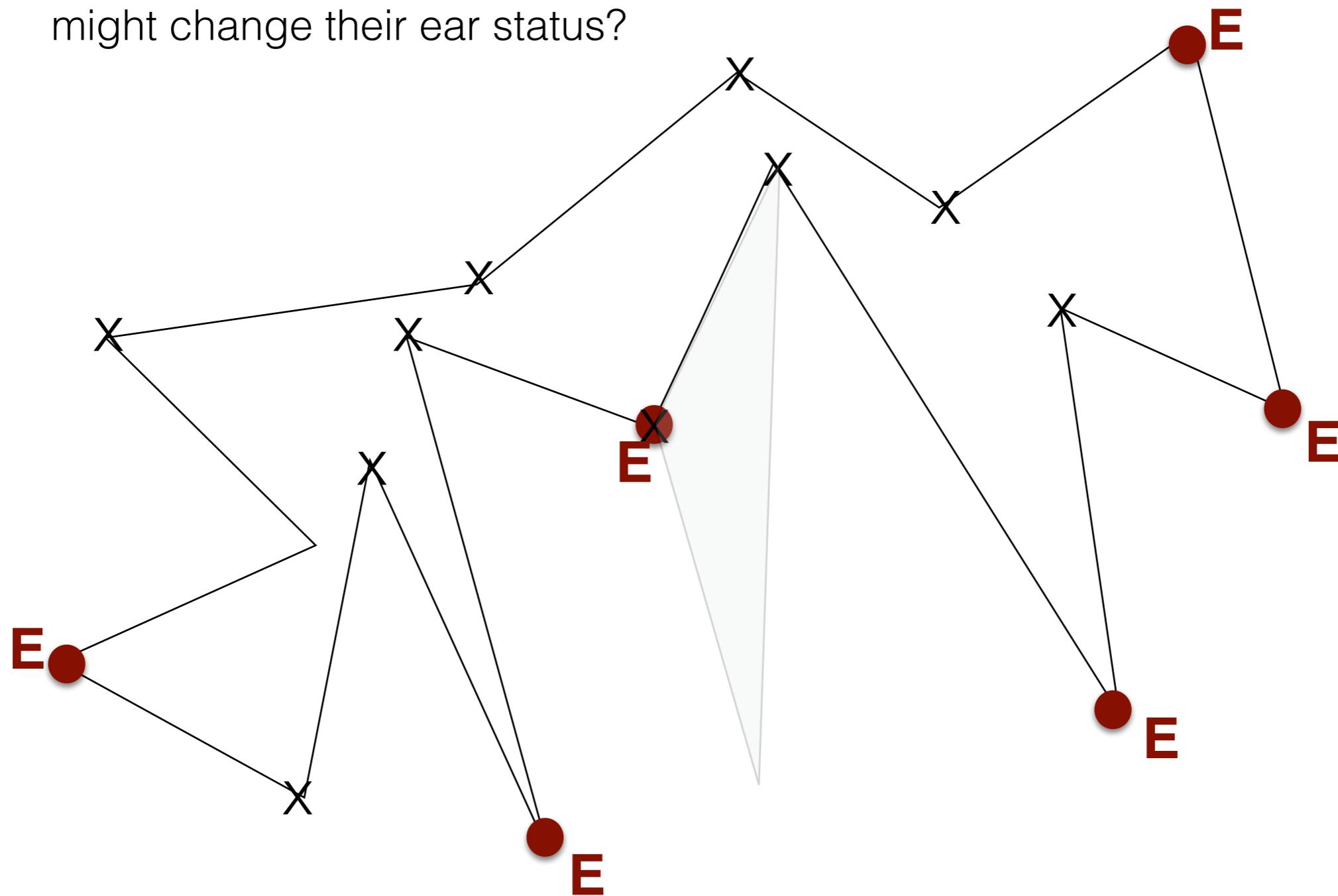
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
 - When you remove an ear tip from the polygon, which vertices might change their ear status?



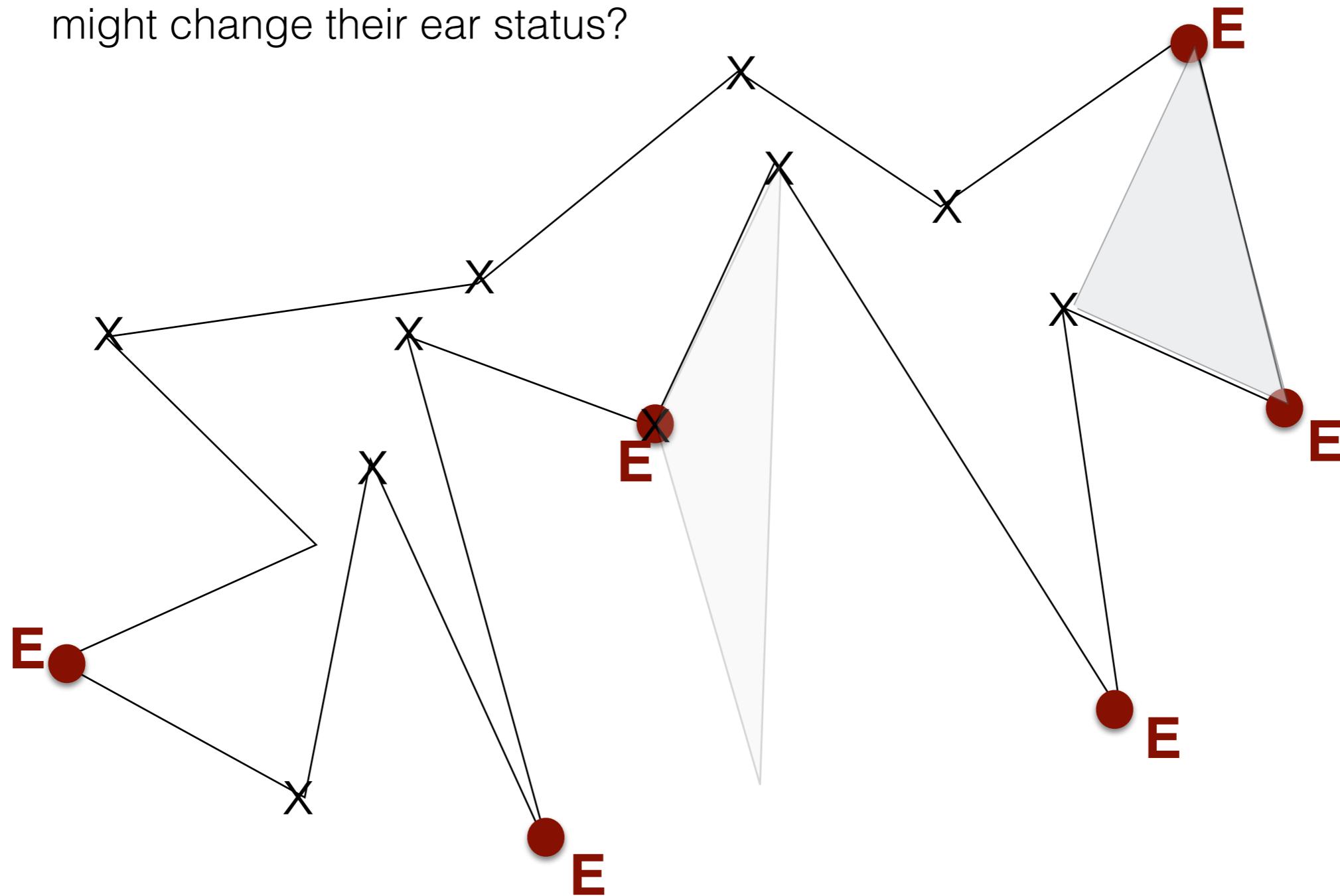
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
 - When you remove an ear tip from the polygon, which vertices might change their ear status?



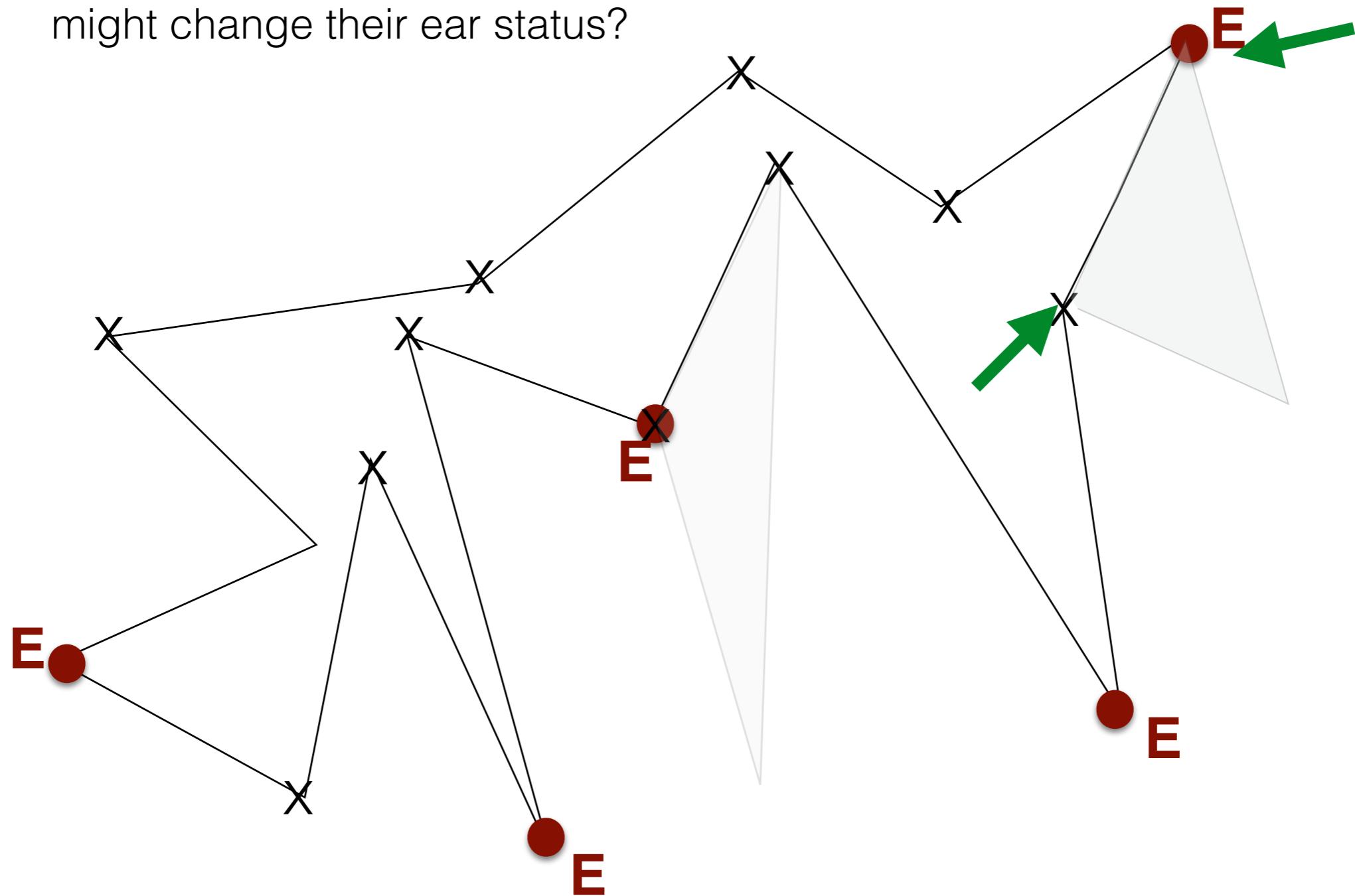
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
 - When you remove an ear tip from the polygon, which vertices might change their ear status?



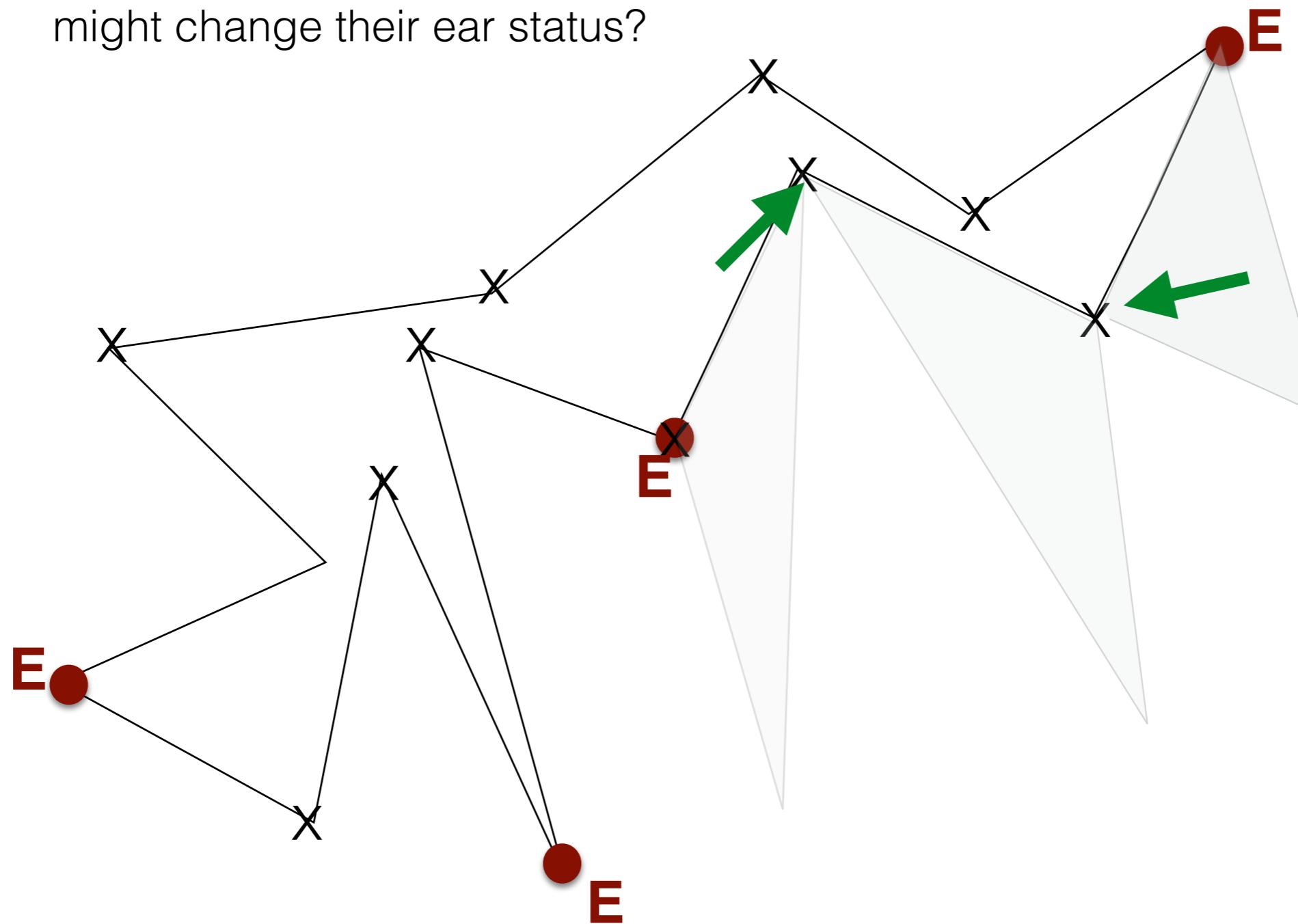
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
 - When you remove an ear tip from the polygon, which vertices might change their ear status?



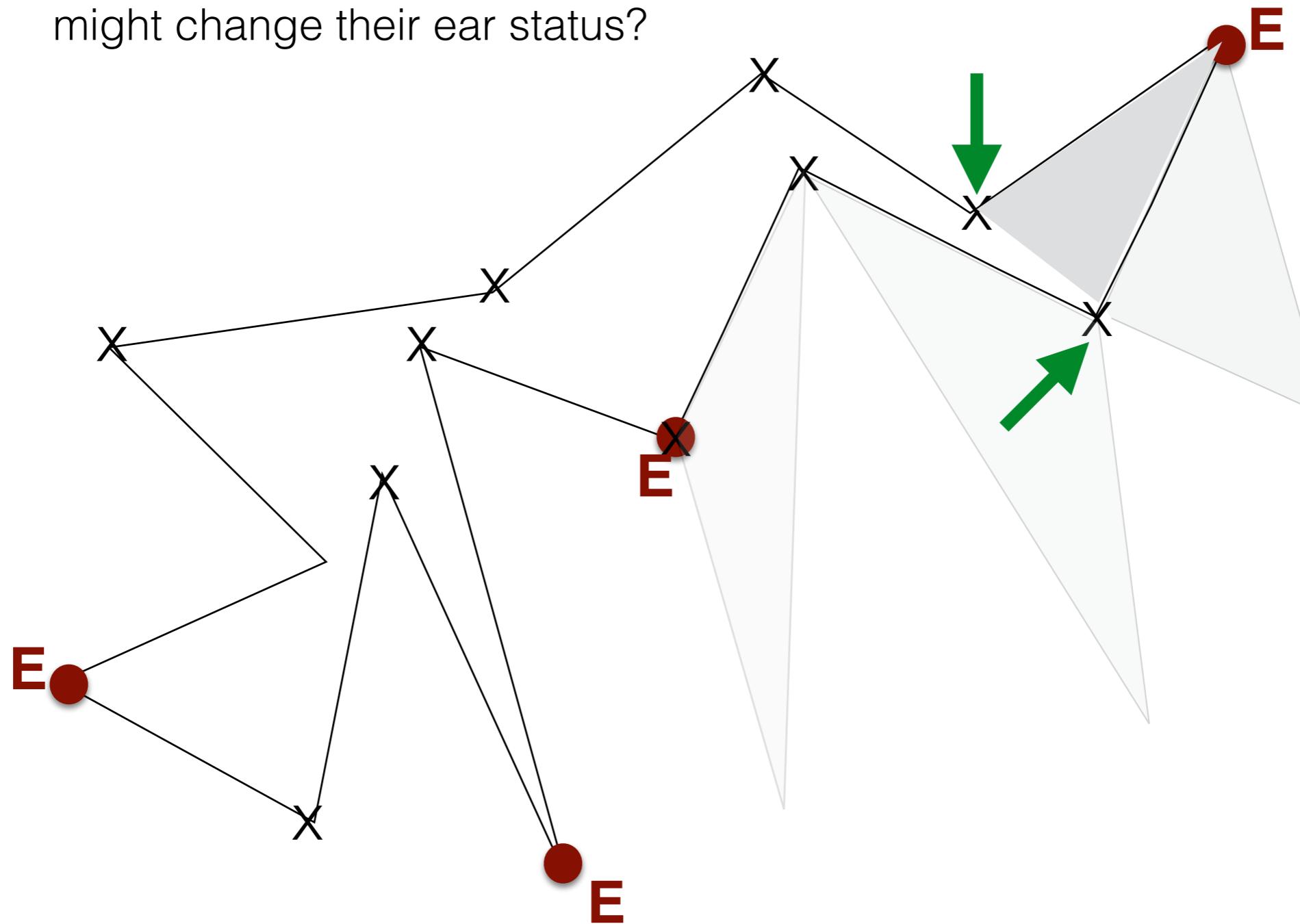
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
 - When you remove an ear tip from the polygon, which vertices might change their ear status?



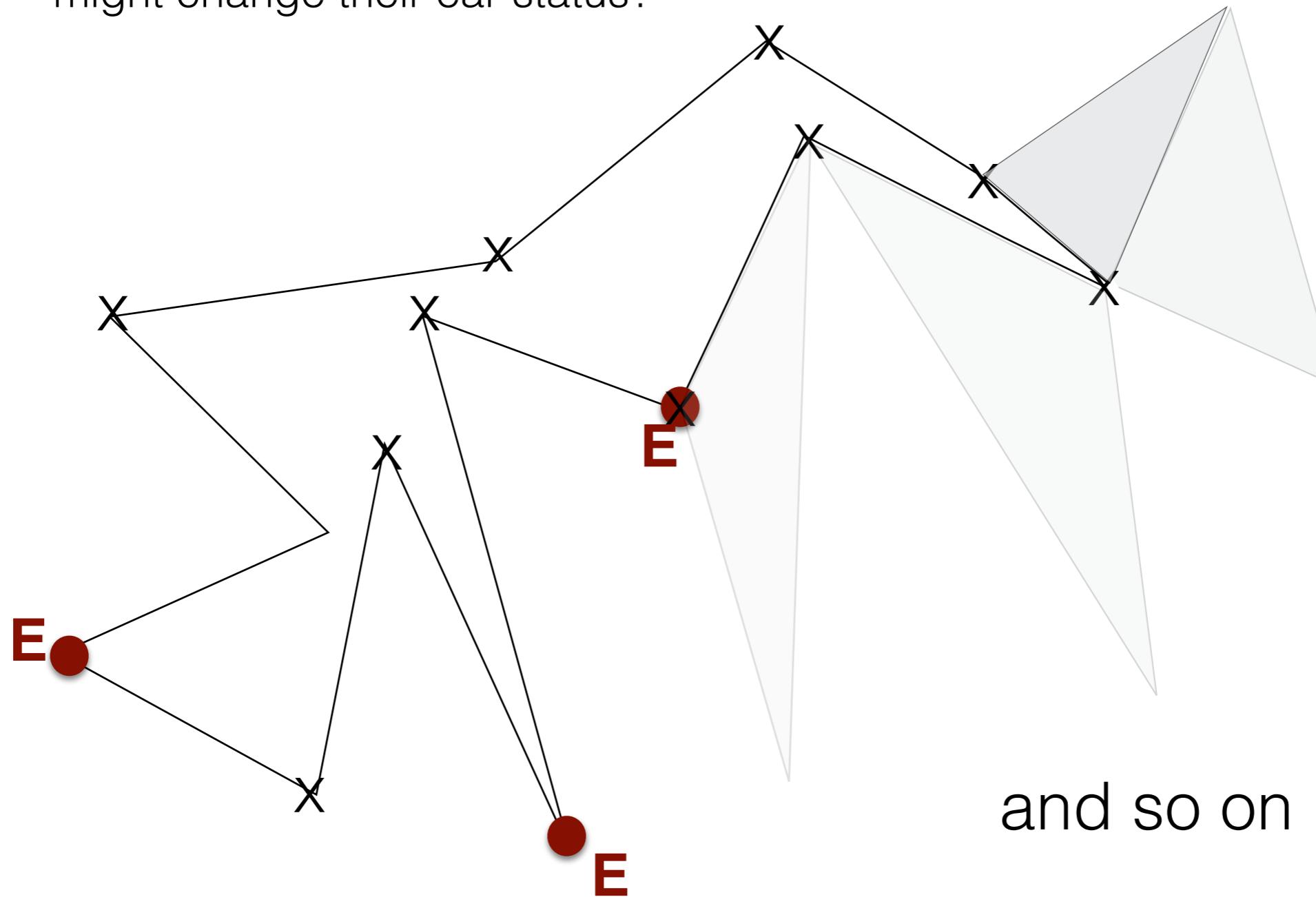
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
 - When you remove an ear tip from the polygon, which vertices might change their ear status?



Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
 - When you remove an ear tip from the polygon, which vertices might change their ear status?



Algorithm 4: Improved ear removal

- Initialize the ear tip status of each vertex of P
- while $n > 3$ do:
 - locate an ear tip p
 - output diagonal p^-p^+
 - delete p
 - update ear tip status of p^- and p^+

Or, with a bit more detail,

Algorithm 4: Improved ear removal

```
//Initialize the ear tip status of each vertex of P
```

- for i=0, i<n, i++
 - p_- is ear if isDiagonal($p^- p^+$)
- while n>3 do:
 - i=0
 - while i < P.size():
 - if p[i] is ear:
 - output diagonal $p[i - 1]p[i + 1]$
 - update ear status for $p[i - 1]$ and $p[i + 1]$
 - delete p[i] from P and set n = n-1
 - else: i++

Algorithm 4: Improved ear removal

//Initialize the ear tip status of each vertex of P

- for $i=0, i < n, i++$
 - p_- is ear if $\text{isDiagonal}(p^- p^+)$
- while $n > 3$ do:
 - $i=0$
 - while $i < P.size()$:
 - if $p[i]$ is ear:
 - output diagonal $p[i - 1]p[i + 1]$
 - update ear status for $p[i - 1]$ and $p[i + 1]$ ← this takes $O(n)$
 - delete $p[i]$ from P and set $n = n-1$
 - else: $i++$

$\uparrow O(n^2)$

a vertex causes ear status updates
for 2 other vertices

Overall: $O(n^2)$ time

$O(n)$ ear status updates

History of Polygon Triangulation

- Early algorithms: $O(n^4)$, $O(n^3)$, $O(n^2)$
- Several $O(n \lg n)$ algorithms known  practical
- ...
- Many papers with improved bounds  not practical
- ...
- 1991: Bernard Chazelle (Princeton) gave an $O(n)$ algorithm
 - <https://www.cs.princeton.edu/~chazelle/pubs/polygon-triang.pdf>
 - Ridiculously complicated, not practical
 - $O(1)$ people actually understand it (seriously) (and I'm not one of them)
- No algorithm is known that is practical enough to run faster than the $O(n \lg n)$ algorithms
- OPEN problem
 - A practical algorithm that's theoretically better than $O(n \lg n)$.

An $O(n \lg n)$ Polygon Triangulation Algorithm

- Ingredients
 - Triangulate **monotone/unimonotone** polygons
 - Convert an arbitrary polygon into monotone/unimonotone polygons

Monotone chains

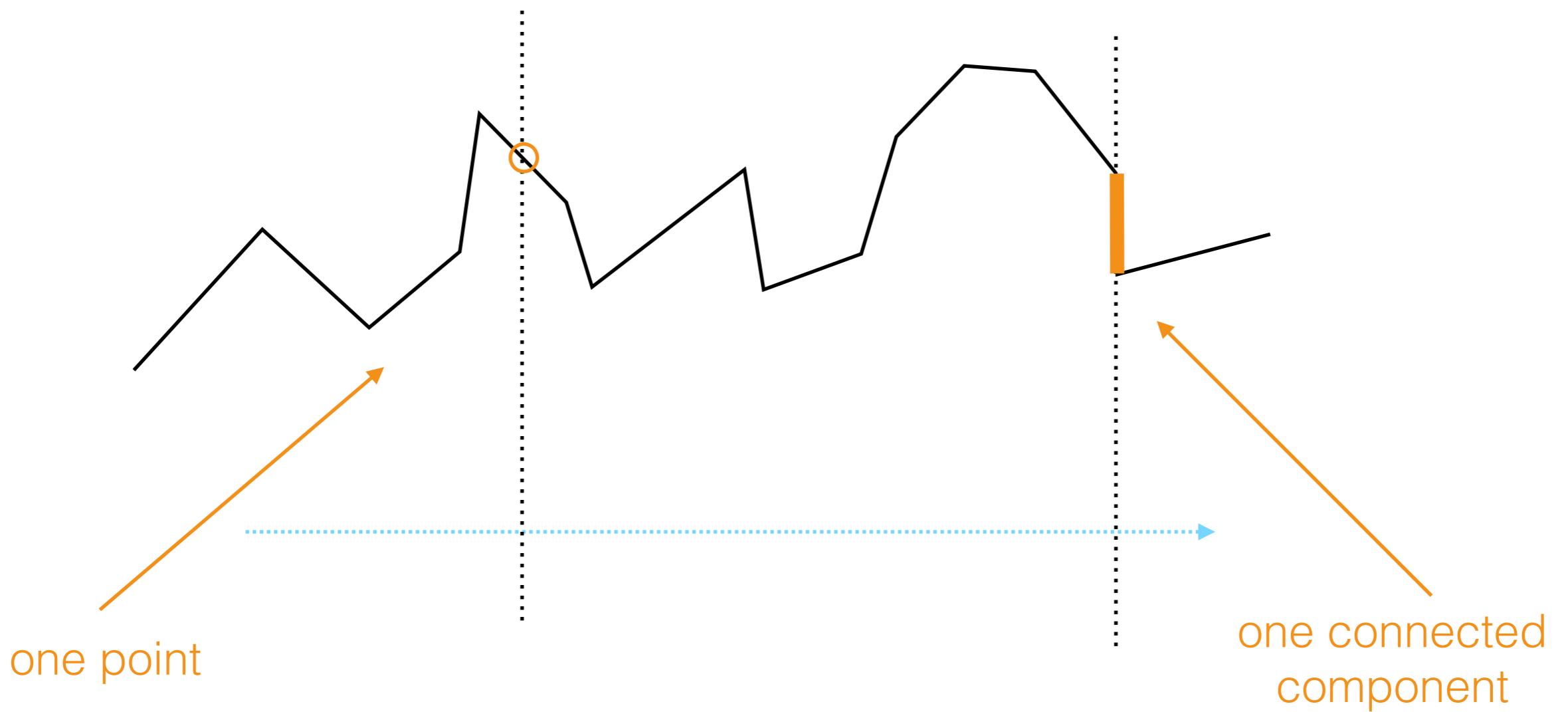
- Let P be a vector of points, not necessary a polygon (polygonal chain)

A polygonal chain is **x-monotone** if any line perpendicular to x-axis intersects it in one point (one connected component).

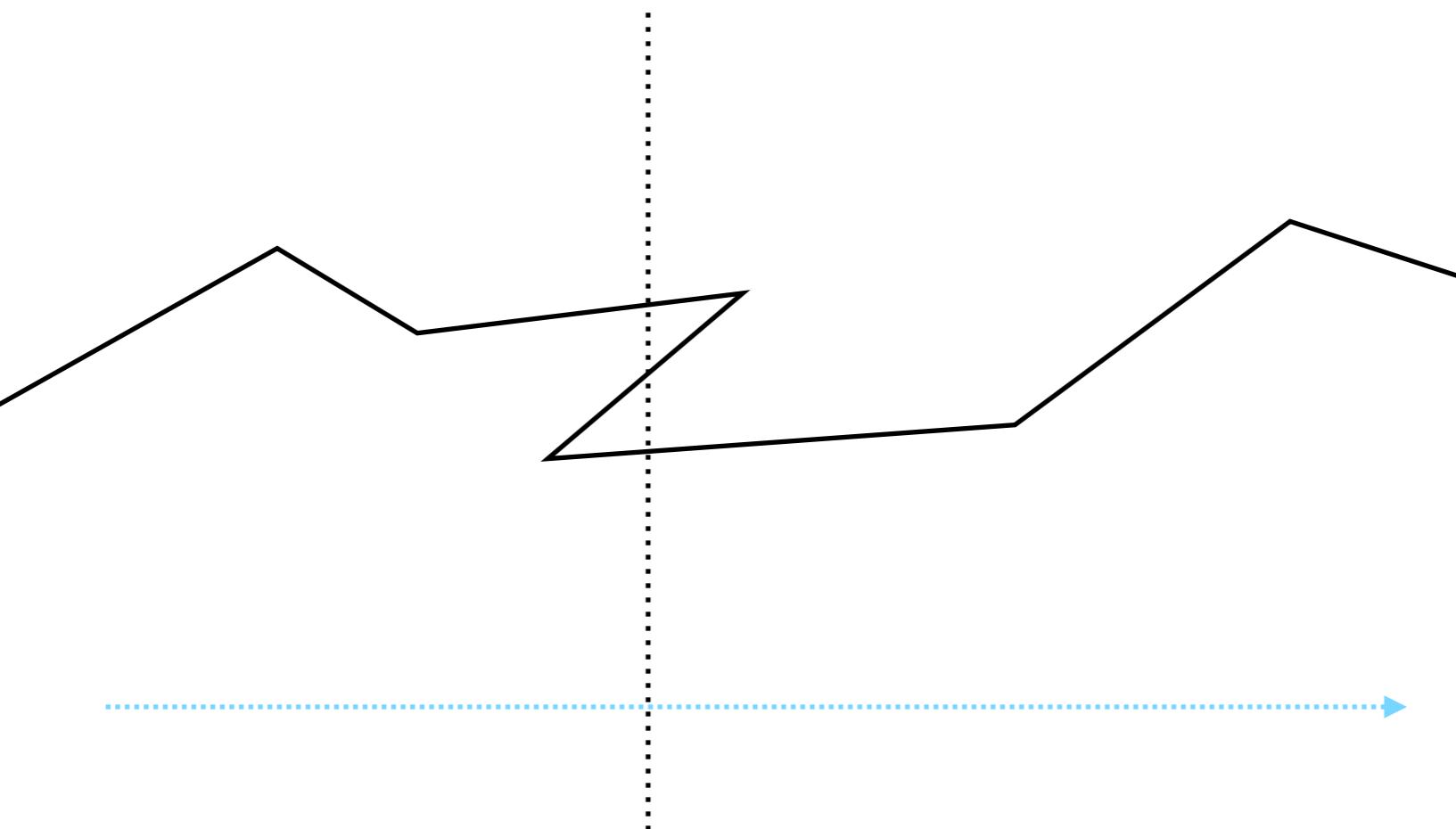


Monotone chains

A polygonal chain is **x-monotone** if any line perpendicular to **x-axis** intersects it in one point (one connected component).



Monotone chains



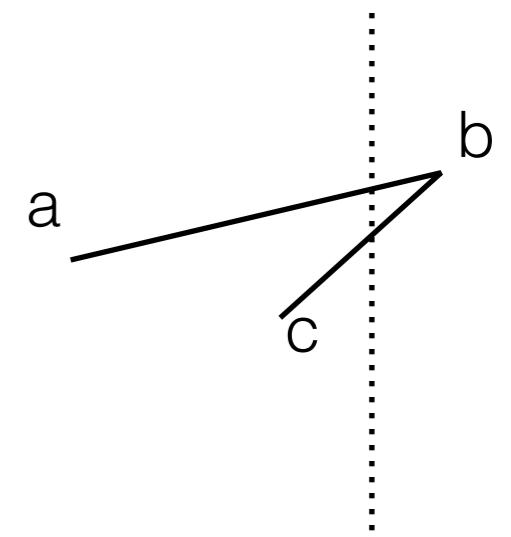
Not x-monotone

Monotone chains

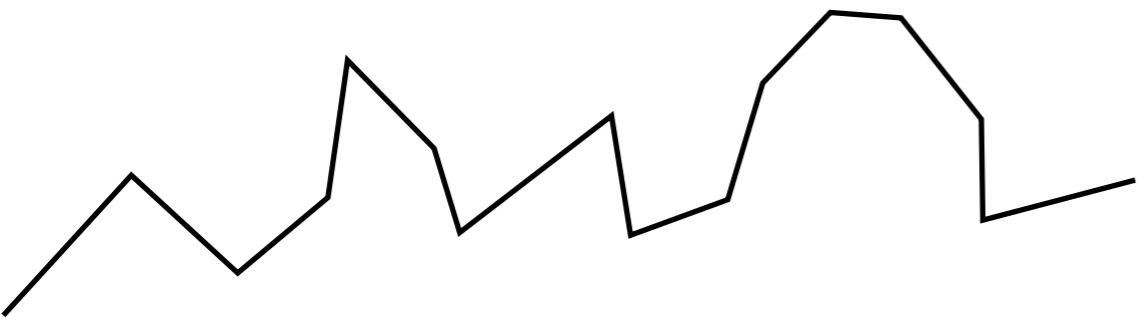
- Claim: Let u and v be the points on the chain with min/max x-coordinate. The vertices on the boundary of an x-monotone chain, going from u to v , are in x-order.



x-monotone



Monotone chains



x-monotone

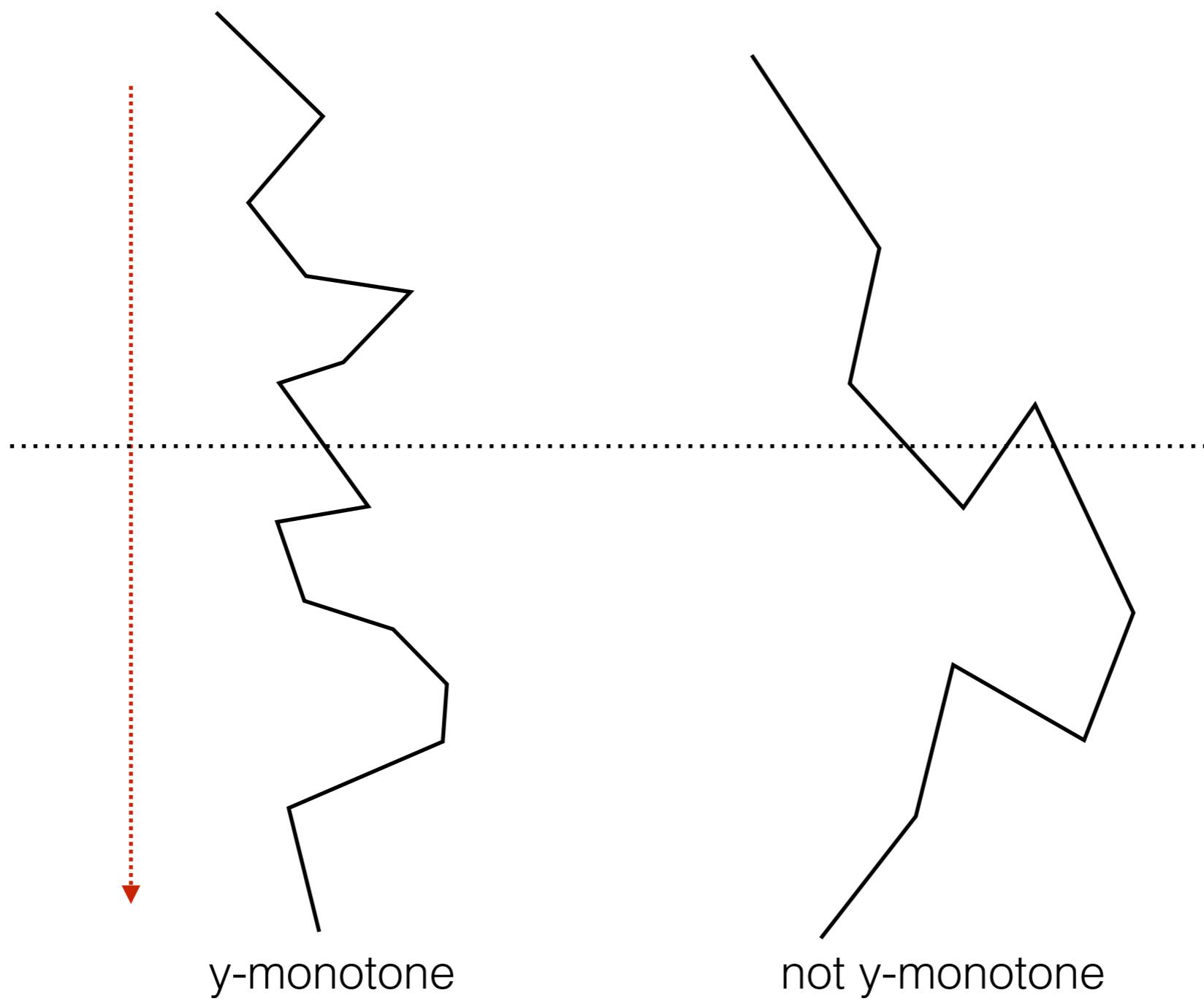


not x-monotone

As you travel along this chain, your x-coordinate is staying the same or increasing

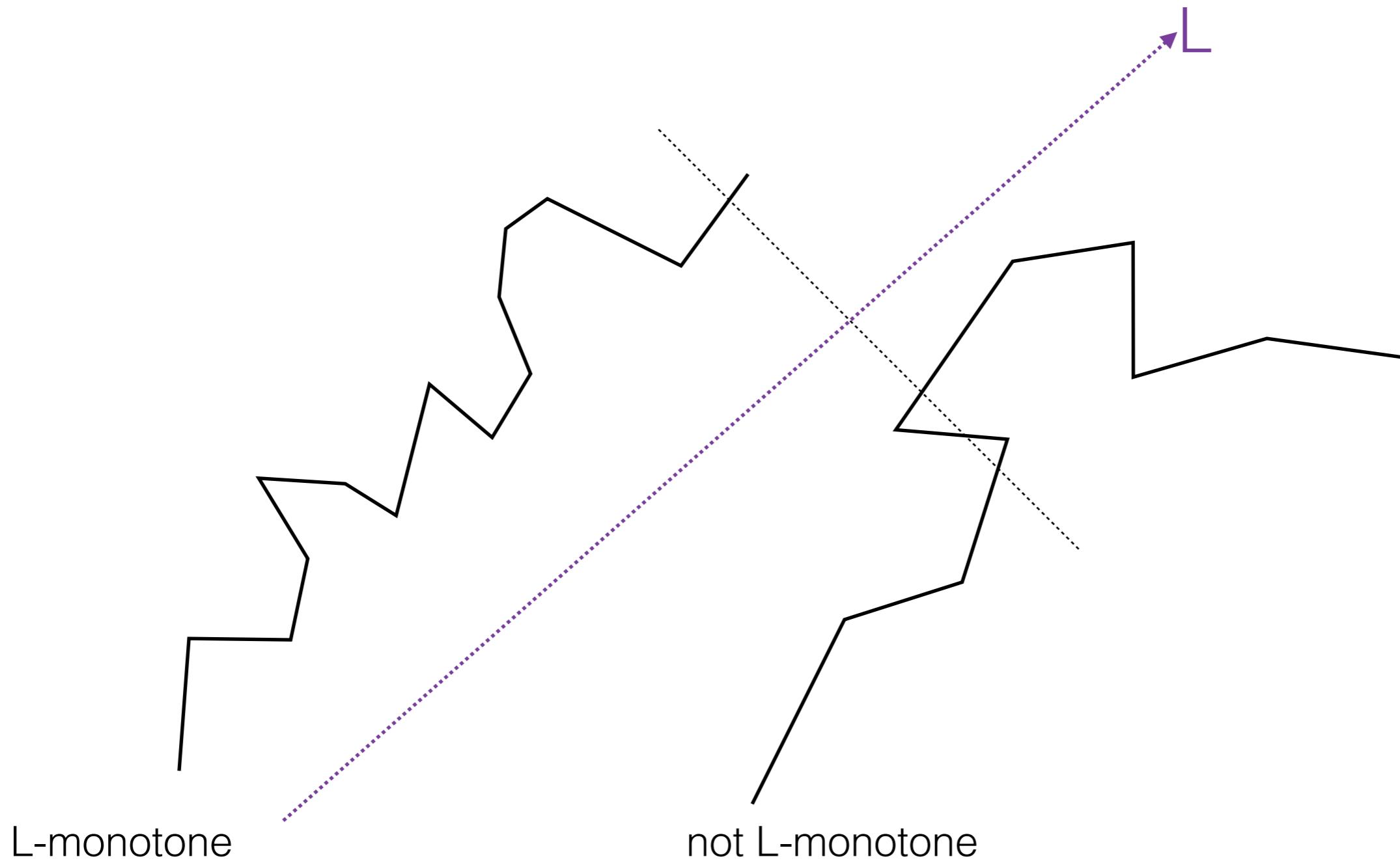
In general..

A polygonal chain is **y-monotone** if any line perpendicular to **y-axis** intersects it in one point (one connected component).



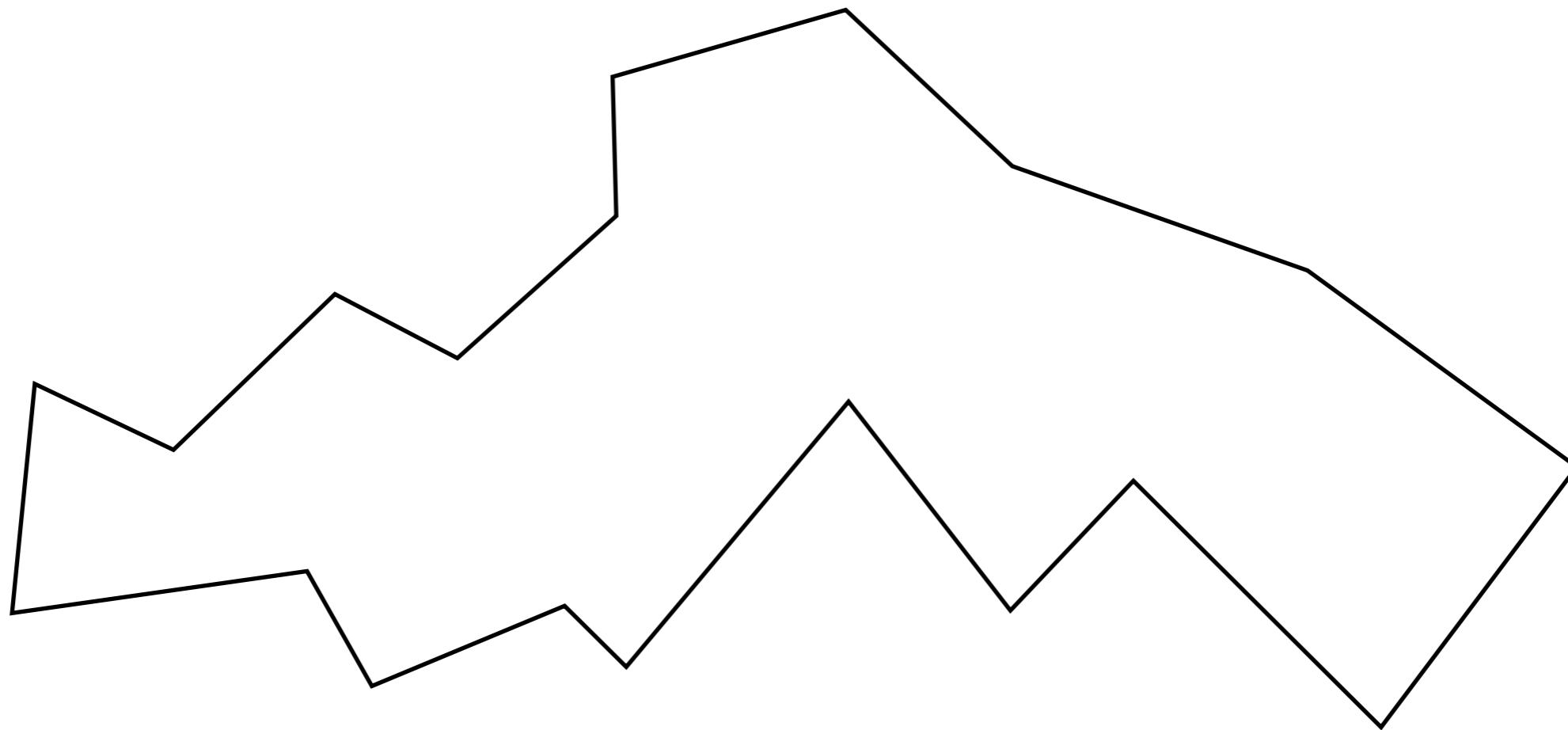
In general..

A polygonal chain is **L-monotone** if any line perpendicular to **line L** intersects it in one point (one connected component).



Monotone polygons

A **polygon** is x-monotone if its boundary can be split into two x-monotone chains.

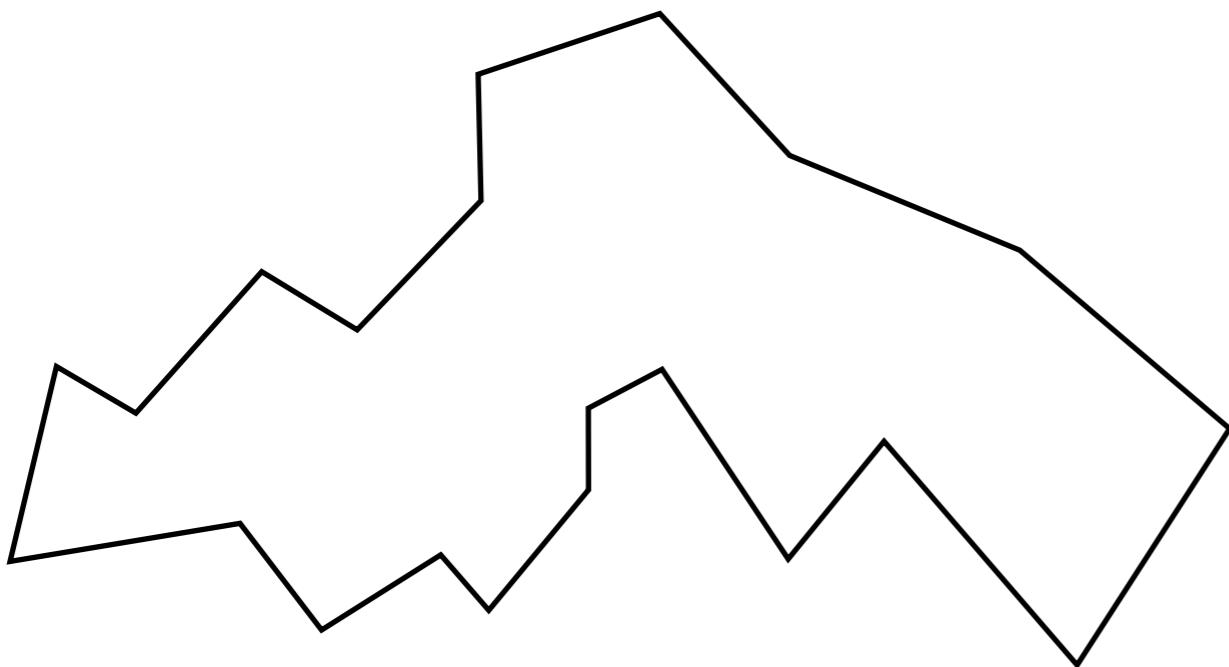


Monotone polygons

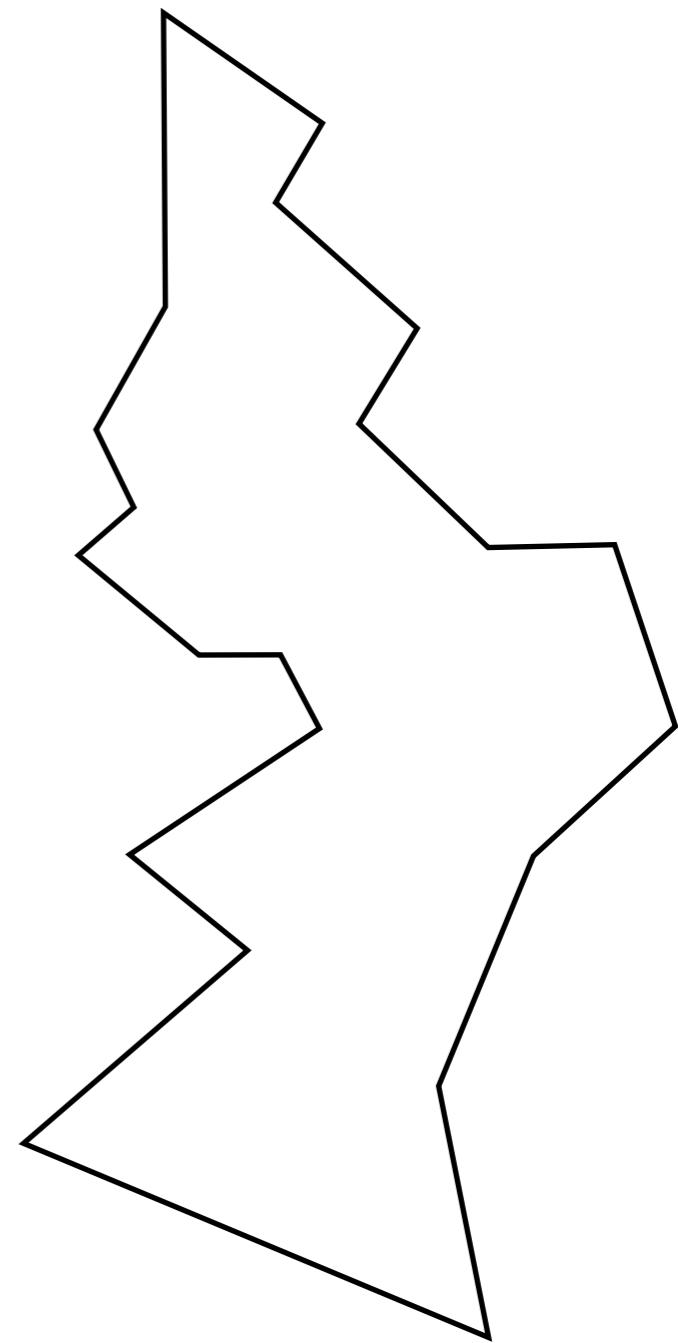
A polygon is x-monotone if its boundary can be split into two x-monotone chains.



Monotone polygons



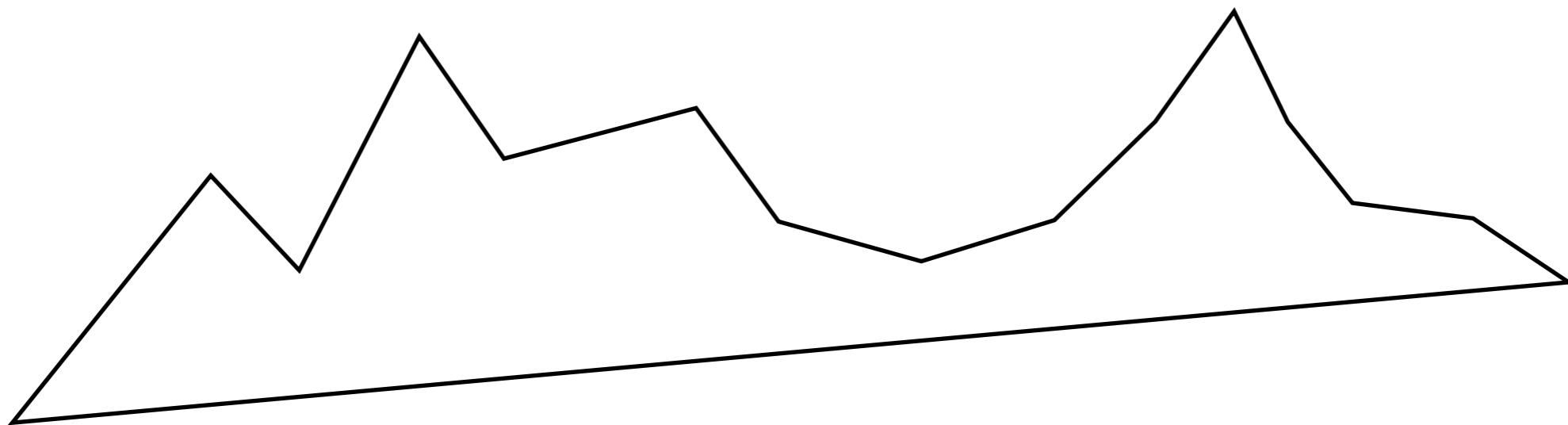
x-monotone



y-monotone

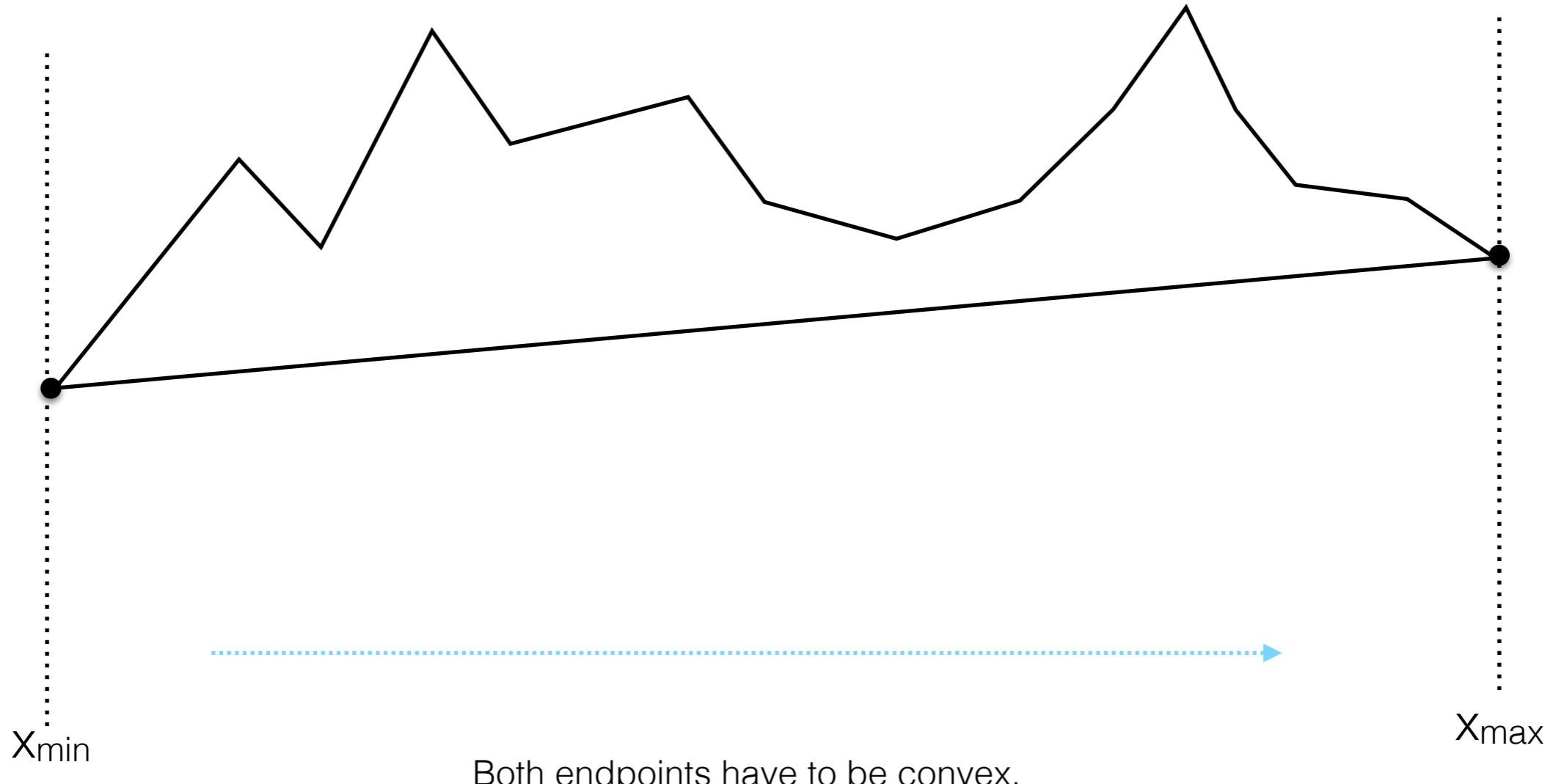
Monotone Mountains (unimonotone polygons)

A polygon is an **x-monotone mountain** if it is monotone and one of the two chains is a single segment.

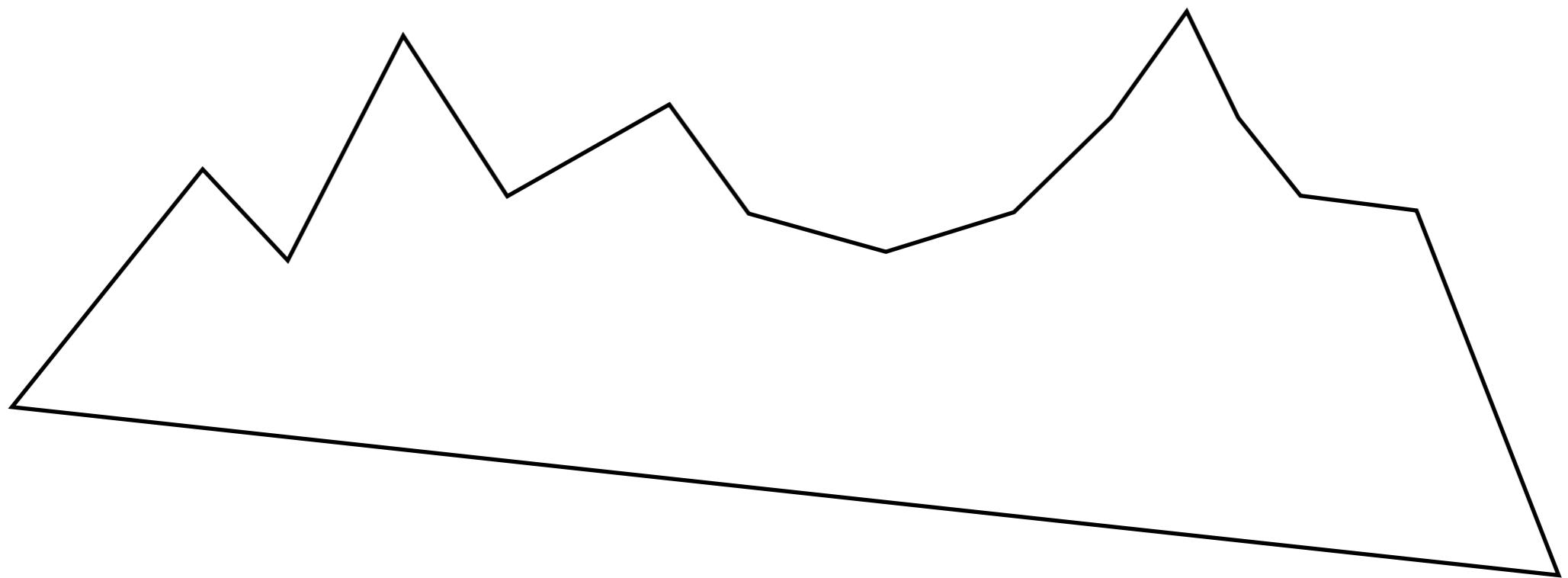


Monotone Mountains (unimonotone polygons)

A polygon is an **x-monotone mountain** if it is monotone and one of the two chains is a single segment.

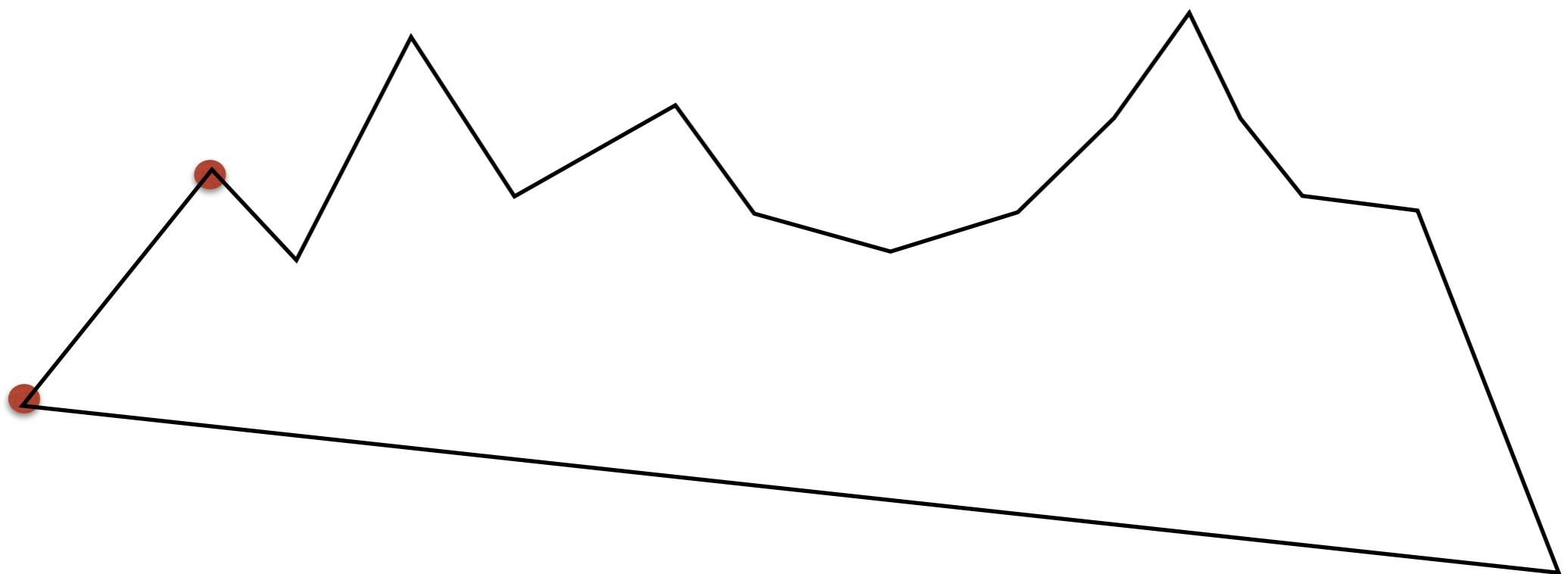


Monotone mountains are easy to triangulate!

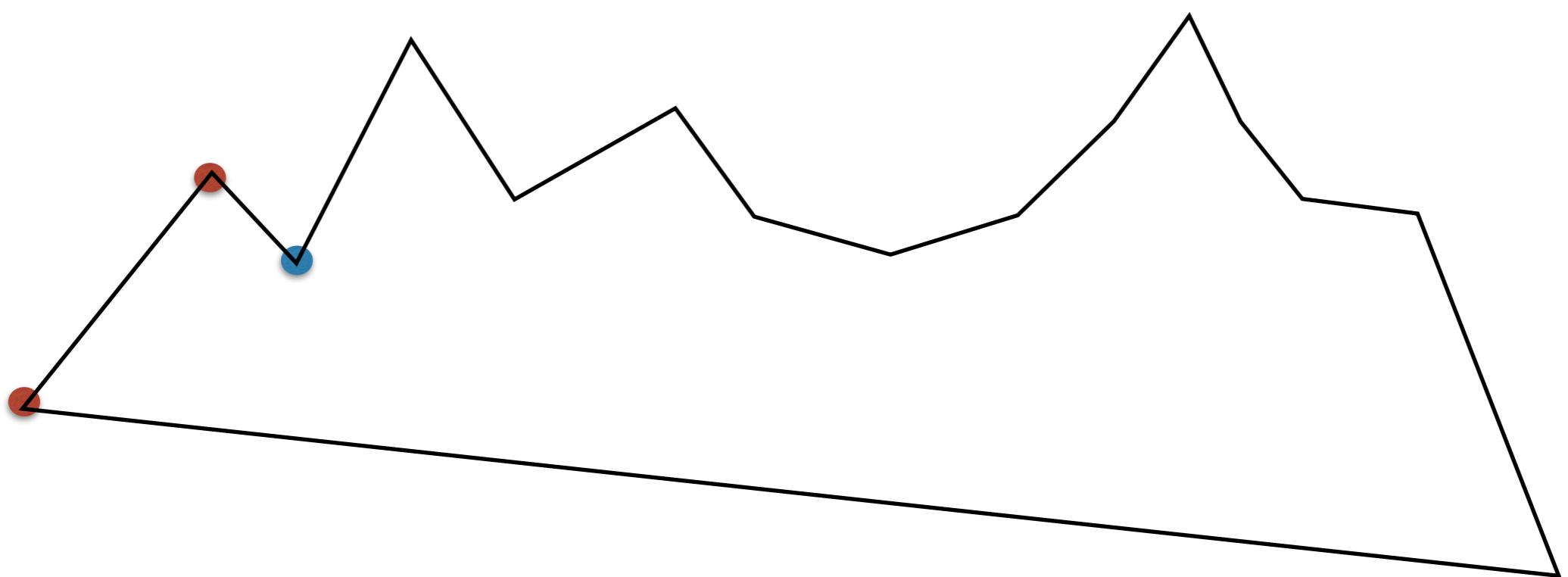


Class work: Let's come up with an algorithm (and analyze it).

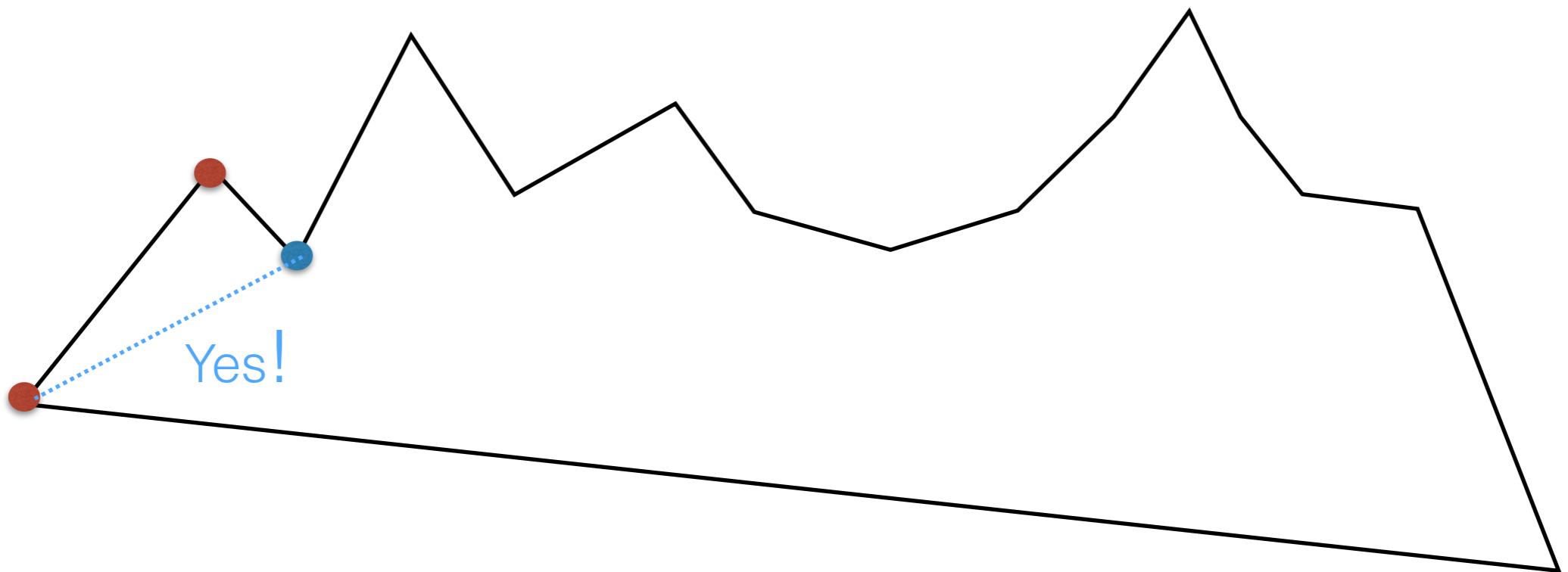
Monotone mountains are easy to triangulate!



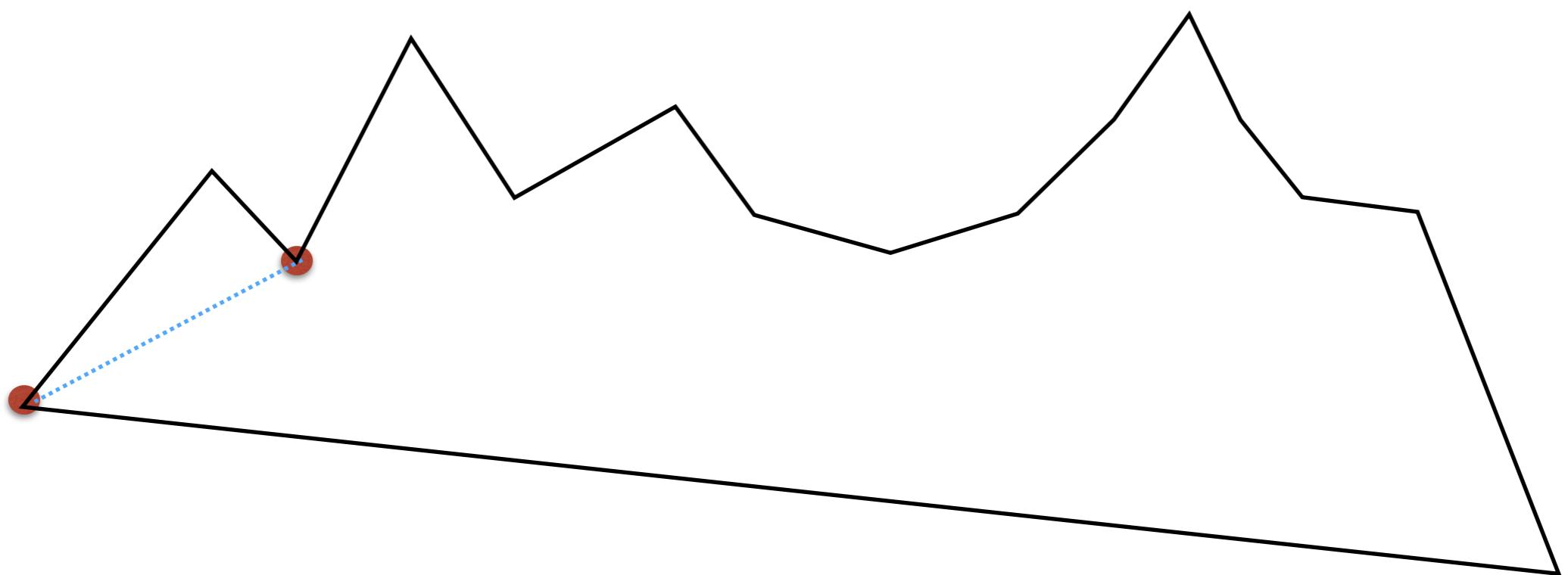
Monotone mountains are easy to triangulate!



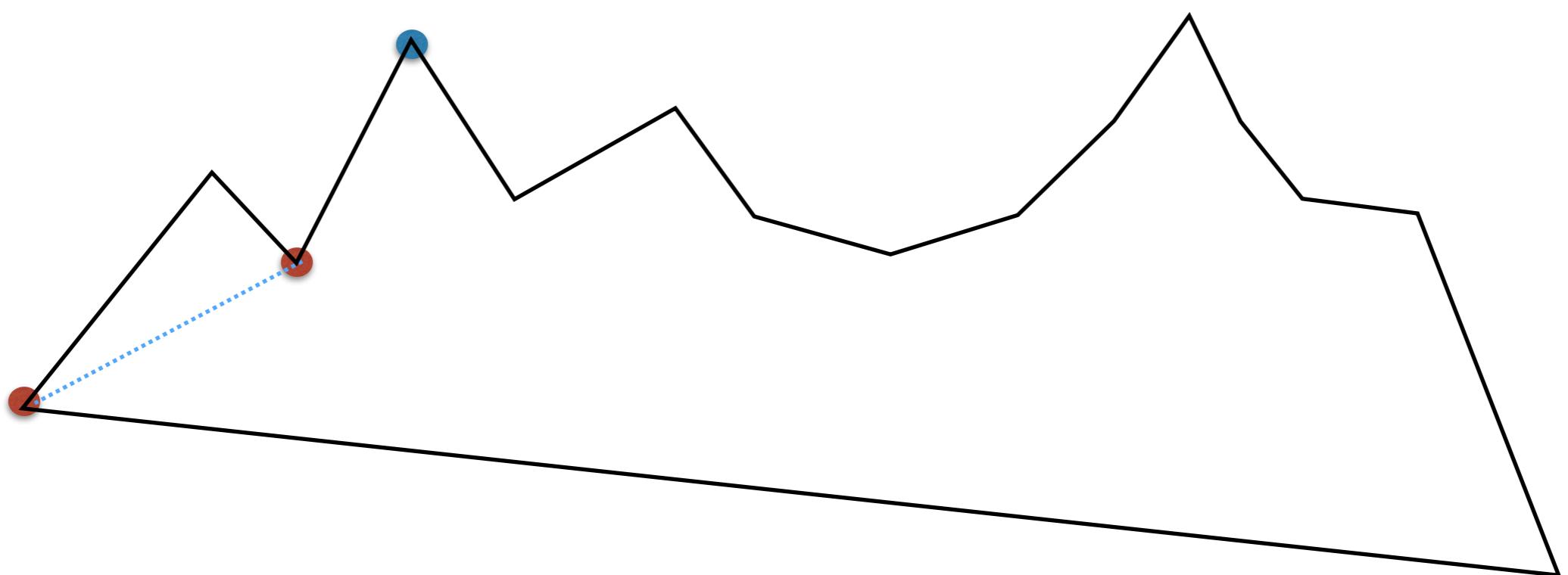
Monotone mountains are easy to triangulate!



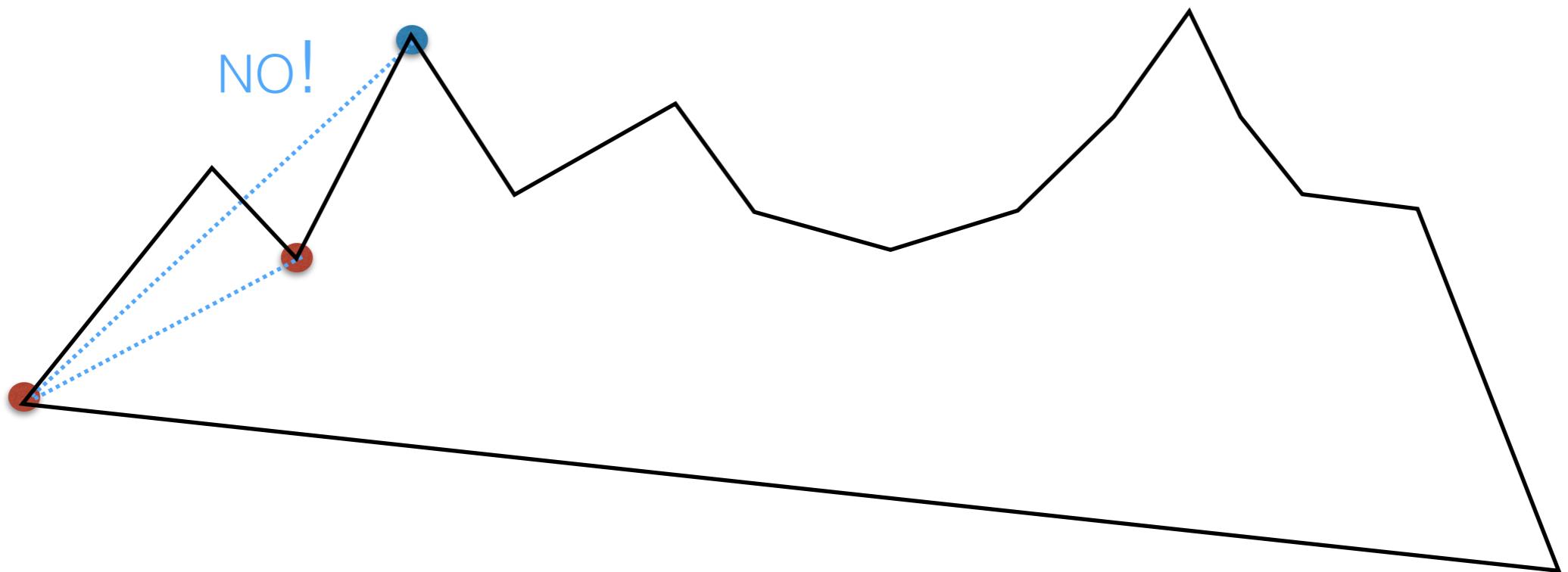
Monotone mountains are easy to triangulate!



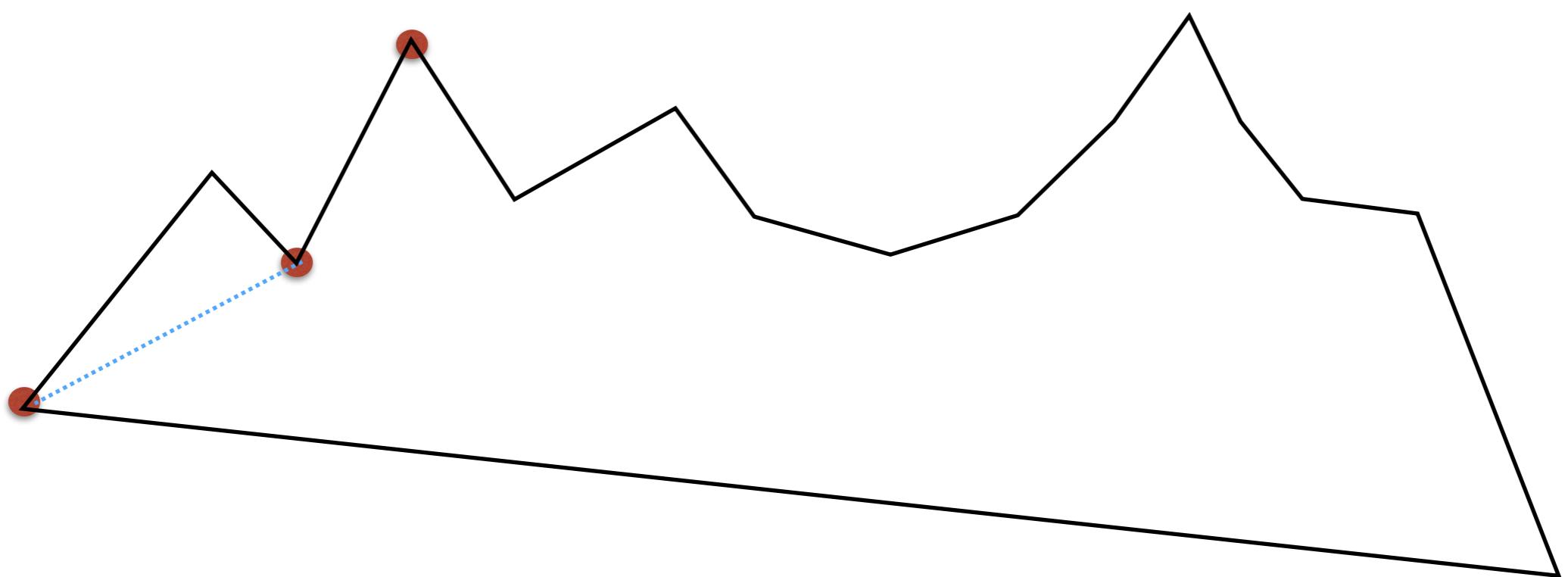
Monotone mountains are easy to triangulate!



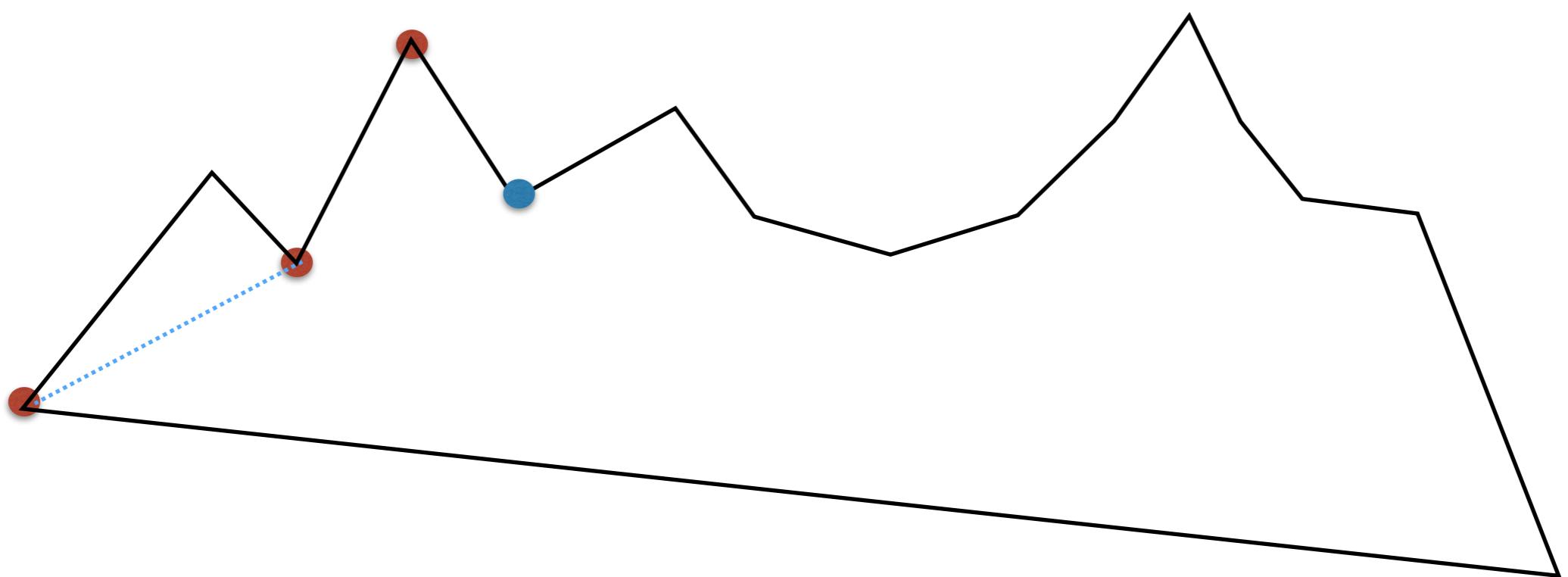
Monotone mountains are easy to triangulate!



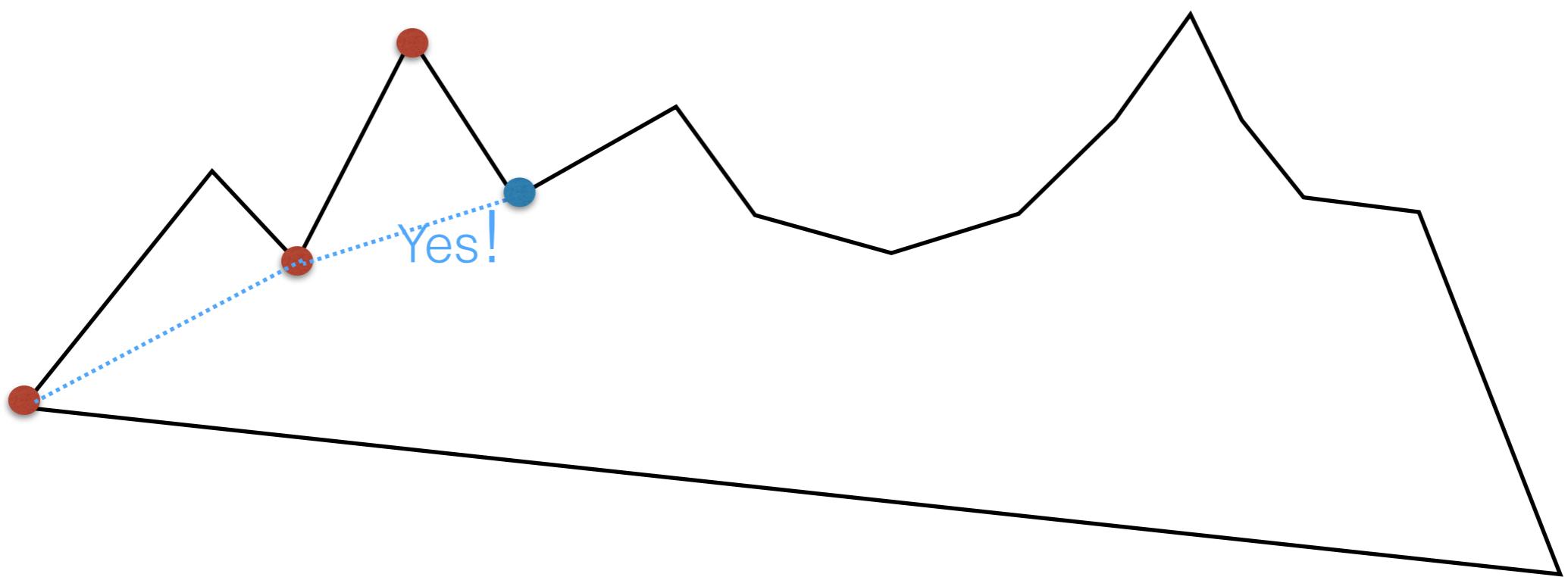
Monotone mountains are easy to triangulate!



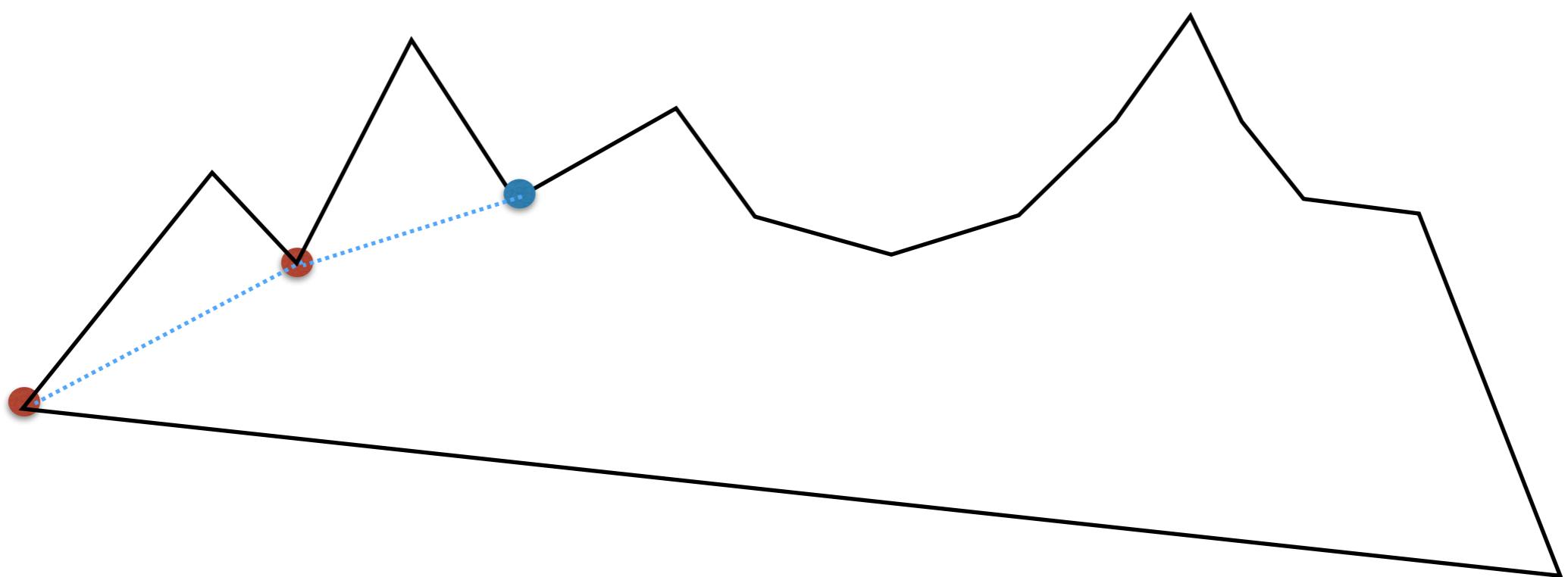
Monotone mountains are easy to triangulate!



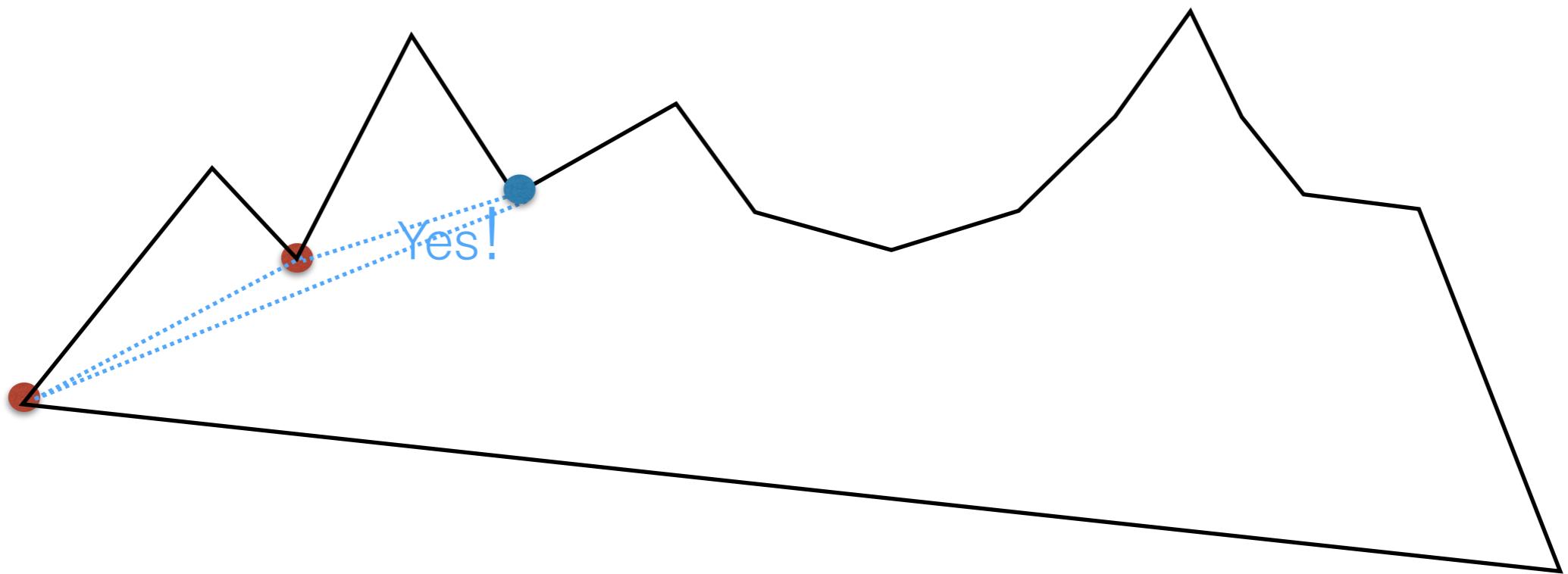
Monotone mountains are easy to triangulate!



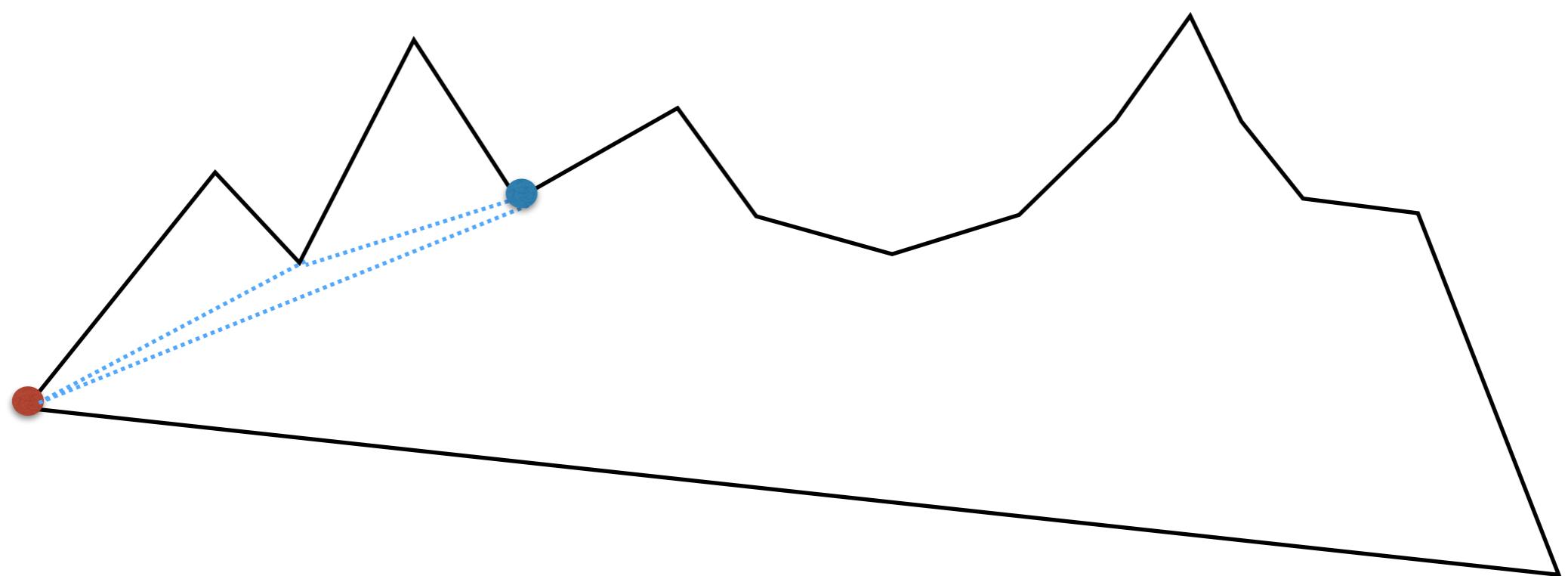
Monotone mountains are easy to triangulate!



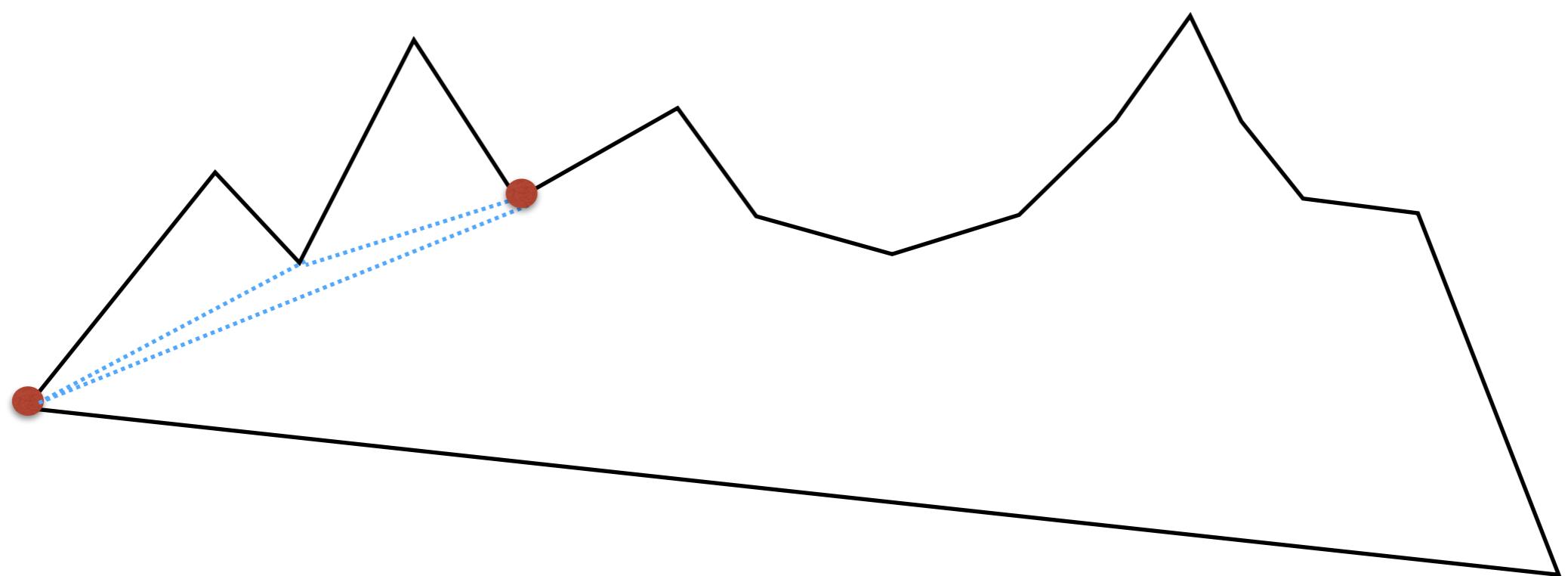
Monotone mountains are easy to triangulate!



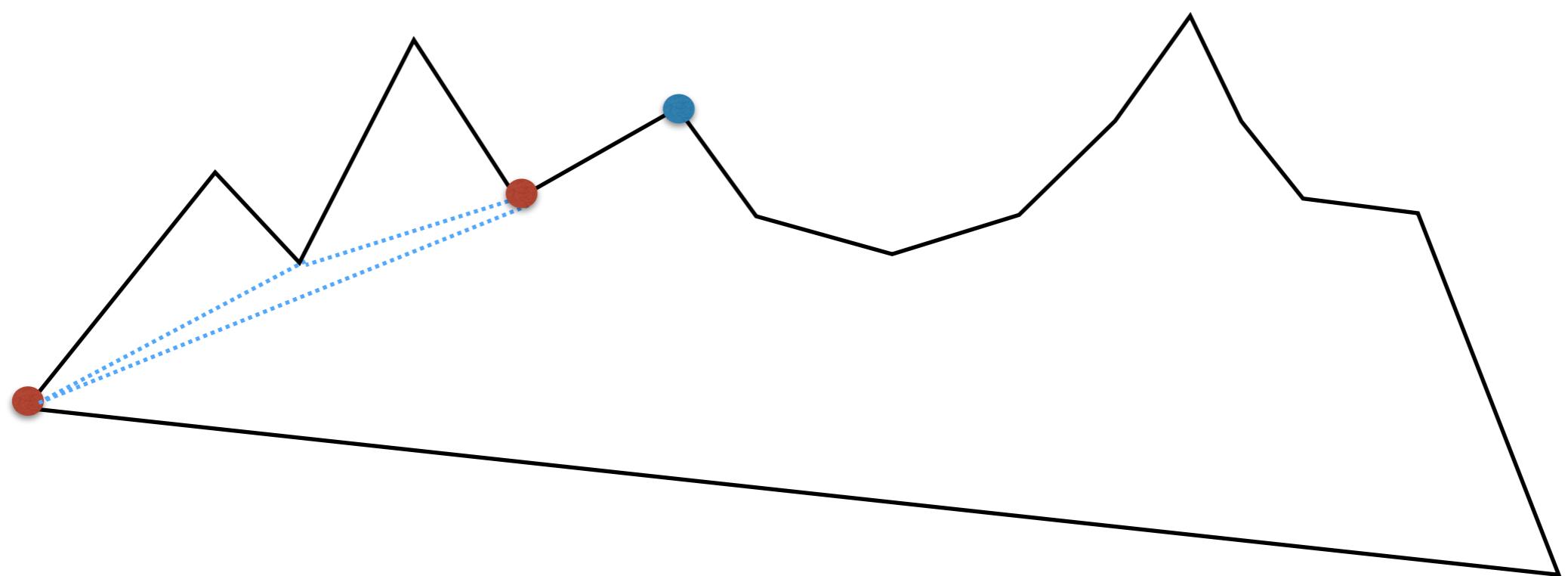
Monotone mountains are easy to triangulate!



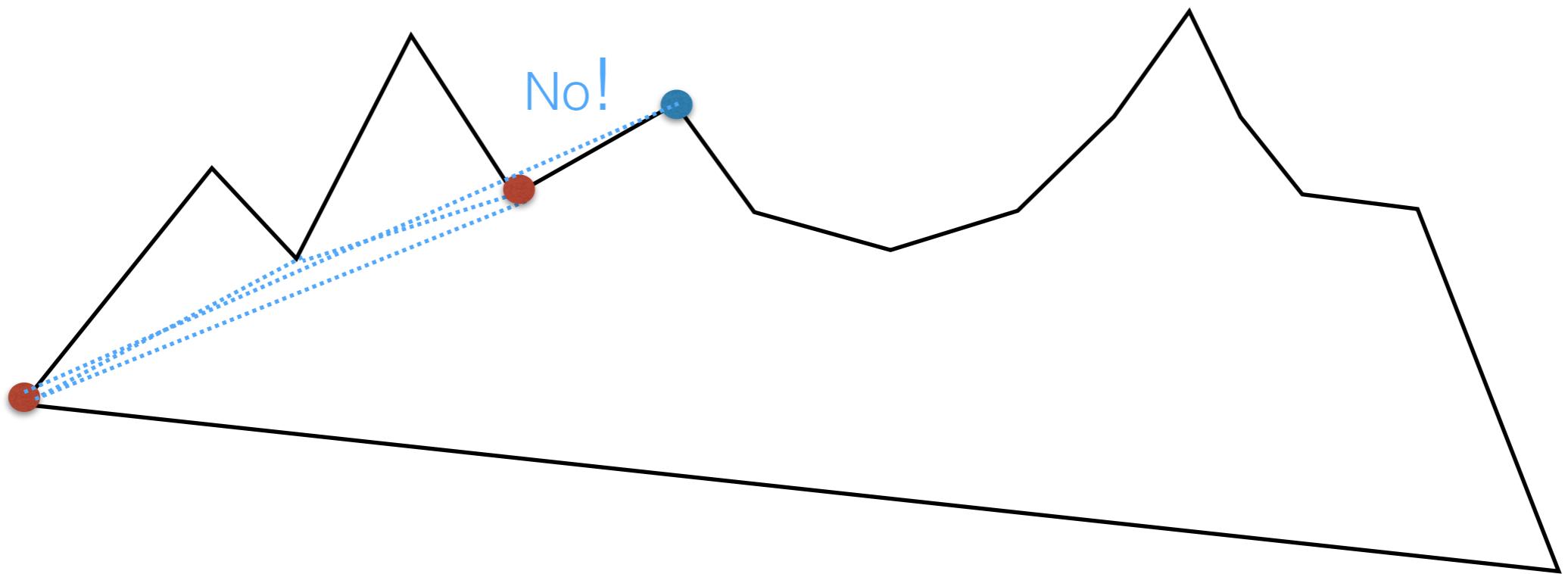
Monotone mountains are easy to triangulate!



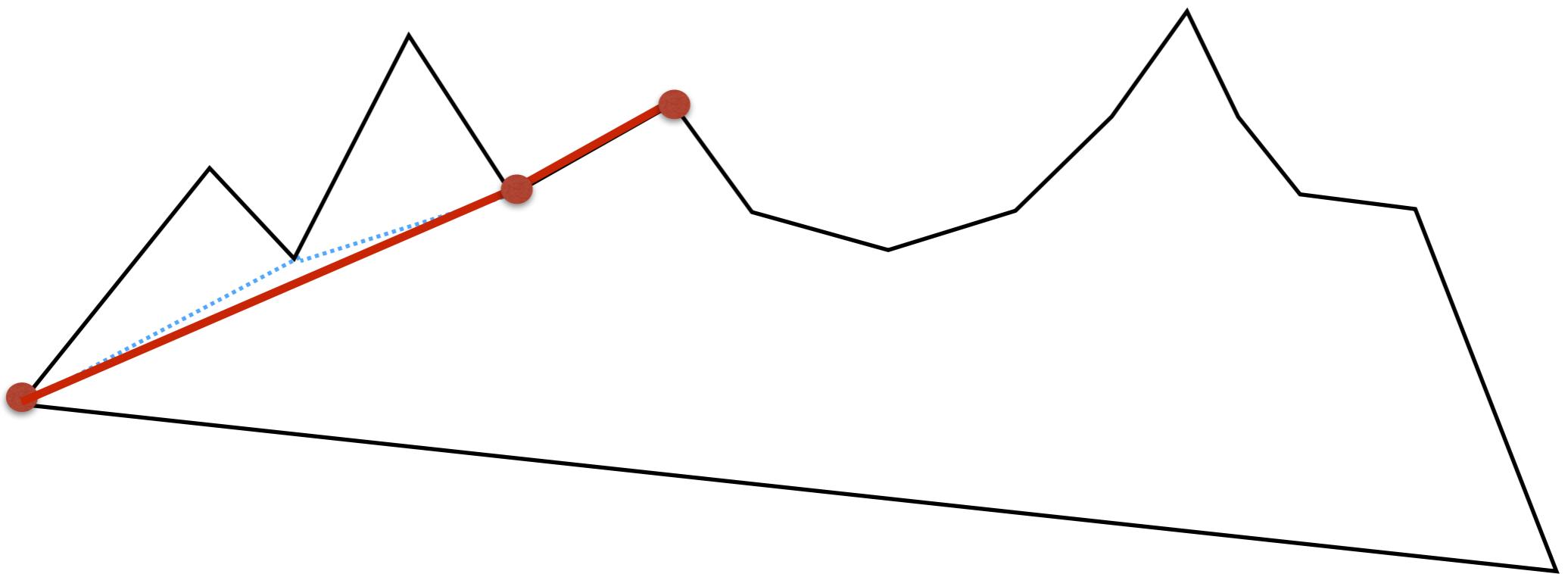
Monotone mountains are easy to triangulate!



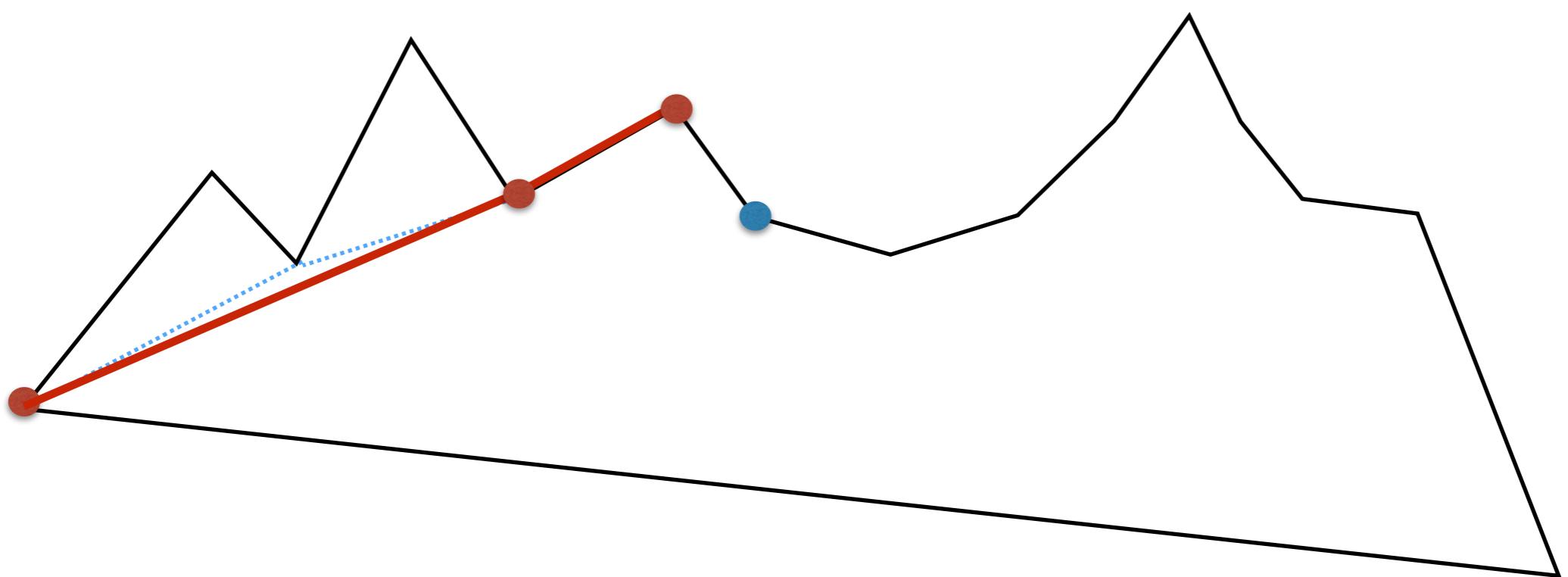
Monotone mountains are easy to triangulate!



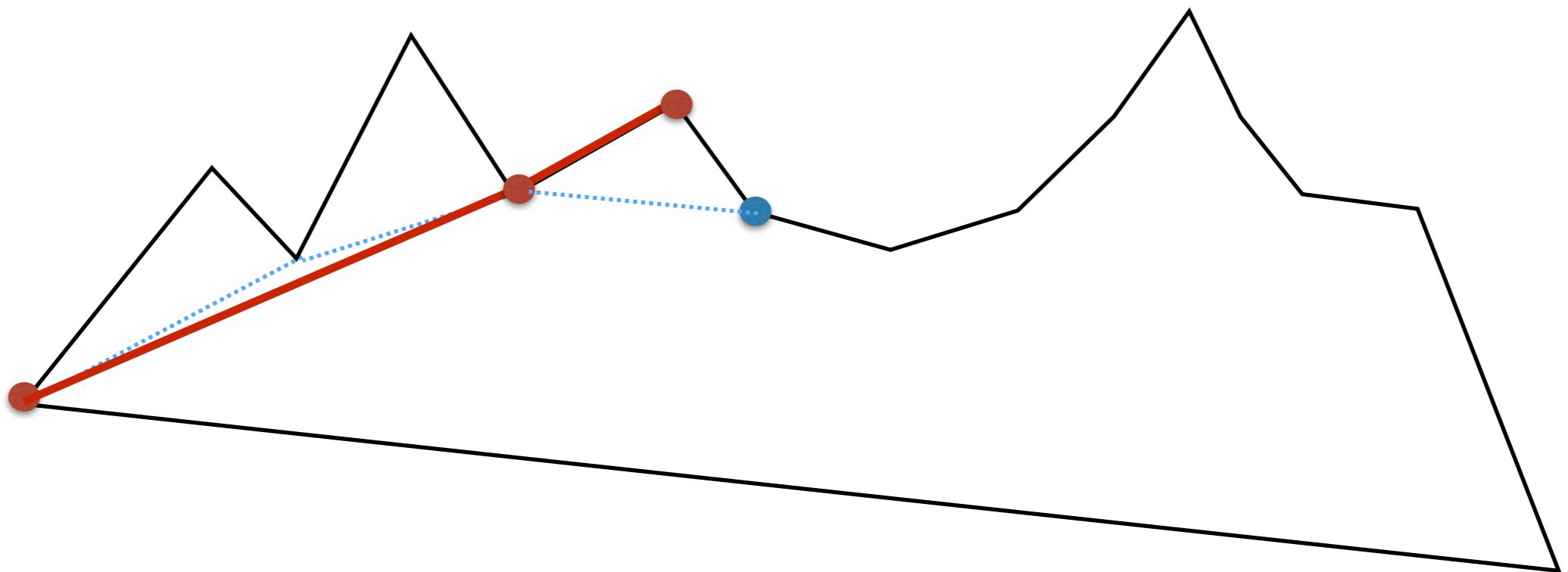
Monotone mountains are easy to triangulate!



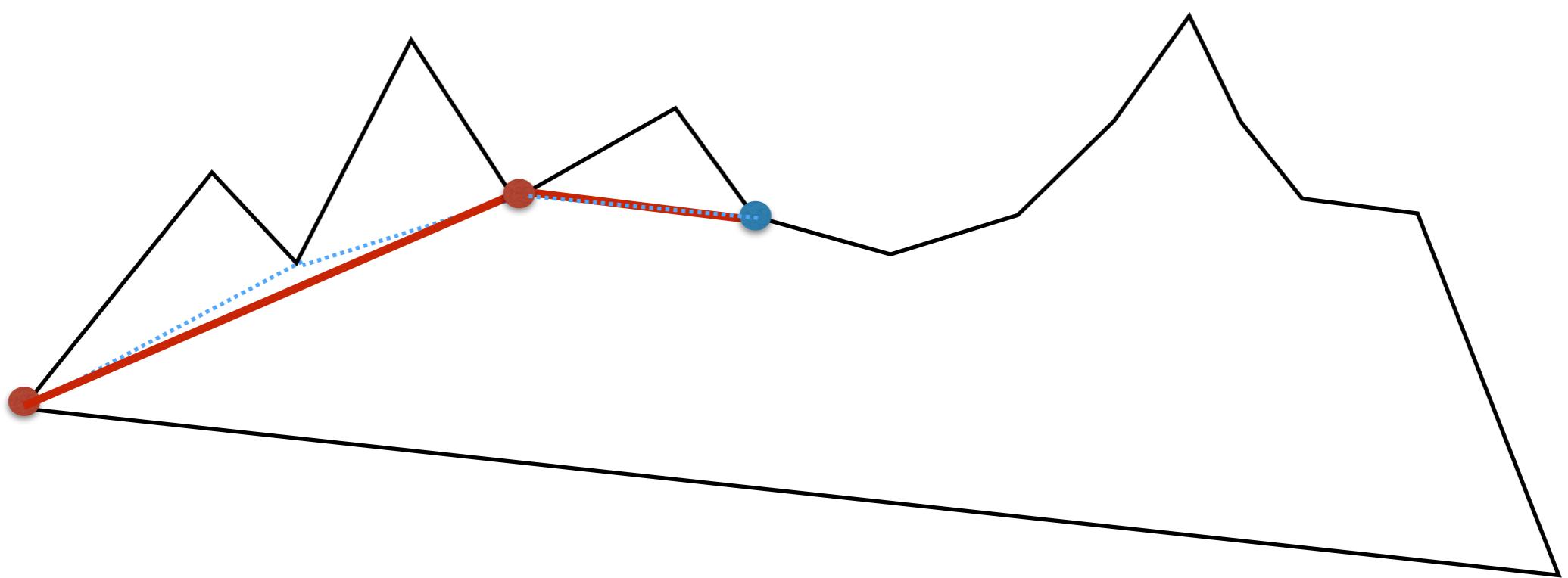
Monotone mountains are easy to triangulate!



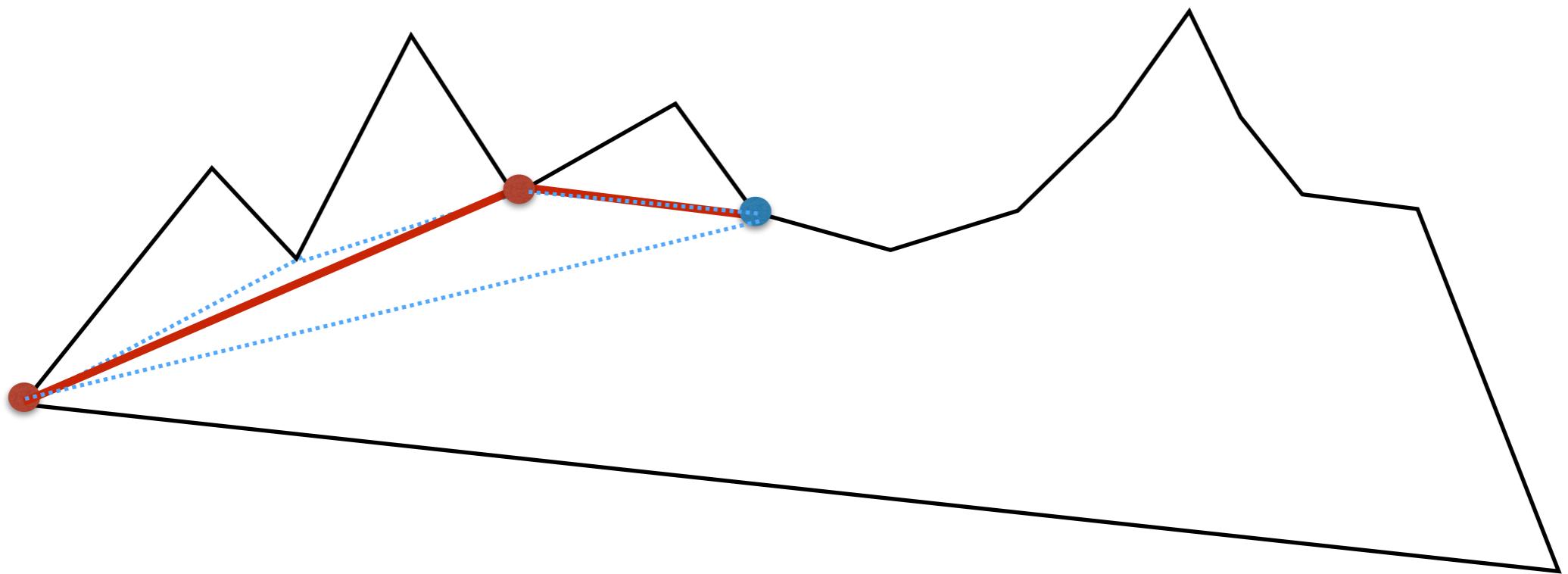
Monotone mountains are easy to triangulate!



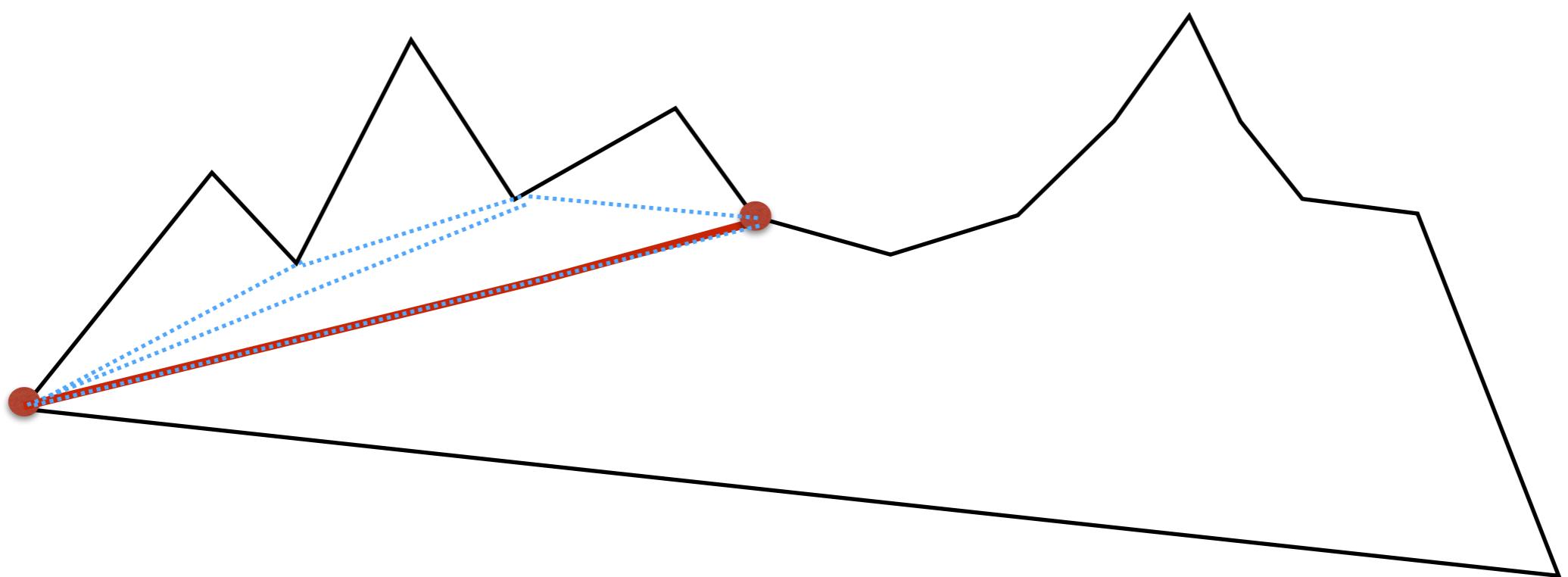
Monotone mountains are easy to triangulate!



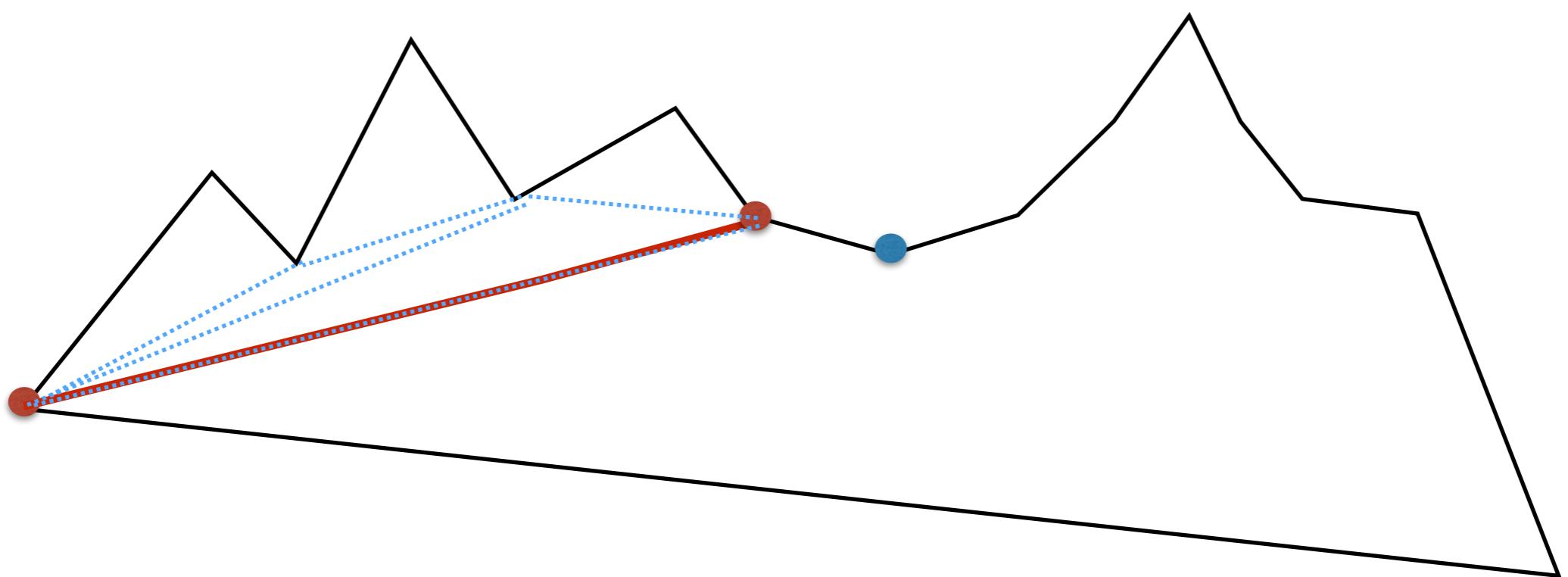
Monotone mountains are easy to triangulate!



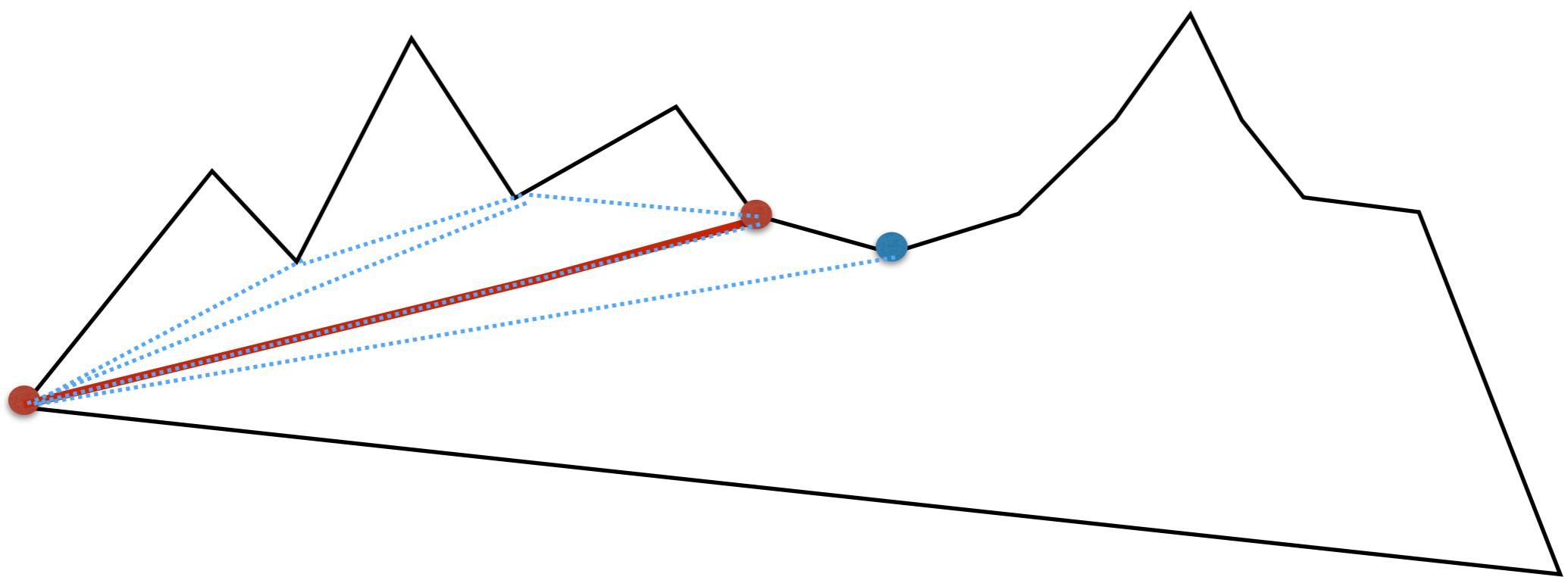
Monotone mountains are easy to triangulate!



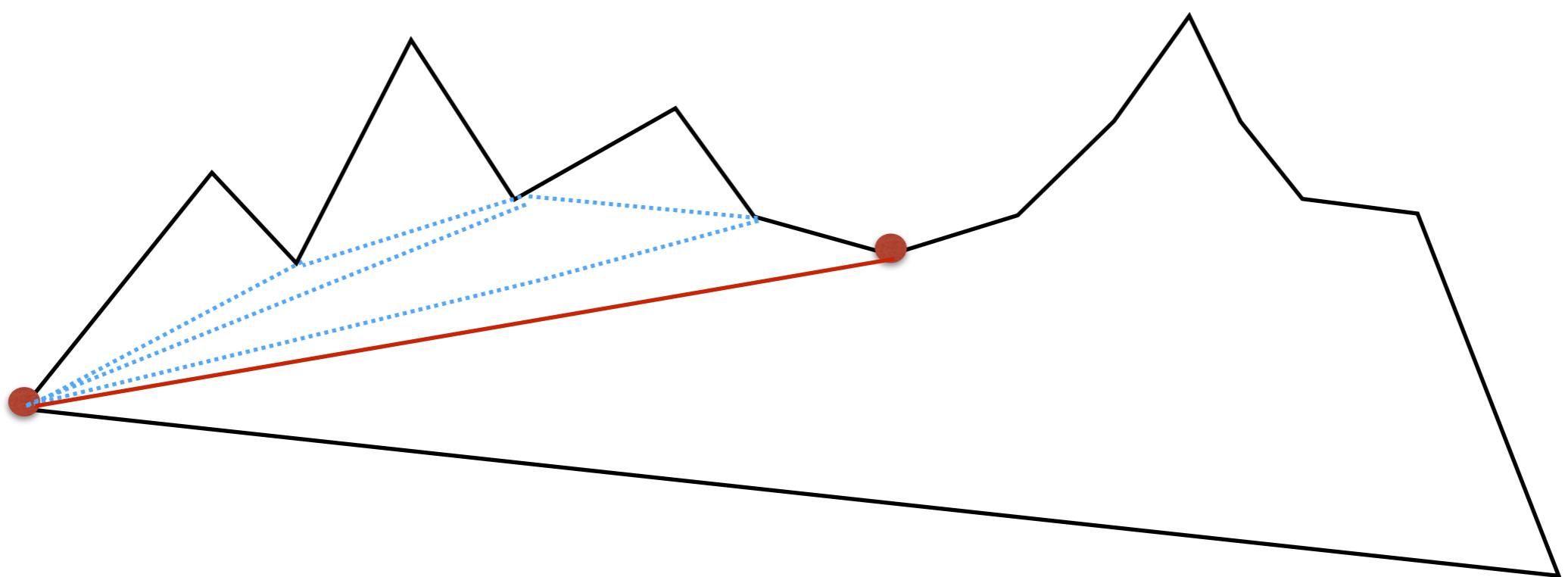
Monotone mountains are easy to triangulate!



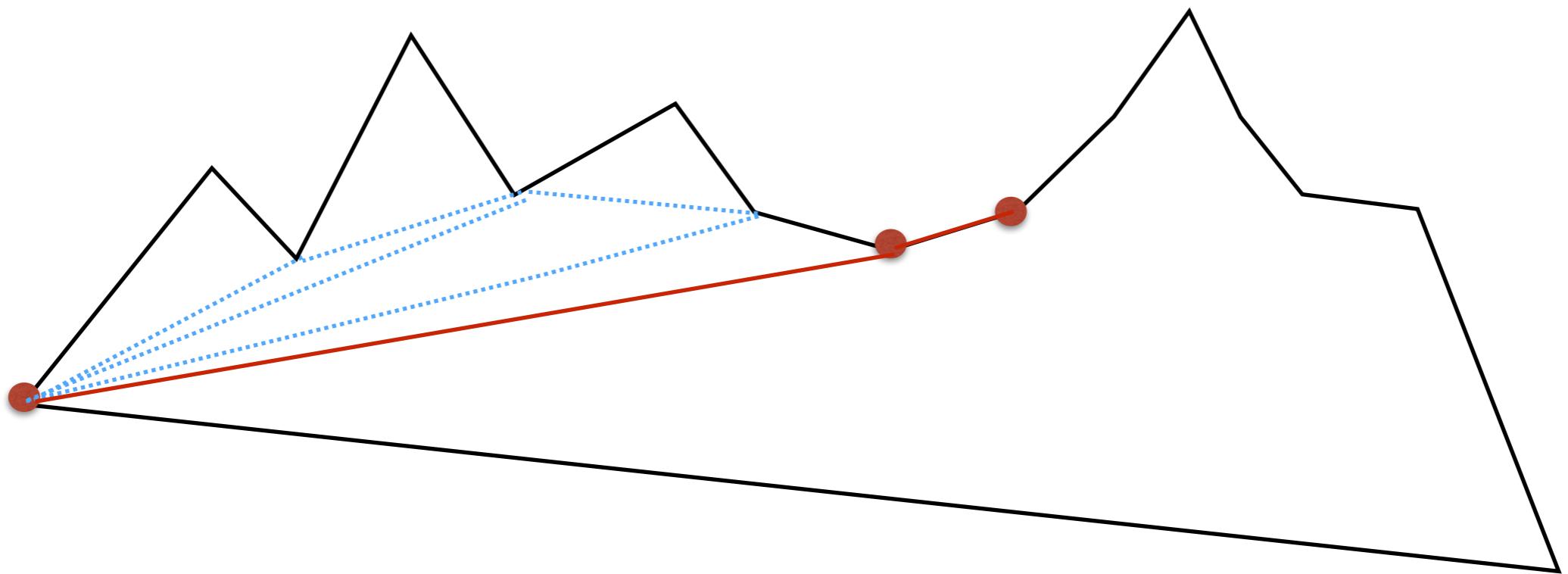
Monotone mountains are easy to triangulate!



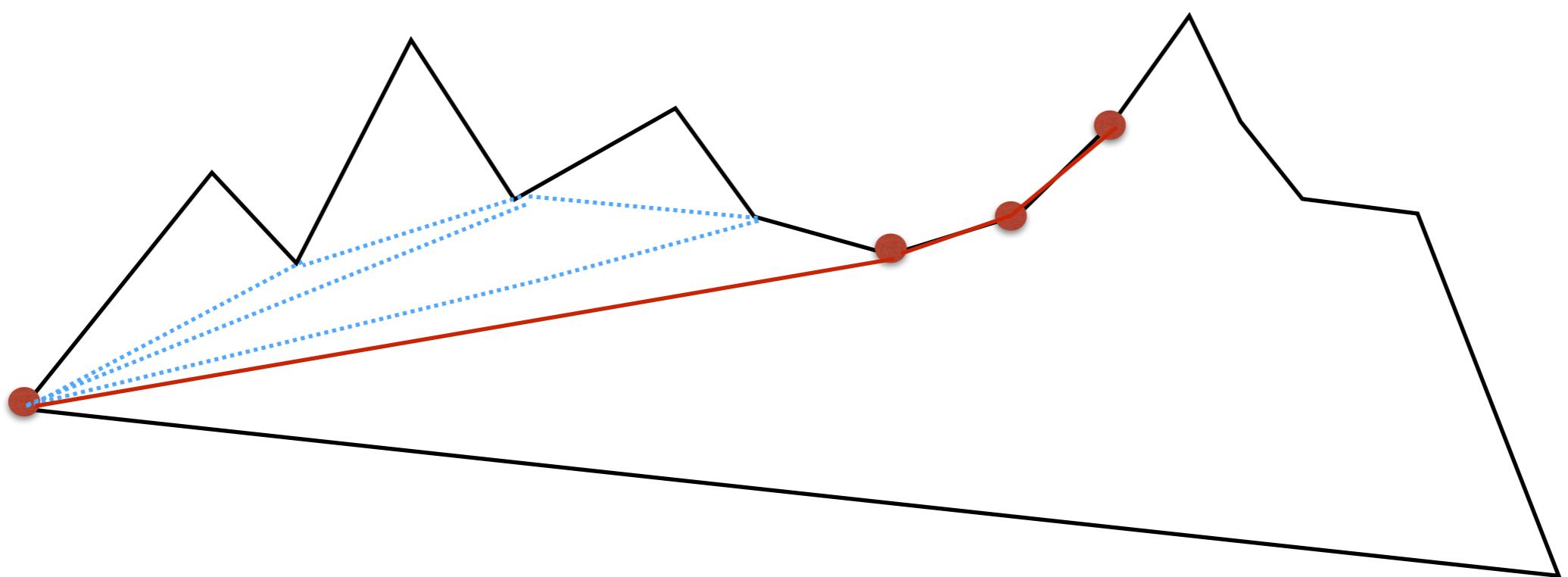
Monotone mountains are easy to triangulate!



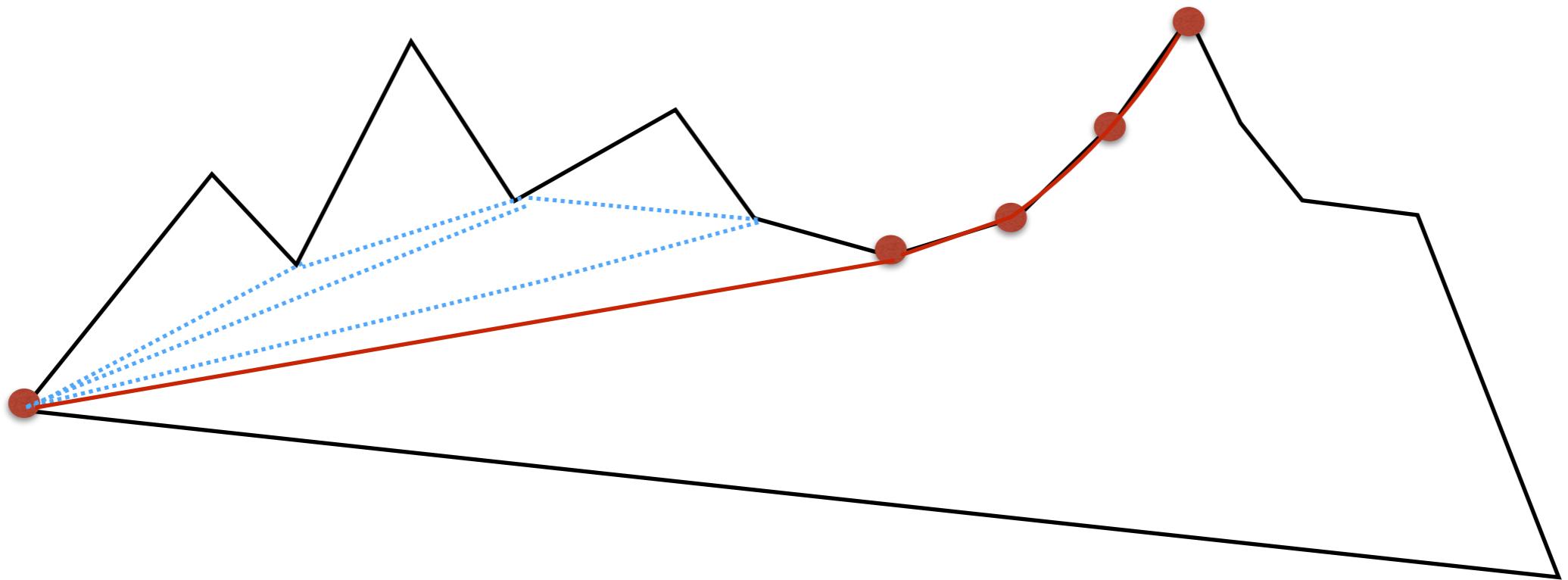
Monotone mountains are easy to triangulate!



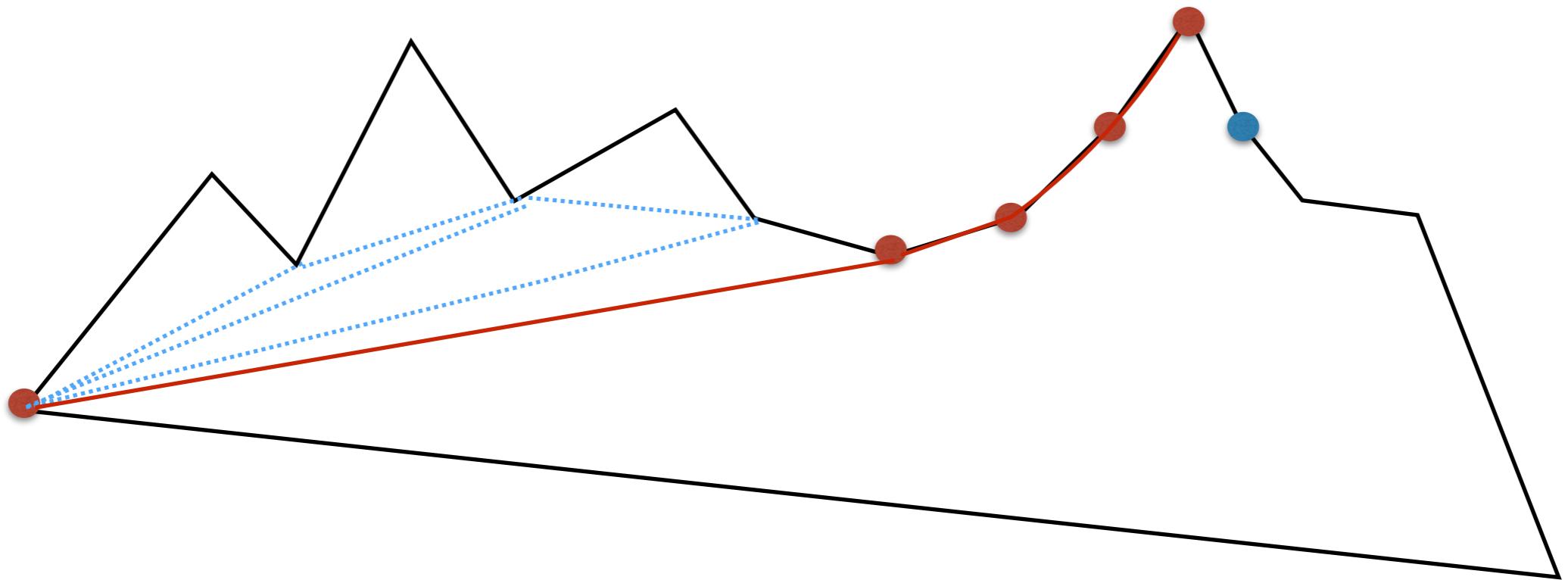
Monotone mountains are easy to triangulate!



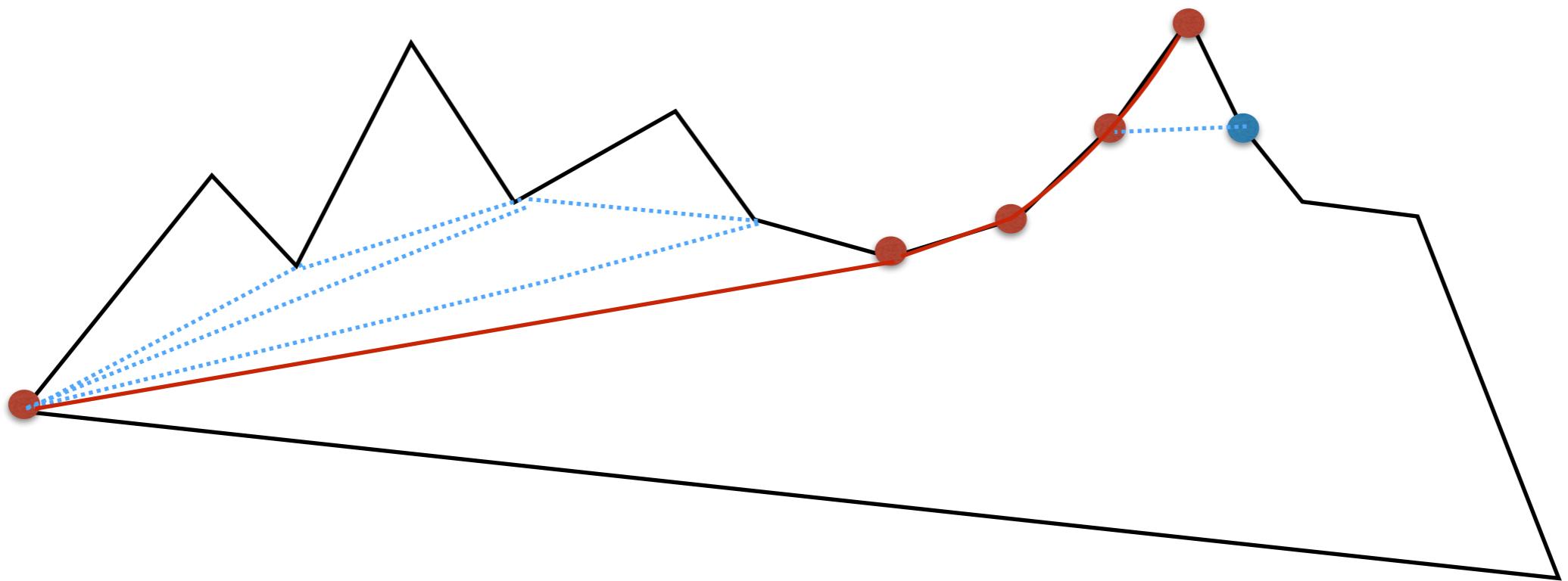
Monotone mountains are easy to triangulate!



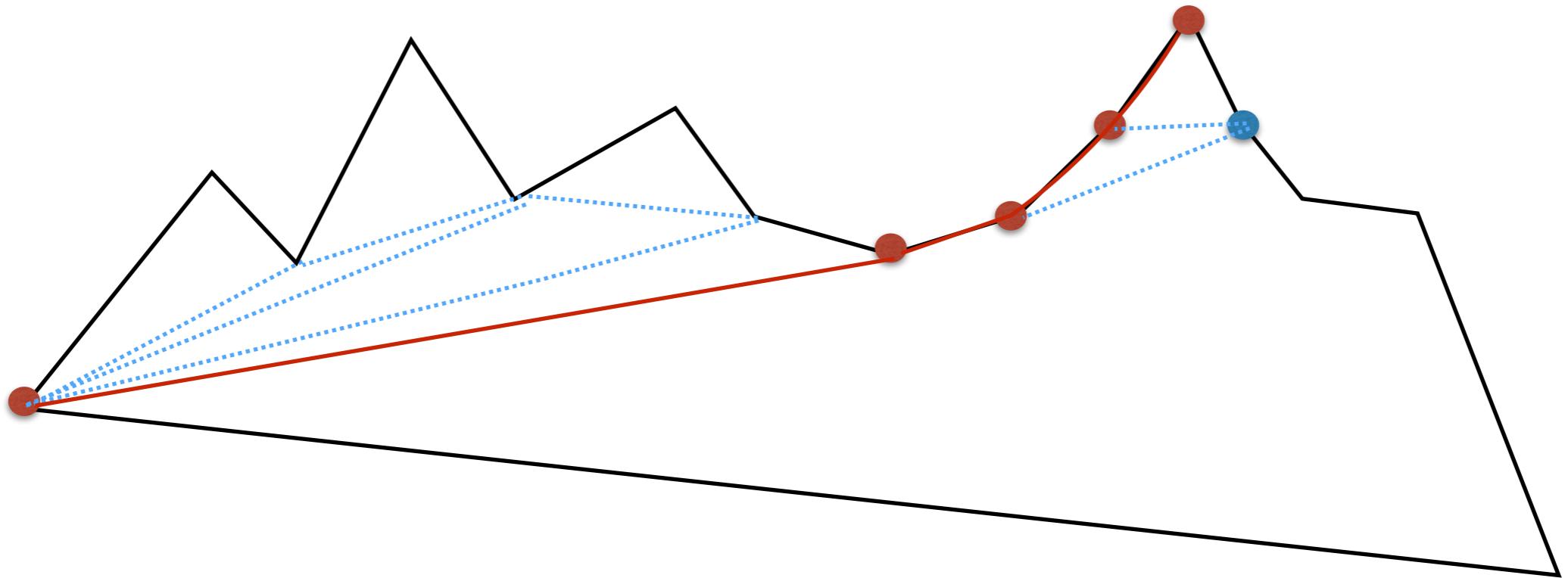
Monotone mountains are easy to triangulate!



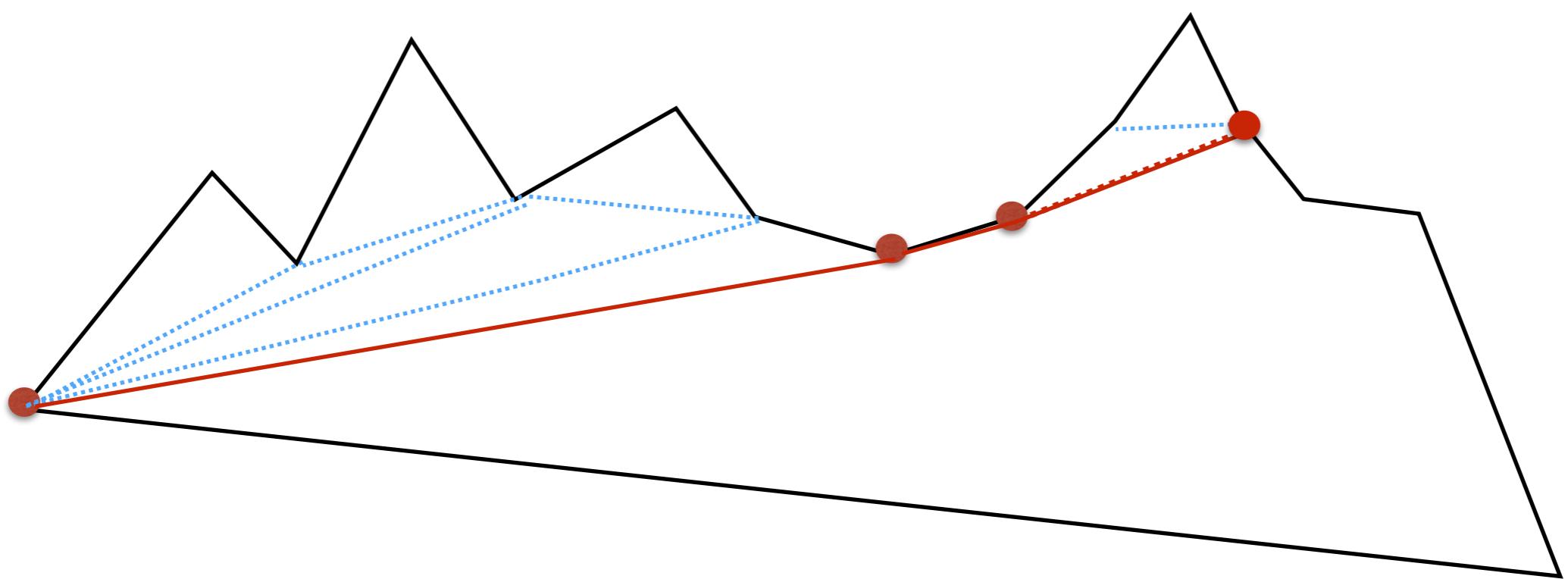
Monotone mountains are easy to triangulate!



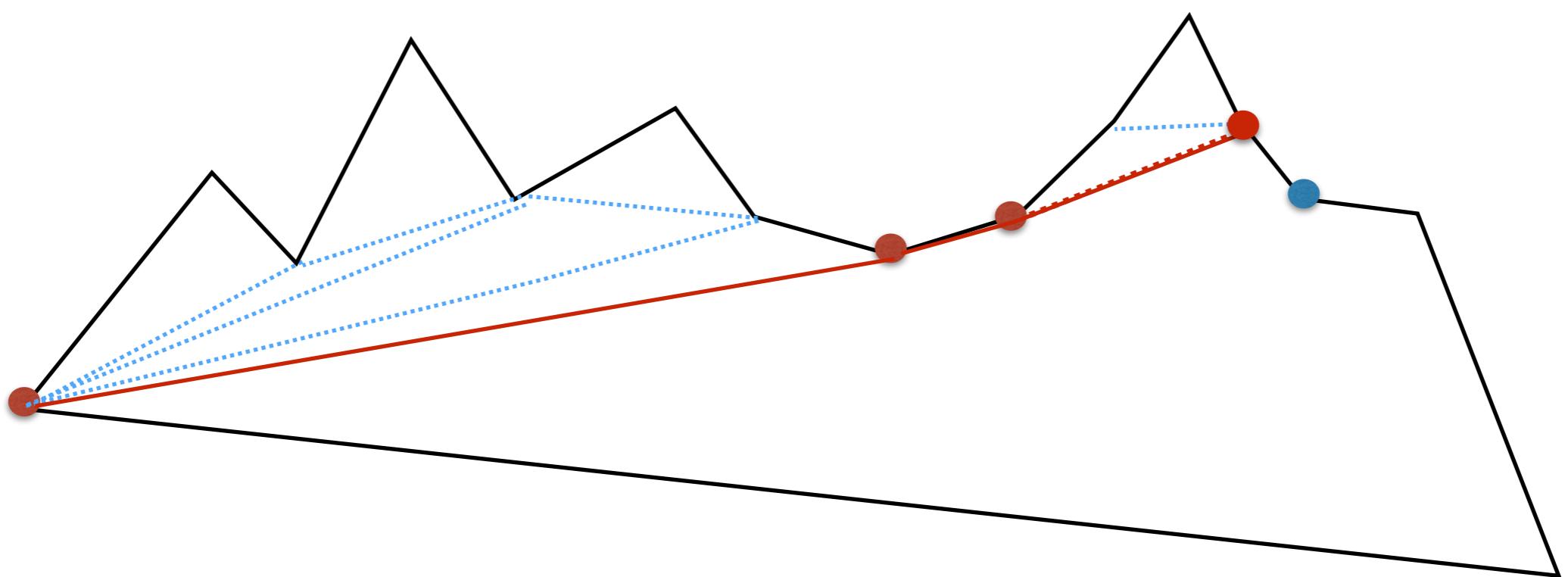
Monotone mountains are easy to triangulate!



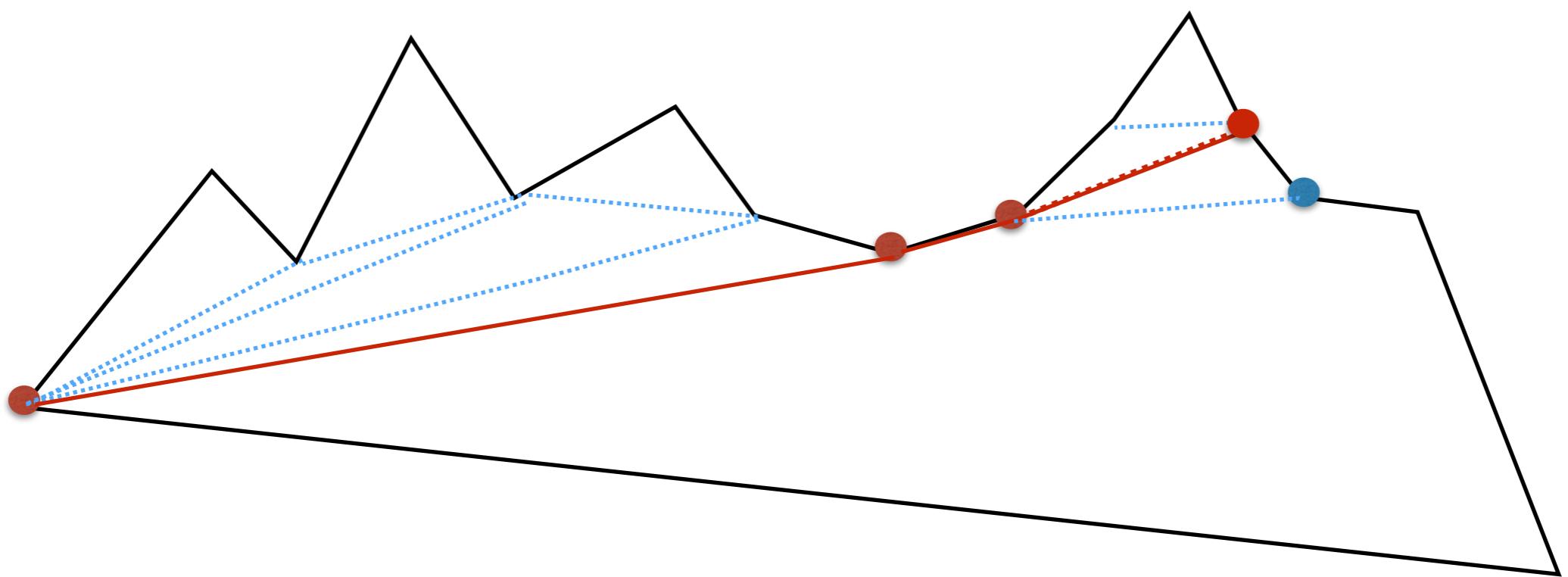
Monotone mountains are easy to triangulate!



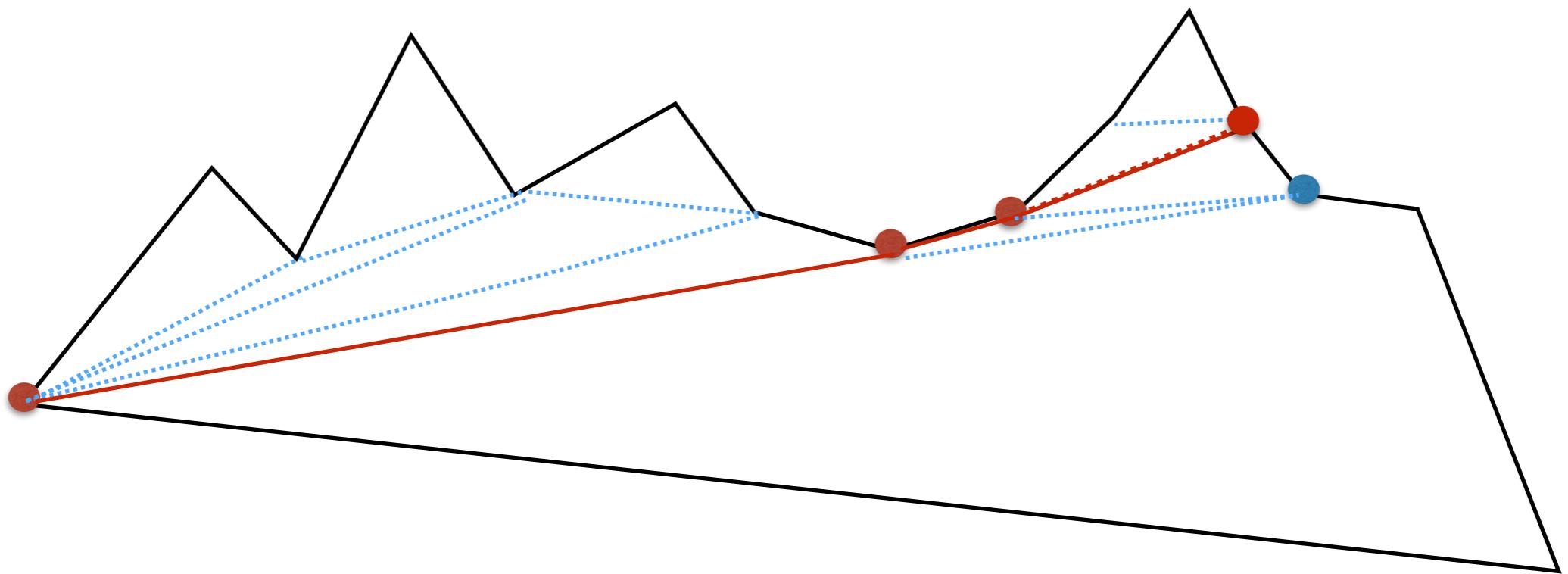
Monotone mountains are easy to triangulate!



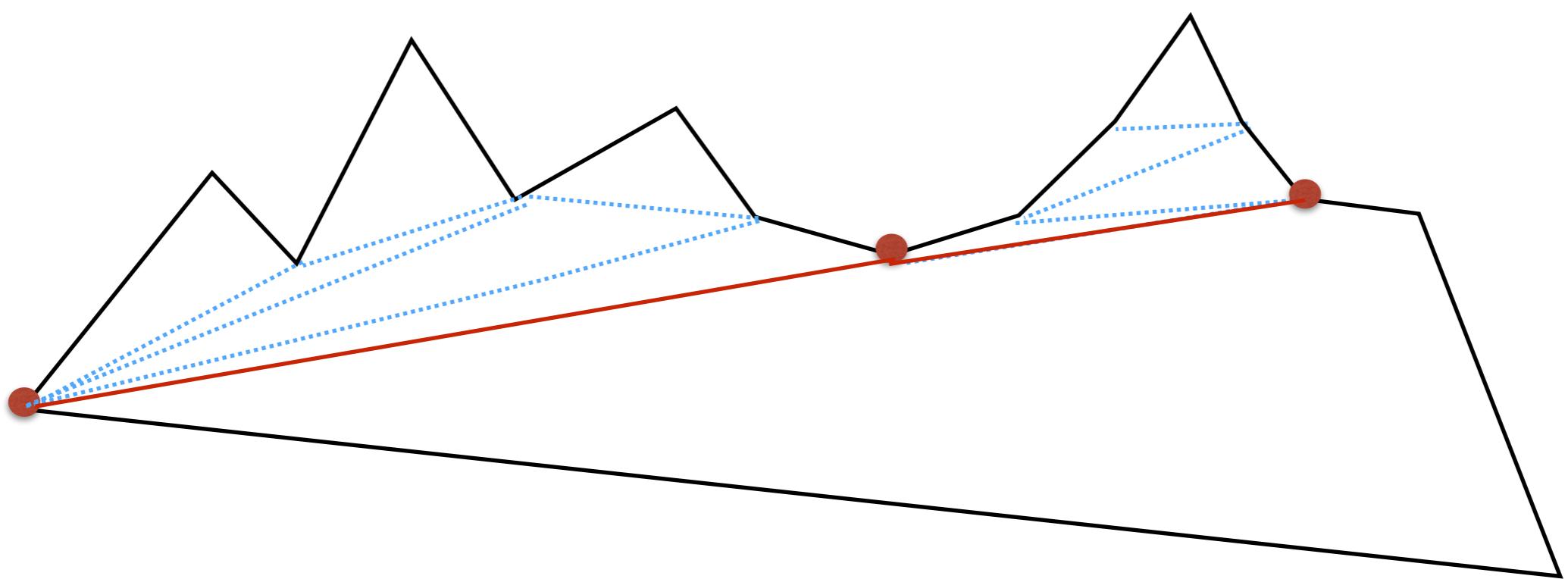
Monotone mountains are easy to triangulate!



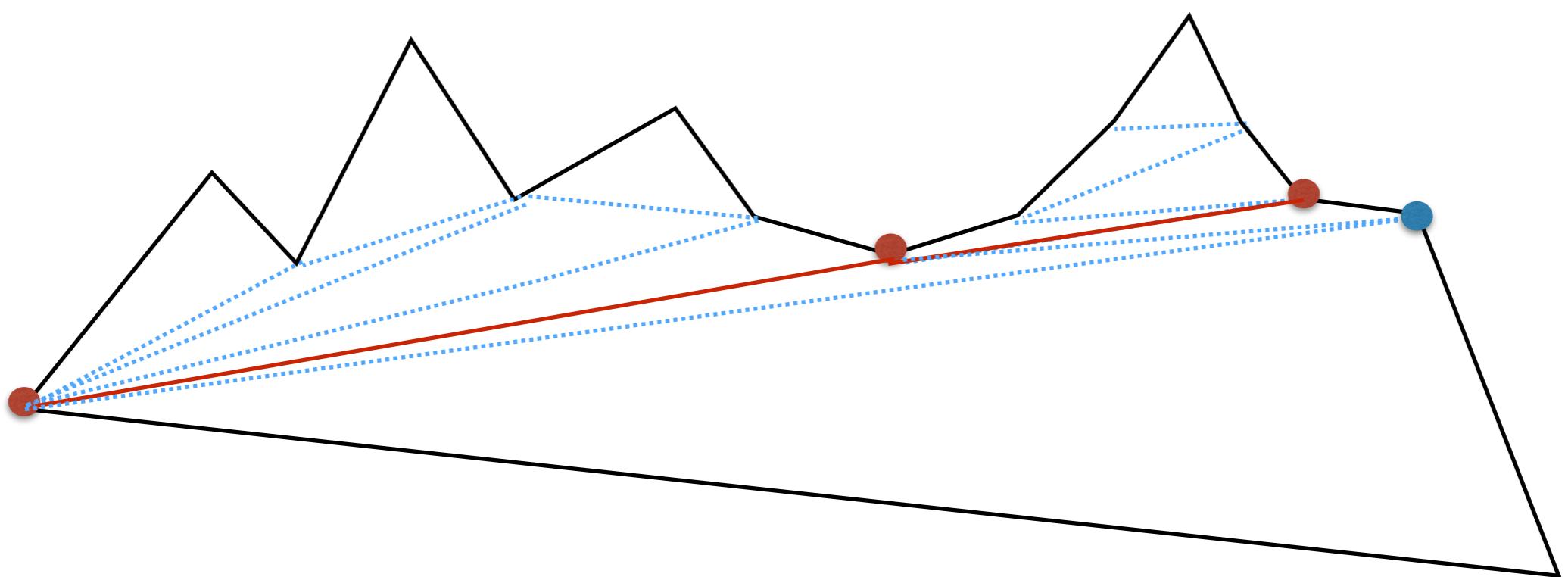
Monotone mountains are easy to triangulate!



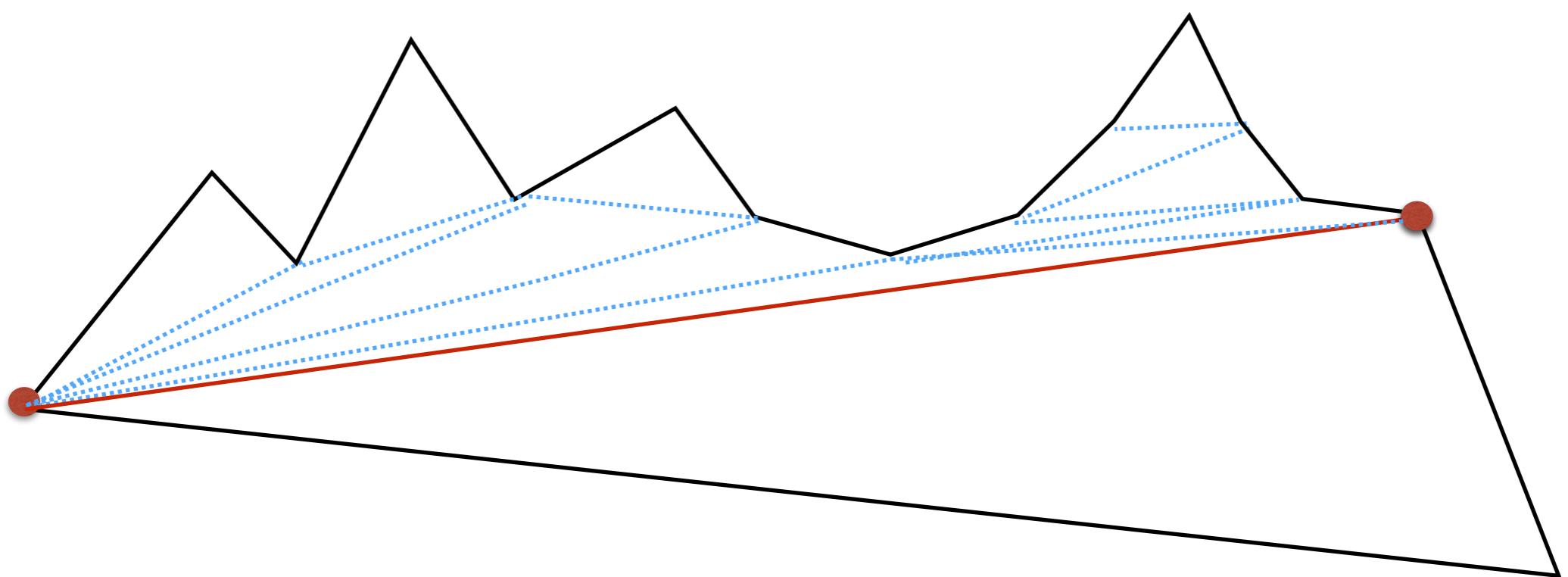
Monotone mountains are easy to triangulate!



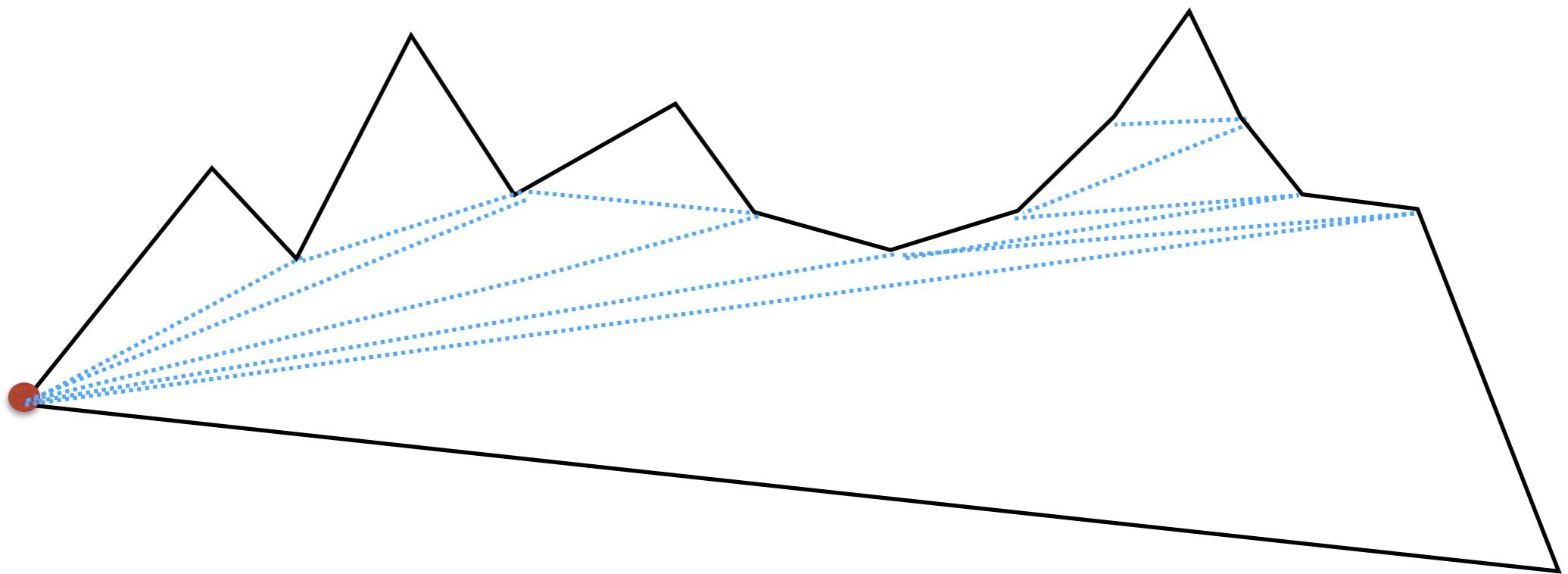
Monotone mountains are easy to triangulate!



Monotone mountains are easy to triangulate!

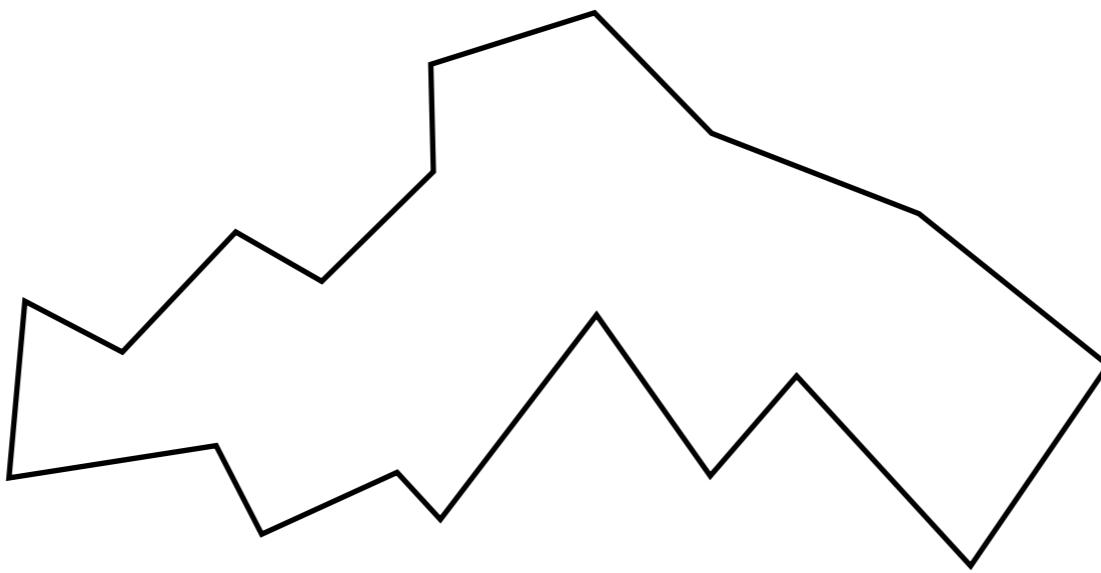


Monotone mountains are easy to triangulate!



Analysis: $O(n)$ time

Monotone polygons

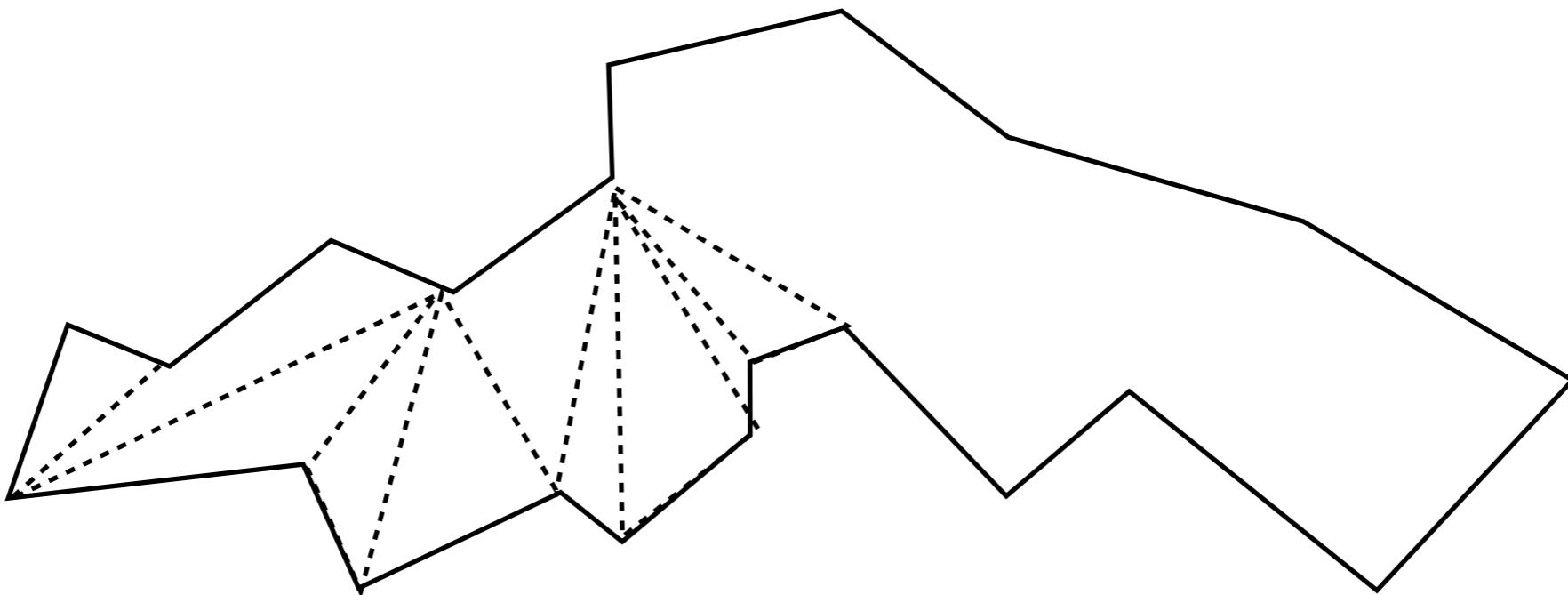


Monotone polygons can (also) be triangulated in $O(n)$ time

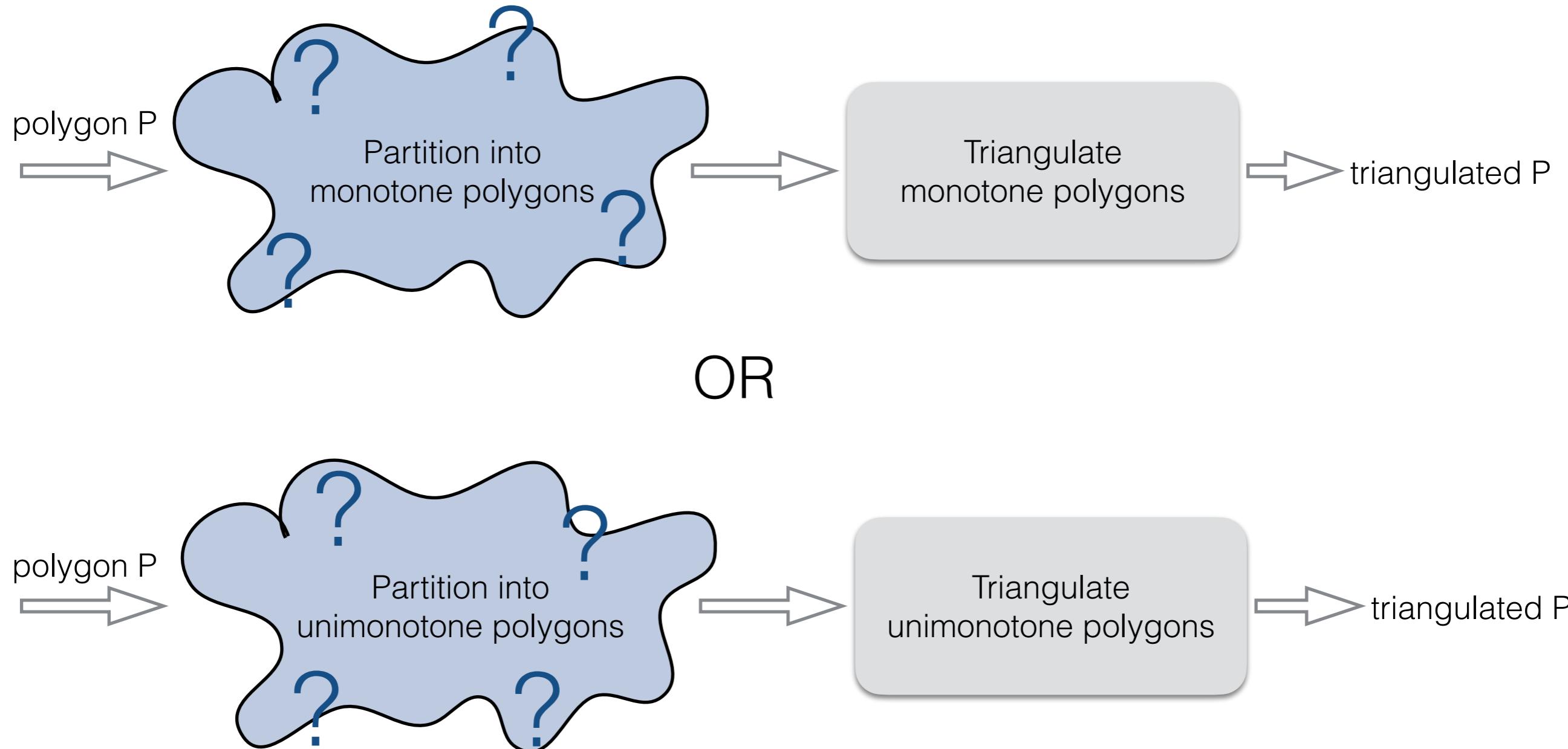
Monotone polygons

Same idea: pick the next vertex in x-order (on upper or lower chain). Connect it to all vertices after it that are visible via a diagonal. Continue to the next vertex.

$O(n)$ time

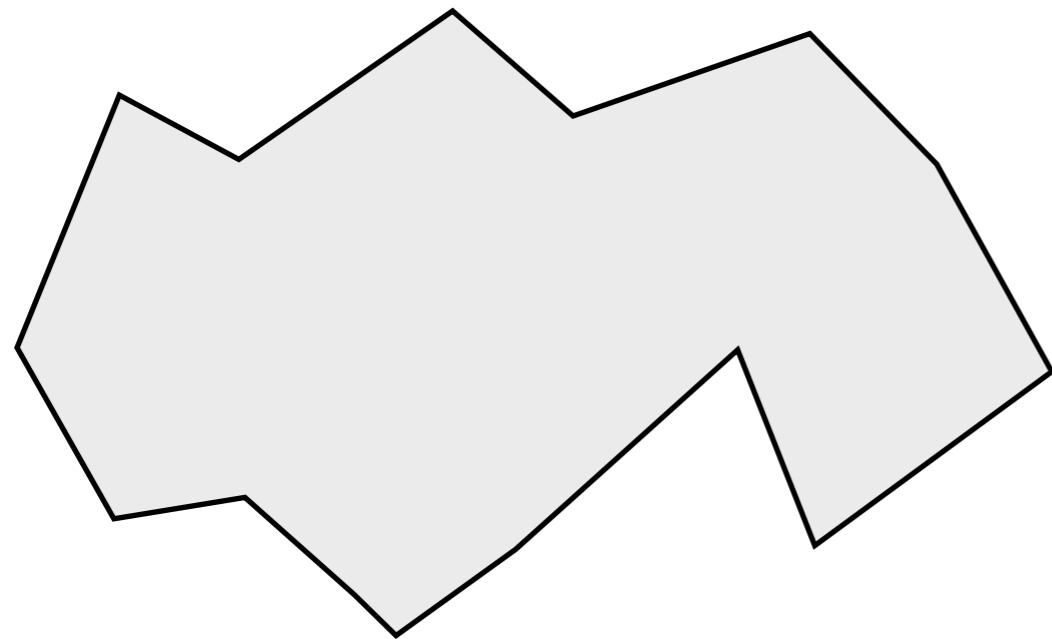


Towards an $O(n \lg n)$ Polygon Triangulation Algorithm

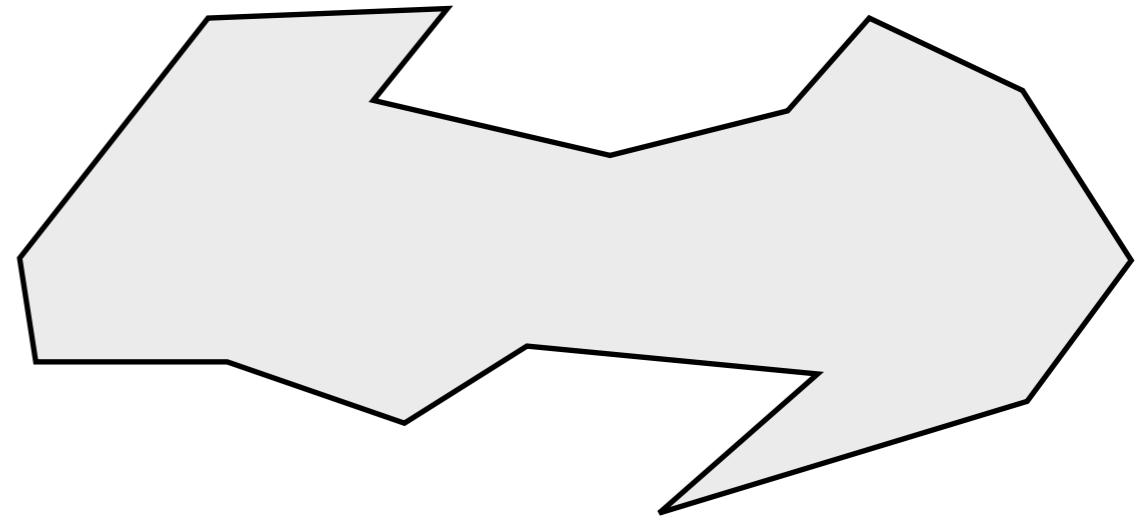


How to partition a polygon into monotone pieces?

Intuition



x-monotone

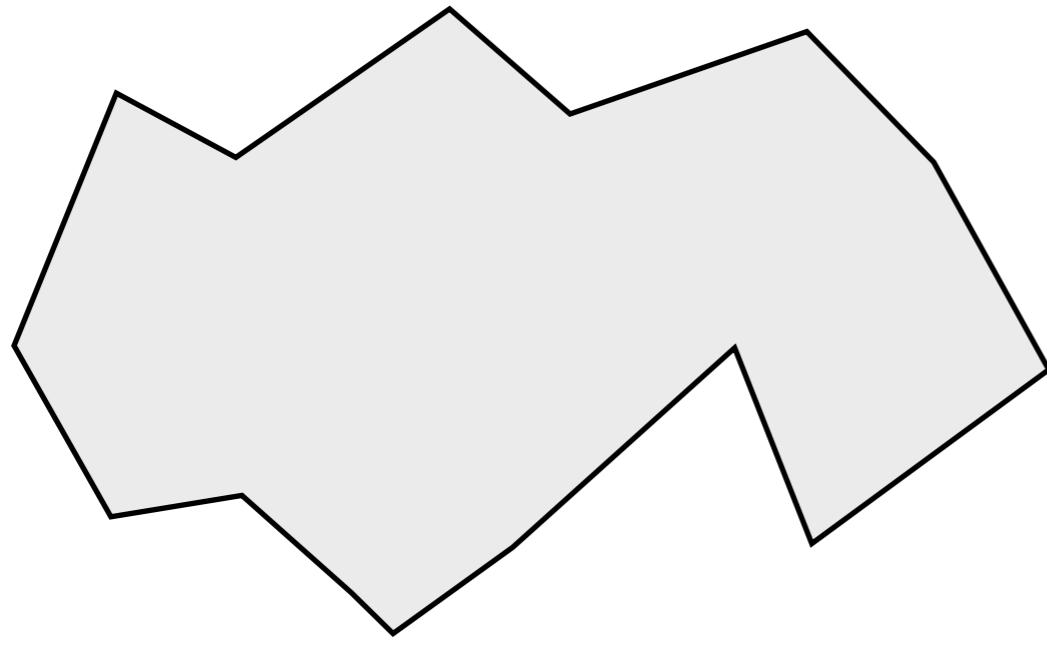


not x-monotone

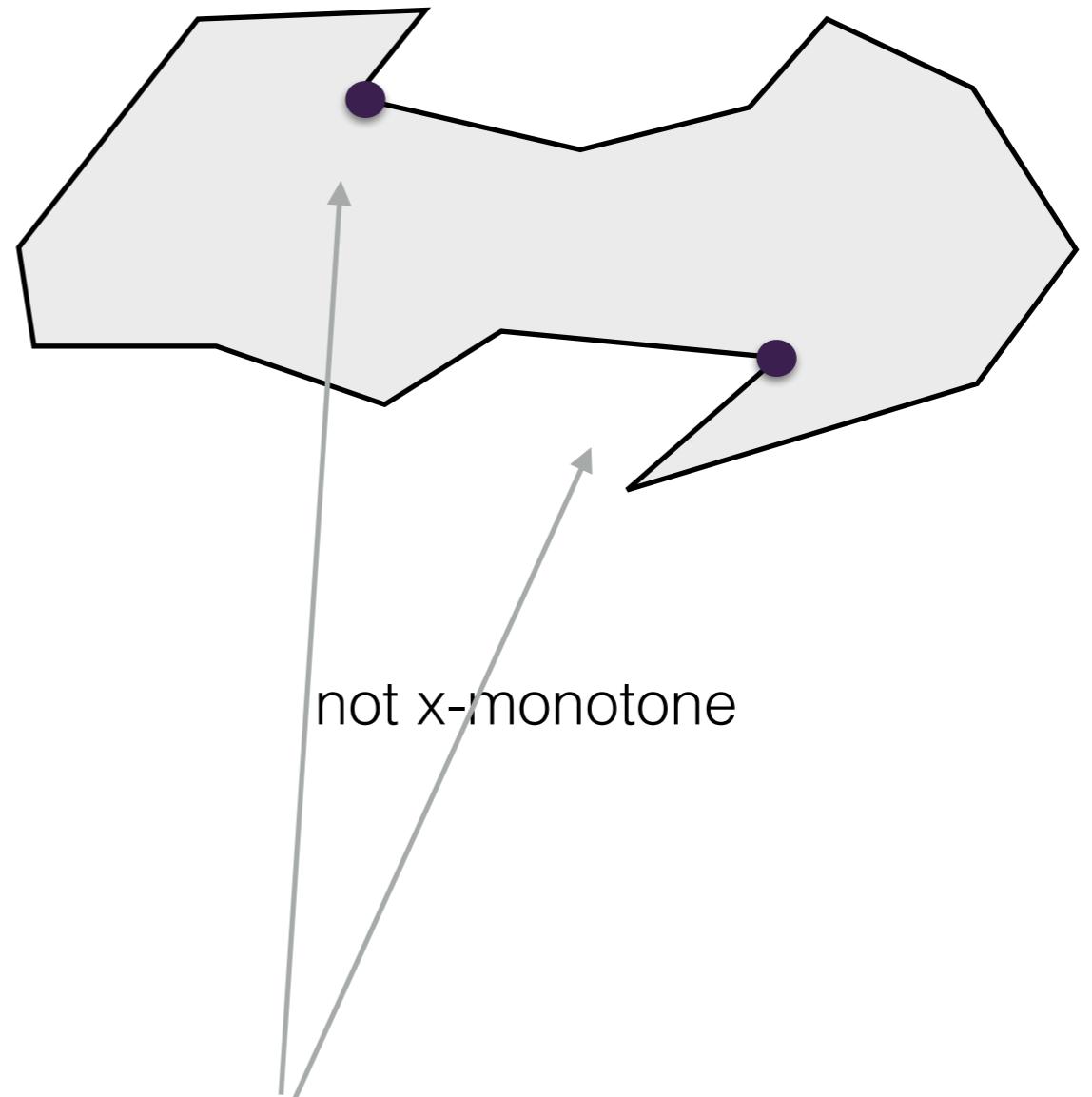
What makes a polygon **not** monotone?

Intuition

Cusp: a reflex vertex v such that the vertices before and after are both smaller or both larger than v (in terms of x-coords).



x-monotone

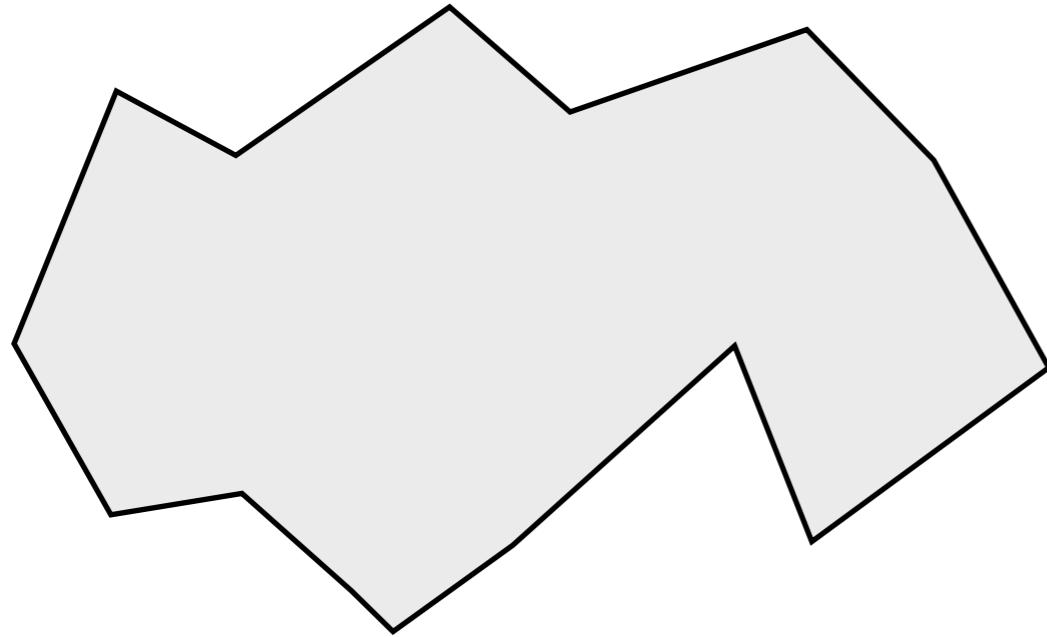


not x-monotone

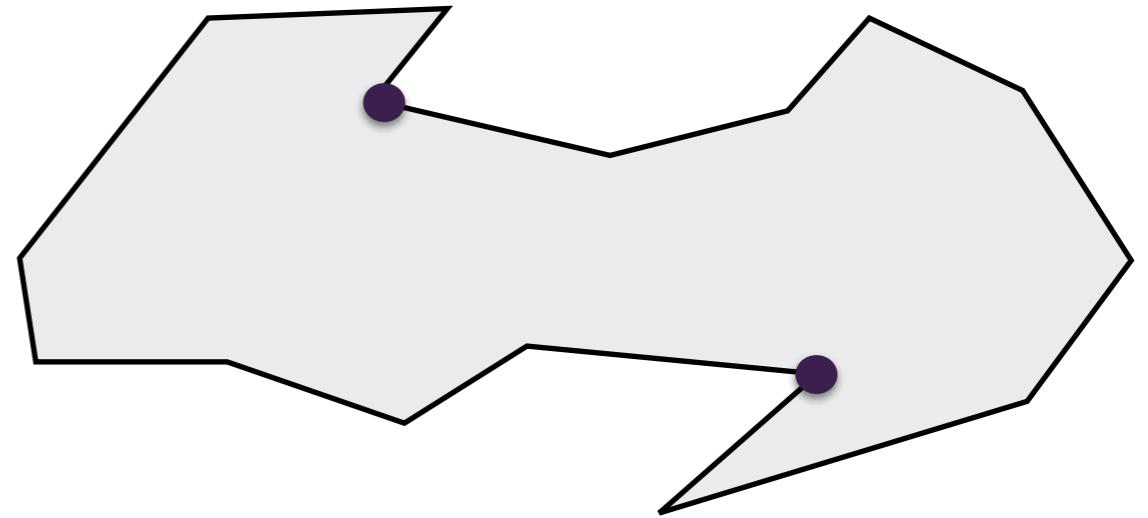
What makes a polygon **not** monotone?

Intuition

Cusp: a reflex vertex v such that the vertices before and after are both smaller or both larger than v (in terms of x-coords).



x-monotone



not x-monotone

- Theorem: If a polygon has no cusps, then it's monotone.
- (Intuitively clear, but proof is tedious)

We'll partition a polygon into monotone polygons by
getting rid of cusps

We'll get rid of cusps using a trapezoidation of P.

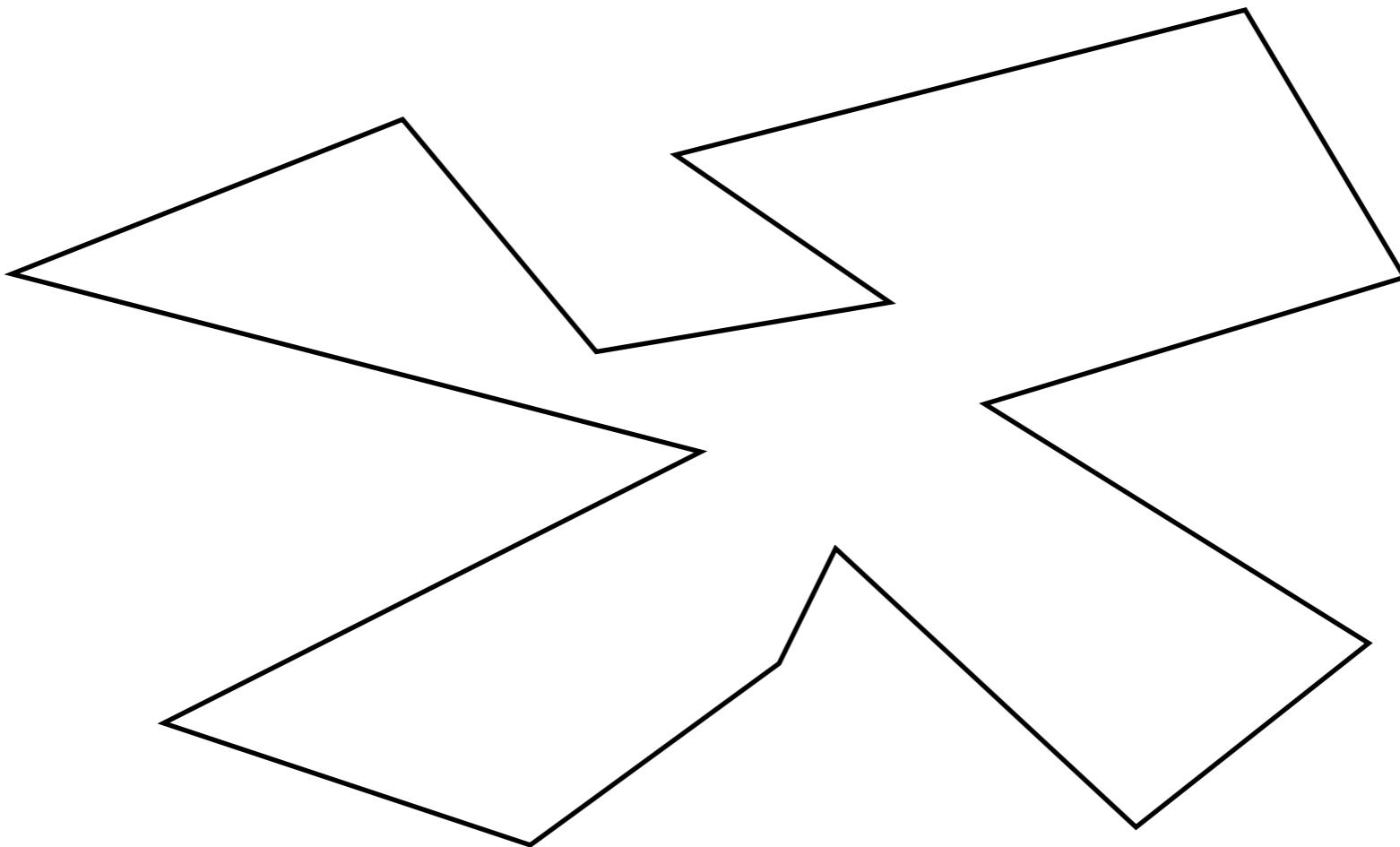


What?

Trapezoid partitions

Shoot vertical rays

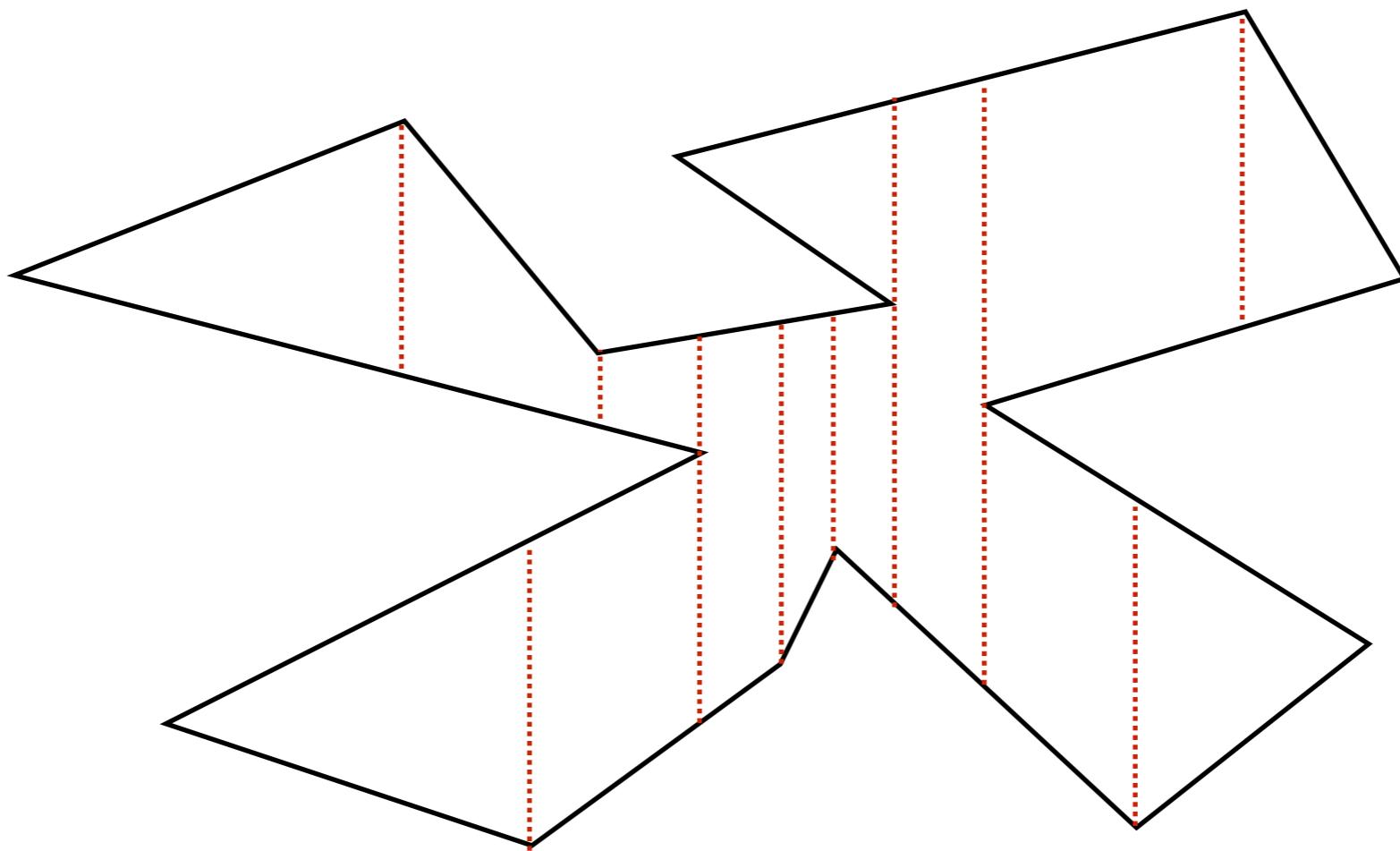
- If polygon is above vertex, shoot vertical ray up until reaches boundary
- If f polygon is below vertex, shoot down
- If polygon is above and below vertex, shoot both up and down



Trapezoid partitions

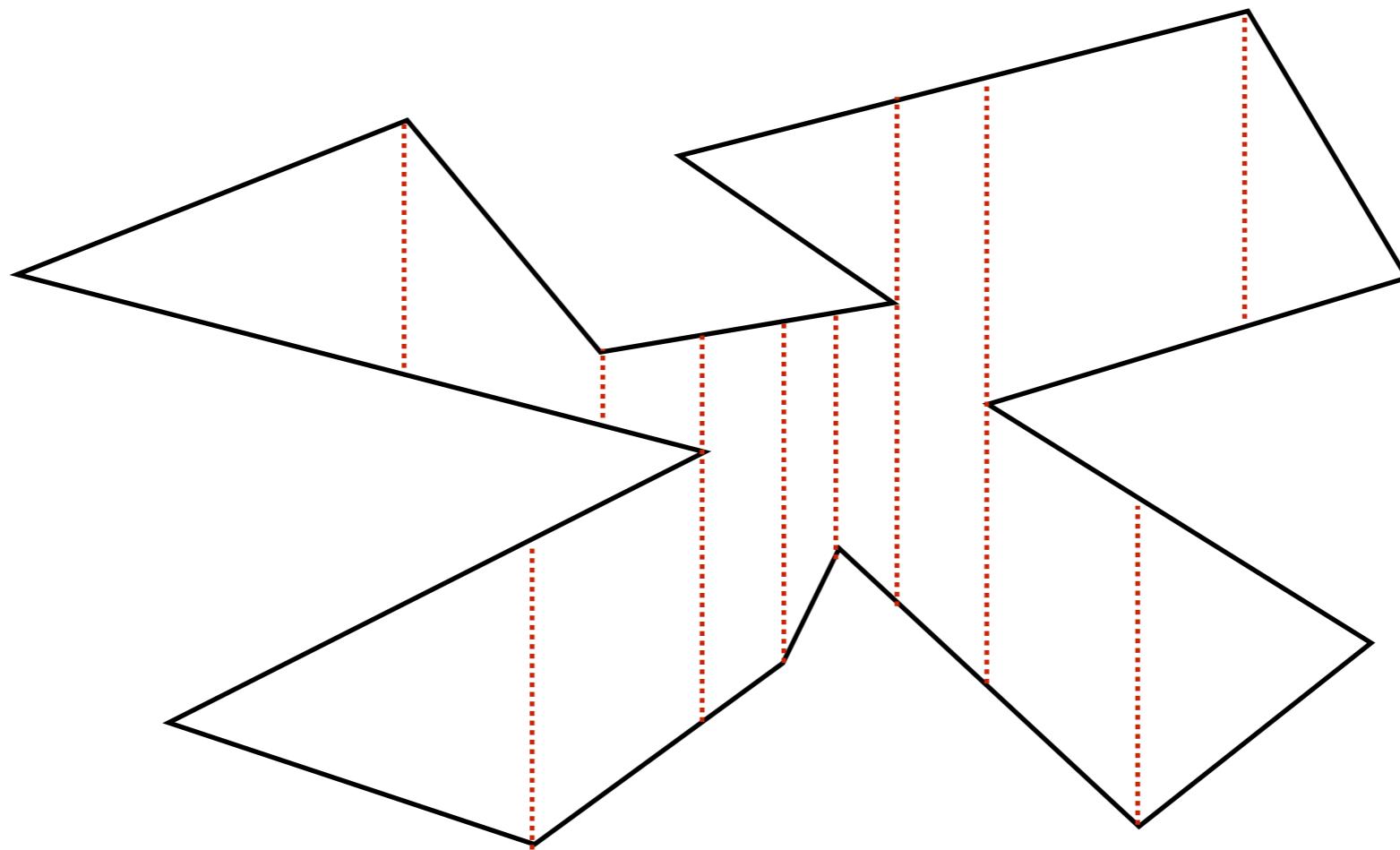
Shoot vertical rays

- If polygon is above vertex, shoot vertical ray up (until reaches boundary)
- If f polygon is below vertex, shoot down
- If polygon is above and below vertex, shoot both up and down



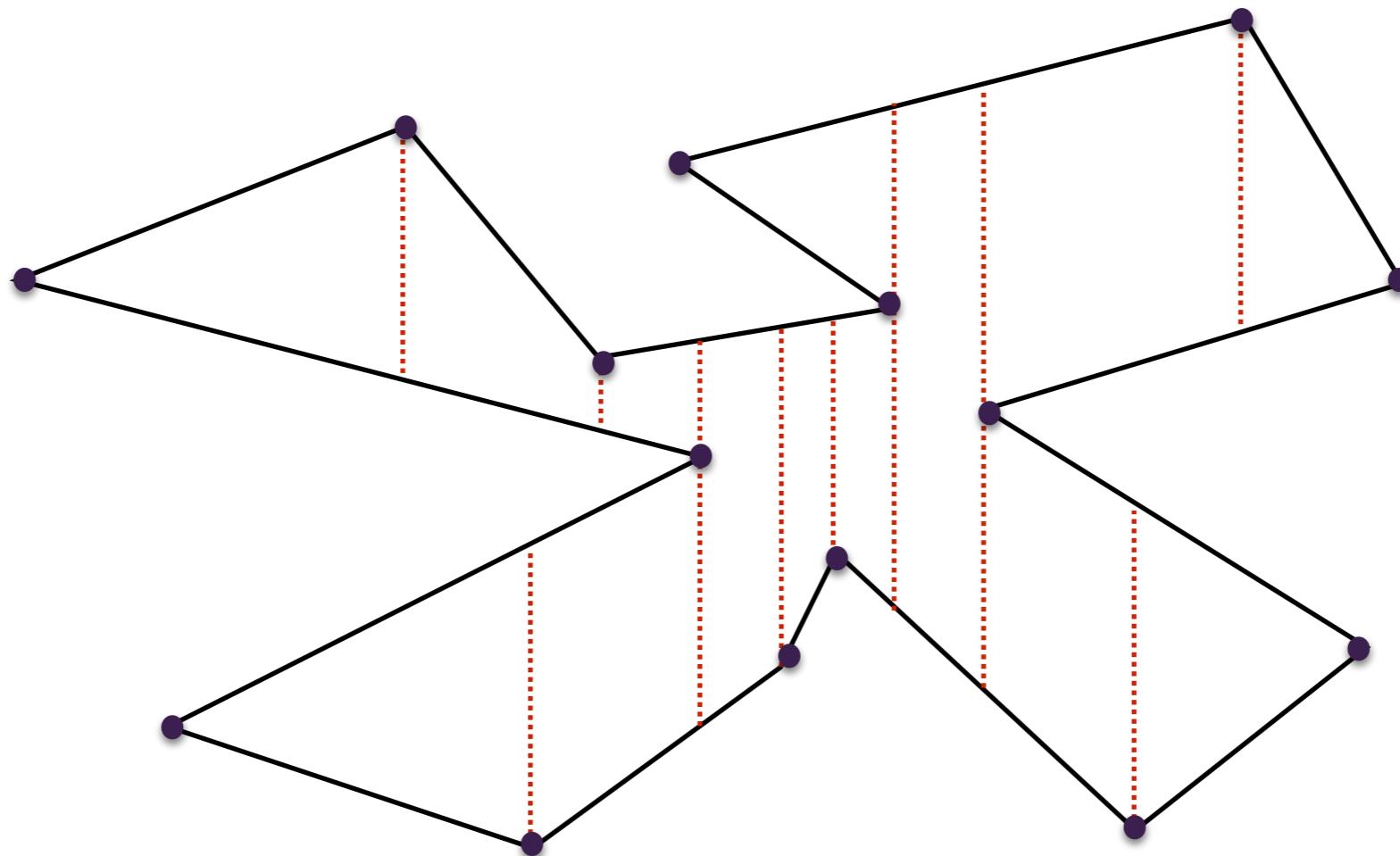
Trapezoid partitions: Properties

- Each polygon in the partition is a trapezoid, because:
 - It has one or two rays as sides.
 - If it has two, they must both hit the same edge above, and the same edge below.
- How many ?
 - At most one ray through each vertex => $O(n)$ threads => $O(n)$ trapezoids



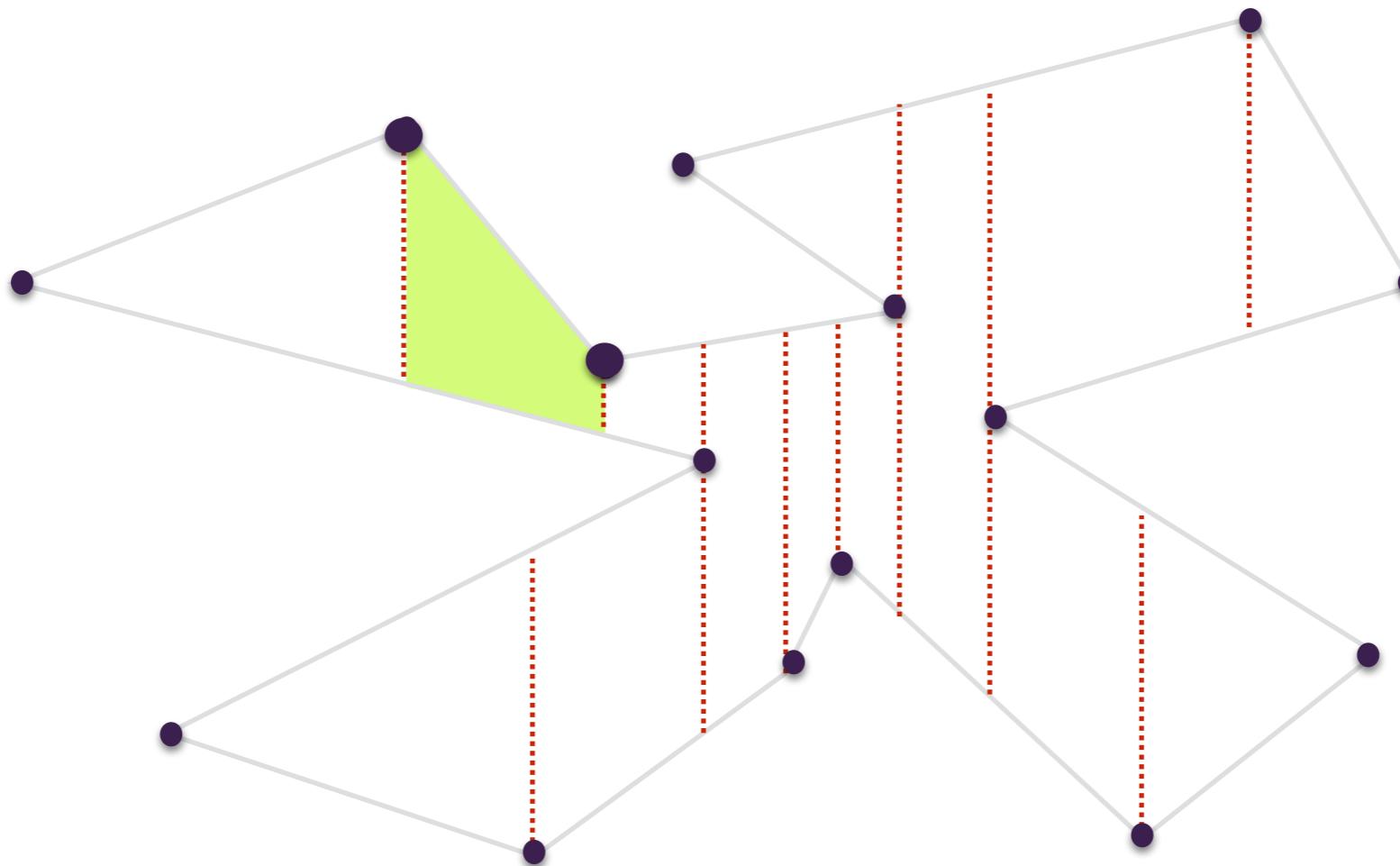
Trapezoid partitions

- Each trapezoid has precisely two vertices of the polygon, one on the left and one on the right. They can be on the top, bottom or middle of the trapezoid.



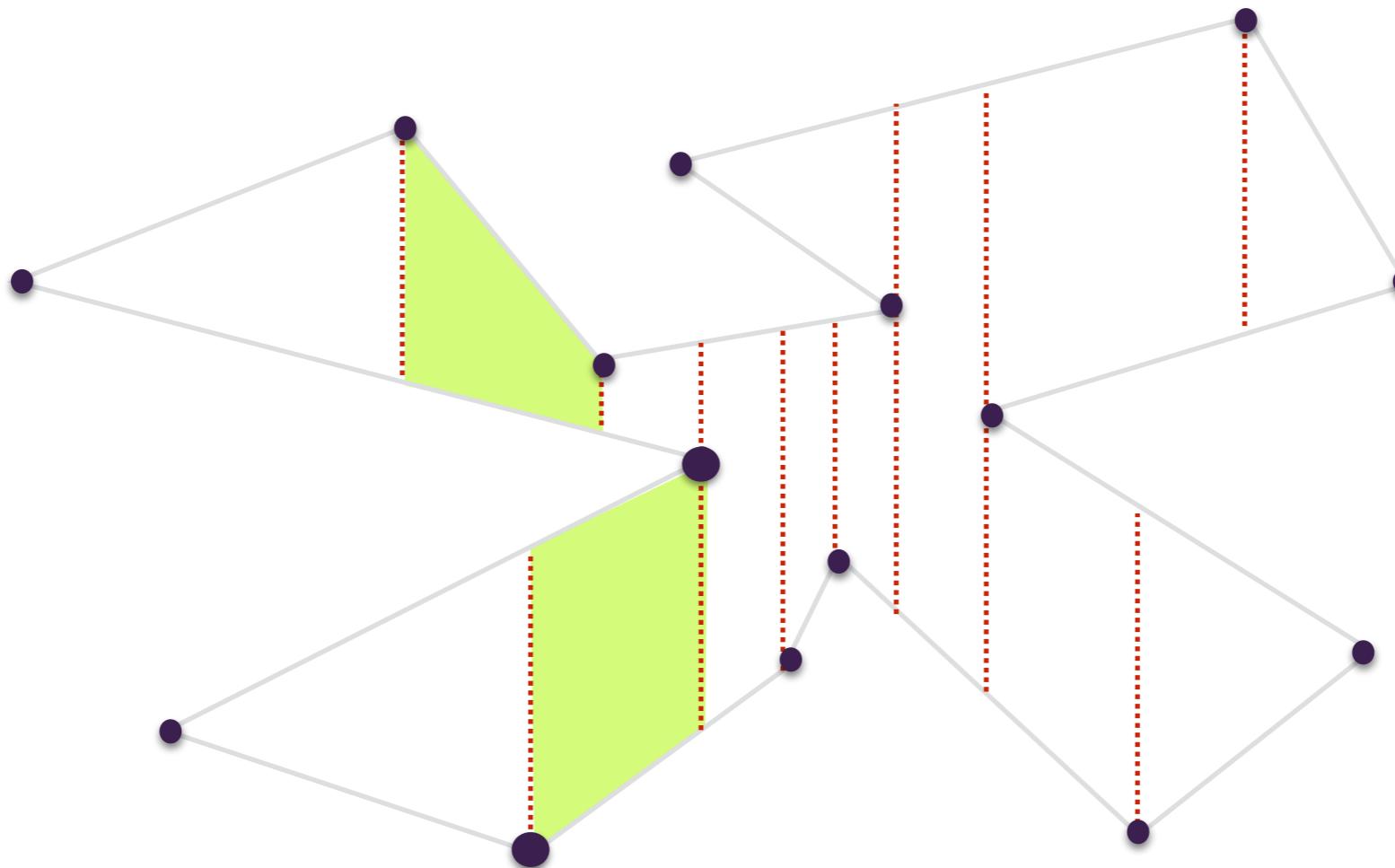
Trapezoid partitions

- Each trapezoid has precisely two vertices of the polygon, one on the left and one on the right. They can be on the top, bottom or middle of the trapezoid.



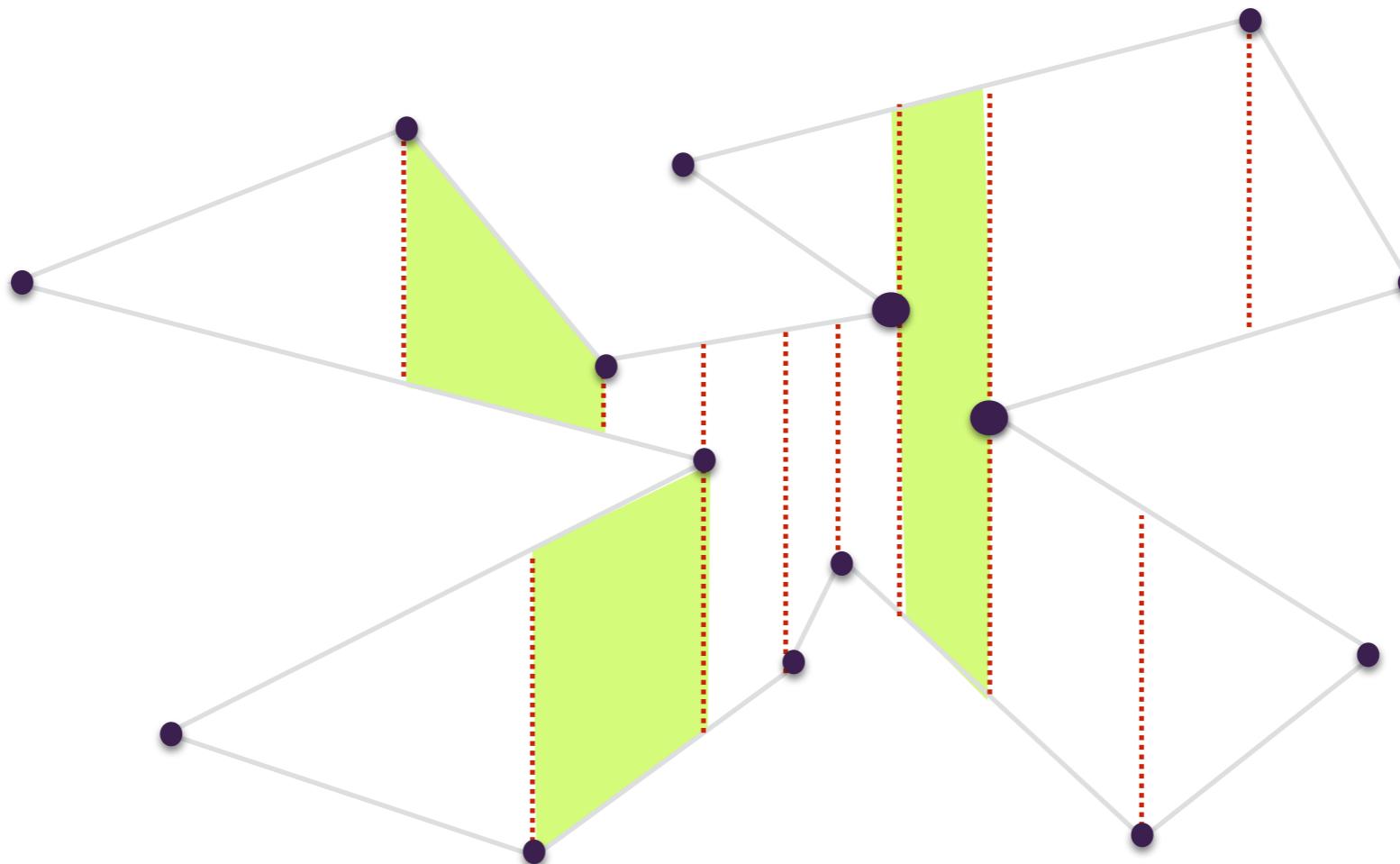
Trapezoid partitions

- Each trapezoid has precisely two vertices of the polygon, one on the left and one on the right. They can be on the top, bottom or middle of the trapezoid.



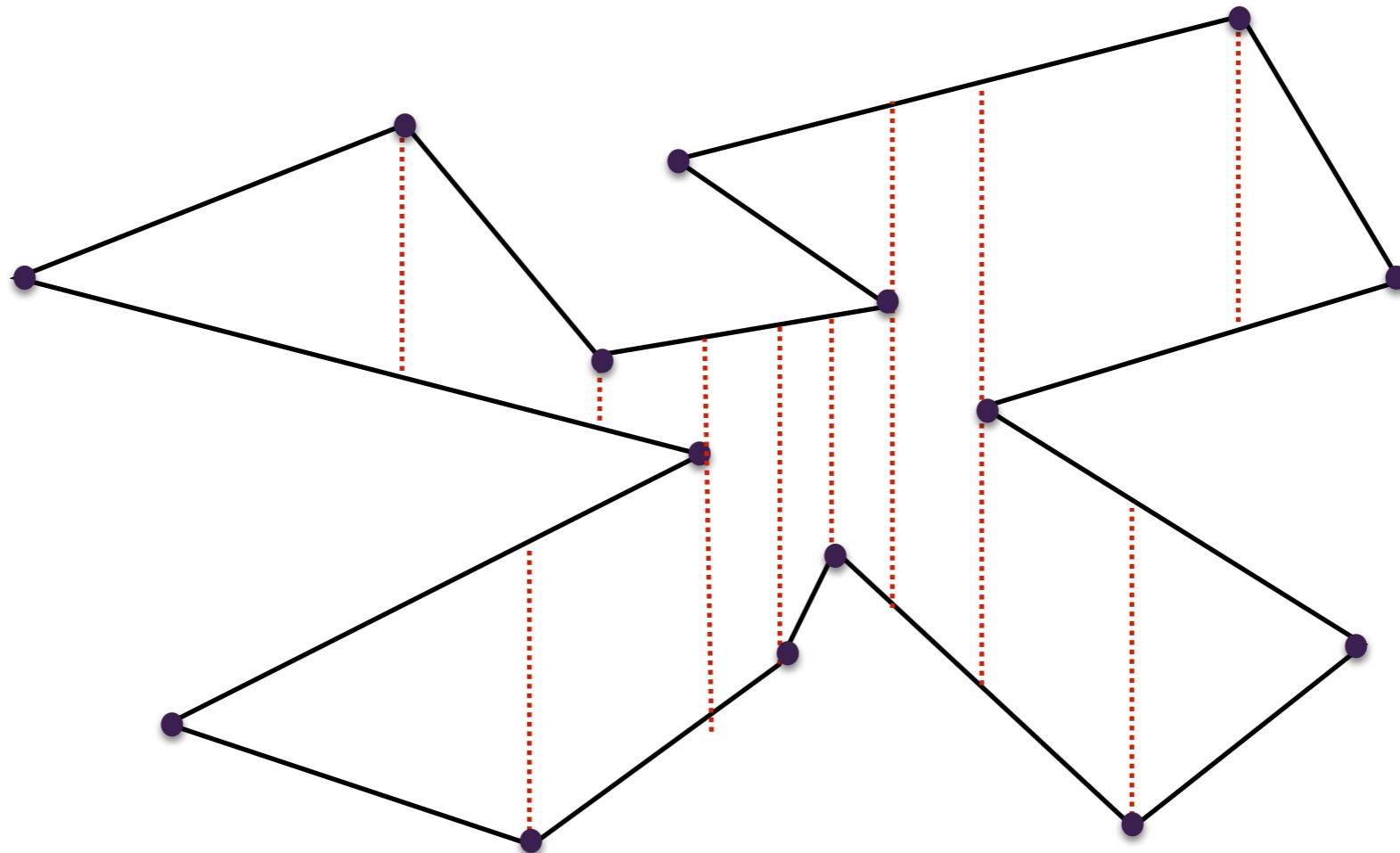
Trapezoid partitions

- Each trapezoid has precisely two vertices of the polygon, one on the left and one on the right. They can be on the top, bottom or middle of the trapezoid.



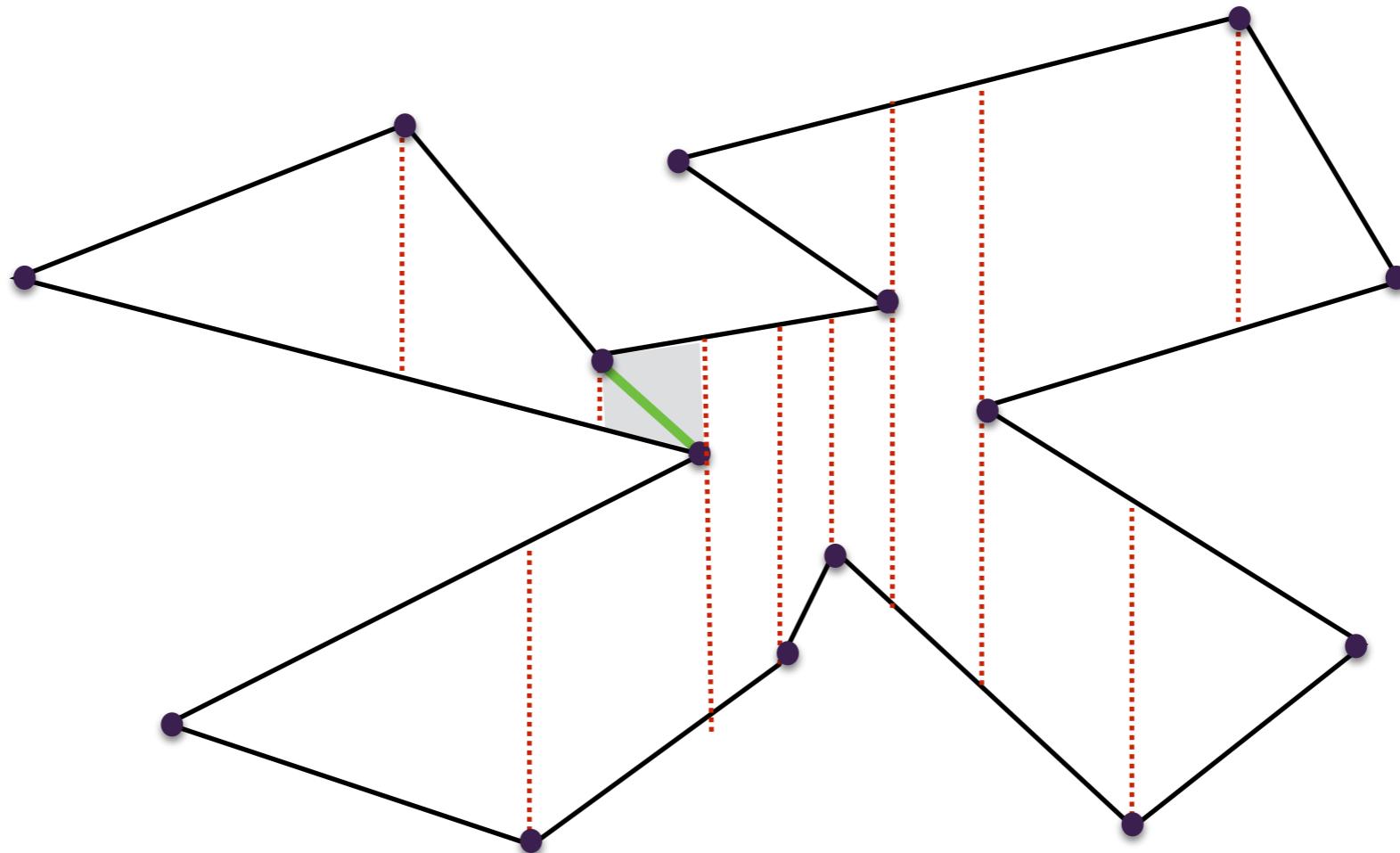
Diagonals

- In each trapezoid: if its two vertices are not on the same edge, they define a **diagonal**.



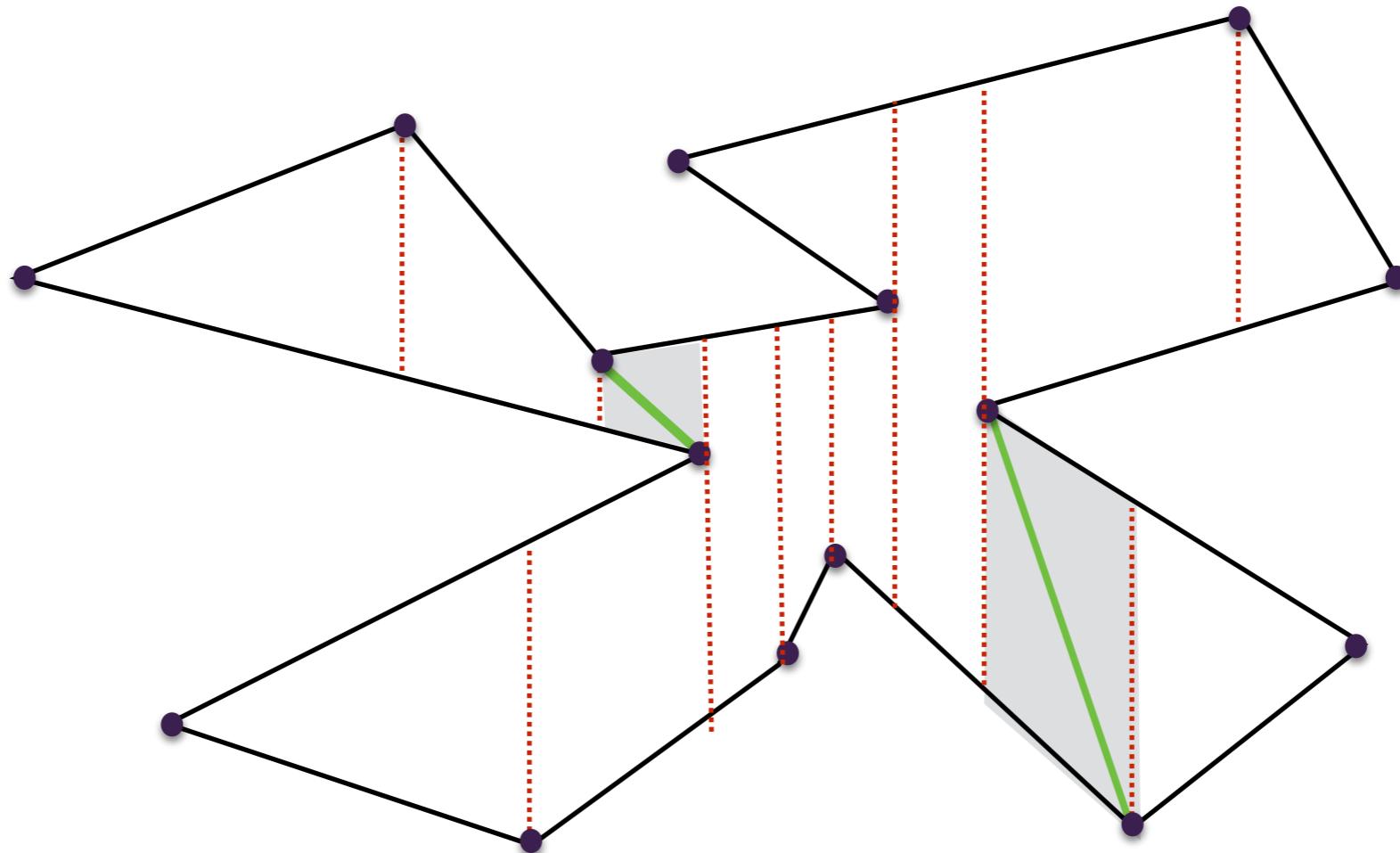
Diagonals

- In each trapezoid: if its two vertices are not on the same edge, they define a **diagonal**.



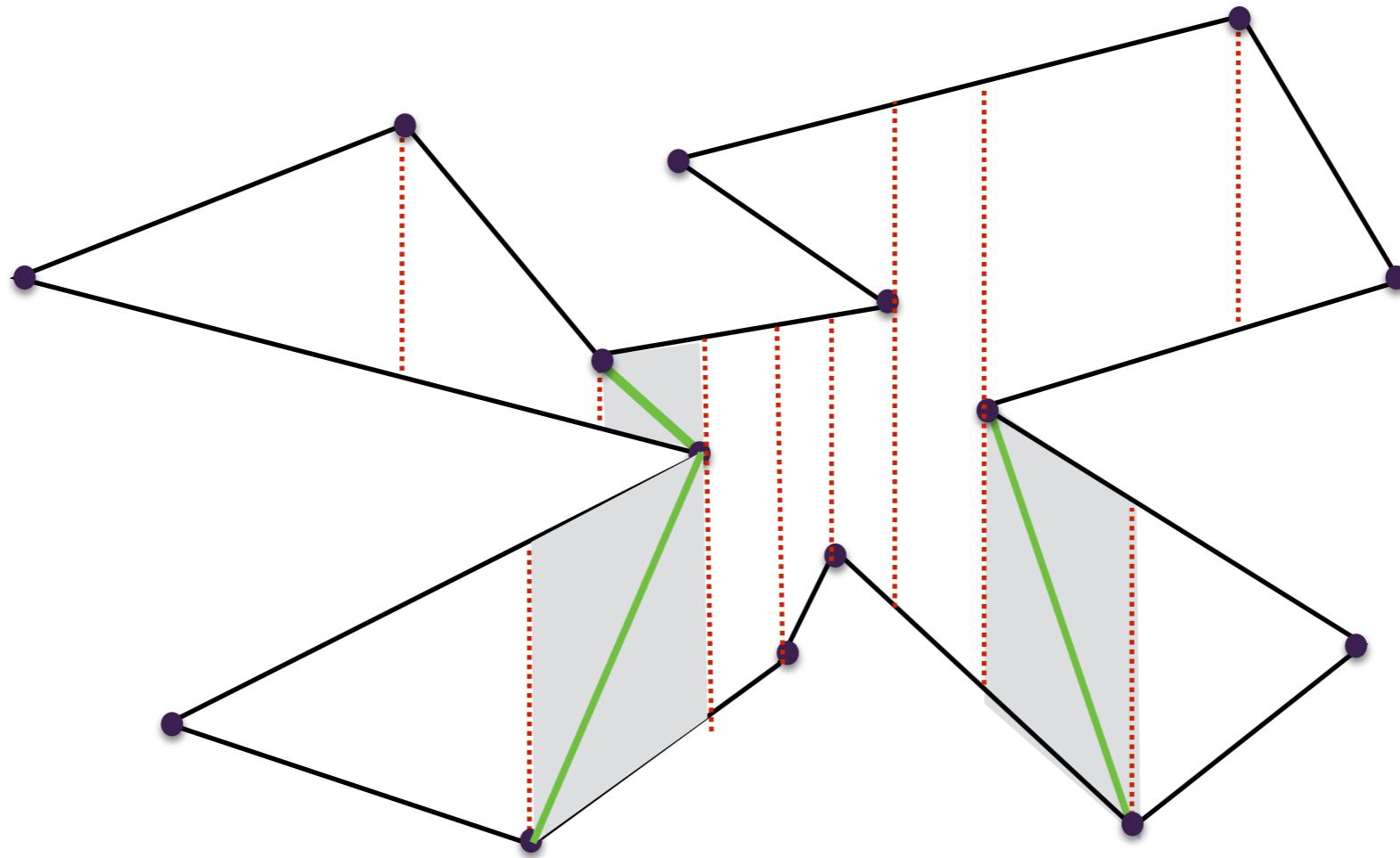
Diagonals

- In each trapezoid: if its two vertices are not on the same edge, they define a **diagonal**.



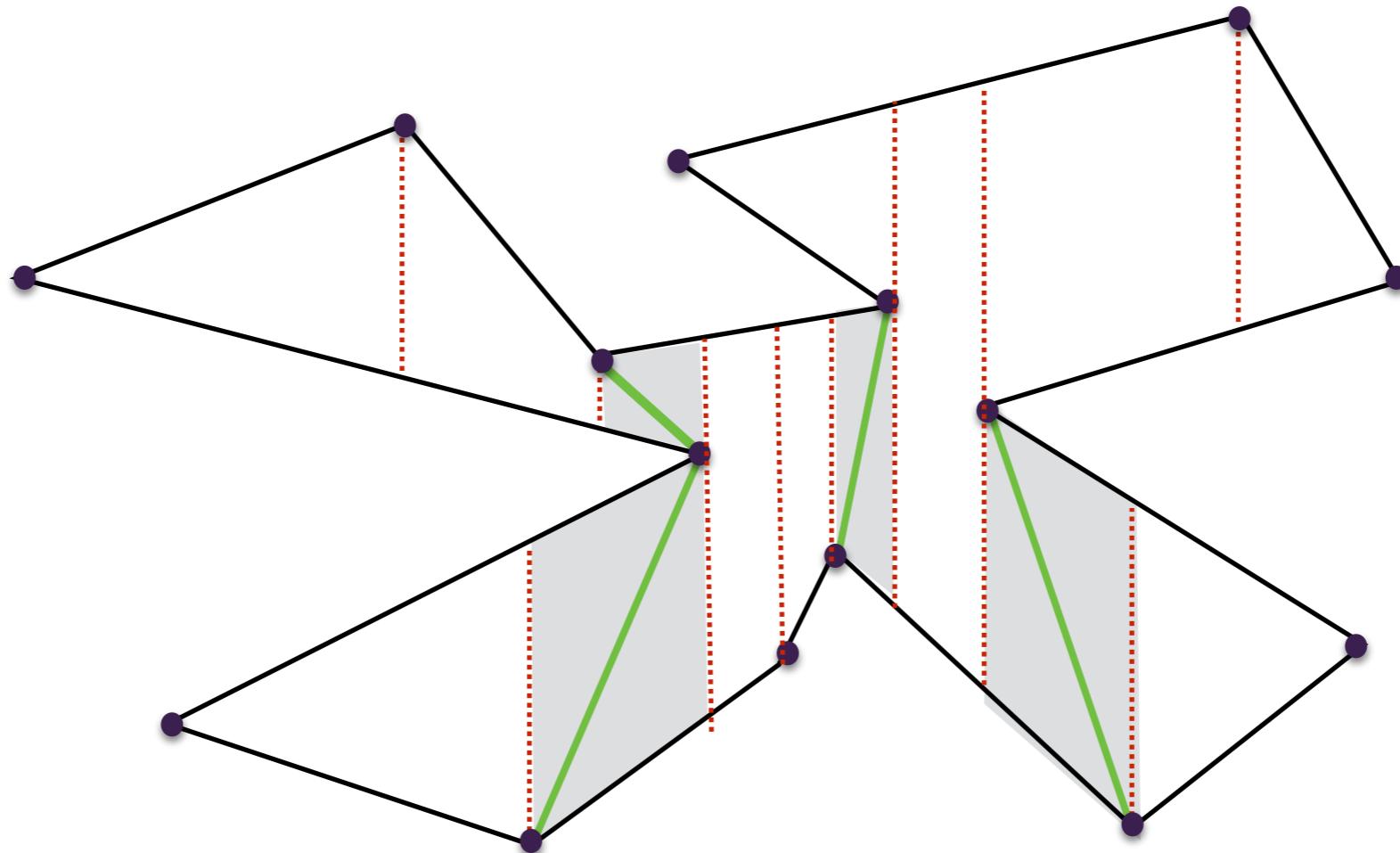
Diagonals

- In each trapezoid: if its two vertices are not on the same edge, they define a **diagonal**.



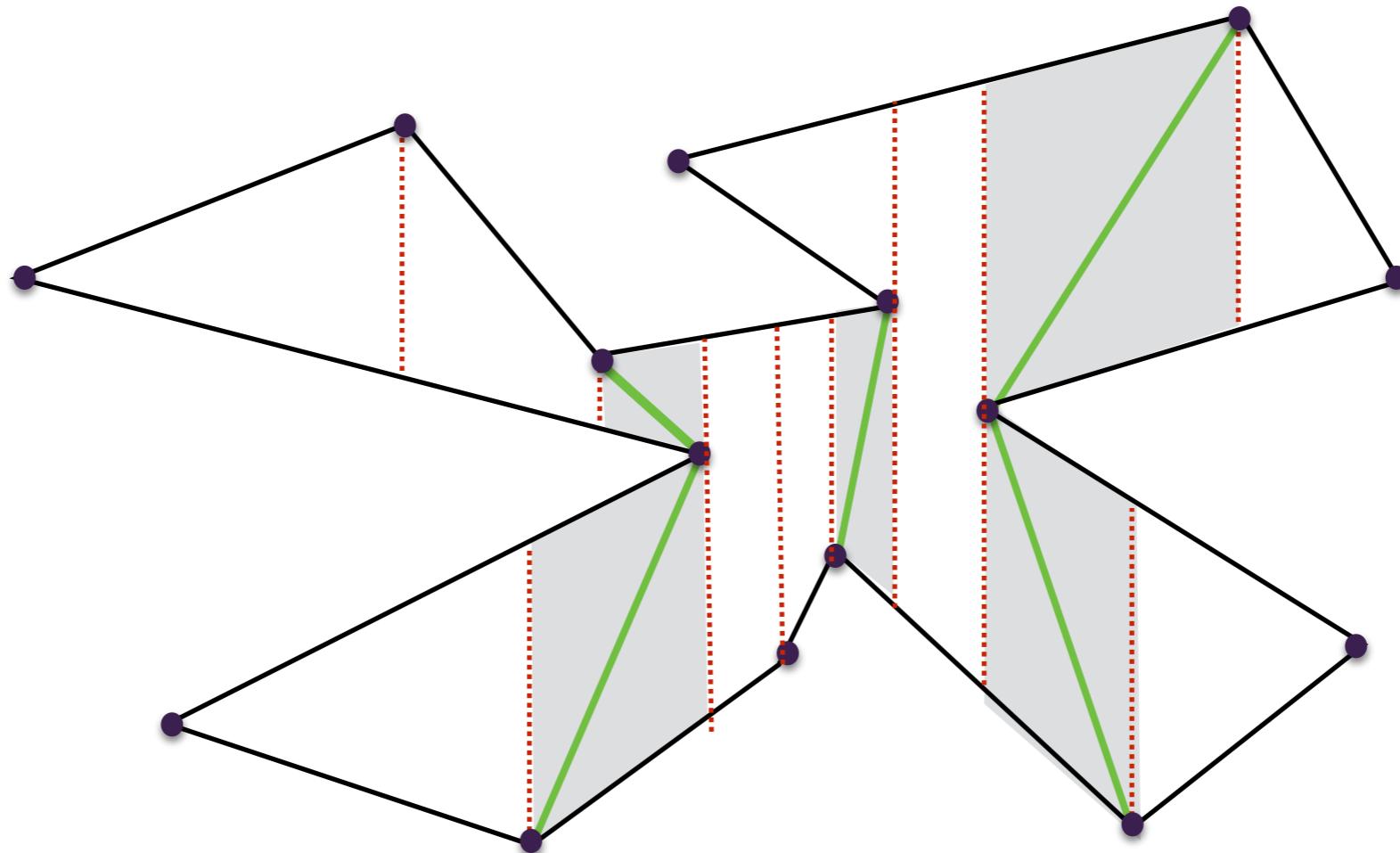
Diagonals

- In each trapezoid: if its two vertices are not on the same edge, they define a **diagonal**.



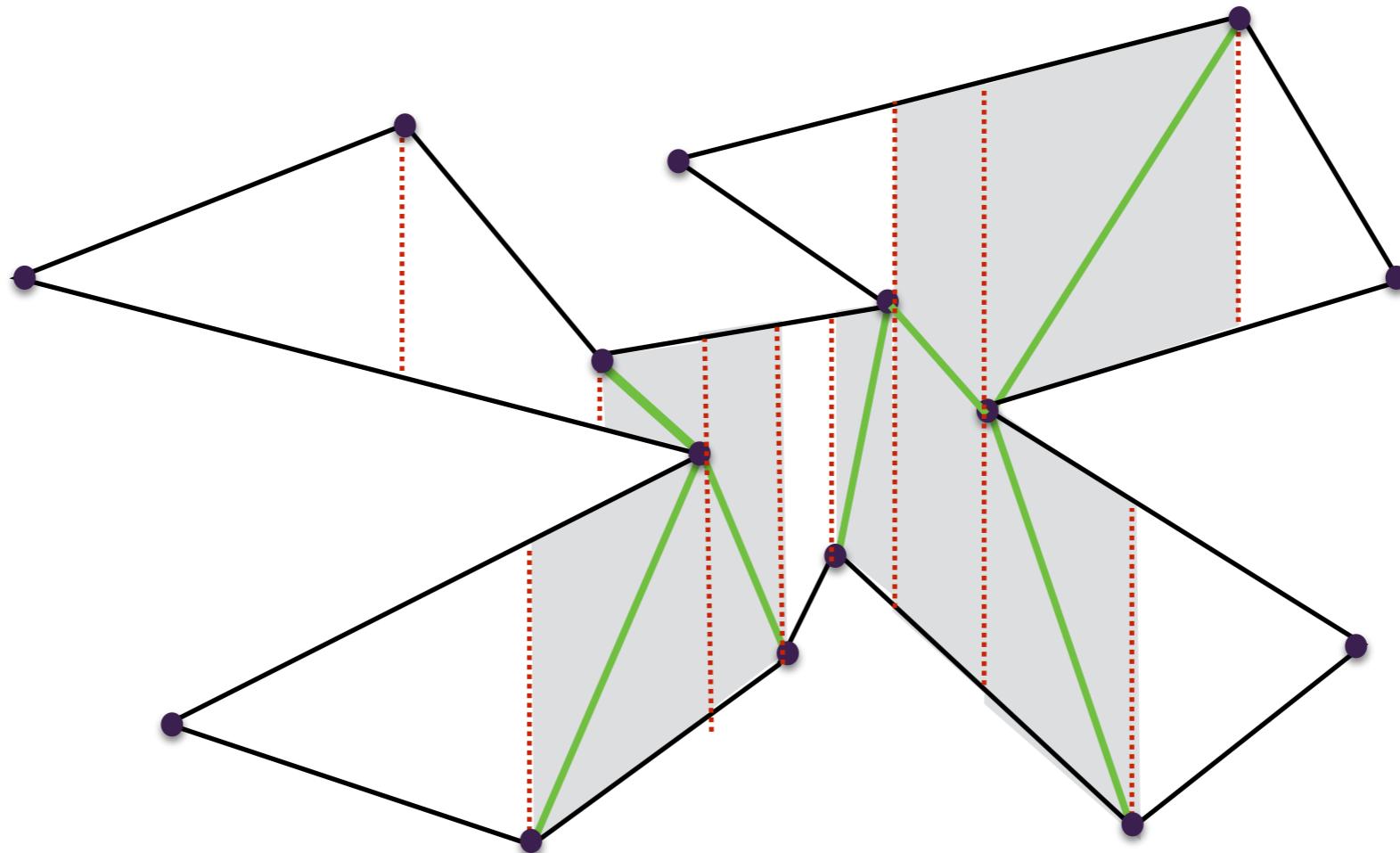
Diagonals

- In each trapezoid: if its two vertices are not on the same edge, they define a **diagonal**.

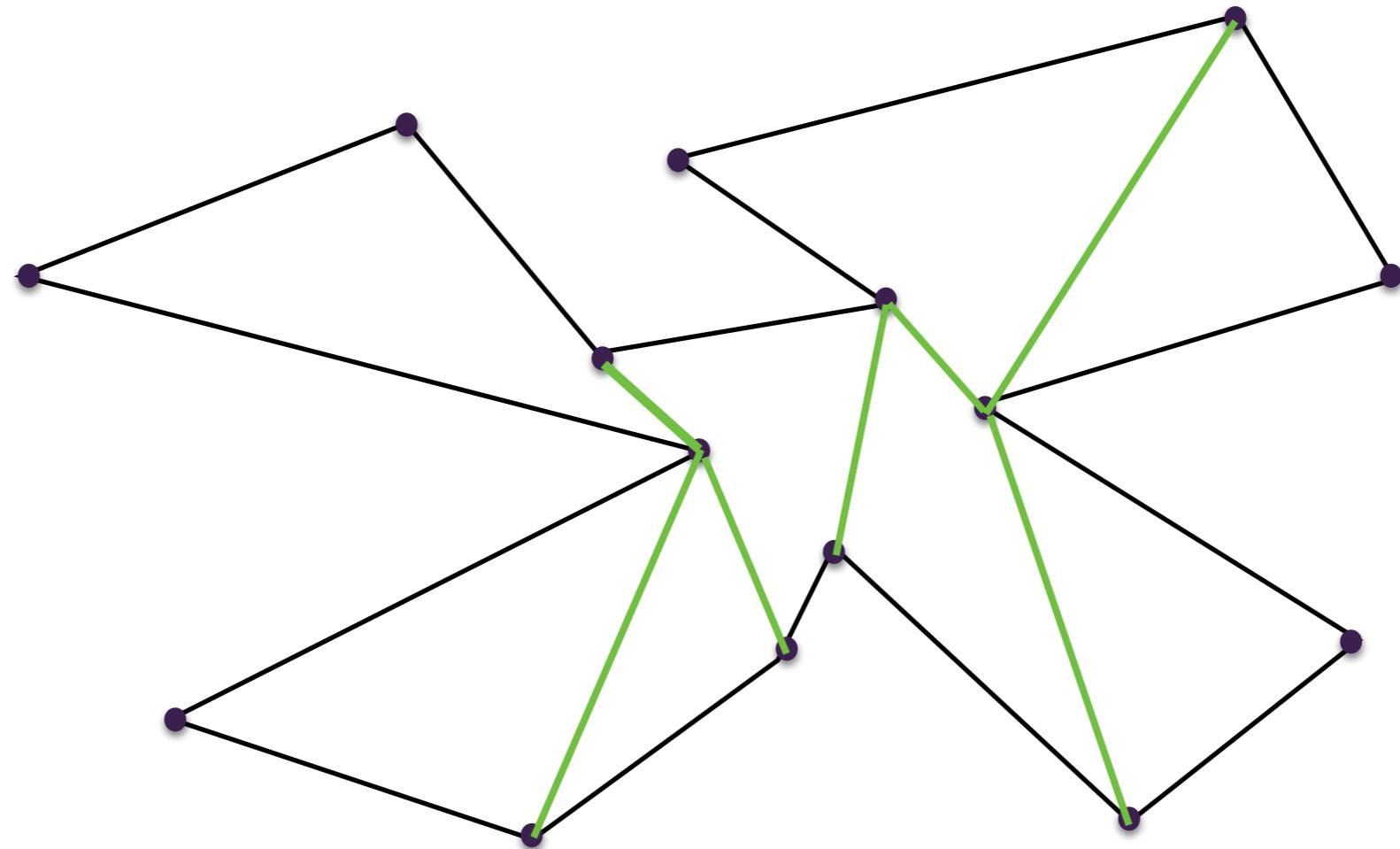


Diagonals

- In each trapezoid: if its two vertices are not on the same edge, they define a **diagonal**.



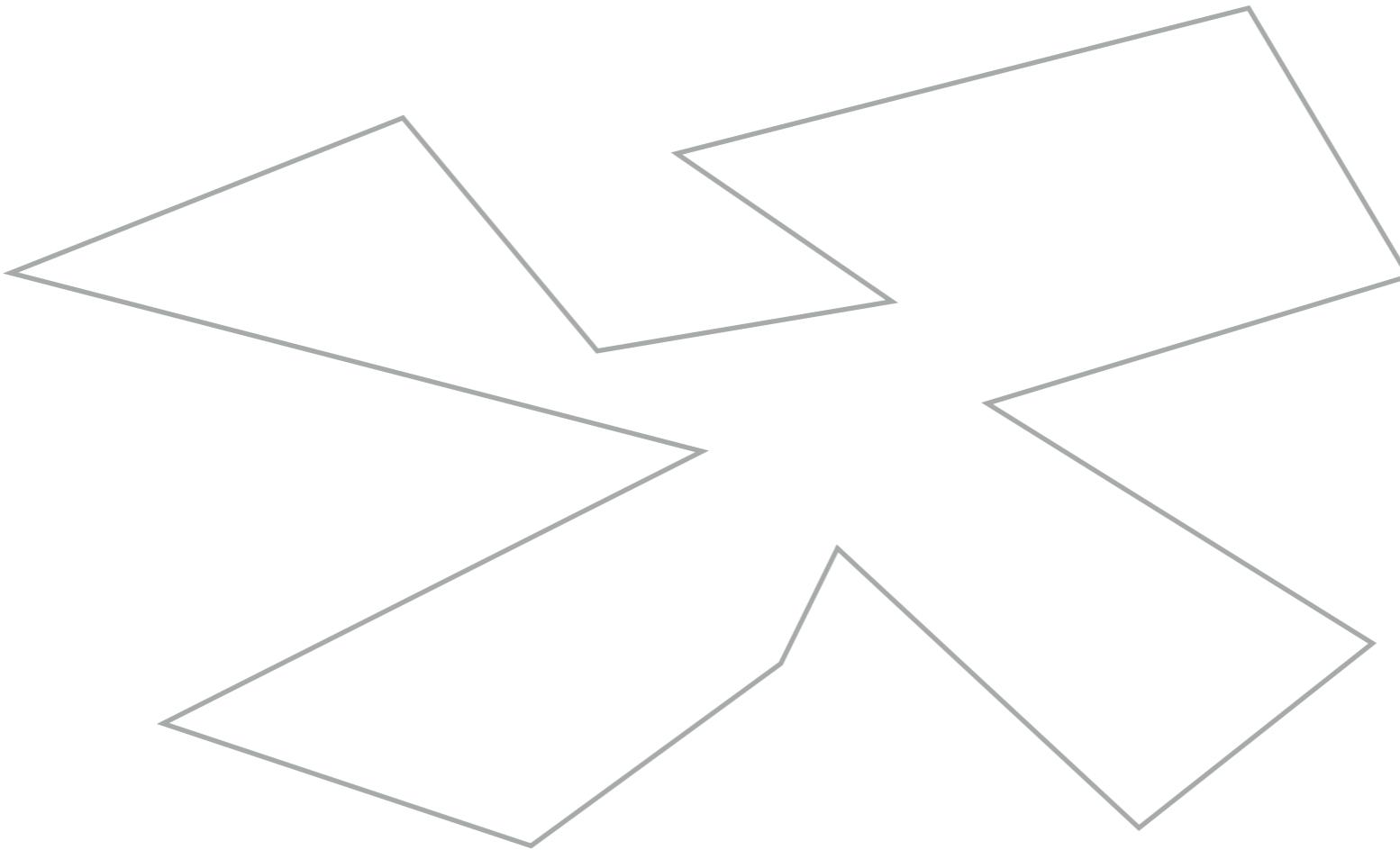
Diagonals in the trapezoid partition of P



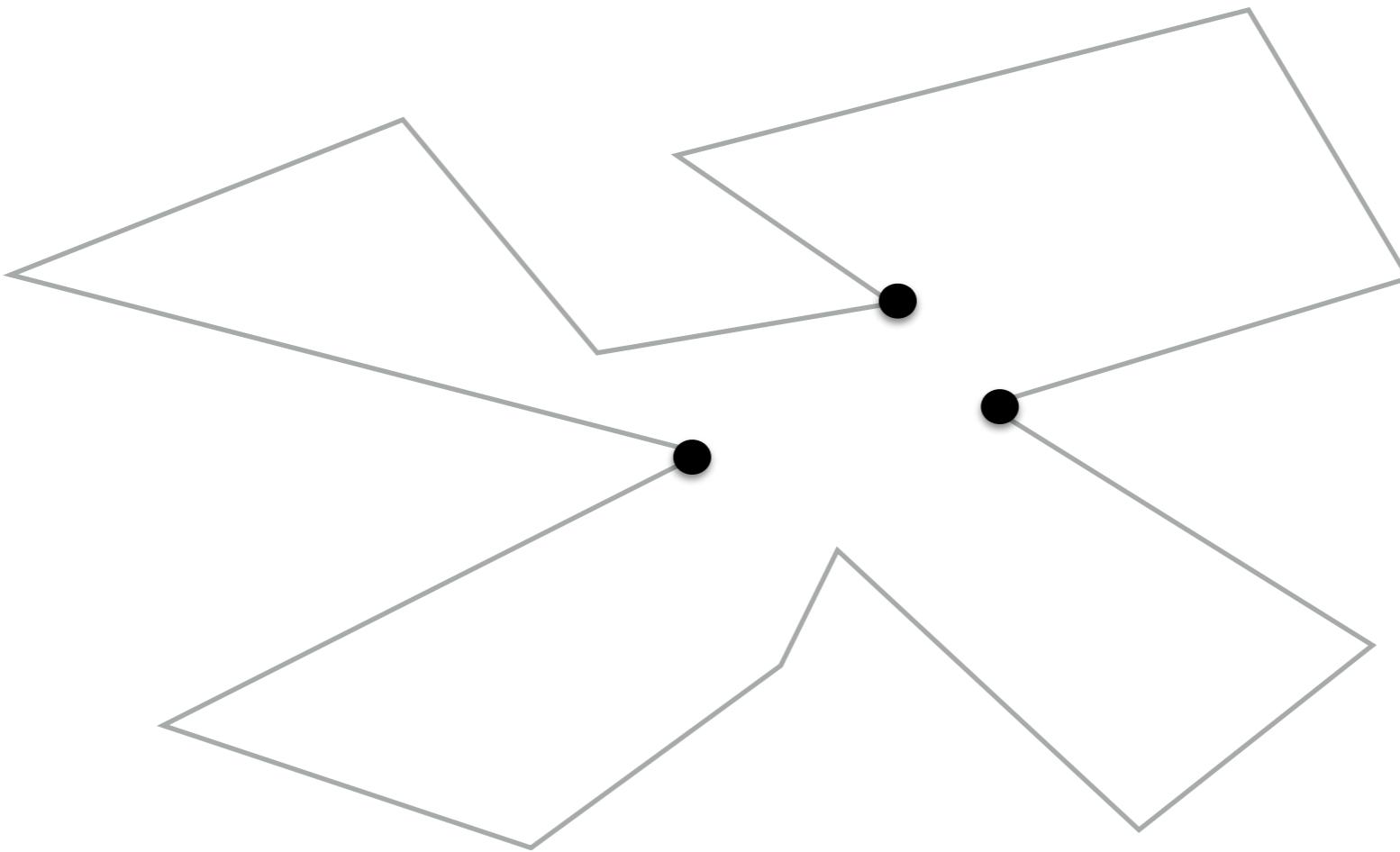
So, where is this going?

We can use the diagonals in the trapezoid partition of P to
“split” the cusps

Removing cusps

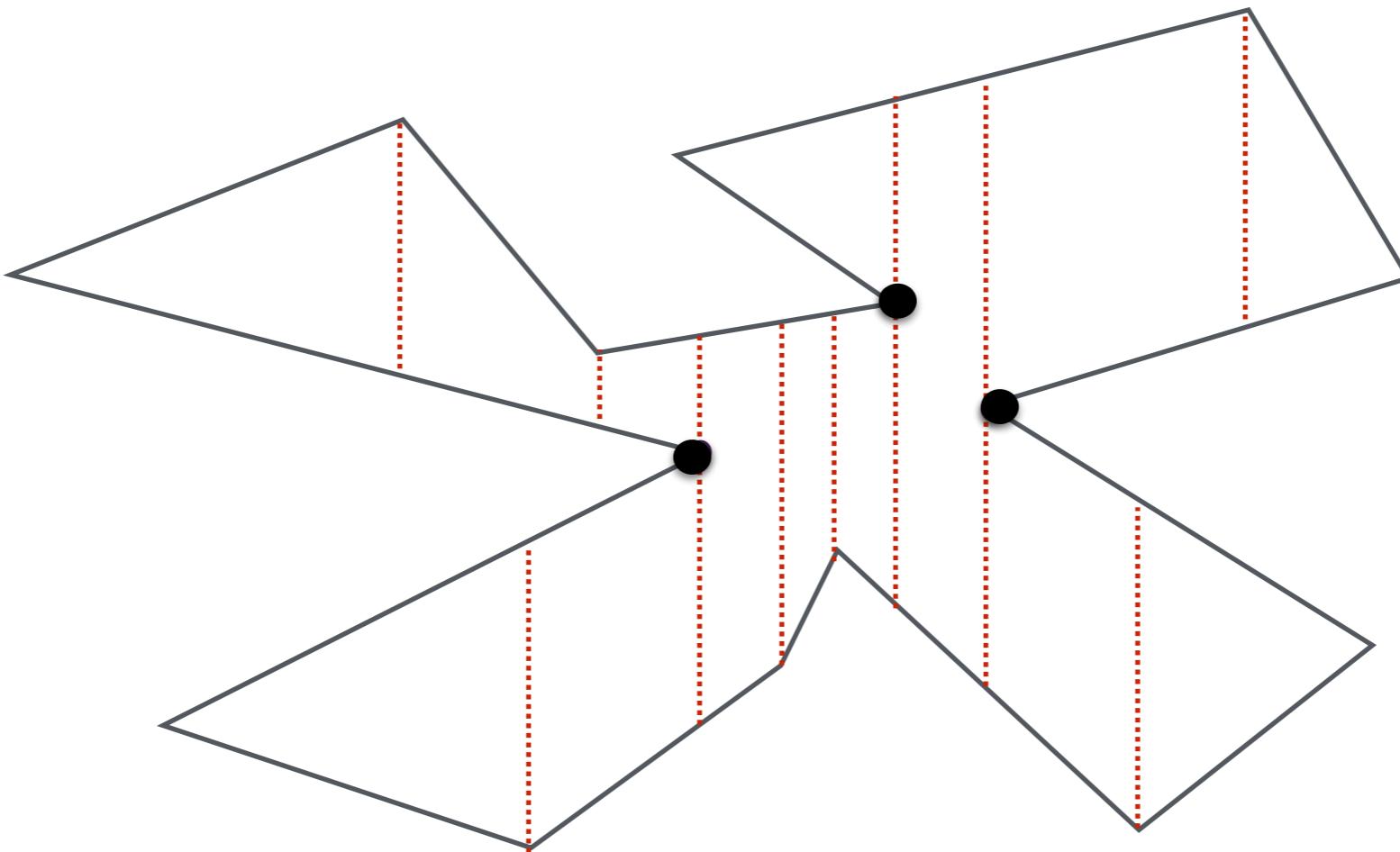


Removing cusps



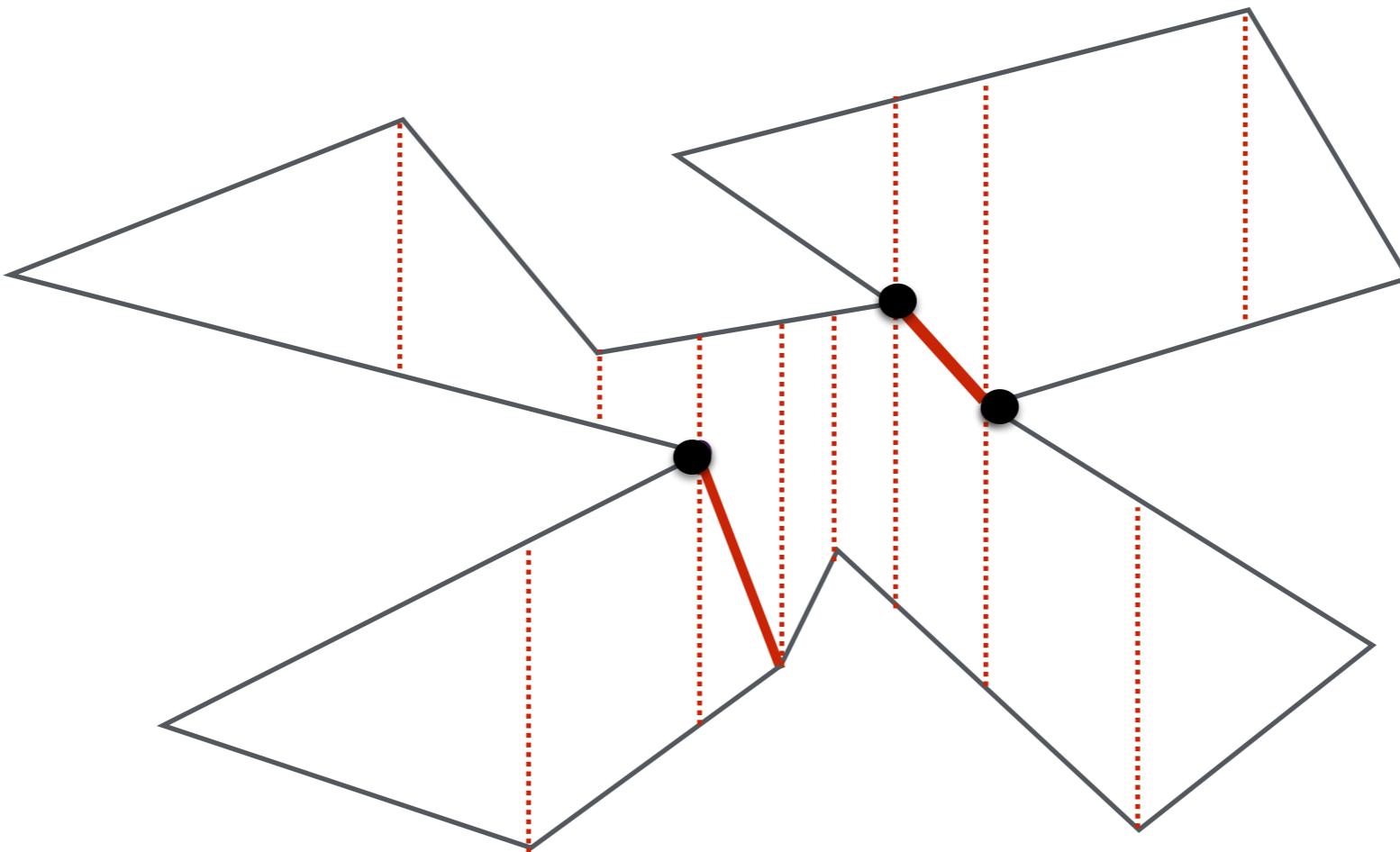
1. Identify cusp vertices

Removing cusps



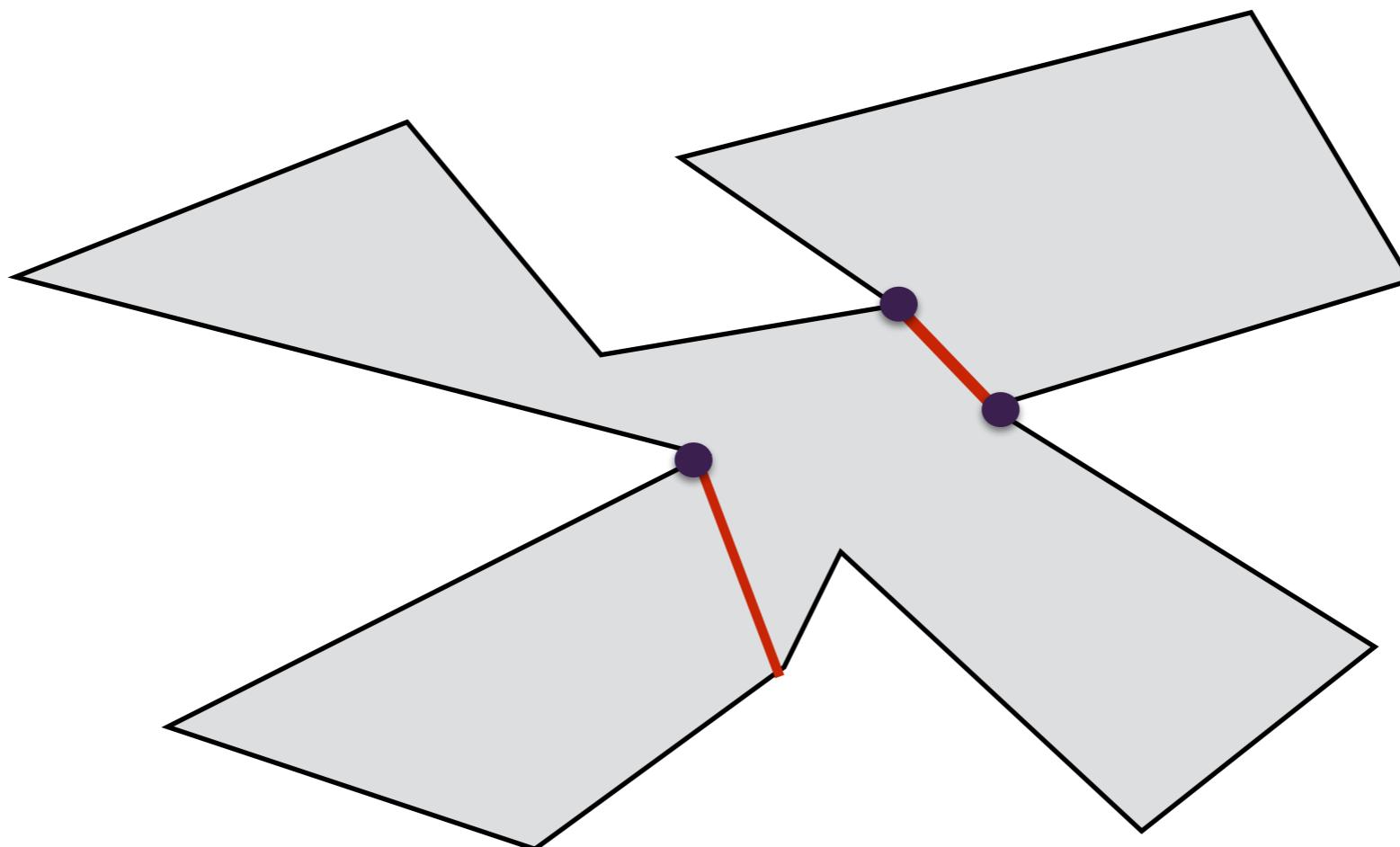
1. Identify cusp vertices
2. Compute a trapezoid partition of P

Removing cusps



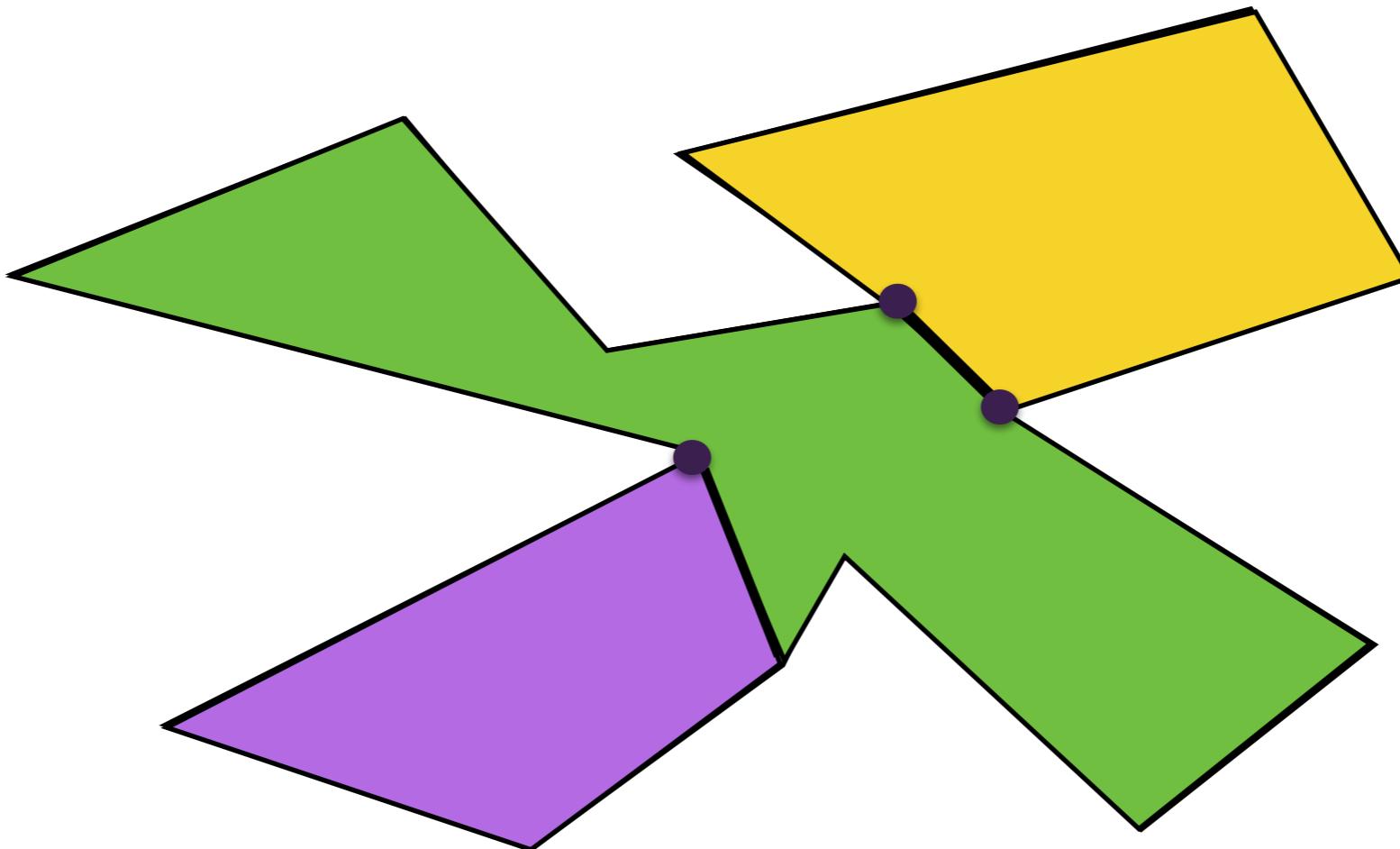
1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. Add diagonal before/after each cusp

Removing cusps



This partitions the polygon into monotone pieces.

Removing cusps



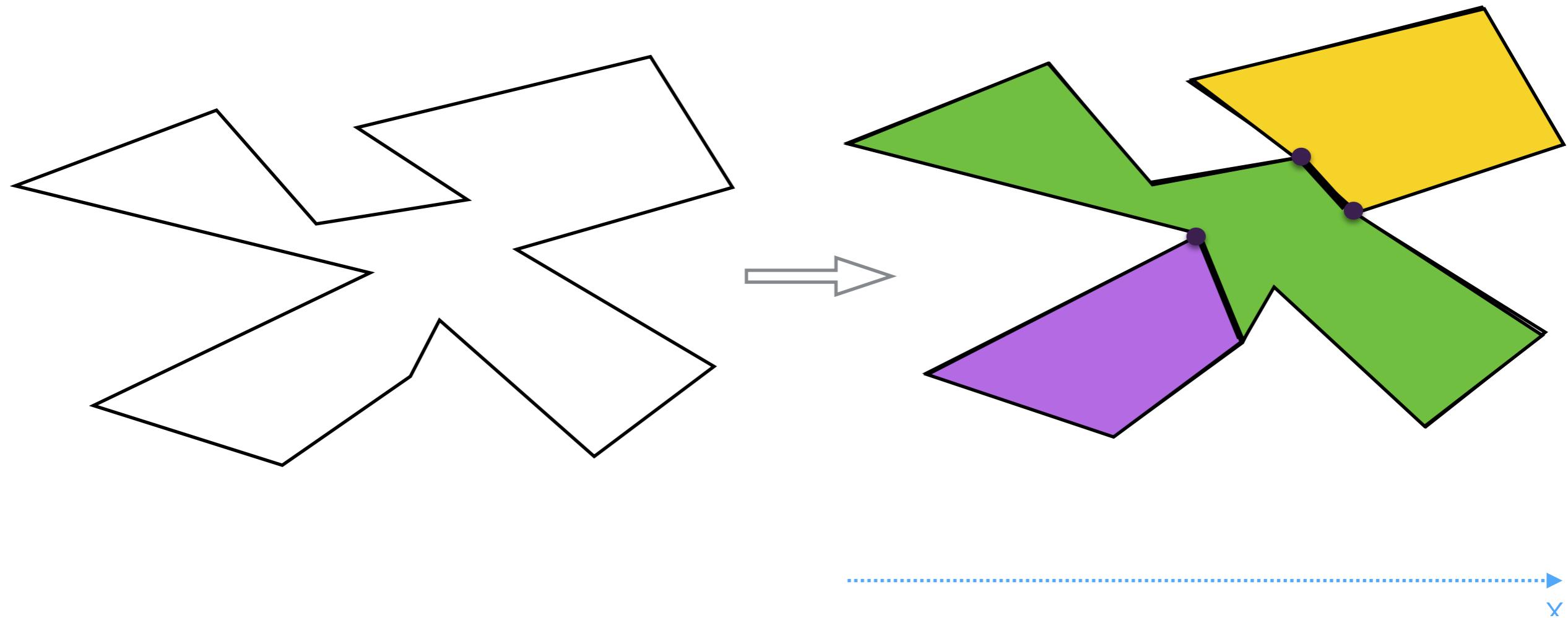
This partitions the polygon into monotone pieces.



Summary

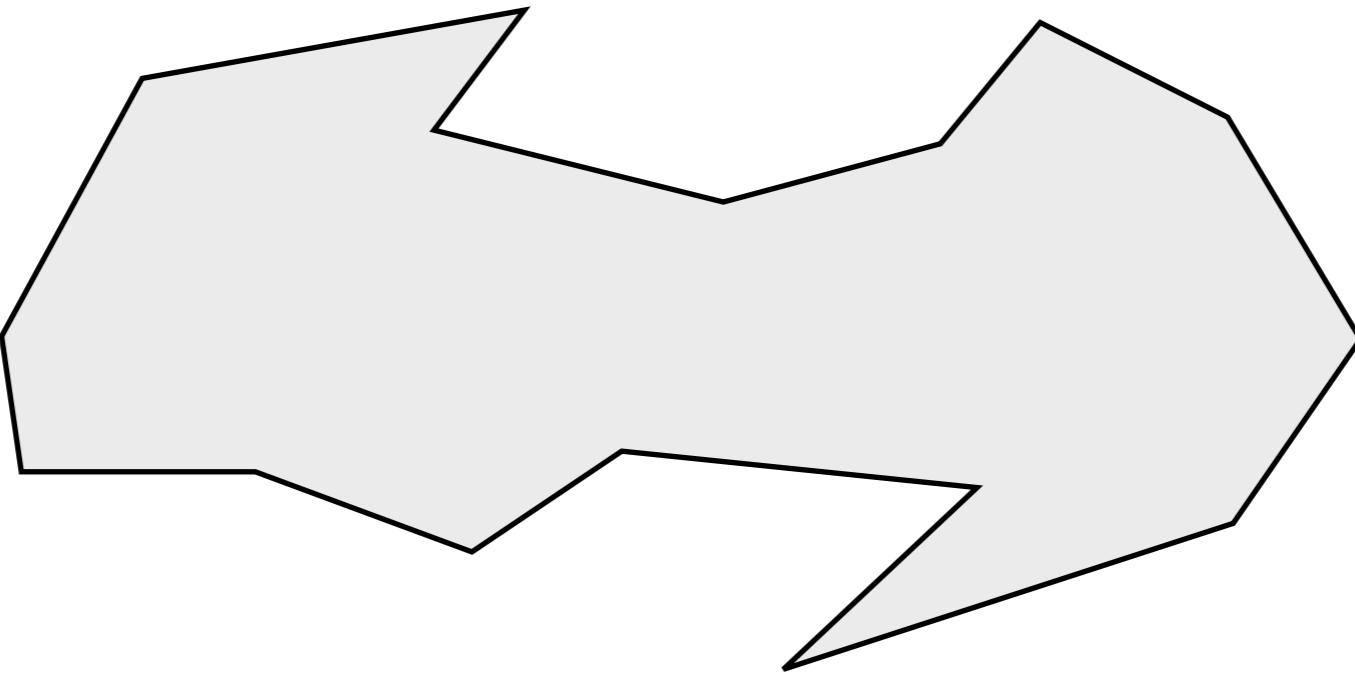
Partition P into monotone polygons

1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. Add diagonal before/after each cusp

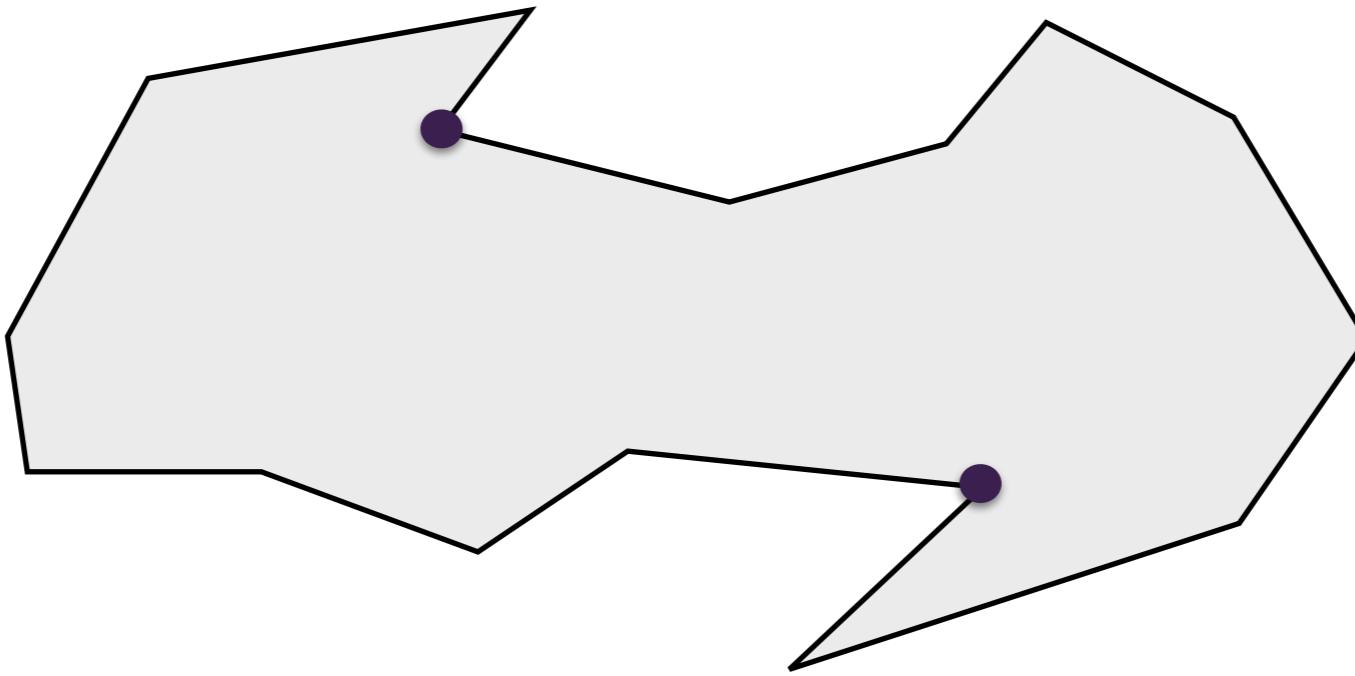


Another example

Removing cusps

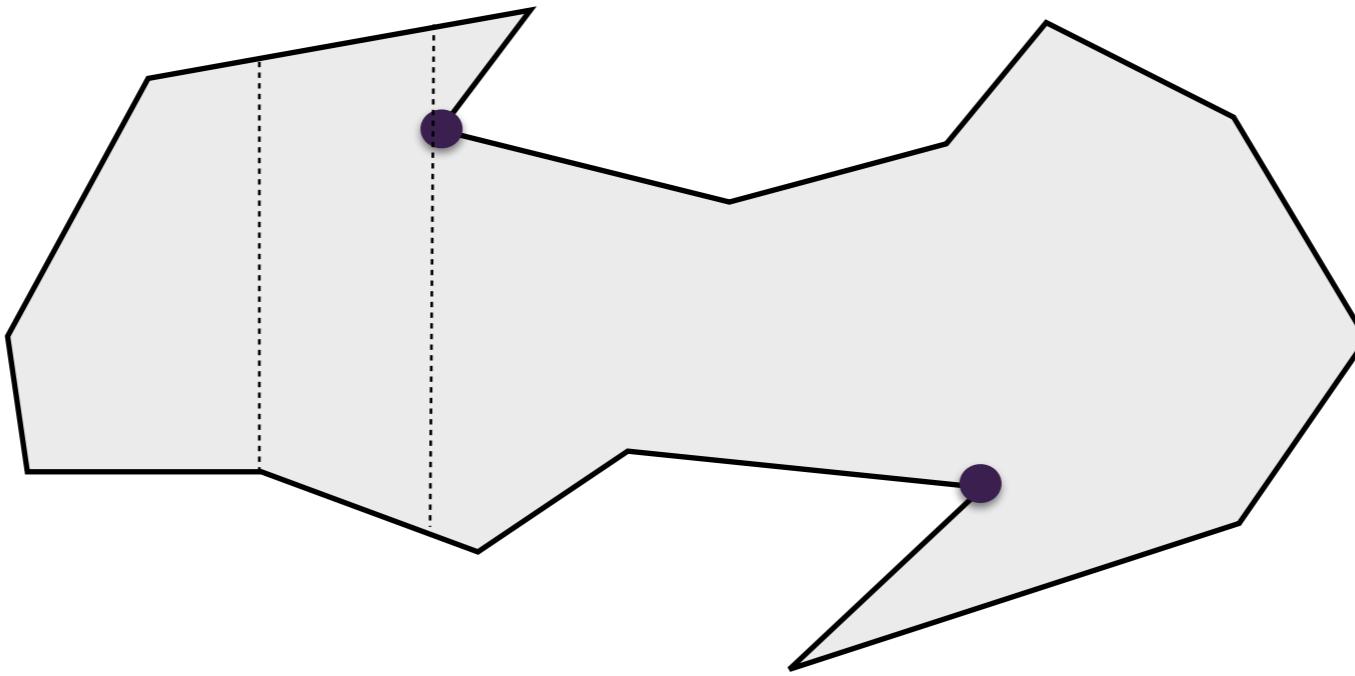


Removing cusps



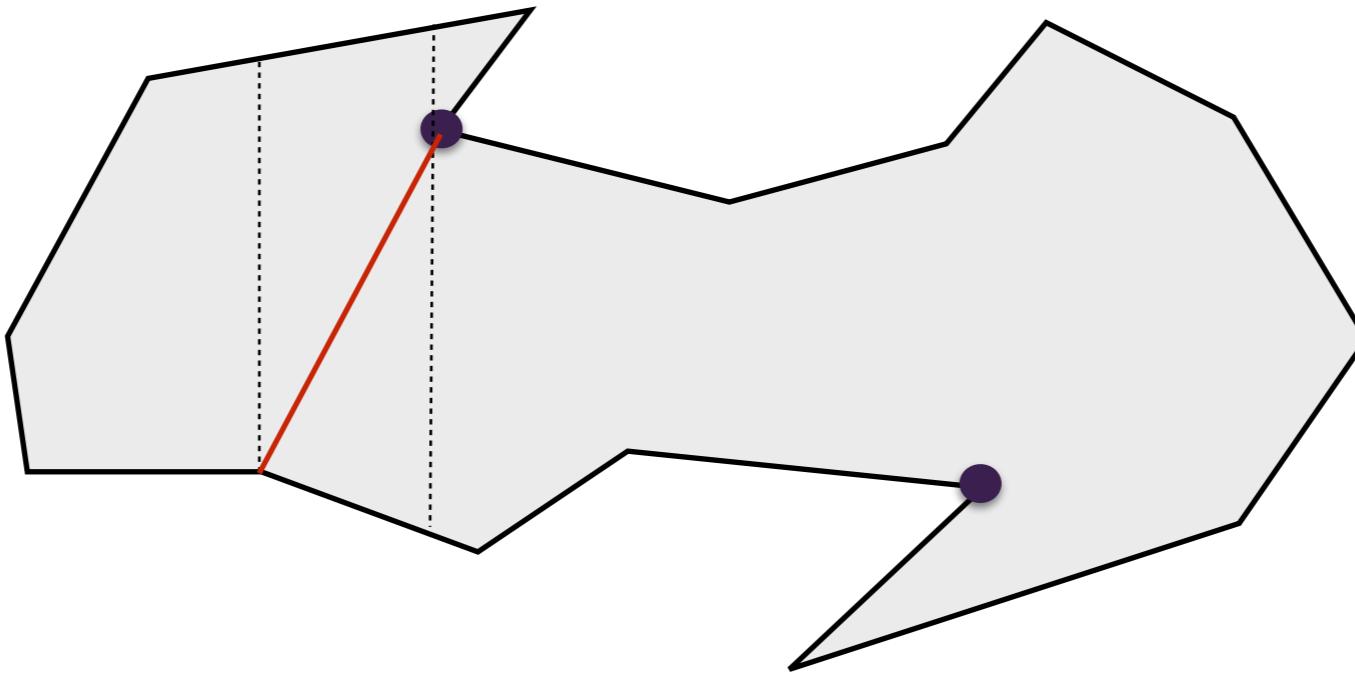
1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For each cusp vertex, add diagonal in trapezoid before/after the cusp

Removing cusps



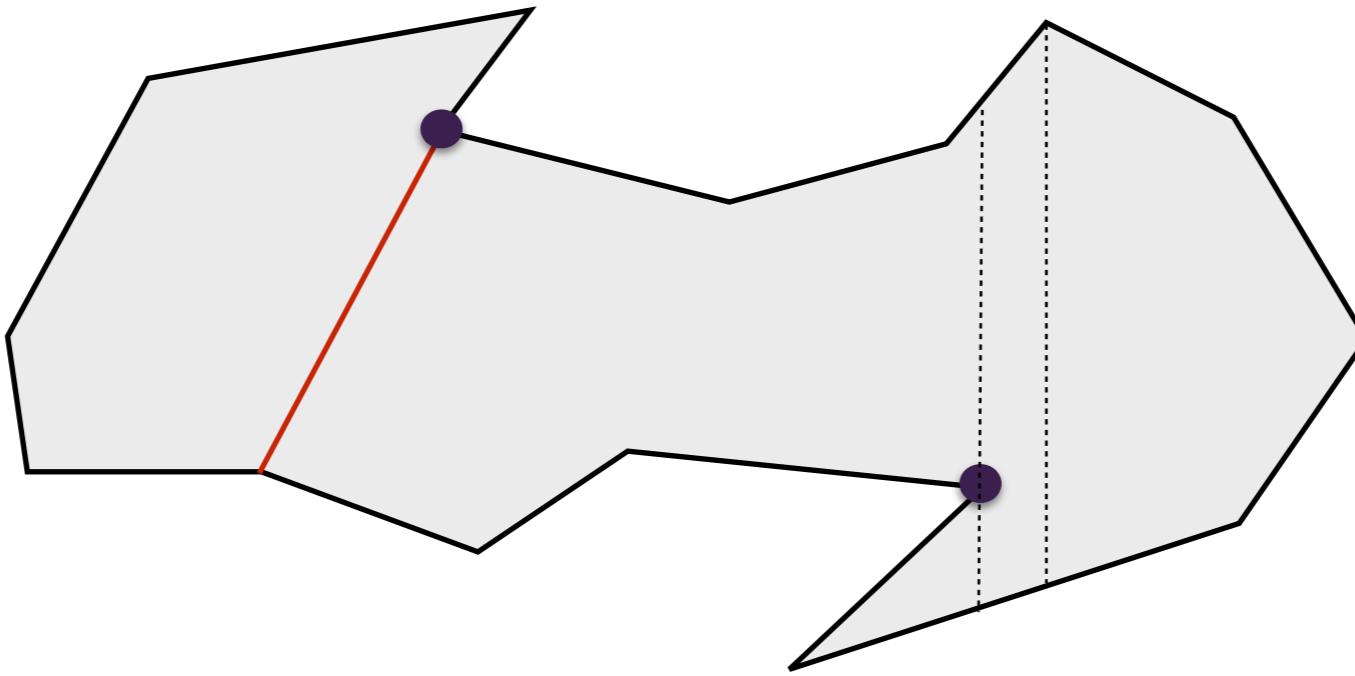
1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For each cusp vertex, add diagonal in trapezoid before/after the cusp

Removing cusps



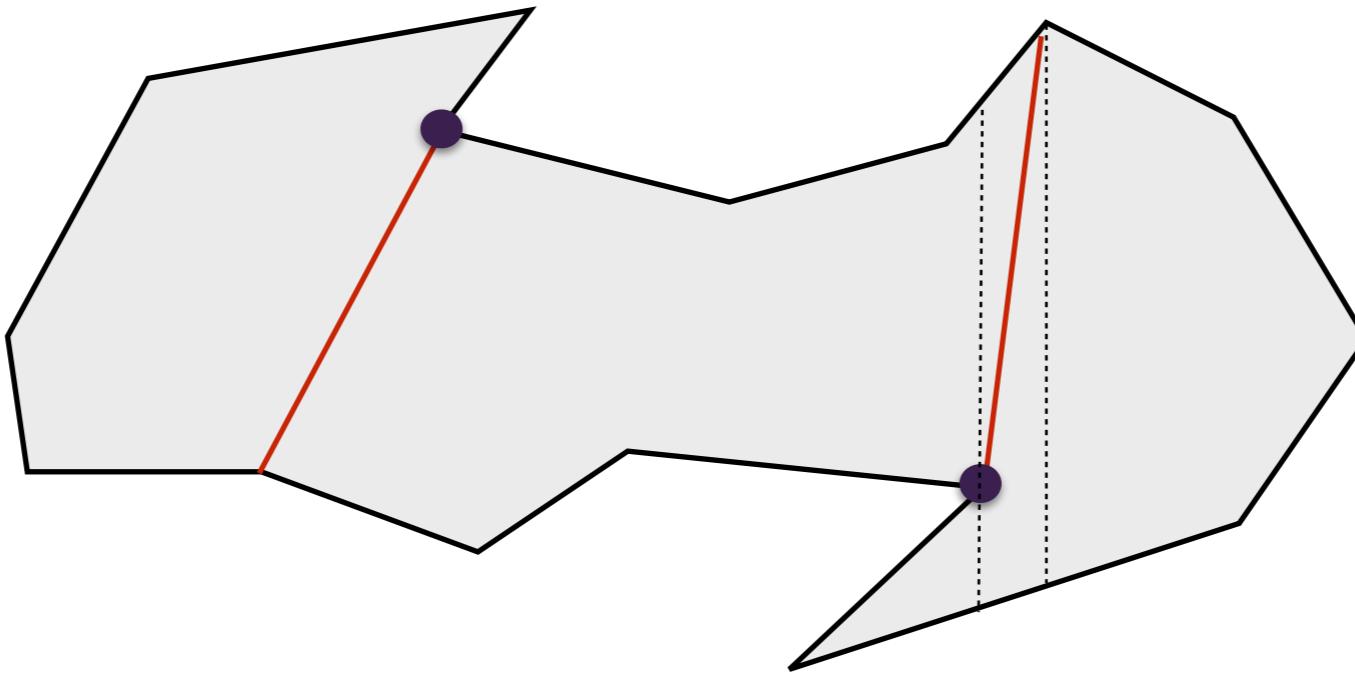
1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For each cusp vertex, add diagonal in trapezoid before/after the cusp

Removing cusps



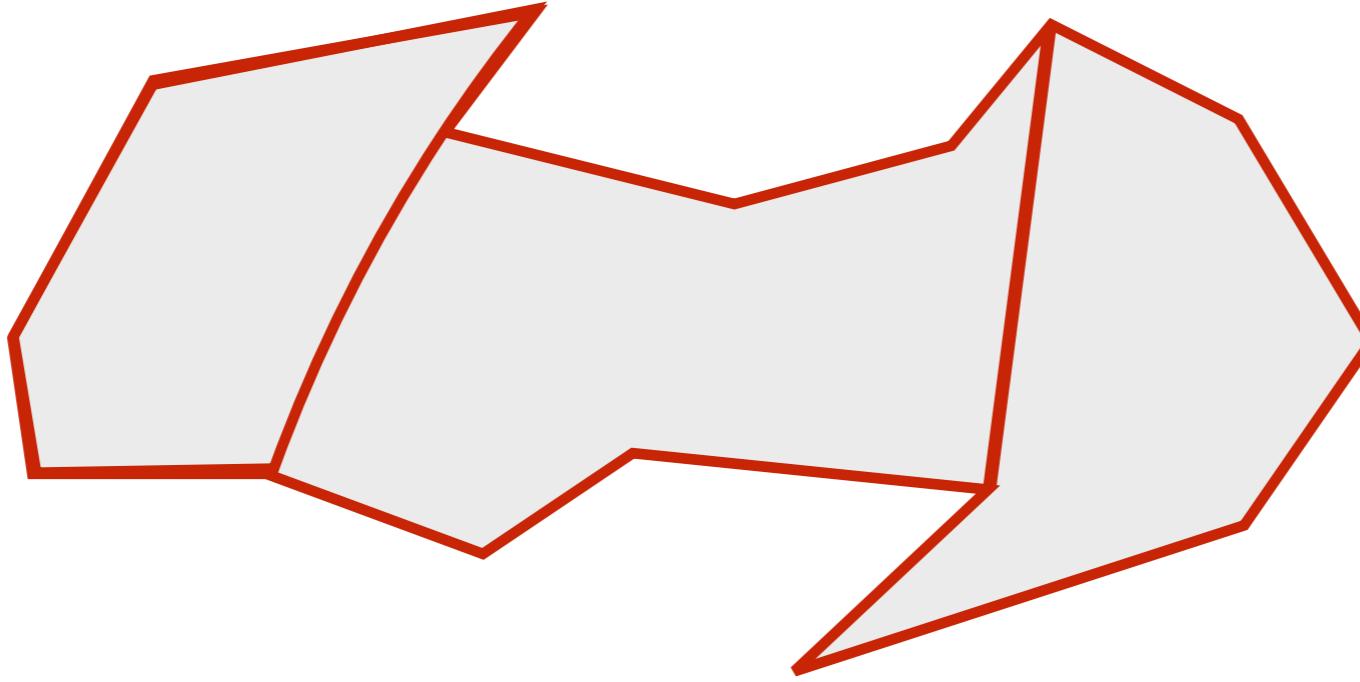
1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For each cusp vertex, add diagonal in trapezoid before/after the cusp

Removing cusps



1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For each cusp vertex, add diagonal in trapezoid before/after the cusp

Removing cusps

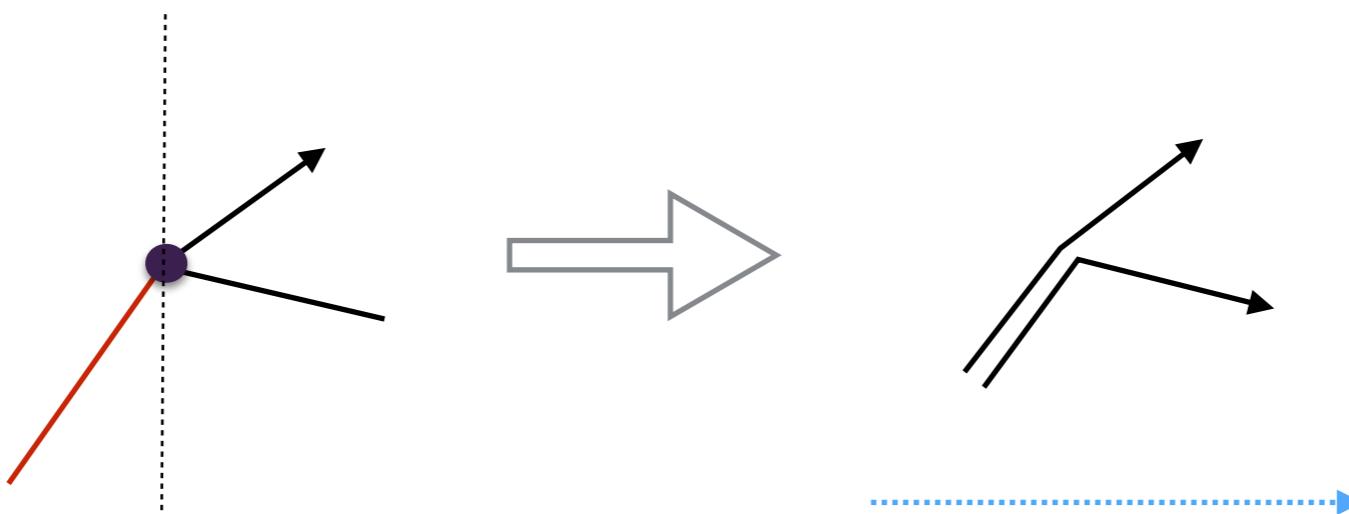
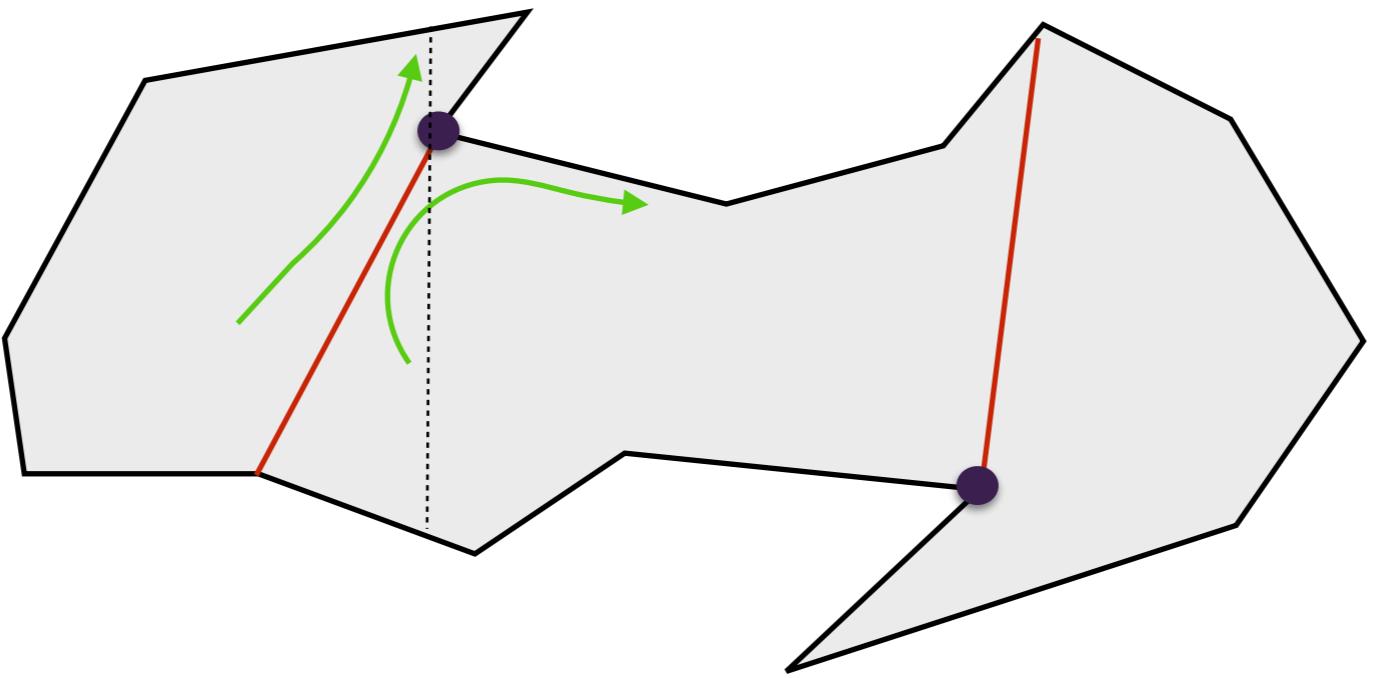


1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For each cusp vertex, add diagonal in trapezoid before/after the cusp

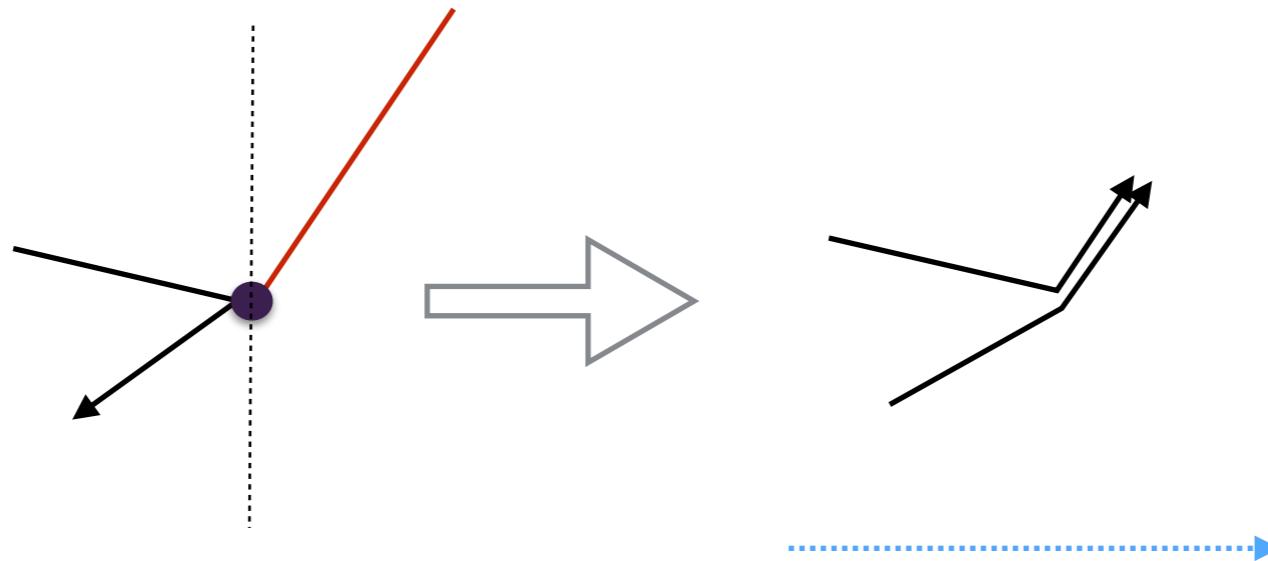
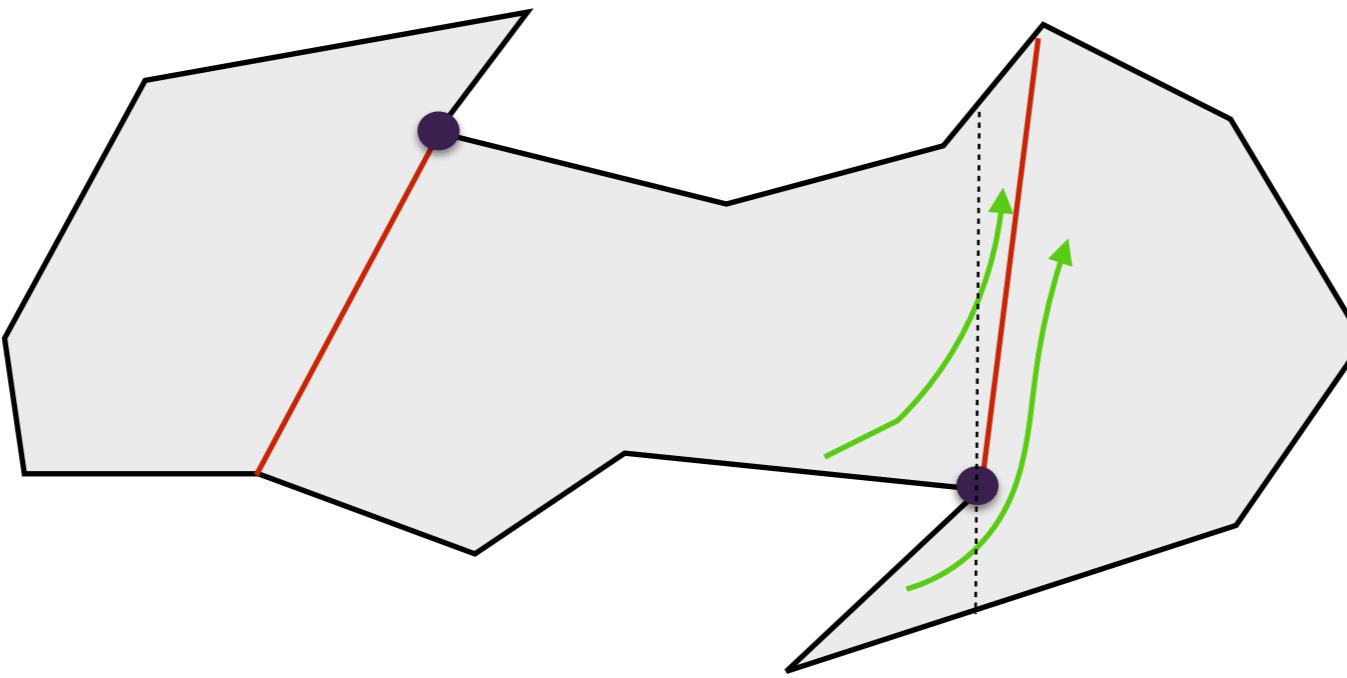
This creates a partition of P .

Claim: The resulting polygons have no cusps and thus are monotone (by prev. theorem).

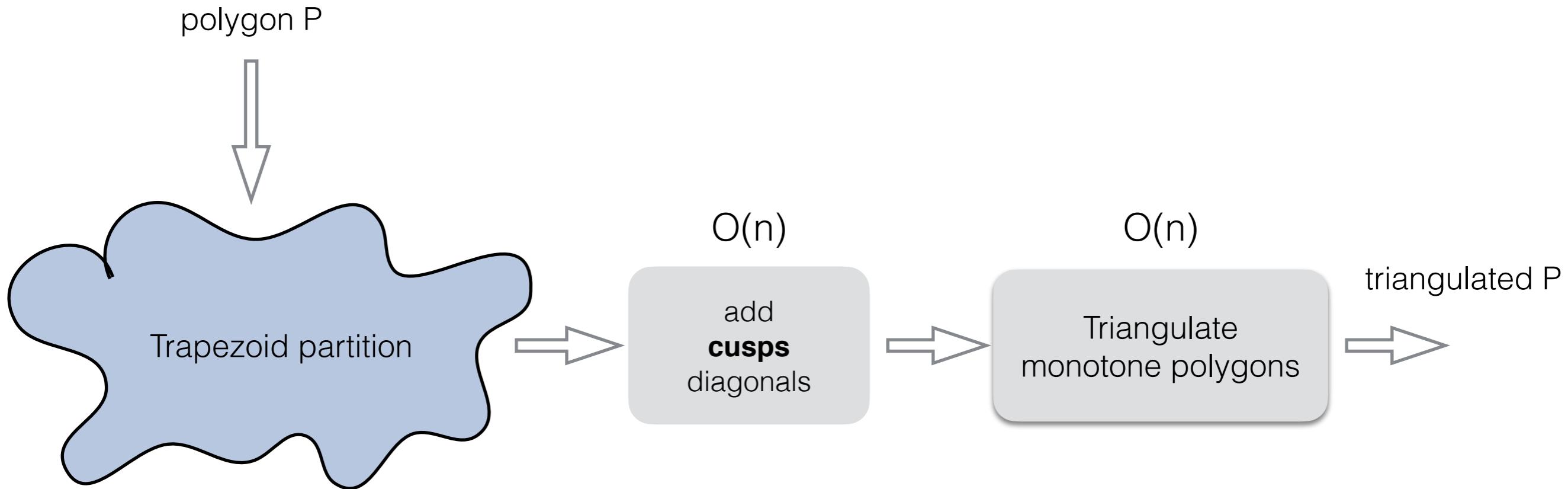
Justification



Justification

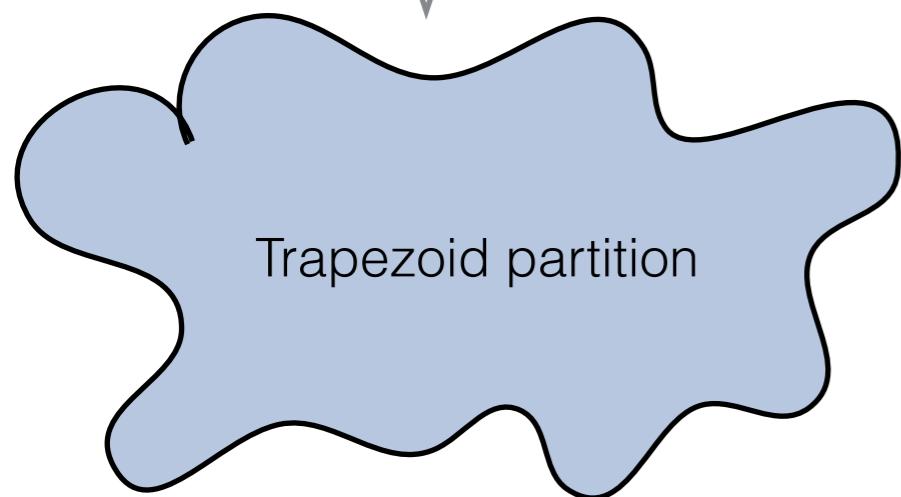


Summary



Given a trapezoid partition of P , we can partition into monotone polygons and triangulate it in $O(n)$ time.

polygon P



Trapezoid partition

In fact, it's possible to partition into monotone mountains



add
all
diagonals

Triangulate
monotone mountains

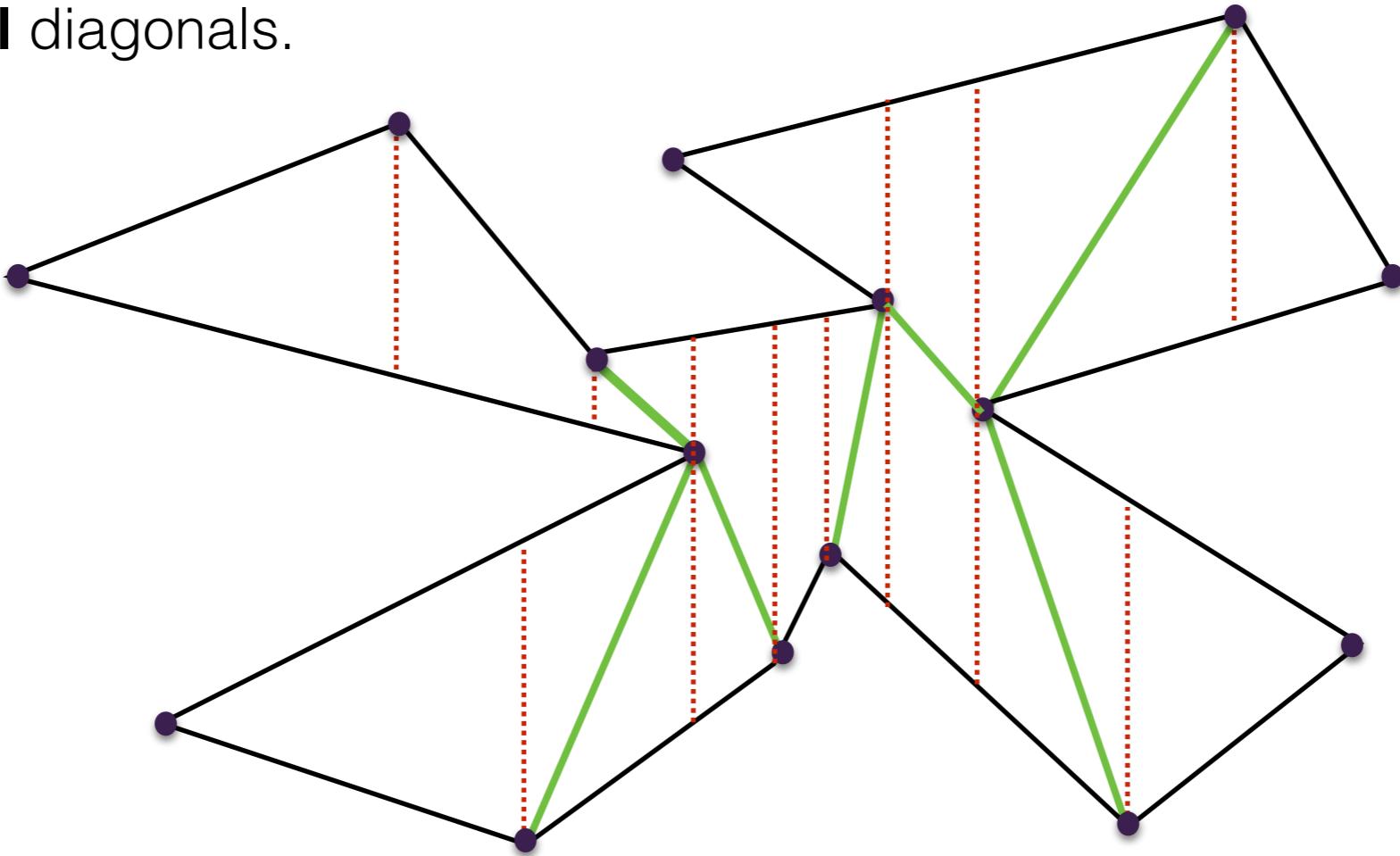
$O(n)$

$O(n)$

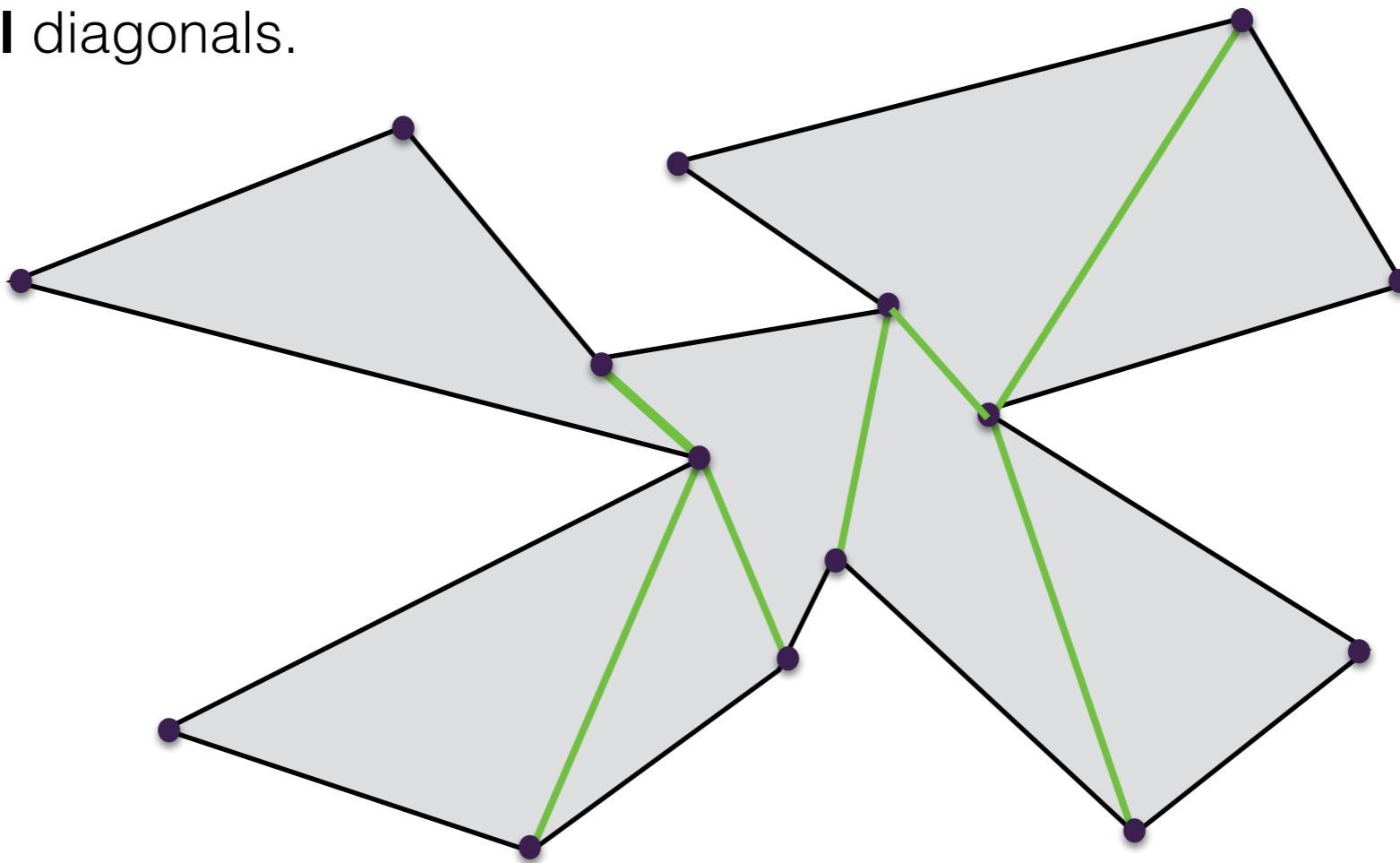
triangulated P

Partition into monotone mountains

1. Compute a trapezoid partition of P
2. Output **all** diagonals.

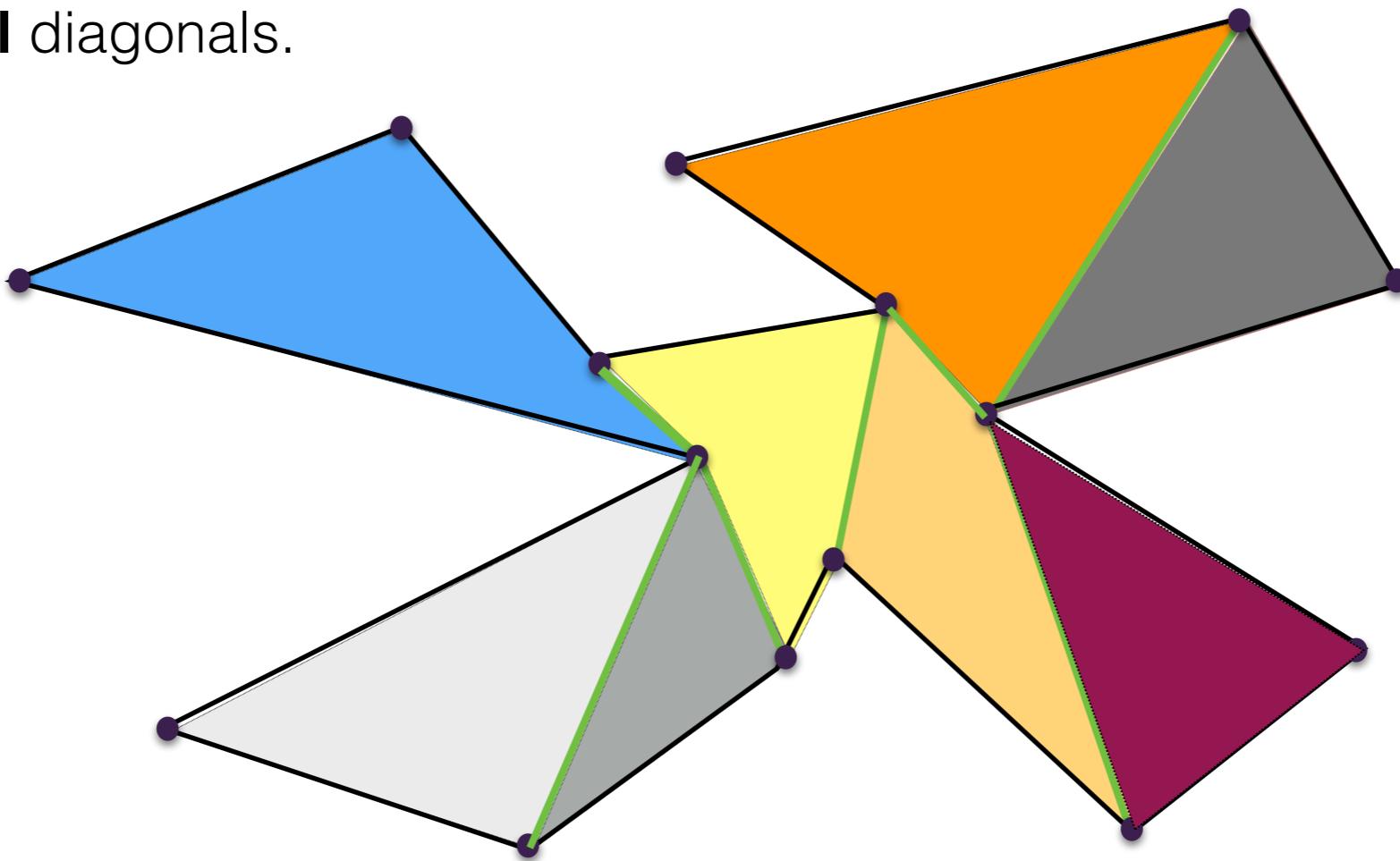


1. Compute a trapezoid partition of P
2. Output **all** diagonals.



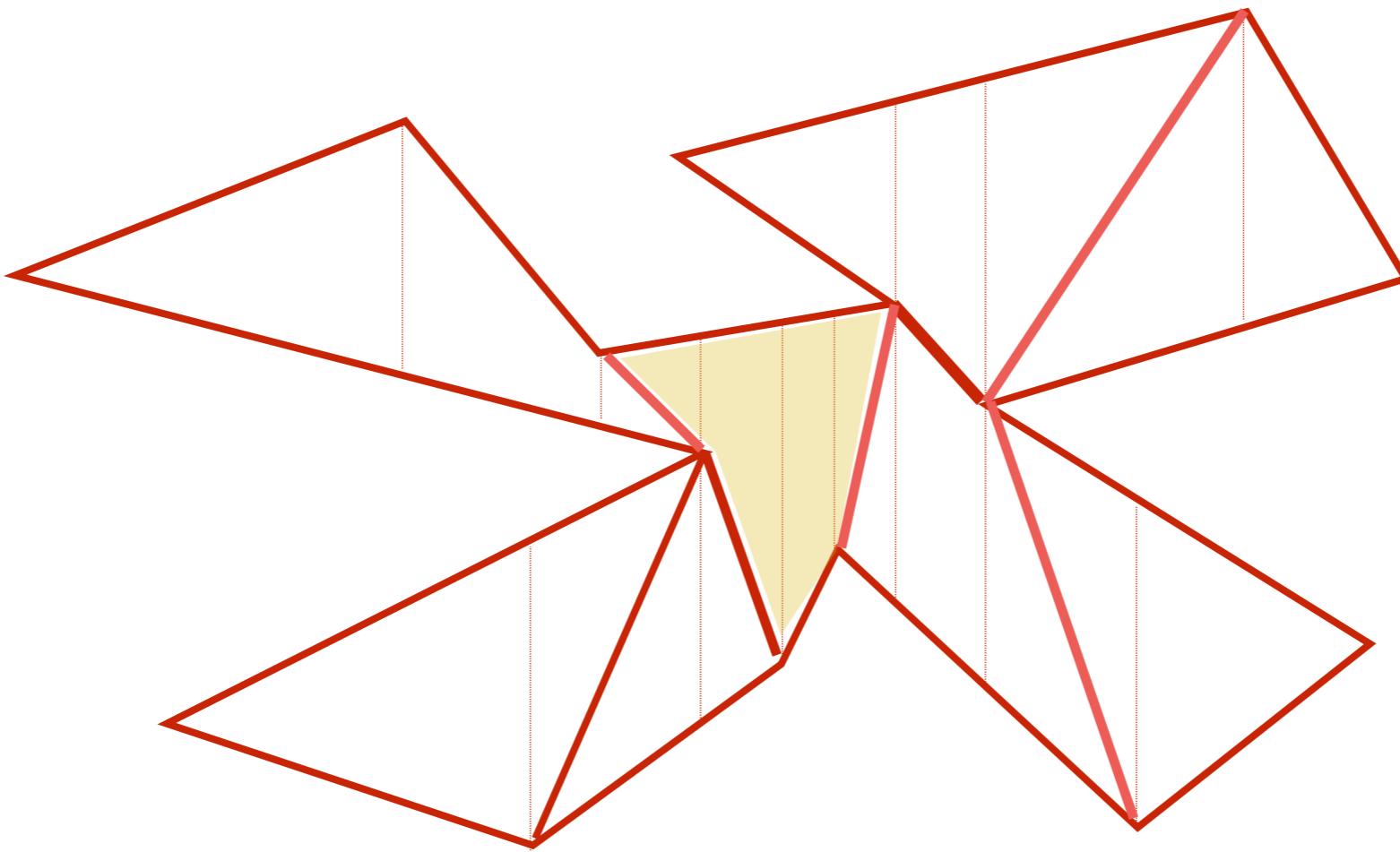
Claim: **All** diagonals partition the polygon into monotone mountains.

1. Compute a trapezoid partition of P
2. Output **all** diagonals.



Claim: **All** diagonals partition the polygon into monotone mountains.

Claim: The diagonals partition the polygon into monotone mountains.

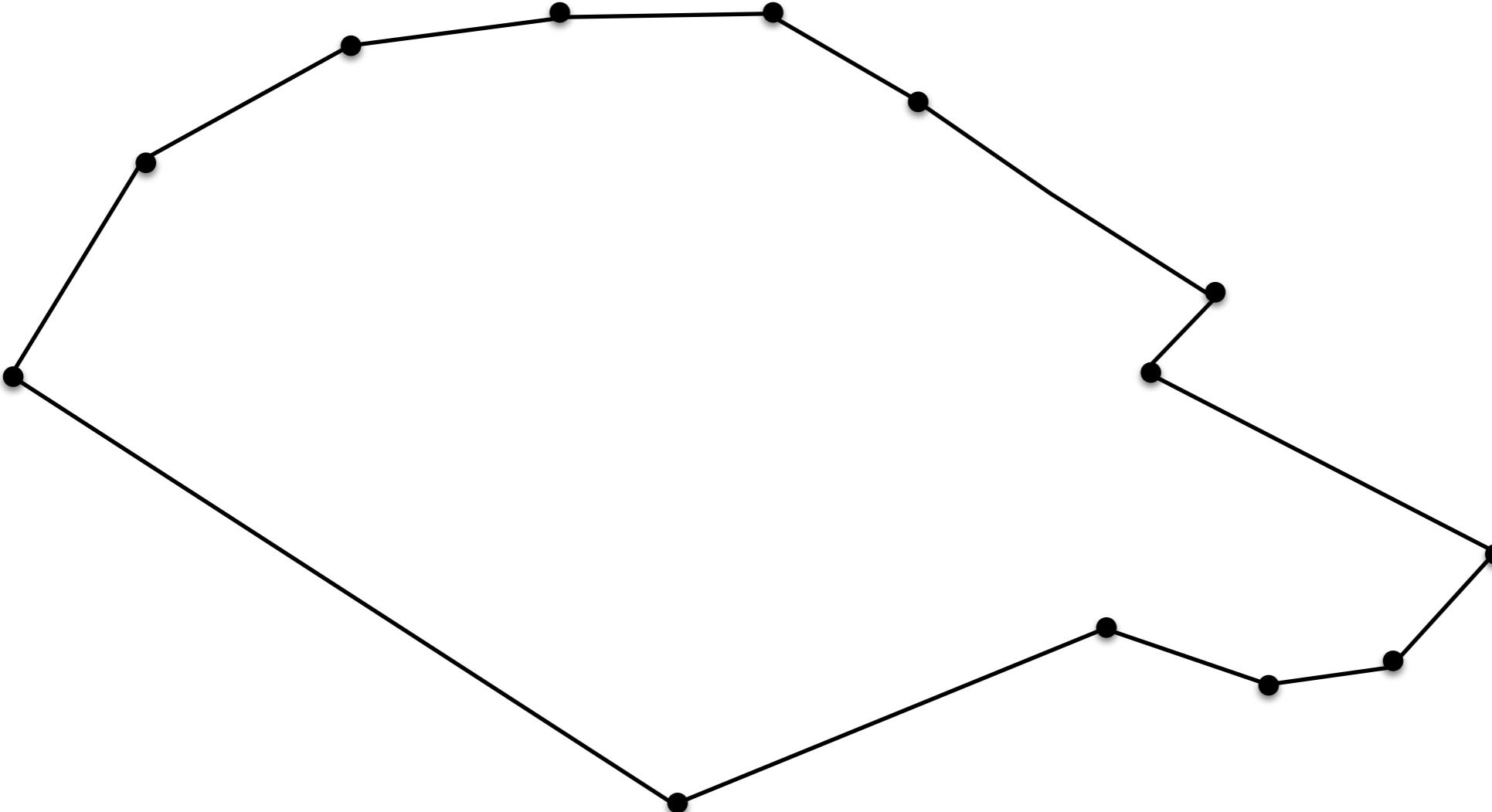


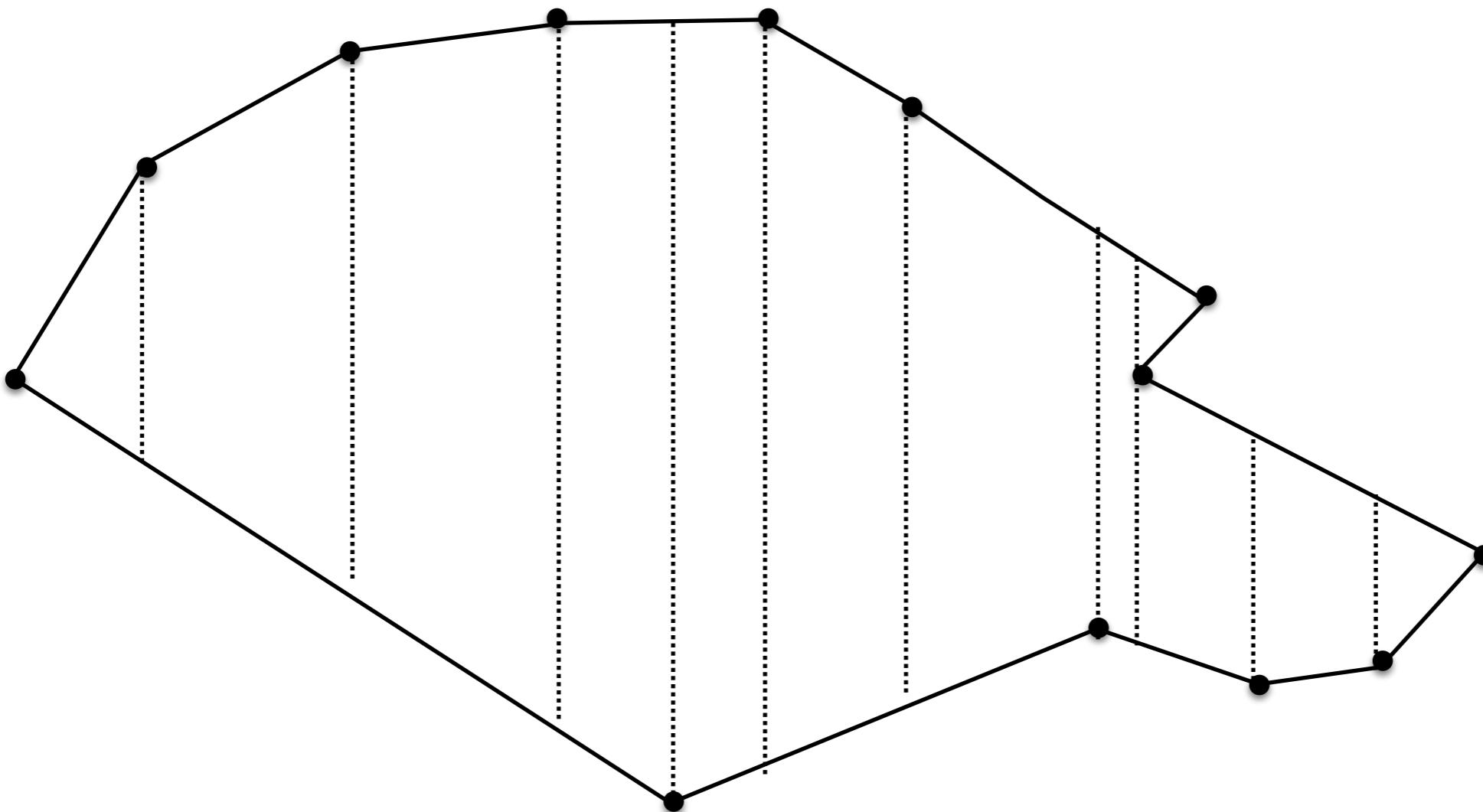
Why?

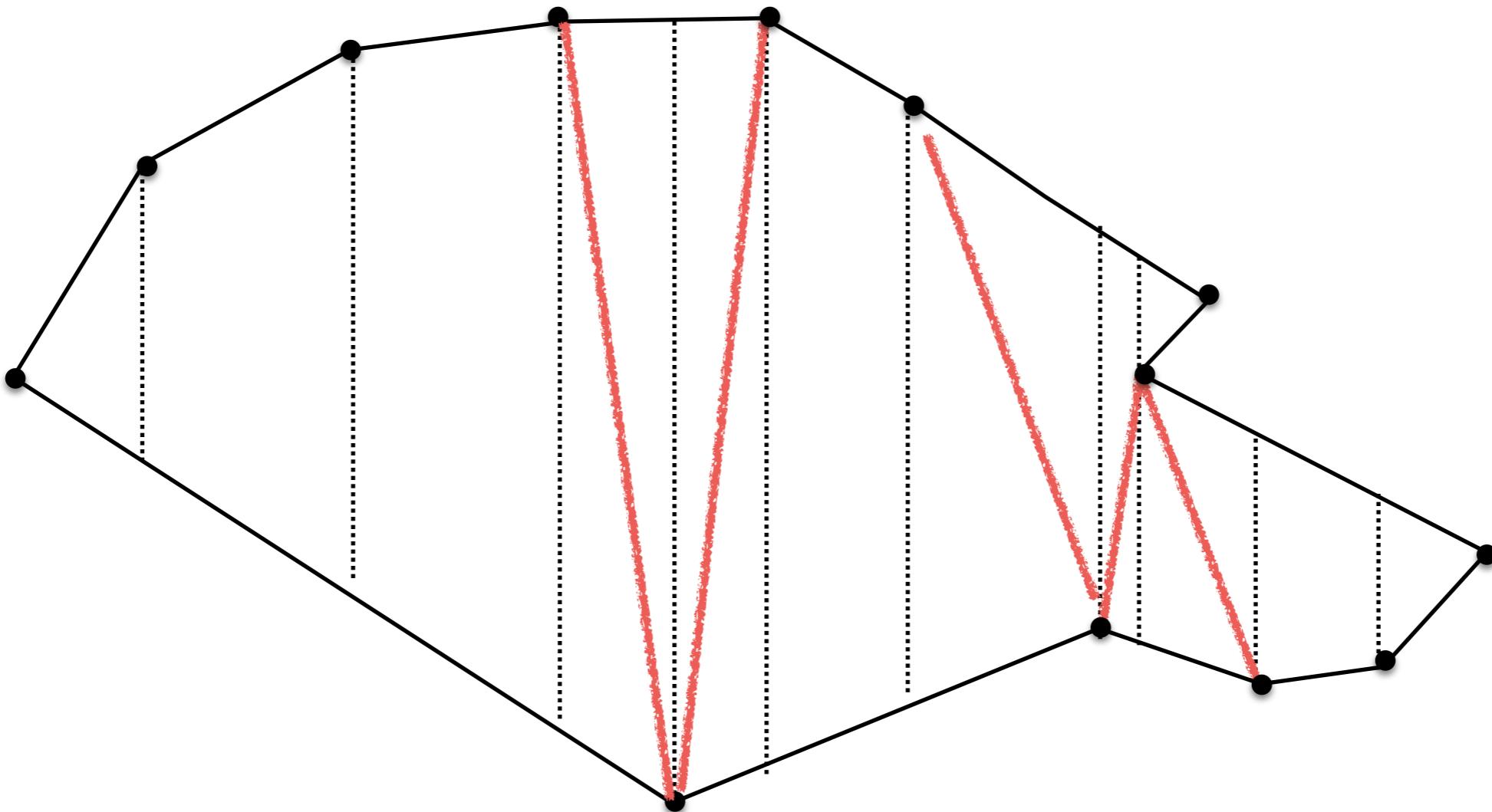
- Each “internal” trapezoid in a polygon must have the polygon vertices either both above (or both below), otherwise they would generate a diagonal => all internal trapezoids have same edge below/above => one edge

Another example

Compute a trapezoid partition and show the diagonals

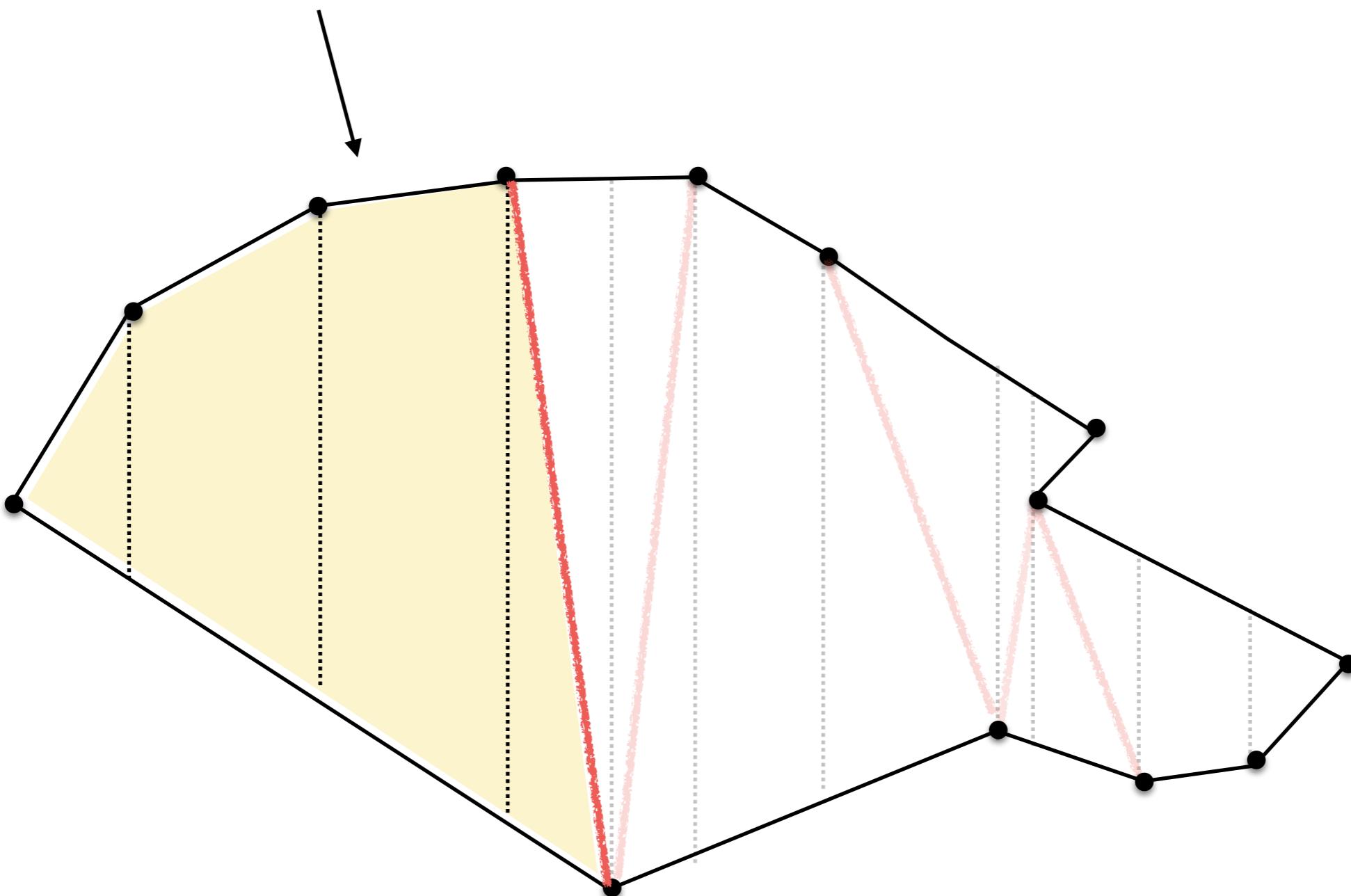






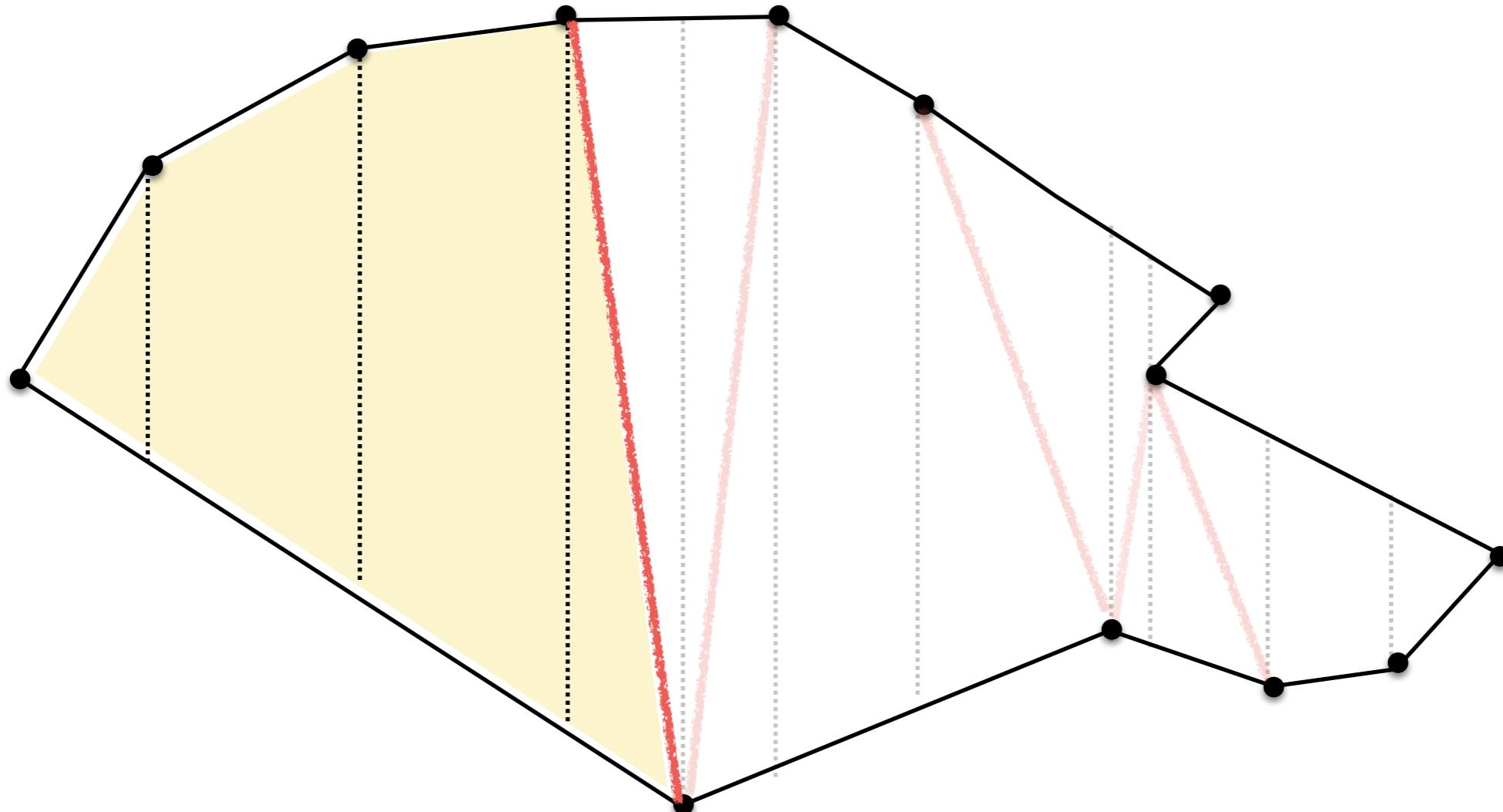
Let's look at one of these pieces

There is no diagonal in here



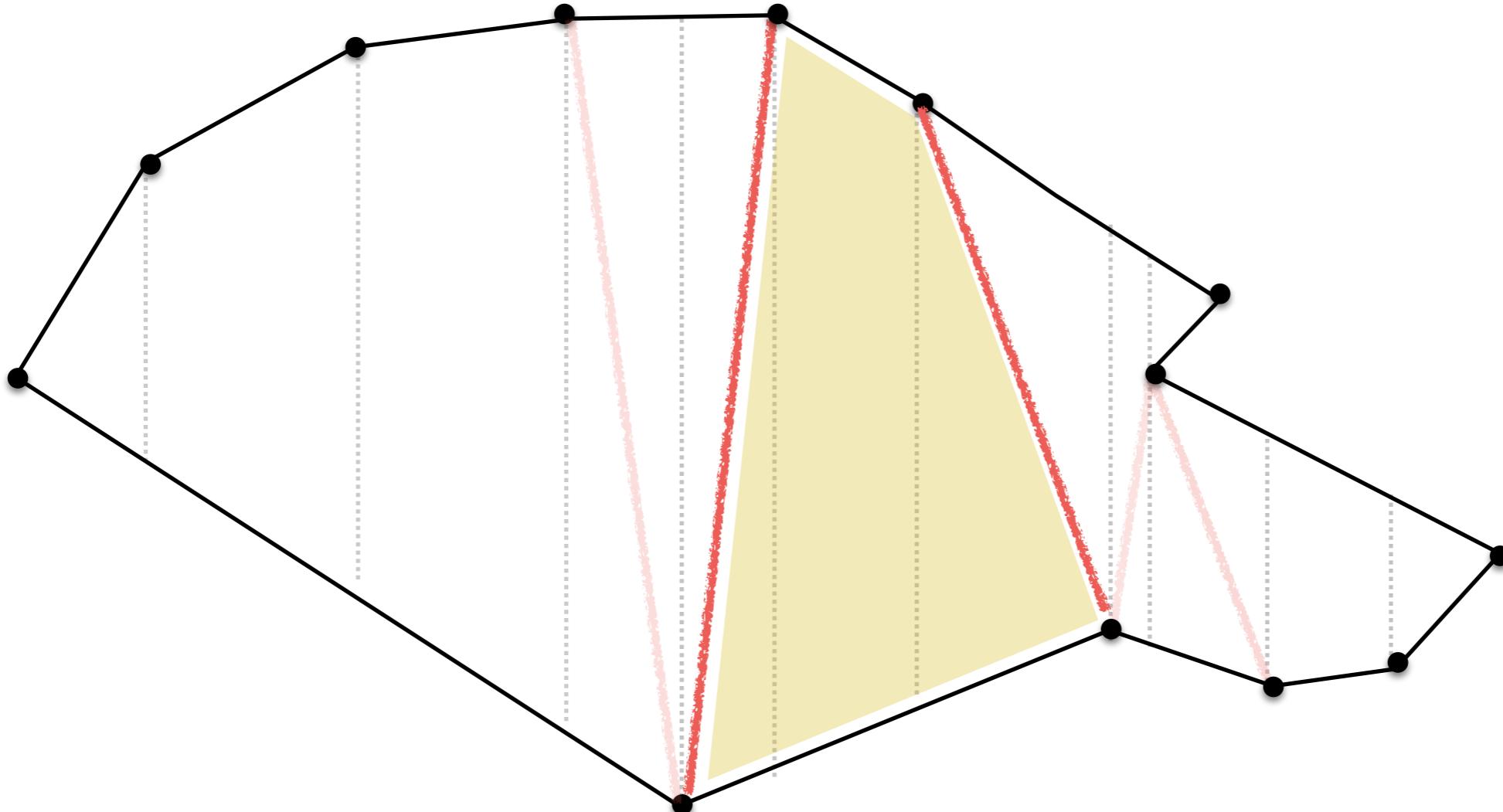
Each trapezoid must have the polygon vertices both above or both below (otherwise they would generate a diagonal)

==> All internal trapezoids have same edge below



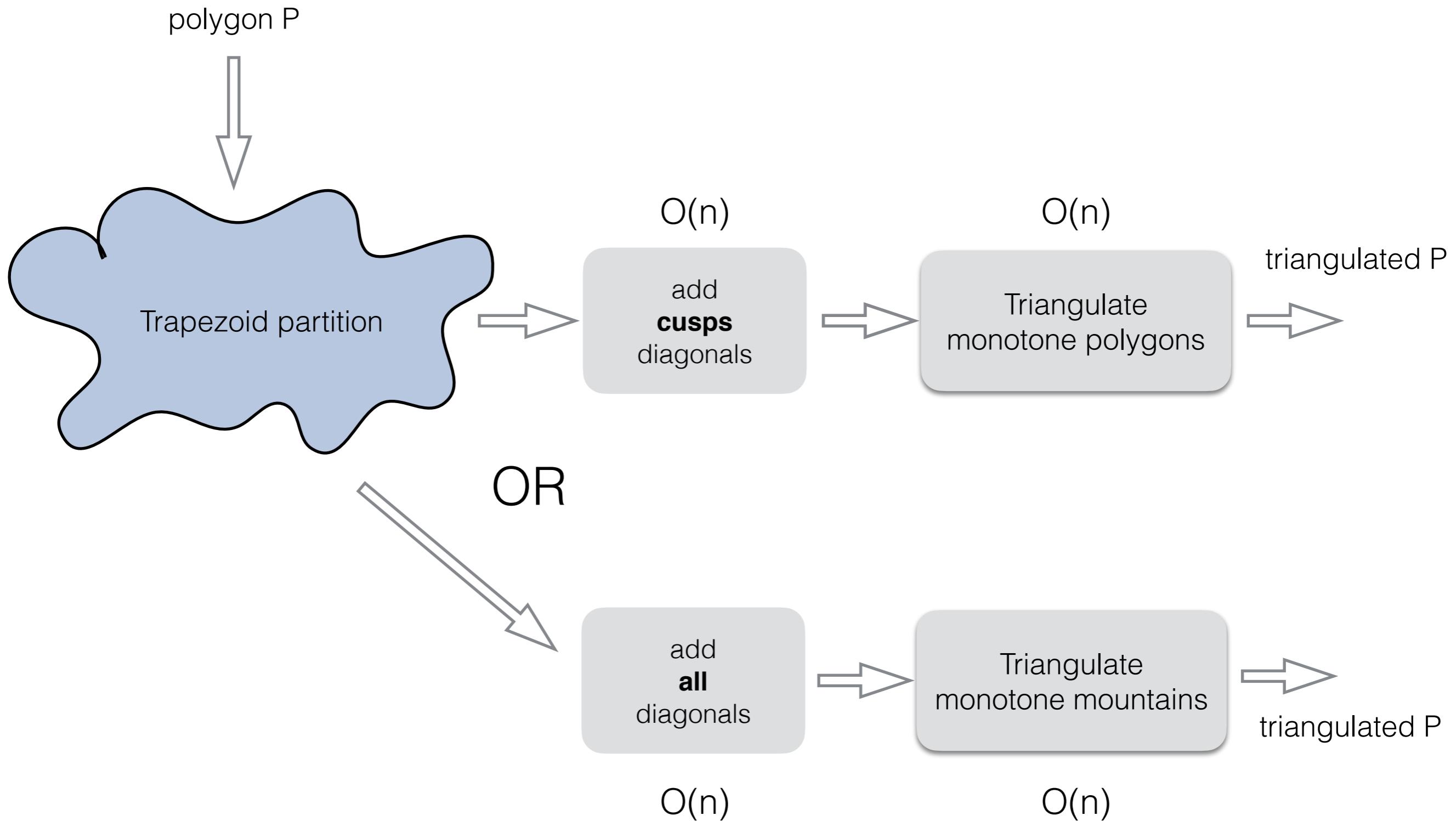
Each trapezoid must have the polygon vertices both above or both below (otherwise they would generate a diagonal)

==> All internal trapezoids have same edge below

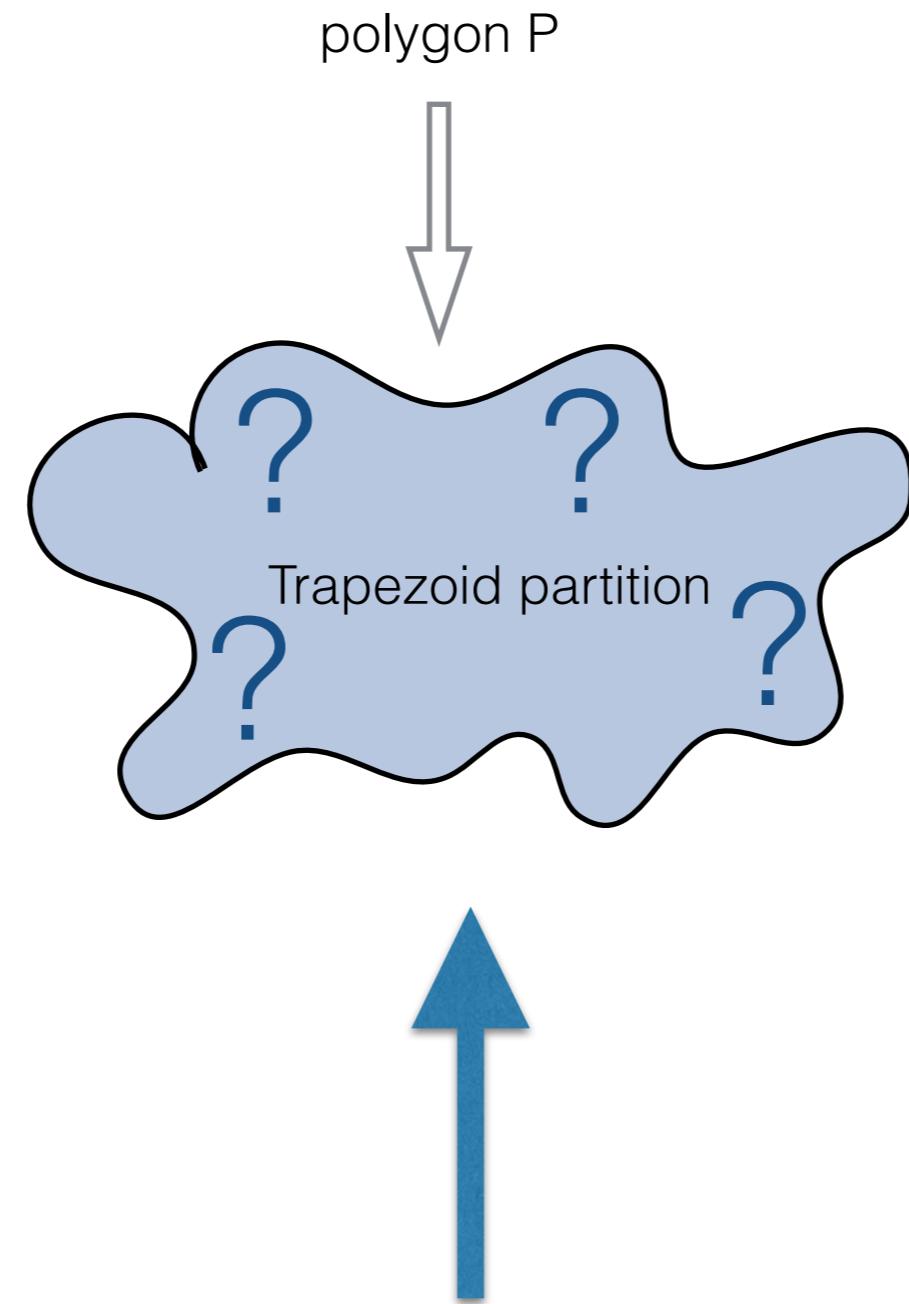


Summary

Given a trapezoid partition of P , we can triangulate it in $O(n)$ time.

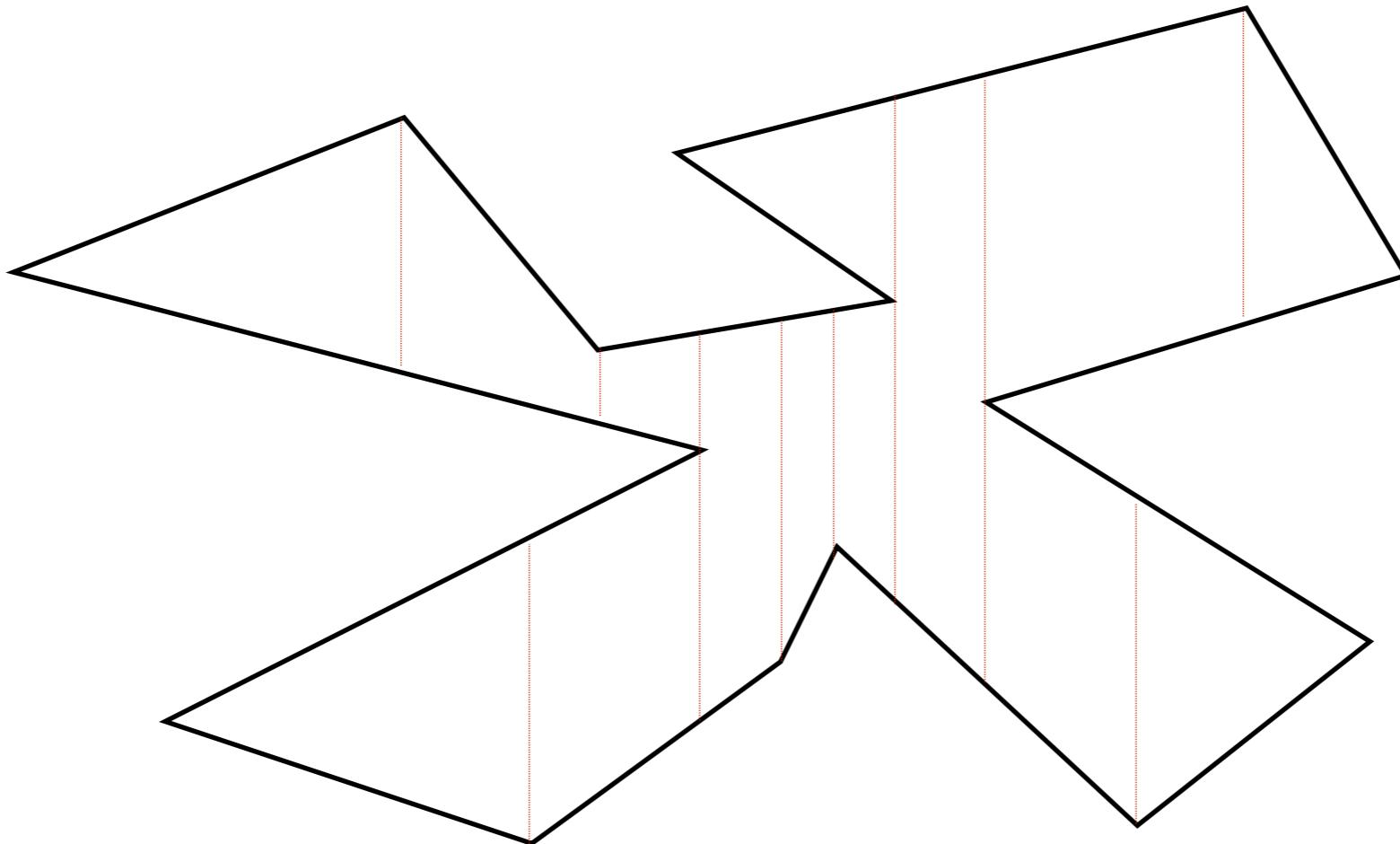


The missing piece



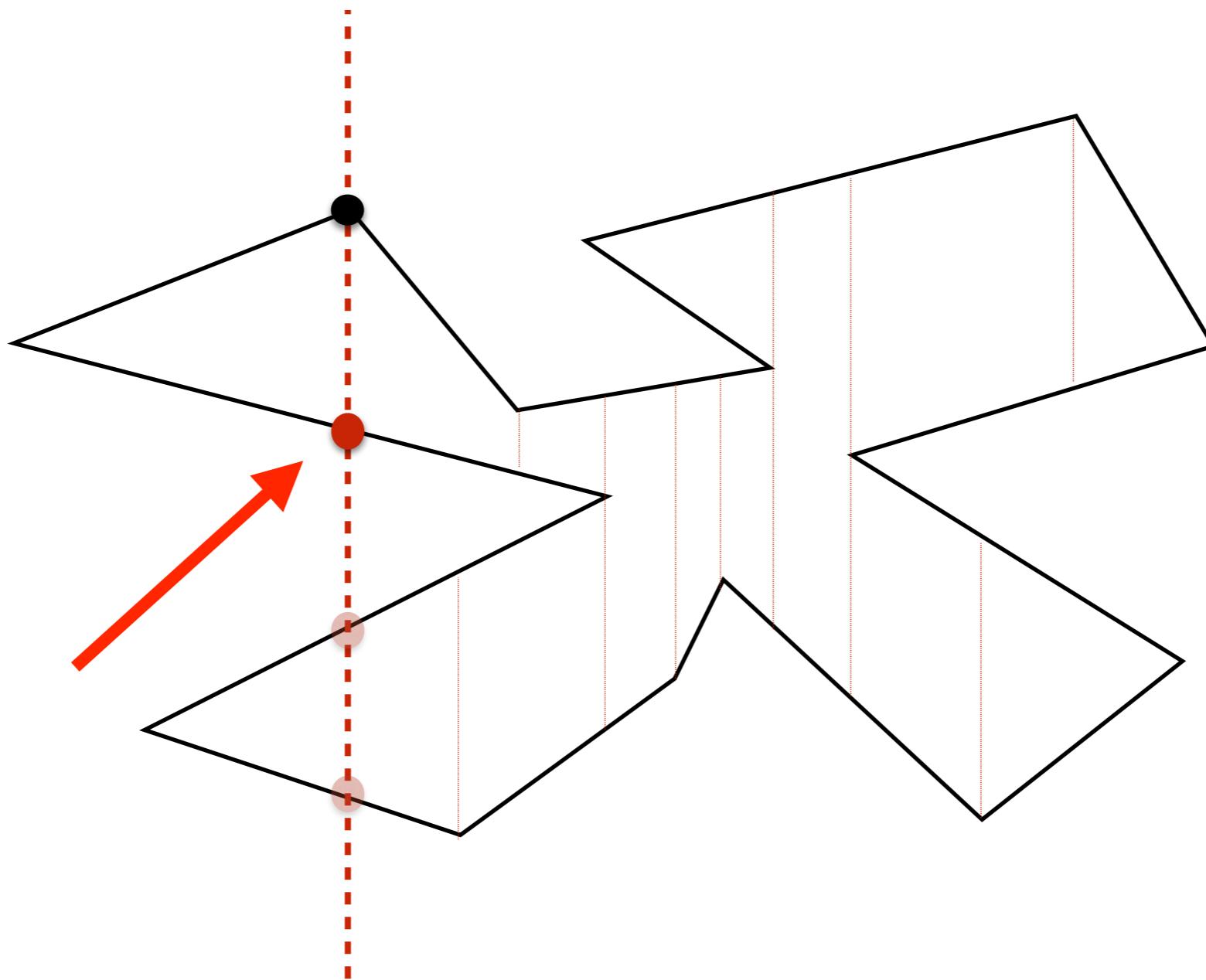
How to compute the trapezoid partition?

Computing the trapezoid partition



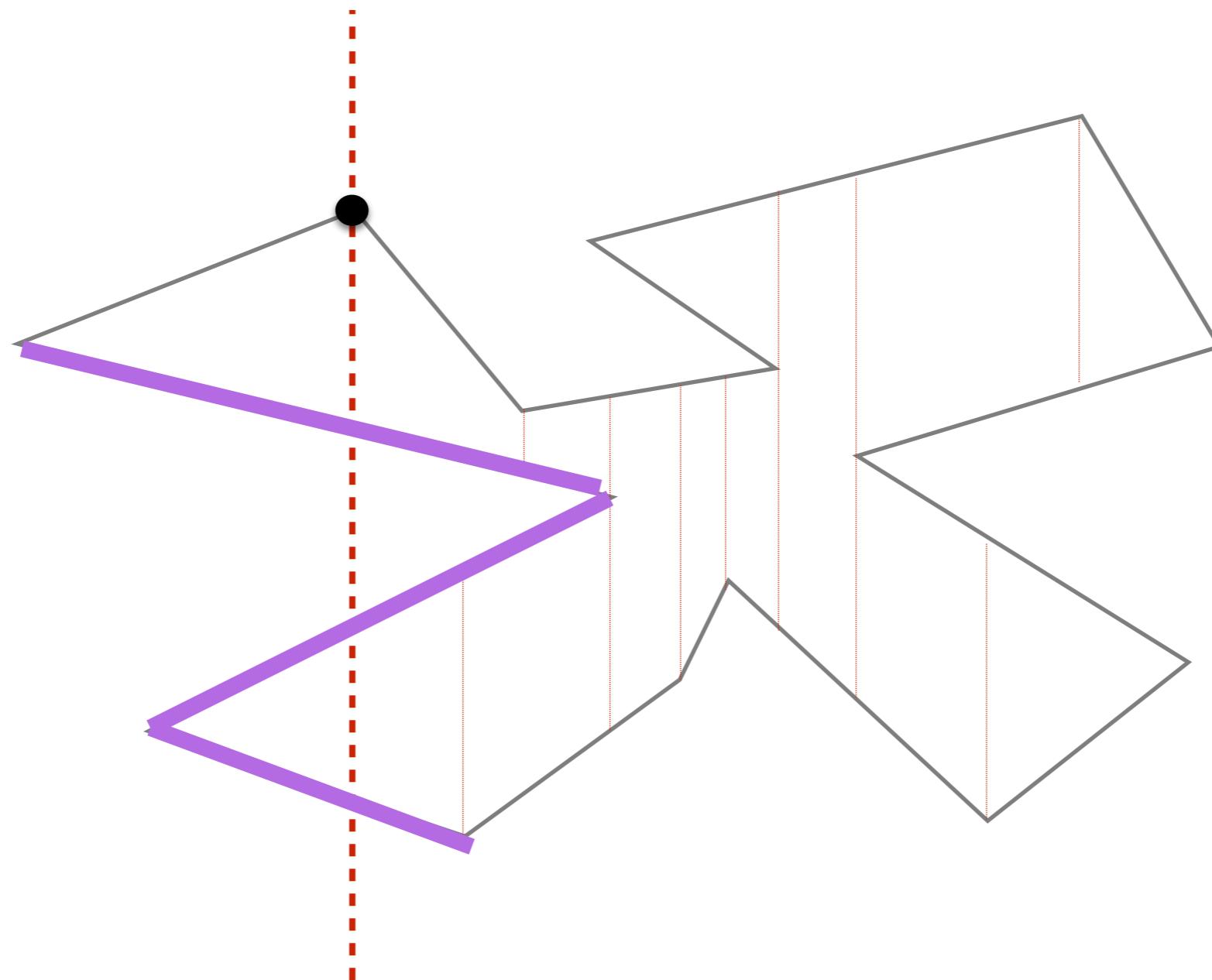
Naive algorithm?

Computing the trapezoid partition

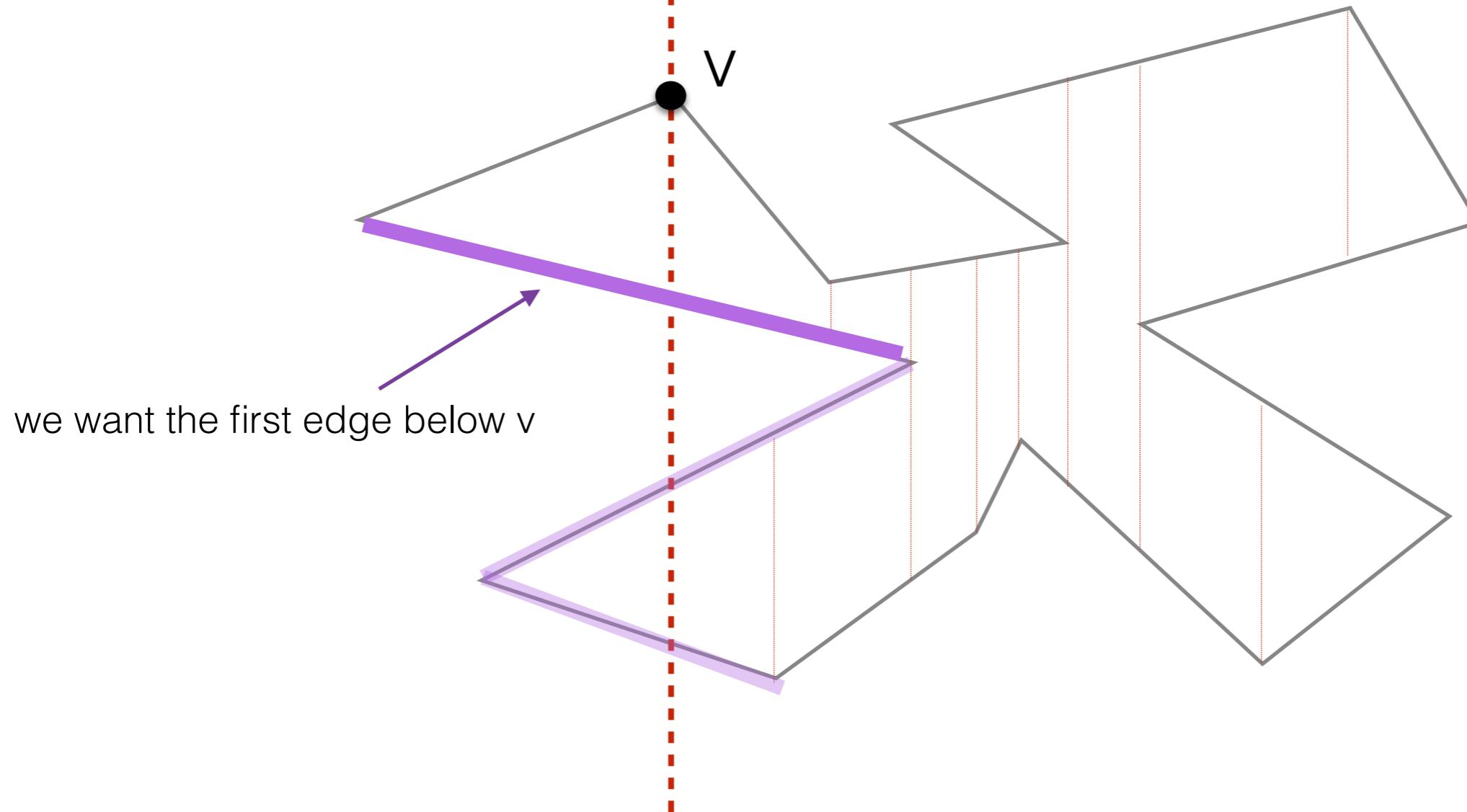


Naive algorithm: for each vertex, shoot a vertical ray through it and compute the “closest” edge that intersects it

What if all the edges that intersect the ray would sit in a nice structure in y-order?

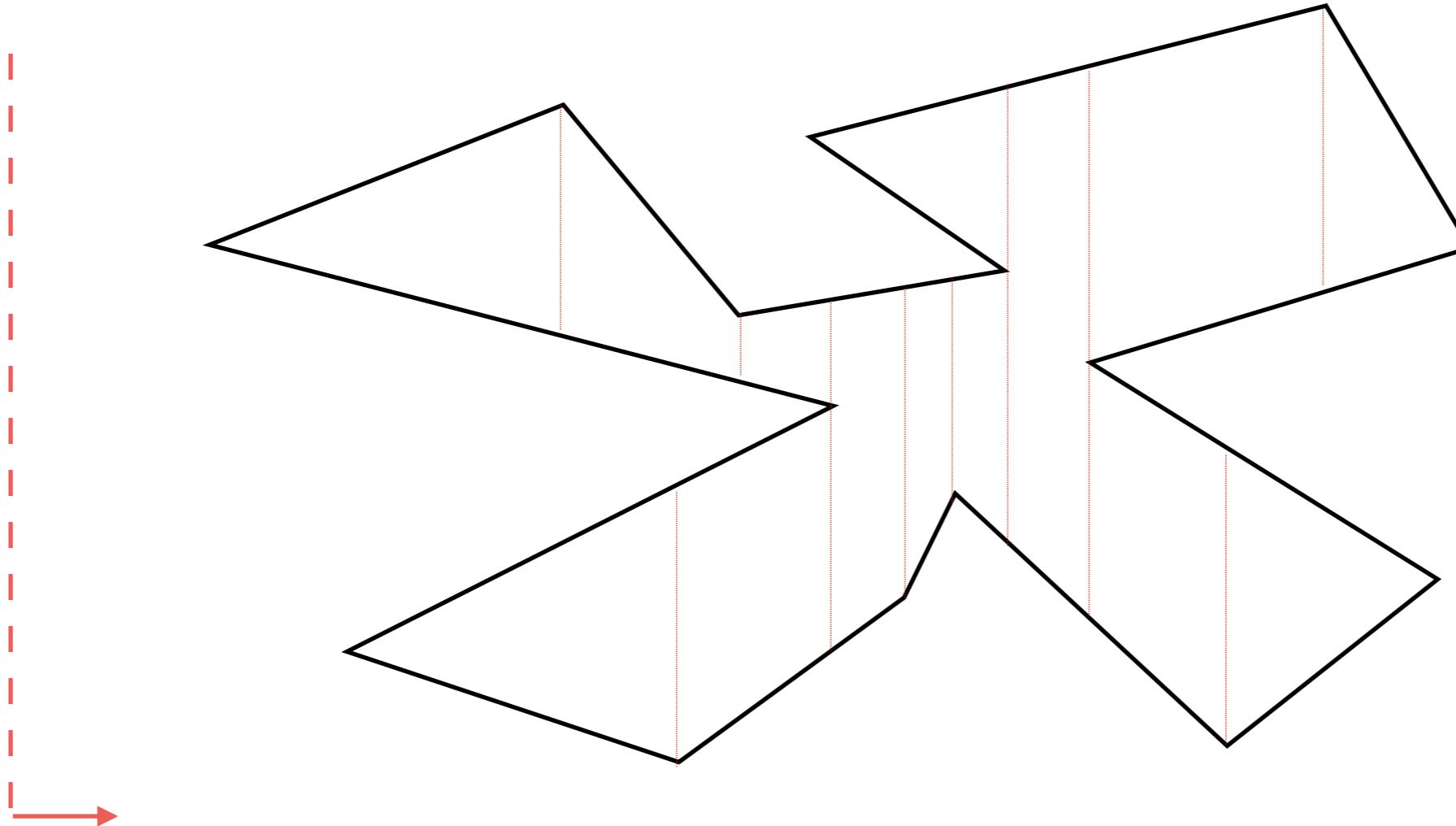


What if all the edges that intersect the ray would sit in a nice structure in y-order?



Computing the trapezoid partition in $O(n \lg n)$

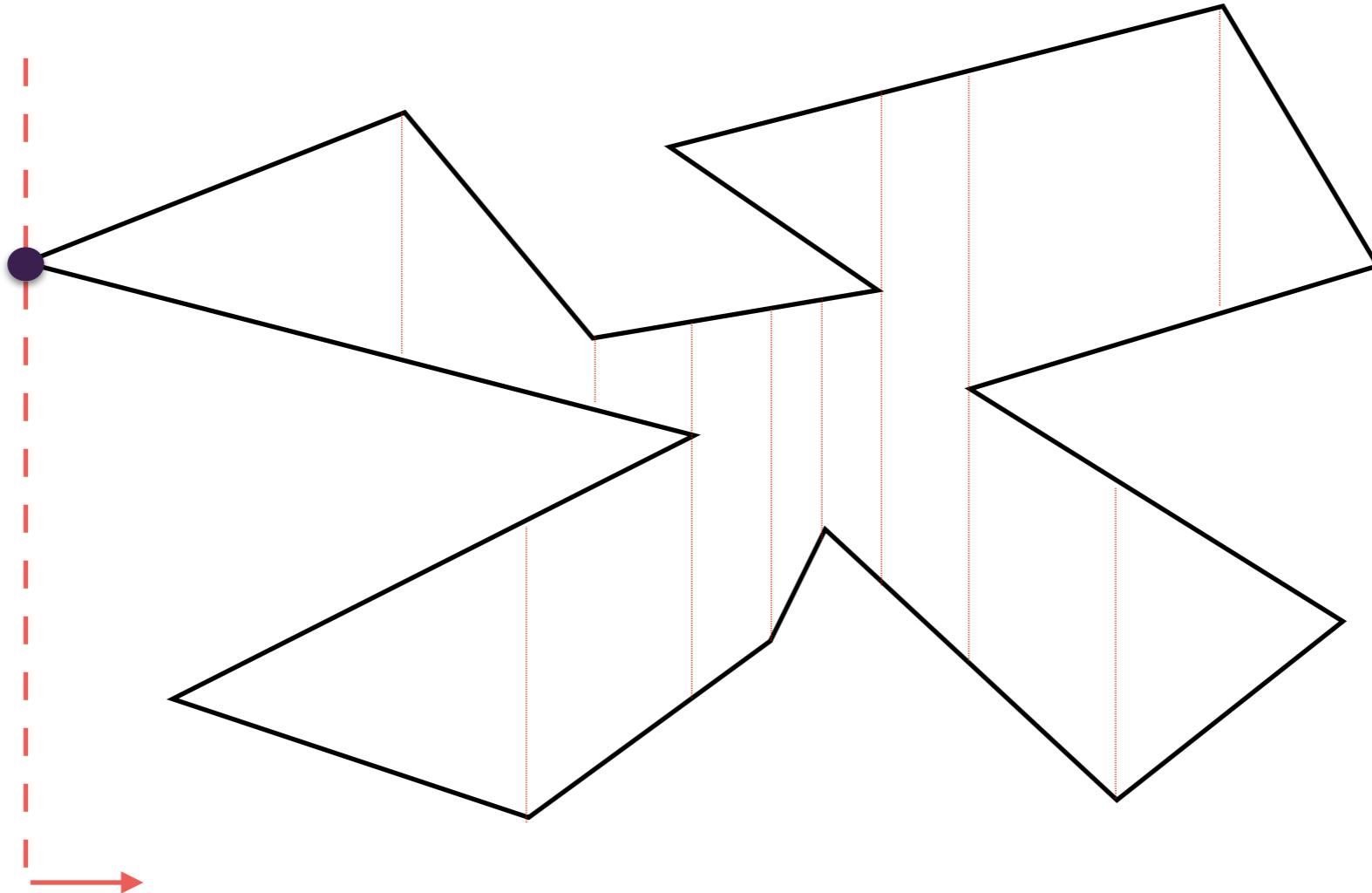
- Line sweep



events = polygon vertices

Computing the trapezoid partition in $O(n \lg n)$

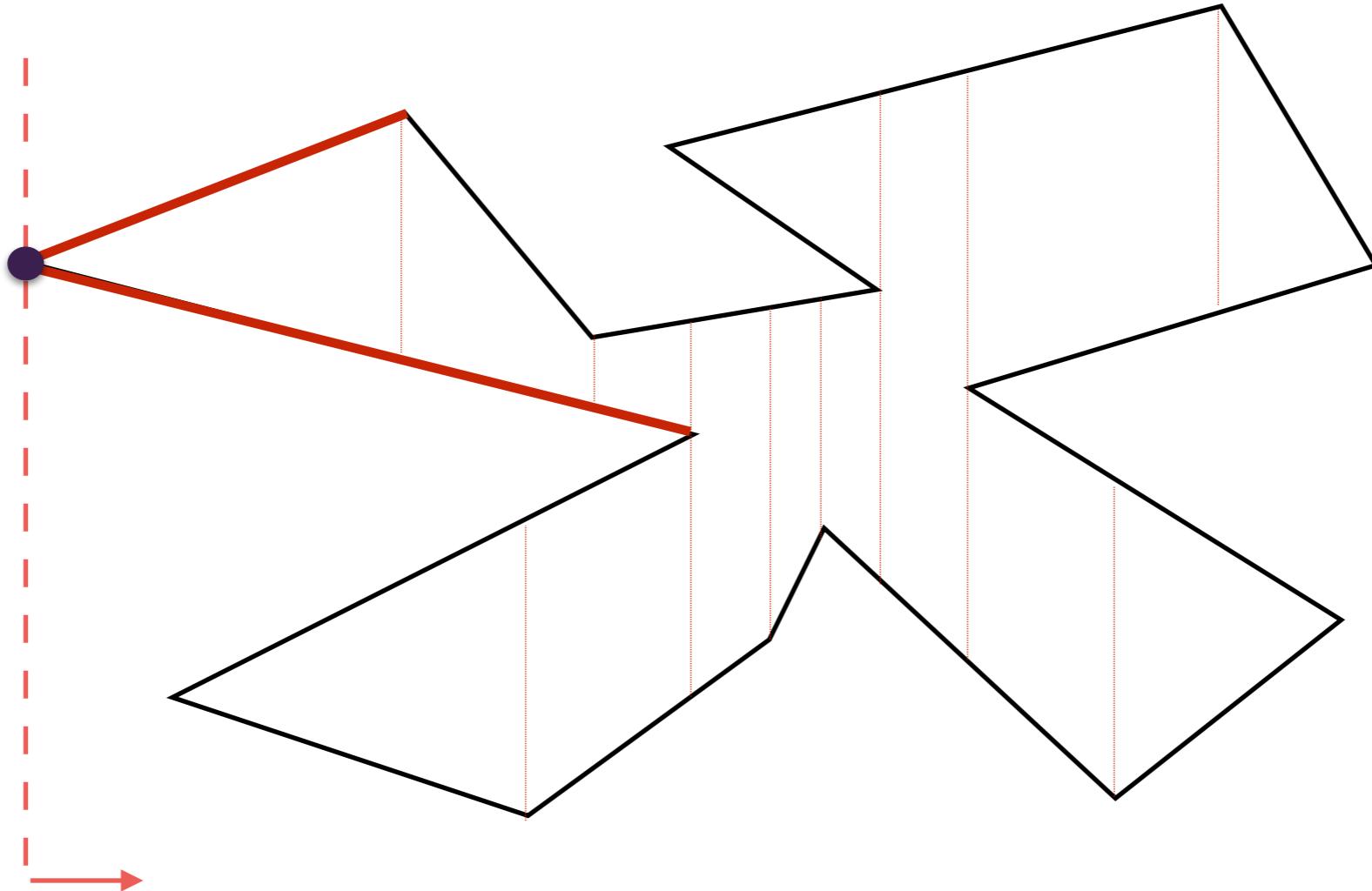
- Line sweep



events = polygon vertices

Computing the trapezoid partition in $O(n \lg n)$

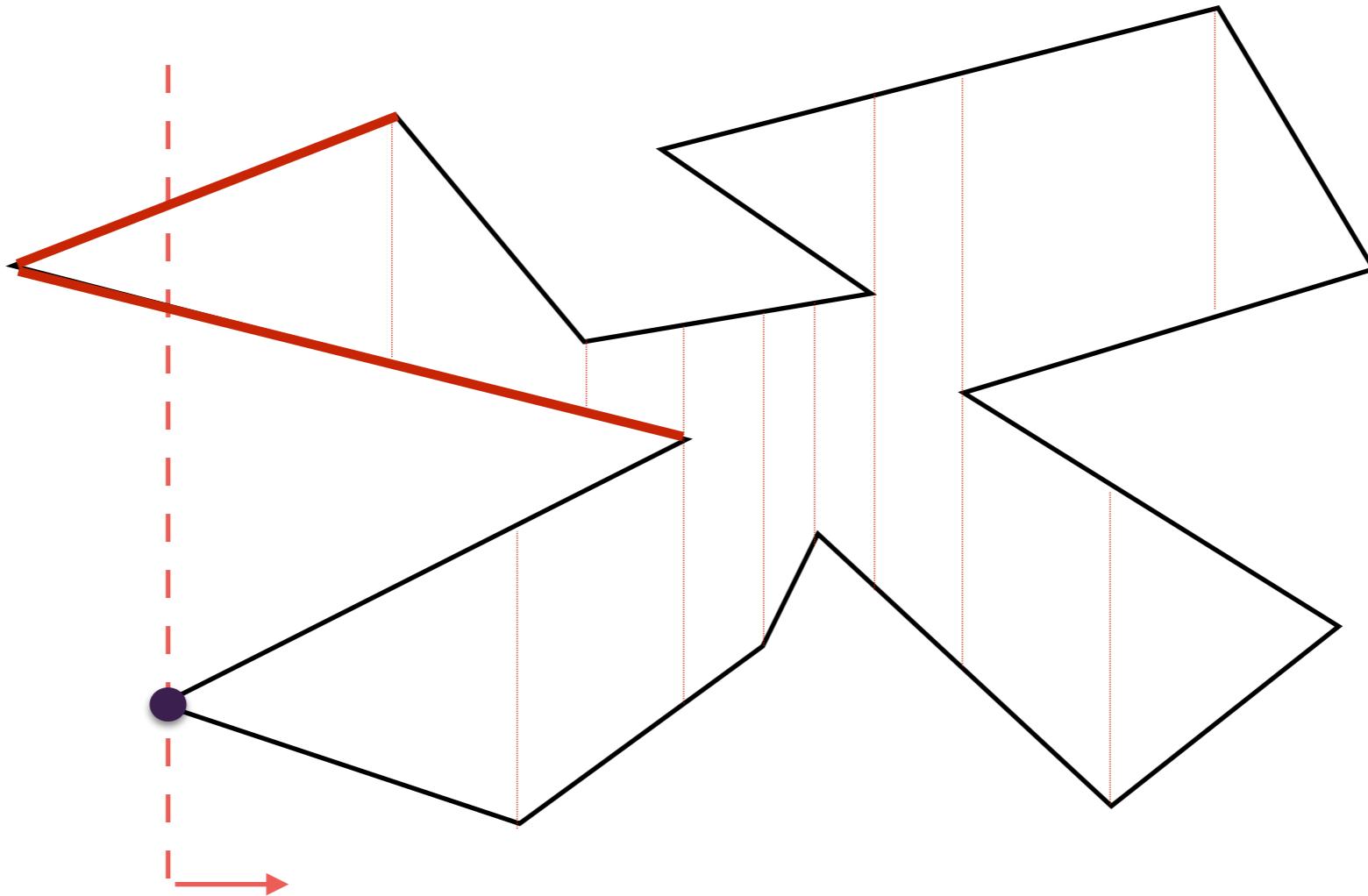
- Line sweep



events = polygon vertices

Computing the trapezoid partition in $O(n \lg n)$

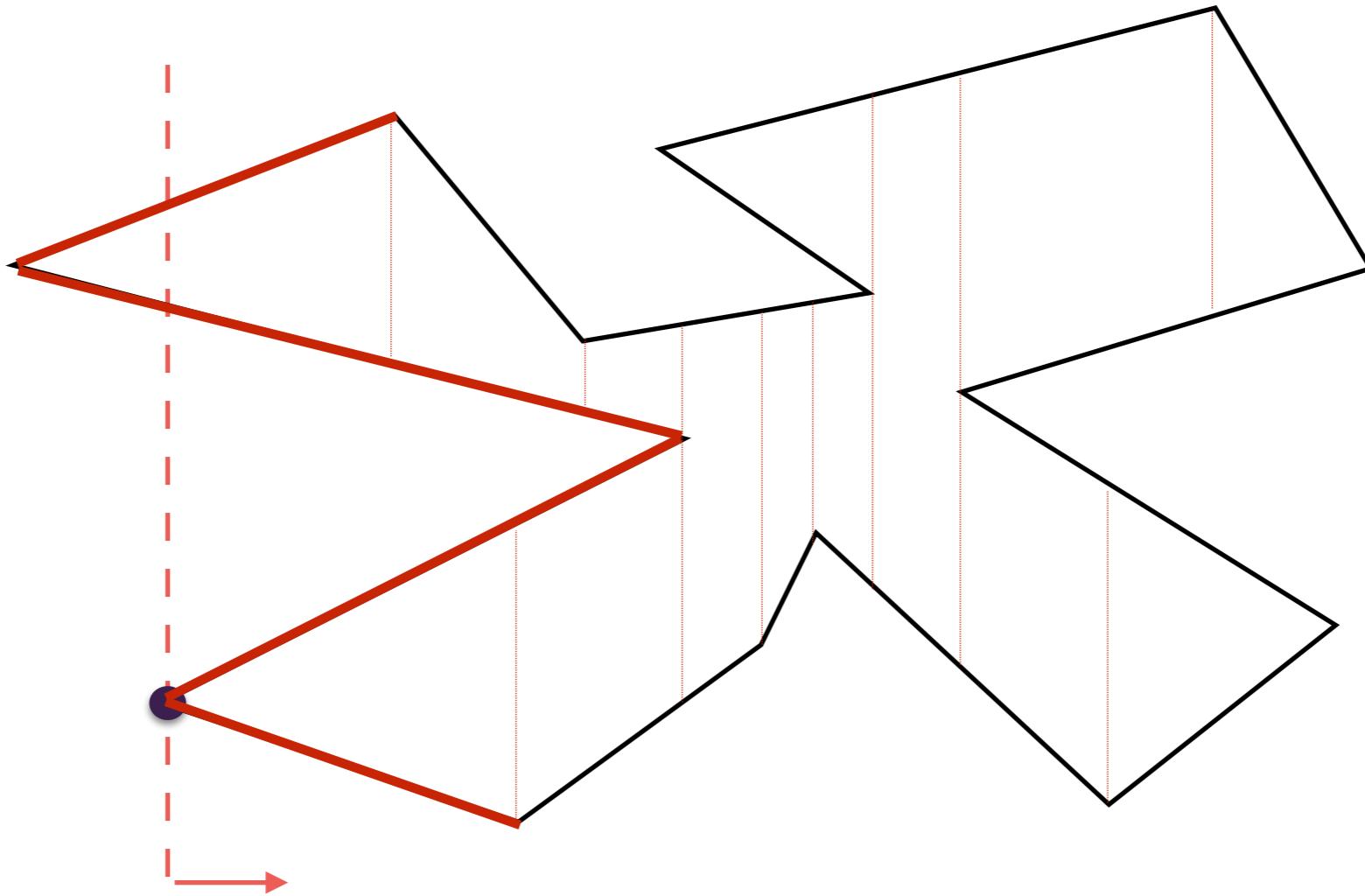
- Line sweep



events = polygon vertices

Computing the trapezoid partition in $O(n \lg n)$

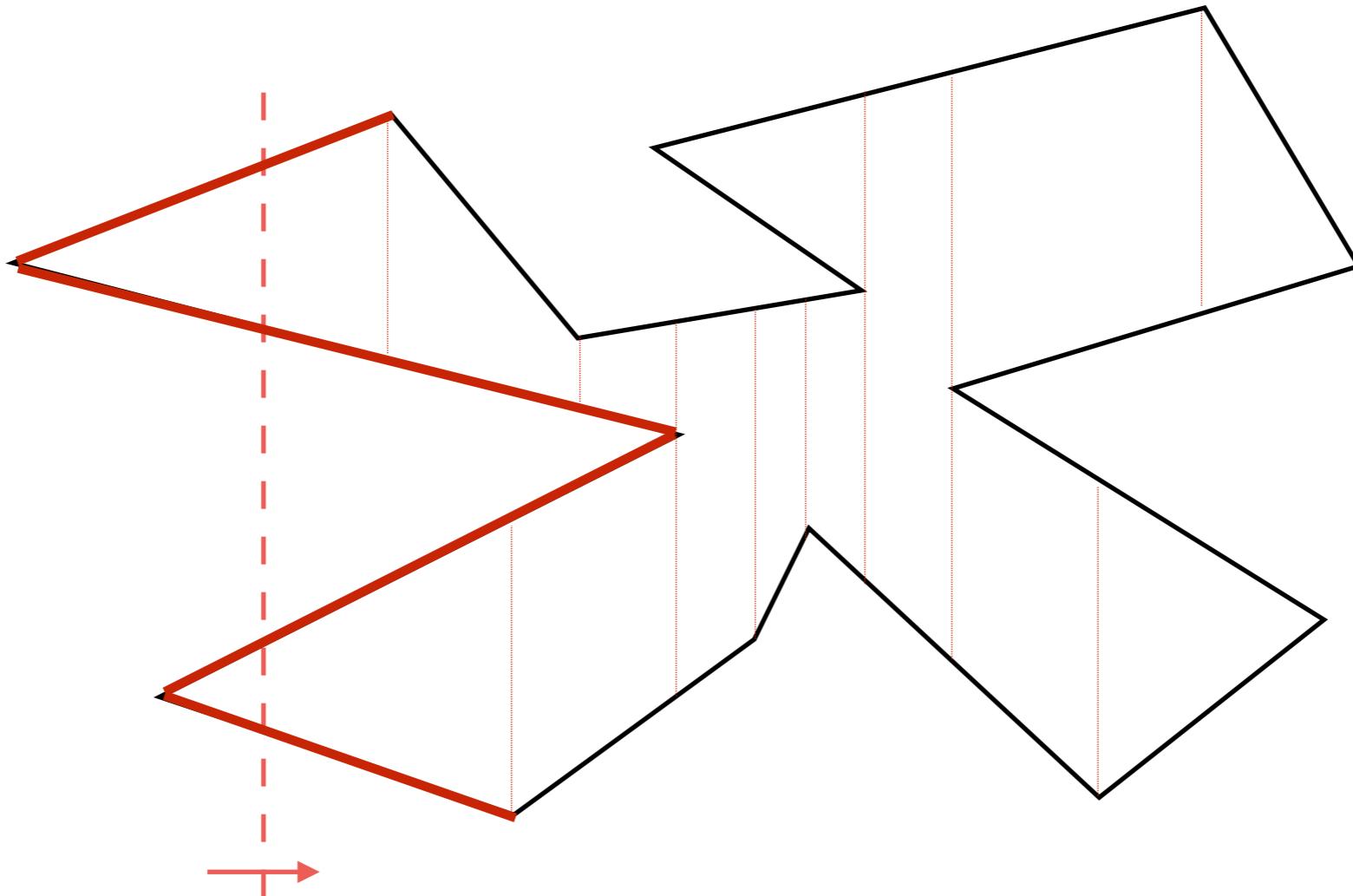
- Line sweep



events = polygon vertices

Computing the trapezoid partition in $O(n \lg n)$

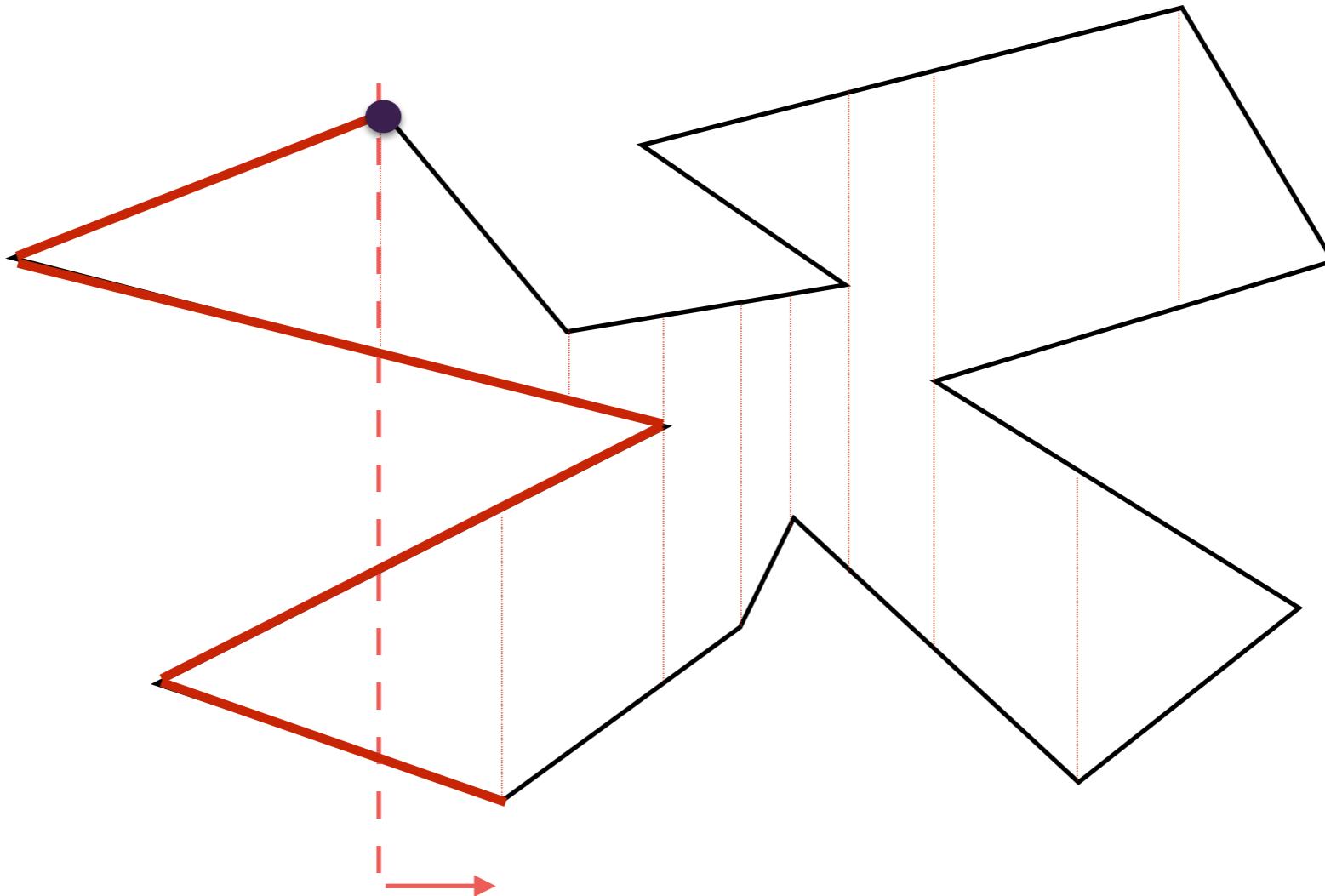
- Line sweep



events = polygon vertices

Computing the trapezoid partition in $O(n \lg n)$

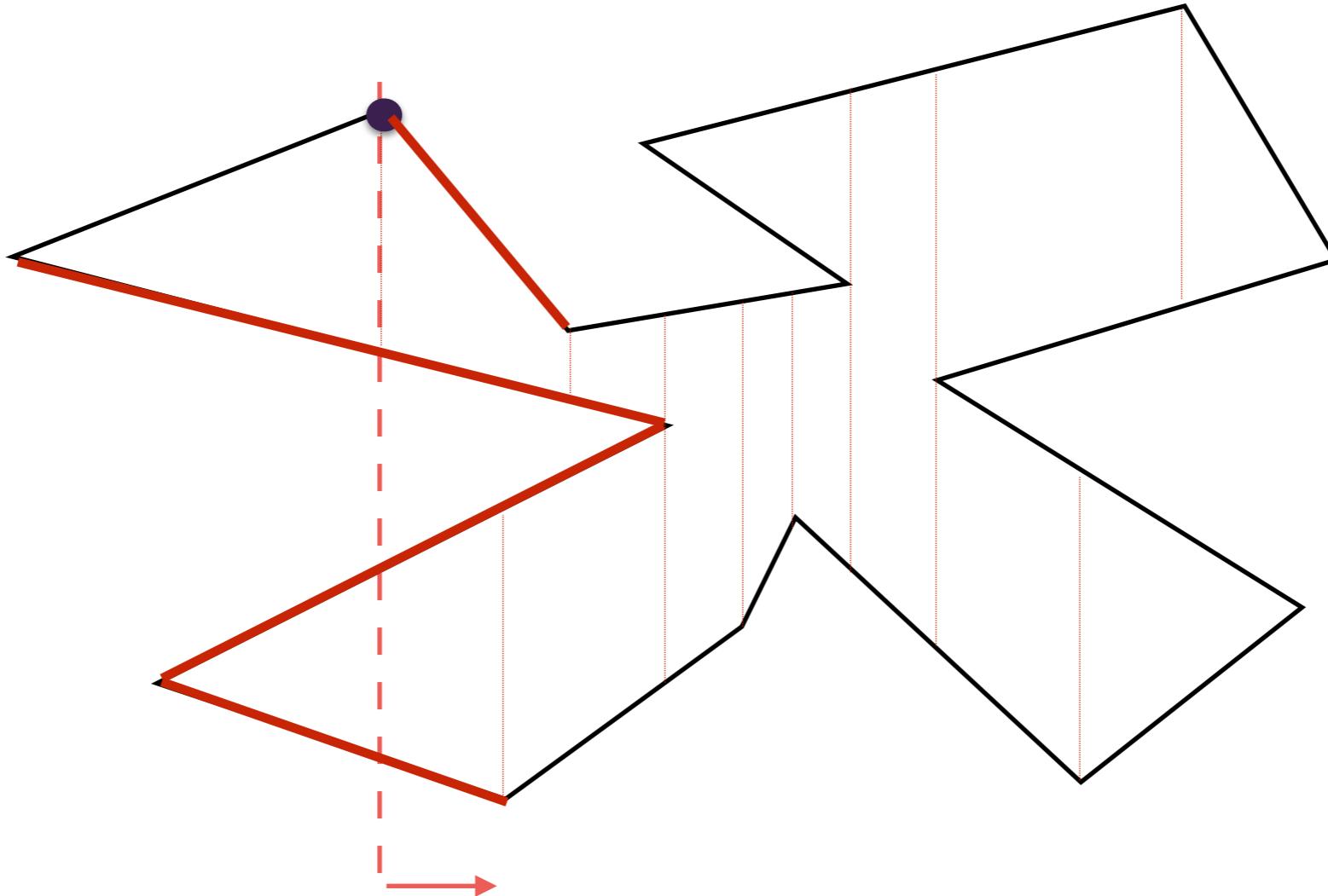
- Line sweep



events = polygon vertices

Computing the trapezoid partition in $O(n \lg n)$

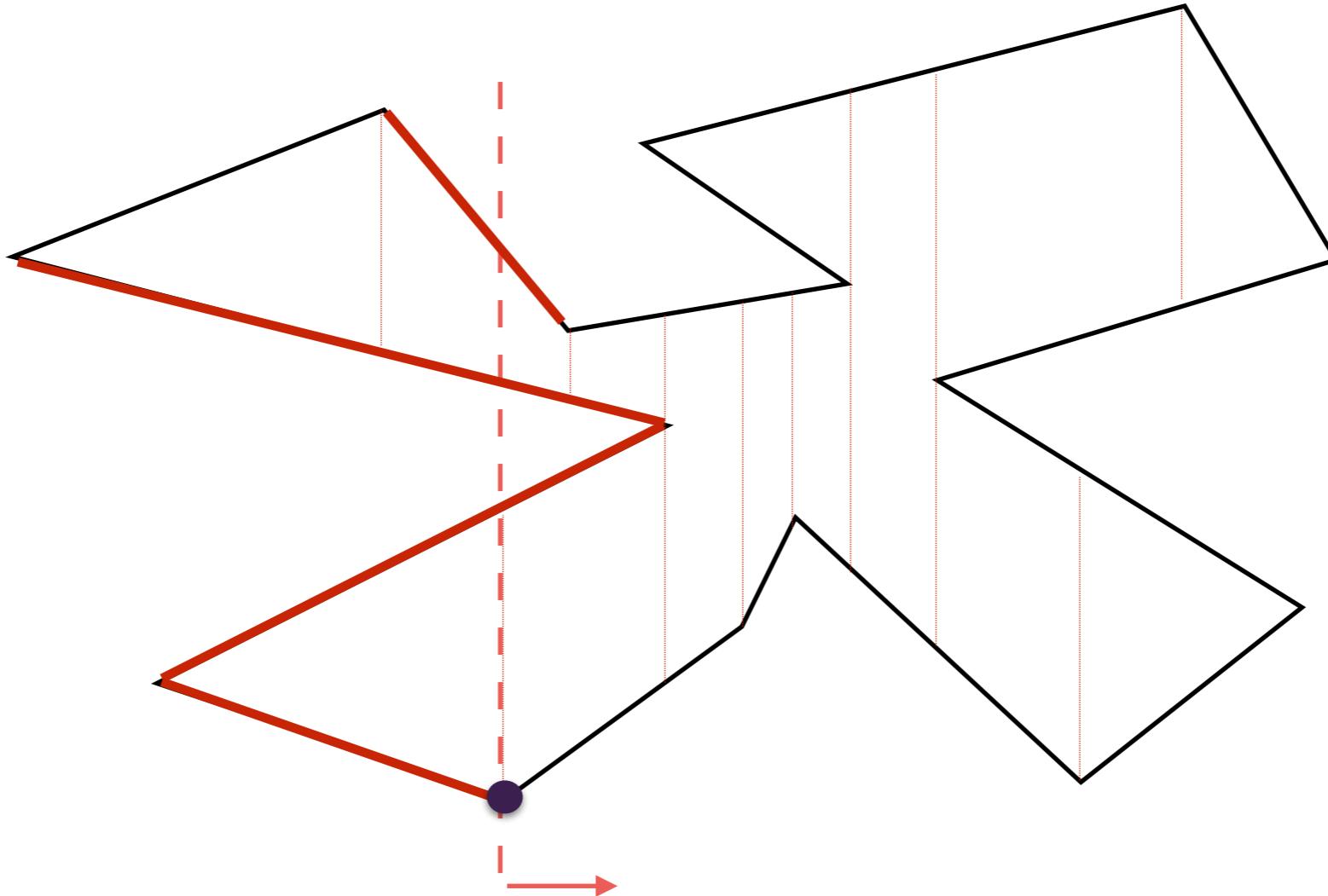
- Line sweep



events = polygon vertices

Computing the trapezoid partition in $O(n \lg n)$

- Line sweep

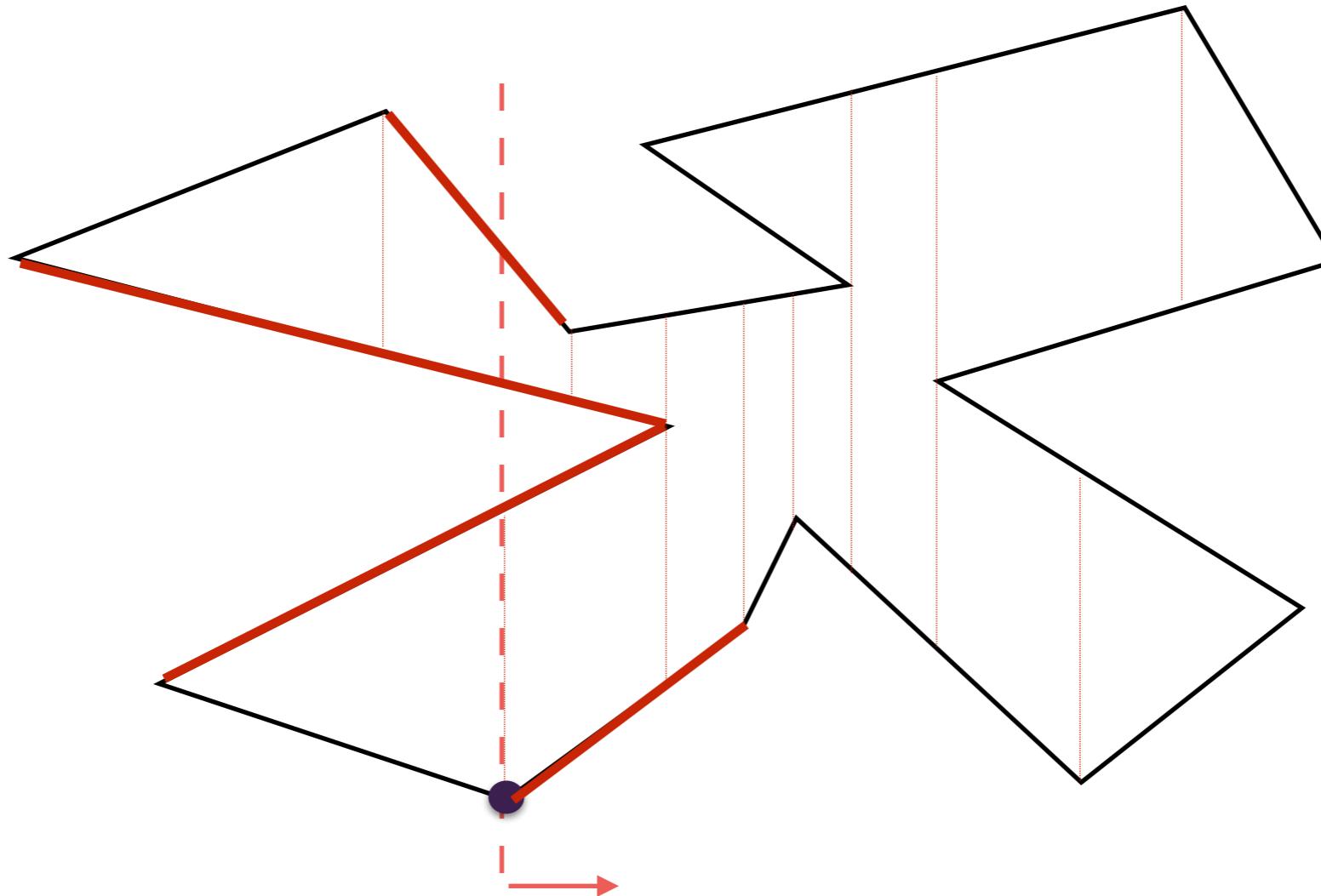


events = polygon vertices

Goal: solve the problem behind the line

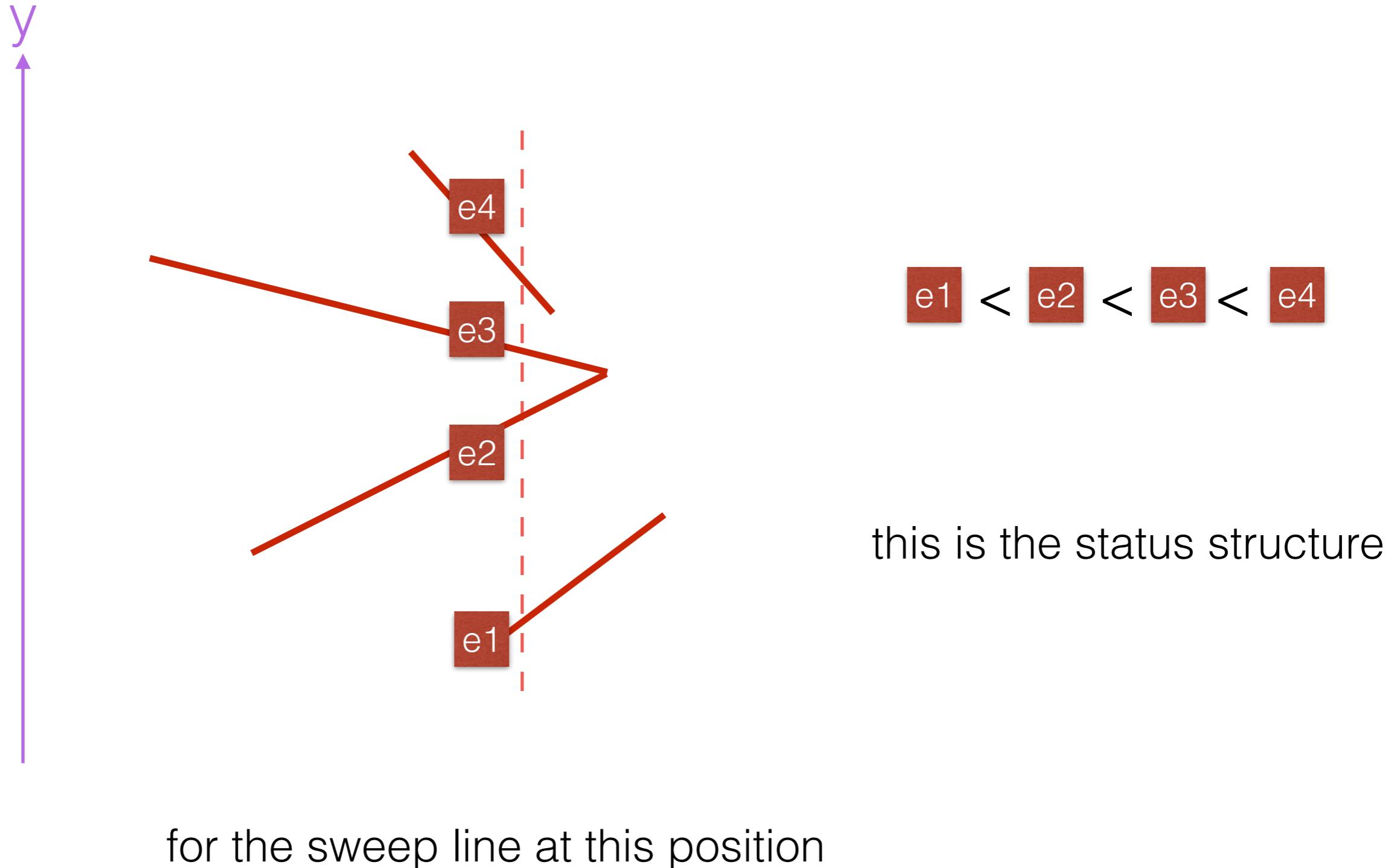
Computing the trapezoid partition in $O(n \lg n)$

- Line sweep

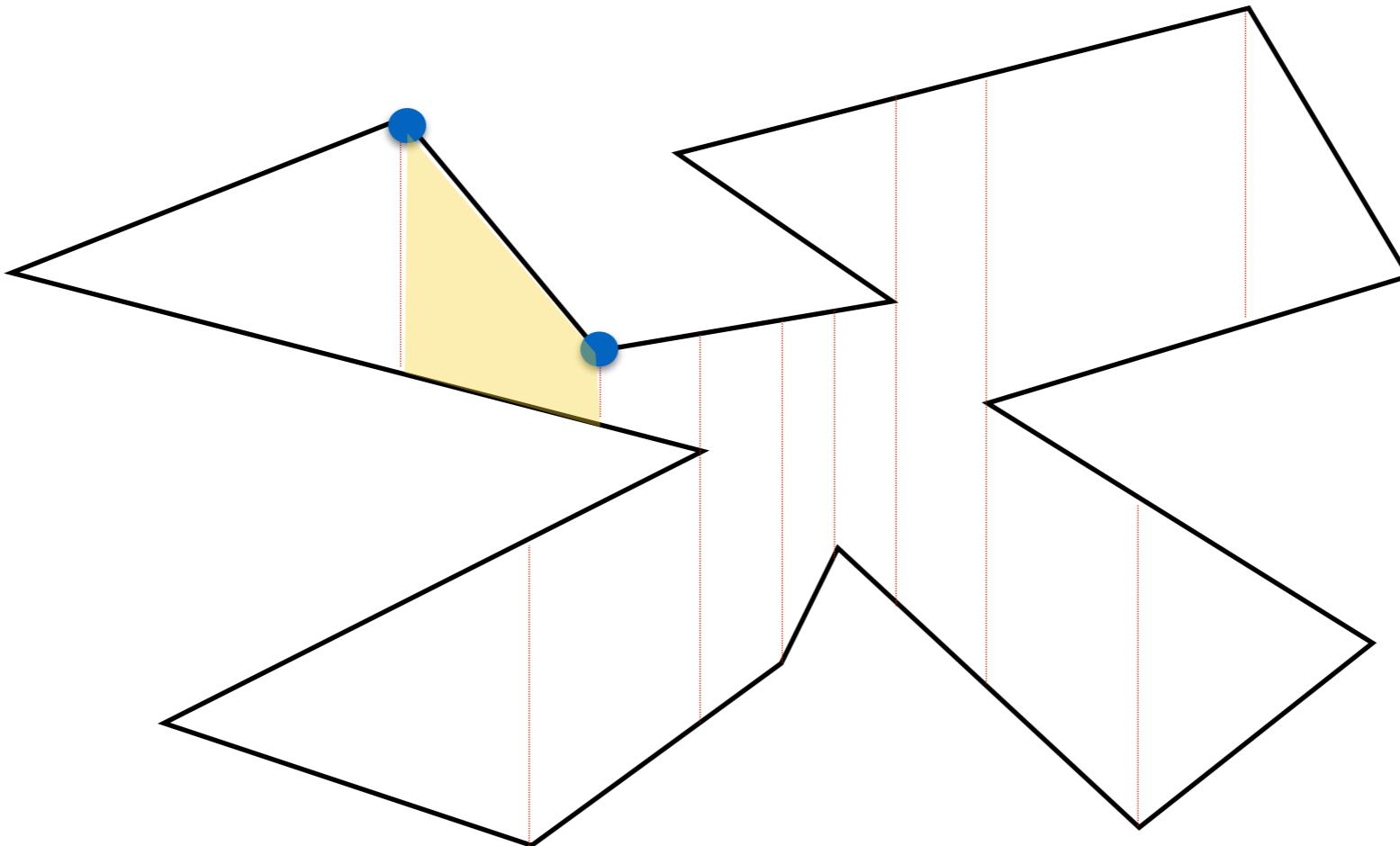


How can we determine the trapezoids?

- Status structure: edges that intersect current sweep line, in y-order

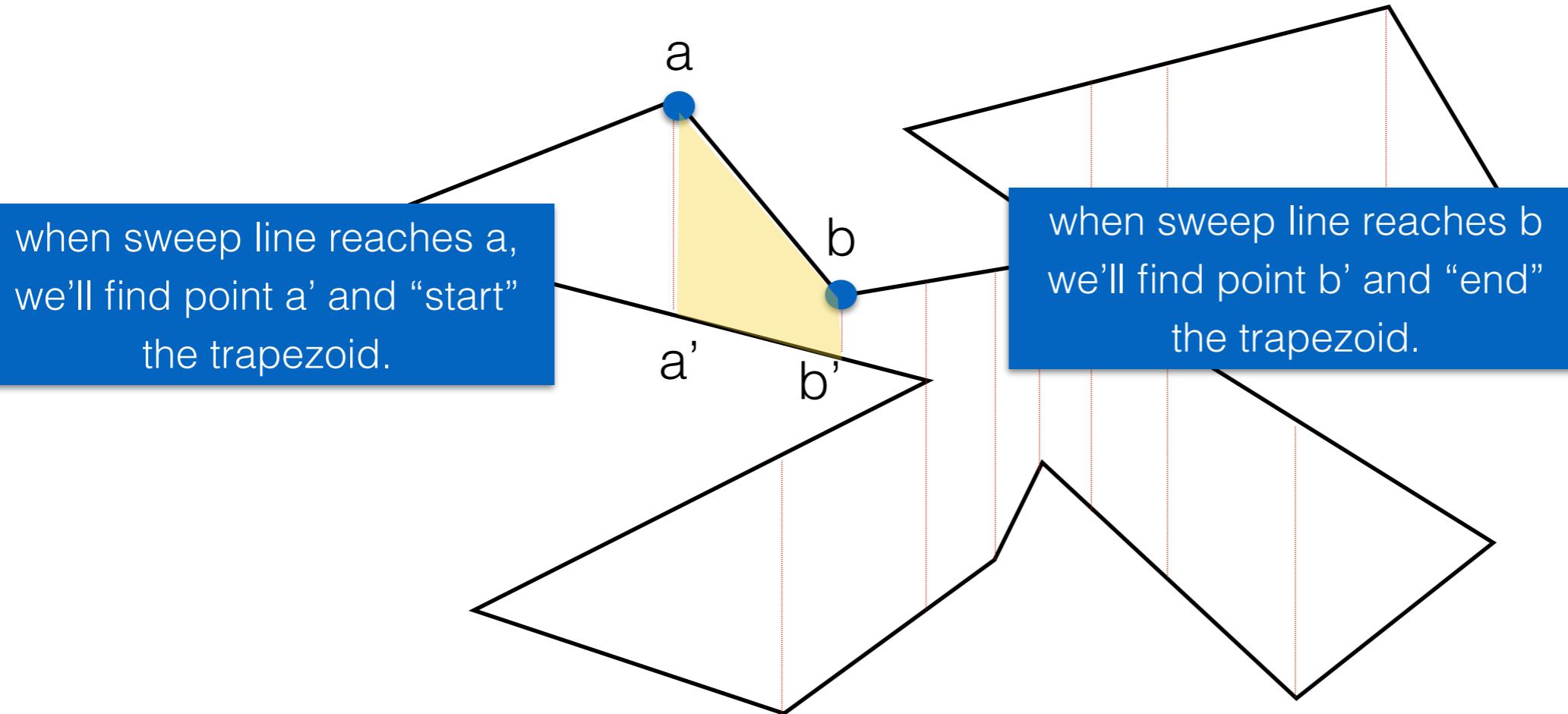


How can we determine the trapezoids?

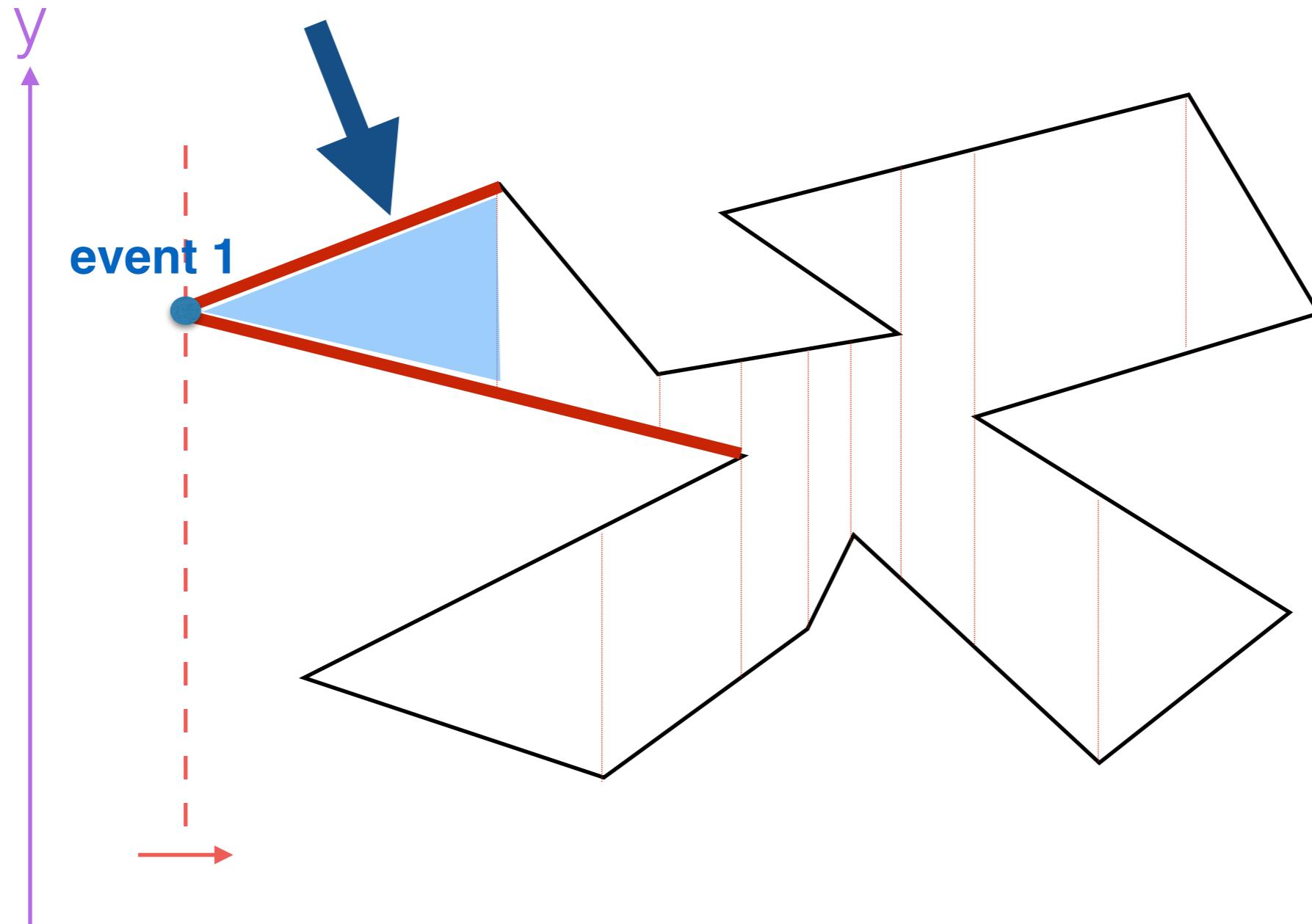


Each trapezoid has 2 vertices of P, one on the left and one on the right

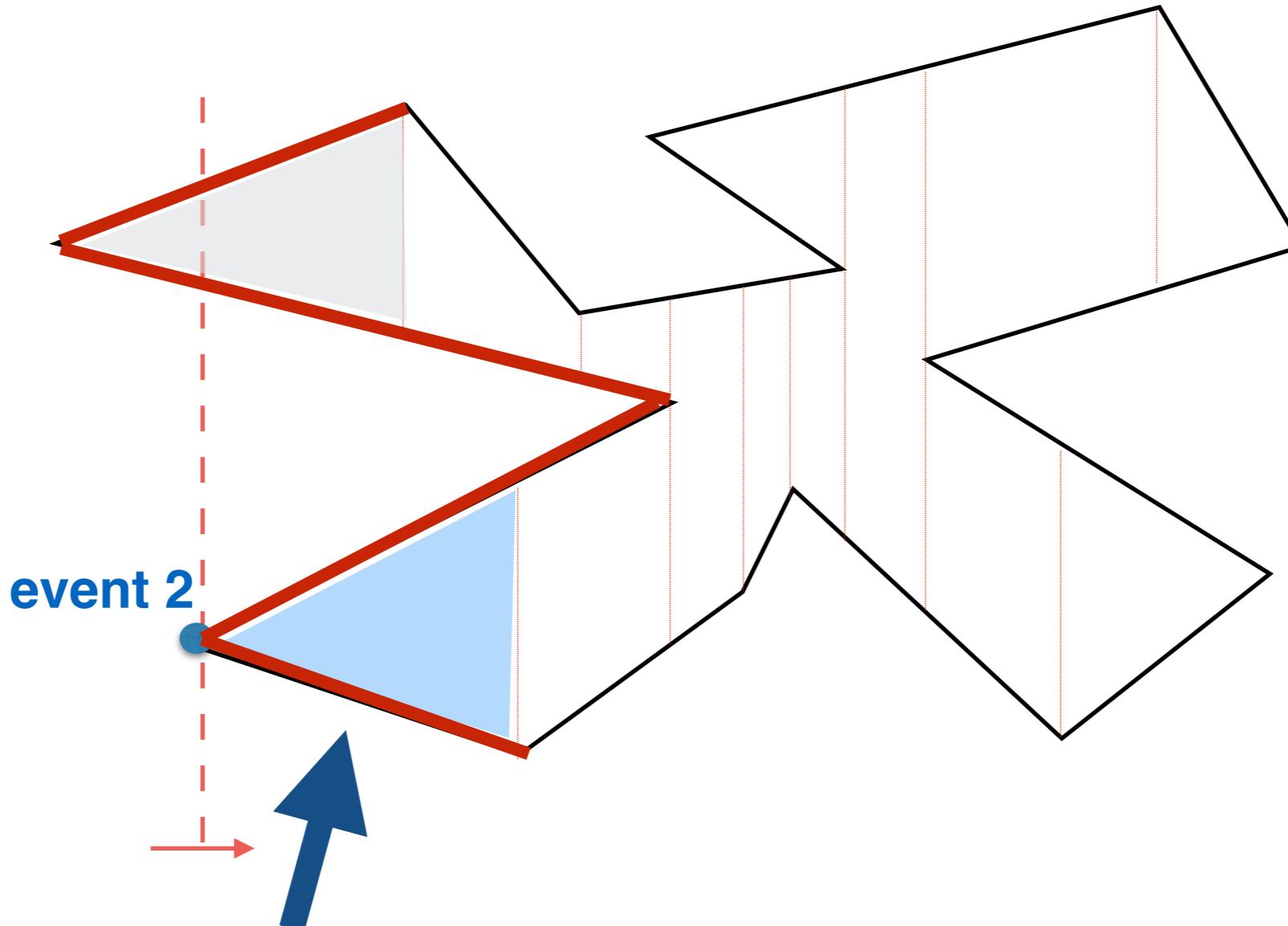
How can we determine the trapezoids?



this trapezoid should be “started” now



y

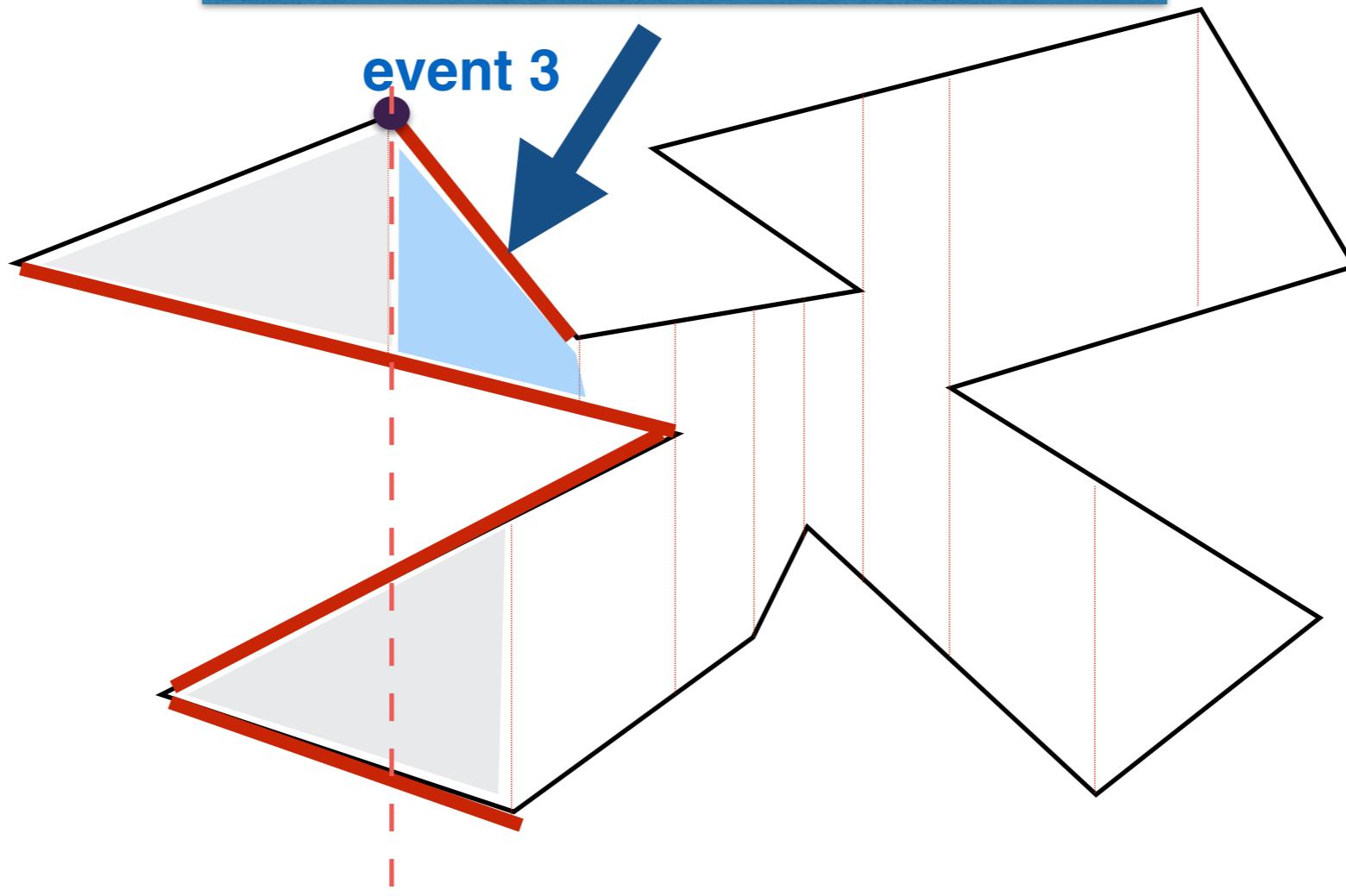


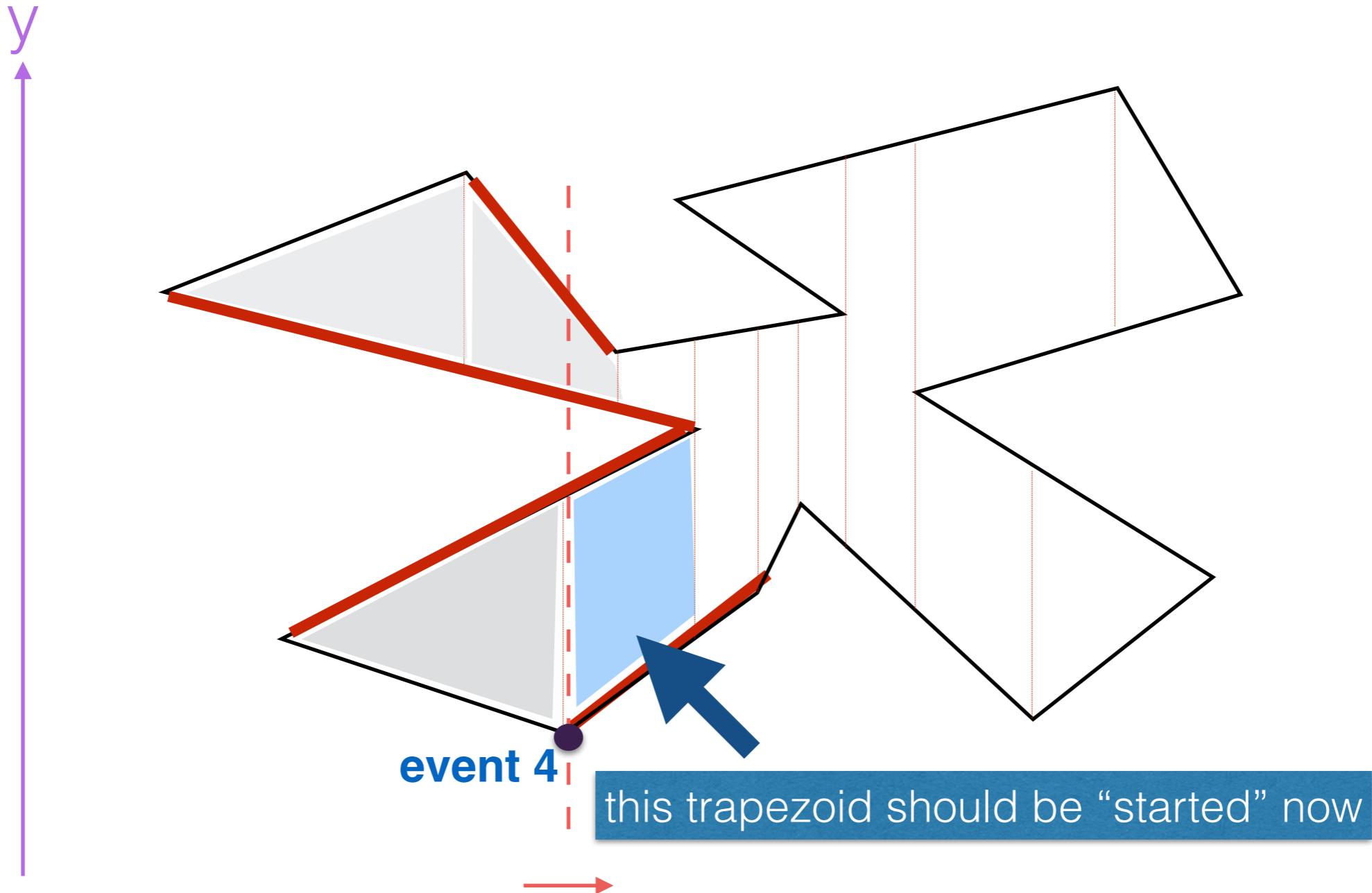
this trapezoid should be “started” now

y

this trapezoid should be “started” now

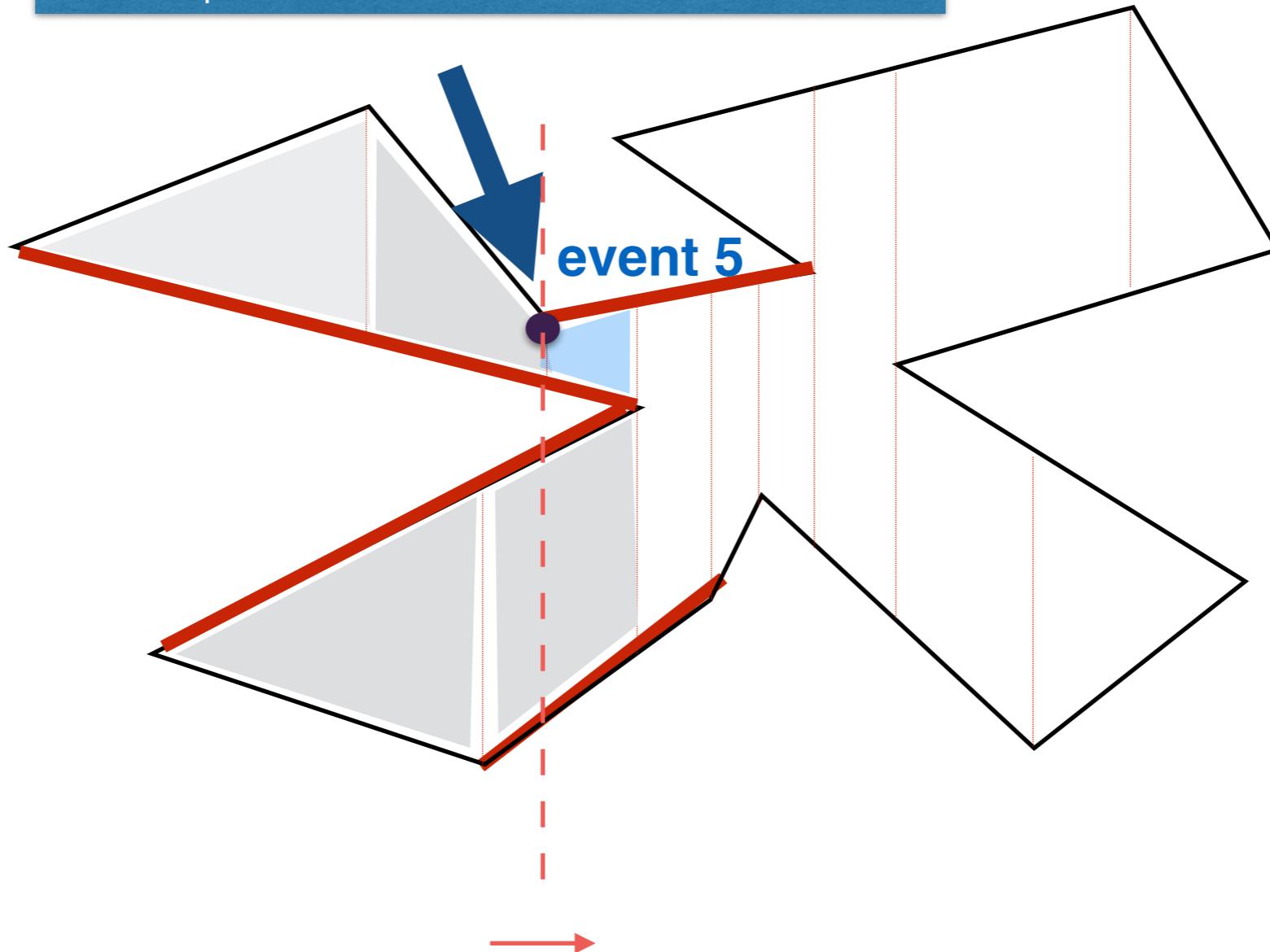
event 3



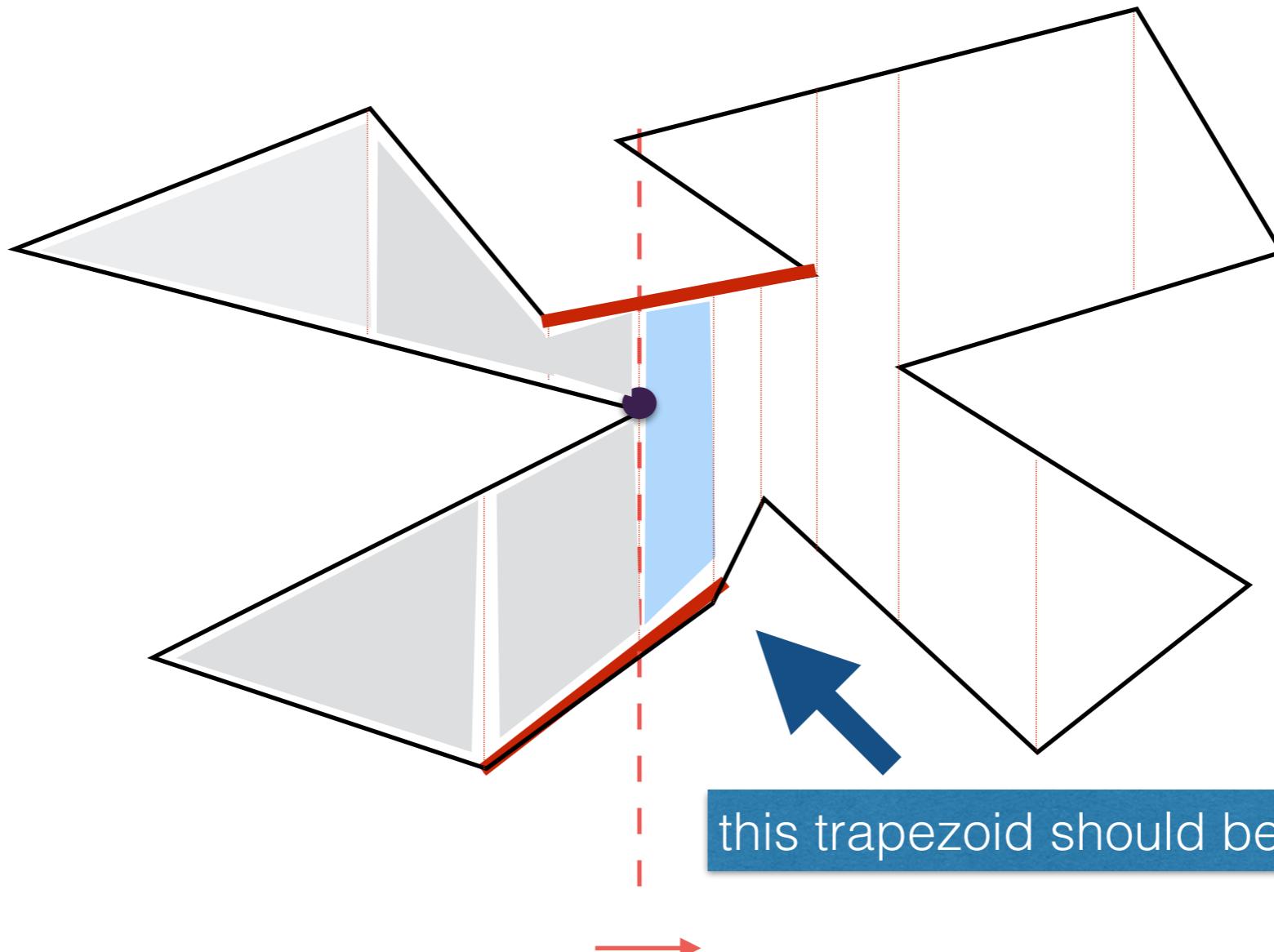


y

this trapezoid should be “started” now

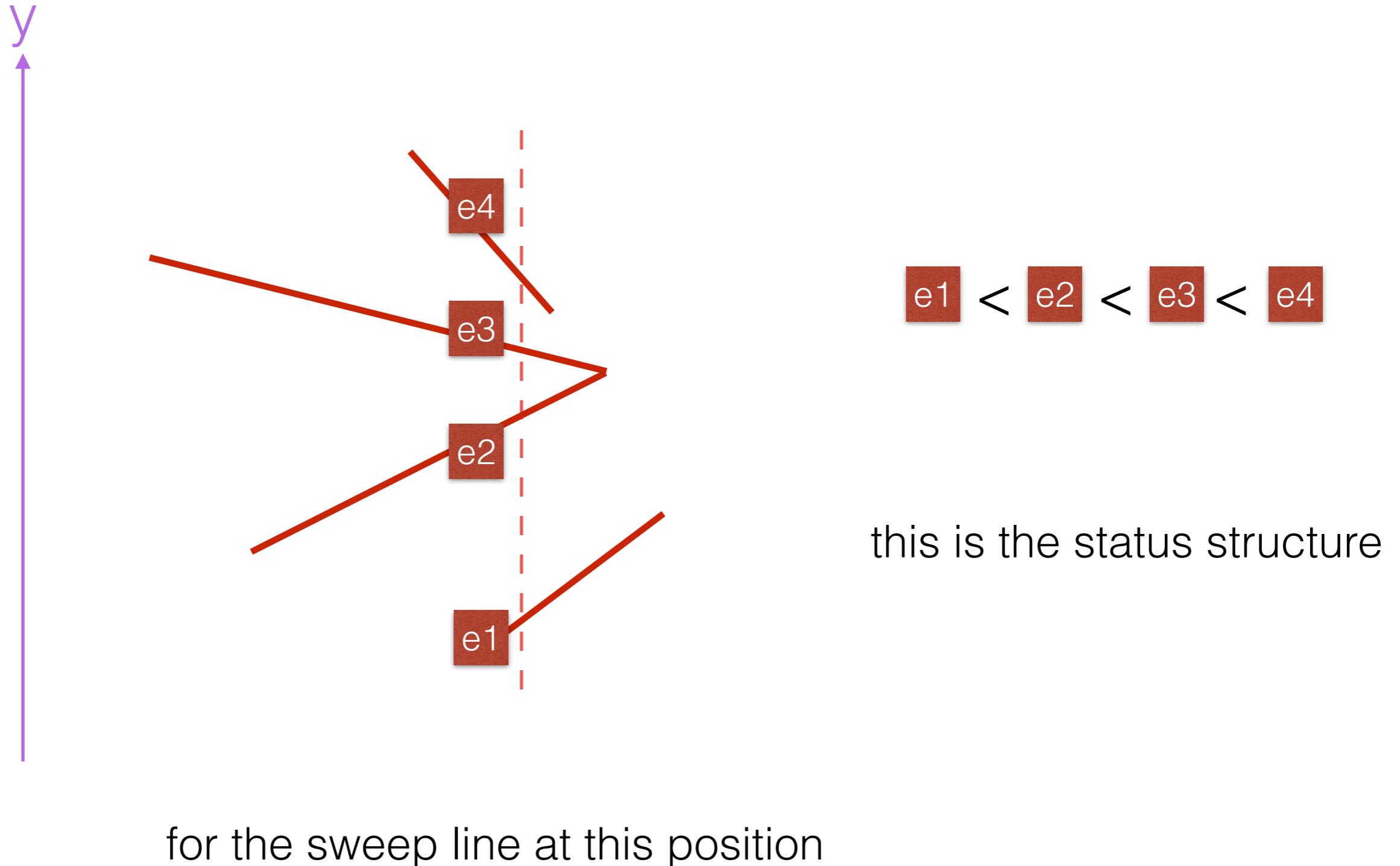


y

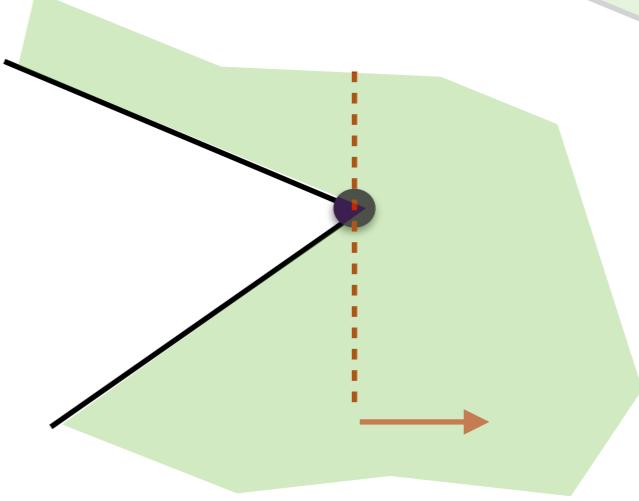
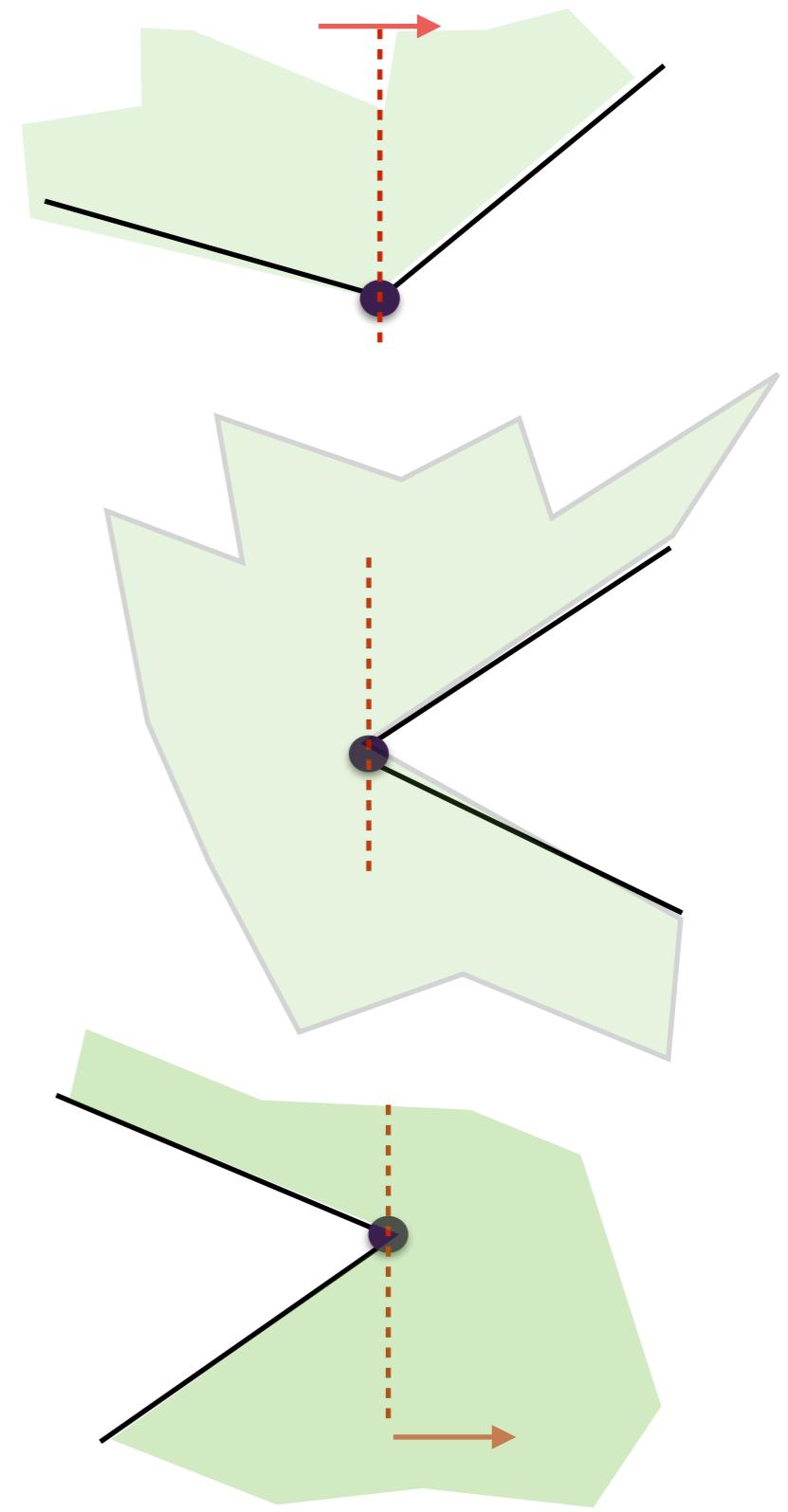
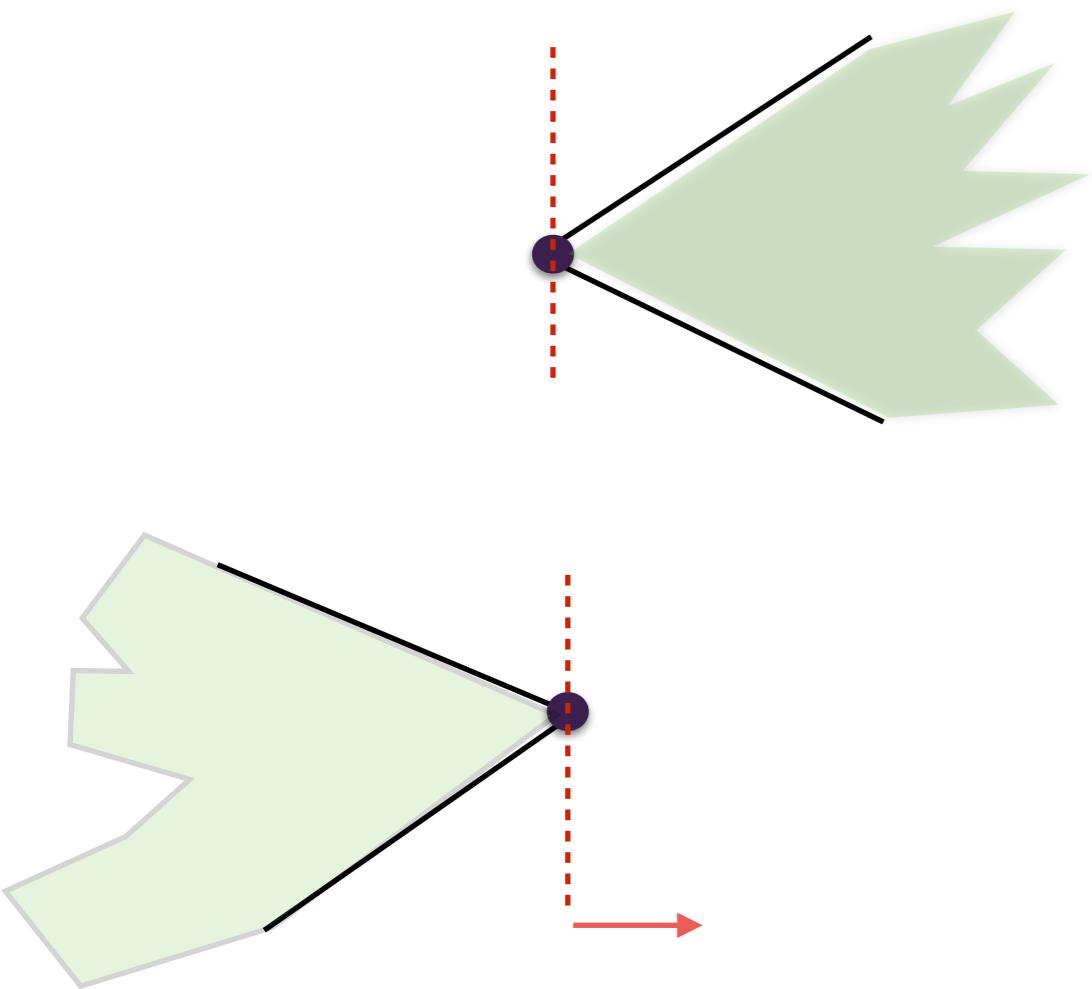
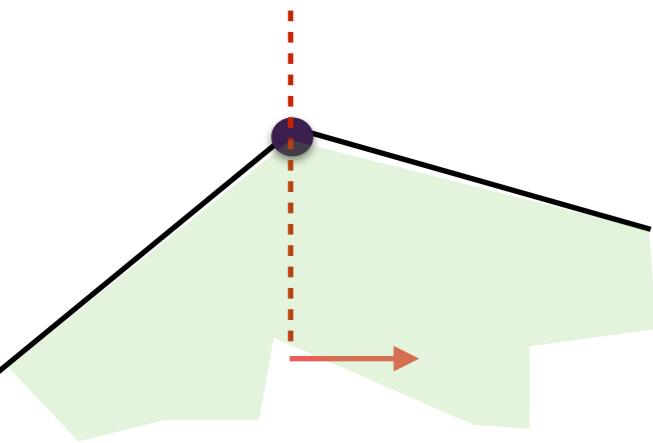


and so on

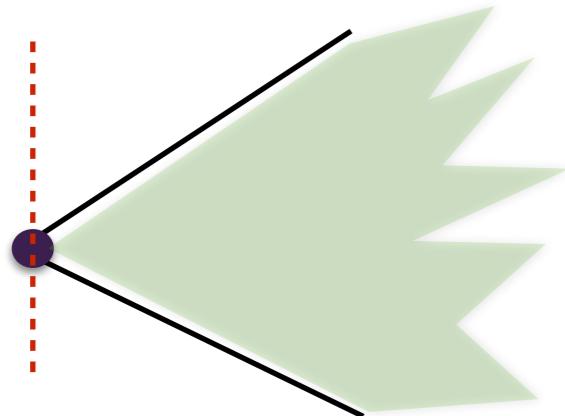
- Status structure: edges that intersect current sweep line, in y-order



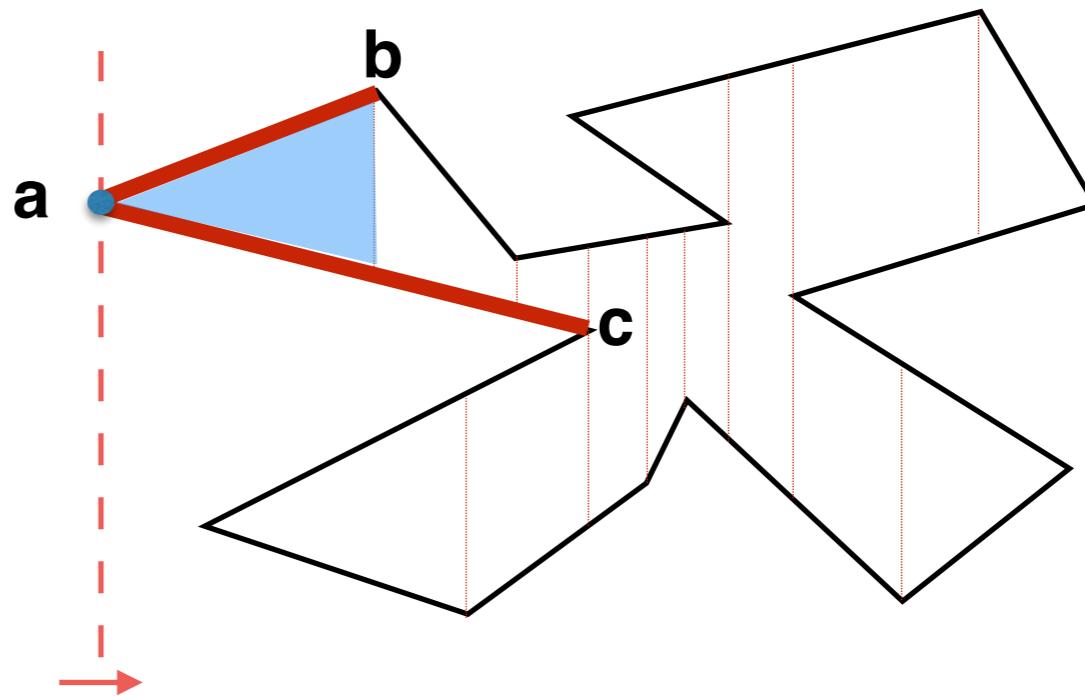
Type of events



Handling events:



Example



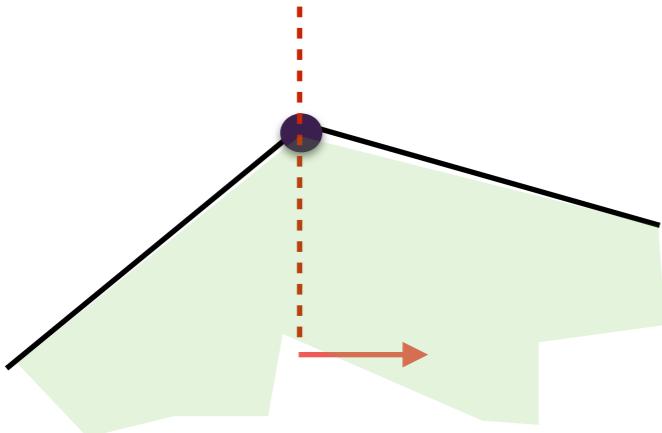
Do:

edge ab becomes active (insert ab in AS)

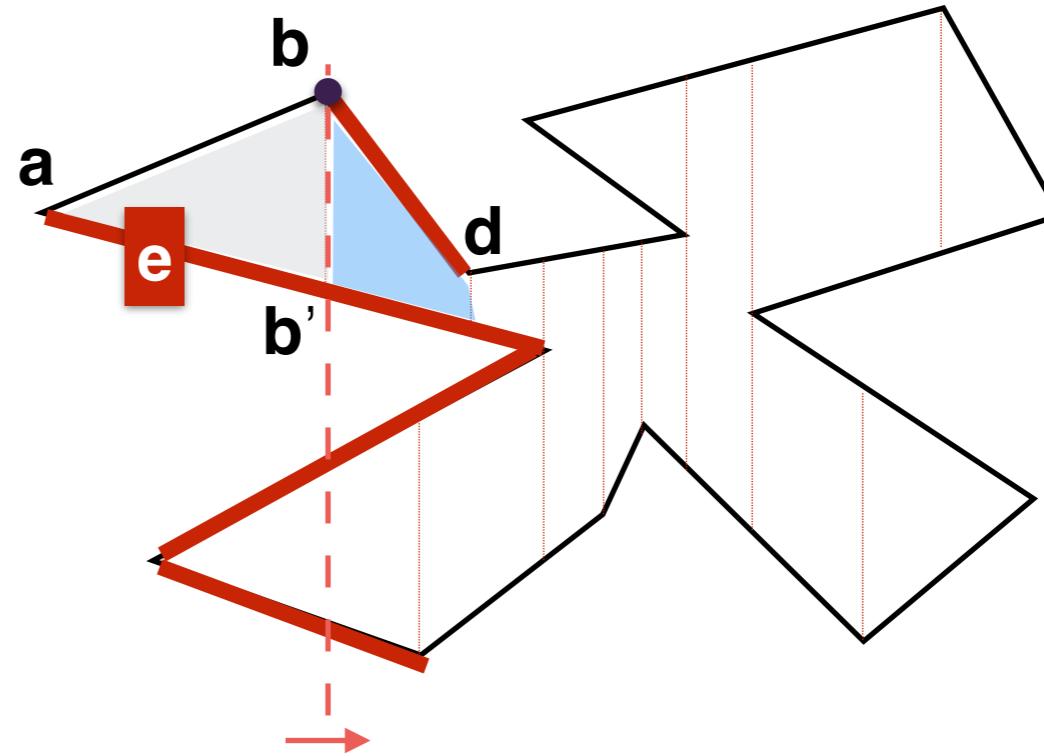
edge ac becomes active (insert ac in AS)

“start” trapezoid to the right of a (a, xx)

Handling events:



Example



b' must be on the edge that is right below b in the active structure

Do:

find $e = \text{pred}(b)$ in AS and find $b' = \text{intersection}(\text{ray}, e)$

edge bd becomes active (insert bd in AS)

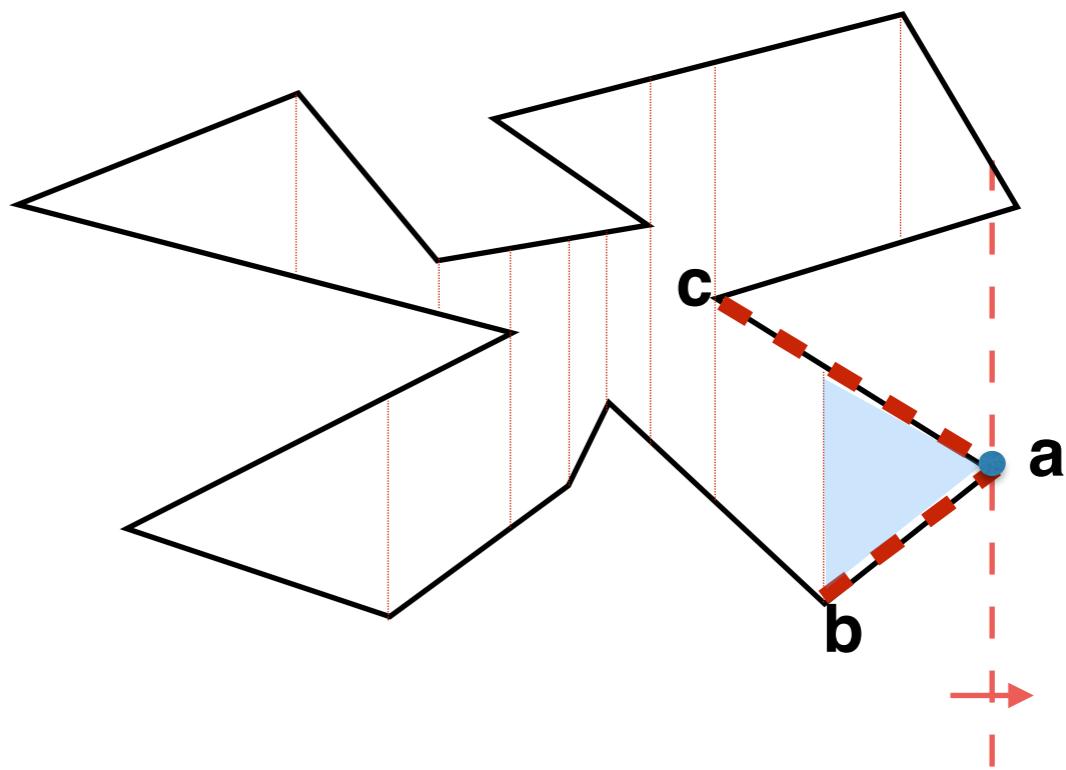
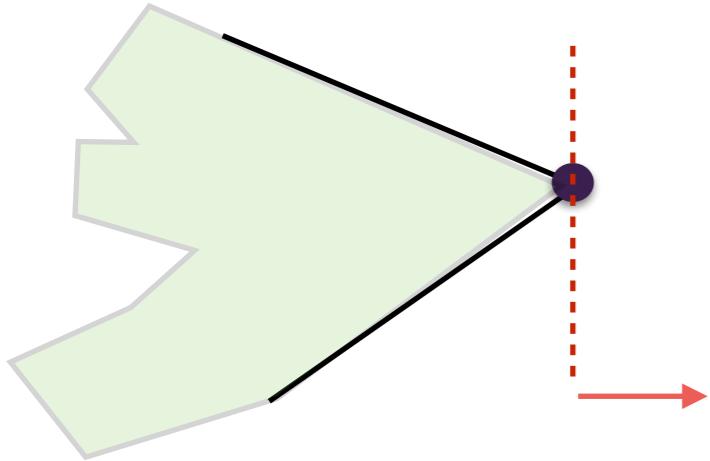
edge ab become inactive (delete ab from AS)

“end” prev. trapezoid to the left of bb'

“start” the trapezoid right of bb' (b, b', xx)

Handling events:

Example



Do:

edge ab becomes inactive (delete ab from AS)

edge ac becomes inactive (delete ac from AS)

“end” trapezoid to the left of a

Summary

- Theorem: A trapezoidal decomposition of a polygon P can be computed in $O(n \lg n)$ time via line sweep.
- This leads to an $O(n \lg n)$ algorithm for polygon triangulation

The end