Polygon Triangulation

Claudio Mirolo

Dip. di Scienze Matematiche, Informatiche e Fisiche
Università di Udine, via delle Scienze 206 – Udine

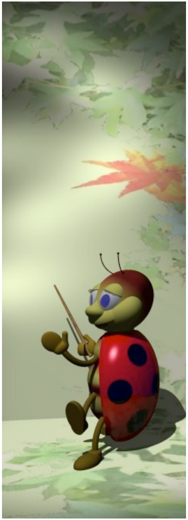
claudio.mirolo@uniud.it

Computational Geometry

www.dimi.uniud.it/claudio

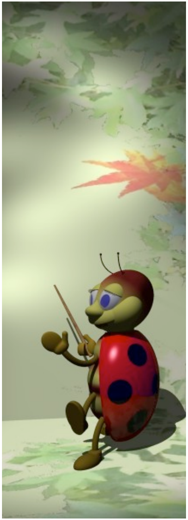
Outline

1. Triangulating a simple polygon
   - definitions
   - existence of a triangulation
   - art gallery problem
2. Monotone partition
   - monotonicity
   - plane sweep
   - analysis
3. Triangulating a monotone polygon
   - invariant arrangement
   - computation costs

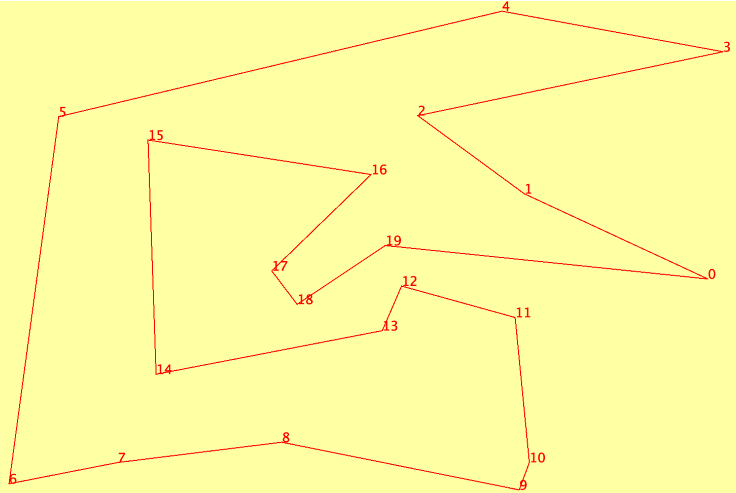


Outline

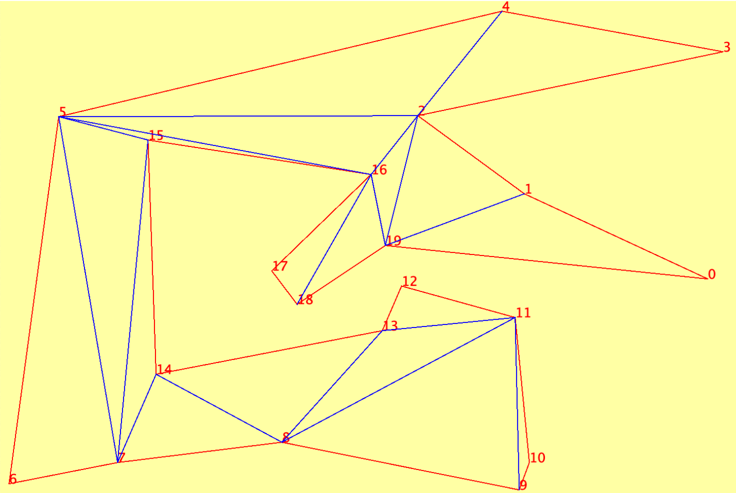
1. Triangulating a simple polygon
   - definitions
   - existence of a triangulation
   - art gallery problem
2. Monotone partition
   - monotonicity
   - plane sweep
   - analysis
3. Triangulating a monotone polygon
   - invariant arrangement
   - computation costs



Triangulation of a simple polygon



Triangulation of a simple polygon



Possible scenario. . .

Guarding an art gallery:

- Polygon triangulation at the core
- Three-coloring of a graph

Possible scenario. . .

Guarding an art gallery:

- Polygon triangulation at the core
- Three-coloring of a graph

Possible scenario. . .

Guarding an art gallery:

- Polygon triangulation at the core
- Three-coloring of a graph

Definitions

Simple polygon:

- No crossings between (open) edges
- No inner holes

Diagonal:

- Open line segment connecting two vertices…
- and wholly contained within the polygon

Definitions

Simple polygon:

- No crossings between (open) edges
- No inner holes

Diagonal:

- Open line segment connecting two vertices...
- and wholly contained within the polygon

Definitions

Simple polygon:

- No crossings between (open) edges
- No inner holes

Diagonal:

- Open line segment connecting two vertices. . .
- and wholly contained within the polygon

Definitions

Simple polygon:

- No crossings between (open) edges
- No inner holes

Diagonal:

- Open line segment connecting two vertices. . .
- and wholly contained within the polygon

Definitions

Simple polygon:

- No crossings between (open) edges
- No inner holes

Diagonal:

- Open line segment connecting two vertices. . .
- and wholly contained within the polygon

Definitions

Simple polygon:

- No crossings between (open) edges
- No inner holes

Diagonal:

- Open line segment connecting two vertices...
- and wholly contained within the polygon

Definitions

Triangulation:

- Decomposition of a polygon into triangles
- by a *maximal* set
- of *non-intersecting* diagonals
- (Maximal set: consider collinear vertices, not in succession. . .)

Triangulations are not in general unique

Definitions

Triangulation:

- Decomposition of a polygon into triangles
- by a *maximal* set
- of *non-intersecting* diagonals
- (Maximal set: consider collinear vertices, not in succession. . .)

Triangulations are not in general unique

Definitions

Triangulation:

- Decomposition of a polygon into triangles
- by a *maximal* set
- of *non-intersecting* diagonals
- (Maximal set: consider collinear vertices, not in succession. . .)

Triangulations are not in general unique

Definitions

Triangulation:

- Decomposition of a polygon into triangles
- by a *maximal* set
- of *non-intersecting* diagonals
- (Maximal set: consider collinear vertices, not in succession. . .)

Triangulations are not in general unique

Definitions

Triangulation:

- Decomposition of a polygon into triangles
- by a *maximal* set
- of *non-intersecting* diagonals
- (Maximal set: consider collinear vertices, not in succession. . .)

Triangulations are not in general unique

Definitions

Triangulation:

- Decomposition of a polygon into triangles
- by a *maximal* set
- of *non-intersecting* diagonals
- (Maximal set: consider collinear vertices, not in succession. . .)

Triangulations are not in general unique

Proposition

A simple polygon P with n vertices
can be partitioned into $n - 2$ triangles

- Proof: by induction on n
- $n = 3$: trivial, since P is a triangle
- Inductive assumption: $k < n$
- Split P by a diagonal into (simple) polygons P' with k' vertices and P'' with k'' vertices

Proposition

A simple polygon P with n vertices
can be partitioned into $n-2$ triangles

- Proof: by induction on n
- $n = 3$: trivial, since P is a triangle
- Inductive assumption: $k < n$
- Split P by a diagonal into (simple) polygons P' with k' vertices and P'' with k'' vertices

Proposition

A simple polygon P with n vertices
can be partitioned into $n-2$ triangles

- Proof: by induction on n
- $n = 3$: trivial, since P is a triangle
- Inductive assumption: $k < n$
- Split P by a diagonal into (simple) polygons P' with k' vertices and P'' with k'' vertices

Proposition

A simple polygon P with n vertices
can be partitioned into $n - 2$ triangles

- Proof: by induction on n
- $n = 3$: trivial, since P is a triangle
- Inductive assumption: $k < n$
- Split P by a diagonal into (simple) polygons P' with k' vertices and P'' with k'' vertices

Proposition

A simple polygon P with n vertices
can be partitioned into $n - 2$ triangles

- Proof: by induction on n
- $n = 3$: trivial, since P is a triangle
- Inductive assumption: $k < n$
- Split P by a diagonal into (simple) polygons P' with k' vertices and P'' with k'' vertices

Inductive step

A simple polygon P with n vertices
can be partitioned into $n - 2$ triangles

- Split P by a diagonal into (simple) polygons P' with k' vertices and P'' with k'' vertices: $k', k'' < n$
- $k' + k'' = n + 2$ since P' and P'' share two vertices
- By the induction assumption P' (P'') can be partitioned into $k' - 2$ ($k'' - 2$) triangles...
- which amounts to $k' + k'' - 4 = n - 2$ triangles overall

Inductive step

A simple polygon P with n vertices
can be partitioned into $n-2$ triangles

- Split P by a diagonal into (simple) polygons P' with k' vertices and P'' with k'' vertices: $k', k'' < n$
- $k' + k'' = n + 2$ since P' and P'' share two vertices
- By the induction assumption P' (P'') can be partitioned into $k' - 2$ ($k'' - 2$) triangles...
- which amounts to $k' + k'' - 4 = n - 2$ triangles overall

Inductive step

A simple polygon P with n vertices
can be partitioned into $n - 2$ triangles

- Split P by a diagonal into (simple) polygons
  P' with k' vertices and P'' with k'' vertices: $k', k'' < n$
- $k' + k'' = n + 2$ since P' and P'' share two vertices
- By the induction assumption
  P' (P'') can be partitioned into $k' - 2$ ($k'' - 2$) triangles...
- which amounts to $k' + k'' - 4 = n - 2$ triangles overall

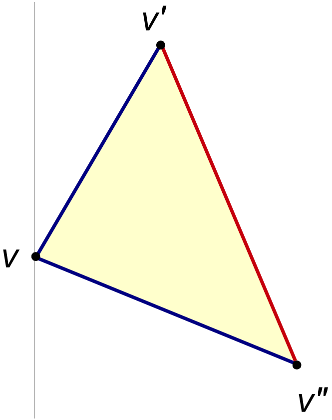
Inductive step

A simple polygon P with n vertices
can be partitioned into $n-2$ triangles

- Split P by a diagonal into (simple) polygons
  P' with k' vertices and P'' with k'' vertices: $k', k'' < n$
- $k' + k'' = n + 2$ since P' and P'' share two vertices
- By the induction assumption
  P' (P'') can be partitioned into $k' - 2$ ($k'' - 2$) triangles...
- which amounts to $k' + k'' - 4 = n - 2$ triangles overall

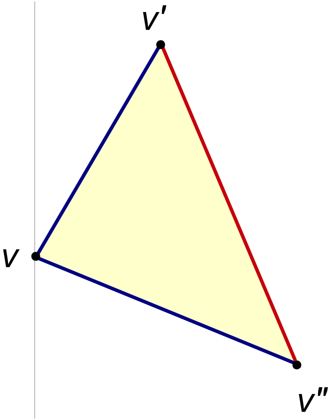
But can we always find a diagonal?

- v leftmost vertex of P
- v' and v'' previous/next of v
- Either no vertex of P inside the triangle $v'vv''$
- and $v'v''$ is a diagonal…



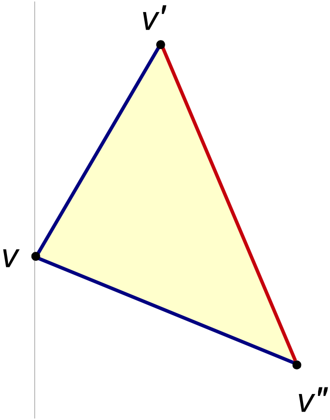
But can we always find a diagonal?

- v leftmost vertex of P
- v' and v'' previous/next of v
- Either no vertex of P inside the triangle $v'vv''$
- and $v'v''$ is a diagonal…



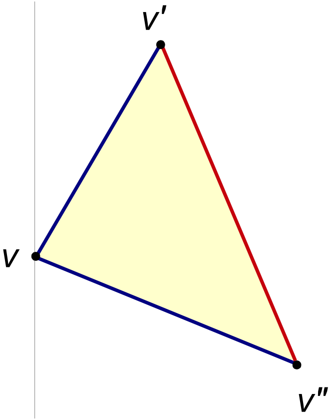
But can we always find a diagonal?

- v leftmost vertex of P
- v' and v'' previous/next of v
- Either no vertex of P inside the triangle $v'vv''$
- and $v'v''$ is a diagonal...



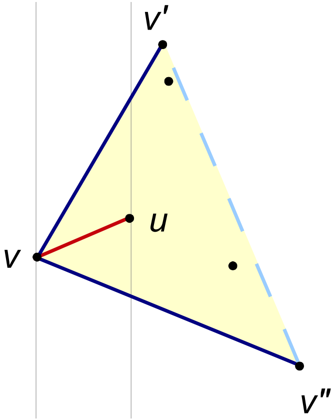
But can we always find a diagonal?

- v leftmost vertex of P
- v' and v'' previous/next of v
- Either no vertex of P inside the triangle $v'vv''$
- and $v'v''$ is a diagonal…



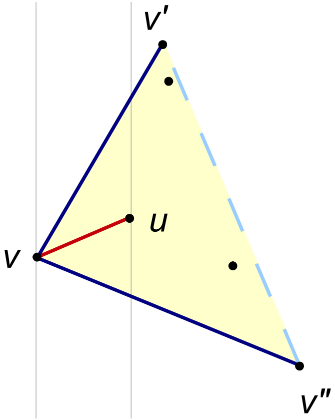
But can we always find a diagonal?

- Or let u be the leftmost of P's vertices lying inside $v'vv''$
- and uv is a diagonal
- since no vertex within $v'vv''$ falls in the vertical strip between v and u
- and there cannot be crossing edges



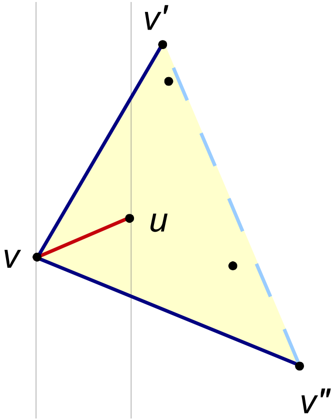
But can we always find a diagonal?

- Or let u be the leftmost of P's vertices lying inside $v'vv''$
- and uv is a diagonal
- since no vertex within $v'vv''$ falls in the vertical strip between v and u
- and there cannot be crossing edges



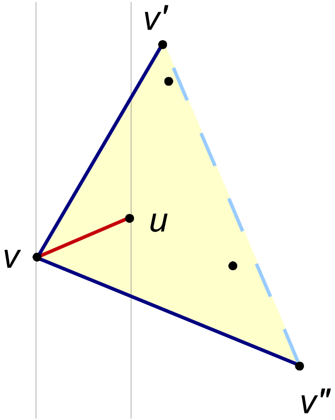
But can we always find a diagonal?

- Or let u be the leftmost of P's vertices lying inside $v'vv''$
- and uv is a diagonal
- since no vertex within $v'vv''$ falls in the vertical strip between v and u
- and there cannot be crossing edges



But can we always find a diagonal?

- Or let u be the leftmost of P's vertices lying inside $v'vv''$
- and uv is a diagonal
- since no vertex within $v'vv''$ falls in the vertical strip between v and u
- and there cannot be crossing edges



Application to art-gallery problems

- Guarding simple polygons
- How many cameras (guarding points)?
- Each triangle must be guarded!
- Minimum number of cameras: *NP-hard* problem
- Pragmatic approach:
  place cameras at vertices shared by several triangles

Application to art-gallery problems

- Guarding simple polygons
- How many cameras (guarding points)?
- Each triangle must be guarded!
- Minimum number of cameras: *NP-hard* problem
- Pragmatic approach:
  place cameras at vertices shared by several triangles

Application to art-gallery problems

- Guarding simple polygons
- How many cameras (guarding points)?
- Each triangle must be guarded!
- Minimum number of cameras: *NP-hard* problem
- Pragmatic approach:
  place cameras at vertices shared by several triangles

Application to art-gallery problems

- Guarding simple polygons
- How many cameras (guarding points)?
- Each triangle must be guarded!
- Minimum number of cameras: *NP-hard* problem
- Pragmatic approach:
  place cameras at vertices shared by several triangles

Application to art-gallery problems

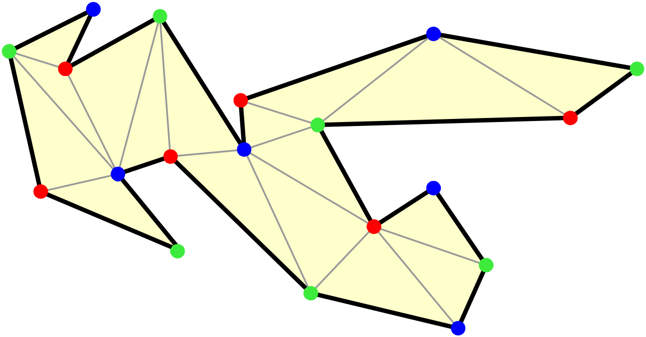
- Guarding simple polygons
- How many cameras (guarding points)?
- Each triangle must be guarded!
- Minimum number of cameras: *NP-hard* problem
- Pragmatic approach:
  place cameras at vertices shared by several triangles

Three-coloring of triangle vertices

- Three-coloring:
  vertices of each triangle are colored differently

Three-coloring of triangle vertices

- Three-coloring, e.g. **RGB** :
  vertices of each triangle are colored differently



Three-coloring of triangle vertices

- Three-coloring, e.g. **RGB** :
  vertices of each triangle are colored differently
- But does one such color assignment exist?
- If it does, we can choose the less frequent color
- guarding points = vertices of this color
- At most $\lfloor \frac{n}{3} \rfloor$ cameras

Three-coloring of triangle vertices

- Three-coloring, e.g. **RGB** :
  vertices of each triangle are colored differently
- But does one such color assignment exist?
- If it does, we can choose the less frequent color
- guarding points = vertices of this color
- At most $\left\lfloor \frac{n}{3} \right\rfloor$ cameras

Three-coloring of triangle vertices

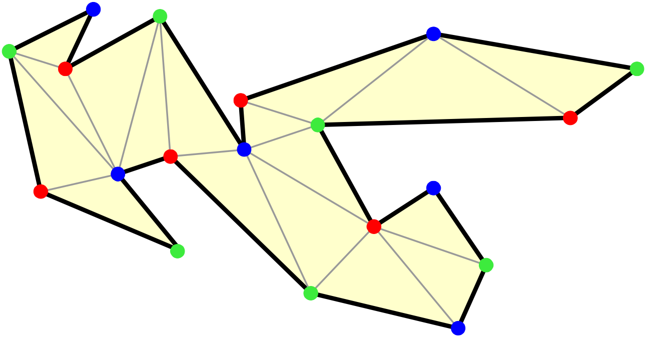
- Three-coloring, e.g. **RGB** :
  vertices of each triangle are colored differently
- But does one such color assignment exist?
- If it does, we can choose the less frequent color
- guarding points = vertices of this color
- At most $\lfloor \frac{n}{3} \rfloor$ cameras

Three-coloring of triangle vertices

- Three-coloring, e.g. **RGB** :
  vertices of each triangle are colored differently
- But does one such color assignment exist?
- If it does, we can choose the less frequent color
- guarding points = vertices of this color
- At most $\lfloor \frac{n}{3} \rfloor$ cameras

Three-coloring of triangle vertices

- In the example, for instance,
  choose either red or blue vertices as guarding points



Existence of a three-coloring

- Observation: the *dual graph* is a tree...
- since removing the edge corresponding to a diagonal results into two disconnected components — no holes!
- Depth-first visit of the dual graph/tree starting from (the node corresponding to) *any* triangle
- Root triangle: any valid three-coloring
- The third, not yet colored vertex of each visited triangle is assigned the color not used in the traversed edge

Existence of a three-coloring

- Observation: the *dual graph* is a tree. . .
- since removing the edge corresponding to a diagonal results into two disconnected components — no holes!
- Depth-first visit of the dual graph/tree starting from (the node corresponding to) *any* triangle
- Root triangle: any valid three-coloring
- The third, not yet colored vertex of each visited triangle is assigned the color not used in the traversed edge

Existence of a three-coloring

- Observation: the *dual graph* is a tree. . .
- since removing the edge corresponding to a diagonal results into two disconnected components — no holes!
- Depth-first visit of the dual graph/tree starting from (the node corresponding to) *any* triangle
- Root triangle: any valid three-coloring
- The third, not yet colored vertex of each visited triangle is assigned the color not used in the traversed edge

Existence of a three-coloring

- Observation: the *dual graph* is a tree…
- since removing the edge corresponding to a diagonal results into two disconnected components — no holes!
- Depth-first visit of the dual graph/tree starting from (the node corresponding to) *any* triangle
- Root triangle: any valid three-coloring
- The third, not yet colored vertex of each visited triangle is assigned the color not used in the traversed edge

Existence of a three-coloring

- Observation: the *dual graph* is a tree…
- since removing the edge corresponding to a diagonal results into two disconnected components — no holes!
- Depth-first visit of the dual graph/tree starting from (the node corresponding to) *any* triangle
- Root triangle: any valid three-coloring
- The third, not yet colored vertex of each visited triangle is assigned the color not used in the traversed edge

Existence of a three-coloring

- Observation: the *dual graph* is a tree. . .
- since removing the edge corresponding to a diagonal results into two disconnected components — no holes!
- Depth-first visit of the dual graph/tree starting from (the node corresponding to) *any* triangle
- Root triangle: any valid three-coloring
- The third, not yet colored vertex of each visited triangle is assigned the color not used in the traversed edge

Existence of a three-coloring

- Tree-traversal *invariant*: for all visited triangles a valid three-coloring has been computed
- No cycles $\rightarrow$ coloring process is not overconstrained
- Hence $\lfloor \frac{n}{3} \rfloor$ cameras are always enough...
- but may also be necessary

Existence of a three-coloring

- Tree-traversal *invariant*: for all visited triangles a valid three-coloring has been computed
- No cycles $\rightarrow$ coloring process is not overconstrained
- Hence $\lfloor \frac{n}{3} \rfloor$ cameras are always enough...
- but may also be necessary

Existence of a three-coloring

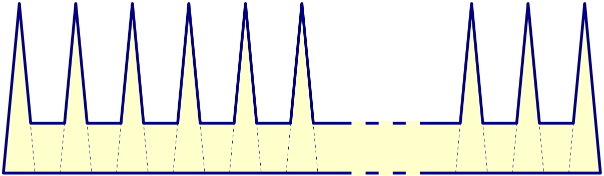
- Tree-traversal *invariant*: for all visited triangles a valid three-coloring has been computed
- No cycles $\rightarrow$ coloring process is not overconstrained
- Hence $\lfloor \frac{n}{3} \rfloor$ cameras are always enough...
- but may also be necessary

Existence of a three-coloring

- Tree-traversal *invariant*: for all visited triangles a valid three-coloring has been computed
- No cycles $\rightarrow$ coloring process is not overconstrained
- Hence $\lfloor \frac{n}{3} \rfloor$ cameras are always enough...
- but may also be necessary

Existence of a three-coloring

- Tree-traversal *invariant*: for all visited triangles a valid three-coloring has been computed
- No cycles $\rightarrow$ coloring process is not overconstrained
- Hence $\lfloor \frac{n}{3} \rfloor$ cameras are always enough...
- but may also be necessary:



Computation costs

- A simple polygon with n vertices
  can be triangulated in $O(\ n \log n\ )$ — see later
- Then, for the *art-gallery* problem…
- a (suboptimal) solution of $\lfloor \frac{n}{3} \rfloor$ guarding points
- can be computed in $O(\ n \log n\ )$

Computation costs

- A simple polygon with n vertices
  can be triangulated in $O(\ n \log n\ )$ — see later
- Then, for the *art-gallery* problem...
- a (suboptimal) solution of $\lfloor \frac{n}{3} \rfloor$ guarding points
- can be computed in $O(\ n \log n\ )$

Computation costs

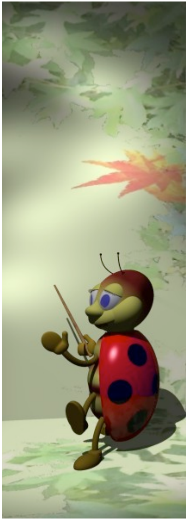
- A simple polygon with n vertices
  can be triangulated in $O(\ n \log n\ )$ — see later
- Then, for the *art-gallery* problem…
- a (suboptimal) solution of $\lfloor \frac{n}{3} \rfloor$ guarding points
- can be computed in $O(\ n \log n\ )$

Computation costs

- A simple polygon with n vertices
  can be triangulated in $O(\ n \log n\ )$ — see later
- Then, for the *art-gallery* problem...
- a (suboptimal) solution of $\lfloor \frac{n}{3} \rfloor$ guarding points
- can be computed in $O(\ n \log n\ )$

Outline

1. Triangulating a simple polygon
   - definitions
   - existence of a triangulation
   - art gallery problem
2. Monotone partition
   - monotonicity
   - plane sweep
   - analysis
3. Triangulating a monotone polygon
   - invariant arrangement
   - computation costs



Approach to triangulation

Triangulation based on the above proposition is inefficient. . .

1. Parition into *monotone* components: *plane sweep*
2. Triangulation of each monotone component: *plane sweep*

Approach to triangulation

Triangulation based on the above proposition is inefficient...

1. Parition into *monotone* components: *plane sweep*
2. Triangulation of each monotone component: *plane sweep*

Approach to triangulation

Triangulation based on the above proposition is inefficient. . .

1. Parition into *monotone* components: *plane sweep*
2. Triangulation of each monotone component: *plane sweep*

Approach to triangulation

Triangulation based on the above proposition is inefficient. . .

I. Parition into *monotone* components: *plane sweep*

II. Triangulation of each monotone component: *plane sweep*

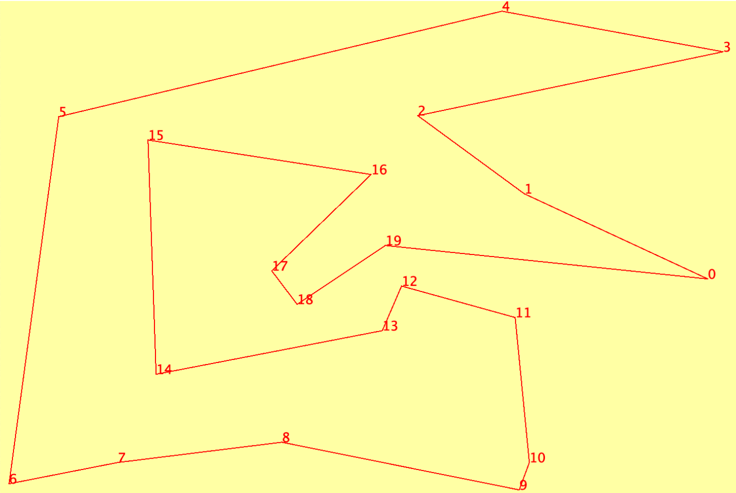
Approach to triangulation

Triangulation based on the above proposition is inefficient. . .

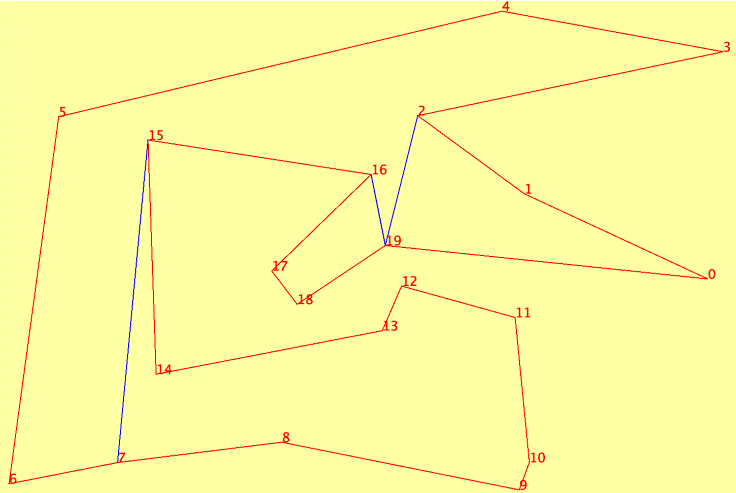
I. Parition into *monotone* components: *plane sweep*

II. Triangulation of each monotone component: *plane sweep*

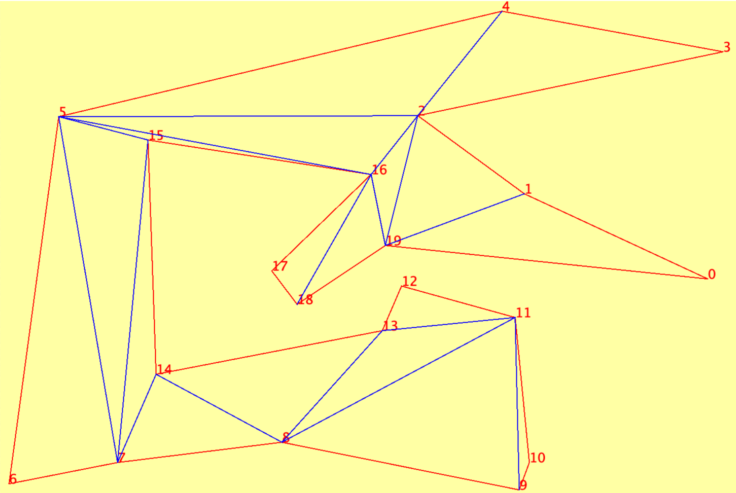
Approach to triangulation: Simple polygon



Approach to triangulation: Monotone partition



Approach to triangulation: Triangulation



Monotonicity

- Not an intrinsic property: relative to a reference direction d
- Weaker property than convexity
- Line segments perpendicular to d connecting points within a *monotone* region M are wholly inside M
- Usually: either x-monotone or y-monotone regions

Monotonicity

- Not an intrinsic property: relative to a reference direction d
- Weaker property than convexity
- Line segments perpendicular to d connecting points within a *monotone* region M are wholly inside M
- Usually: either x-monotone or y-monotone regions

Monotonicity

- Not an intrinsic property: relative to a reference direction d
- Weaker property than convexity
- Line segments perpendicular to d connecting points within a *monotone* region M are wholly inside M
- Usually: either x-monotone or y-monotone regions

Monotonicity

- Not an intrinsic property: relative to a reference direction d
- Weaker property than convexity
- Line segments perpendicular to d connecting points within a *monotone* region M are wholly inside M
- Usually: either x-monotone or y-monotone regions

x-Monotone polygon

- The intersection of a vertical line and an x-monotone polygon P is either empty or connected (a segment)
- P's *upper* and *lower* boundaries are well defined
- While walking from the leftmost vertex to the rightmost vertex along the upper/lower boundary…
- we never move backwards

x-Monotone polygon

- The intersection of a vertical line and an x-monotone polygon P is either empty or connected (a segment)
- *P*'s *upper* and *lower* boundaries are well defined
- While walking from the leftmost vertex to the rightmost vertex along the upper/lower boundary…
- we never move backwards

x-Monotone polygon

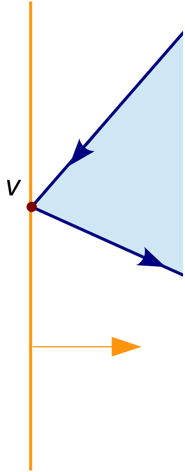
- The intersection of a vertical line and an x-monotone polygon P is either empty or connected (a segment)
- P's *upper* and *lower* boundaries are well defined
- While walking from the leftmost vertex to the rightmost vertex along the upper/lower boundary…
- we never move backwards

x-Monotone polygon

- The intersection of a vertical line and an x-monotone polygon P is either empty or connected (a segment)
- P's *upper* and *lower* boundaries are well defined
- While walking from the leftmost vertex to the rightmost vertex along the upper/lower boundary…
- we never move backwards

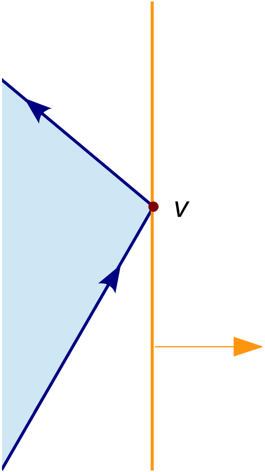
Vertex classification

- *START* vertex
- *END* vertex
- Upper/lower *REGULAR* vertex
- *SPLIT* vertex
- *MERGE* vertex



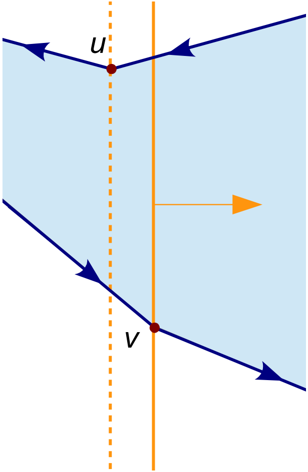
Vertex classification

- *START* vertex
- *END* vertex
- Upper/lower *REGULAR* vertex
- *SPLIT* vertex
- *MERGE* vertex



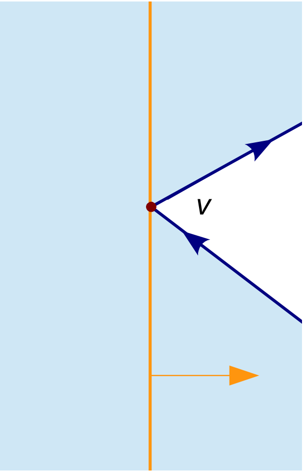
Vertex classification

- *START* vertex
- *END* vertex
- Upper/lower *REGULAR* vertex
- *SPLIT* vertex
- *MERGE* vertex



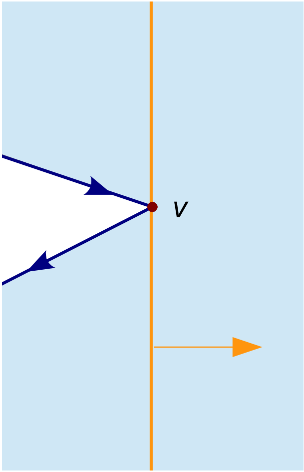
Vertex classification

- *START* vertex
- *END* vertex
- Upper/lower *REGULAR* vertex
- *SPLIT* vertex
- *MERGE* vertex



Vertex classification

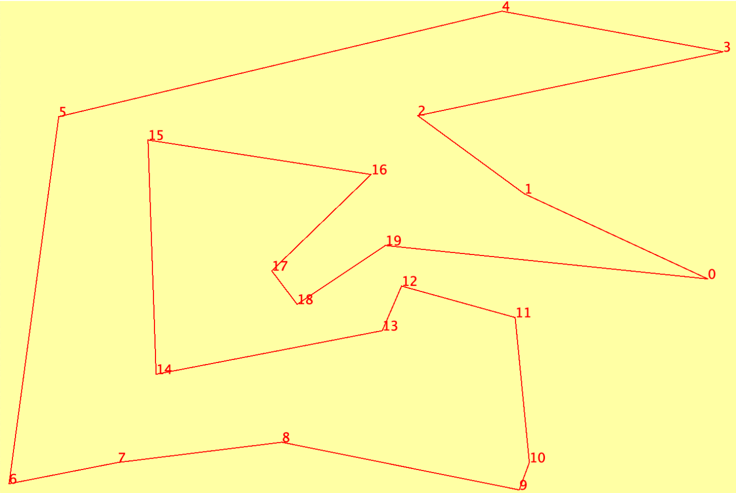
- *START* vertex
- *END* vertex
- Upper/lower *REGULAR* vertex
- *SPLIT* vertex
- *MERGE* vertex



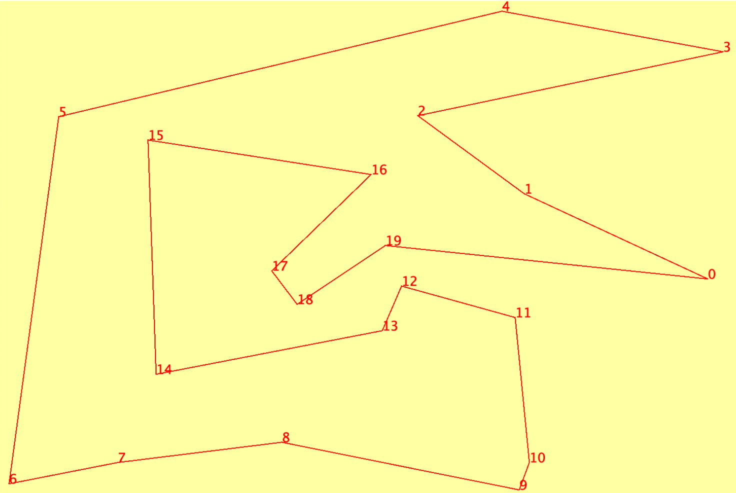
Vertex classification

- *START* vertex
- *END* vertex
- Upper/lower *REGULAR* vertex
- *SPLIT* vertex
- *MERGE* vertex

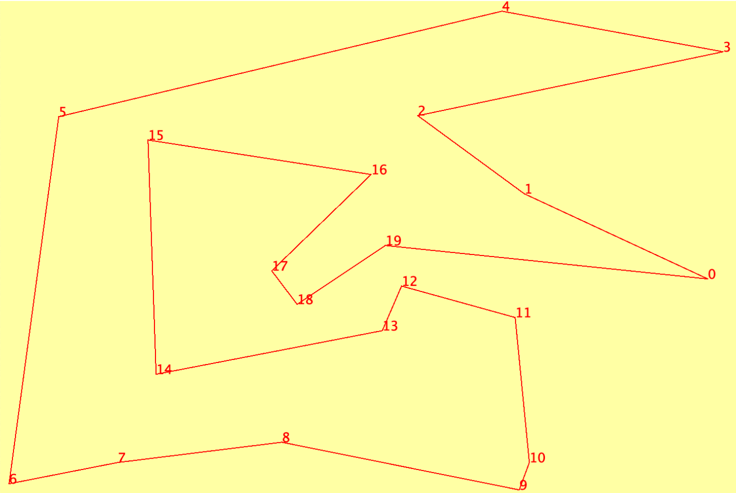
Example – *START* vertices:



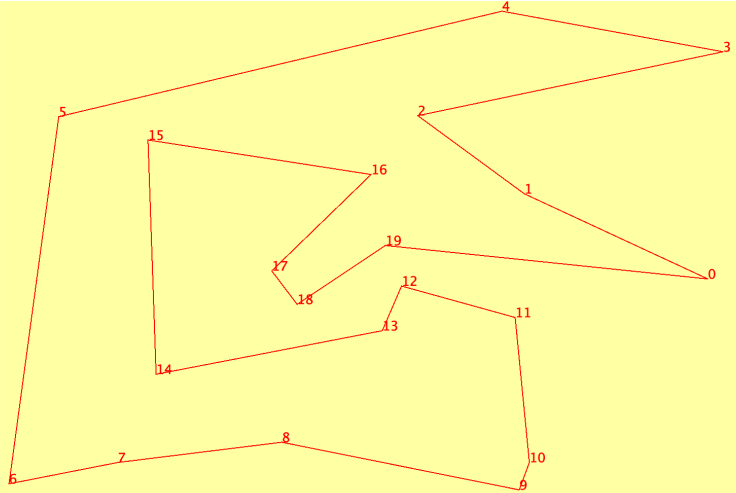
Example – *START* vertices: 6, 17



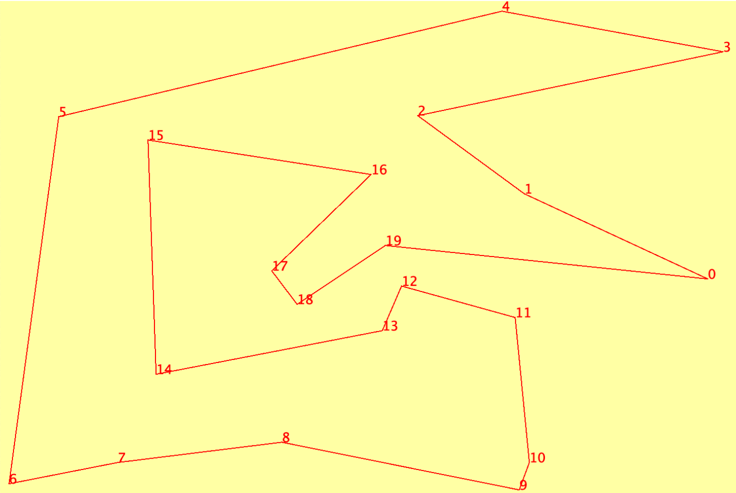
Example – *END* vertices:



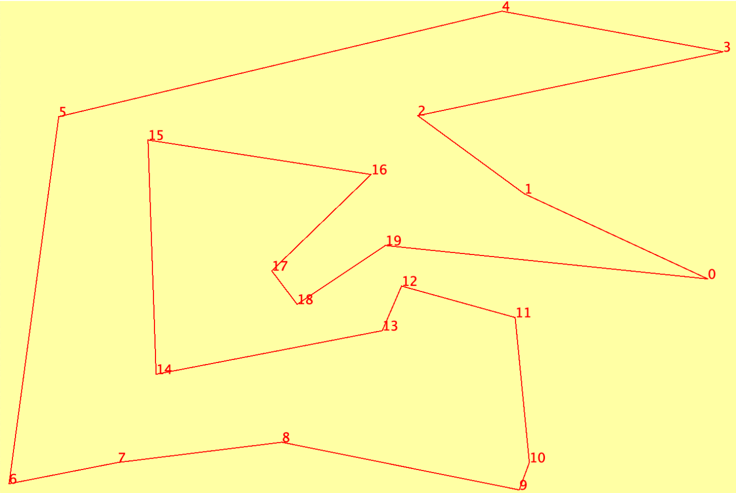
Example – *END* vertices: 0, 3, 10



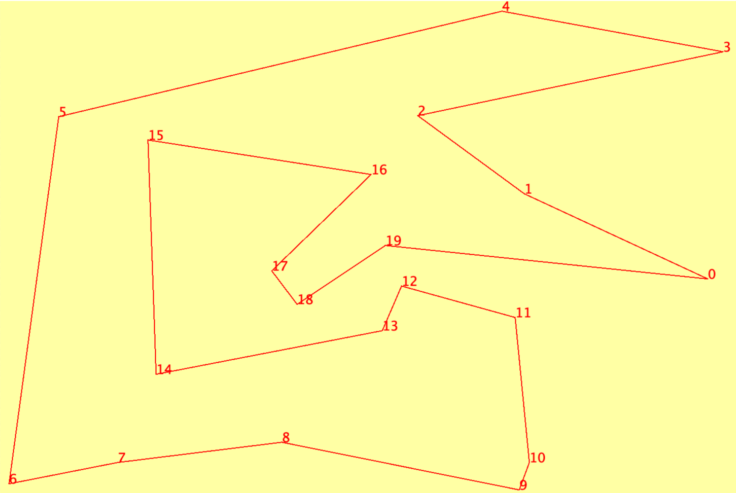
Example – Lower *REGULAR*:



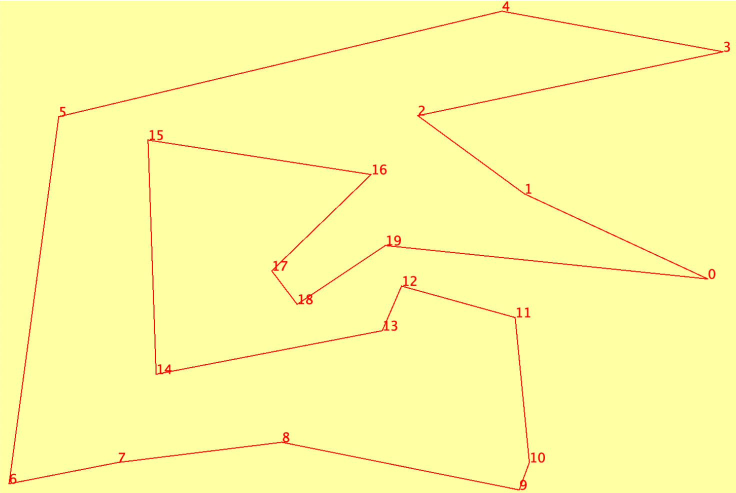
Example – Lower *REGULAR*: 7, 8, 9, 18, 19



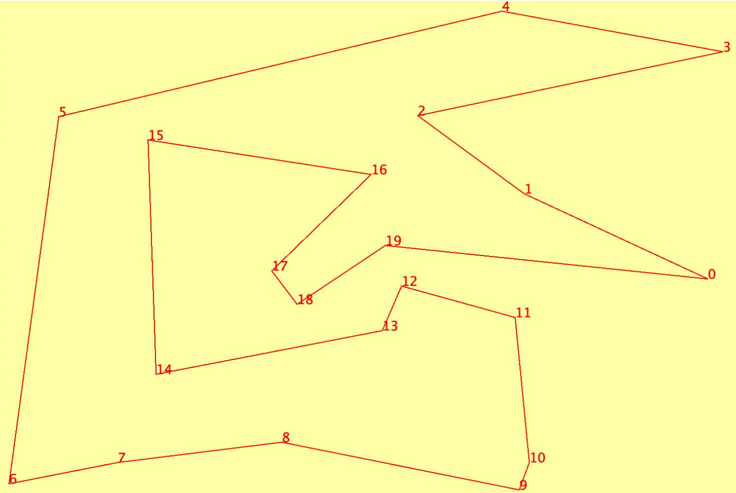
Example – Upper *REGULAR*:



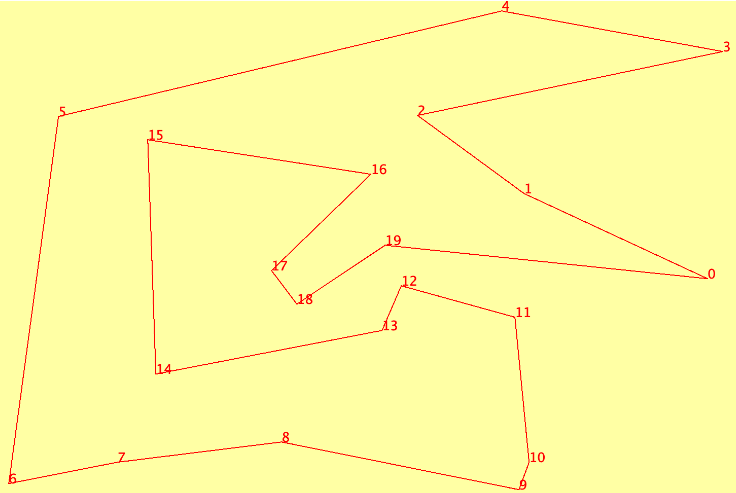
Example – Upper *REGULAR*: 1, 4, 5, 11, 12, 13, 14



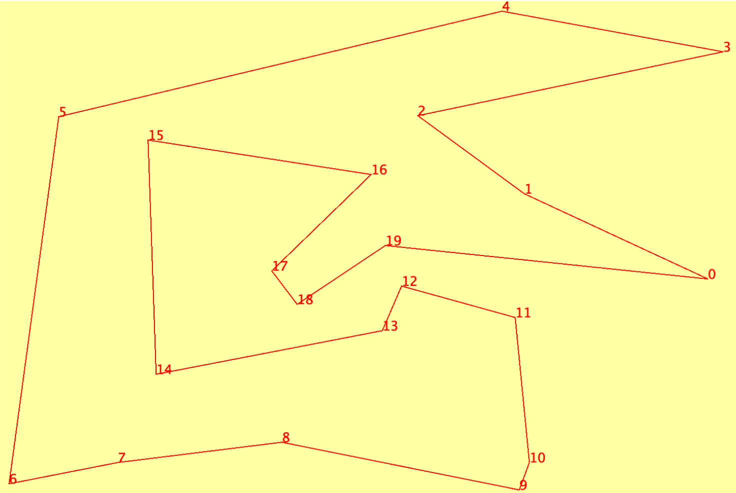
Example – *SPLIT* vertices:



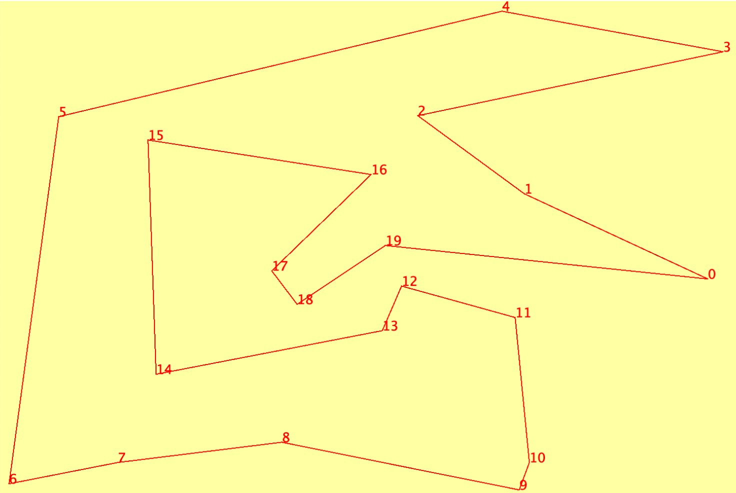
Example – *SPLIT* vertices: 2, 15



Example – *MERGE* vertices:



Example – *MERGE* vertices: 16

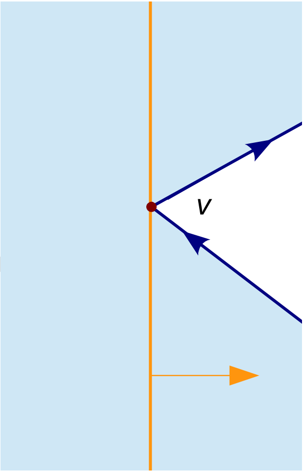


SPLIT and *MERGE* vertices

- Polygon P locally not x-monotone
  near *SPLIT* and *MERGE* vertices
- i.e., *SPLIT* / *MERGE* vertices $\Rightarrow$ P not x-monotone
- Moreover (remarkable property):
  no *SPLIT* / *MERGE* vertices $\Rightarrow$ P x-monotone
- Idea: splitting P by diagonals
  at *SPLIT* and *MERGE* vertices

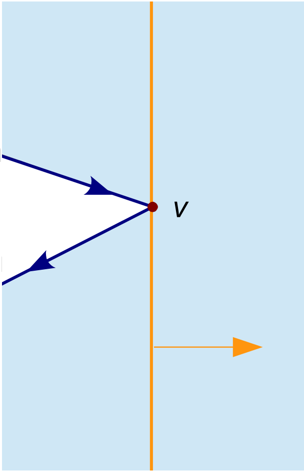
SPLIT and *MERGE* vertices

- Polygon P locally not x-monotone near *SPLIT* and *MERGE* vertices
- i.e., *SPLIT* / *MERGE* vertices $\Rightarrow$ P n
- Moreover (remarkable property): no *SPLIT* / *MERGE* vertices $\Rightarrow$ P x-
- Idea: splitting P by diagonals at *SPLIT* and *MERGE* vertices



SPLIT and *MERGE* vertices

- Polygon P locally not x-monotone near *SPLIT* and *MERGE* vertices
- i.e., *SPLIT* / *MERGE* vertices $\Rightarrow$ *P* n
- Moreover (remarkable property): no *SPLIT* / *MERGE* vertices $\Rightarrow$ *P x*-
- Idea: splitting *P* by diagonals at *SPLIT* and *MERGE* vertices



SPLIT and *MERGE* vertices

- Polygon P locally not x-monotone near *SPLIT* and *MERGE* vertices
- i.e., *SPLIT* / *MERGE* vertices $\Rightarrow$ P not x-monotone
- Moreover (remarkable property): no *SPLIT* / *MERGE* vertices $\Rightarrow$ P x-monotone
- Idea: splitting P by diagonals at *SPLIT* and *MERGE* vertices

SPLIT and *MERGE* vertices

- Polygon P locally not x-monotone near *SPLIT* and *MERGE* vertices
- i.e., *SPLIT* / *MERGE* vertices $\Rightarrow$ P not x-monotone
- Moreover (remarkable property): no *SPLIT* / *MERGE* vertices $\Rightarrow$ P x-monotone
- Idea: splitting P by diagonals at *SPLIT* and *MERGE* vertices

SPLIT and *MERGE* vertices

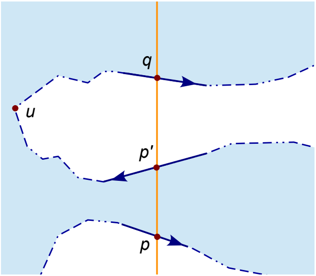
- Polygon P locally not x-monotone
  near *SPLIT* and *MERGE* vertices
- i.e., *SPLIT* / *MERGE* vertices $\Rightarrow$ P not x-monotone
- Moreover (remarkable property):
  no *SPLIT* / *MERGE* vertices $\Rightarrow$ P x-monotone
- Idea: splitting P by diagonals
  at *SPLIT* and *MERGE* vertices

Proof of the monotonicity property

- Suppose P is *not* x-monotone, then a vertical line l intersects P in two or more disconnected segments
- Let pp' be the lowest such segment, from its upper endpoint p'...
- Walk along P's boundary in such a way that P lies to the left
- Until l is crossed again at some point q, say above p'
- Then the leftmost point u along the path from p' to q is a *SPLIT* vertex

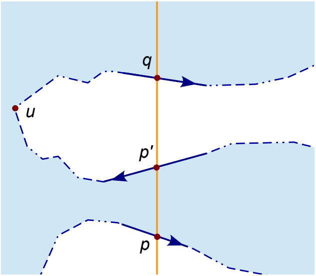
Proof of the monotonicity property

- Suppose P is *not* x-monotone, then a vertical line l intersects P in two or more disconnected segments
- Let pp' be the lowest such segment, from its upper endpoint p'…
- Walk along P's boundary in such a way that P lies to the left
- Until l is crossed again at some point q, say above p'
- Then the leftmost point u along the path from p' to q is a *SPLIT* vertex



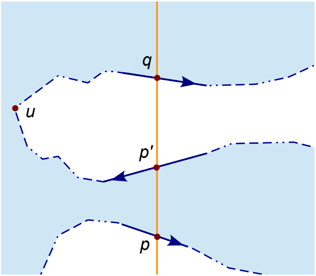
Proof of the monotonicity property

- Suppose P is *not* x-monotone, then a vertical line l intersects P in two or more disconnected segments
- Let pp' be the lowest such segment, from its upper endpoint p'...
- Walk along P's boundary in such a way that P lies to the left
- Until l is crossed again at some point q, say above p'
- Then the leftmost point u along the path from p' to q is a *SPLIT* vertex



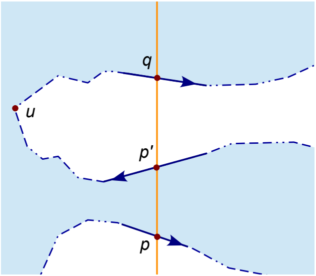
Proof of the monotonicity property

- Suppose P is *not* x-monotone, then a vertical line l intersects P in two or more disconnected segments
- Let pp' be the lowest such segment, from its upper endpoint p'...
- Walk along P's boundary in such a way that P lies to the left
- Until l is crossed again at some point q, say above p'
- Then the leftmost point u along the path from p' to q is a *SPLIT* vertex



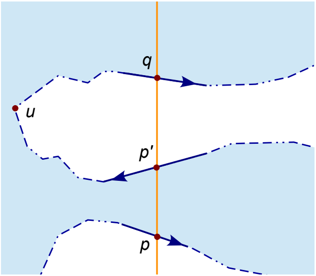
Proof of the monotonicity property

- Suppose P is *not* x-monotone, then a vertical line l intersects P in two or more disconnected segments
- Let pp' be the lowest such segment, from its upper endpoint p'...
- Walk along P's boundary in such a way that P lies to the left
- Until l is crossed again at some point q, say above p'
- Then the leftmost point u along the path from p' to q is a *SPLIT* vertex



Proof of the monotonicity property

- Suppose P is *not* x-monotone, then a vertical line l intersects P in two or more disconnected segments
- Let pp' be the lowest such segment, from its upper endpoint p'...
- Walk along P's boundary in such a way that P lies to the left
- Until l is crossed again at some point q, say above p'
- Then the leftmost point u along the path from p' to q is a *SPLIT* vertex

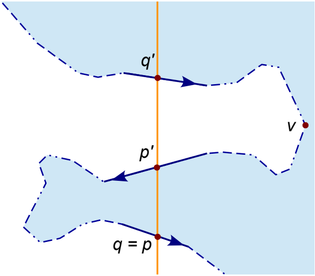


Proof of the monotonicity property

- If q is not above p', it must be the case that $q = p$ (since there are no points of P below p)
- Then from p' walk along P's boundary in the opposite direction
- Until l is crossed again at q' — above p'
- Notice that $q' = p$ would mean that $l \cap P = pp'$ is connected
- Then the rightmost point v along the path from p' to q' is a *MERGE* vertex

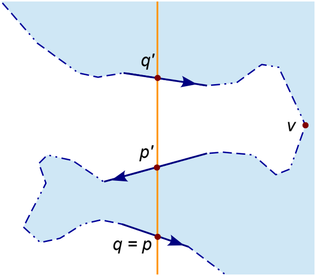
Proof of the monotonicity property

- If q is not above p', it must be the case that $q = p$ (since there are no points of P below p)
- Then from p' walk along P's boundary in the opposite direction
- Until l is crossed again at q' — above p'
- Notice that $q' = p$ would mean that $l \cap P = pp'$ is connected
- Then the rightmost point v along the path from p' to q' is a *MERGE* vertex



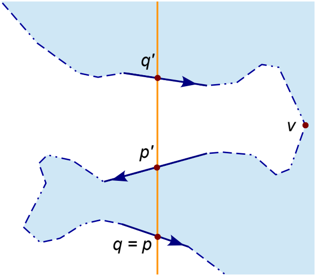
Proof of the monotonicity property

- If q is not above p', it must be the case that $q = p$ (since there are no points of P below p)
- Then from p' walk along P's boundary in the opposite direction
- Until l is crossed again at q' — above p'
- Notice that $q' = p$ would mean that $l \cap P = pp'$ is connected
- Then the rightmost point v along the path from p' to q' is a *MERGE* vertex



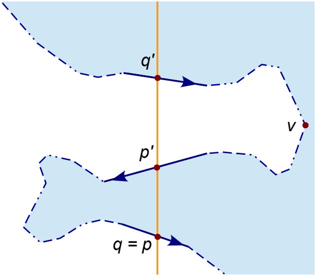
Proof of the monotonicity property

- If q is not above p', it must be the case that $q = p$ (since there are no points of P below p)
- Then from p' walk along P's boundary in the opposite direction
- Until l is crossed again at q' — above p'
- Notice that $q' = p$ would mean that $l \cap P = pp'$ is connected
- Then the rightmost point v along the path from p' to q' is a *MERGE* vertex



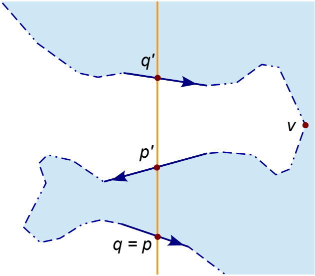
Proof of the monotonicity property

- If q is not above p', it must be the case that $q = p$ (since there are no points of P below p)
- Then from p' walk along P's boundary in the opposite direction
- Until l is crossed again at q' — above p'
- Notice that $q' = p$ would mean that $l \cap P = pp'$ is connected
- Then the rightmost point v along the path from p' to q' is a *MERGE* vertex



Proof of the monotonicity property

- If q is not above p', it must be the case that $q = p$ (since there are no points of P below p)
- Then from p' walk along P's boundary in the opposite direction
- Until l is crossed again at q' — above p'
- Notice that $q' = p$ would mean that $l \cap P = pp'$ is connected
- Then the rightmost point v along the path from p' to q' is a *MERGE* vertex



Plane sweep approach

- Goal: Partition of P into x-monotone components
- Means: Diagonals splitting P at *SPLIT* / *MERGE* vertices
- Approach: Plane sweep

Plane sweep approach

- Goal: Partition of P into x-monotone components
- Means: Diagonals splitting P at *SPLIT* / *MERGE* vertices
- Approach: Plane sweep

Plane sweep approach

- Goal: Partition of P into x-monotone components
- Means: Diagonals splitting P at $SPLIT\,/\,MERGE$ vertices
- Approach: Plane sweep

Plane sweep approach

- Events: *P*'s vertices (all available since the beginning)
- Event types:
  START, *END*, *LOWER_REGULAR*,
  UPPER_REGULAR, *SPLIT*, *MERGE*
- Sweep-line structure:
  (just) lower boundaries of the
  monotone components being built

Plane sweep approach

- Events: *P*'s vertices (all available since the beginning)
- Event types:
  START, *END*, *LOWER_REGULAR*,
  UPPER_REGULAR, *SPLIT*, *MERGE*
- Sweep-line structure:
  (just) lower boundaries of the
  monotone components being built

Plane sweep approach

- Events: *P*'s vertices (all available since the beginning)
- Event types:
  START, *END*, *LOWER_REGULAR*,
  UPPER_REGULAR, *SPLIT*, *MERGE*
- Sweep-line structure:
  (just) lower boundaries of the
  monotone components being built

Plane sweep approach

- *SPLIT* event: diagonal can be promptly drawn
- *MERGE* event: pending task
- Appropriate vertices to be connected with *MERGE* vertices will be found later

Plane sweep approach

- *SPLIT* event: diagonal can be promptly drawn
- *MERGE* event: pending task
- Appropriate vertices to be connected with *MERGE* vertices will be found later

Plane sweep approach

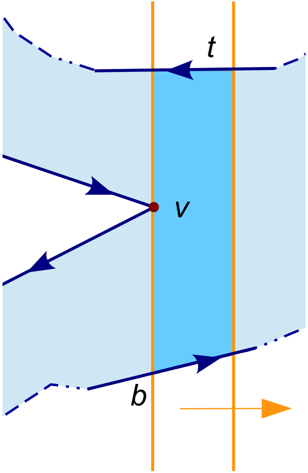
- *SPLIT* event: diagonal can be promptly drawn
- *MERGE* event: pending task
- Appropriate vertices to be connected with *MERGE* vertices will be found later

Invariant

- *SPLIT* vertices to the left
  of the sweep line: diagonal added
- *MERGE* vertices to the left
  of the sweep line: ...
- diagonal added if and only if
  a second vertex of P falls in the
  trapezoid between edges b and t
- $v = helper(b)$

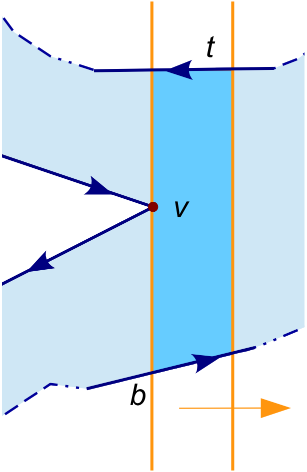
Invariant

- *SPLIT* vertices to the left of the sweep line: diagonal added
- *MERGE* vertices to the left of the sweep line: . . .
- diagonal added if and only if a second vertex of P falls in the trapezoid between edges b and t
- $v = helper(b)$



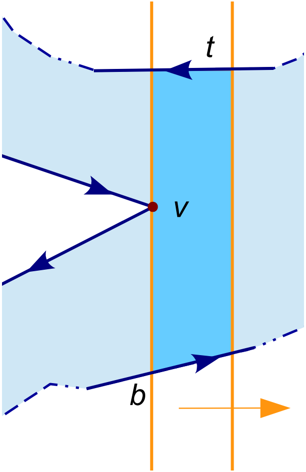
Invariant

- *SPLIT* vertices to the left of the sweep line: diagonal added
- *MERGE* vertices to the left of the sweep line: ...
- diagonal added if and only if a second vertex of P falls in the trapezoid between edges b and t
- $v = helper(b)$



Invariant

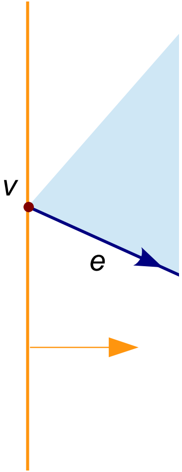
- *SPLIT* vertices to the left of the sweep line: diagonal added
- *MERGE* vertices to the left of the sweep line: ...
- diagonal added if and only if a second vertex of P falls in the trapezoid between edges b and t
- $v = helper(b)$



Three main operations

insert(e) :

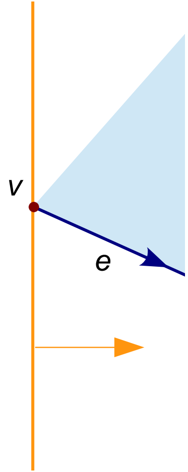
- insert lower boundary edge e into the *sweep-line* structure
- *helper*(e) $:=$ v



Three main operations

insert(e) :

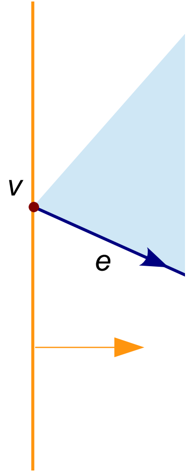
- insert lower boundary edge e into the *sweep-line* structure
- *helper*(e) $:=$ v



Three main operations

insert(*e*) :

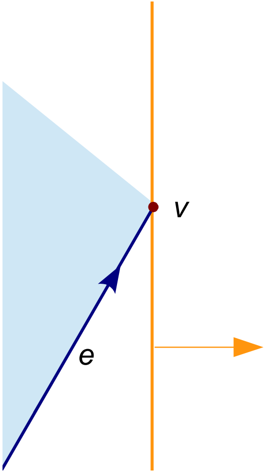
- insert lower boundary edge *e* into the *sweep-line* structure
- *helper*(*e*) $:=$ *v*



Three main operations

remove(e) :

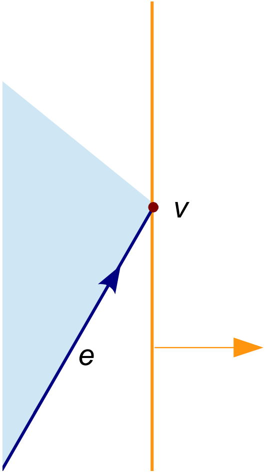
- $u \ := \ \textit{helper}(\ e\ )$
- if $\ \textit{type}(u) = \textit{MERGE}\ $ then
  add diagonal $\ uv$
- remove lower boundary edge e from the *sweep-line* structure



Three main operations

remove(e) :

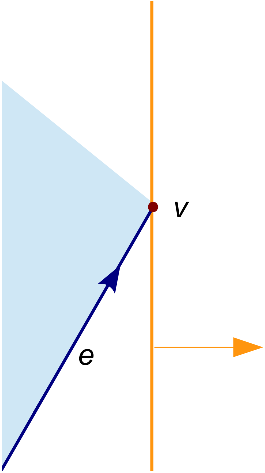
- $u \;:=\; \textit{helper}(\; e\;)$
- if $\textit{type}(u) = \textit{MERGE}$ then
  add diagonal uv
- remove lower boundary edge e from the *sweep-line* structure



Three main operations

remove(*e*) :

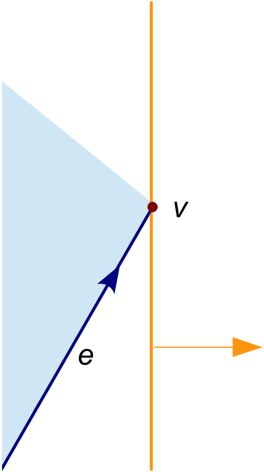
- u := *helper*(*e*)
- if $type(u) = MERGE$ then
  add diagonal uv
- remove lower boundary edge *e* from the *sweep-line* structure



Three main operations

remove(e) :

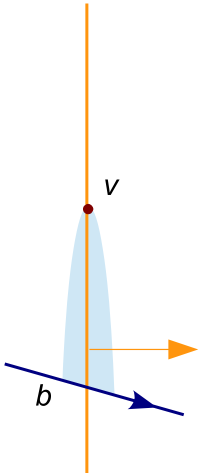
- $u \ := \ helper(\ e\ )$
- if $type(u) = MERGE$ then
  add diagonal uv
- remove lower boundary edge e from the *sweep-line* structure



Three main operations

process(v) :

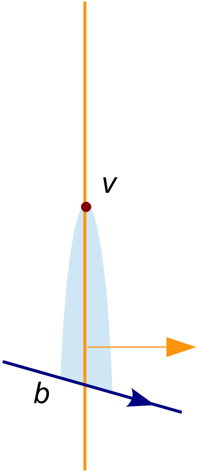
- v upper boundary vertex just above edge b
- $u \ := \ helper(\ b\ )$
- if either $type(u) = MERGE$
  or $type(v) = SPLIT$ then
  add diagonal uv
- $helper(\ b\ ) \ := \ v$



Three main operations

process(v) :

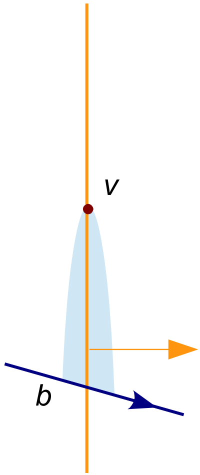
- v upper boundary vertex just above edge b
- $u \;:=\; helper(\; b\;)$
- if either $type(u) = MERGE$ or $type(v) = SPLIT$ then add diagonal uv
- $helper(\; b\;) \;:=\; v$



Three main operations

process(v) :

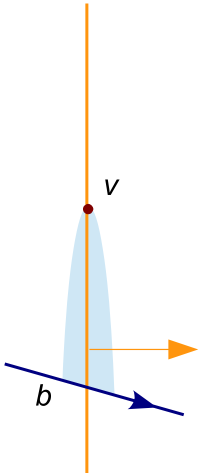
- v upper boundary vertex just above edge b
- $u \;:=\; \mathit{helper}(\ b\ )$
- if either $\mathit{type}(u) = \mathit{MERGE}$ or $\mathit{type}(v) = \mathit{SPLIT}$ then add diagonal uv
- $\mathit{helper}(\ b\ ) \;:=\; v$



Three main operations

process(v) :

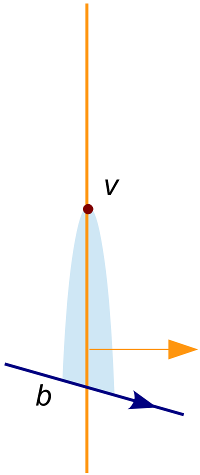
- v upper boundary vertex just above edge b
- $u \;:=\; \textit{helper}(\, b \,)$
- if either $\textit{type}(u) = \textit{MERGE}$ or $\textit{type}(v) = \textit{SPLIT}$ then add diagonal uv
- $\textit{helper}(\, b \,) \;:=\; v$



Three main operations

process(v) :

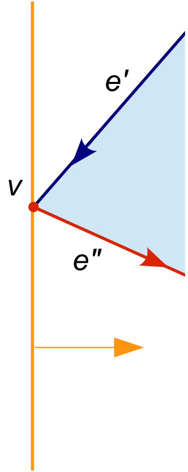
- v upper boundary vertex just above edge b
- $u \;:=\; helper(\, b\,)$
- if either $type(u) = MERGE$ or $type(v) = SPLIT$ then add diagonal uv
- $helper(\, b\,) \;:=\; v$



Event processing

START event :

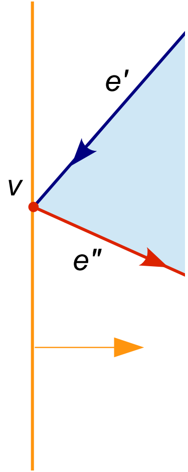
- *insert*(e'')



Event processing

START event :

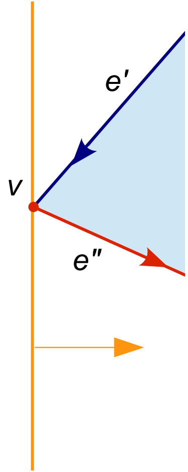
- *insert*(e'')



Event processing

START event :

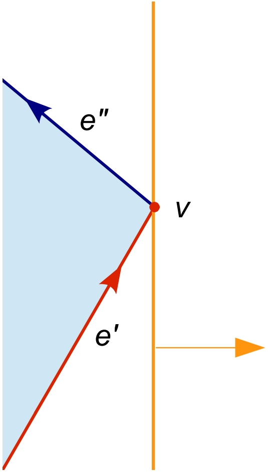
- *insert*(e'')
  - insert lower boundary edge e'' into the *sweep-line* structure
  - *helper*(e'') $:=$ v



Event processing

END event :

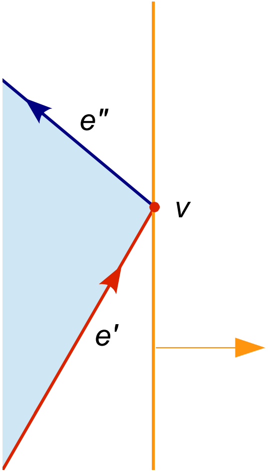
- *remove*(e')



Event processing

END event :

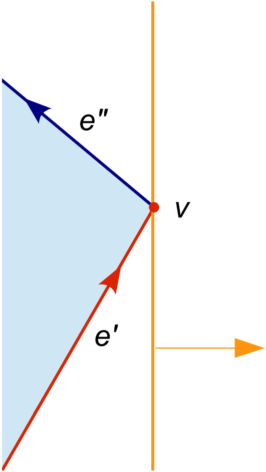
- *remove*(e')



Event processing

END event :

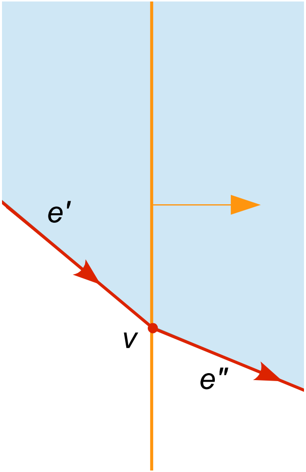
- *remove*(e')
  - u := *helper*(e')
  - if $type(u) = MERGE$ then
    add diagonal uv
  - remove lower boundary edge e' from the *sweep-line* structure



Event processing

LOWER_REGULAR event :

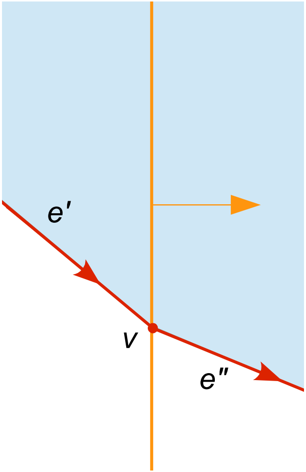
- *remove*(e')
- *insert*(e'')



Event processing

LOWER_REGULAR event :

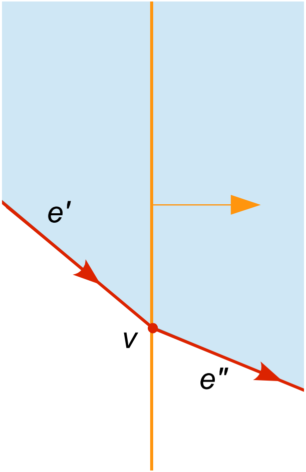
- *remove*(e')
- *insert*(e'')



Event processing

LOWER_REGULAR event :

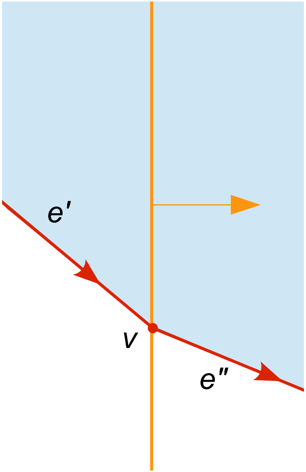
- *remove*(e')
- *insert*(e'')



Event processing

LOWER_REGULAR event :

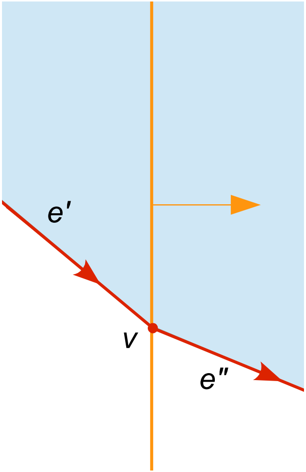
- *remove*(e')
  - $u \ := \ helper(\ e'\ )$
  - if $type(u) = MERGE$ then
    add diagonal uv
  - remove lower boundary edge e' from the *sweep-line* structure
- *insert*(e'')



Event processing

LOWER_REGULAR event :

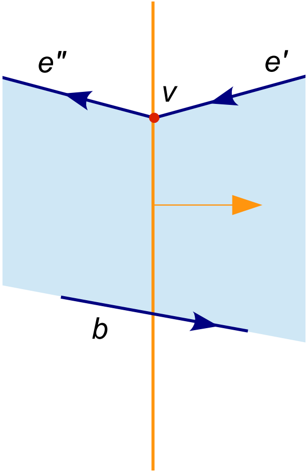
- *remove*(e')
- *insert*(e'')
  - insert lower boundary edge e'' into the *sweep-line* structure
  - *helper*(e'') $:=$ v



Event processing

UPPER_REGULAR event :

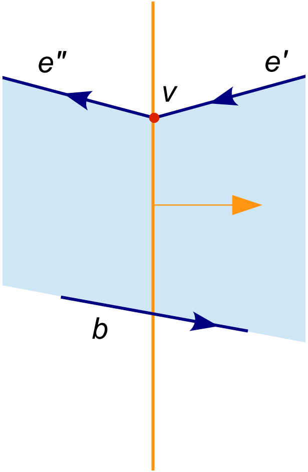
- *process(v)*



Event processing

UPPER_REGULAR event :

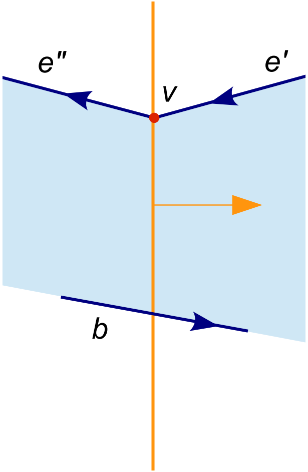
- *process*(v)



Event processing

UPPER_REGULAR event :

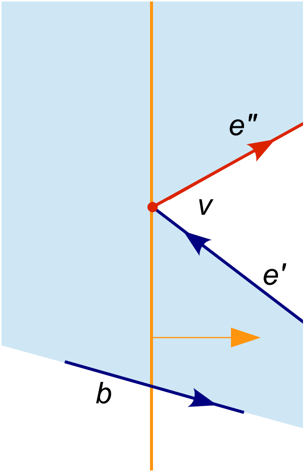
- *process*(v)
  - u := *helper*(b)
  - if *type*(u) = *MERGE* then
    // *type*(v) ≠ *SPLIT*
    add diagonal uv
  - *helper*(b) := v



Event processing

SPLIT event :

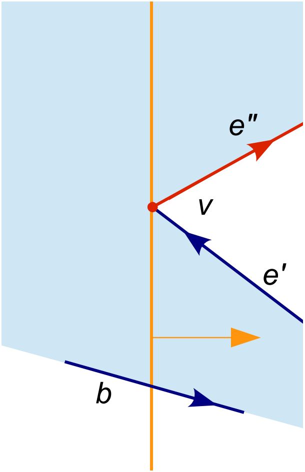
- *process*(v)
- *insert*(e'')



Event processing

SPLIT event :

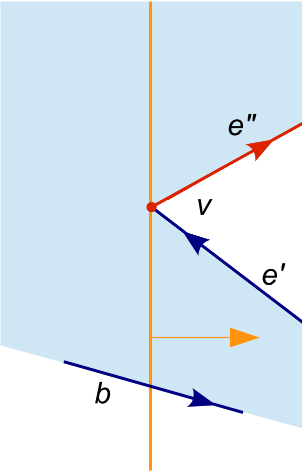
- *process*(v)
- *insert*(e'')



Event processing

SPLIT event :

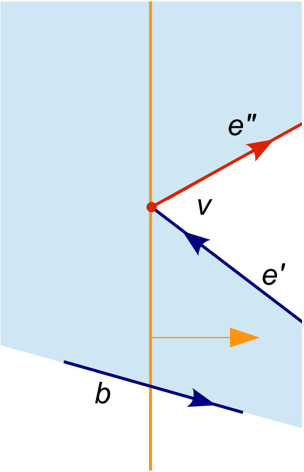
- *process*(v)
- *insert*(e'')



Event processing

SPLIT event :

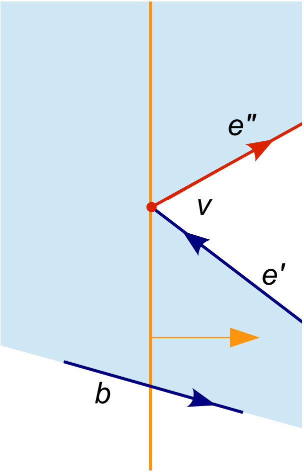
- *process*(v)
  - $u \ := \ helper(\ b\ )$
  - // $type(v) = SPLIT$
    add diagonal uv
  - $helper(\ b\ ) \ := \ v$
- *insert*(e'')



Event processing

SPLIT event :

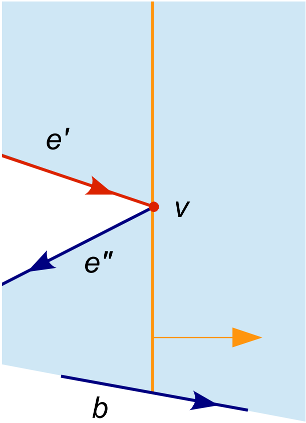
- *process*(v)
- *insert*(e'')
    - insert lower boundary edge e'' into the *sweep-line* structure
    - *helper*(e'') $:=$ v



Event processing

MERGE event :

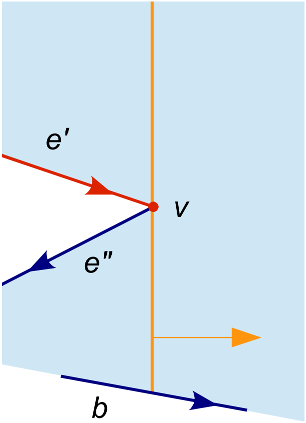
- *remove*(e')
- *process*(v)



Event processing

MERGE event :

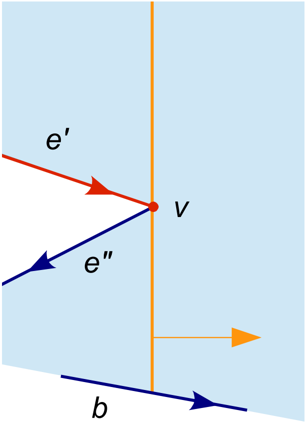
- *remove*(e')
- *process*(v)



Event processing

MERGE event :

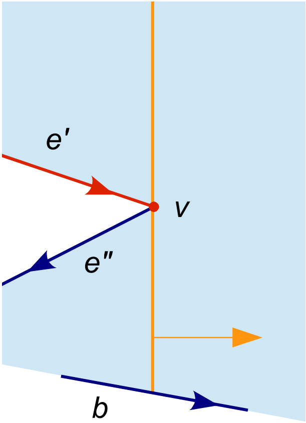
- *remove*(e')
- *process*(v)



Event processing

MERGE event :

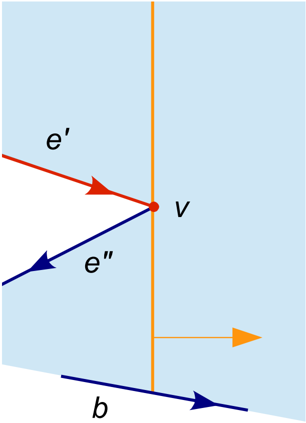
- *remove*(e')
  - $u \;:=\; helper(\; e' \;)$
  - if $type(u) = MERGE$ then
    add diagonal uv
  - remove lower boundary edge e' from the *sweep-line* structure
- *process*(v)



Event processing

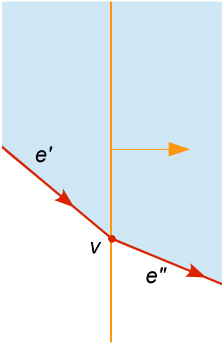
MERGE event :

- *remove*(e')
- *process*(v)
  - u := *helper*(b)
  - if *type*(u) = *MERGE* then
    // *type*(v) ≠ *SPLIT*
    add diagonal uv
  - *helper*(b) := v

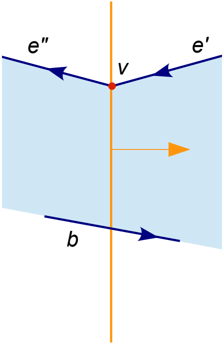


Identifying vertex type: lower/upper *REGULAR*

e', e'' on the opposite sides of the sweep line



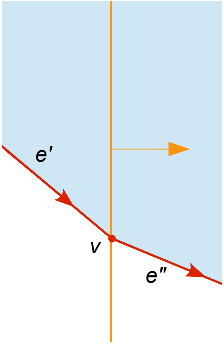
e' on the left side



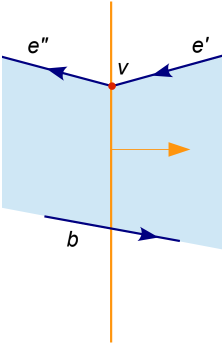
e'' on the left side

Identifying vertex type: lower/upper *REGULAR*

e', e'' on the opposite sides of the sweep line



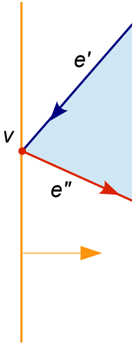
e' on the left side



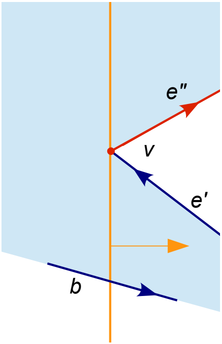
e'' on the left side

Identifying vertex type: *START*, *SPLIT*

e', e'' both to the right of the sweep line



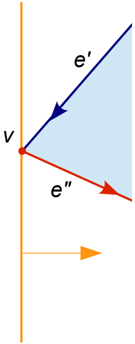
left turn



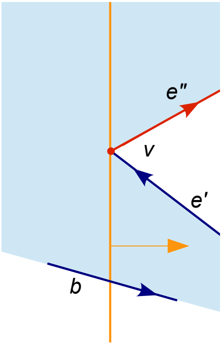
right turn

Identifying vertex type: *START*, *SPLIT*

e', e'' both to the right of the sweep line



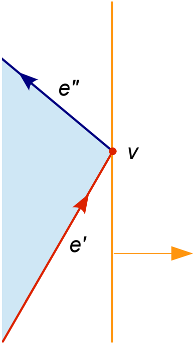
left turn



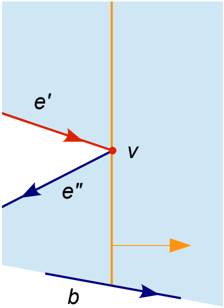
right turn

Identifying vertex type: *END*, *MERGE*

e', e'' both to the left of the sweep line



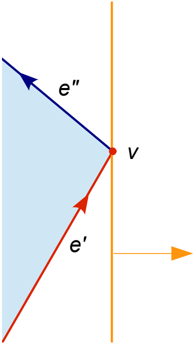
left turn



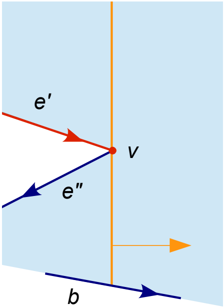
right turn

Identifying vertex type: *END*, *MERGE*

e', e'' both to the left of the sweep line



left turn



right turn

Historical note

- The idea essentially goes back to Lee & Preparata (1977)
- Aiming at “regularizing” a planar subdivision into y-monotone components
- Plane-sweep *descending* pass: “incoming” diagonals for *SPLIT* vertices
- Plane-sweep *ascending* pass: “outcoming” diagonals for *MERGE* vertices

Historical note

- The idea essentially goes back to Lee & Preparata (1977)
- Aiming at “regularizing” a planar subdivision into y-monotone components
- Plane-sweep *descending* pass: “incoming” diagonals for *SPLIT* vertices
- Plane-sweep *ascending* pass: “outcoming” diagonals for *MERGE* vertices

Historical note

- The idea essentially goes back to Lee & Preparata (1977)
- Aiming at “regularizing” a planar subdivision into y-monotone components
- Plane-sweep *descending* pass: “incoming” diagonals for *SPLIT* vertices
- Plane-sweep *ascending* pass: “outcoming” diagonals for *MERGE* vertices

Historical note

- The idea essentially goes back to Lee & Preparata (1977)
- Aiming at “regularizing” a planar subdivision into y-monotone components
- Plane-sweep *descending* pass: “incoming” diagonals for *SPLIT* vertices
- Plane-sweep *ascending* pass: “outcoming” diagonals for *MERGE* vertices

Plane sweep + DCEL

- Easy access to the x-monotone subpolygons: DCEL
- Cross-pointers between DCEL edges and corresponding edges in the sweep-line structure
- Diagonal inserted in constant time
  provided the treatment of faces is delayed to the end

Plane sweep + DCEL

- Easy access to the x-monotone subpolygons: DCEL
- Cross-pointers between DCEL edges and corresponding edges in the sweep-line structure
- Diagonal inserted in constant time provided the treatment of faces is delayed to the end

Plane sweep + DCEL

- Easy access to the x-monotone subpolygons: DCEL
- Cross-pointers between DCEL edges and corresponding edges in the sweep-line structure
- Diagonal inserted in constant time
  provided the treatment of faces is delayed to the end

Plane sweep + DCEL

- Easy access to the x-monotone subpolygons: DCEL
- Cross-pointers between DCEL edges and corresponding edges in the sweep-line structure
- Diagonal inserted in constant time provided the treatment of faces is delayed to the end

Soundness of the subdivision

- No *SPLIT* and *MERGE* vertices
- Hence: x-monotone subpolygons (see above property)
- But may diagonals cross each other?
- Consider possible cases and *helper*'s role

Soundness of the subdivision

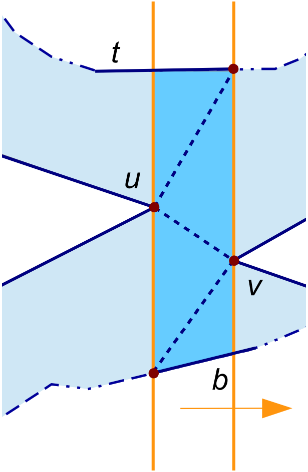
- No *SPLIT* and *MERGE* vertices
- Hence: x-monotone subpolygons (see above property)
- But may diagonals cross each other?
- Consider possible cases and *helper*'s role

Soundness of the subdivision

- No *SPLIT* and *MERGE* vertices
- Hence: x-monotone subpolygons (see above property)
- But may diagonals cross each other?
- Consider possible cases and *helper*'s role

Soundness of the subdivision

- No *SPLIT* and *MERGE* vertices
- Hence: x-monotone subpolygons (see above property)
- But may diagonals cross each other?
- Consider possible cases and *helper*'s role



Computational costs

- Event processing steps: n
- Plane-sweep processing: $O(\ n \log n\ )$
  (event queue, sweep-line structure)
- Specific operations: $O(\ 1\ )$ per step
  (adding diagonals, accessing/updating DCEL — but faces)
- Overall: $O(\ n \log n\ )$ running time and $O(\ n\ )$ storage

Computational costs

- Event processing steps: n
- Plane-sweep processing: $O(\ n \log n\ )$
  (event queue, sweep-line structure)
- Specific operations: $O(\ 1\ )$ per step
  (adding diagonals, accessing/updating DCEL — but faces)
- Overall: $O(\ n \log n\ )$ running time and $O(\ n\ )$ storage

Computational costs

- Event processing steps: n
- Plane-sweep processing: $O(\ n \log n\ )$
  (event queue, sweep-line structure)
- Specific operations: $O(\ 1\ )$ per step
  (adding diagonals, accessing/updating DCEL — but faces)
- Overall: $O(\ n \log n\ )$ running time and $O(\ n\ )$ storage

Computational costs

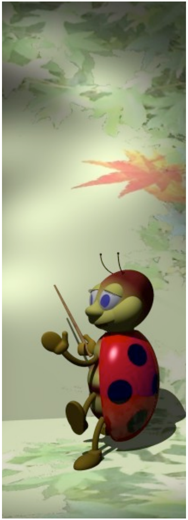
- Event processing steps: n
- Plane-sweep processing: $O(\ n \log n\ )$
  (event queue, sweep-line structure)
- Specific operations: $O(\ 1\ )$ per step
  (adding diagonals, accessing/updating DCEL — but faces)
- Overall: $O(\ n \log n\ )$ running time and $O(\ n\ )$ storage

Computational costs

- Event processing steps: n
- Plane-sweep processing: $O(\ n \log n\ )$
  (event queue, sweep-line structure)
- Specific operations: $O(\ 1\ )$ per step
  (adding diagonals, accessing/updating DCEL — but faces)
- Overall: $O(\ n \log n\ )$ running time and $O(\ n\ )$ storage

Outline

1. Triangulating a simple polygon
   - definitions
   - existence of a triangulation
   - art gallery problem
2. Monotone partition
   - monotonicity
   - plane sweep
   - analysis
3. Triangulating a monotone polygon
   - invariant arrangement
   - computation costs



Approach to triangulating a monotone (sub)polygon

- Vertices are processed left to right
- Lower vs. upper half-boundary
- “Greedy” approach:
  diagonals are added whenever possible
- Auxiliary *stack*:
  pending vertices, from both lower and upper boundary

Approach to triangulating a monotone (sub)polygon

- Vertices are processed left to right
- Lower vs. upper half-boundary
- “Greedy” approach:
  diagonals are added whenever possible
- Auxiliary *stack*:
  pending vertices, from both lower and upper boundary

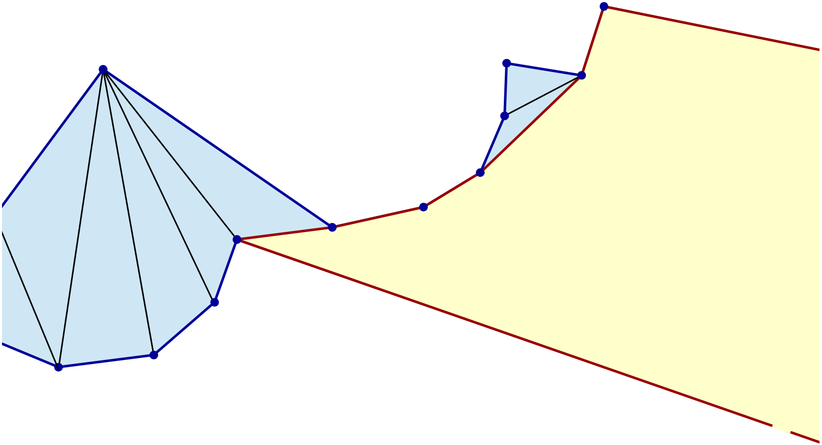
Approach to triangulating a monotone (sub)polygon

- Vertices are processed left to right
- Lower vs. upper half-boundary
- “Greedy” approach:
  diagonals are added whenever possible
- Auxiliary *stack*:
  pending vertices, from both lower and upper boundary

Approach to triangulating a monotone (sub)polygon

- Vertices are processed left to right
- Lower vs. upper half-boundary
- “Greedy” approach:
  diagonals are added whenever possible
- Auxiliary *stack*:
  pending vertices, from both lower and upper boundary

“Funnel”-shaped area to be triangulated



Pending vertices in the stack

- *One* vertex on a half-boundary
- Chain of *reflex* vertices + last visited vertex on the opposite half-boundary
- Starting with the first two vertices in the stack
- Next vertex to be considered may be on either side. . .

Pending vertices in the stack

- *One* vertex on a half-boundary
- Chain of *reflex* vertices + last visited vertex on the opposite half-boundary
- Starting with the first two vertices in the stack
- Next vertex to be considered may be on either side. . .

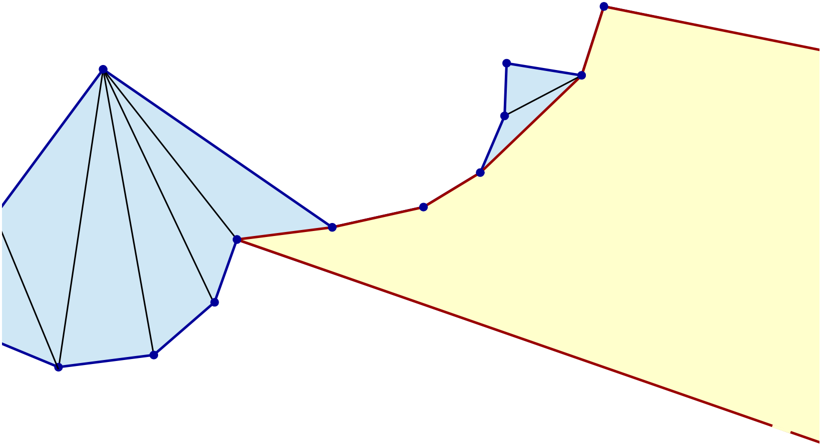
Pending vertices in the stack

- *One* vertex on a half-boundary
- Chain of *reflex* vertices + last visited vertex on the opposite half-boundary
- Starting with the first two vertices in the stack
- Next vertex to be considered may be on either side. . .

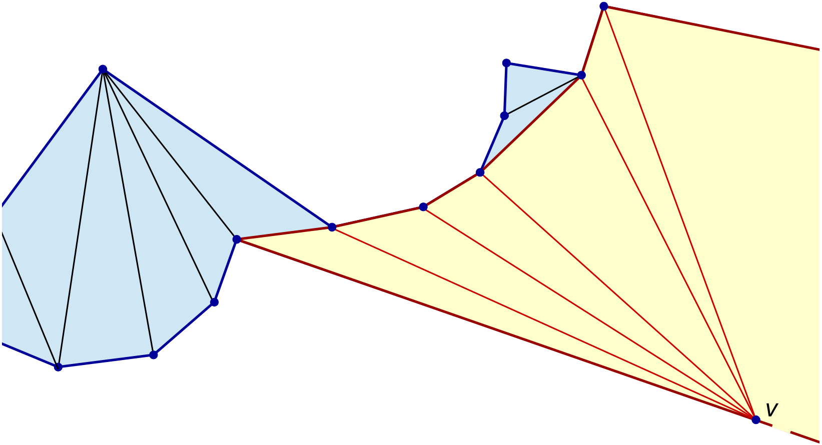
Pending vertices in the stack

- *One* vertex on a half-boundary (e.g. lower boundary)
- Chain of *reflex* vertices + last visited vertex
  on the opposite half-boundary (e.g. upper boundary)
- Starting with the first two vertices in the stack
- Next vertex to be considered may be on either side. . .

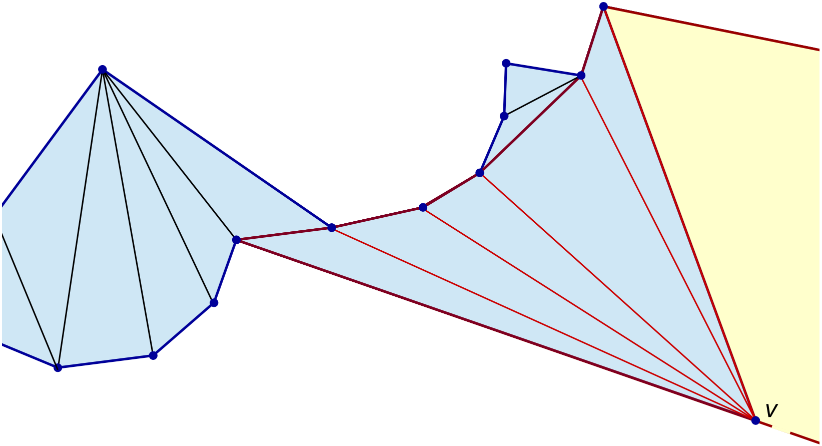
“Funnel”-shaped area to be triangulated. . .



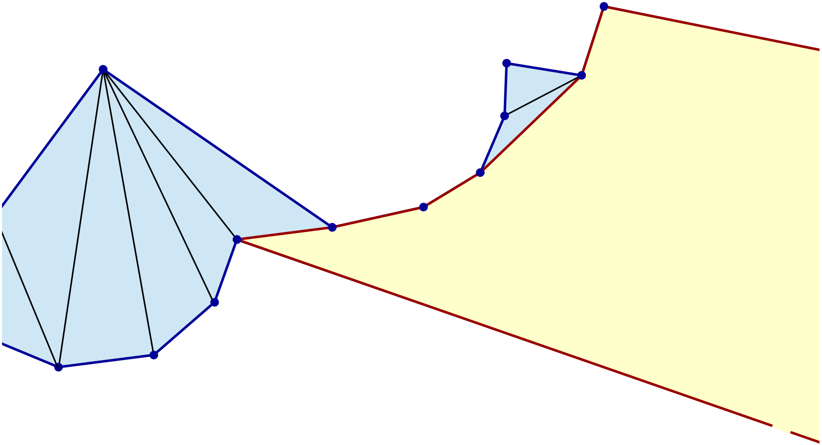
Next vertex opposite to the chain



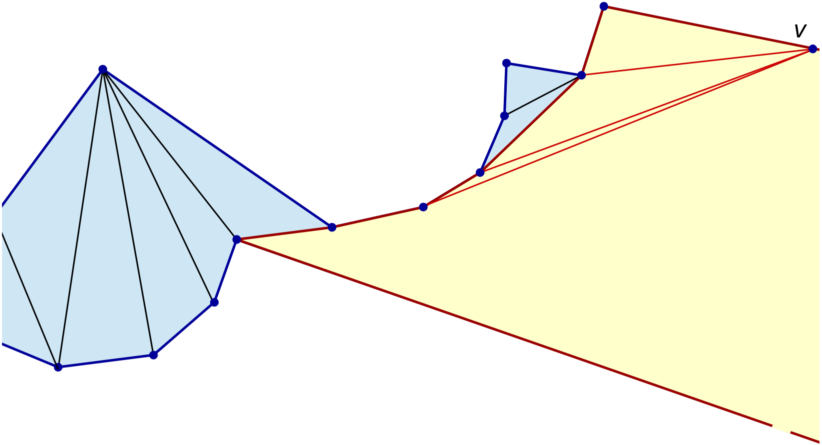
Invariant “funnel” arrangement



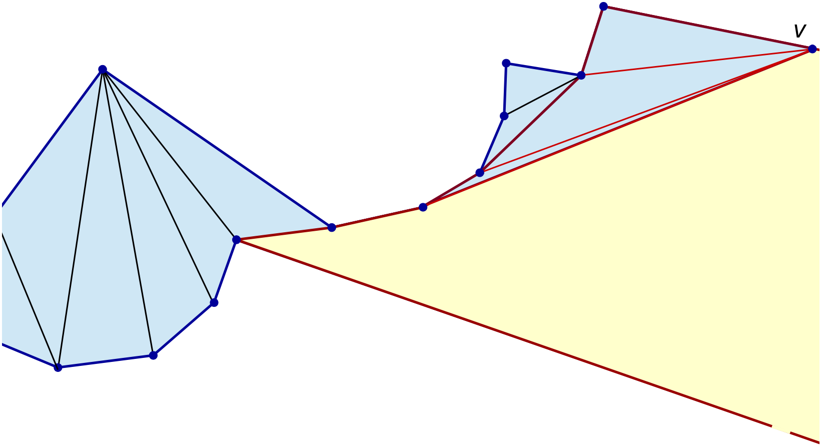
“Funnel”-shaped area to be triangulated. . .



Next vertex following the chain



Invariant "funnel" arrangement



Running costs

- Sorting vertices left to right: $O(\ n\ )$
  (from boundary items in counterclockwise order)
- Iterations to process next vertex: $O(\ n\ )$
- At each iteration
  at most two vertices are (re-)pushed onto stack
- Overall operations on stack and stacked vertices: $O(\ n\ )$

Running costs

- Sorting vertices left to right: $O(\ n\ )$
  (from boundary items in counterclockwise order)
- Iterations to process next vertex: $O(\ n\ )$
- At each iteration
  at most two vertices are (re-)pushed onto stack
- Overall operations on stack and stacked vertices: $O(\ n\ )$

Running costs

- Sorting vertices left to right: $O(\ n\ )$
  (from boundary items in counterclockwise order)
- Iterations to process next vertex: $O(\ n\ )$
- At each iteration
  at most two vertices are (re-)pushed onto stack
- Overall operations on stack and stacked vertices: $O(\ n\ )$

Running costs

- Sorting vertices left to right: $O(\ n\ )$
  (from boundary items in counterclockwise order)
- Iterations to process next vertex: $O(\ n\ )$
- At each iteration
  at most two vertices are (re-)pushed onto stack
- Overall operations on stack and stacked vertices: $O(\ n\ )$

To sum up. . .

- Partitioning a simple polygon into monotone subpolygons: $O(\ n \log n\ )$
- Triangulating all monotone subpolygons: $O(\ n\ )$
- Triangulating a simple polygon: $O(\ n \log n\ )$
- Storage requirements: $O(\ n\ )$

To sum up...

- Partitioning a simple polygon into monotone subpolygons: $O(n \log n)$
- Triangulating all monotone subpolygons: $O(n)$
- Triangulating a simple polygon: $O(n \log n)$
- Storage requirements: $O(n)$

To sum up...

- Partitioning a simple polygon
  into monotone subpolygons: $O(\ n \log n\ )$
- Triangulating all monotone subpolygons: $O(\ n\ )$
- Triangulating a simple polygon: $O(\ n \log n\ )$
- Storage requirements: $O(\ n\ )$

To sum up. . .

- Partitioning a simple polygon
  into monotone subpolygons: $O(\ n \log n\ )$
- Triangulating all monotone subpolygons: $O(\ n\ )$
- Triangulating a simple polygon: $O(\ n \log n\ )$
- Storage requirements: $O(\ n\ )$

Further remarks

- What about polygons with *holes*?
- The assumption that there are no holes was never used!
- More in general, essentially the same algorithm works for any planar subdivision within a bounding box
- Triangulating a planar subdivision:
  $O(\,n \log n\,)$ running time and $O(\,n\,)$ storage

Further remarks

- What about polygons with *holes*?
- The assumption that there are no holes was never used!
- More in general, essentially the same algorithm works for any planar subdivision within a bounding box
- Triangulating a planar subdivision: $O(\ n \log n\ )$ running time and $O(\ n\ )$ storage

Further remarks

- What about polygons with *holes*?
- The assumption that there are no holes was never used!
- More in general, essentially the same algorithm works for any planar subdivision within a bounding box
- Triangulating a planar subdivision: $O(\ n \log n\ )$ running time and $O(\ n\ )$ storage

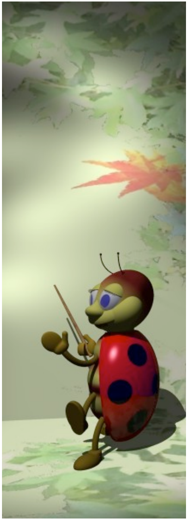
Further remarks

- What about polygons with *holes*?
- The assumption that there are no holes was never used!
- More in general, essentially the same algorithm works for any planar subdivision within a bounding box
- Triangulating a planar subdivision:
  $O(\, n \log n \,)$ running time and $O(\, n \,)$ storage

Outline

4 Related results

5 References



Related results

- Optimal algorithm for triangulating a simple polygon (Chazelle, 1990 & 1991): $O(\ n\ )$
- Tetrahedralization of a simple polytope in 3D may require $\Theta(\ n^2\ )$ additional vertices!
- There are indeed polyhedra which cannot be decomposed into fewer than $\Omega(\ n^2\ )$ *convex* parts (Chazelle, 1984)

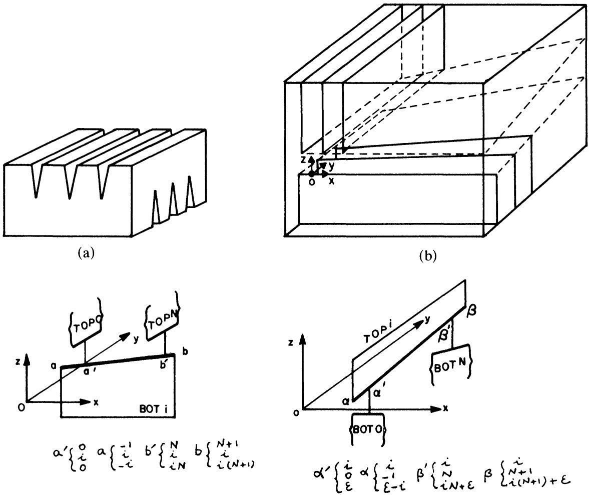
Related results

- Optimal algorithm for triangulating a simple polygon (Chazelle, 1990 & 1991): $O(\ n\ )$
- Tetrahedralization of a simple polytope in 3D may require $\Theta(\ n^2\ )$ additional vertices!
- There are indeed polyhedra which cannot be decomposed into fewer than $\Omega(\ n^2\ )$ *convex* parts (Chazelle, 1984)

Related results

- Optimal algorithm for triangulating a simple polygon (Chazelle, 1990 & 1991): $O(\ n\ )$
- Tetrahedralization of a simple polytope in 3D may require $\Theta(\ n^2\ )$ additional vertices!
- There are indeed polyhedra which cannot be decomposed into fewer than $\Omega(\ n^2\ )$ *convex* parts (Chazelle, 1984)

Chazelle, 1984



(c)

Chazelle, 1984

3.3. Decomposing *P* into convex parts. We define Σ as the portion of P comprised between the two hyperbolic paraboloids $z = xy$ and $z = xy + \varepsilon$ and the four planes $x = 0$, $x = N$, $y = 0$, $y = N$. Σ is a cylinder parallel to the z-axis, of height ε, whose base is the region of the hyperbolic paraboloid $z = xy$ with $0 \leqq x, y \leqq N$ (Fig. 5). Let $Q_1, \cdots, Q_m$ be any convex decomposition of P and let Q_i^* denote the intersection of Q_i and Σ. Since Σ lies inside P, the set of Q_i^* forms a partition of Σ. Note that Q_i^* may consist of $0, 1$, or several blocks, most of which are likely not to be polyhedra. Our goal is to prove that $m \geqq cN^2$ for some constant c, by showing that the volume of Q_i^* cannot be too large. By volume of Q_i^*, we mean the sum of all the volumes of the blocks composing Q_i^*. We first characterize the shape and the orientation of the large Q_i^*'s, which permits us to derive an upper bound on their maximum volume.

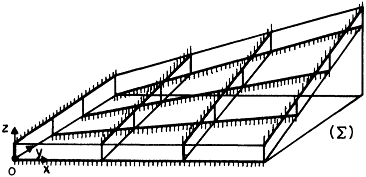
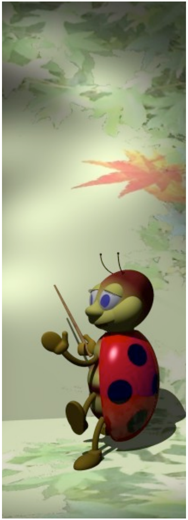


FIG. 5. *The warped region* Σ.

Outline

4 Related results

5 References



References

D.T. Lee & F.P. Preparata (1977)
Location of a Point in a Planar Subdivision
and Its Applications
SIAM Journal on Computing, 6(3)

B. Chazelle (1991)
Triangulating a Simple Polygon in Linear Time
Discrete & Computational Geometry, 6(5)

B. Chazelle (1984)
Convex Partitions of Polyhedra:
A Lower Bound and Worst-case Optimal Algorithm
SIAM Journal on Computing, 13(3)