Computational Geometry

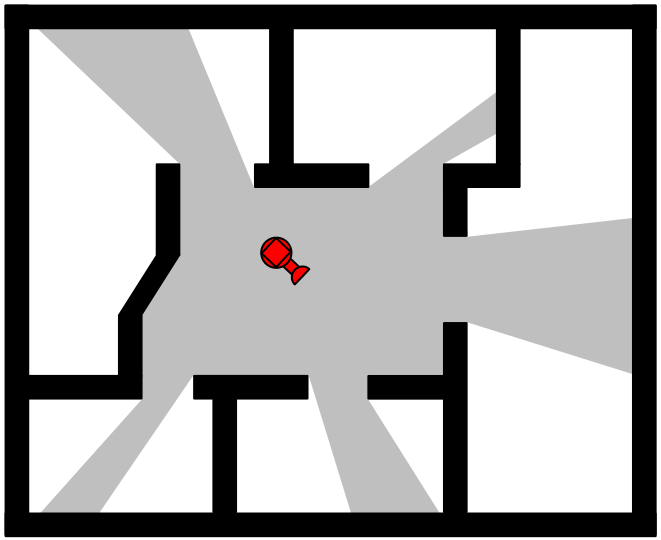
7. Polygon Triangulation

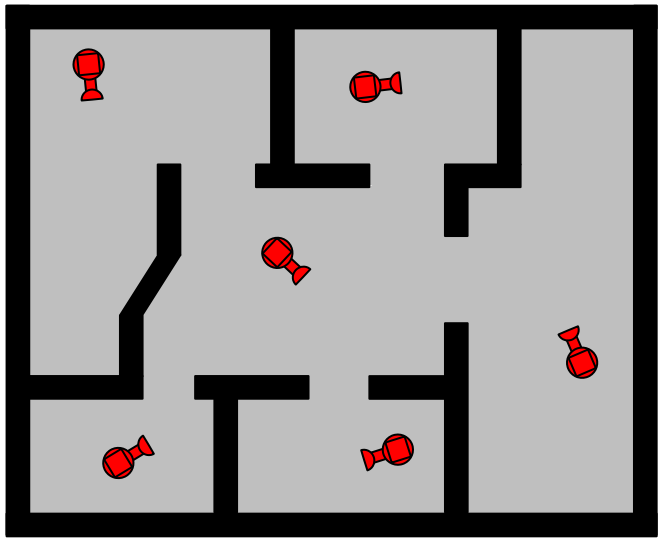
7.1 The Art-Gallery Problem

A museum shall be equipped with cameras, each can monitor an angle of 360°, covering the whole area of an art gallery.

The *Art-Gallery-Problem* is, to determine

- the smallest number of cameras, and
- where these cameras should be placed.





7.1 The Art-Gallery Problem

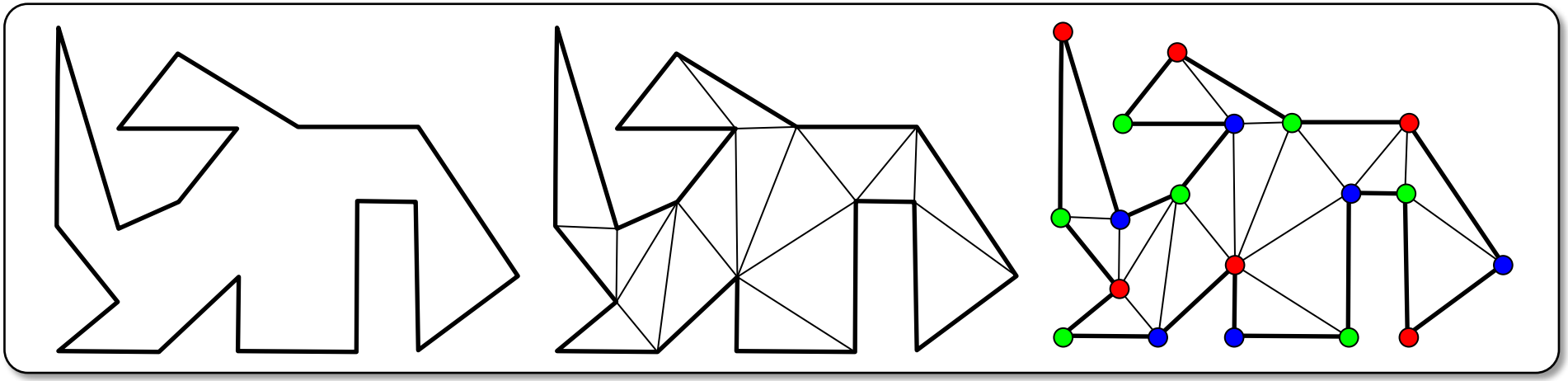
- The area of the gallery is assumed to be a simple polygon P (i.e., a single component without holes) with n vertices, where adjacent vertices are not collinear.
- To determine the minimal number of cameras and their positions is NP-hard for arbitrary polygons.

- **But:** For a simple polygon P with n vertices there is an upper bound on the number of cameras: $\lfloor n/3 \rfloor$.
- **Idea:** Partitioning the polygon into triangles (*triangulation*) can solve this problem efficiently.

7.1 The Art-Gallery Problem

Idea

- Triangulate the polygon.
- Construct a 3-coloring (red, green, blue) for the vertices, such that the vertices of a triangle have different colors.
- Place cameras at the vertices with the least frequent color.
  - ► This color occurs in every triangle exactly once.



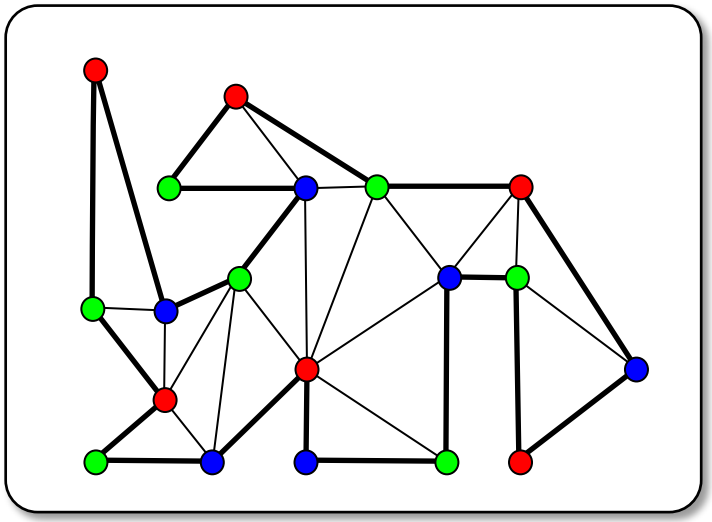
7.1 The Art-Gallery Problem

Questions

a) Is it possible to partition every simple polygon without self-intersections into triangles?

b) Is there always a 3-coloring of the vertices?

c) Are both efficiently computable?



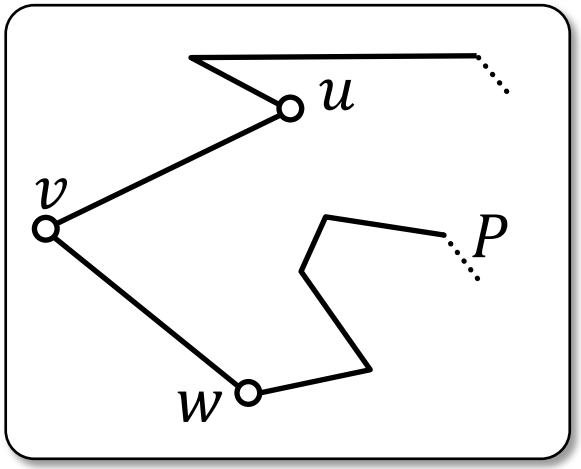
7.1 The Art-Gallery Problem

Proposition 1 (Answer to question a))

Every simple polygon with n vertices has a triangulation and every triangulation has exactly $n-2$ triangles.

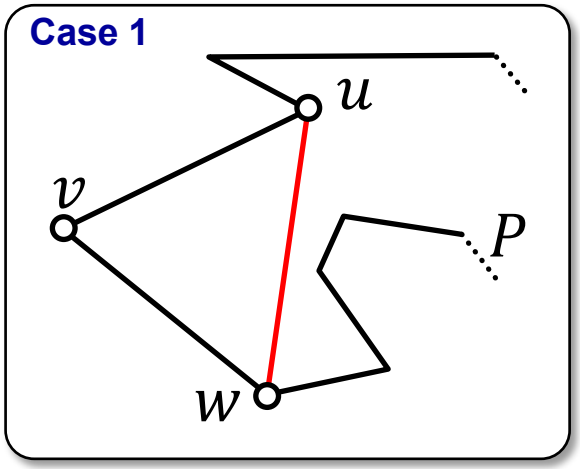
Proof (Induction on n)

- **Base case:** $n = 3$.
- **Induction hypothesis:** Prop. 1 holds for all $m < n, n > 3$.
- **Inductive step:**
  - Let P be a polygon with n vertices, and
  - u, v, w a sequence of **adjacent** vertices, where v has the smallest x-coordinate of P (and among those the smallest y-coordinate).



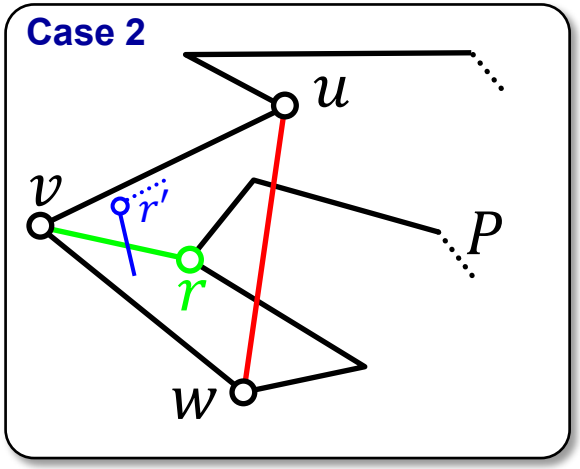
7.1 The Art-Gallery Problem

Case 1: If the line segment $\overline{uw}$ lies in the interior of P, the polygon can be subdivided along $\overline{uw}$.



Case 2: Otherwise, P has at least one vertex in the interior of the triangle uvw.

- Chose among those the vertex r with maximal distance to the edge $\overline{uw}$.
- Then $\overline{vr}$ is in the interior of P.
  - If $\overline{vr}$ were intersected by P, there would be a vertex r' with larger distance to $\overline{uw}$.



7.1 The Art-Gallery Problem

- Subdivide the polygon along $\overline{uw}$ or $\overline{vr}$ into two polygons with m_1 und m_2 vertices ($m_1 + m_2 = n + 2$)
- From the induction hypothesis both can be triangulated.
- The total number of triangles is

$$(m_1 - 2) + (m_2 - 2) = n - 2.$$

□

Remarks

- Placing the cameras at inner edges, every camera can monitor **two** triangles.
- Placing the cameras at the vertices, every camera can monitor **at least two** triangles.

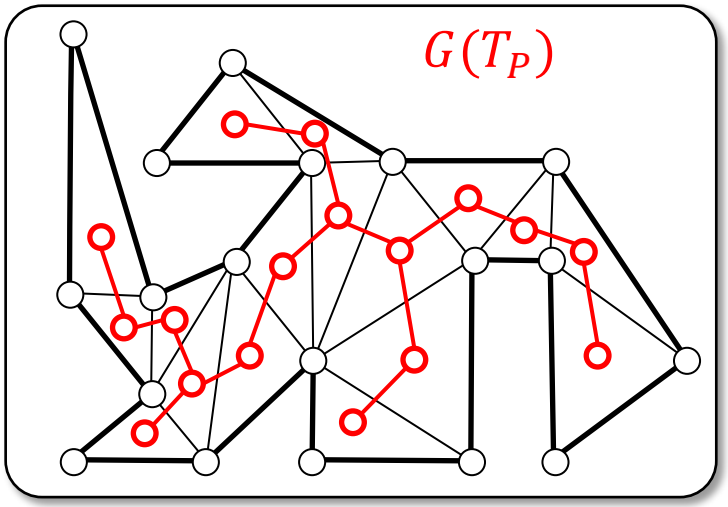
7.1 The Art-Gallery Problem

Proposition 2 (Art-Gallery-Theorem) (Answer to question b))

For a simple polygon P with n vertices, $\lfloor n/3 \rfloor$ cameras are sufficient and, in some cases, also necessary, such that every inner point is visible from at least one camera

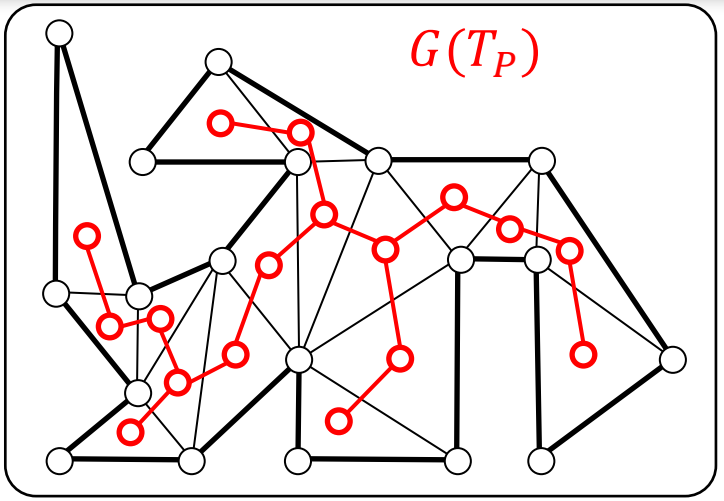
Proof

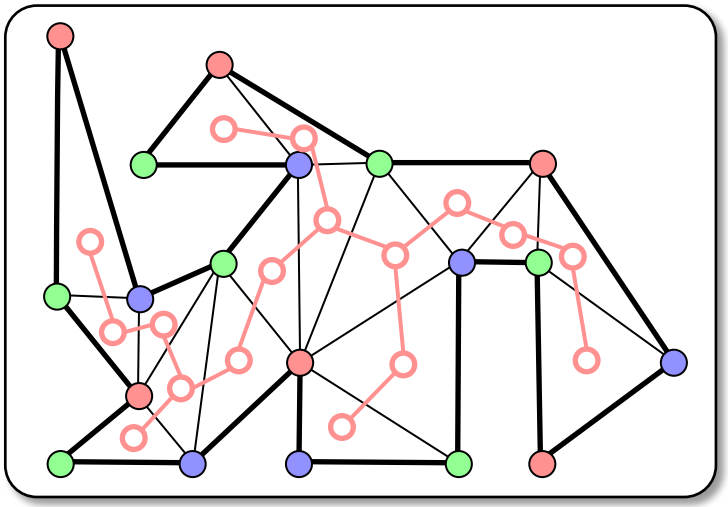
1. First prove that a 3-coloring of the triangulation T_P of P always exists.
   - The dual graph $G(T_P)$ of T_P has one node for every triangle in T_P.
   - The nodes of edge-adjacent triangles are connected by an edge in $G(T_P)$.



7.1 The Art-Gallery Problem

- Each edge of $G(T_P)$ intersects one inner edge of T_P, which partitions the polygon P in two separate sub-polygons.
- Thus, $G(T_P)$ does not have cycles, i.e. it is a tree.
- For the 3-coloring start with an arbitrary triangle and follow recursively the edges of $G(T_P)$.
- For every triangle there remains exactly one vertex to color, so that its color is determined uniquely.



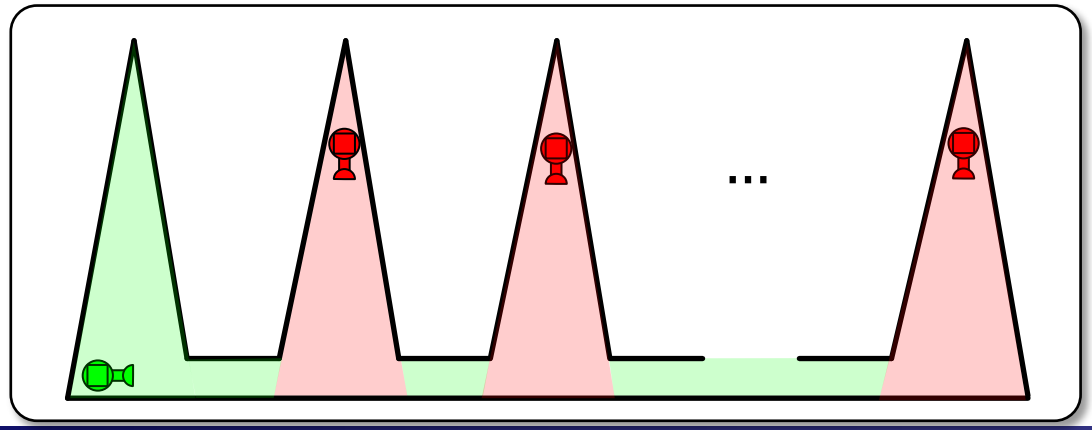


7.1 The Art-Gallery Problem

2. The number of vertices with the least frequent color is at most $\lfloor n/3 \rfloor$.

   Because every triangle has one vertex of this color, $\lfloor n/3 \rfloor$ cameras are sufficient.

3. $n/3$ cameras are also necessary, because there is a polygon with k peaks ($n = 3k$), where each peak is visible from one camera only:



□

7.2 Monotone partitioning

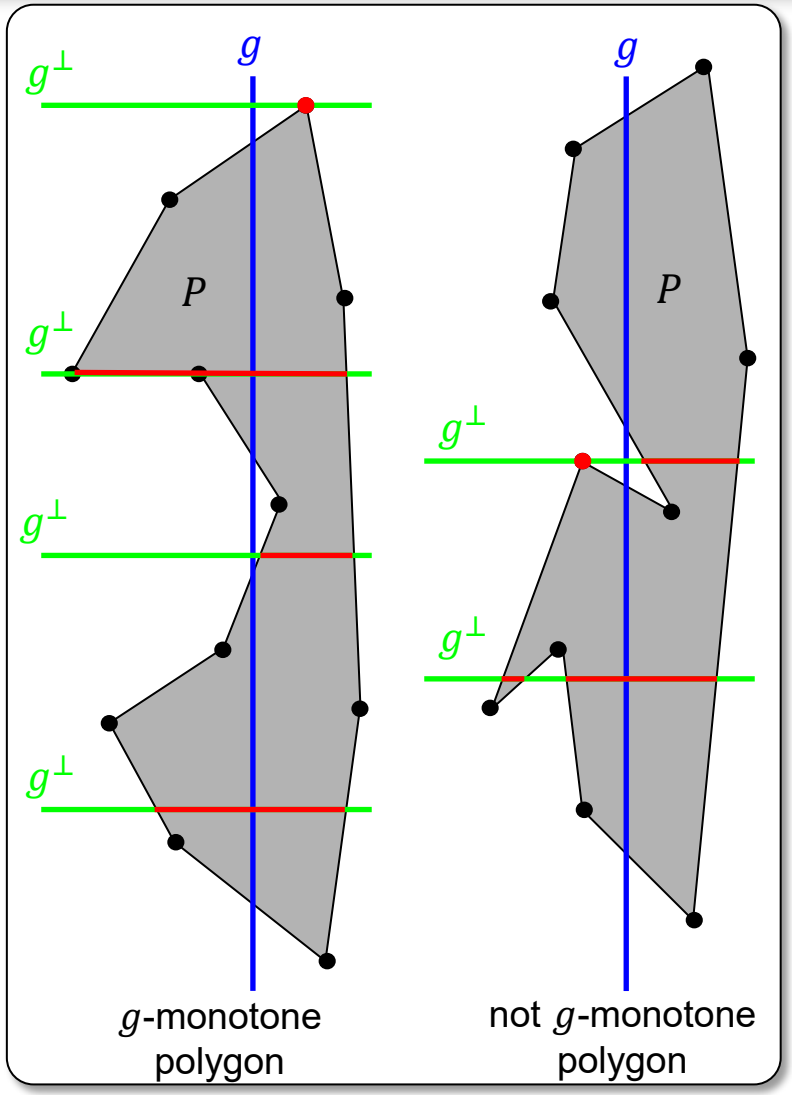
- The proof of Proposition 1 gives a recursive algorithm to construct a triangulation.
  - Searching for a diagonal to subdivide the polygon takes $O(n)$, yielding a total run time of $O(n^2)$.
- Convex polygons can be triangulated in $O(n)$.
  - **But:** Partitioning a polygon into convex regions is as complex as computing a triangulation.
- An efficient approach is the partitioning into *monotone polygons*.

7.2 Monotone partitioning

Definition 1

A polygon P is called ***mono-tone*** with respect to a line g (aka ***g-monotone***), if for **every** perpendicular line $g^{\perp} \perp g$ the intersection $P \cap g^{\perp}$ is connected.

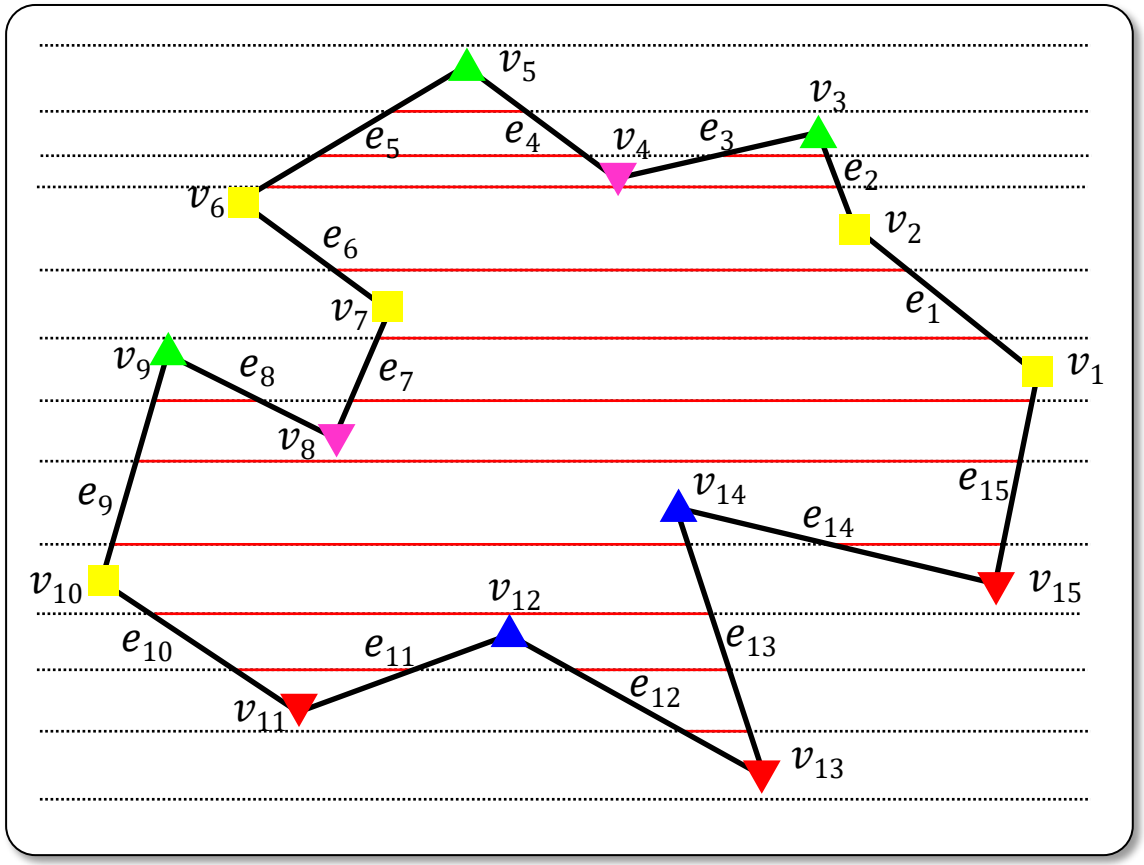
- $P \cap g^{\perp}$ is either a line segment, a point, or empty.
- A polygon monotone with respect to the y-axis is called ***y-monotone***.



7.2 Monotone partitioning

Approach: 1. Partition P in y-monotone pieces,
2. which are triangulated top to bottom.

- For the monotone partitioning the corners are classified into five categories:
  - Start vertex
  - End vertex
  - Regular vertex
  - Split vertex
  - Merge vertex



7.2 Monotone partitioning

- **Definition 2**
  A point u is *below* of v ($u < v$), if $u_y < v_y$ or $u_y = v_y$ and $u_x > v_x$. A point u is *above* of v, if $v < u$.

- **Definition 3**
  A vertex v of a polygon with inner angle $\alpha(v)$ and neighbors u, w is called

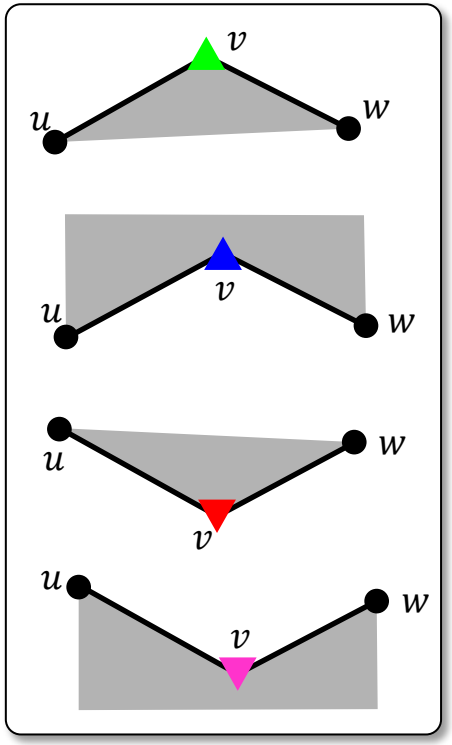
  ▲ ***start vertex***, if $u < v, w < v$ and $\alpha(v) < \pi$

  ▲ ***split vertex***, if $u < v, w < v$ and $\alpha(v) > \pi$

  ▼ ***end vertex***, if $u > v, w > v$ and $\alpha(v) < \pi$

  ▼ ***merge vertex***, if $u > v, w > v$ and $\alpha(v) > \pi$

  ■ ***regular vertex***, otherwise.



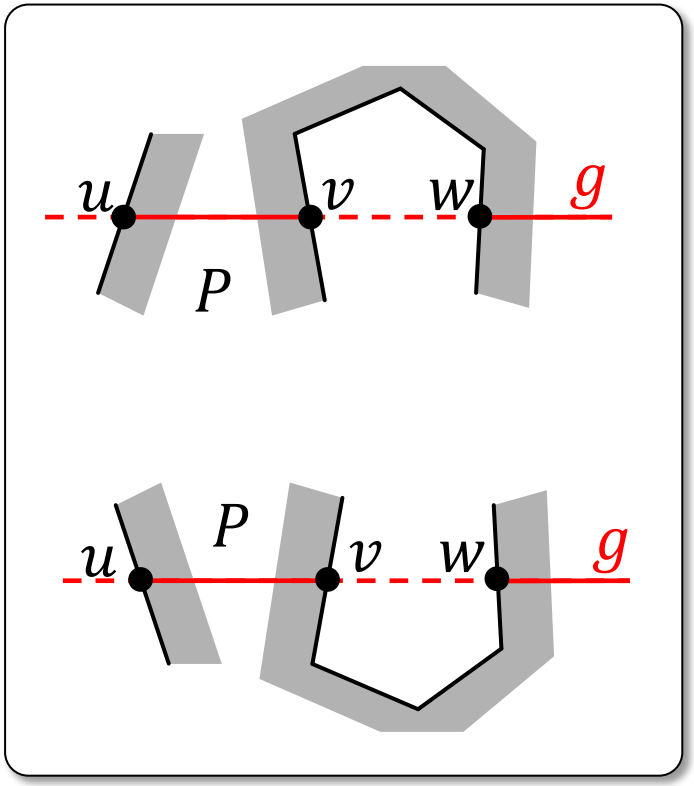
7.2 Monotone partitioning

Lemma 3

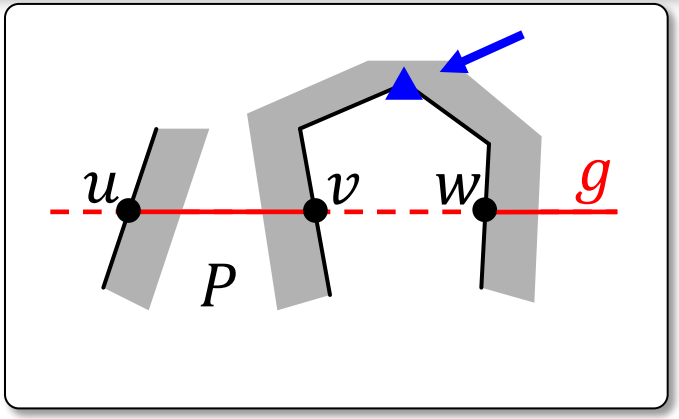
A polygon P is y-monotone, if it does not have split and merge vertices.

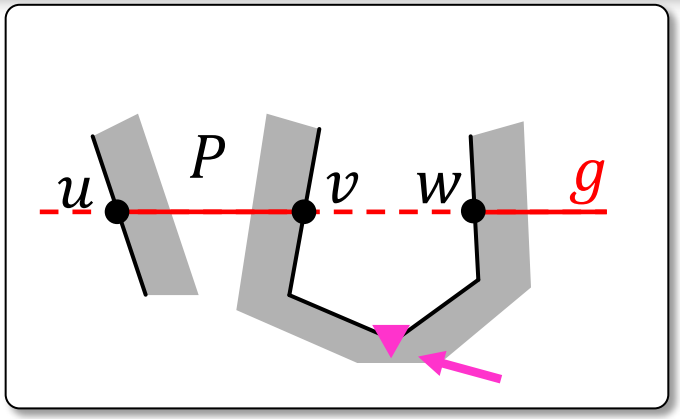
Proof

- Assume P is not y-monotone.
- ▶ Then P should have a split or merge vertex.
- If P is not y-monotone, there is a horizontal line g, that intersects or touches P in at least three points u, v, w (from left to right): w.l.o.g. a line segment $\overline{uv}$ and a point w.



7.2 Monotone partitioning





- Traversing the polygon chain starting in v in both directions, the next intersections with g are u and w.
- If the polygon chain $u..v$ has an end (start) vertex, the polygon chain $v..w$ must have a split (merge) vertex.
  - Analog if $v..w$ has an end or start vertex.
- Thus, P has at least one split or merge vertex.

□

7.2 Monotone partitioning

- The partitioning into y-monotone polygons can be computed by inserting suitable diagonals to eliminate the split and merge vertices.
  - These diagonals should go upward from a split vertex and downward from a merge vertex (without intersection).
  - This way they become regular vertices of the resulting sub-polygons,
    - because in both sub-polygons the predecessor and successor vertices of the former split/merge vertex lie on different sides (above and below) of the former split/merge vertex.

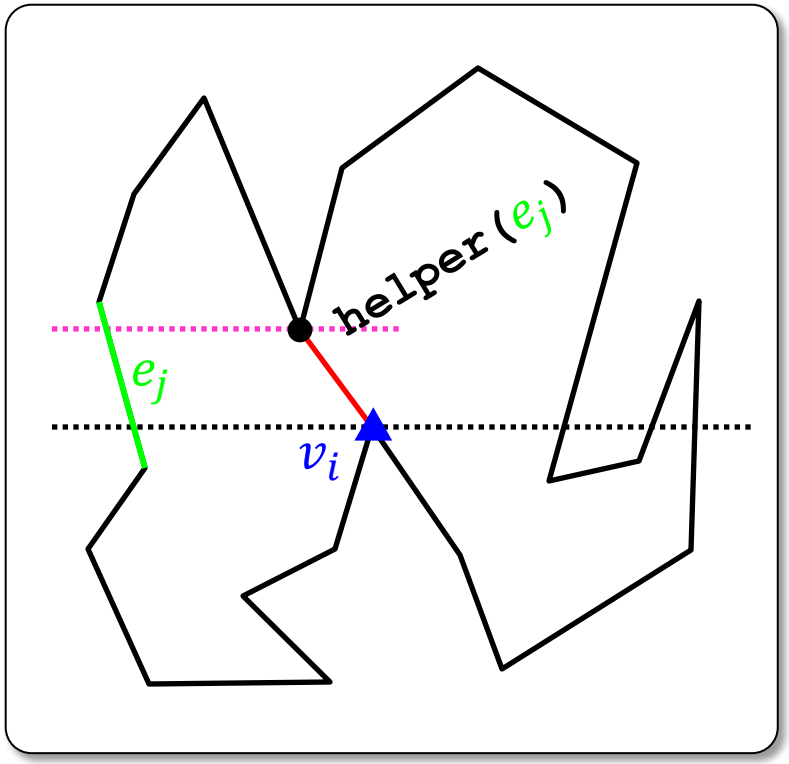
7.2 Monotone partitioning

- Use a horizontal sweep-line moving from top to bottom.
  - At every vertex, an event is triggered and processed.
  - The vertices v_i of P are stored in a priority queue **Q** sorted from top to bottom, i.e., the priority is the y-coordinate.
  - Edges $e_i = \overline{v_i v_{i+1}}$ are stored in a doubly linked list, to compute neighbors and partitionings in constant time.

7.2 Monotone partitioning

Goal: Add diagonal from a split vertex v_i to a vertex above.

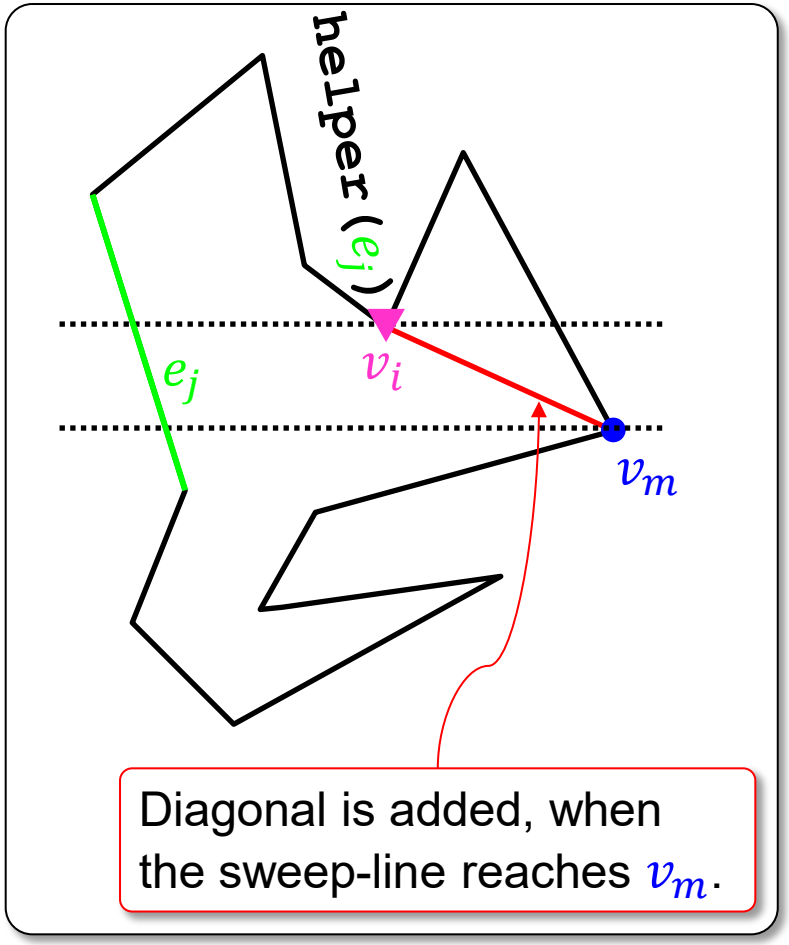
- Let v_i be a split vertex on the sweep-line and e_j the edge left of it on the sweep-line.
- The vertex **`helper(`**e_j**`)`** is the lowest vertex above the sweep-line, such that the horizontal line segment from e_j to **`helper(`**e_j**`)`** lies completely in P.
- Note, that the upper end point of e_j is always a candidate for **`helper(`**e_j**`)`**.
- Insert diagonal from v_i to **`helper(`**e_j**`)`**.



7.2 Monotone partitioning

Goal: Add diagonal from a merge vertex v_i to a vertex below.

- More difficult, because the area below a merge vertex has not been scanned by the sweep-line when it reaches the merge vertex.
- Merge vertices are stored initially as **`helper`** of the edge left (here e_j) of v_i on the sweep-line.
- As soon as e_j gets a new **`helper`**, this new **`helper`** v_m is connected with the merge vertex v_i.



7.2 Monotone partitioning

- The edges and their **helpers** are stored efficiently in a search-tree **T**.
- The sweep-line algorithm operates directly on the doubly linked list **D** of edges $e_i = \overline{v_i v_{i+1}}$.

Algorithm 1: Partition in y-monotone polygons

Input: Simple polygon P in doubly linked edge list D.
Output: Partitioning of P in y-monotone pieces, stored in D.

```
1: Fill priority queue Q from top to bottom with vertices;
2: Initialize empty search tree T;
3: while (Q is not empty) {
4:    Take topmost vertex v_i from Q;
5:    Process the event depending on the type of v_i;
6: }
```

7.2 Monotone partitioning

- Events are processed depending on the type of the vertex.

StartVertexEvent(v_i)

```
1: Add e_i with helper(e_i):=v_i to T;
```

EndVertexEvent(v_i)

```
1: if (helper(e_{i-1}) is a merge vertex) then {
2:    Add diagonal from v_i to helper(e_{i-1}) to D;
3: }
4: Remove e_{i-1} from T;
```

SplitVertexEvent(v_i)

```
1: Search in T for the edge e_j left of v_i;
2: Add diagonal from v_i to helper(e_j) to D;
3: helper(e_j):=v_i;
4: Add e_i with helper(e_i):=v_i to T;
```

7.2 Monotone partitioning

MergeVertexEvent(v_i)

```
1: if (helper(e_{i-1}) is a merge vertex) then {
2:    Add diagonal from v_i to helper(e_{i-1}) to D;
3: }
4: Remove e_{i-1} from T;
5: Search in T for the edge e_j left of v_i;
6: if (helper(e_j) is a merge vertex) then {
7:    Add diagonal from v_i to helper(e_j) to D;
8: }
9: helper(e_j) :=v_i;
```

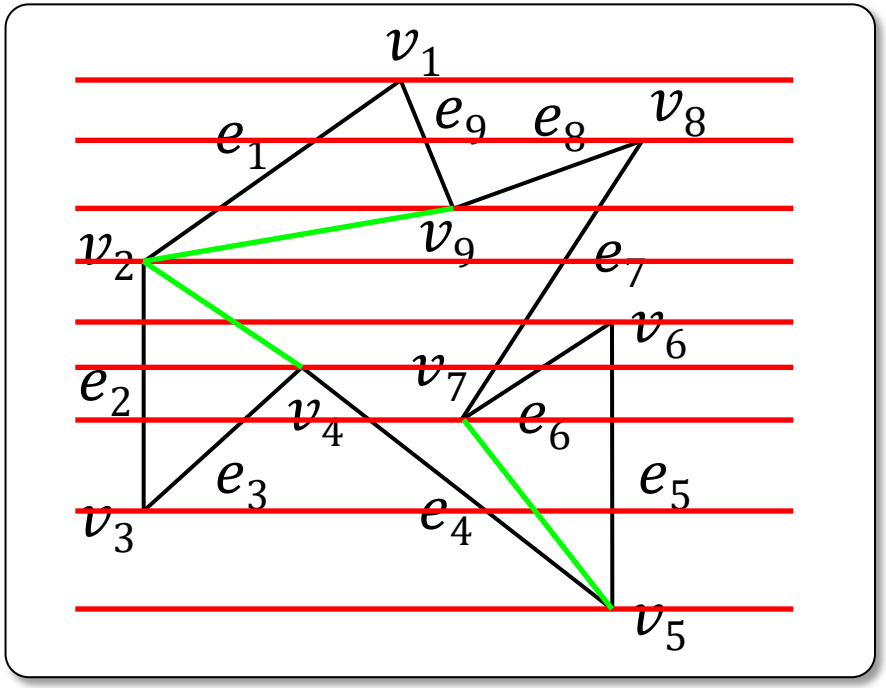
7.2 Monotone partitioning

RegularVertexEvent(v_i)

```
 1: if (the interior of P is right of v_i) then {
 2:    if (helper(e_{i-1}) is a merge vertex) then {
 3:       Add diagonal from v_i to helper(e_{i-1}) to D;
 4:    }
 5:    Remove e_{i-1} from T;
 6:    Add e_i with helper(e_i):=v_i to T;
 7: } else {
 8:    Search in T for the edge e_j left of v_i;
 9:    if (helper(e_j) is a merge vertex) then {
10:       Add diagonal from v_i to helper(e_j) to D;
11:    }
12:    helper(e_j):=v_i;
13: }
```

7.2 Monotone partitioning

Example



event	content of $\mathbf{T}$
v_1	$(e_1 v_1)$
v_8	$(e_1 v_1)(e_8 v_8)$
v_9	$(e_1 v_9)$
v_2	$(e_2 v_2)$
v_6	$(e_2 v_2)(e_6 v_6)$
v_4	$(e_2 v_4)(e_4 v_4)(e_6 v_6)$
v_7	$(e_2 v_4)(e_4 v_7)$
v_3	$(e_4 v_7)$
v_5	

- What is the content of the search-tree $\mathbf{T}$ after each event?
- Which diagonals are added?

7.2 Monotone partitioning

Lemma 4

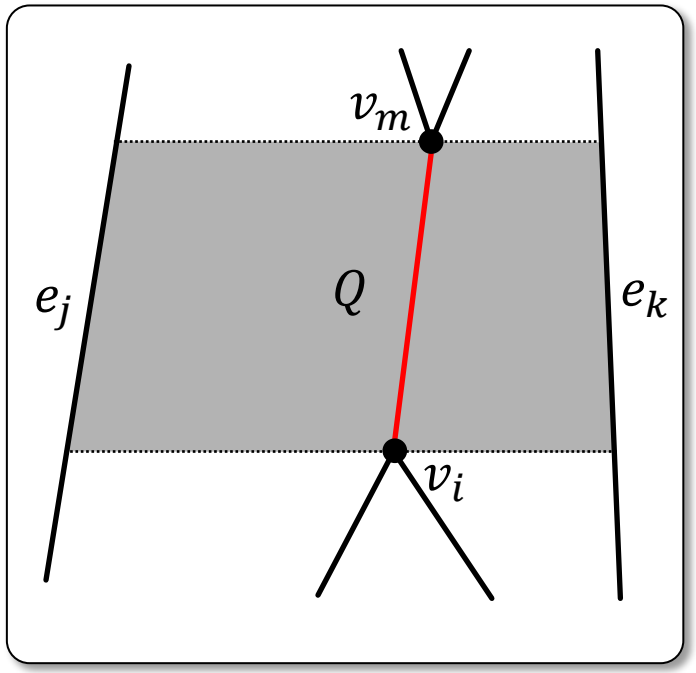
Algorithm 1 partitions P into y-monotone sub-polygons by adding non-intersecting diagonals.

Proof

- All split vertices are removed by upward and all merge vertices by downward diagonals.
- By Lemma 3 all remaining sub-polygons are y-monotone.
- To prove that these diagonals do not intersect, we examine for example the diagonals generated by **`splitVertexEvent`**.

7.2 Monotone partitioning

- Let v_i be a split vertex and $\overline{v_m v_i}$ be the diagonal inserted in **`SplitVertexEvent`**,
- e_j and e_k are the edges left and right of v_i, and
- **`helper`**$(e_j) = v_m$.



- Consider the area Q between e_j and e_k, bounded from below and above by horizontal lines through v_i and v_m.
- Because v_m is the last event before/ above v_i relative to e_j, Q does not contain any further edges or vertices.
- Thus, the diagonal has no intersections.
- The other events are treated analogously.

□

7.2 Monotone Partitioning

Proposition 5

A simple polygon with n vertices can be partitioned into y-monotone pieces in $O(n \log n)$.

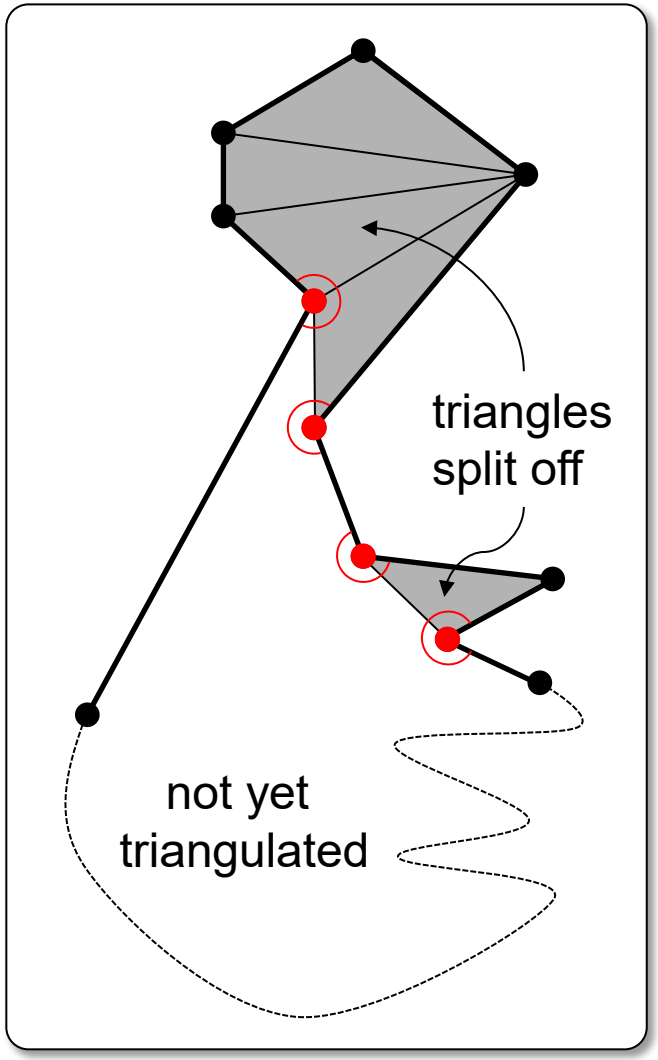
Proof

- Constructing **Q** takes $O(n)$, and **T** is initialized in $O(1)$.
- Every event takes at most one operation on queue **Q**, two insertions on list **D** and one search, one insertion and one removal on tree **T**, which takes in total $O(\log n)$.
- Because there are n events, the total run time is $O(n \log n)$.

□

7.3 Triangulation of monotone polygons

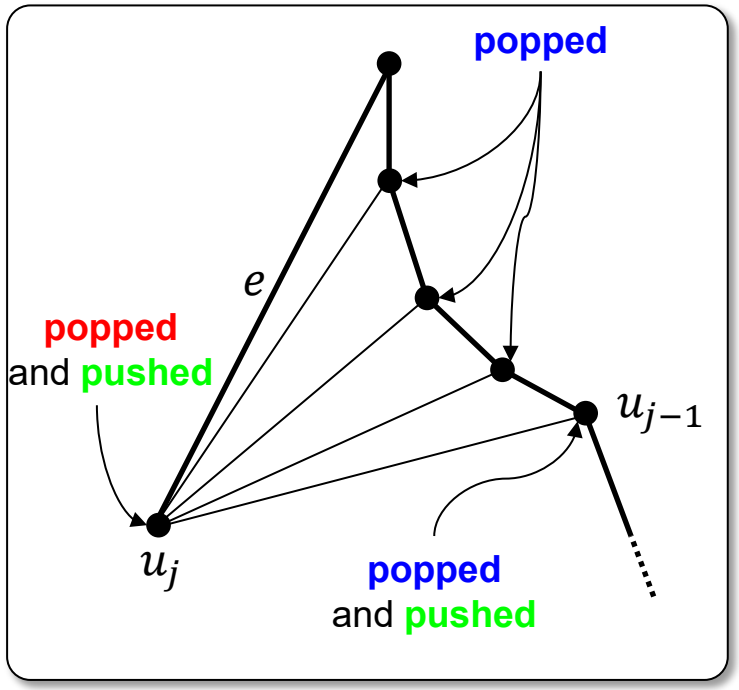
- To triangulate a y-monotone polygon the left and the right boundaries are processed top to bottom and the vertices a connected accordingly.
- Problematic: ***Reflex vertices*** with an inner angle of $\alpha(v) > \pi$.
  - The vertices are stored in a stack, containing the vertices of the polygon above the sweep-line that are not yet triangulated.
  - The lowest element of the stack belongs to the opposite side of the polygon.



7.3 Triangulation of monotone polygons

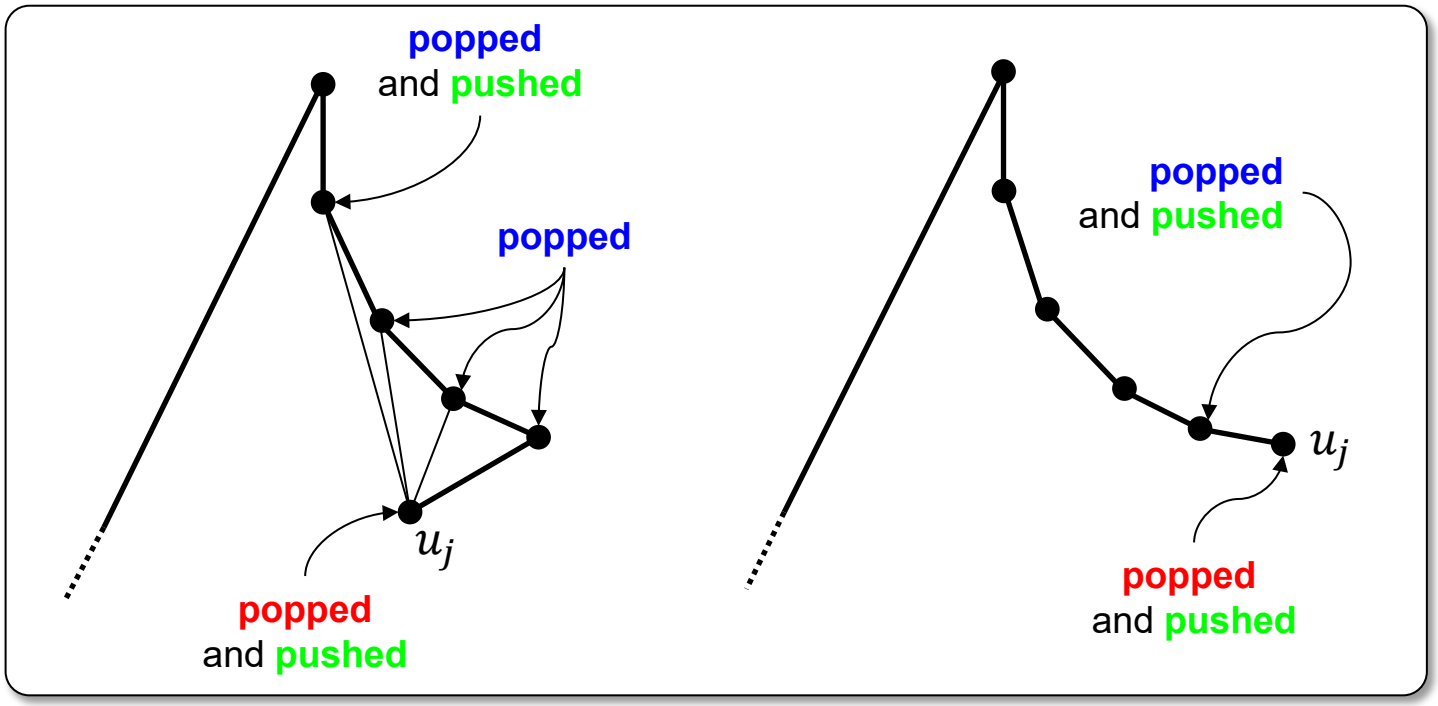
- Process the points $u_1, \ldots, u_n$ of the y-monotone polygon from top to bottom.
- For a new point u_j, test if the top-most element on the stack lies on the opposite side of P.

1. In this case,
   a) **pop** u_j from the stack and connect to all points from the stack, except for the last one, i.e. **pop** from the stack.
   b) Then **push** u_{j-1} and u_j back on the stack (u_j on top).



7.3 Triangulation of monotone polygons

2. Otherwise,
   - **a)** **pop** u_j from the stack and connect (i.e. **pop**) it to as many points on the stack as possible (i.e. the diagonal is inside P), and
   - **b)** the last of these and u_j are **pushed** back on the stack.



7.3 Triangulation of monotone polygons

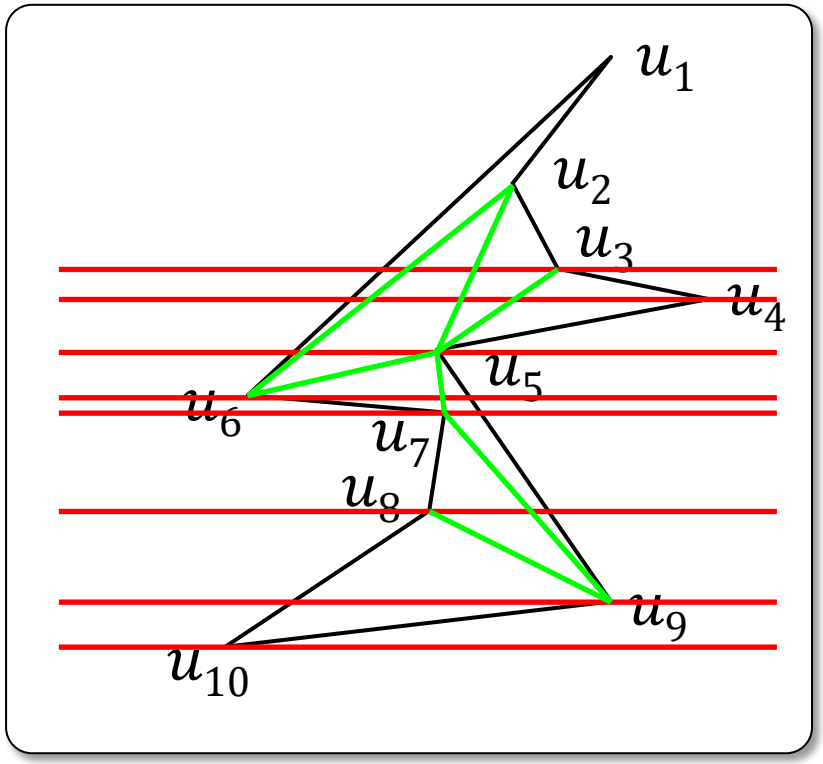
Algorithm 2: Triangulate y-monotone polygon

Input: A y-monotone polygon P in a doubly linked edge list D.
Output: Triangulation of P in D.

```
 1: Sort vertices top to bottom u_1,...,u_n;
 2: Initialize empty stack S; S.push(u_1); S.push(u_2);
 3: for (j = 3,...,n-1) {
 4:   if (u_j and S.top() are on different sides) then {
 5:     Pop all vertices from S and add their diagonals to u_j
        to D, except for the last one;
 6:     S.push(u_{j-1}); S.push(u_j);
 7: } else {
 8:     Pop one vertex from S;
 9:     Pop the other vertices from S and add their diagonals
        to u_j to D, as long as this diagonal lies inside P;
10:     Push the last popped vertex back; S.push(u_j);
11: }   }
12: Pop all vertices from S and add their diagonals to u_n to D,
    except for the first and last one;
```

7.3 Triangulation of monotone polygons

Example



j	stack S
	u_2u_1
3	$u_3u_2u_1$
4	$u_4u_3u_2u_1$
5	$u_5\,u_2u_1$
6	u_6u_5
7	u_7u_5
8	$u_8u_7u_5$
9	u_9u_8
10	

- What is the content of the stack after each iteration?
- Which vertices are connected?

7.3 Triangulation of monotone polygons

Lemma 6

A y-monotone polygon with n vertices can be triangulated in linear run time.

Proposition 7

A polygon with n vertices can be triangulated in $O(n \log n)$ using $O(n)$ of memory.

- This follows from the combination of Algorithms 1 and 2.

Remarks

- Algorithm 1 is also correct for polygons with holes.
- For triangulation of arbitrary polygons $\Omega(n \log n)$ is a lower bound [1].
- Simple polygons can be triangulated in $O(n)$.
  - A very complicated linear algorithm is due to Chazelle [2], 1991.
- The problem to triangulate (*tetrahedralize*) a 3d polytope is in general not solvable without auxiliary inner vertices.
  - There are efficient algorithms for such polytopes, but
  - to decide if auxiliary inner vertices are necessary is NP-complete.

7.4 Literature

[1] Marc de Berg et al., *Computational Geometry: Algorithms and Applications*, 2nd Edition, Springer, 2000, Chapter 3.

[2] B. Chazelle, *Triangulating a simple polygon in linear time*, Discrete Computational Geometry, 6:485-524, 1991.