[Triangulation]
Computational Geometry (CSc 635)

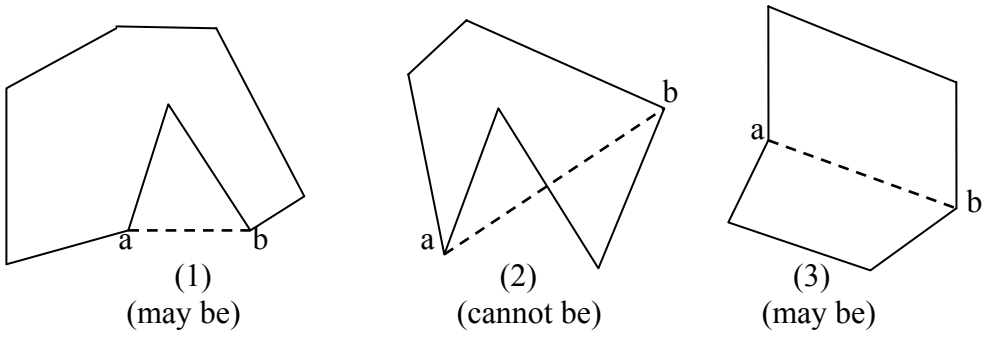
Jagdish Bhatta

Central Department of Computer Science & Information Technology
Tribhuvan University

Triangulation Issues

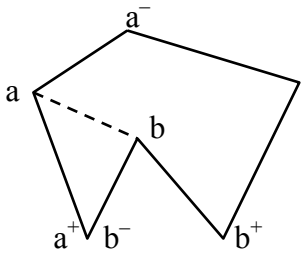
Identifying the diagonal

- Diagonals do not intersect the edges of polygon except those whose end points become the end point of diagonal themselves.
- Diagonals can't be exterior to the polygon. Thus, checking these properties, we can identify whether a segment is diagonal or not.
- If we ignore, for a while, the distinction between internal & external diagonals, determining (a, b) to be a diagonal includes application of intersection. For every edge e of the polygon not incident to either end point a & b, see if e intersects (a, b). As soon as intersection is detected, it is known that (a, b) is not a diagonal. If no such edge intersects (a, b), then (a, b) might be a diagonal.



- Now, next is to test whether (a, b) is interior or exterior to the polygon.

Consider the edges incident to the diagonal end points a and b. let edges are along a^- & a^+ from point a & similarly b^- & b^+ from b. the two edges incident to any point a or b from a cone which may be with convex or non-convex apex.



Here, the cone a^-aa^+ has convex apex

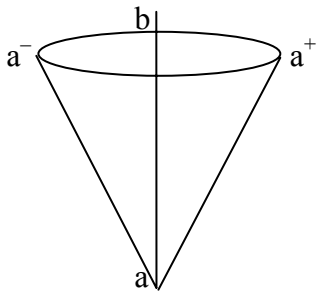
the cone b^-bb^+ has non-convex apex.

If the segment (a, b) is inside the cone, it is interior to polygon otherwise exterior.

Case I:

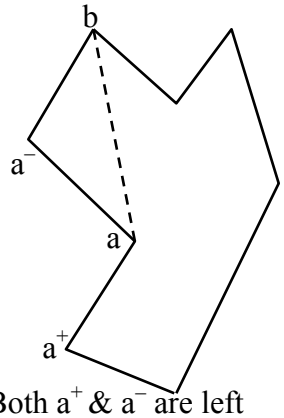
If the apex of cone is convex, then (a, b) is diagonal i.e. (a, b) is interior to the cone if aba^- is left turn and baa^+ is also left turn.

i.e. (a, b) is diagonal if, left turn (a, b, a^-) & left turn (b, a, a^+) is true.

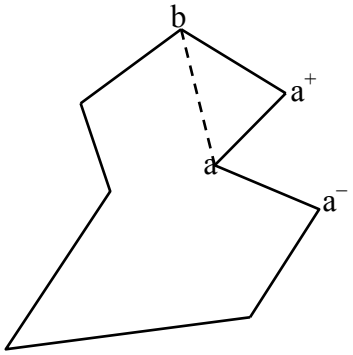


Case II: If apex is non-convex:

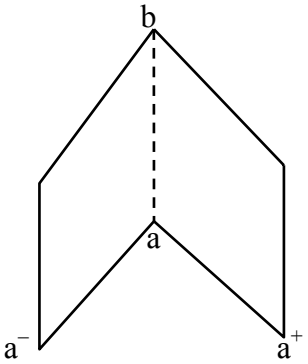
If (a, b) is diagonal then a^- & a^+ could be both left or both right or one left & one right of (a, b).



Both a^+ & a^- are left

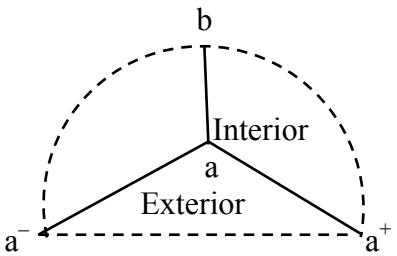


Both a^+ & a^- are right

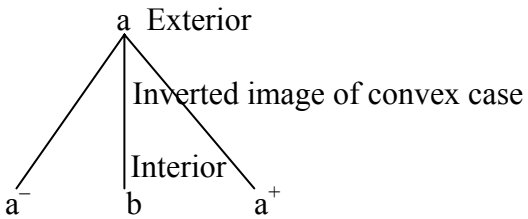


a^- is left and a^+ is right.

The exterior of neighborhood of 'a' is a cone as in convex case. The reflex vertex looks like a convex vertex if interior & exterior are changed.

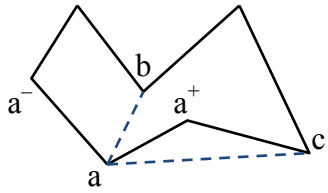


(1)



Reversing (1)

So, (a, b) is interior if it is not exterior i.e. aba^+ is left and baa^- is left is not true. So, (a, b) is diagonal if it does not intersect the edges in any point other than a & b and lies inside cone of incident edges at a or b.



aba^- → left turn
baa^+ → left turn
} ⇒ ab is diagonal i.e. interior to cone.

But for ac, a & c are reflex. So, a^+ is left or on ac and a^- is left or on ca should be false. Here, aca^+ is left turn and caa^- is right turn. So, ca is external diagonal

Triangulation Algorithms

Diagonal Based Approach

- Triangulation is done by adding the diagonals.
- For polygon with n-vertices, the candidate diagonals can be O (n^2).
- Check intersection of O (n^2) segments with O (n) edges it costs O (n^3).
- There can be total of n–3 diagonals on triangulation. So total cost in triangulation is O (n^4). ***(Can be speeded up by a factor of n)***

Repeated Ear Removal Approach

Input: - Given a polygon P with n-vertices.

Output: - Triangulated polygon. (Diagonals)

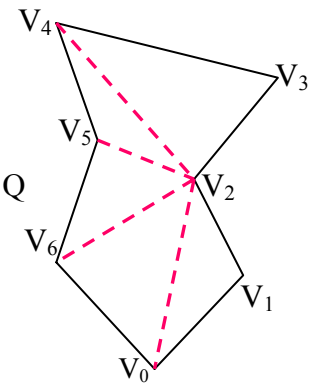
Step 1: - Initialize, Q = P

Step 2: - While Q is not a triangle do

Step 3: - Find an ear Δabc by traversing the boundary of Q

Step 4: - Q = Q – Δabc

end do



(V_0, V_2) (V_2, V_4) (V_2, V_5) (V_2, V_6)

Complexity Analysis:

→ Step 1 takes O (1) time.
→ Step 2 takes O (n) as while loop runs times.
→ Step 3 takes O (n) as to find an ear takes O (n).
→ Step 4 takes constant time i.e. (O (1)).

Step 3 is executed n times so total complexity O (n^2).

Polygon Partitioning

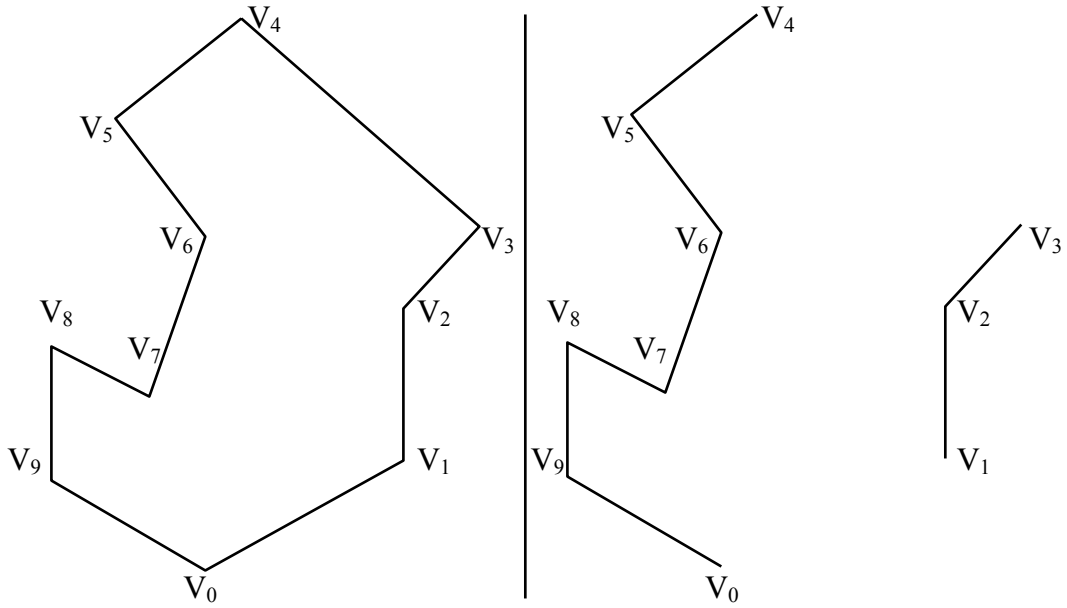
A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of original polygon P. Thus, partitioning a polygon means completely dividing its interior region into non-overlapping pieces.

- Partitioning is to reduce complexity as to work with the convex pieces is efficient than to work on non-convex.
- Application might be as in character recognition.
    - Optionally scanned characters can be represented as polygons & partitioned into convex pieces. The resulting structures can be matched against a database of shapes to identify the characters.

Monotone Partitioning

Monotonicity: - It is defined with respect to a line.

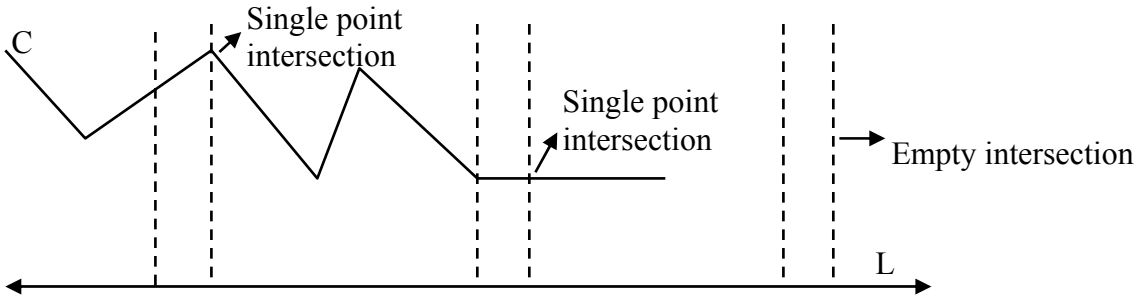
Polygon Chain: - The open boundary of a polygon is called polygon chain. Removing at least one vertex from a polygon gives the polygon chain.



Monotonicity of a polygon chain:-

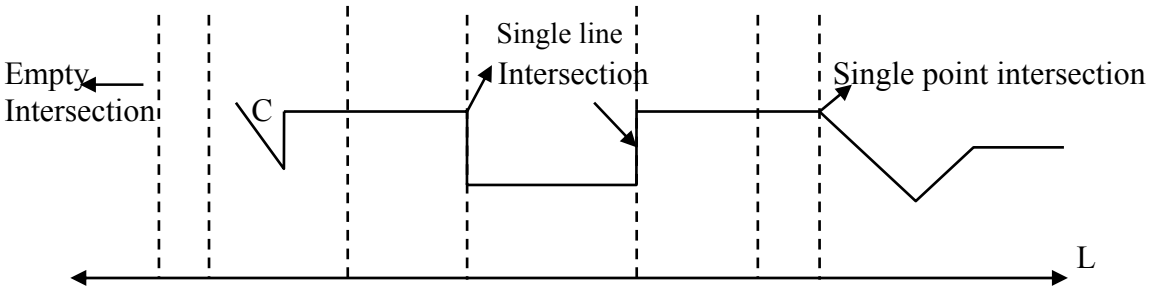
1) **Strictly monotone polygon chain:**

   A polygon chain C is strictly monotone with respect to a line L, if intersection of perpendicular from L to C is either empty or a single point.



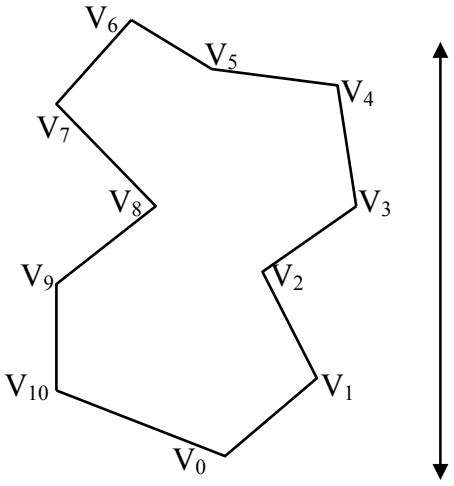
2) Monotone polygon chain:

A polygon chain C is monotone with respect to a line L, if intersection of perpendiculars from L to C is either empty or a single point or a single line segment.



Monotone Polygon

A simple polygon P is said to be monotone with respect to a line L if P can be split into two separate chains A & B and each A & B are monotone with respect to the line L.



Left chain V_7, V_8, V_9, V_{10}
Right chain: $V_0, V_1, V_2, V_3, V_4, V_5, V_6$
Polygon may be Y-monotone→w. r. t. y-axis
Polygon may be X-monotone→w. r. t. x-axis.

Triangulating a monotone polygon:

A monotone polygon can be triangulated in linear time making use of greedy triangulation algorithm. Here, the idea is:

- Partition the polygon into two monotone chains with respect to a line say, y-axis.
- Merge the monotone chains into a single sorted list of vertices.
- Scan vertices from top to bottom (i.e. from left to right in sorted list), cutting off triangles n greedy manner if three vertices define a triangle. The greedy approach here is to remove the first available triangle.

Sorting of vertices:

Let us consider a polygon which is monotone with respect to y-axis. Let the input vertices are V_0, V_1, V_2, V_3, ---------, V_{n-1}.

For sorting vertices by y-coordinate;

Step 1: Find the topmost vertex V_t → O (n) (with highest y-coordinate)

Step 2: Find the bottommost vertex V_b → O (n).

- All the vertices from V_b to V_t are on right chain.
- All the vertices remaining are on left chain.

Step 3: Merge two sorted list i.e. the left chain and right chain into single list. This results sorted vertex list. → O (n).

Data Structures

- *Linked list of vertices (doubly connected)*
- *Auxiliary data structure – as stack (initially empty)*

Algorithm:

Step 1: Sort vertices by y-coordinate. Let the list is; L = {p_0, p_1, p_2, p_3, --------, p_{n-1}} in decreasing order of y-coordinate.

Step 2: Push p_0 & p_1 into a stack S. Stack content at any instance may be a set of vertices as;

V_0', V_1', ----------, V_t' from bottom to top. i.e. V_t' is topmost element.

Step 3:

for p = 2 to n (i.e. i = 2; i<n; i++)

Scan each vertex pi from the list.

if pi is adjacent to V_0' (i.e. bottom of stack)

```
                temp = Vt' (stack top)
                while (t > 0) do                    //top > 0
                        draw diagonal pi to Vt'
                        pop (S)
                        t - -
                end do
                pop (S)
                push (temp)
                push (pi)
        else if pi is adjacent to Vt' (top of stack)
                while (t > 0 && Vt' is not reflex) do
                        Draw diagonal pi to Vt-1'
                        pop (S)
                        t - -
                end do
                push (pi)
        end if
end for
```

Complexity Analysis: -

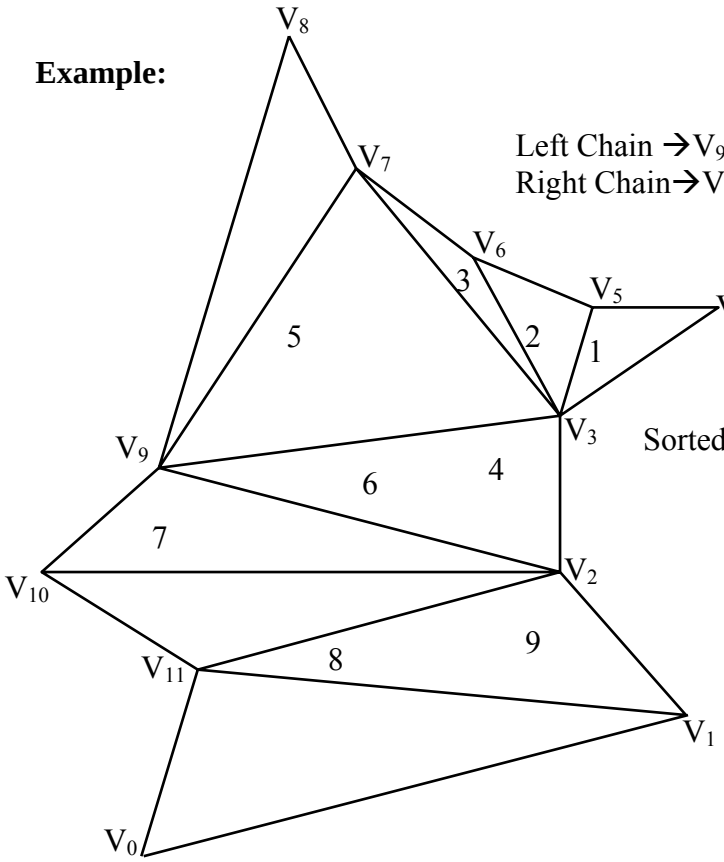
Here, Step (1) takes O (n) time. (Creating sorted list)

Step (2) takes O (1) time. (Initializing in constant time)

Step (3) takes O (n) time. (Loops for n-time)

Thus, O (n) time

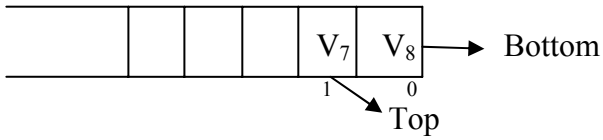
Example:



Left Chain → V_9, V_{10}, V_{11}
Right Chain → V_8, V_7, V_6, V_5, V_4, V_3, V_2, V_1, V_0

Sorted list:- V_8, V_7, V_6, V_5, V_4, V_3, V_9, V_2, V_{10}, V_{11}, V_1, V_0

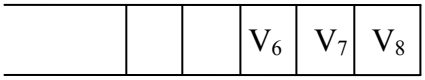
Push V_8 & V_7 into stack S. So,



Scan V_6: ($i = 2 < 11$)

V_6 is adjacent to tos; V_7

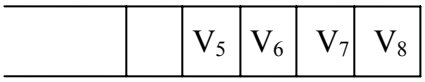
$t = 1 > 0$ but V_7 is reflex. So push V_6 into stack.



Scan V_5:

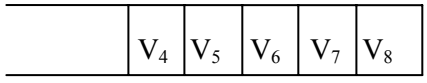
V_5 is adjacent to V_6 & V_6 is reflex ($i = 3 < 11$)

So, push V_5 into stack



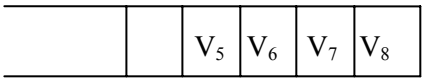
Scan V_4:

Same as above so push V_4 ($i = 4 < 11$)



Scan V_3:

- V_3 is adjacent to V_4. (I = 5 < 11), t = 4 > 0 & V_4 is not reflex vertex.
  Draw diagonal V_3 to V_5 & pop V_4 & t --. So,



- V_3 is adjacent to V_5 & V_5 is not reflex so draw diagonal V_3 to V_6 & pop V_5 & t - -.

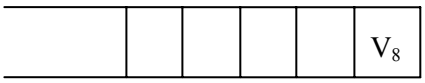
- V_3 is adjacent to V_6 & V_6 is not reflex so draw diagonal V_3 to V_7 & pop V_6 & t - -.

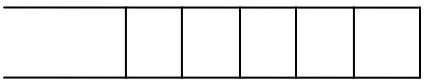
- V_3 is adjacent to V_7 but V_7 is reflex so push V_3.

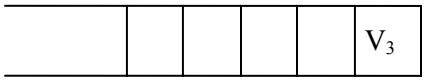
Scan V_9:

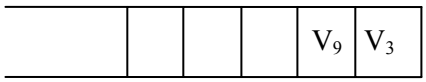
V_9 is adjacent to V_8 (bottom of stack) (I = 6 <11)
So, temp = V_3.
Draw diagonal V_9 to V_3 & pop V_3 & t --.

Draw diagonal V_9 to V_7 & pop V_7 & t -- $\Rightarrow$ t = 0.



Pop V_8 →  Empty

Push (temp) → 

Push V_9 → 

Scan V_2:

V_2 is adjacent to V_3 (bottom of stack) (i = 7 < 11), temp = V_9.

Draw diagonal V_2 →V_9, pop (V_9) & t--.

Pop (V_3)

Push (V_9)

Push (V_2) 

Scan V_{10}:

V_{10} is adjacent to V_9. (i= 8 < 11), so temp = V_2.

Draw diagonal $V_{10} \rightarrow V_2$, pop (V_2) & t--.

Pop (V_9)

Push (V_2)

Push (V_{10})

Scan V_{11}:

V_{11} is adjacent to V_{10} (top of stack). (i= 9 < 11), & V_{10} is not reflex.

Draw diagonal V_{11} to V_2 & pop (V_{10}) & t--.

Push V_{11}

Scan V_1:

V_1 is adjacent to V_2, bottom of stack, (i= 10 < 11); i++

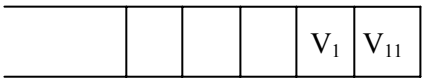
So temp = V_{11}

Draw diagonal V_1 to V_{11} & pop (V_{11}) & t--.

Pop (V_2)

Push (V_{11})

Push (V_1)



Scan V_0:

i= 11 < 11 for loop get terminated. So, final triangulation

Triangulating a simple polygon by monotone partitioning:

- Clearly, monotone polygons can be triangulated in linear time. So, partitioning non-monotone polygons into monotone pieces and triangulating each monotone piece becomes a better efficient approach than the diagonal based triangulation approach. A method for monotone partitioning is *trapezodalization*.

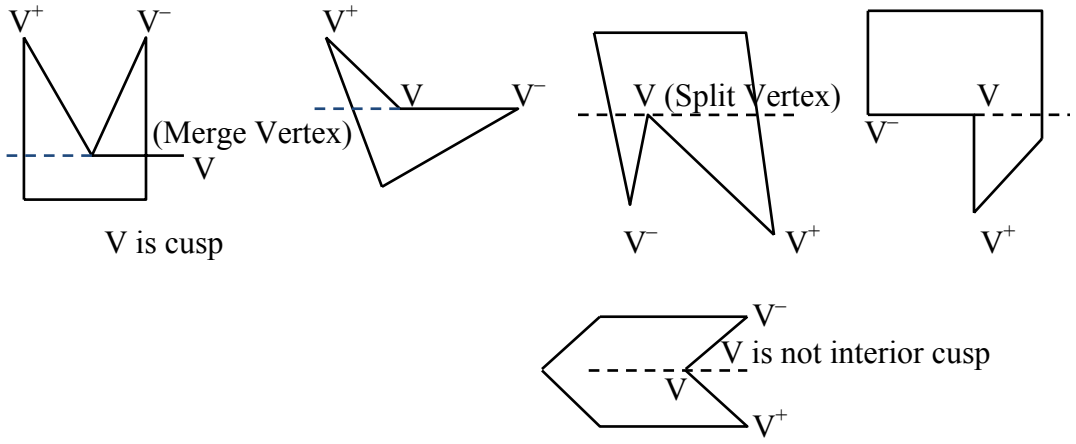
Trapezodalization:

Trapezodalization of a polygon is done by drawing horizontal line through every vertex of the polygon, and hence also known as horizontal trapezodalization.

- A trapezoid is a quadrilateral with two parallel edges.
- A triangle is a trapezoid, with one of the two parallel edges of zero length.
- The vertices through which the horizontal lines are drawn are called supporting vertices.
- A trapezoid so obtained has two supporting vertices.

Interior Cusp:

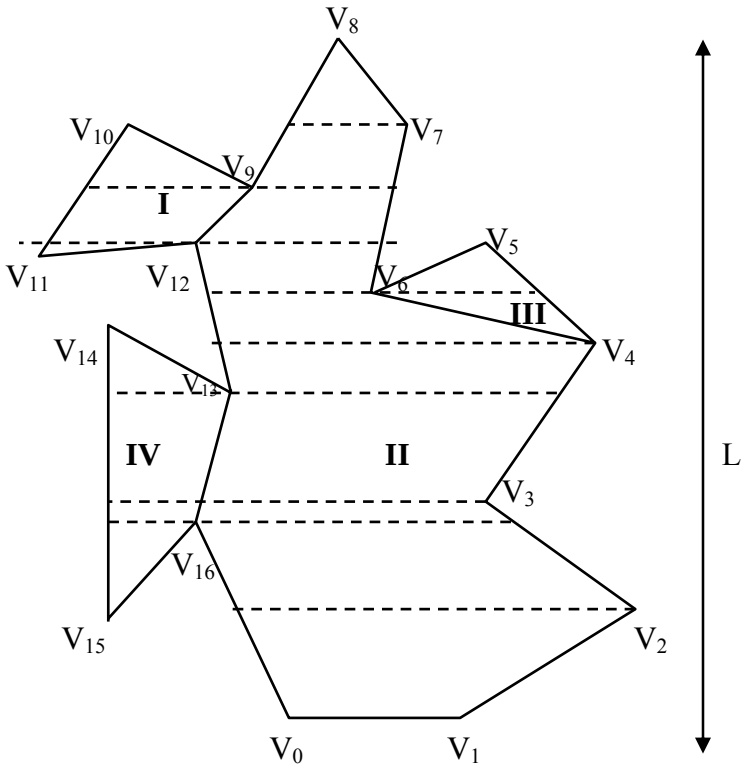
An interior cusp of a polygon is a reflex vertex V whose adjacent vertices V^- & V^+ are either at or above or at or below V.



Partitioning Polygon by Trapezodalization: -

The idea with this trapezodalization approach is;

- Draw horizontal lines (chords) through all vertices,
- If every interior supporting vertex of the trapezoid V supports, downward for "downward" cusp & upward for "upward" cusp, then (these diagonals) partition the polygon into monotone pieces with respect to vertical line.
- Each monotone piece so obtained above can be triangulated by using linear time algorithm.



Interior Cusp:

V_9, V_{12}, V_6, V_{13}, V_{16}

V_{16}, V_{12} are split vertices

V_9, V_6, V_{13} are merge vertices.

Monotone Pieces are:

I: V_9, V_{10}, V_{11}, V_{12}

II: V_0, V_1, V_2, V_3, V_6, V_7, V_8, V_9, V_{12}, V_{13}, V_{16}

III: V_4, V_5, V_6

IV: V_{13}, V_{14}, V_{15}, V_{16}

Algorithm:

Step 1: Sort vertices by y-coordinate and perform plane sweep approach for trapezodalization.

Step 2: Partition polygon into monotone pieces by connecting cusps with supporting vertex.

Step 3: Triangulate each monotone pieces in linear time using the linear time algorithm for monotone triangulation.

Complexity Analysis:

Step 1 takes O (n logn) → as sorting in plane sweep & searching also determining V's position → logn time for each n→n logn.

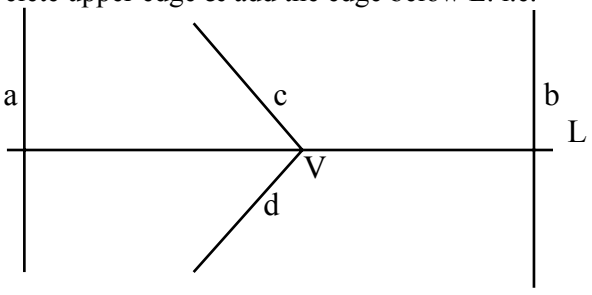
Step 2 takes O (n)
Step 3 takes O (n)
So total time complexity = O (n logn)

Plane Sweep Paradigm:

The trapezodalization approach for monotone partitioning of simple polygons depends upon this plane sweep method. The main idea of plane sweep method is to sweep a line over the plane maintaining some type of data structures along the line. The sweep stops at discrete events (vertices) where preprocessing occurs and subsequent data structure is updated.

- Sort the vertices by y-coordinate which requires O (n log n) for general polygons and maintain the edge list in some ordering as E = {e_0, e_1, e_2, ---------, e_{n-1}}
- Consider a sweep line L from top to bottom where the data structure along L is the sorted list of polygon edges that L intersects with. Initially, it is empty; i.e. E_1 = { }
- The processing required at each stoppage is finding the edge immediately left and right of V along L. (V is in between two edges in the edge list in data structure along L).
- To find edges immediately left/right of the vertex, let e_i is the pointer to an edge can be retrieved. Let vertical coordinate of V (or L) is y. From the coordinates of end points of e_i & y, x-coordinate of intersection of e_i with L can be obtained. From this, we can compute left/right edge.
- Hence, V's position in the edge list along L which L intersects can be obtained.
- At each event processing the data structure should be updated, i.e. insertion or deletion of edge in E1. There are three possible types of events. Let V be a vertex between edges a & b on L and shared by edges c & d.

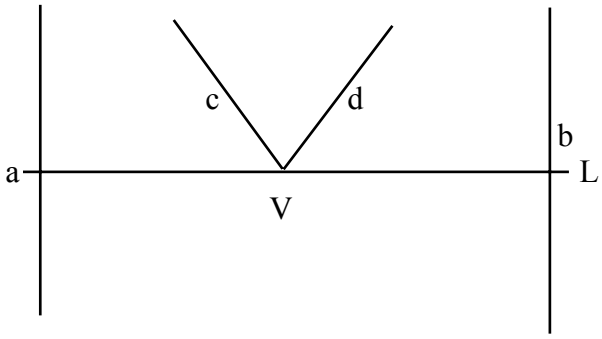
  1) c is above L & d is below L (Regular Vertex)
     - Delete upper edge & add the edge below L. i.e.



     Then delete c & add d.

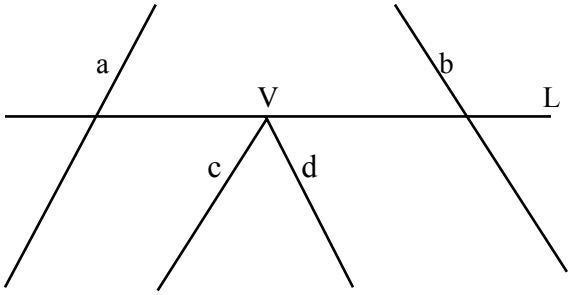
     E1 = {……..a, c, b, ……..} ⇒ {……, a, d, b, …….}

  2) Both c & d are above L. (End vertex or merge vertex)
     - Delete both edges from the edge list. i.e.



Then delete both c & d.

3) Both c & d are below L, (start vertex or split vertex)



Add both edges to the list i.e. add both c & d.

Handling the Cusps:

- when vertex V is split vertex:
  - Link the vertex v to another vertex t that is nearest & above v.
- When vertex v is merge vertex:
  - Link the vertex v to another vertex u that is nearest & below V.

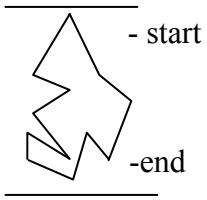
Adding Diagonals:

- Define helper (e) to be the lowest vertex above the sweep line such that e is immediately next to it according to the sweep line.
- For a split vertex, connect a diagonal to the helper of the edge on the left.
- For a merge vertex, connect a diagonal to the next helper replacement.

For each edger e_i helper (e) – lowest vertex (seen so far) above the sweep line (visible to the right of the edge) such that horizontal segment connecting the vertex to e lies inside polygon. (Can be upper end point of e itself)

Start Vertex:

- Add e_i to L
- helper (e_i) ←v_i

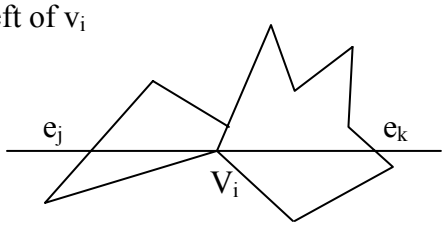


End Vertex:

- if helper (e_{i-1}) is merge vertex, connect v_i to helper (v_{i-1})
- Remove e_{i-1} from L

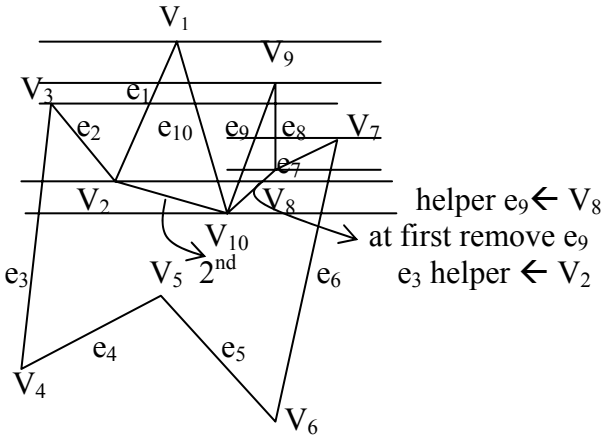
Split Vertex:

- Find in L the edge e_j directly to left of v_i
- Connect v_i to helper (e_j)
- helper (e_j)← v_i
- insert e_i into L
- helper (e_i) ←v_i



Merge Vertex:

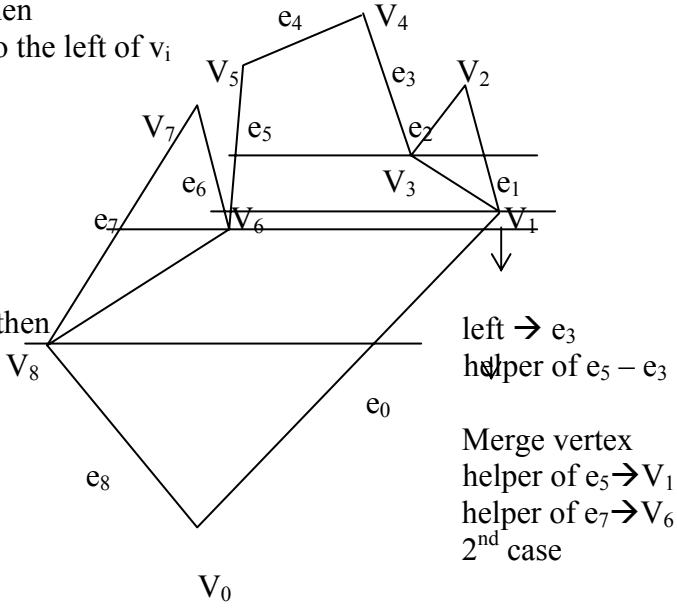
- If helper (e_{i-1}) is merge vertex,
  connect v_i to helper (e_{i-1})
- Remove e_{i-1} from L
- Find edge e_j in L directly to left of v_i
- If helper (e_j) is merge vertex then
  connect v_i to helper (e_j)
- helper (e_j) ← v_i

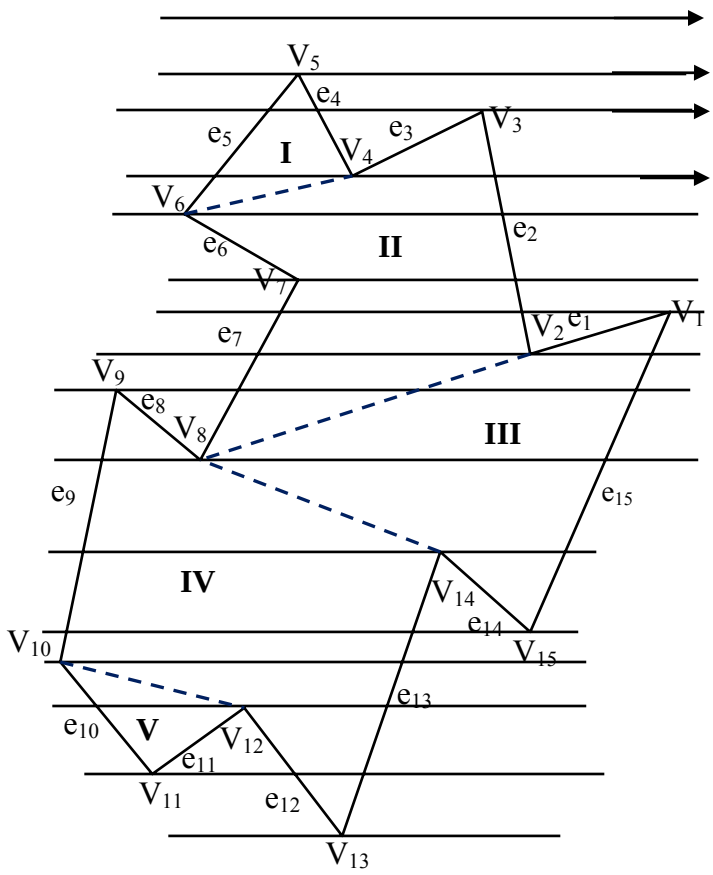


Regular Vertex:

If polygon interior lies to left of v_i then
- Find in L the edge e_j directly to the left of v_i
- If helper (e_j) is merge vertex,
  then connect v_i to helper (e_j)
- helper (e_j) ← v_i

else
- if helper (e_{i-1}) is merge vertex, then
  connect v_i to helper (e_{i-1})
- Remove e_{i-1} from L
- Insert e_i into L
- helper (e_i) ← v_i





$E_1 = \{\ \}$

$E_1 = \{e_5, e_4\}$

$E_1 = \{e_5, e_4, e_3, e_2\}$

$E_1 = \{e_5, e_2\}$ – remove e_4 & e_3

At V_6

helper(e_5) is merge vertex

So, draw diagonal V_6V_4, remove e_5

and add e_6 in L

helper $e_6 \leftarrow V_6$

So, $E_1 = \{e_6, e_2\}$

At V_7

Regular vertex

$E_1 = \{e_7, e_2\}$

At V_1

$E_1 = \{e_7, e_2, e_1, e_{15}\}$

At V_2 Merge Veterx

$E_1 = \{e_7, e_{15}\}$

helper(e_7) $\Rightarrow V_2$

At V_9

$E_1 = \{e_9, e_8, e_7, e_{15}\}$

At V8

Merge vertex

helper(e_7) $\rightarrow V_2$ is merged

So diagonal V_8V_2

$E_1 = \{e_9, e_{15}\}$

helper(e_9) $\Rightarrow V_8$

At V14

Split vertex

helper(e_9) $\rightarrow V_8$

So connect diagonal V_{14} to V_8 And so on…

Analysis:

- For partitioning
- Sorting → O (nlogn)
- Time for finding position of V's in list O (n) for each if linear list is used. But using efficient data structure as height balanced tree it will be O (logn). (updating/querying tree)
- Since search occurs one per each event. So entire place sweep requires O (nlogn) as there are n vertices.

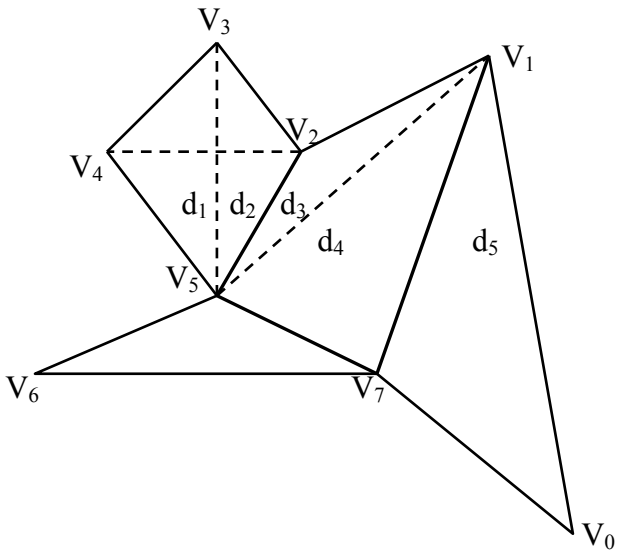
  So, total cost will be O (nlogn)

  Clearly, triangulating monotone pieces is done is O (n) so altogether it costs O (nlogn)

Convex Partitioning Problem (CPP)

Given a simple polygon, P with n-vertices, partition P into minimum number of convex pieces.

Concept of Essential Diagonal



- A diagonal d is essential at vertex V if its removal creates a piece that is non-convex V.
- For convex partitioning, a diagonal at reflex vertex will be essential.
- A diagonal that is not essential for either of its end points is called inessential.

In the figure above,

d_2 is essential at V_5 & V_2

d_4 is essential at V_5& V_7

d_5 is essential at V_7

d_1 is essential at V_3& V_5

d_3 is essential at V_1

Partition of a polygon P may be of two types;

- partition by diagonals
- partition by segments

Here, the difference among the two approaches is that with first, the diagonal end points must be vertices while segment end points need only to lie on boundary of the polygon.

The partitions by segments are in general more complicated in a sense that their end points must be computed somehow, but there is also a freedom to result efficient partitions.

Hertel's – Mehlhorn's Algorithm for CPP (1983)

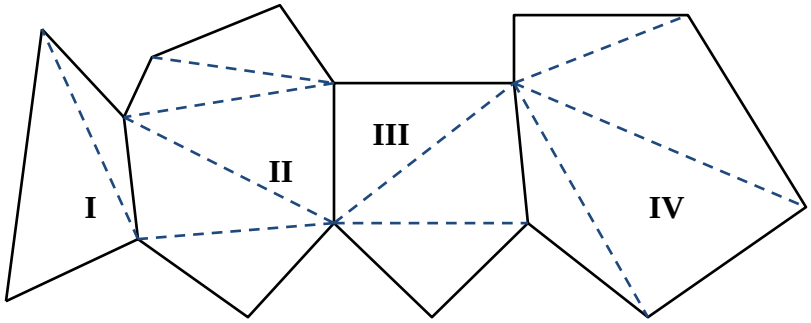
Here, the approach is to partition the polygon by diagonals.

Input: Polygon with n-vertices

Output: Partitioned polygon with convex pieces.

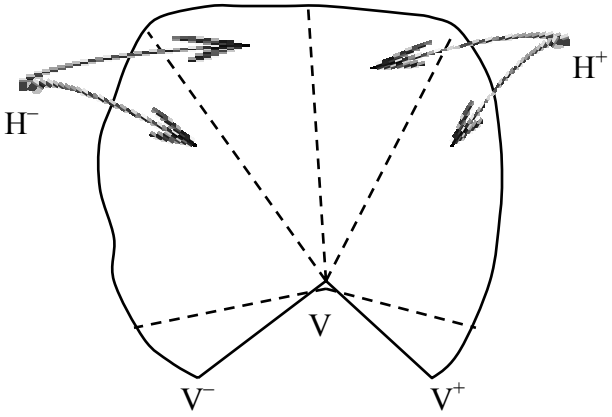
Step 1: Triangulate the polygon.

Step 2: Repeatedly remove the inessential diagonals. The resulting polygon has only convex pieces.



Lemma:

There can be at most two essential diagonals at any given reflex vertex.



Proof:

Let V be a reflex vertex and V^- & V^+ be its adjacent vertices. Let H^+ be the half plane to the left of VV^+ and H^- be another half plane to the left of V^-V.

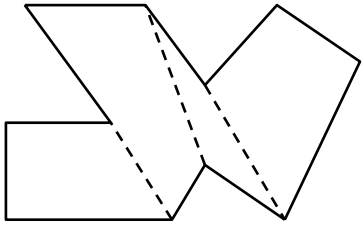
Observe that there can be at most one essential diagonal in H^+. If there two, one can be removal without creating a non-convex piece at V. similarly, there can be at most one

essential diagonal in H^-. Both of the half planes together cover interior angles at V; this implies that there can be at most two diagonals essential at reflex vertex V.

Optimal Convex Partitioning:

Theorem: Let ϕ be the fewest number of convex pieces into which a polygon may be partitioned. For a polygon of r-reflex vertices, $\lceil r/2 \rceil + 1 \leq \phi \leq r + 1$

Proof:



(Here, r = 3 so, at most r + 1 = 3 + 1 = 4 pieces.)

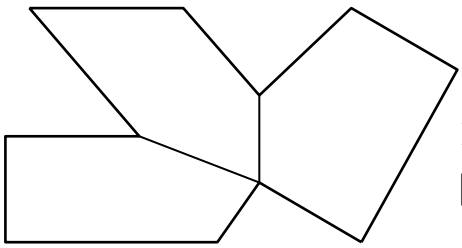
If bisector of each reflex vertex is drawn, it removes all the reflex angles and creates the convex pieces. If r be the number of reflex vertices, then the number of convex pieces will be at most r + 1 when (individual) bisectors of each reflex angle are drawn. If a single (common) bisector is used for resolving two reflex vertices, then ϕ may be less than r+1. But at most two reflex vertices can be resolved into non-reflex by a single bisector (partitioning segment).

Thus,

$\lceil r/2 \rceil + 1$ convex pieces will be obtained

$\therefore \lceil r/2 \rceil + 1 \leq \phi \leq r + 1$

Proved



Here, r = 3,

$\lceil r/3 \rceil + 1 \leq 2 + 1 < r + 1 = 4$

Theorem:

The Hertel's Mehlhorn's algorithm is never worse than four times optimal in the number of convex pieces.

Proof:

Let r be the number of reflex vertices for any non-convex simple polygon. Since, every reflex vertex can have at most two essential diagonals. The maximum number of essential diagonals for convex partitioning will be 2r.

Let M be the number of convex pieces obtained by the H-M algorithm.

Then, $M \leq 2r + 1$ (as 2r bisectors at most)

Since, $\phi \geq \lceil r/2 \rceil + 1$, where ϕ is optional number of convex pieces.

Clearly, $4\phi \geq 2r + 4$

$4\phi > 2r + 1$

But $M \leq 2r + 1$

So, $M \leq 2r + 1 < \phi$

This means, number of convex pieces obtained by the HM algorithm is les than four times the optimal number of convex pieces.