Triangulation de polygone, problème de la galerie d'art.

28839 PÉRAUD Arthur

Épreuve de TIPE

2022-2023

Introduction

- Définition 1

Une triangulation d'un polygone P est une partition de P en un ensemble de triangles qui ne se recouvrent pas, et dont l'union est P.

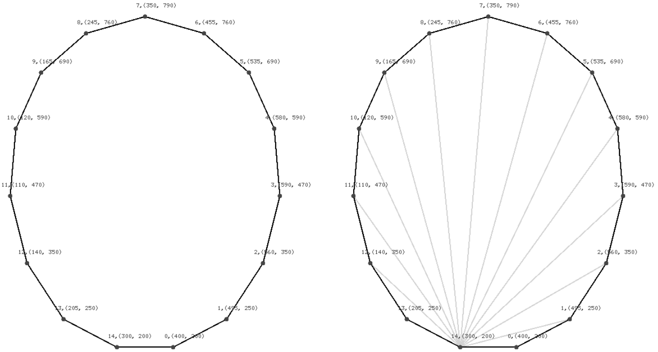


Fig 1 : Pentadécagone triangulé

Introduction

Problématique : Comment placer des caméras afin de couvrir une surface donnée de façon optimale ?

Sommaire

1 - Polygone convexe
- a) Nombre de triangulations
- b) Nombre de Catalan

2 - Triangulation de polygone simple
- a) Méthode des oreilles
- b) Décomposition Monotone

3 - Problème de la galerie d'art
- a) 3-Coloriage
- b) Application

- Annexe

Sommaire

1 - Polygone convexe
- a) Nombre de triangulations
- b) Nombre de Catalan

2 - Triangulation de polygone simple

3 - Problème de la galerie d'art

- Annexe

- Définition 2

Un polygone P est dit convexe si l'ensemble des points dans P est un ensemble convexe, ou bien que les angles intérieurs de P sont inférieurs à π, autrement il est qualifié de concave.

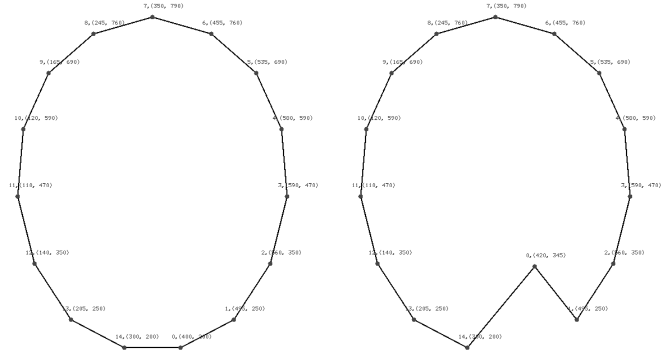


Fig 2 : Pentadécagone convexe, concave

Nombre de triangulations

Soit $n > 2$

On pose C_n le nombre de triangulations d'un polygone convexe à $n + 2$ sommets.
Il est clair que $C_1 = 1$ et $C_2 = 2$.

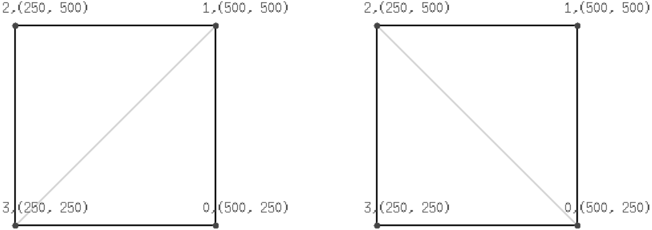


Fig 3 : Différentes triangulations d'un carré

Nombre de Triangulations

Cherchons une relation de récurrence permettant de calculer le nombre de triangulations d'un polygone convexe pour tout $n > 2$.

Comptons-les sur l'exemple en figure 1, $n = 15$, $C_{13} = ?$.

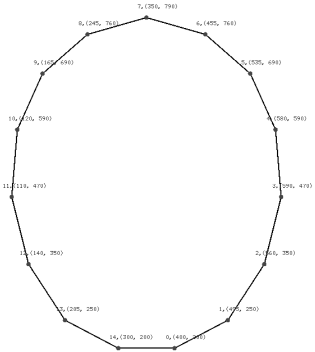
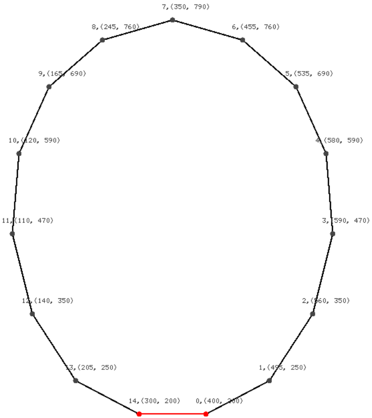


Fig 1

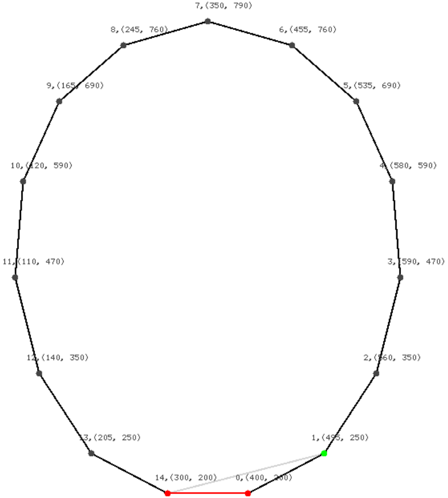
Nombre de Triangulations

Pour cela, choisissons une arête comme base, à partir de celle-ci, on peut construire différents triangles en la reliant à un sommet du polygone. Prenons l'arête **14-0**.

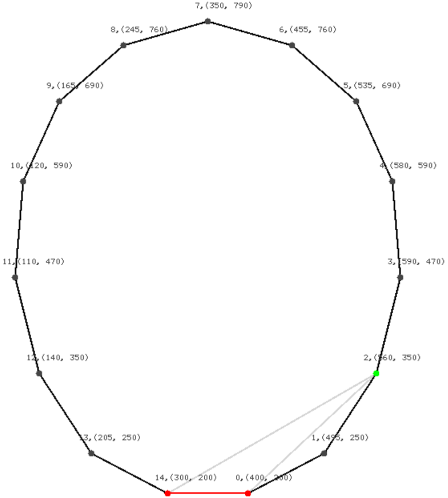


Nombre de Triangulations

Formons le triangle à l'aide du sommet **1**.
Dans ce cas, il y a C_{12} triangulations possibles.

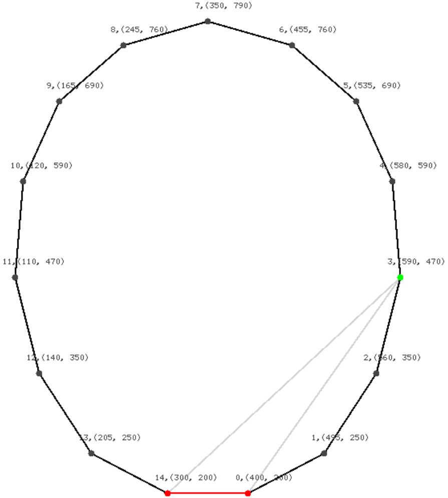


Formons le triangle à l'aide du sommet **2**.
Dans ce cas, il y a $C_{11}C_1$ triangulations possibles.



Nombre de Triangulations

Formons le triangle à l'aide du sommet **3**.
Dans ce cas il, y a $C_{10}C_2$ triangulations possibles.



Nombre de Triangulations

Prenons la convention $C_0 = 1$, ainsi à chaque fois il y a $C_k C_{n-k}$ triangulations possibles. En sommant le tout, on obtient :

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k} \tag{1}$$

Il s'agit de la relation de récurrence vérifiée par les nombres de Catalan.

Nombre de Catalan

Les 18 premiers nombres de Catalan

```
utop # List.init 18 catalan_recurrence;;
- : int list =
[1; 1; 2; 5; 14; 42; 132; 429; 1430; 4862; 16796; 58786; 208012; 742900;
 2674440; 9694845; 35357670; 129644790]
```

Source : annexe

Nombre de Catalan

- Déterminons le nombre de Catalan :

On pose : $f(x) = \sum_{n=0}^{\infty} C_n x^n, \ \forall x \in D(0,R)$

À l'aide d'un produit de Cauchy, on a pour $x \neq 0$:

$$f(x)^2 = \frac{f(x) - 1}{x} \tag{2}$$

$$f(x) = \frac{1 - \sqrt{1 - 4x}}{2x} \tag{3}$$

Par continuité en 0.

Nombre de Catalan

- Par le calcul on trouve un dse_0 de $\frac{1-\sqrt{1-4x}}{2x}$

$$\forall x \in D(0, \frac{1}{4}), \quad \frac{1-\sqrt{1-4x}}{2x} = \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2n}{n} x^n \qquad (4)$$

C_n définit aussi le nombre de façons de parenthenser les mots de taille $2n$, on obtient ainsi une majoration de C_n, donc une minoration de son rayon $R > 0$.

Nombre de Catalan

- On obtient donc :

$$\forall n \geqslant 0, \quad C_n = \frac{1}{n+1}\binom{2n}{n} \tag{5}$$

Sommaire

1 - Polygone convexe

2 - Triangulation de polygone simple
- a) Méthode des oreilles
- b) Décomposition Monotone

3 - Problème de la galerie d'art

- Annexe

Polygone simple

- Définition 3

Considérons un polygone P à n sommets, il est dit simple si ses n côtés ne se croisent pas et tels que deux côtés consécutifs n'ont qu'un seul point commun.

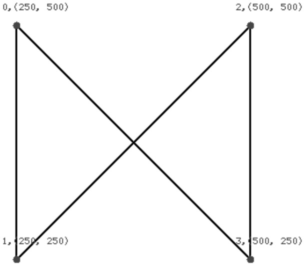


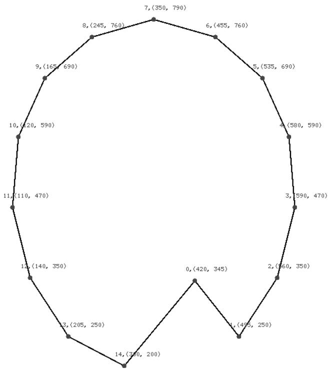
Fig 4 polygone non simple

Méthode des oreilles

- Définition 4

L'oreille d'un polygone est un triangle dont deux des côtés sont des cotés du polygone et dont le troisième côté est situé à l'intérieur du polygone.

Ex : 13.14.0 est une oreille, 14.0.1 et 0.2.3 ne le sont pas. (Fig 2)



Méthode des oreilles

- Proposition (admise)

Tout polygone
simple P avec $n > 3$ sommets possède au moins 2 oreilles distinctes.

- De par la méthode des oreilles, il en découlera que tout polygone simple admet une triangulation de $n-2$ triangles.

Méthode des oreilles

Algorithme earclipping :

Entrée : P un polygone à n sommets
Sortie : Triangulation de P sous une liste de triangle.

Soit L une liste vide

1. On cherche une oreille A.B.C de P qu'on ajoute dans L.

Méthode des oreilles

Algorithme earclipping :

Entrée : P un polygone à n sommets
Sortie : Triangulation de P sous une liste de triangle.

Soit L une liste vide

1. On cherche une oreille A.B.C de P qu'on ajoute dans L.
2. On enlève le sommet B de P.

Méthode des oreilles

Algorithme earclipping :

Entrée : P un polygone à n sommets
Sortie : Triangulation de P sous une liste de triangle.

Soit L une liste vide

1. On cherche une oreille A.B.C de P qu'on ajoute dans L.
2. On enlève le sommet B de P.
3. On réitère jusqu'à triangulation.

Méthode des oreilles

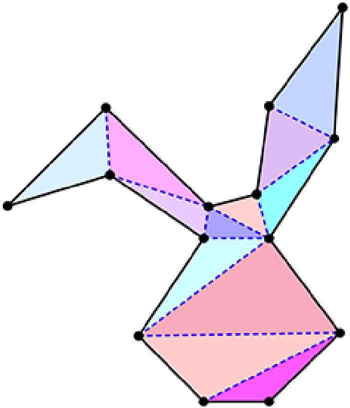


Fig 5 : polygone simple triangulé par amputations successives d'oreilles. Source : Tangente

Méthode des oreilles

- Définition 5

Un sommet est dit convexe si l'angle formé au niveau de son sommet dans P est plus petit que π concave sinon.

Ex : 3 est convexe et 5 est concave.

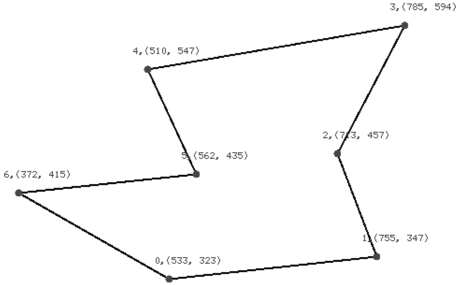


Fig 6 : polygone simple

Méthode des oreilles

Implémentation en Ocaml :

```
type point = int * int
type polygon = point list
type triangle = point * point * point

type vertex = {
    pos : point ;
    mutable convex : bool ;
    mutable concav : bool ;
    mutable ear : bool
}
```

Méthode des oreilles

On trouve une oreille de la façon suivante :

Soit 3 sommets consécutifs v_{i-1}, v_i, v_{i+1}, ils forment une oreille si :

Méthode des oreilles

On trouve une oreille de la façon suivante :

Soit 3 sommets consécutifs v_{i-1}, v_i, v_{i+1}, ils forment une oreille si :

1. v_i est convex

Méthode des oreilles

On trouve une oreille de la façon suivante :

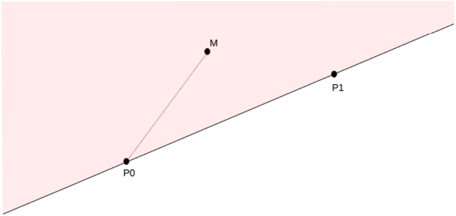
Soit 3 sommets consécutifs v_{i-1}, v_i, v_{i+1}, ils forment une oreille si :

1. v_i est convex
2. Aucun sommet concave du polygone n'est dans le triangle $v_{i-1}v_iv_{i+1}$ sauf possiblement v_{i-1} et v_{i+1}

Méthode des oreilles

Comment savoir si un sommet est concave ou convexe ?

- On prend pour convention que les sommets du polygone soient cycliques et ordonnés dans le sens trigonométrique.



Pour savoir de quel côté de la droite P_0P_1 le point M se trouve, on considère le produit vectoriel suivant :

$$L_z = (\overrightarrow{P_0P_1} \wedge \overrightarrow{P_0M}) \cdot \overrightarrow{u_z} \quad (1)$$

Source : MCOT[1]

Méthode des oreilles

1. $L_z > 0$ m se trouve à gauche du segment
2. $L_z < 0$ m se trouve à droite du segment
3. $L_z = 0$ m se trouve sur le segment

- À l'aide de L_z on peut facilement savoir si un sommet est convexe, et si un point se trouve à l'intérieur d'un triangle.

Méthode des oreilles

Complexité de l'algorithme :

Entrée : P un polygone à n sommets

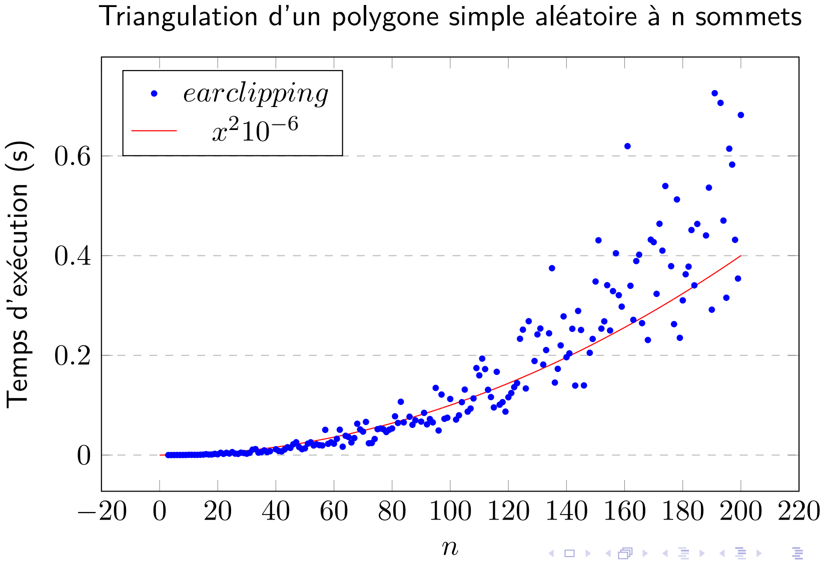
Sortie : Triangulation de P sous une liste de triangle.

Soit L une liste vide

1. On cherche une oreille A.B.C de P qu'on ajoute dans L. **O(n)**
2. On enlève le sommet B de P. **O(n)**
3. On réitère jusqu'à triangulation. ($n - 3$ fois) **O(n)**

L'algorithme est en **O(n²)**.

Méthode des oreilles



Décomposition Monotone

- Définition 5

Un polygone P est dit Y (resp X) monotone si chaque droite orthogonale à l'axe Y (resp X) coupe la frontière de P au plus deux fois.

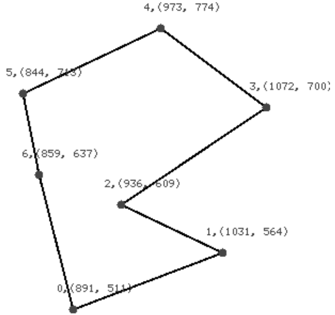


Fig 7 : polygone y-monotone

Décomposition Monotone

Traitons le cas des polygones y-monotones.

Soit P y-monotone.

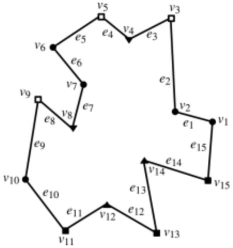
- Le déplacement sur la frontière gauche ou droite de P fait en sorte que nous descendons toujours du sommet le plus élevé de P au plus bas.

Cette propriété nous permet de le trianguler en $\mathbf{O(n)}$

Décomposition Monotone

- Partitionnement en sous-polygone monotone d'un polygone simple.

  - □ `start` : voisins d'ordonnées inférieures et angle intérieur inférieur à π,
  - ▲ `split` : idem `start` mais d'angle supérieur à π,
  - ■ `end` : voisins d'ordonnées supérieures et angle intérieur inférieur à π,
  - ▼ `merge` : idem `end` mais d'angle supérieur à π.



- Proposition

Tout polygone P est y-monotone s'il ne possède ni sommet split ni sommet merge.

- Ainsi, le partitionnement se fait en ajoutant des diagonales depuis les sommets split et merge à l'aide d'un algorithme de ligne de balayage qui se fait en **O(nlog n)**
- La décomposition monotone est donc en **O(nlog n)**

Sommaire

1 - Polygone convexe

2 - Triangulation de polygone simple

3 - Problème de la galerie d'art
a) 3-Coloriage
b) Application

- Annexe

Problème de la galerie d'art

« Quel est le nombre de caméras nécessaires pour surveiller une galerie d'art, et où faut-il les placer ? »

- Théorème

Pour garder un polygone simple à n sommets, $\lfloor n/3 \rfloor$ caméras suffisent, et cette borne peut être atteinte.

3-Coloriage

- Proposition (admise)

Tout polygone simple P à partir de sa triangulation T admet un 3-coloriage. $T = (S, A)$ avec S l'ensemble des sommets de P et A l'ensemble des arretes de la triangulation.

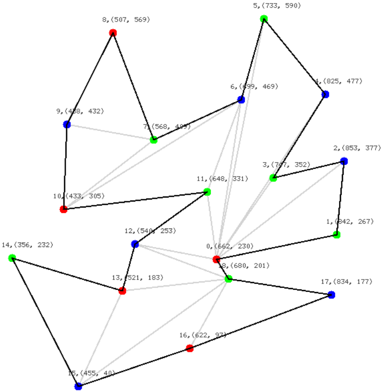


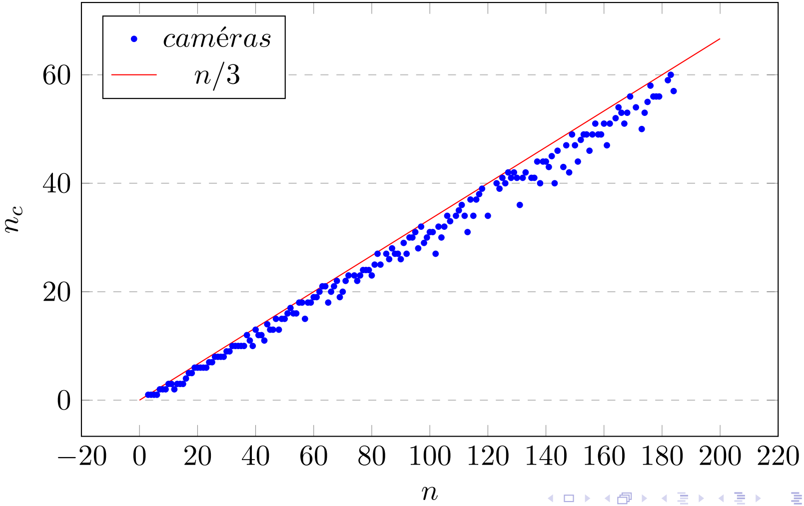
Fig 8 : graphe 3-colorié

3-Coloriage

Ainsi, en plaçant une caméra sur chaque sommet de la même couleur, on couvre le polygone. En choisissant la couleur qui minimise le nombre de caméras, le nombre de caméras sera majoré par $\lfloor n/3 \rfloor$.

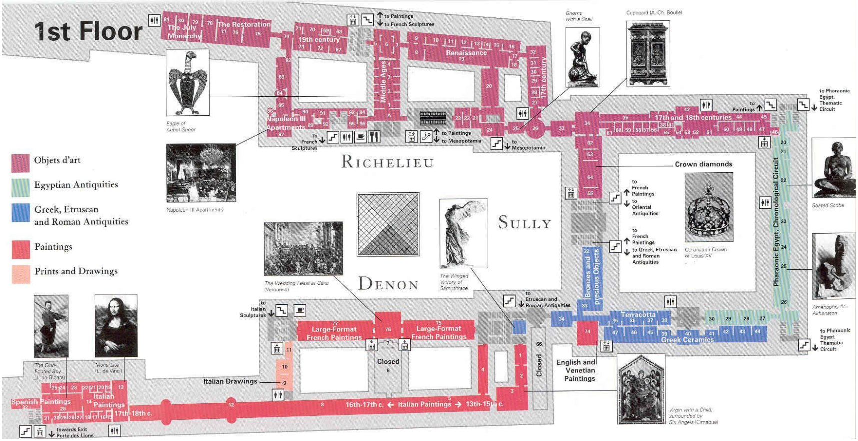
3-Coloriage

Nombre de caméras pour un polygone simple aléatoire à n sommets



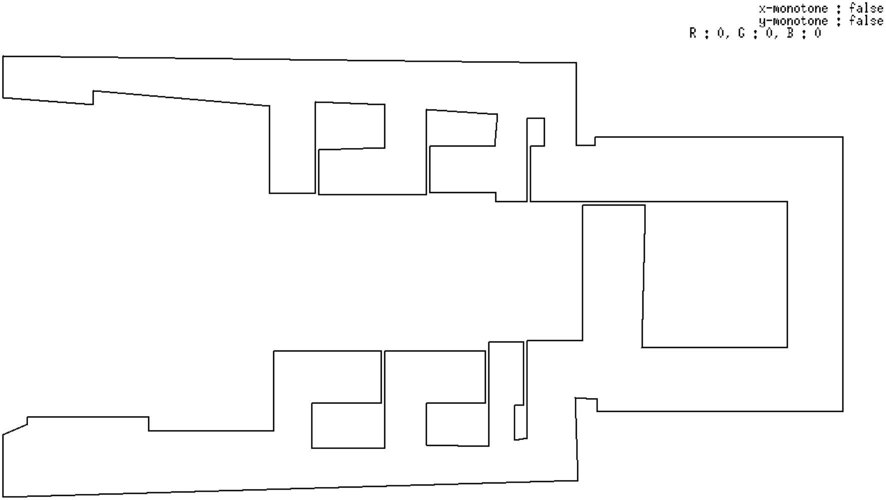
Application

Plan Musée du Louvre

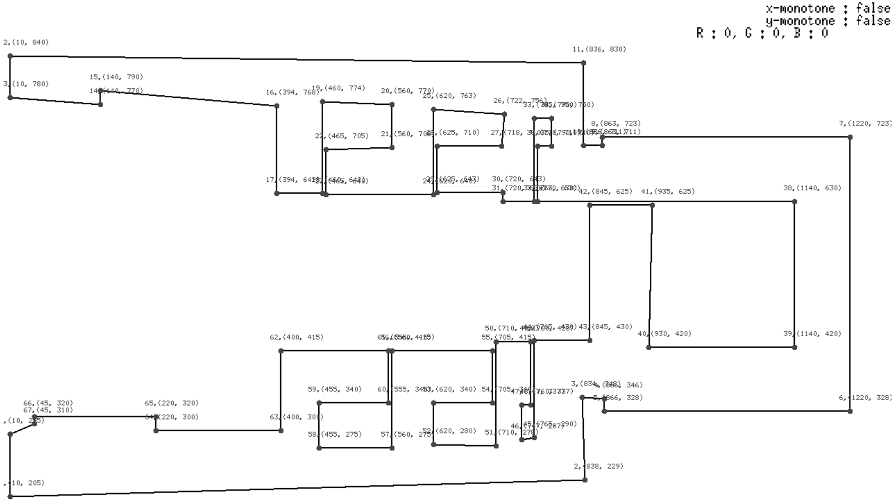


Source : plandeparis.info

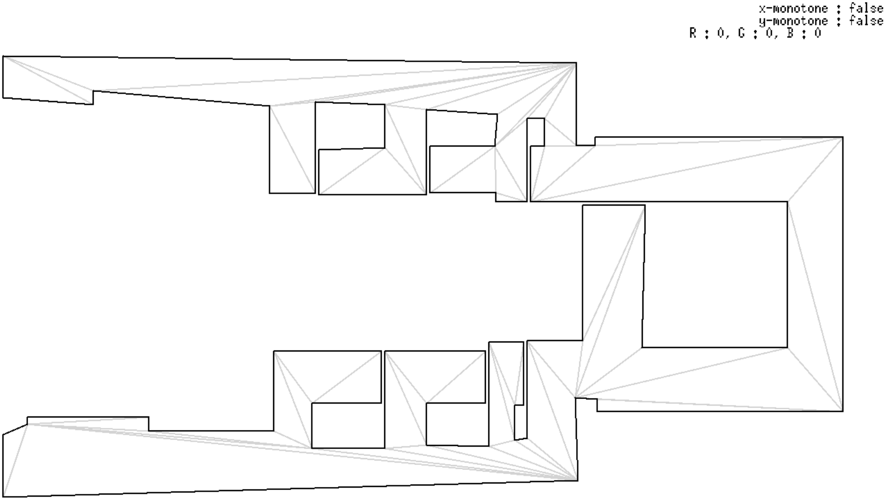
Application



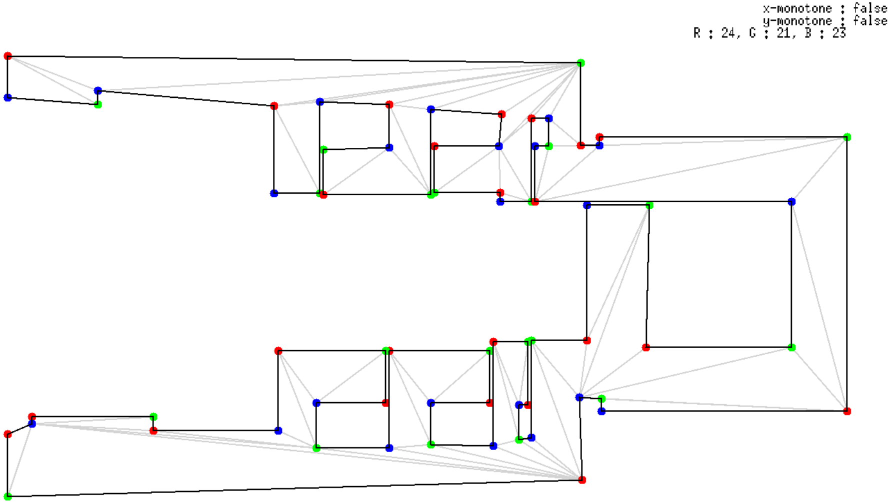
Application



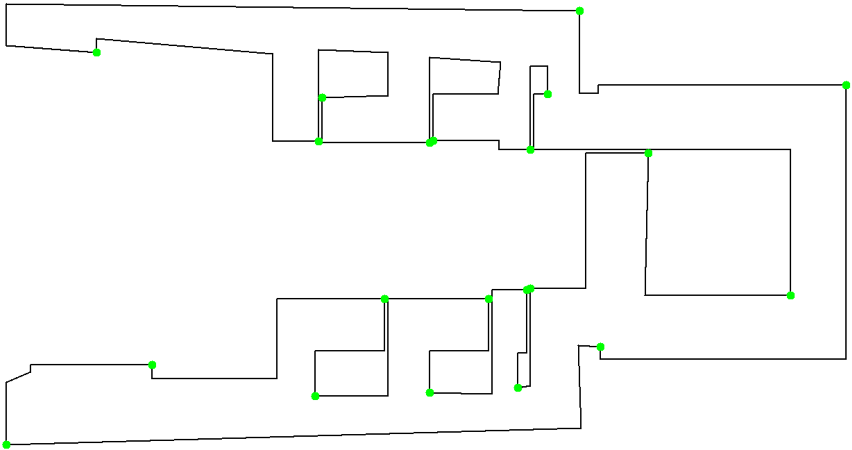
Application



Application



Application



On peut se ramener à $21 - 5 = 16$ caméras pour $68 - 14 = 54$ sommets.

MERCI POUR VOTRE ATTENTION

Sommaire

1 - Polygone convexe

2 - Triangulation de polygone simple

3 - Problème de la galerie d'art

- Annexe

Programme Informatique

Liste du code utilisé :

- Catalan.ml
- Affichage.ml (fichier principal)
- Triangulation.ml
- Coloriage.ml
- Arbre_BR.ml (Structure pour la décomposition monotone)
- random_poly.py (Utilisé avec graphe.ml pour générer les données utiles à la présentation)
- graphe.ml

Catalan.ml

CATALAN.ML

```
open Printf
open Float
open Sys

(*********************
    Nombres de Catalan
***********************)

(** int -> int
renvoie le n ème nombre de Catalan à l'aide de  la formule de récurrence **)
let catalan_recurrence n =
  (* On somme de k=0 à n les Ck*Cn-k *)
  let rec cn_plus_1 n k acc =
    if n = -1 then acc + 1 else
    match k with
    |_ when k = n+1 -> acc
    |_ ->
      begin (* tmp = Ck * Cn-k *)
        let tmp = (cn_plus_1 (k-1) 0 0) * (cn_plus_1 (n-k-1) 0 0) in
        cn_plus_1 n (k+1) (acc + tmp)
      end
  in cn_plus_1 (n-1) 0 0


(** int -> int
renvoie le n ème nombre de Catalan à l'aide la formule avec coeff binomiaux **)
let rec catalan_binomial n =
  match n with
  |0 -> 1
    |_ -> (2*(2*n-1) * catalan_binomial (n-1) ) / (n+1)


(** int -> float
équivalent asymptotique en utilisant la Formule de Stirling **)
let catalan_asymptotique n =
  (* a = 4^n , b = n^3/2 * sqrt(pi) *)
```

```
    let a = (Float.pow 4. (float_of_int n) ) in
    let b = ( (Float.pow (float_of_int n) (3./.2.) ) *. (Float.sqrt (4. *. (Float.atan 1.) ) ) ) in
    int_of_float (a /. b)


(*  'a -> 'b) -> 'a -> 'b * float
renvoie le résultat de la fonction appliqué à n et son temps d'éxécution *)
let time f n =
  let t1 = Sys.time() in
    let a = f n in
    let t2 = Sys.time() in
      (a, t2 -. t1)


(* int -> unit *)
let affiche n =
  let affiche_aux f =
    let (a,b) = time f n in
    Printf.printf ": %d               C : %f\n" a b
  in
  print_string "Catalan_rec : " ; affiche_aux catalan_recurrence ;
  print_string "Catalan_bin : " ; affiche_aux catalan_binomial ;
  print_string "Catalan_asy : " ; affiche_aux catalan_asymptotique
```

AFFICHAGE.ML

fichier main (on compile de la façon suivante : ocamlfind ocamlc -package graphics -linkpkg Arbre_BR.ml Triangulation.ml Coloriage.ml Affichage.ml)

```
open Graphics
open Printf
open Triangulation
open Coloriage

let default_color = Graphics.black
let grey_color = (rgb 211 211 211)
let light_black_color = (rgb 69 69 69)
let center = (320, 240)

(* point -> int -> unit *)
let draw_point ((a, b) : point) color size =
  Graphics.set_color color ;
  Graphics.fill_circle a b size

let draw_point_coordinates_order ((a, b) : point) color size i =
  draw_point (a,b) color size ;
  Graphics.moveto (a - 15) (b + 15) ;
  Graphics.draw_string (Printf.sprintf "%d,(%d, %d)" i a b )

let draw_line (a, b) (c, d) color =
  Graphics.set_color color ;
  Graphics.moveto a b ; Graphics.lineto c d

(** polygon -> int -> unit
dessine un polygone à partir d'une liste de points **)
let draw_polygon (lst : polygon) color =
  Graphics.set_color color ;
  let rec aux lst (a, b) =
    match lst with
    |[] -> ()
    |(x, y) :: tl -> Graphics.moveto a b ; Graphics.lineto x y ; aux tl (x, y)
  in
  let h = List.hd lst in
  aux (lst @ [h]) h
```

```

(** polygon -> int -> unit **)
let draw_triangle (tri : triangle) color =
  let (a, b, c) = tri in
  draw_polygon [a; b; c] color


(* point list -> unit (même idée que draw_polygon)
Dessine les lignes pour la fonction display_draw() lorsqu'on place les sommets dans la fenêtre graphique *)
let show_drawing_polygon lst color =
  let rec aux lst (a, b) =
    match lst with
    |[] -> ()
    |(x, y) :: tl ->
      begin
        draw_point (a, b) light_black_color 3 ;
        Graphics.set_color color ;
        Graphics.moveto a b ;
        Graphics.lineto x y ; aux tl (x, y)
      end
  in
  match lst with
  |[] -> ()
  |[x] -> draw_point x light_black_color 3
  |_ -> aux lst (List.hd lst)


(* polygon -> (bool * int array) (= resultat de 3color_graphe ) -> unit
Dessine les sommets du polygone 3-colorés après triangulation *)
let draw_coloring (lst : polygon) (colorable, colours) =
  let vertices = Array.of_list lst in

  if not(colorable) then failwith "Pas 3 coloriable ?" else
    for i = 0 to List.length lst -1 do
      match colours.(i) with
```

```
        |1 -> draw_point vertices.(i) Graphics.red 6
        |2 -> draw_point vertices.(i) Graphics.green 6
        |3 -> draw_point vertices.(i) Graphics.blue 6
        |_ -> ()
      done

  let show_explanations() = ""

  (** unit -> unit
  Permet le dessin d'un polygone à trianguler dans la fenêtre puis le trianguler etc ... **)
  let display_draw() =
  begin
    let vertices = ref [] in
    let triangulated = ref [] in (* On garde en mémoire la triangulation *)
    let coloriage = ref (false, [||]) in (* Idem on garde en mémoire le coloriage *)

    let draw_fun = ref show_drawing_polygon in
    let tmp_fun = ref show_drawing_polygon in (* Pour affichage des coordonnées des points 'p' *)
    let f() = !draw_fun !vertices default_color in (* Superpose plusieurs affichages *)

    let drawn_once = ref 0 in (* Pour 'd' ne pas rev la liste si elle l'a déja été *)
    let c_pressed = ref false in (* pour ne pas superposer un point coloré et un point noir *)
    let key_pressed = ref false in (* Pour ne plus pouvoir dessiner après construction du polygone
  ↪  *)
    let number_pressed = ref false in
    let monotone = ref (false, false) in
    let cred = ref 0 and cgreen = ref 0 and cblue = ref 0 in

    let rec display f =
    try
      let e = Graphics.wait_next_event [Graphics.Mouse_motion; Graphics.Button_down; Graphics.
     ↪  Key_pressed] in
      let mouse_description = Printf.sprintf "Mouse position : (%d, %d)" e.mouse_x e.mouse_y in

        clear_graph() ; f() ;

```

```
      (* affiche la position du curseur *)
      Graphics.moveto 0 0 ;
      Graphics.set_color default_color ;
      Graphics.set_font "-*-fixed-medium-r-semicondensed--18-*-*-*-*-*-iso8859-1" ;
      Graphics.draw_string mouse_description ;

      let is_mx = Printf.sprintf "x-monotone : %b" (fst !monotone) in
      let is_my = Printf.sprintf "y-monotone : %b" (snd !monotone) in
      Graphics.set_font "-*-fixed-medium-r-semicondensed--20-*-*-*-*-*-iso8859-1" ;
      Graphics.moveto 1100 900 ;
      Graphics.draw_string is_mx ;
      Graphics.moveto 1100 882 ;
      Graphics.draw_string is_my ;
      Graphics.moveto 1000 864 ;
      Graphics.draw_string (Printf.sprintf "R : %d, G : %d, B : %d" !cred !cgreen !cblue) ;
      (* show_explanations() ; *)

      let pos = (e.mouse_x, e.mouse_y) in

      if e.button && not(!key_pressed) then
      begin
        draw_point pos Graphics.black 4 ;
        vertices := pos :: !vertices ;
      end;

    match e.key with
    |'1' -> vertices := louvre ; number_pressed := true ; display f
    |'2' -> vertices := pentadecagone ; number_pressed := true ; display f
    |'3' -> vertices := List.map (fun (x,y) -> (x*2,y*2)) polygone_xi; number_pressed := true ;
        display f
     (* Presets *)
    |'d' ->
     (* done/draw, fin du placement des points pour le dessins du polygone *)
      begin
        if !drawn_once = 0 && not(!number_pressed) then vertices := List.rev !vertices ;
        draw_fun := draw_polygon ;
```

```
          tmp_fun := draw_polygon ;
          c_pressed := false ;
          key_pressed := true ;
          incr drawn_once ;
          monotone := (is_monotone !vertices X, is_monotone !vertices Y) ;
          display f
        end
      |'t' ->
       (* triangulate *)
        begin
          triangulated := ear_clipping !vertices ;
          (* v c ne servent à rien juste une question de type, considérer triangulate(void) *)
          let rec triangulate v c =
            List.iter (fun x -> draw_triangle x grey_color) !triangulated ;
            draw_polygon !vertices Graphics.black ;
          in
          draw_fun := triangulate ;
          tmp_fun := triangulate ;
          c_pressed := false ;
          key_pressed := true ;
          display f
        end
      |'c' ->
       (* 3-color *)
        begin
          let graphe = make_graph_from_triangulation !vertices !triangulated in
          coloriage := three_color_graph graphe ;
          (* v c ne servent à rien juste une question de type, considérer
          ↪  triangulate_and_color(void) *)
          let rec triangulate_and_color v c =
            List.iter (fun x -> draw_triangle x grey_color) !triangulated ;
            draw_coloring !vertices !coloriage ;
            draw_polygon !vertices Graphics.black ;
          in
          let (red, green, blue) = count_colour !vertices in
          cred := red ; cgreen := green ; cblue := blue ;
```

```
            draw_fun := triangulate_and_color ;
            tmp_fun := triangulate_and_color ;
            c_pressed := true ;
            key_pressed := true ;
            display f
          end
        |'r' ->
        (* refresh *)
          begin
            List.iter (fun (a, b) -> Printf.printf "(%d, %d); " a b) !vertices ; print_string "\n\n" ;
            vertices := [] ;
            triangulated := [] ;
            draw_fun := show_drawing_polygon ;
            tmp_fun := show_drawing_polygon ;
            c_pressed := false ;
            key_pressed := false ;
            number_pressed := false ;
            monotone := (false, false) ;
            cred := 0 ; cgreen := 0 ; cred := 0 ;
            display f
          end
        |'p' ->
        (* points coordinates + order *)
          begin
            let rec show_coordinates_order v c =
              !tmp_fun !vertices default_color ;
              let count = ref 0 in
              Graphics.set_font "-*-fixed-medium-r-semicondensed--11-*-*-*-*-iso8859-1" ;

              if !c_pressed then
                List.iter (fun x -> draw_point_coordinates_order x light_black_color 0 !count ; incr count) !vertices
              else
                List.iter (fun x -> draw_point_coordinates_order x light_black_color 4 !count ; incr count) !vertices
```

```ocaml

        in
        draw_fun := show_coordinates_order ;
        key_pressed := true ;
        display f ;
        c_pressed := false
      end
    |'q' -> Graphics.close_graph()
    (* quit *)
    |_ -> display f
  with
  |Failure _ -> Printf.printf "------------------ERROR---------------------\n" ;
  in
display f ;
List.iter (fun (a, b) -> Printf.printf "(%d, %d); " a b) !vertices ; print_string "\n"
end


(*** MAIN FUNCTION ***)
let main() = begin

  Graphics.open_graph " 1280x960" ;
  Graphics.set_window_title " Polygon triangulation " ;
  Graphics.set_line_width 2 ;
  Graphics.set_color default_color ;
  Graphics.set_font "-*-fixed-medium-r-semicondensed--18-*-*-*-*-*-iso8859-1" ;

  display_draw()
end
let _ = main()

(* ocamlfind ocamlc -package graphics -linkpkg Triangulation.ml Coloriage.ml Affichage.ml *)
```

Triangulation.ml

TRIANGULATION.ML

```
open Array
open Stack
open List
open Arbre_BR

type point = int * int

type polygon = point list

type triangle = point * point * point

let (mod) x y = ((x mod y) + y) mod y

(* 'a -> 'a list -> int *)
let pos_lst e lst =
  let rec aux l count =
  match l with
  |[] -> failwith "not in list"
  |hd :: tl -> if hd = e then count else aux tl (count+1)
in aux lst 0

(*************************
  Méthode des oreilles
*************************)

(** point -> point -> point -> int
Renvoie (p1p2  p1m).uz si > 0 m se trouve à gauche du segment, < 0 à droite, = 0 sur le segment
 ↪  **)
let position_to_line (p1 : point) (p2 : point) (m : point) =
  let (a, b) = p1 and (c, d) = p2 and (x, y) = m in
  (c - a)*(y - b) - (d - b)*(x - a)


(** polygon -> point -> bool **)
let in_triangle (triangle : polygon) (m : point) =
  (* precedent = point precedent et verifie que position_to_line sur m pour les trois segments du
 ↪  triangle soit > 0*)
```

```ocaml
  let rec aux lst precedent acc =
    match lst with
    |[] -> acc
    |h :: t -> aux t h (acc && ( (position_to_line precedent h m) > 0) )
  in
  let h = List.hd triangle in
  aux ( (List.tl triangle) @ [h] ) h true

(* Type pour définir les points *)
type vertex = {
  pos : point ;
  mutable convex : bool ;
  mutable concav : bool ;
  mutable ear : bool
}

(** int -> polygon -> bool * triangle
Renvoie si le point k est une oreille de "son triangle" **)
let is_ear k (lst : polygon) =
  let n = List.length lst in
  let v_prop =
    Array.of_list ( List.map ( fun x -> {pos = x; convex = false; concav = false; ear = false} ) lst )
  in
  for i = 0 to n-1 do
    v_prop.(i).convex <- (position_to_line v_prop.((i-1) mod n).pos v_prop.((i+1) mod n).pos v_prop.(i).pos) < 0 ;
    v_prop.(i).concav <- not(v_prop.(i).convex)
  done;
  if v_prop.(k).convex then
    begin
      let triangle = [v_prop.((k-1) mod n).pos; v_prop.(k).pos; v_prop.((k+1) mod n).pos] in
    v_prop.(k).ear <- not( Array.exists (fun x -> (in_triangle triangle x.pos) ) v_prop )
    end;
  ( v_prop.(k).ear, (v_prop.((k-1) mod n).pos, v_prop.(k).pos, v_prop.((k+1) mod n).pos) )
```

```
(* A faire : prendre en compte les angles afin d'avoir une triangulation "plus belle" peut etre ? *)

(** polygon -> triangle list **)
let ear_clipping (lst : polygon) =
  let n = List.length lst in
  let rec triangulate (l : polygon) (acc : triangle list) count =
    let h = List.length l in
    if count = n - 2 then acc else
      (* int -> polygon -> (int * triangle) ; parcours la liste l' afin de trouver une oreille *)
      let rec find_ear i (l' : polygon) =
        match i with
        |x when x = h -> failwith "pas d'oreilles"
        |_ ->
          let (a, b) = is_ear i l' in
          if a then (i, b) else find_ear (i+1) l'
      in
    let (a, b) = find_ear 0 l in
    (* On enlève l'oreille et on itère *)
    triangulate (List.filter (fun x -> x <> List.nth l a) l ) (b :: acc) (count +1)
  in
  triangulate lst [] 0

(************************
  Polygones Monotones
************************)

(* Type pour définir les points *)
type vertex_type =
|Start
|End
|Regular
|Split
|Merge
|Ndefined
```

```ocaml

type axis = X | Y

(* point1, point2, indice de l'arete *)
type edge = point * point * int

type status_line = abr ref

(*****************************
  Décomposition Monotone
*****************************)

(** Renvoie l'indice des points (min, max) par rapport à X ou Y **)
let find_indice_min_max (l1 : polygon) (ax : axis) =
  let f = match ax with
    |X -> fst
    |Y -> snd
  in
  (* (x,y) : point * point *)
  let rec aux l (x, y) (posx, posy) count =
    match l with
    |[] -> (posx, posy)
    |hd :: tl ->
      begin
        if f x >= f hd then aux tl (hd , y) (count, posy) (count+1) else
        if f y <= f hd then aux tl (x, hd) (posx, count) (count+1) else
        aux tl (x, y) (posx, posy) (count+1)
      end
  in
  aux l1 (List.hd l1, List.hd l1) (0, 0) 0

(* *)
let compare p1 p2 =
  match p1, p2 with
  |(_, b), (_, d) when b > d -> -1
  |(_, b), (_, d) when b < d -> 1
```

```
  |(a, b), (c, d) when b = d  && a < c -> -1
  |(a, b), (c, d) when b = d  && a > c -> 1
  |_, _ -> 0

(* ('a * 'b) list -> axis -> ('a * 'b) list (tri fusion) *)
let laxis_sorted lst axis =
  match axis with
  |X -> List.sort (fun (x,y) (x',y') -> compare (y,x) (y',x')) lst
  |Y -> List.sort (fun (x,y) (x',y') -> compare (x,y) (x',y')) lst

let laxis_sorted_w_types lst axis =
  match axis with
  |X -> List.sort (fun ((x,y), t1) ((x',y'), t2) -> compare (y,x) (y',x')) lst
  |Y -> List.sort (fun ((x,y), t1) ((x',y'), t2) -> compare (x,y) (x',y')) lst

(** Ne Fonctionne pas **)
let make_monotone (lst : polygon) (ax : axis) =

  let n = List.length lst in
  let tree = ref Nil in
  let diag = ref [] in

  (* fonction qui renvoie (first, second) selon X ou Y *)
  let f = match ax with
    |X -> fst
    |Y -> snd
  in
  let vertices =
    Array.of_list ( List.map ( fun x -> {pos = x; convex = false; concav = false; ear = false} )
   ↪  lst )
  in
  let edges = Array.make n ((0,0), (0,0), 0) in
    for i = 0 to n-1 do
      edges.(i) <- (vertices.(i).pos , vertices.((i+1) mod n).pos, i)
    done;
  (* point list -> list *)
```

```
    let set_v_types l =
      for i = 0 to n-1 do
        vertices.(i).convex <- (position_to_line vertices.((i-1) mod n).pos vertices.((i+1) mod n).pos vertices.(i).pos) < 0 ;
        vertices.(i).concav <- not(vertices.(i).convex)
      done;
      let rec aux i acc =
        if i = n then acc else
        let v = vertices.(i).pos and vg = vertices.((i-1) mod n).pos and vd = vertices.((i+1) mod n).pos in
        match vertices.(i).convex with
        |true when (f vg) < (f v) && (f vd) < (f v) -> aux (i+1) ((vertices.(i).pos, Start) :: acc)
        |false when (f vg) < (f v) && (f vd) < (f v) -> aux (i+1) ((vertices.(i).pos, Split) :: acc)
        |true when (f vg) > (f v) && (f vd) > (f v) -> aux (i+1) ((vertices.(i).pos, End) :: acc)
        |false when (f vg) > (f v) && (f vd) > (f v) -> aux (i+1) ((vertices.(i).pos, Merge) :: acc)
        |_ -> aux (i+1) ((vertices.(i).pos, Regular) :: acc)
      in aux 0 []
    in
    let pqueue = (laxis_sorted_w_types (set_v_types lst) ax) in
    let helper = Array.make n ((0,0), Ndefined) in
    let rec parcours l i =
    Printf.printf "%d\n" i ;
      match l with
      |[] -> !diag
      |(v, t) :: tl ->
          try
            begin
              let (x1, y) = v in
              let x = float_of_int x1 in
              tree := set_y_position !tree y ;
              match t with
              |Start   ->
                  (
```

```
                Printf.printf("Start\n");
                (* affiche_tree !tree ; *)
                tree := insert (x_intersection edges.(i) y, edges.(i)) !tree ;
                helper.(i) <- (v, t);
                parcours tl (i+1)
              )
            |End     ->
              (
                Printf.printf("End\n");
                (* affiche_tree !tree ; *)
                if snd helper.((i-1) mod n) = Merge then(
                  diag := (v, fst helper.((i-1) mod n)) :: !diag );
                tree := suppression edges.((i-1) mod n) !tree ;
                parcours tl (i+1)
              )
            |Split   ->
              (
                Printf.printf("Split\n");
                (* affiche_tree !tree ; *)
                let (p1, p2, j) = find_left_edge !tree x in
                diag := (v, fst helper.(j)):: !diag ;
                helper.(j) <- (v, t);
                tree := insert (x_intersection edges.(i) y, edges.(i)) !tree ;
                helper.(i) <- (v, t) ;
                parcours tl (i+1)
              )
            |Merge   ->
              (
                Printf.printf("Merge\n");
                (* affiche_tree !tree ; *)
                if snd helper.((i-1) mod n) = Merge then(
                  diag := (v, fst helper.((i-1) mod n)):: !diag );
                tree := suppression edges.((i-1) mod n) !tree ;
                let (p1, p2, j) = find_left_edge !tree x in
                if snd helper.(j) = Merge then(
                  diag := (v, fst helper.(j)):: !diag );
```

```
         helper.(j) <- (v, t) ;
         parcours tl (i+1)
        )
      |Regular ->
        (
         Printf.printf("Regular\n");
         (* affiche_tree !tree ; *)
         let (pvmax, pvmin) = find_indice_min_max lst ax in
         if not(pvmin < pvmax && i >= pvmin && i <= pvmax) then
         (
          if snd helper.((i-1) mod n) = Merge then
          diag := (v, fst helper.((i-1) mod n)):: !diag ;
          tree := suppression edges.((i-1) mod n) !tree ;
          tree := insert (x_intersection edges.(i) y, edges.(i)) !tree ;
          helper.(i) <- (v, t);
          parcours tl (i+1)
         )
         else
         (
          (* affiche_tree !tree ; *)
          let (p1, p2, j) = find_left_edge !tree x in
          if snd helper.(j) = Merge then(
           diag := (v, fst helper.(j)):: !diag );
          helper.(j) <- (v, t);
          parcours tl (i+1)
         )
        )
      |Ndefined -> failwith "zz"
     end
    with
    |Failure "arbre vide" -> parcours tl (i+1)
 in
 parcours pqueue 0

(********************************
 Triangularisation Monotone
```

```
*********************************)

(** polygon -> axis -> bool **)
let is_monotone (lst : polygon) (ax : axis) =
  let n = List.length lst in
  let f = match ax with
    |X -> fst
    |Y -> snd
  in
  (* c est la fonction de comparaison *)
  let parcours_chaine l1 i j c =
    let rec aux l pos acc precedent =
      match l with
      |_ when pos = j+1 -> acc
      |hd :: tl when pos <= i -> aux tl (pos +1) acc hd
      |hd :: tl -> (* Printf.printf "%b %d\n" (c precedent hd ) pos ;*) aux tl (pos+1) (acc &&
     ↪  (c precedent hd) ) hd
      |_ -> acc
    in aux l1 0 true (List.hd l1)
  in
  let (a, b) = find_indice_min_max lst ax in
  (* Printf.printf "%d %d\n" a b ; *)
  let c1 = (fun x y -> f x <= f y) in
  let c2 = (fun x y -> f x >= f y) in
  if a < b then
    (parcours_chaine lst a b c1 ) && (parcours_chaine (lst@lst) b (n+a) c2 )
  else
    (parcours_chaine lst b a c2 ) && (parcours_chaine (lst@lst) a (n+b) c1 )

(* *)
let unstack_all s =
  let rec aux acc =
    if Stack.is_empty s then acc else aux (Stack.pop s :: acc)
  in aux []

(* *)
```

```
let test lst p1 p2 ax =
(
  let (spmin, spmax) =
    let (posmin, posmax) = find_indice_min_max lst ax in
    if posmin <= posmax then (posmin, posmax) else (posmax, posmin)
    in
  let pos_vtop = pos_lst p1 lst and pos_uj = pos_lst p2 lst in
  Printf.printf "(%d, %d) (%d, %d)\n" spmin spmax pos_vtop pos_uj ;

  (* si uj et le sommet sur la pile ne sont pas sur la meme chaine *)
  let b1 = (pos_vtop >= spmin && pos_vtop <= spmax) && (pos_uj >= spmin && pos_uj <= spmax) in
  let b2 = not((pos_vtop > spmin && pos_vtop < spmax) || (pos_uj > spmin && pos_uj < spmax)) in
  Printf.printf "%b %b %b\n" b1 b2 (not(b1 || b2)) ;
)

(** polygon -> axis -> triangle list
Ne Fonctionne pas **)
let triangulate_polygon_monotone (lst : polygon) (ax : axis) =
  if not(is_monotone lst ax) then failwith "polygon is not ax monotone" else

  let u = Array.of_list (laxis_sorted lst ax) in
  let s = Stack.create() in
    Stack.push u.(0) s ;
    Stack.push u.(1) s ;
  let d = ref [] in (* liste de diagonales *)

  let revf = match ax with
    |Y -> fst
    |X -> snd
  in
  (* unit -> lst *)
  let unstack_all st =
    let rec aux acc =
      if Stack.is_empty st then acc else aux (Stack.pop st :: acc)
    in aux []
  in
```

```
    (* ajoute les diagonales (u_i , elements de lst_popped) dans d *)
    let rec insert_diagonals l i =
      match l with
      |[] -> ()
      |hd :: tl -> d := (u.(i), hd) :: !d ; insert_diagonals tl i
    in
    for j = 2 to (Array.length u)-2 do
    (
      Printf.printf "%d\n" j ;
      (* valeur du sommet sur la pile *)
      let vtop = Stack.pop s in
             Stack.push vtop s ;

      (* Pour savoir si uj et le sommet sur la pile ne sont pas sur la meme chaine *)
      let (spmin, spmax) =
        let (posmin, posmax) = find_indice_min_max lst ax in
        if posmin <= posmax then (posmin, posmax) else (posmax, posmin)
        in
      let pos_vtop = pos_lst vtop lst and pos_uj = pos_lst u.(j) lst in
      let b1 = (pos_vtop >= spmin && pos_vtop <= spmax) && (pos_uj >= spmin && pos_uj <= spmax) in
      let b2 = not((pos_vtop > spmin && pos_vtop < spmax) || (pos_uj > spmin && pos_uj < spmax))
     ↪  in

      if not(b1 || b2) then
       (
        Printf.printf "a\n" ;
        let lst_popped = unstack_all s in
        insert_diagonals (List.tl lst_popped) j ;
        Stack.push u.(j-1) s ;
        Stack.push u.(j) s ;
       )
      else
       (
        Printf.printf "b\n" ;
        let ul = Stack.pop s in
        let vpop = ref ul in
```

```
        while (revf u.(j)) <= (revf !vpop) do
          d := (u.(j), !vpop) :: !d ;
          vpop := Stack.pop s ;
        done;
        Stack.push ul s ;
        Stack.push u.(j) s
       )
    )
   done;
   let l_S = Stack.length s and count = ref (-1) in
   (* on enlève le 1er et dernier élément *)
   let lst_popped = List.filter (fun x -> incr count ; !count <> 0 && !count <> l_S-1)
   ↪  (unstack_all s) in
     insert_diagonals lst_popped (Array.length u -1) ;
     !d

 (* Polygones construit dans le sens trigo *)

let polygone = [(100, 100); (200, 150); (350, 100); (400, 150); (450, 300); (350, 400); (250,
↪  350); (200, 400); (100, 350); (50, 250); (50, 150); (75, 100)]

let polygone_w = [(200,100); (250,200); (300,100); (350,200); (400,100); (450,200); (450,300);
↪  (400,400); (300,400); (250,350); (200,400); (100,400); (50,300); (50,200); (100,100)]

let polygone_xi = [(250, 350); (200, 300); (250, 250); (200, 200); (250, 150); (300, 200); (350,
↪  150); (400, 200); (350, 250); (400, 300); (350, 350)]

let pentadecagone = [(400, 200); (495, 250); (560, 350); (590, 470); (580, 590); (535, 690);
↪  (455, 760); (350, 790); (245, 760); (165, 690); (120, 590); (110, 470); (140, 350); (205,
↪  250); (300, 200)]
```

```
let louvre = List.map (fun (x,y) -> (x + 10, (960 - y)-100)) [(0, 565); (0, 655); (828, 631); (824, 512); (856, 514); (856, 532); (1210, 532); (1210, 137); (853, 137); (853, 149); (826, 149); (826, 30); (0, 20); (0, 80); (130, 90); (130, 70); (384, 92); (384, 218); (450, 218); (450, 86); (550, 90); (550, 152); (455, 155); (455, 220); (610, 220); (610, 97); (712, 104); (708, 150); (615, 150); (615, 217); (710, 217); (710, 230); (755, 230); (755, 110); (780, 110); (780, 150); (760, 150); (760, 230); (1130, 230); (1130, 440); (920, 440); (925, 235); (835, 235); (835, 430); (755, 430); (755, 570); (737, 573); (737, 523); (750, 523); (750, 432); (700, 432); (700, 582); (610, 580); (610, 520); (695, 520); (695, 445); (550, 445); (550, 585); (445, 585); (445, 520); (545, 520); (545, 445); (390, 445); (390, 560); (210, 560); (210, 540); (35, 540); (35, 550)]

let mono = [(690, 471); (736, 527); (664, 543); (627, 619); (675, 718); (564, 791); (506, 752); (508, 706); (558, 656); (499, 343); (677, 410)]
```

Coloriage.ml

COLORIAGE.ML

```
open Triangulation

(************************
      3-Coloriage
************************)

type graph = bool array array

(* polygon -> graph *)
let make_graph_from_triangulation (lst : polygon) (triangulated : triangle list) =
  let n = List.length lst in
  let vertices = Array.of_list lst in
  let graphe = Array.make_matrix n n false in
  let indice_point p =
    let rec loop i =
      if i > n then failwith "error" else
      if p = vertices.(i) then i else loop (i+1)
    in loop 0
  in
  (* *)
  let rec make_graphe l =
    match l with
    |[] -> ()
    |(a, b, c) :: tl ->
      begin
        let i_a = indice_point a and i_b = indice_point b and i_c = indice_point c in
          graphe.(i_a).(i_b) <- true ; graphe.(i_b).(i_a) <- true ;
          graphe.(i_a).(i_c) <- true ; graphe.(i_c).(i_a) <- true ;
          graphe.(i_b).(i_c) <- true ; graphe.(i_c).(i_b) <- true ;
          make_graphe tl
      end
  in make_graphe triangulated ;
  graphe


exception Break
```

```
(* graph -> bool * int array *)
let three_color_graph graphe =
  let n = Array.length graphe.(0) in
  let vertices_color = Array.make n 0 in
  (* *)
  let colorable v color =
    let rec loop i =
      match i with
      |x when x = n-1 -> true
      |_ when graphe.(v).(i) && color = vertices_color.(i) -> false
      |_ -> loop (i+1)
    in loop 0
  in
  (* *)
  let rec backtrack v =
    if v = n then true else
    try
      for color = 1 to 3 do (* On peut augmenter pour un k-Coloriage *)
        if(colorable v color) then
        begin
          vertices_color.(v) <- color ;
          if(backtrack (v+1)) then raise Break ;
          vertices_color.(v) <- 0 ;
        end
      done;
      false
    with
    |Break -> true
  in ( (backtrack 0) , vertices_color )

(* polygon -> int * int * int *)
let count_colour lst =
  let triangulated = ear_clipping lst in
  let graphe = make_graph_from_triangulation lst triangulated in
  let (colorable, colours) = three_color_graph graphe in
```

```
    if not(colorable) then failwith "pas coloriable" else
    let cred = ref 0 and cgreen = ref 0 and cblue = ref 0 in
    for i = 0 to List.length lst -1 do
      match colours.(i) with
      |1 -> incr cred
      |2 -> incr cgreen
      |3 -> incr cblue
      |_ -> ()
    done;
    (!cred, !cgreen, !cblue)

  (* int * int * int -> int *)
  let min_color (red, green, blue) = Stdlib.min (Stdlib.min red green) blue
```

ARBRE_BR.ML

```
(********************************
  Arbre binaire de recherche
********************************)

type point = int * int

type edge = point * point * int

type abr =
|Nil
|ABR of abr * (float * edge) * abr

type status_line = abr ref

(* *)
let rec affiche_tree a =
  match a with
  |Nil -> Printf.printf "Nil"
  |ABR(g,_ ,d) -> Printf.printf "ABR(" ; affiche_tree g ; Printf.printf ", _ "; affiche_tree d ;
     Printf.printf ", )"


(* *)
let rec insert (x, e) a =
  match a with
  | Nil -> ABR(Nil , (x, e) , Nil)
  | ABR(_, (y, e1) , _) when e1 = e -> a
  | ABR(g, (y, e1), d) -> if x < y then ABR (insert (x, e) g, (y, e1), d)
            else ABR (g, (y, e1), insert (x, e) d)
(* *)
let rec min a =
  match a with
  | Nil -> failwith "pas de min"
  | ABR(Nil, x , _) -> x
  | ABR(g, _, _) -> min g

```

```

(* *)
let rec find_left_edge a x =
  match a with
  |Nil -> failwith "arbre vide"
  |ABR(g, (y, e1), d) when y <= x ->
      begin
        match d with
        |ABR(dg, (c, e2), dd) when c <= x ->
              find_left_edge d c
        |_ -> e1
      end
  |ABR(g, _, _) -> find_left_edge g x

(* *)
let x_intersection ( ((a, b) , (c, d), _) : edge) y =
  if b = d then
    if c > a then float_of_int c else float_of_int a
  else
    (float_of_int y -. float_of_int b) /. (float_of_int d -. float_of_int b) *. (float_of_int c
    ↪  -. float_of_int a) +. float_of_int a

(* *)
let rec set_y_position a y =
  match a with
  |Nil -> Nil
  |ABR(g, (_, e), d) ->
    ABR(set_y_position g y, (x_intersection e y, e), set_y_position d y)


(* *)
let rec suppression e a =
  match a with
  |Nil -> Nil
  |ABR (g, (y, e1) , d) when e = e1 ->
      begin
```

```
          match g, d with
          | Nil, Nil -> Nil
          | t, Nil | Nil, t -> t
          | _, _ -> let s = min d in ABR (g, s, suppression (snd s) d)
            end
      |ABR (g, (y, e1), d) -> (* if x<y then ABR (suppression (x, e) g, (y, e1), d)
                 else ABR(g, (y, e1), suppression (x, e) d) *)
                 ABR (suppression e g, (y, e1), suppression e d)
```

random_poly.py

RANDOM_POLY.PY (On met les données dans val.ml python3 random_poly.py > val.ml)

```python
from polygenerator import (
    random_polygon,
    random_star_shaped_polygon,
    random_convex_polygon,
)
#source : https://pypi.org/project/polygenerator/

#Renvoie un polygone simple aléatoire avec des coordonnées entières et mise à l'échelle
def rpoly (v, scale) :
    polygon = random_polygon(num_points=v)
    for i in range(v):
        polygon[i] = int(polygon[i][0] * scale), int(polygon[i][1] * scale)
    return polygon

def print_ocaml_lst (v, scale) :
    polygon = rpoly(v, scale)
    print("[", end = " ")
    for i in range(v):
        print("(", polygon[i][0], ",", polygon[i][1], ")", end = " ")
        if i != v-1 :
            print(";", end = " ")
    print("]", end = " ")

def val_graph (scale, size):
    print("let g = [", end = " ")
    for i in range(size):
        print_ocaml_lst(i+3, scale)
        if i != size-1 :
            print(";", end = " ")
    print("]", end = " ")

test = val_graph(1000, 200)
# python3 random_poly.py > val.ml
```

graphe.ml

GRAPHE.ML

```ocaml
open Coloriage
open Val

let time_ear_clipping (lst : polygon) =
  let t1 = Sys.time() in
  try
    let _ = ear_clipping lst in
    let t2 = Sys.time() in
    Printf.printf "%d  %f\n" (List.length lst) (t2 -. t1)
  with
  (* Erreur due à la génération aléatoire polygone simple *)
  |Failure "pas d'oreilles" -> ()

let affiche_ear (l : polygon list) =
  Printf.printf "n temps\n" ;
  List.iter time_ear_clipping l

let nb_cameras lst =
  try
    let n = List.length lst in
    let c = count_colour lst in
    Printf.printf "%d  %d\n" n (min_color c)
  with
  (* Erreur due à la génération aléatoire polygone simple *)
  |Failure "pas d'oreilles" -> ()

let affiche_camera l =
  Printf.printf "n camera borne\n" ;
  List.iter nb_cameras l
```