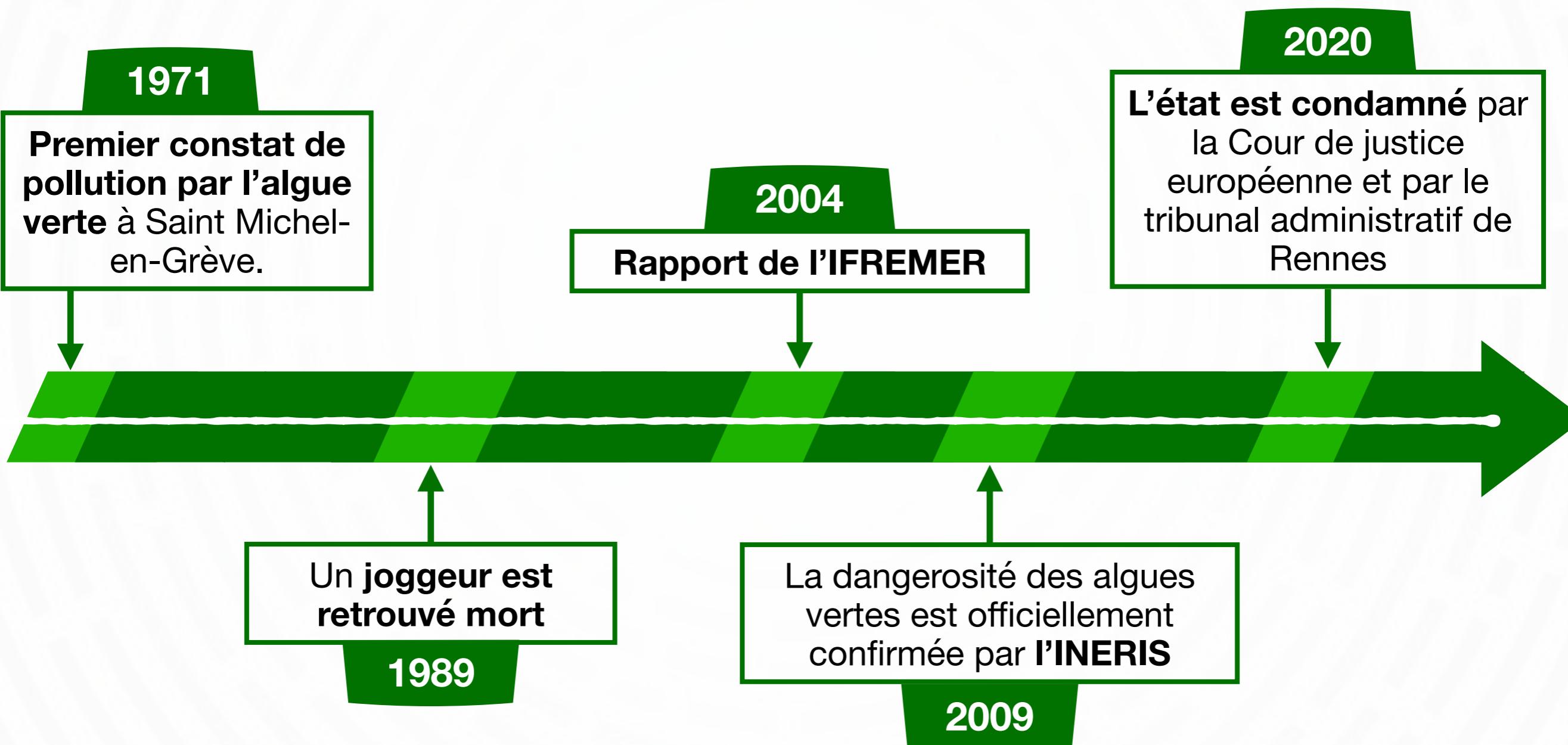

Limitation de la pollution agricole par un positionnement optimal des stocks d'engrais

Algues vertes en Bretagne, un problème de santé publique

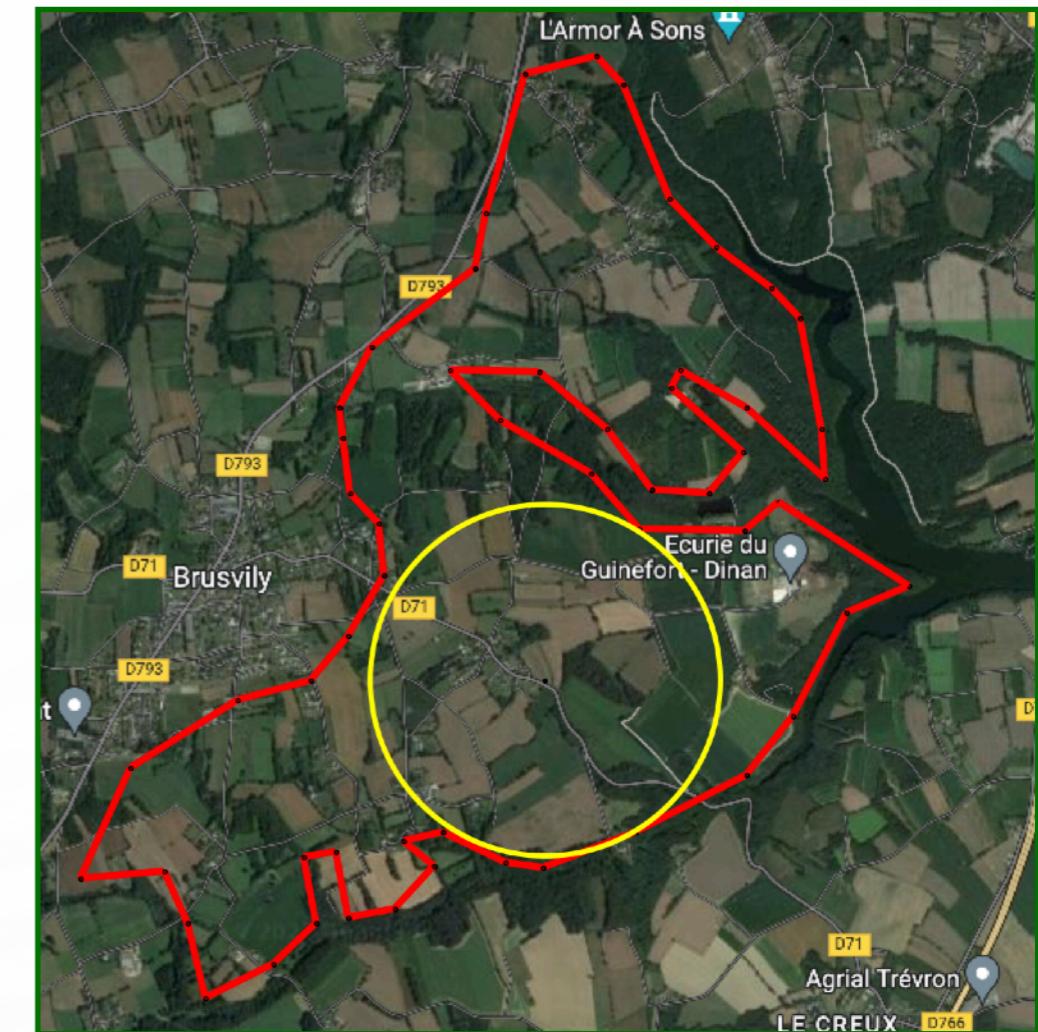
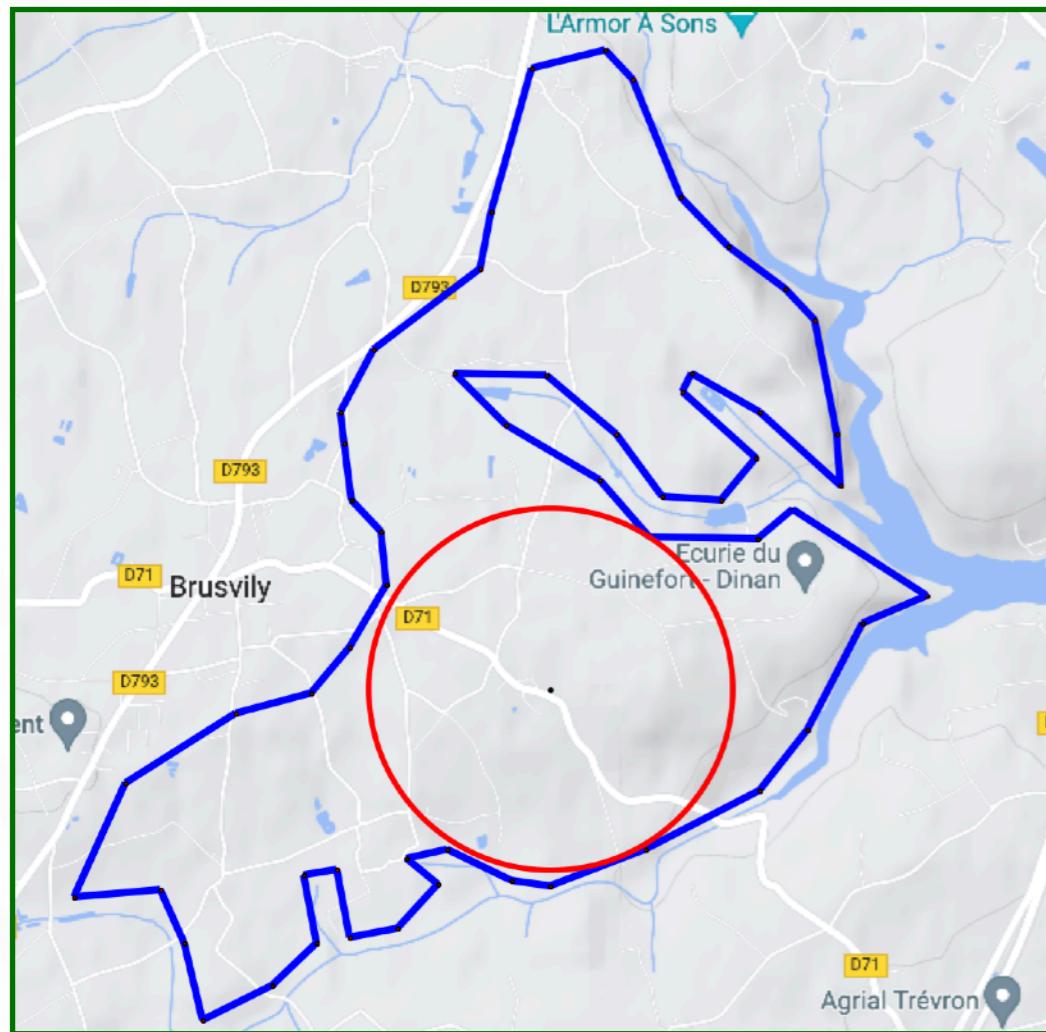


Algues vertes en Bretagne, un problème de santé publique



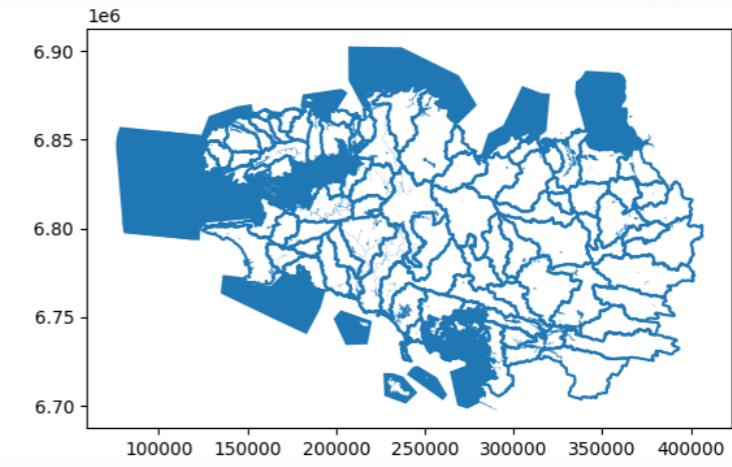
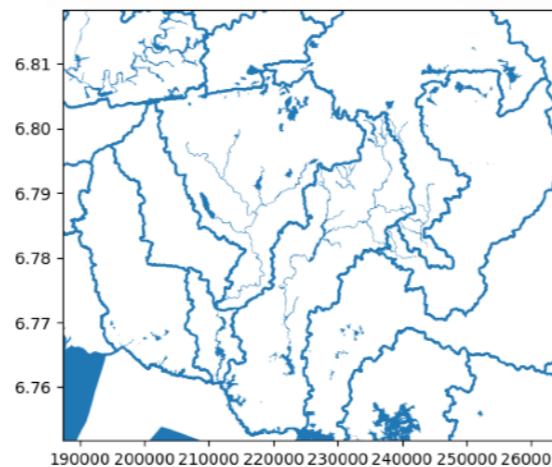
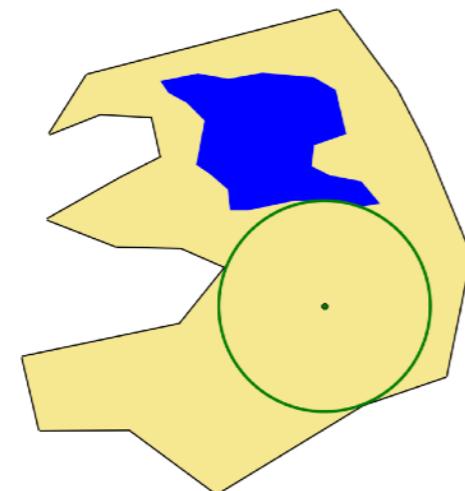
Photo GoogleMaps

Problématique & formalisation



0

Modélisation du problème



1

Analogie avec les pôles d'inaccessibilité

2

Généralisation à plusieurs silos

3

Méthode des diagrammes de Voronoï

4

Application à des cartes réelles

Méthode des pôles d'inaccessibilité

Définition : Un pôle d'inaccessibilité est un endroit éloigné au maximum d'un ensemble de caractéristiques géographiques.

Exemple : le pôle terrestre d'inaccessibilité est le lieu terrestre le plus éloigné des océans.

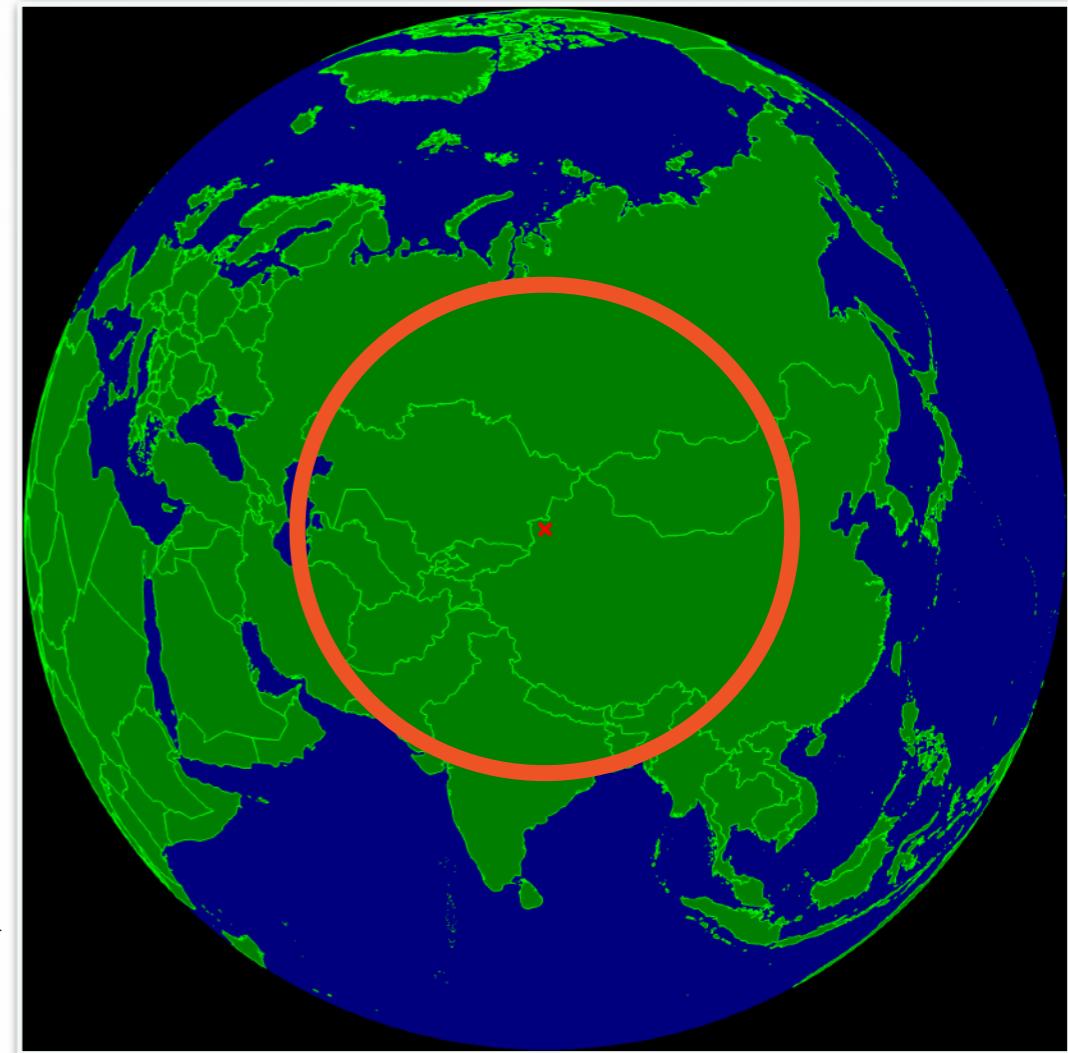
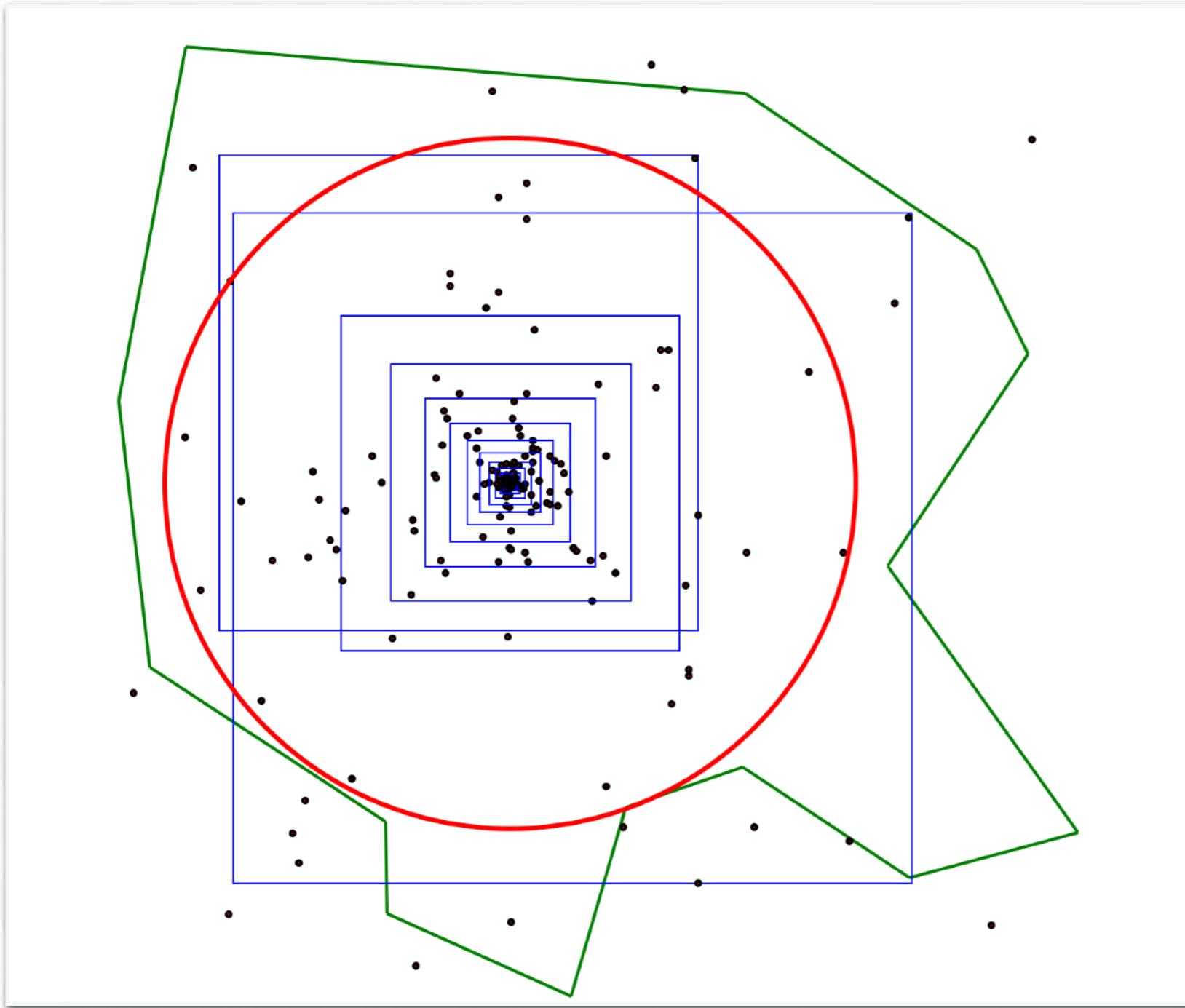


Illustration du pôle d'inaccessibilité terrestre
(Wikipédia)

Approche étudiée : Génération aléatoire de points dans la région et affinage progressif de la zone de recherche du centre autour des candidats optimaux

Méthode des pôles d'inaccessibilité : l'algorithme



Tracé des zones de recherches (carrés bleus) et des candidats générés (points noirs)

CerclePole

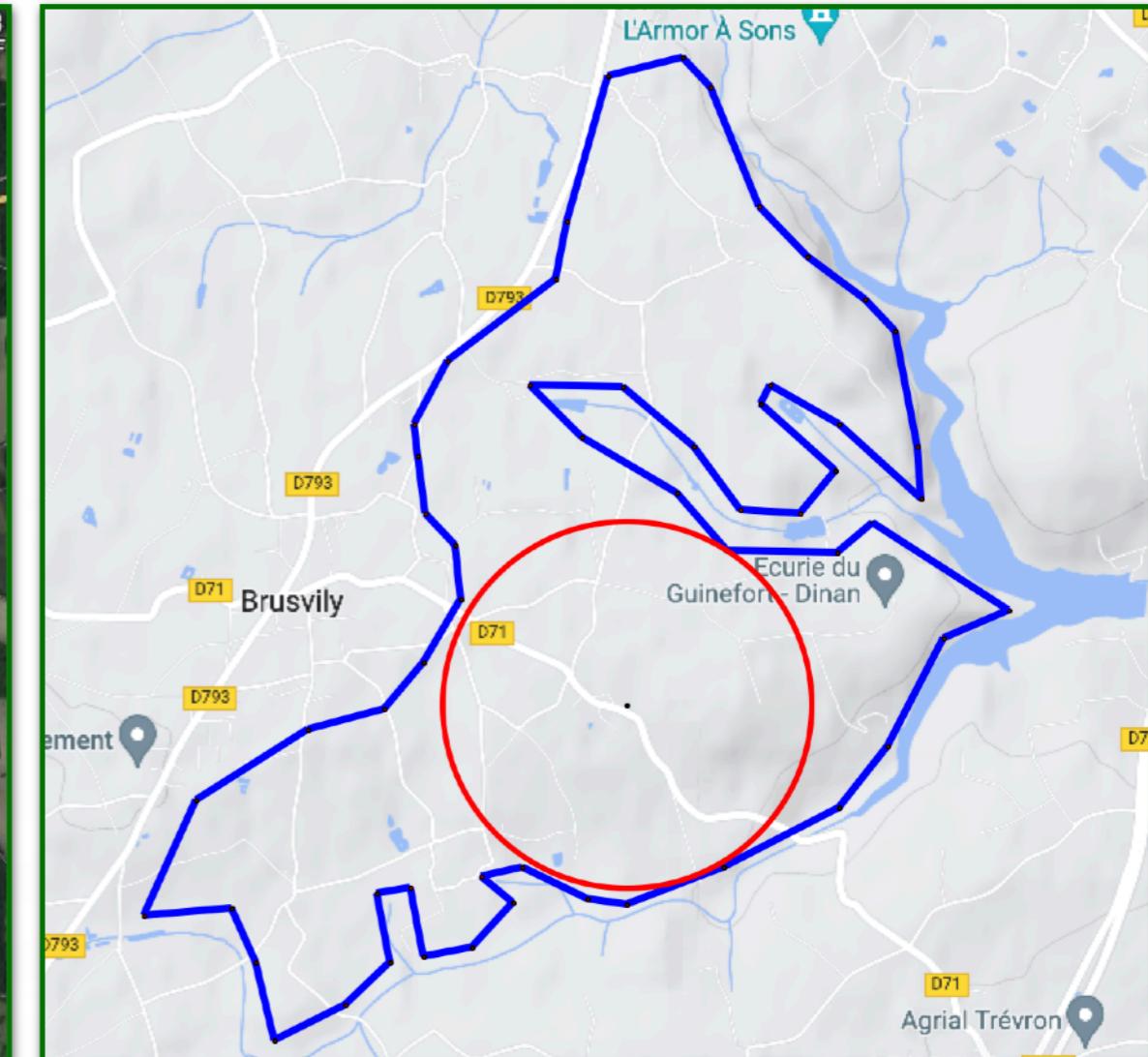
entrée :

- P =Polygone
- k =nombres de points
- e =précision

sortie :

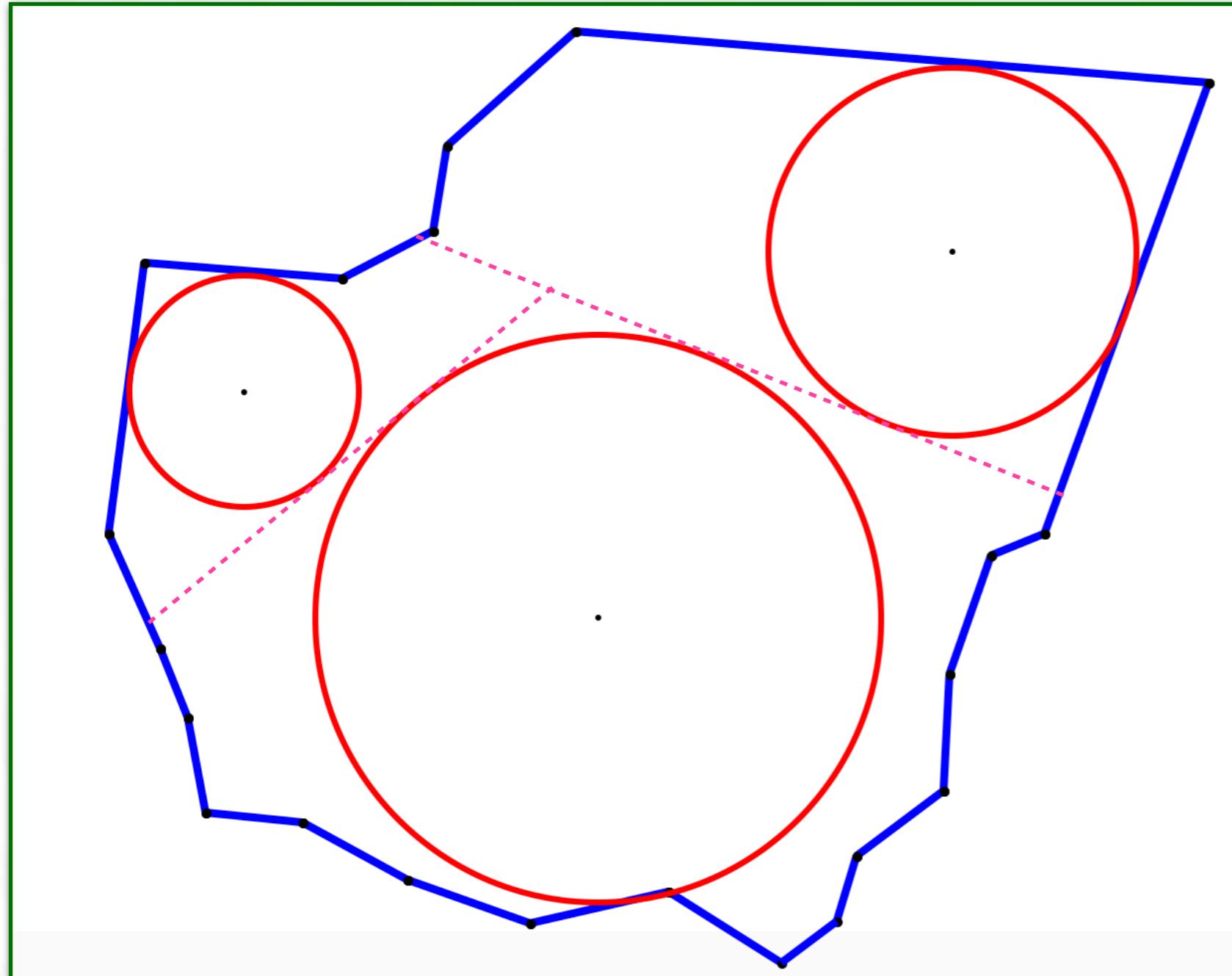
- centre
- rayon

Méthode des pôles d'inaccessibilité : résultats



Application sur des fonds GoogleMaps tracée avec Tkinter

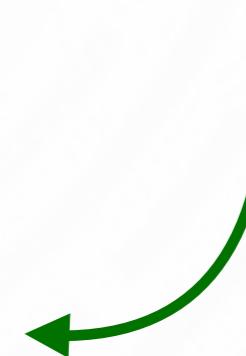
Méthode des pôles d'inaccessibilité : plusieurs cercles



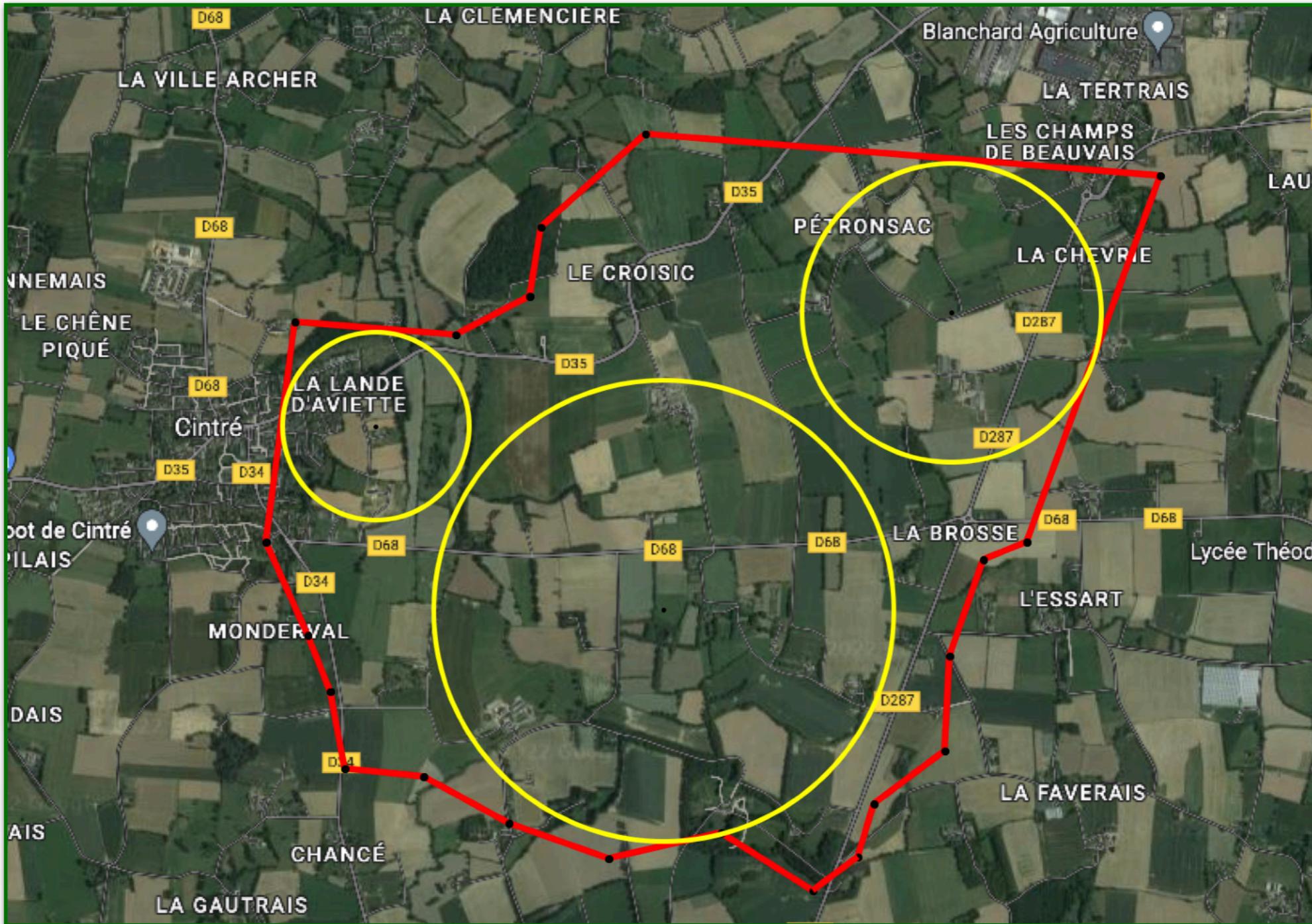
Résultat obtenu pour 3 cercles tracé avec Tkinter

Génération aléatoire de **partitions du polygone**, application de l'algorithme dans les sous-polygone.

Selection du trio de cercle de **somme des rayons maximale**



Méthode des pôles d'inaccessibilité : plusieurs cercles



Application sur un fond GoogleMaps

Méthode des pôles d'inaccessibilité : complexité

Complexité temporelle : $O\left[kn \times \log_2\left(\frac{D}{e}\right)\right]$ où :

- D la distance maximale entre deux points du polygone
 - n le nombre de points du polygone
 - k le nombre de génération de points et e la précision.
-

Commentaires :

- Complexité linéaire en k , bien trop élevée pour des résultats acceptables
- Complexité dépendant de la forme initiale du polygone

Diagrammes de Voronoï

On note P un ensemble fini de points de \mathbb{R}^2

Cellule de Voronoï (germe) : Pour point $p_i \in P$ notée C_i . C'est l'ensemble des points de E plus proches de p_i que de tout autre point de P :

$$C_i = \{q \in E, \forall (j, i), \|\overrightarrow{qp_i}\| \leq \|\overrightarrow{qp_j}\| \}$$

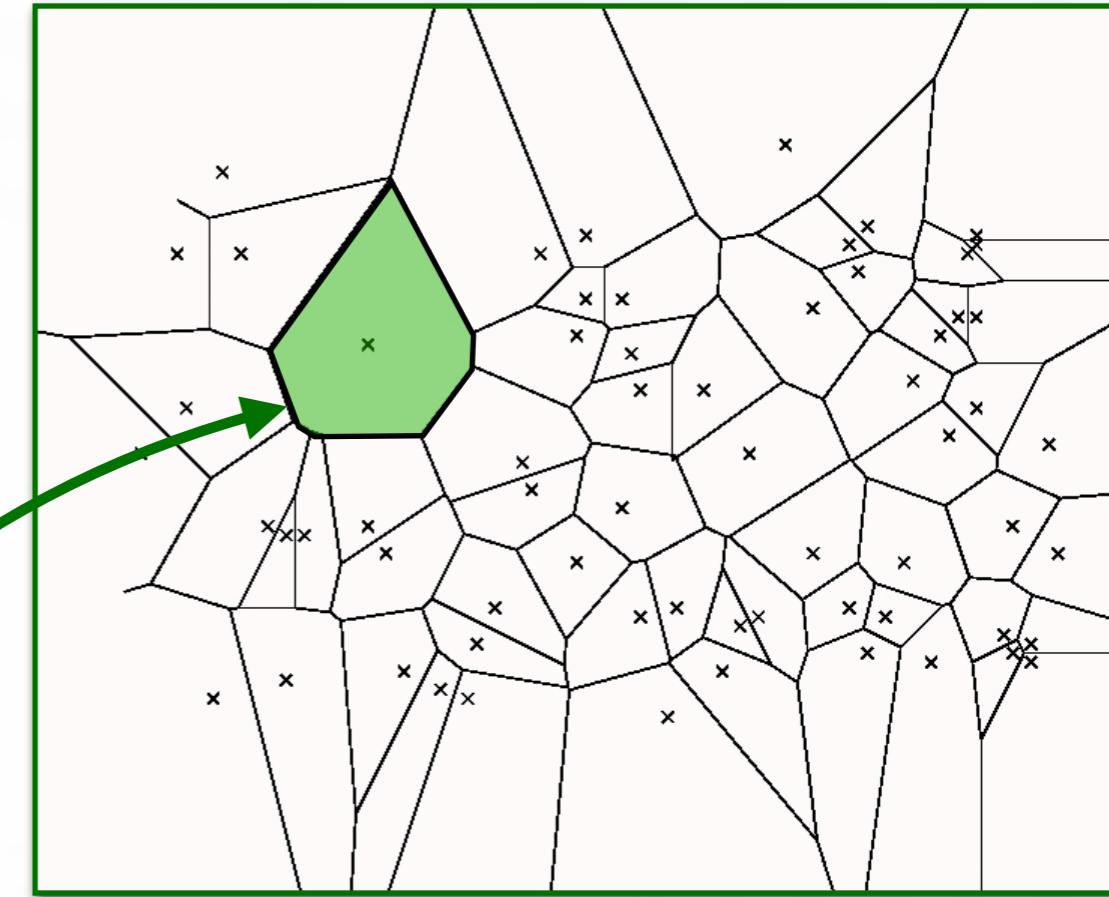


Diagramme de Voronoï : $Vor(P) = \bigcup_{p_i \in P} C_i$

- Une arête de Voronoï, séparant deux cellules C_i et C_j , est portée par la **médiatrice du segment** p_ip_j .
- Le sommet de Voronoï séparant trois cellules C_i , C_j , C_k est le centre du **cercle circonscrit** au triangle de sommet p_i , p_j et p_k .

Approximation de l'axe médian

Axe médian : L'axe médian d'un polygone P est le **lieu des centres des cercles qui sont tangents à P en deux points ou plus**, et qui sont contenus dans P

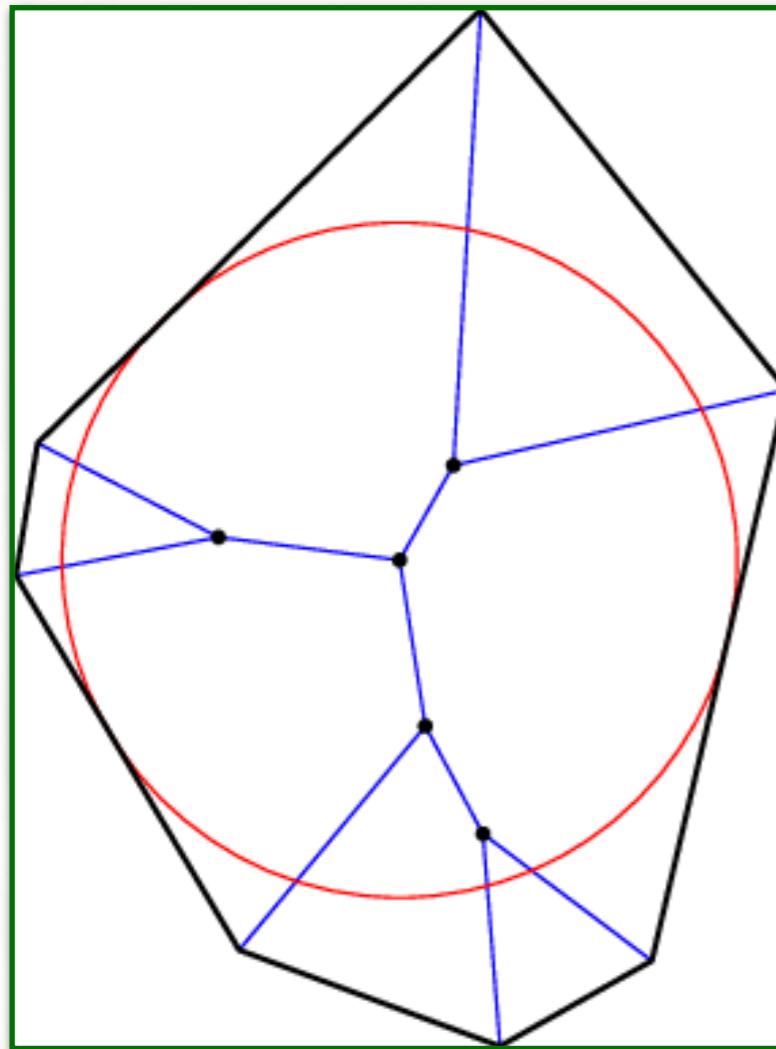
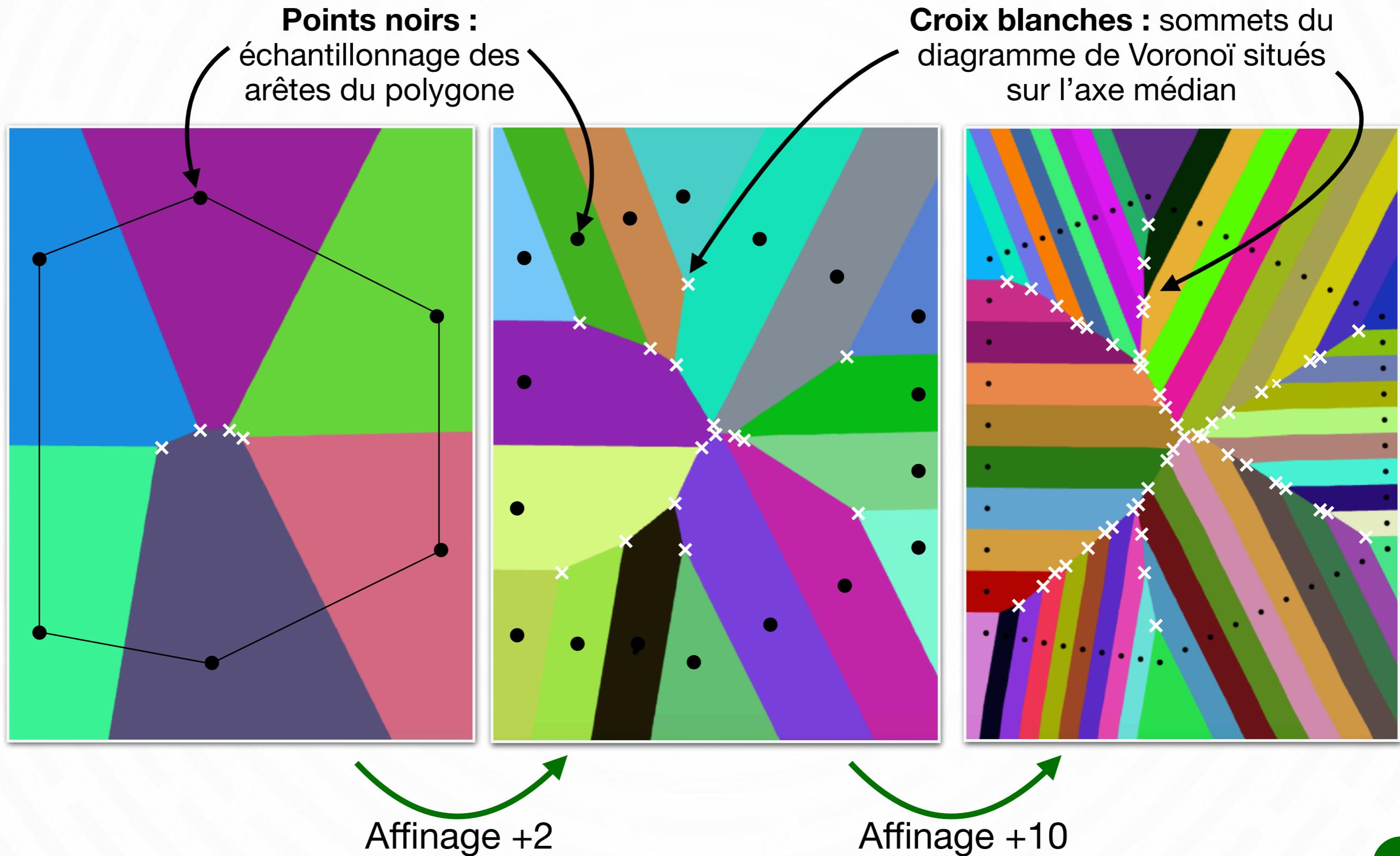


Illustration de l'axe médian d'un polygone quelconque

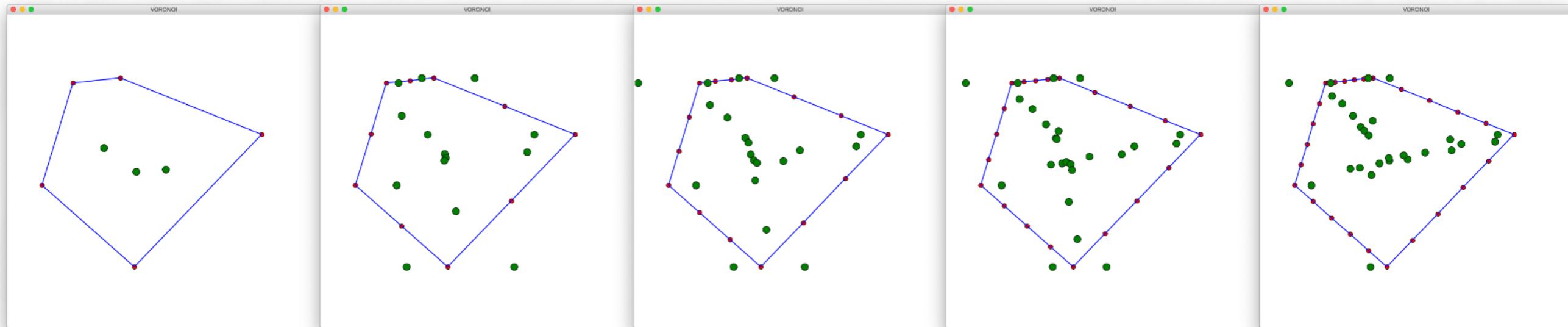
L'axe médian est représenté comme un sous-ensemble de la frontière des cellules de Voronoï

On réalise ici une **approximation discrète de l'axe médian en échantillonnant le polygone**

Approximation de l'axe médian



Approximation de l'axe médian



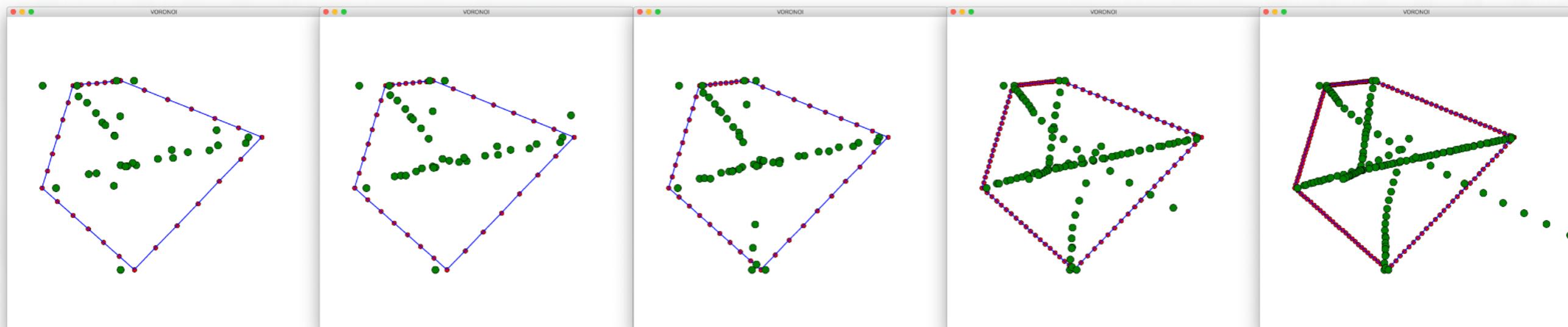
e=0

e=1

e=2

e=3

e=4



e=5

e=6

e=10

e=20

e=30

Tracé de l'axe médian par affinage progressif du polygone avec Tkinter

Diagrammes de Voronoï : construction

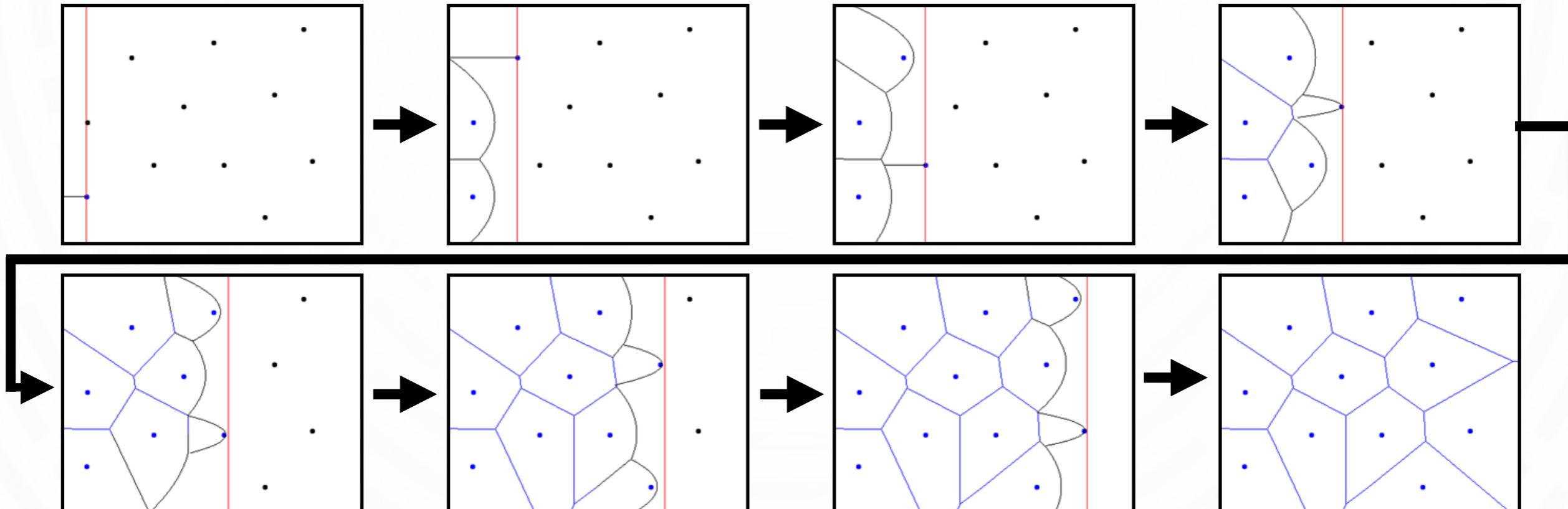
Panel des méthodes de construction

Construction « incrémentale » : $O(n^2)$

Construction « diviser pour régner » : $O(n \log n)$ (compliqué à implémenter)

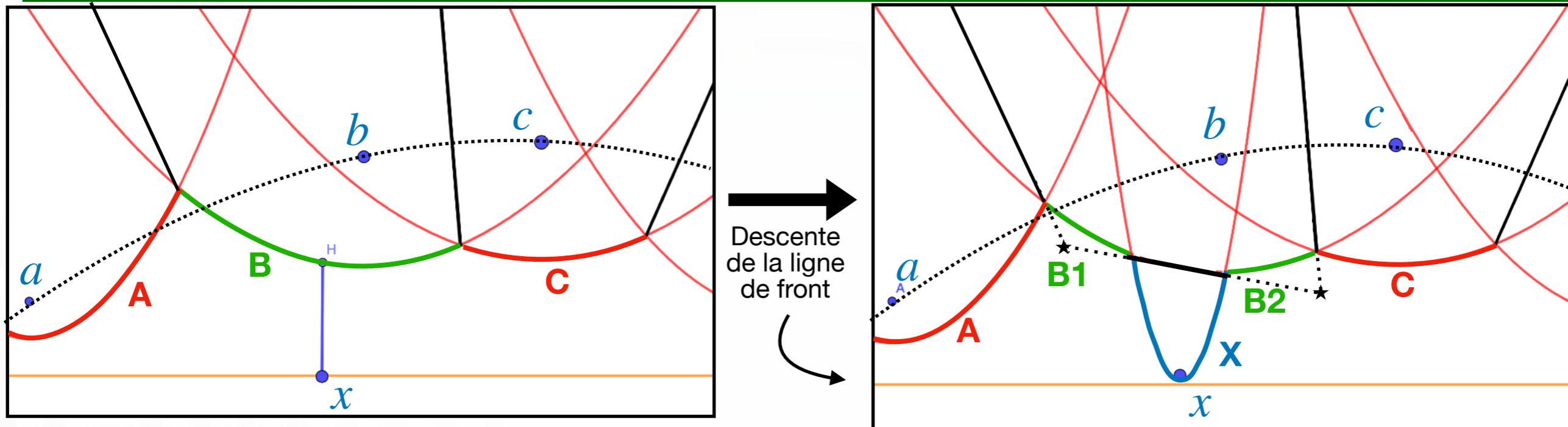
Construction « par balayage » (*Fortune*) : $O(n \log n)$

Choix de l'algorithme de Fortune : balayage du plan

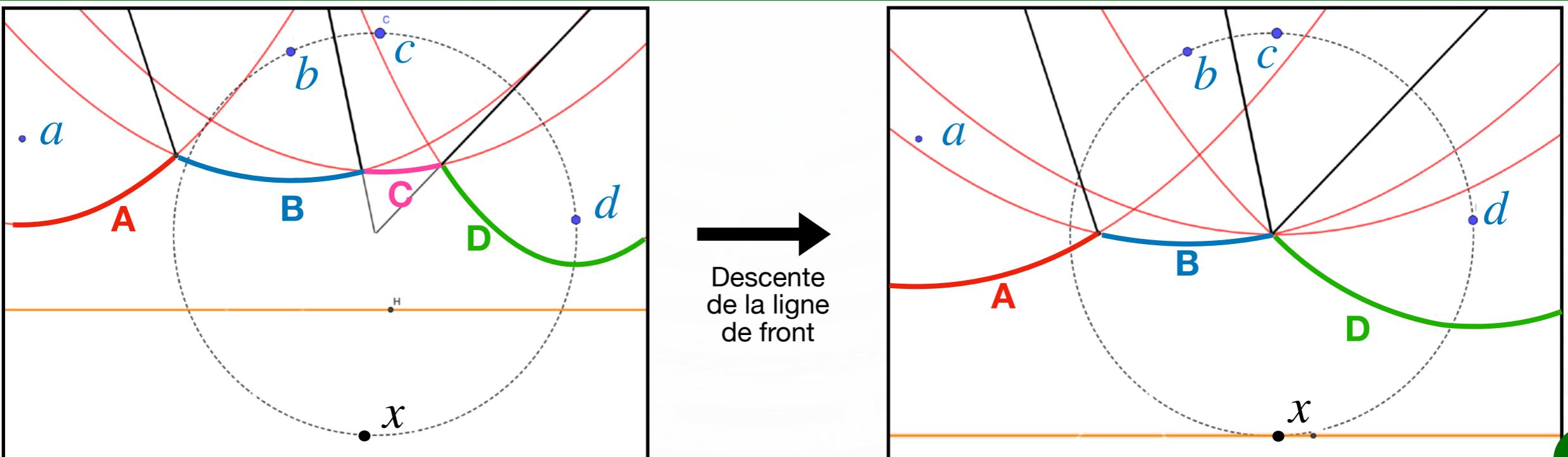


Algorithme de Fortune : événements

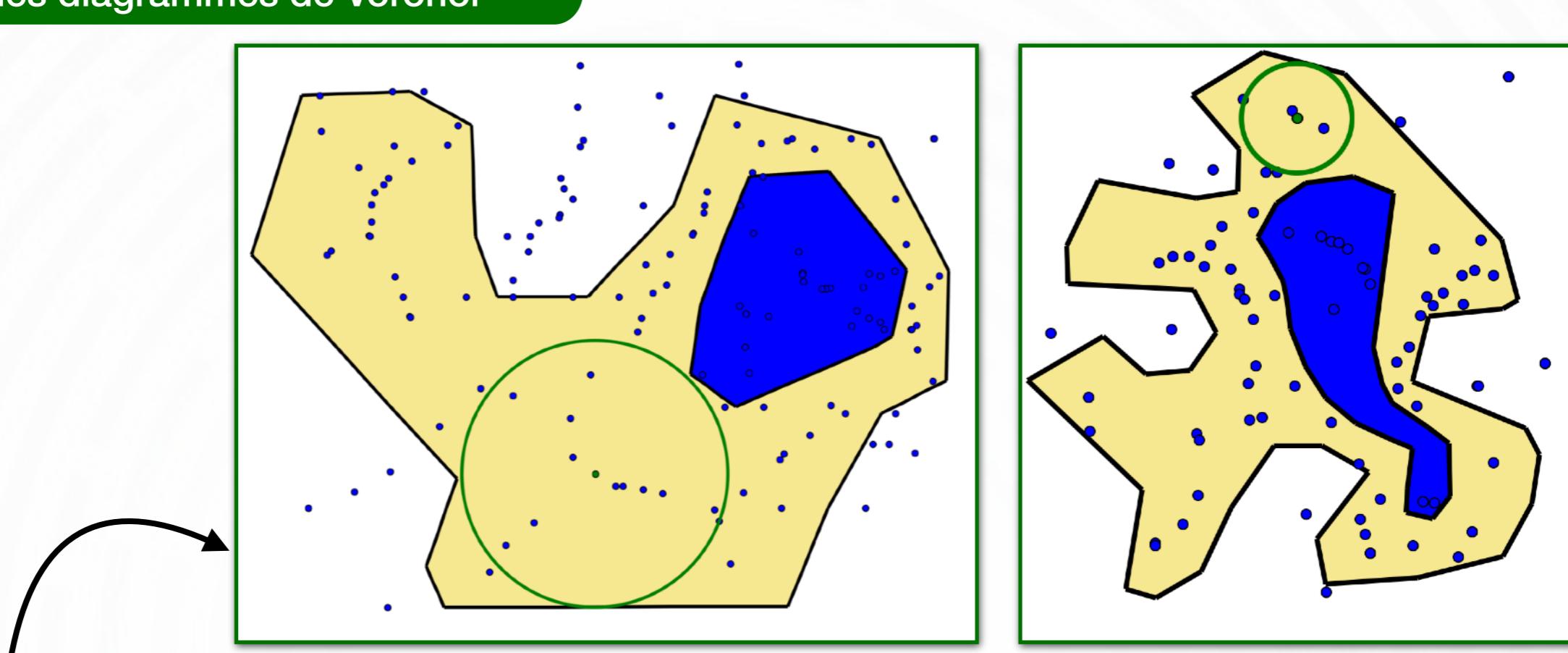
Traitements des « site events »



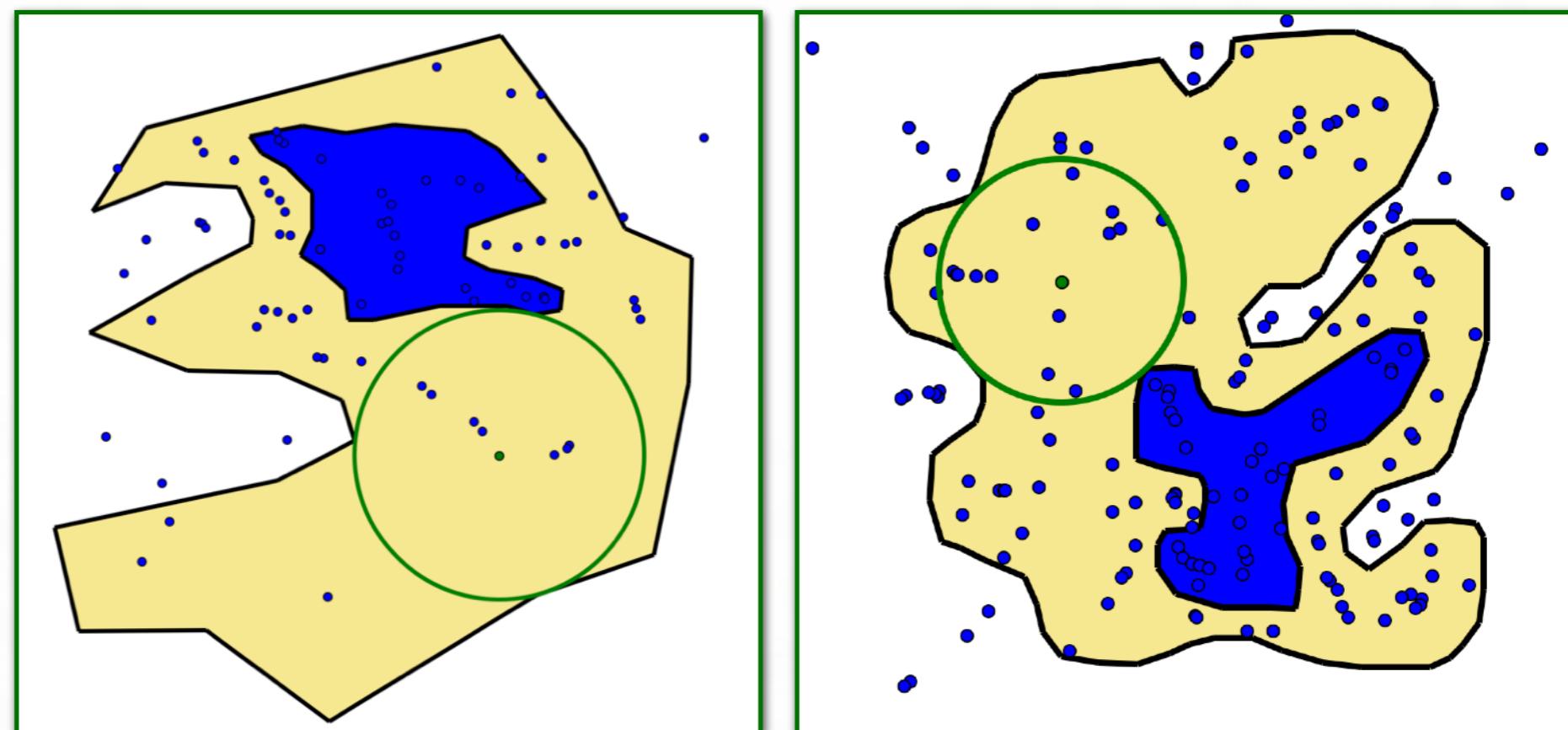
Traitements des « circle events »



3 Méthode des diagrammes de Voronoi

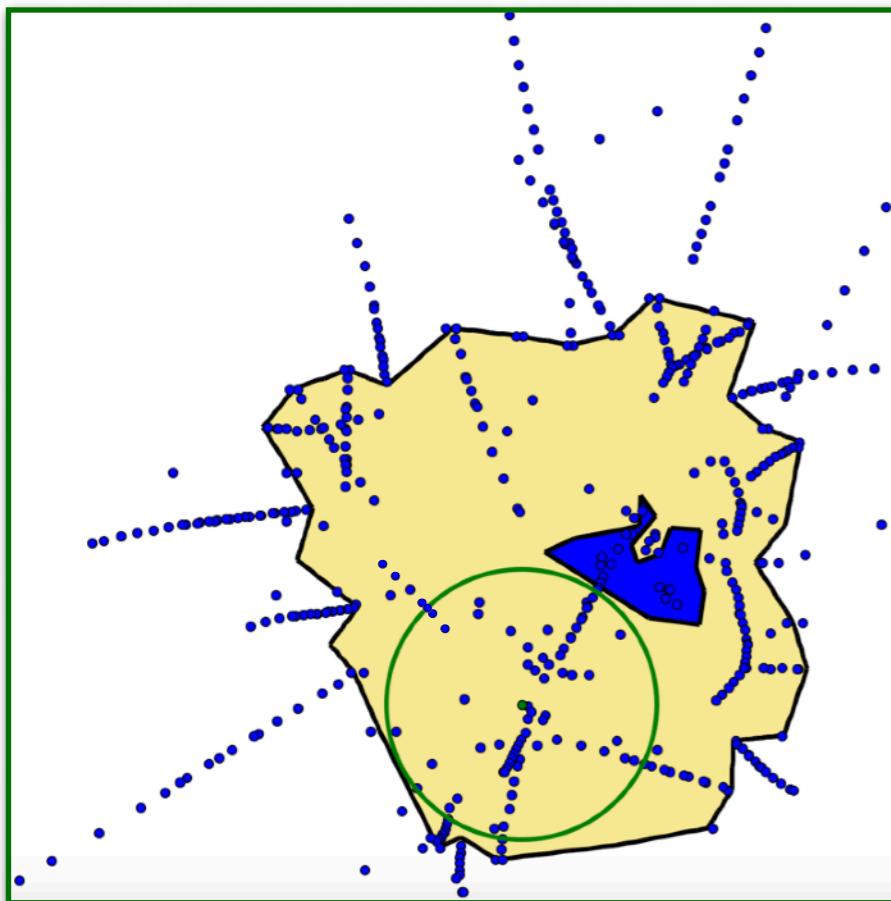


Points bleu désignant les sommets du diagramme de Voronoï calculé



Résultats obtenus pour des polygones avec intérieur tracés avec Tkinter

3) Méthode des diagrammes de Voronoi



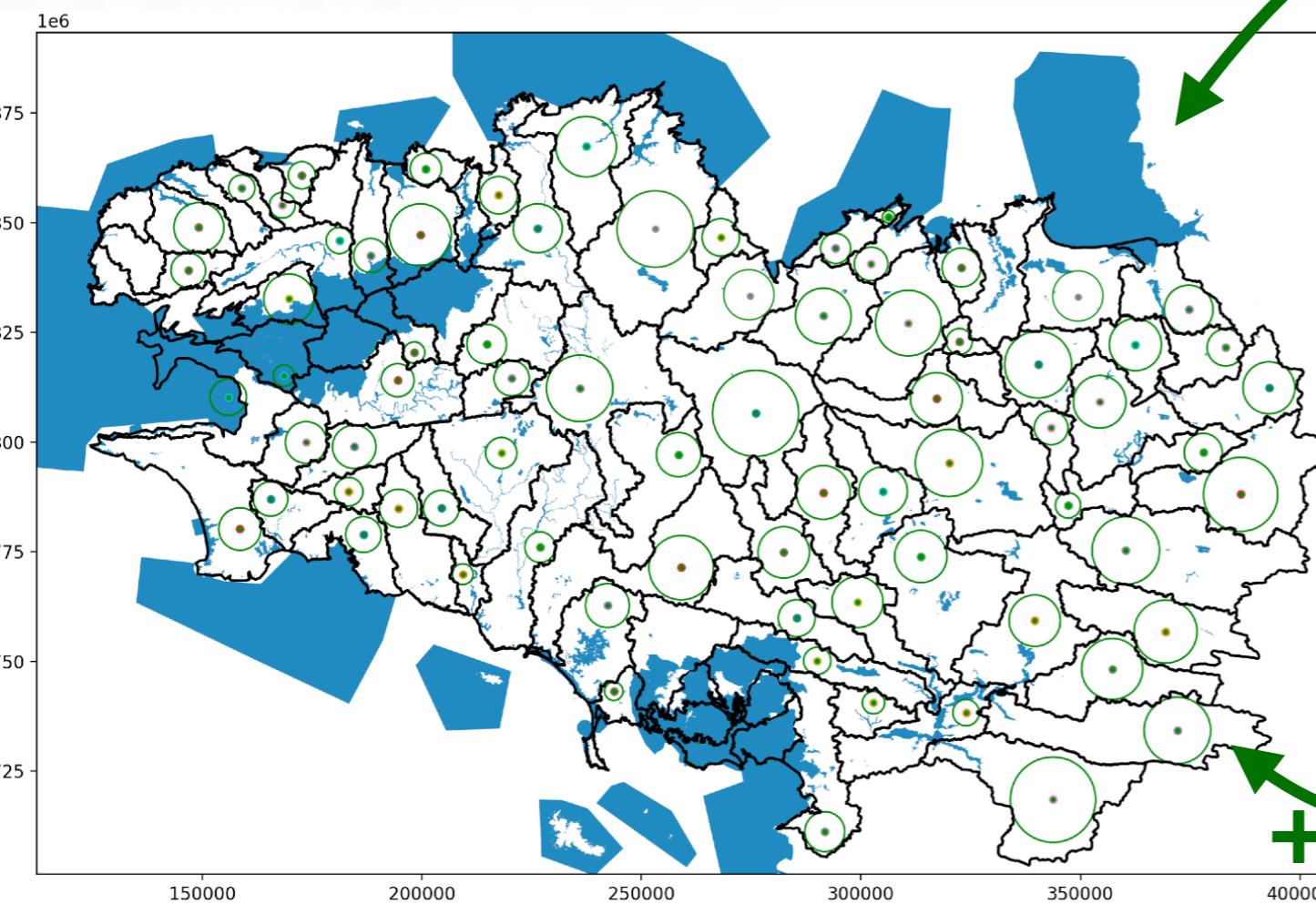
Résultat tracé avec Tkinter



Résultat tracé sur un fond GoogleMaps avec Tkinter

Utilisation de cartes publiques

Utilisation des cartes **geobretagne.fr**, catalogue de cartes de données géographiques, topographique, cadastrales, administratives, hydrographiques, cours d'eau, données démographiques...



Cartes des bassins versants de Bretagne superposée aux sites de préservation du patrimoine naturel en Bretagne tracée avec Matplotlib

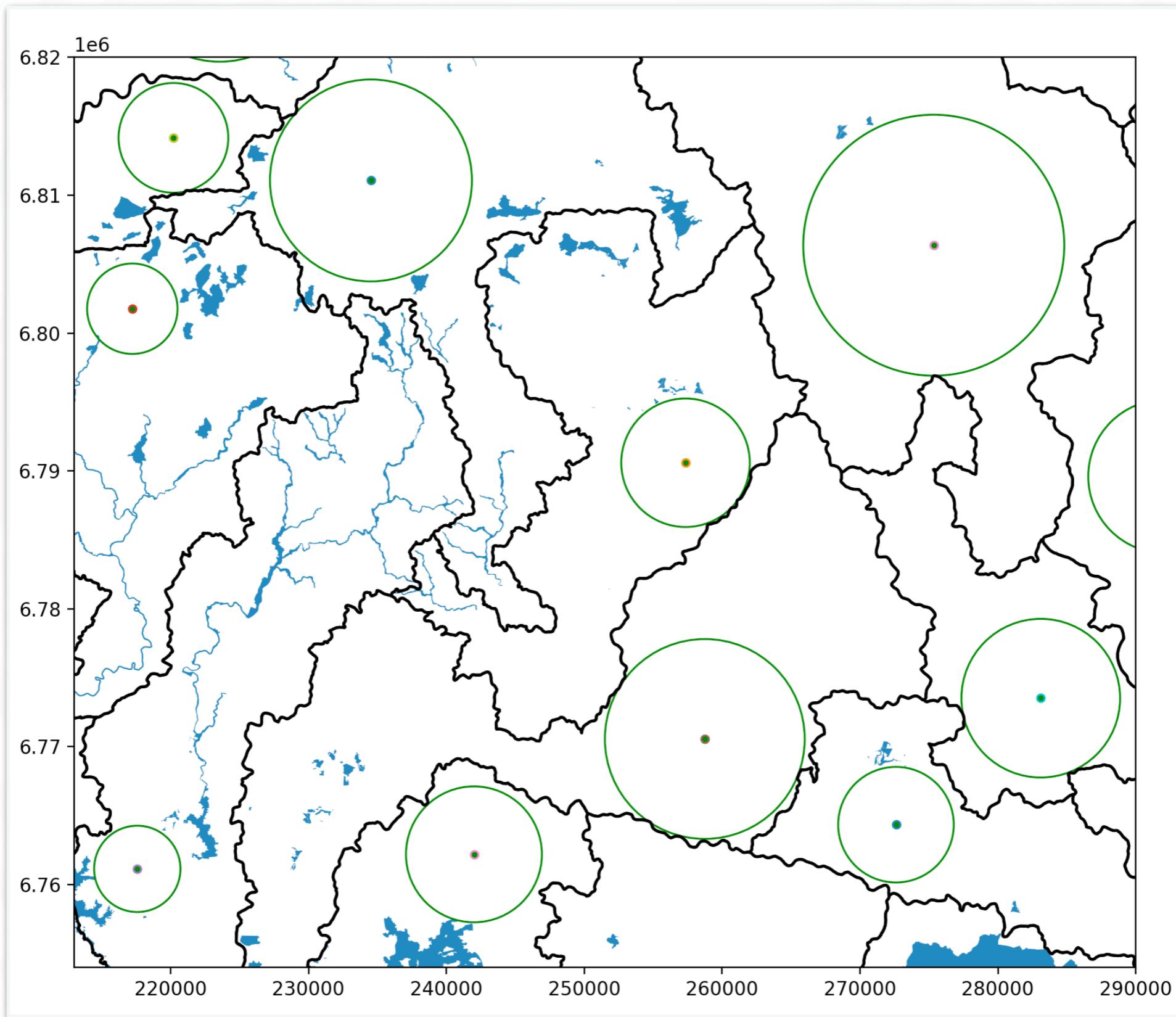


Bassins versants de Bretagne



Sites de préservation du patrimoine naturel

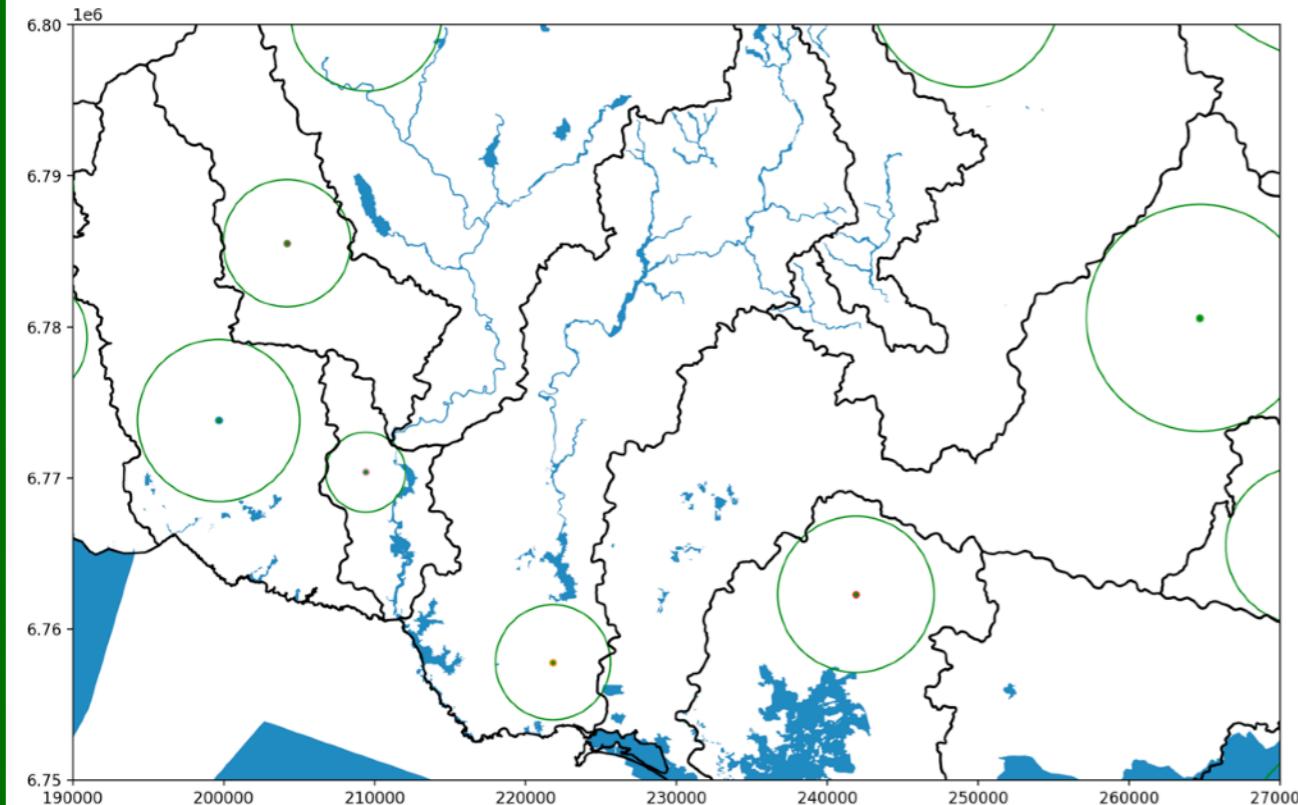
Utilisation de cartes publiques



Comparaison des algorithmes

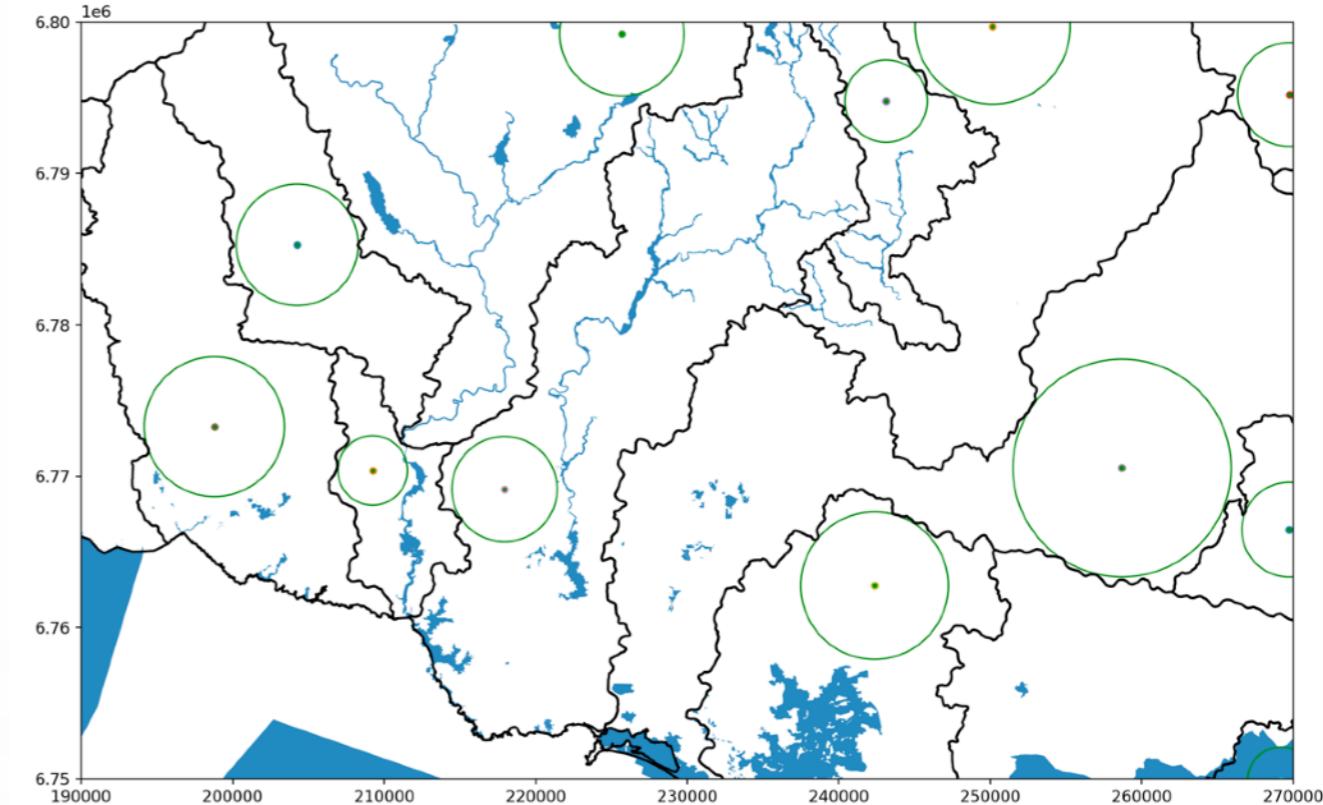
Comparaison des temps d'exécution entre les deux méthodes pour un nombre important de points

Méthode pôles d'inaccessibilité



46 secondes

Méthode axe médian

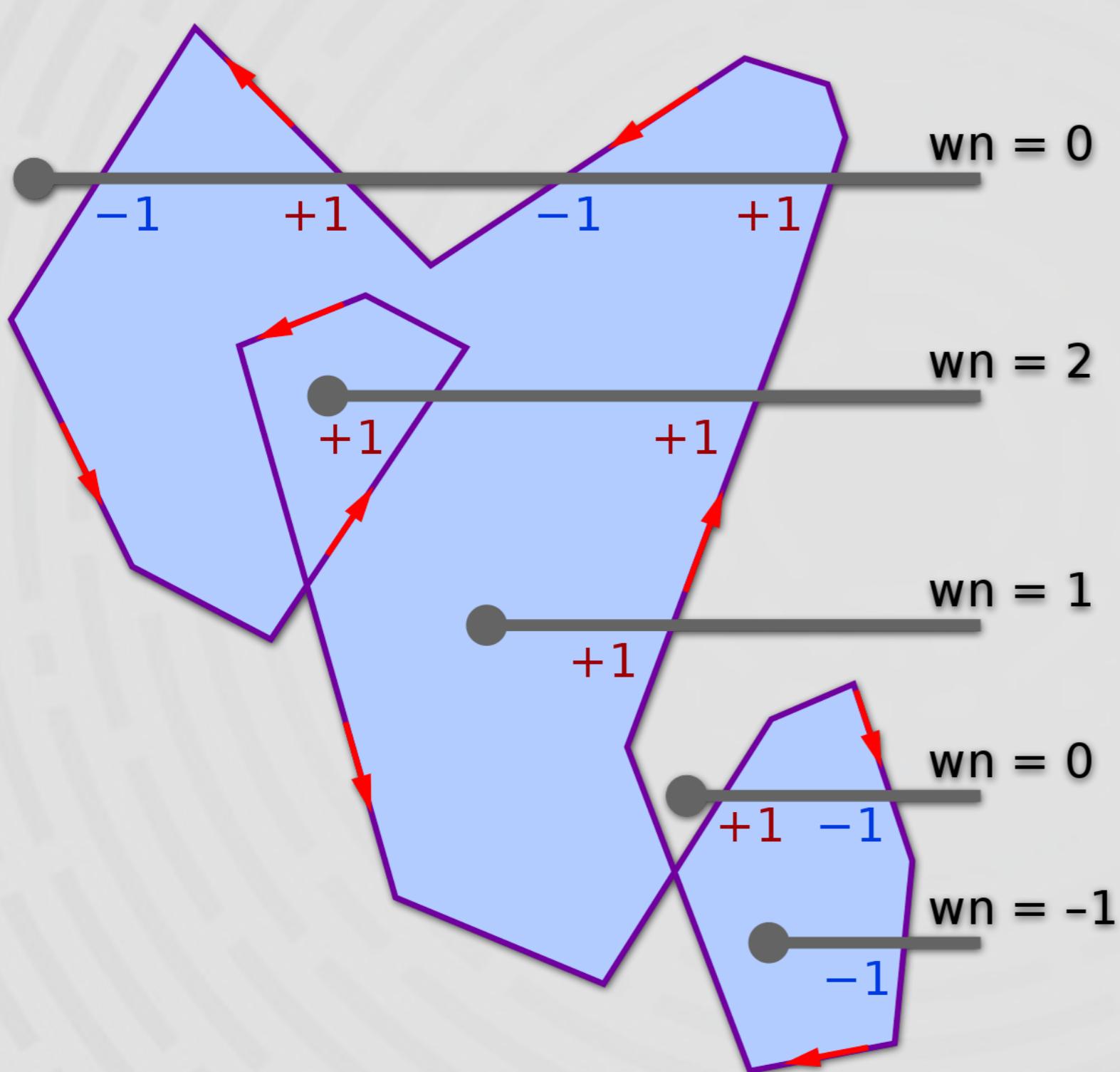


1.7 secondes

1823 points au total traités sur cet exemple

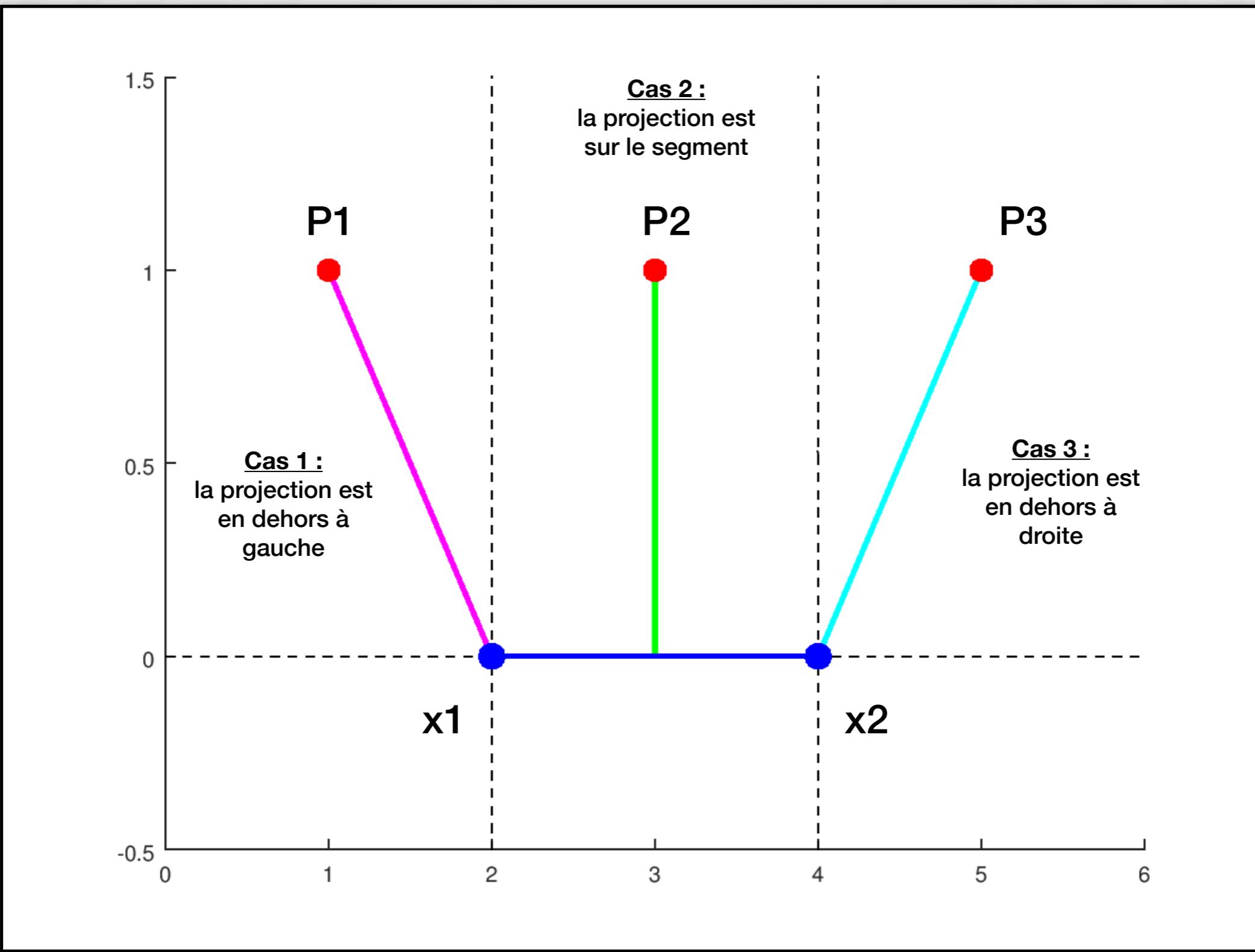
Merci pour votre attention

Le ray-casting

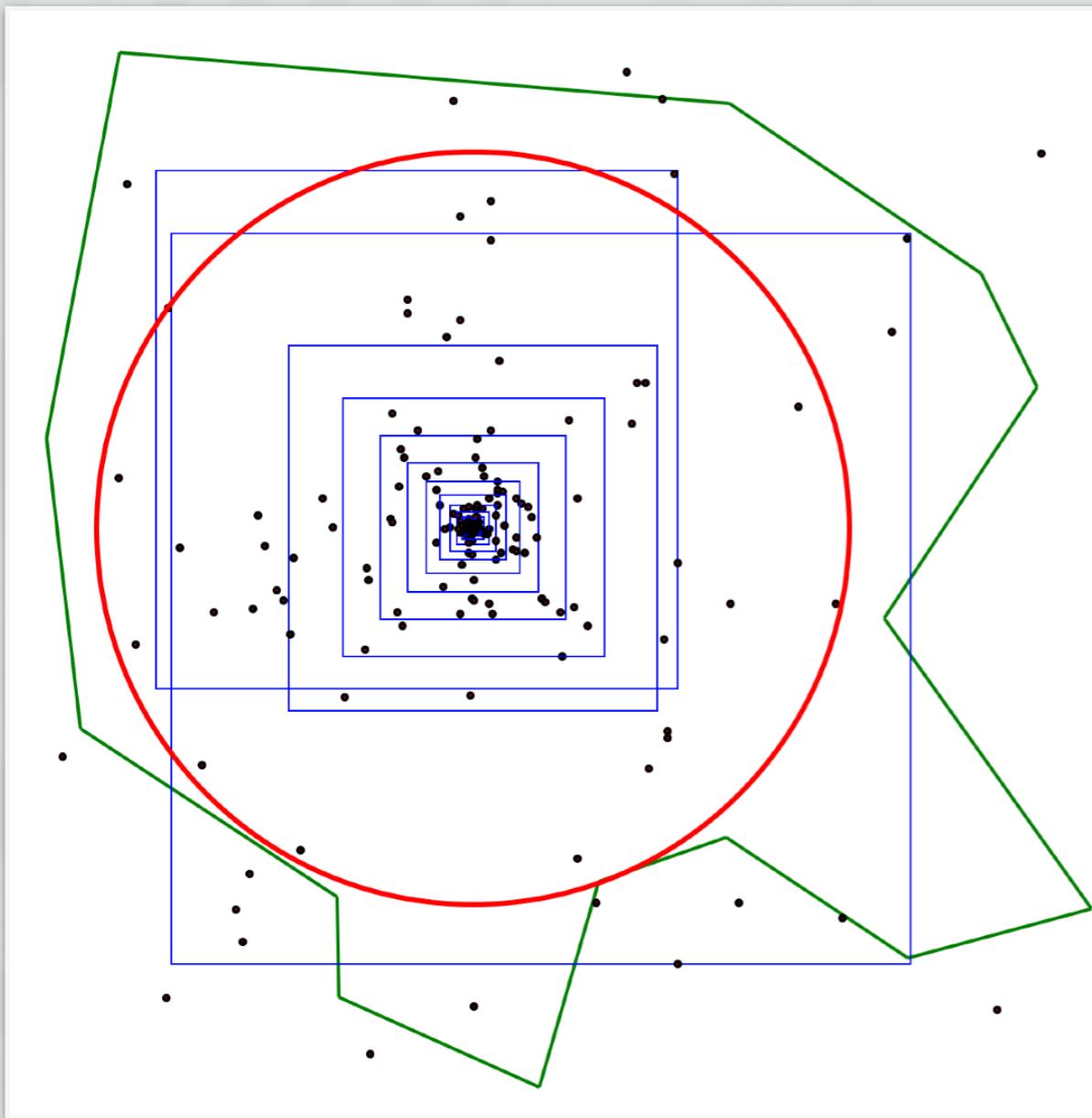


L'indice d'un point par rapport à un lacet est intuitivement le nombre de tours (dans le sens contraire des aiguilles d'une montre) réalisé par le lacet autour du point.

Le calcul de la distance aux arêtes



Méthode des pôles d'inaccessibilité : l'algorithme



CerclePole

entrée : (Polygone, k=nombres de points, e=précision)

sortie : (centre, rayon)

initialiser : zone=distance max entre deux points du polygone

initialiser : rayon=0

initialiser : centre=None

boucler tant que zone > e :

initialiser : essai = 0

boucler tant que essai < k :

générer un point aléatoirement dans zone

si point est dans Polygone :

dmin = distance minimale du point aux frontières

si dmin > rayon :

dmin = rayon

centre = point

réduire zone autour de centre d'un rapport $\sqrt{2}$

Algorithme de Fortune : algorithme

Algorithme de Fortune (1987)

entrée : un ensemble de sites (points)

sortie : (liste doublement chainée des arêtes, liste des sommets)

initialiser la queue en ajoutant les points selon leur coordonnée y croissante

initialiser le diagramme de Voronoi **V** à vide

initialiser la queue **Q** à vide

initialiser l'arbre binaire **T** vide

boucler tant que la queue n'est pas vide :

 récupérer le premier événement de la queue :

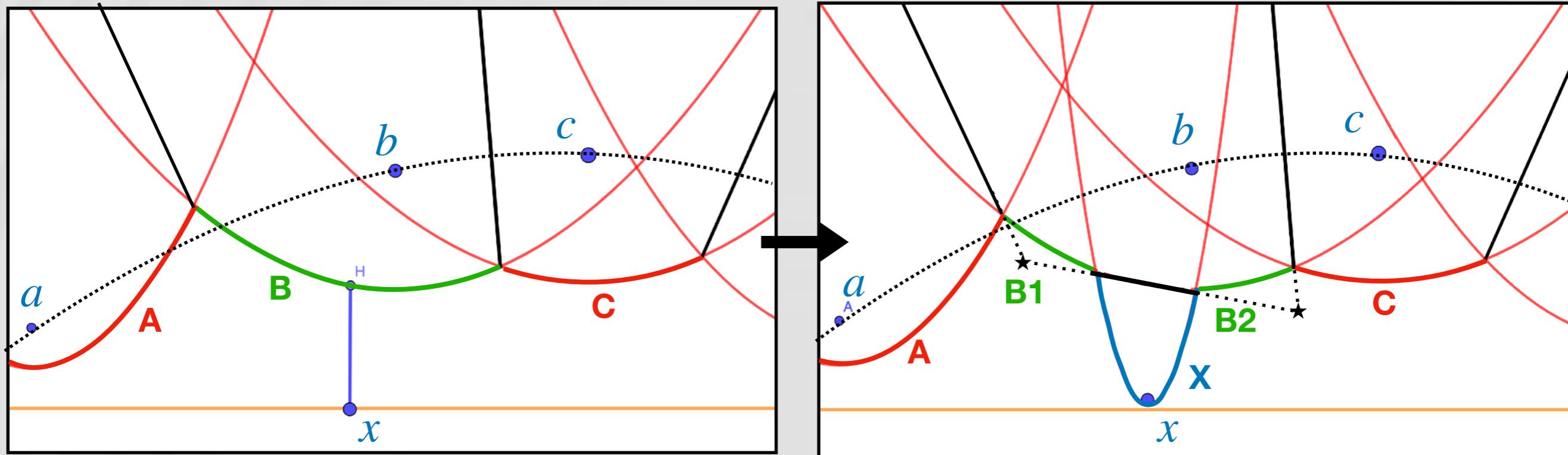
si c'est un SiteEvent : **x**

TraiterSiteEvent(x)

si c'est un CercleEvent : **x**

TraiterCercleEvent(x, c) ou **c** le centre associé

Algorithme de Fortune : site event



TraiterSiteEvent (x) :

si T est vide :

ajouter la parabole de foyer **x** à la racine

sinon :

trouver la parabole directement au dessus (B) et ses voisins (A, C)

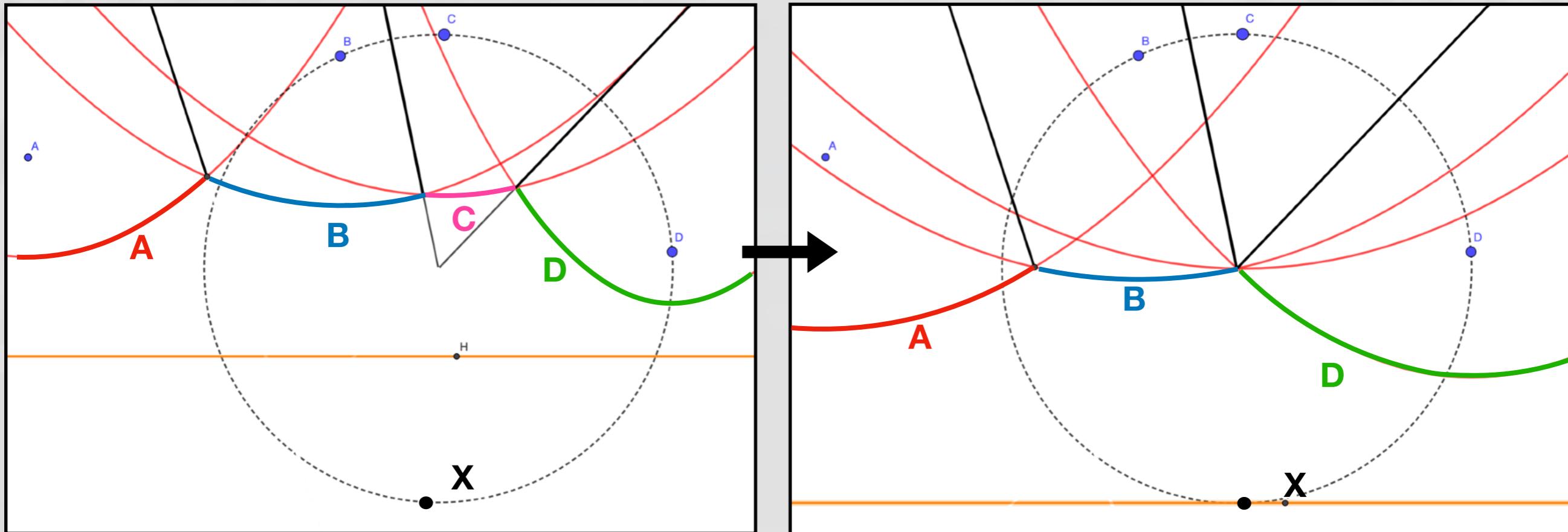
insérer la nouvelle parabole de foyer **x** à **T**

découper la parabole **B**

supprimer les CercleEvent liés à A-B-C de **Q**

insérer les 3 cercles events (A-B1-X, B1-X-B2, X-B2-C) à **Q**

Algorithme de Fortune : cercle event



TraiterCercleEvent (x, c) : $(ABxDE \rightarrow ABDE)$

ajouter le sommet à **V**

clore les segments d'extrémité **c**

supprimer la parabole de **T**

supprimer de **Q** les potentiels CercleEvent liés à x : A-B-x, B-x-D, x-D-E

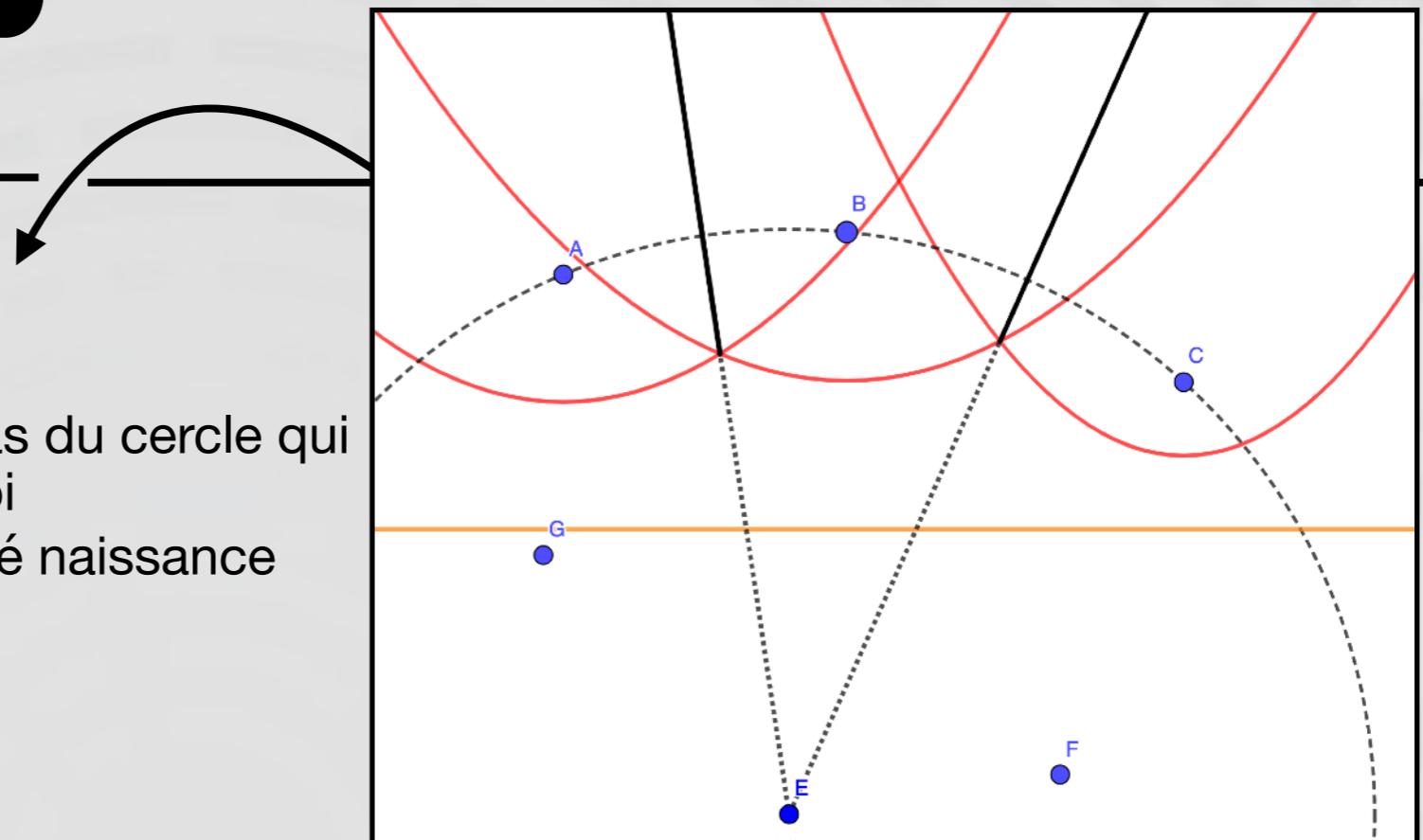
ajouter les CercleEvent : A-B-D, B-D-E

Objets :

- **Events :**

- **CercleEvent :**

- **(x,y)** : coordonnées du point du bas du cercle qui correspond à un sommet de Voronoi
 - **parabole** : parabole qui lui a donné naissance
 - **c** : centre du cercle



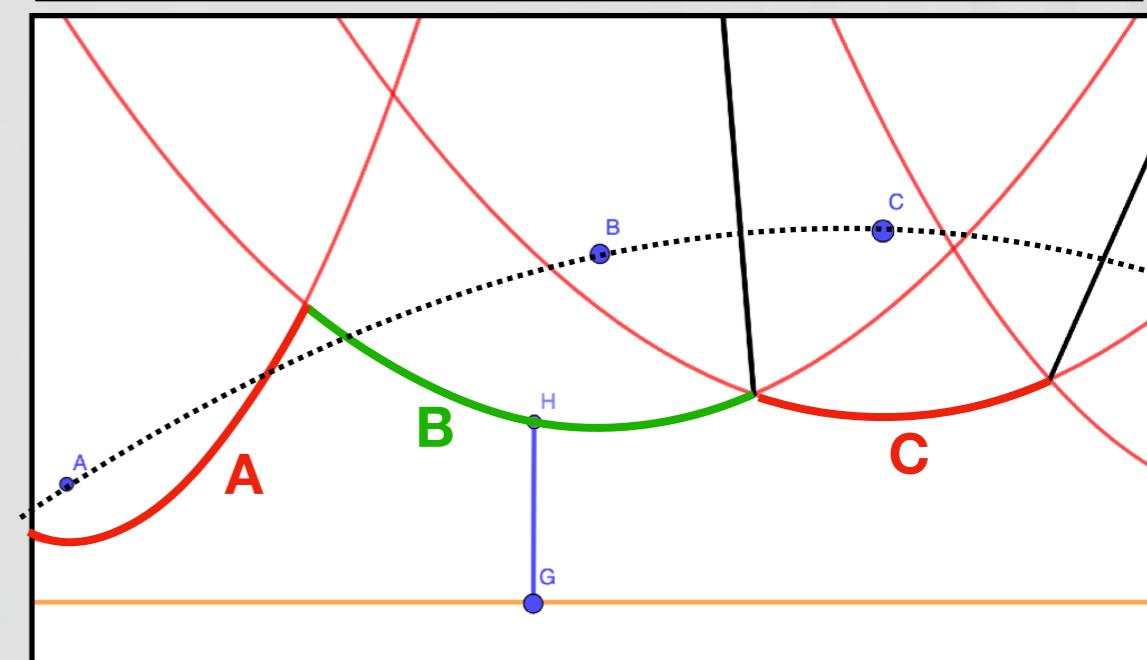
- **SiteEvent :**

- **(x,y)** coordonnées du point



- **Beachline (T) :** arbre binaire référençant l'enchaînement des paraboles. Les feuilles de l'arbre représentent l'avancement de la ligne de front.

- **EventQueue (Q) :** Pile de priorité stockant les event par coordonnées y croissants



Algorithme de Fortune : complexité

Nombre de paraboles dans l'arbre de paraboles (T) : $O(n)$

Insertion ou recherche d'un point dans (T) : $O(\log n)$

Un diagramme de Voronoi de n points du plan est la **projection d'un polyèdre à n faces** quitte à rajouter un sommet à l'infini pour que le polyèdre soit déformable en une sphère.

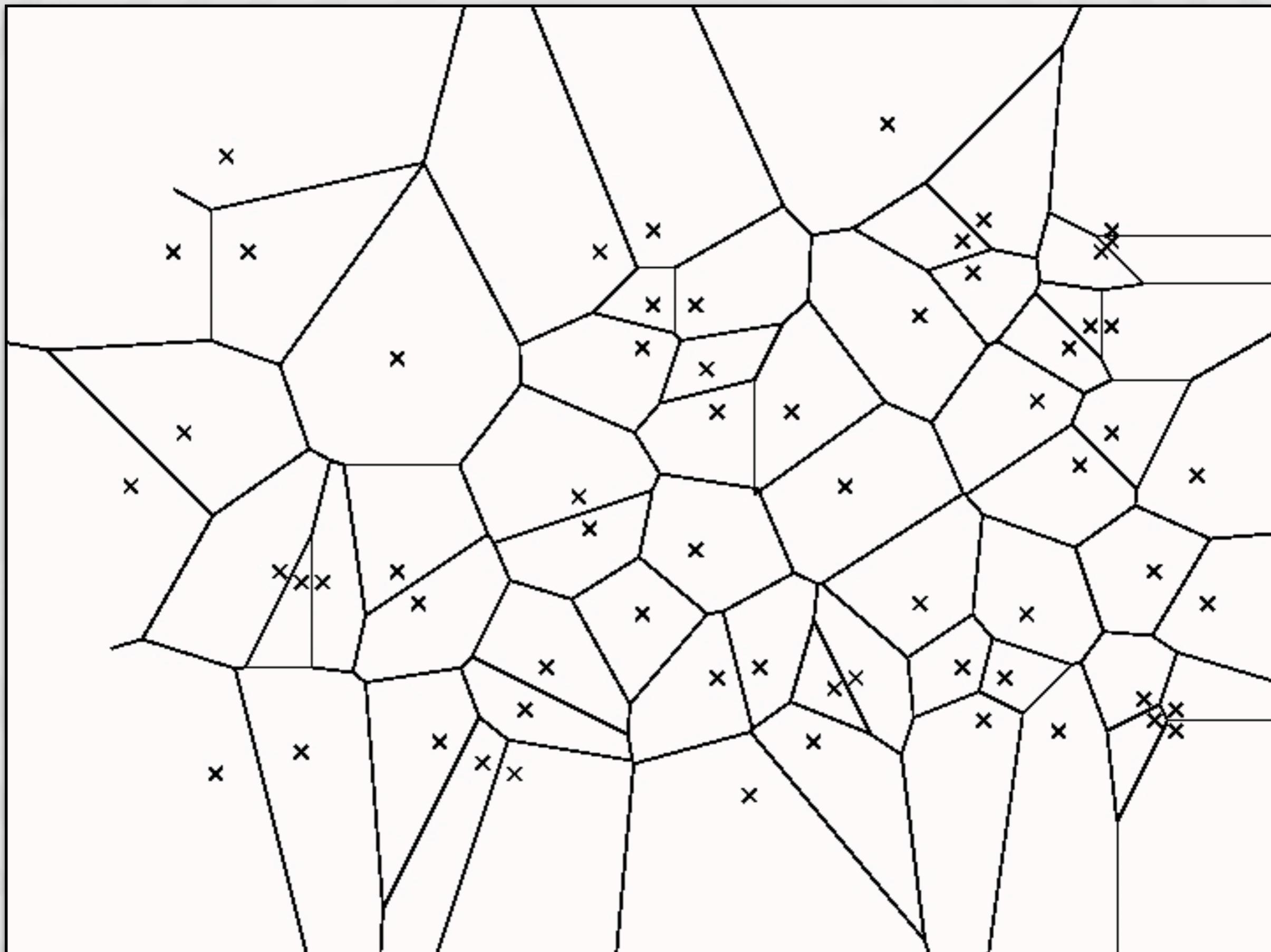
La **formule de Descartes-Euler** lie le nombre s de sommets, le nombre a d'arêtes et le nombre n de faces de la façon suivante : $(s + 1) - a + n = 2$

Chaque arête a exactement deux sommets, chaque sommet est incident à au moins trois arêtes : $2a = \#\{(s, a) \mid s \in a\} \geq 3(s + 1)$

Donc le nombre de sommets du diagramme de n points du plan est borné par $2n - 5$

La queue a donc $O(n)$ éléments et chaque opération est en $O(\log n)$

Donc une complexité en $O(n \log n)$



10 Annexe - Algorithme de Fortune

```

1 import math
2 from tkinter import *
3 import numpy as np
4 import sys
5 from Structures import Point
6 from PolygonUtils import affiner_poly
7 from TkinterUtils import *
8 from Examples import *
9
10 # VARIABLES GLOBALES
11 y_pos=None # La position y de la ligne de passage
12 Queue=None # La queue des événements
13 Racine=None # L'arbre de la beachline
14 Voronoi=None # La DCEL du diagramme
15 Sommets=[] # Sommets du diagramme
16 debug_logs=False # Afficher les infos de debug
17 debug_coords=False # Affiche les coordonées du pointeur dans le terminal
18 debug_grap=False
19 debug_aretes=False # Aficher les aretes (certaines sont buguées pour l'instant mais
# c'est pour l'esthétique car ce qui nous intéresse c'est les points)
20 eps=0.00001 # Epsilon
21 cvs=None #Le canvas global
22 dessin_on=False #Est-on en mode dessin
23
24 y_max=1000 # Les bordures du cadre
25 x_max=1000 # Les bordures du cadre
26
27
28 # OBJETS
29 class Arete:
30     valide = False # Pour savoir quand l'arete est achevée
31
32     # Forme de la droite bissectrice du segment [gauche, droite]:
33     # Formule :  $y = m * x + k$ 
34     m = None
35     k = None
36
37     # Point d'origine et de fin
38     origine = None
39     fin = None # initialisé à None, remplit à la complétion d'un circle event
40
41     # Initialisation de l'arête à partir de deux point de l'espace
42     # droite : point à droite du segment
43     # gauche : point à gauche du segment
44     # origine_x : abscisse du point d'origine associé
45     def __init__(self, gauche, droite, origine_x = None):
46         self.gauche = gauche
47         self.droite = droite
48
49         # équation de la bissectrice des deux points
50         if droite.y == gauche.y: # Cas particulier où les deux points sont sur la
# même ordonnée Y

```

```

51             self.m = math.inf
52
53             self.k = 0
54         else:
55             self.m = - (gauche.x - droite.x) / (gauche.y - droite.y)
56             self.k = (0.5 * (gauche.x ** 2 - droite.x ** 2 + gauche.y ** 2 -
droite.y ** 2)) / (gauche.y - droite.y)
57
58         if origine_x != None:
59             self.origine = Point(origine_x, self.calcul_y(origine_x))
60
61         #  $y=mx+k$ 
62         def calcul_y(self, x):
63             if self.m == math.inf:
64                 return None
65             else:
66                 return x * self.m + self.k
67
68         #  $x=(y-k)/m$ 
69         def calcul_x(self, y):
70             if self.m == math.inf:
71                 return self.origine.x
72             else:
73                 return (y - self.k) / self.m
74
75         defachever(self, fin):
76             self.fin = fin
77             self.valide = True
78
79         def __str__(self):
80             return "Arete: ori={origine}, fin={fin}".format(origine = self.origine, fin =
self.fin)
81
82 class CircleEvt(Point):
83     # parabole : la parabole qui a généré l'évènement
84     # centre : le centre du cercle
85     # x,y : les coordonnées du bas du cercle
86
87     #
88     #
89     #
90     # ;
91     #
92     # |-----> centre
93     # :
94     #
95     #
96     #
97     #
98     #
99     #
100    #
101    #

```

11 Annexe - Algorithme de Fortune

```

102
103
104     def __init__(self, x, y, parabole, centre):
105         super().__init__(x, y)
106         self.parabole=parabole
107         self.centre=centre
108         self.actif=True
109
110     def __str__(self):
111         return "Circle Event : {point}".format(point = super(CercleEvnt,
112 self).__str__())
113
114 class SiteEvnt(Point):
115     def __init__(self, x, y, parabole=None):
116         super().__init__(x, y)
117         self.parabole=parabole
118         self.id = ""
119
120     def __str__(self):
121         return "Site Event -> point:{point} / parabole:{para}".format(point =
super(SiteEvnt, self).__str__(), para=self.parabole.__str__())
122
123 class Parabole:
124     # site : le point directeur de la parabole
125     # prec : la parabole précédente dans le défilement gauche droite des paraboles
126     # suiv : la parabole suivante dans le défilement gauche droite des paraboles
127     # ar_droite / ar_gauche : arete droite / gauche
128     # evnt : si la parabole est lié à un circle event
129
130     # On fait une liste chaînée des paraboles : -prec est la parabole de gauche dans
la file
131     #                                     -suiv est la parabole de droite dans
la file
132
133     prec = None
134     suiv = None
135     evnt = None
136
137     def __init__(self, site, prec=None, suiv=None, ar_gauche=None, ar_droite=None):
138         self.site = site
139         self.prec = prec
140         self.suiv = suiv
141         self.ar_droite = ar_droite
142         self.ar_gauche = ar_gauche
143
144     # Retourne a, b et c
145     # solution de y=ax^2+bx+c pour la parabole
146     def expr_poly(self, y_actuel):
147         xs=self.site.x
148         ys=self.site.y
149         if ys != y_actuel:
150             g=1/(2*(ys-y_actuel))
151             a=1*g

```

```

151             b=-2*xs*g
152             c=(ys**2 - y_actuel**2 + xs**2)*g
153             return a, b, c
154         else:
155             g=math.inf
156             # On retourne None si les deux paraboles sont au même y
157             return None, None, None
158
159
160     def __str__(self):
161         return "Parabole : site = {site}, précédente ? {a_prec}, suivante ?
{a_suiv}".format(site = self.site, a_prec = self.prec != None, a_suiv = self.suiv !=
None)
162
163     # Gestion de la pile d'évènements
164     class QueueOfEvents:
165         queue=[]
166
167         def __init__(self, points):
168             sites=[ SiteEvnt(p.x,p.y) for p in points ]
169             sites.sort(key=lambda p: p.y)
170             self.queue=sites
171             self.fix_meme_coord()
172
173         def fix_meme_coord(self):
174             for i in range(1,len(self.queue)):
175                 if self.queue[i].y==self.queue[i-1].y:
176                     self.queue[i].y=self.queue[i].y+0.1
177
178         def vide(self):
179             if len(self.queue)==0:
180                 return True
181             else:
182                 return False
183
184         # Insertion de l'évènement de façon ordonnée par rapport à sa coordonnée Y
185         # (utiliser une pile binaire à la place)
186         def insert(self, evnt):
187             i=0
188             while i<len(self.queue):
189                 if self.queue[i].y >= evnt.y:
190                     break
191                 i+=1
192             self.queue= self.queue[:i] + [evnt] + self.queue[i:]
193
194         # Retire le premier élément de la queue et le retourne
195         def pop(self):
196             if self.vide():
197                 return False
198             else:
199                 x=self.queue.pop(0)
200                 return x

```

12) Annexe - Algorithme de Fortune

```

201 def remove(self, evnt):
202     try:
203         self.queue.remove(evnt)
204     except ValueError:
205         return
206
207 def debug(self):
208     debug("Dumping de la pile : ")
209     for event in self.queue:
210         debug(event)
211
212 def para_intersect_x(para1, para2, y=None):
213     global y_pos, eps
214     if not y: y=y_pos
215
216     if para1.site.y == y:
217         y=y+eps
218
219     a1,b1,c1=para1.expr_poly(y)
220     a2,b2,c2=para2.expr_poly(y)
221
222     # Si a2==None, la Beachline est actuellement à la position y_pos=para2.site.y
223     # Donc le point d'intersection avec para2 est le site de cette même parabole
224     if a2 == None:
225         return para2.site.x
226     else:
227         a=a1-a2
228         b=b1-b2
229         c=c1-c2
230
231         det = b*b - 4*a*c
232
233         # Si det<0, cela signifie (étude empirique) que para2.site.y est quasiment
234         # égal à y_pos
235         # les valeurs renvoyées par expr_poly sont alors de l'ordre de math.inf d'où
236         # le signe <0
237         if det<0:
238             return para2.site.x
239         else:
240             if a==0:
241                 if c==0:
242                     return para2.site.x
243                 else:
244                     return -c/b
245
246             x1=(-b-math.sqrt(det))/(2*a)
247             x2=(-b+math.sqrt(det))/(2*a)
248
249             # On prend toujours l'intersection de droite
250             if para1.site.y < para2.site.y:
251                 return min(x1, x2)
252             else:
253                 return max(x1, x2)

```

```

252
253 def traite_site_evnt(evnt):
254     global Racine, Voronoi, Queue
255
256     debug("Site event -> Traitement du site {}".format(evnt))
257
258     # Si c'est le premier évènement traité, on a pas encore de racine dans le graphe
259     # des paraboles
260     if Racine == None:
261         Racine = Parabole(evnt) # Création de la première parabole et affectation à
262         la racine
263         debug("Site Event -> Pas encore de parabole, création de la première. " +
264         Racine.__str__())
265         return
266
267     # Sinon, recherche de la parabole qui est directement au dessus du site en cours
268     # de traitement, et qui va être divisée
269     parabole = Racine
270     while parabole.suiv != None and para_intersect_x(parabole, parabole.suiv) <=
271         evnt.x:
272         parabole = parabole.suiv
273
274     # Création des deux arrêtes opposées à partir de ce point d'intersection
275     arete_gauche = Arete(parabole.site, evnt, evnt.x)
276     arete_droite = Arete(evnt, parabole.site, evnt.x)
277
278     # Réordonne les paraboles : parabole.prec <-> parabole <-> parabole_evnt <->
279     parabole_droite <-> parabole.suiv
280
281     # Création de la parabole associé au site traité et la parabole trouvée devient
282     # la parabole gauche de cette nouvelle parabole
283     parabole_evnt = Parabole(evnt, parabole, None, arete_gauche, arete_droite)
284     parabole_extreme_droite = parabole.suiv
285     parabole.suiv = parabole_evnt
286     parabole.ar_droite = arete_gauche
287
288     # Découpe la parabole trouvée en deux, avec les arêtes au milieu
289     parabole_droite = Parabole(parabole.site, parabole_evnt,
290     parabole_extreme_droite, arete_droite, parabole_extreme_droite.ar_gauche if
291     parabole_extreme_droite else None)
292     parabole_evnt.suiv = parabole_droite
293     if parabole_extreme_droite:
294         parabole_extreme_droite.prec = parabole_droite
295         if parabole_extreme_droite.ar_gauche.valide == False:
296             parabole_extreme_droite.ar_gauche = arete_droite

```

13 Annexe - Algorithme de Fortune

```

290 # Parabole r est coupée :
291 # X X X X X X X X X
292 # X X X X X X X X X
293 # X X X X X X X X X
294 # X X X X X X X X X
295 # XX XX XX XX XX XX X
296 # X X X X X X X X X
297 # XX XX +---> X X XX --- X X XX --- X X
298 # XX XX X XXX XX X XX XX X X X
299 # XXX XXX XXX XXX X X X
300 #
301 #
302 # X
303 #
304 # Parabole r de départ      Rajout de evnt    Les deux paraboles    para1
305 #                                 r et para2        à part
306 #                                 séparées
307 #
308 # On enlève les possibles circle events associés
309 if parabole.evnt:
310     Queue.remove(parabole.evnt)
311     parabole.evnt.actif=False
312     parabole.evnt=None
313
314 creer_cercle_evnt(parabole)
315 creer_cercle_evnt(parabole_droite)
316
317 Voronoi.append(arete_gauche)
318 Voronoi.append(arete_droite)
319
320
321 # Crédation d'un "Circle" event, c'est à dire une ordonnée Y correspondant au cercle passant par les trois sites
322 # formés par les sites de la parabole à gauche, la courante et celle à droite du site de la parabole courante
323 def creer_cercle_evnt(parabole):
324     global y_pos, y_max, Queue, debug_grap, cvs, dessin_on
325
326     debug("Création d'un circle event. " + parabole.__str__())
327
328     # Il faut trois sites pour pouvoir produire un Circle events
329     if parabole.prec == None or parabole.suiv == None: return
330
331     # Extraction des trois sites à utiliser pour trouver le cercle inscrit
332     site_gauche = parabole.prec.site
333     site_parabole = parabole.site
334     site_droit = parabole.suiv.site
335
336     # Est il possible de créer un cercle inscrit passant par les trois sites
337     # Calcul du déterminant des trois points pour voir si ils sont dans le sens antihoraire
338     if ((site_parabole.x - site_gauche.x) * (site_droit.y - site_gauche.y) - (site_droit.x - site_gauche.x) *
339         (site_parabole.y - site_gauche.y) <= 0):
340         return

```

```

339
340     # Si la parabole courant n'a pas d'arête gauche ou droite, pas de Circle Event
341     if parabole.ar_gauche == None or parabole.ar_droite == None: return
342
343     # Le centre du cercle est à l'intersection des arêtes de la parabole traitée
344     centre_cercle = intersection_aretes(parabole.ar_gauche, parabole.ar_droite)
345     if centre_cercle==None or parabole.site==None: return
346     else : rayon = math.sqrt((centre_cercle.x - parabole.site.x) ** 2 +
347                               (centre_cercle.y - parabole.site.y) ** 2)
348
349     if debug_grap and dessin_on:
350         cvs.create_oval(centre_cercle.x - rayon,
351                         centre_cercle.y - rayon,
352                         centre_cercle.x + rayon,
353                         centre_cercle.y + rayon,
354                         outline = "grey",
355                         dash=(3,5),
356                         width = 1)
357
358     # On détermine l'ordonnée Y de la tangente afin de positionner le circle event
359     # dans la pile
360     evnt_position_y = rayon + centre_cercle.y
361
362     # Cette position doit être après la beach line (et plus petite que la limite i.e
363     evnt_position_y < y_max mais élevé pour l'instant)
364     if evnt_position_y > y_pos:
365         cirle_event = CercleEvnt(centre_cercle.x, evnt_position_y, parabole,
366                                   centre_cercle)
367         parabole.evnt = cirle_event
368         debug("Circle event ajoutée à la pile. " + cirle_event.__str__())
369         Queue.insert(cirle_event)
370
371     # Permet de déterminer le point d'intersection de deux arêtes
372     def intersection_aretes(arete_1, arete_2):
373         if arete_1.m == math.inf:
374             return Point(arete_1.origine.x, arete_2.calcul_y(arete_1.origine.x))
375         else:
376             if arete_2.m == math.inf:
377                 return Point(arete_2.origine.x, arete_1.calcul_y(arete_2.origine.x))
378             else:
379                 dir_dif = arete_1.m - arete_2.m
380                 if dir_dif == 0: return None
381                 x = (arete_2.k - arete_1.k) / dir_dif
382                 y = arete_1.calcul_y(x)
383                 return Point(x, y)

```

14 Annexe - Algorithme de Fortune

```

381 # Traitement d'un Circle Event
382 def traite_cercle_evnt(evnt):
383     global Voronoi, Sommets, Queue
384
385     debug("Cercle event -> Traitement du cercle induit par le site
386     {}".format(evnt.parbole.site))
387
388     parbole = evnt.parbole # Parabole qui a crée cet event
389
390     # On désactive les circle event des paraboles adjacentes, car ils ne sont plus
391     # nécessaires
392     if parbole.prec.evnt:
393         Queue.remove(parbole.prec.evnt)
394         parbole.prec.evnt.actif = False
395     if parbole.suiv.evnt:
396         parbole.suiv.evnt.actif = False
397         Queue.remove(parbole.suiv.evnt)
398
399     # Création d'une nouvelle arête entre deux sites des paraboles adjacentes et on
400     # les relie aux paraboles associées
401     nouvelle_arete = Arete(parbole.prec.site, parbole.suiv.site)
402     nouvelle_arete.origine = evnt.centre
403
404     # On ajuste la beachline en retirant cette parbole
405     parbole.prec.ar_droite = nouvelle_arete
406     parbole.suiv.ar_gauche = nouvelle_arete
407     parbole.prec.suiv = parbole.suiv
408     parbole.suiv.prec = parbole.prec
409
410     Voronoi.append(nouvelle_arete)
411
412     if not point_dehors(evnt.centre):
413         Sommets.append(evnt.centre)
414
415     # Les arêtes liées à la parbole courante sont maintenant achevées
416     parbole.ar_droite.achever(evnt.centre)
417     parbole.ar_gauche.achever(evnt.centre)
418
419     creer_cercle_evnt(parbole.prec)
420     creer_cercle_evnt(parbole.suiv)
421
422 def terminer_aretes():
423     global Voronoi, x_max, y_max
424
425     for ar in Voronoi:
426         if ar.origine.y==None:
427             return
428         else:
429             if ar.fin == None and not point_dehors(ar.origine) and not ar.valide:
430                 yl = ar.calcul_y(x_max)
431                 if ar.k >= 0:
432                     if yl>y_max:
433                         xl=ar.calcul_x(y_max)
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482

```

```

432             xl=x_max
433             ar.achever(Point(xl,yl))
434
435         else:
436             yl=ar.calcul_y(x_max)
437             if yl<0:
438                 xl=ar.calcul_x(0)
439                 yl=0
440             else:
441                 xl=x_max
442                 ar.achever(Point(xl,yl))
443
444 def point_dehors(point):
445     global x_max, y_max
446     return point.x < 0 or point.x > x_max or point.y < 0 or point.y > y_max
447
448 # Fonction principale
449 def fortune(points, xmax=None, ymax=None, canvas=None):
450     global Queue, Racine, Voronoi, y_pos, Sommets, x_max, y_max
451
452     if xmax!=None and ymax!=None:
453         x_max=xmax
454         y_max=yymax
455
456     Queue = QueueOfEvents(points)
457     Voronoi=[]
458
459     while not Queue.vide():
460         event = Queue.pop()
461         y_pos = event.y
462         debug("Traitement d'un nouvel évènement. Position de la beach_line =
463     {}".format(y_pos))
464         debug(event)
465         if isinstance(event, SiteEvt):
466             traite_site_evnt(event)
467         else:
468             if event.actif:
469                 traite_cercle_evnt(event)
470             else:
471                 print(event.__str__())
472
473     return Sommets
474
475
476 ## Fonctions d'utils / GUI
477
478 def convertir_point(points_t):
479     res=[]
480     for p in points_t:
481         res.append(Point(p[0], p[1]))
482
483     return res

```

15) Annexe - Algorithme de Fortune

```
483 def debug(data):
484     global debug_logs
485     if debug_logs:
486         print(data)
487
488 def coeff_droite(p1, p2):
489     if p1.x == p2.x:
490         return 0
491     else:
492         return (p2.y-p1.y)/(p2.x-p1.x)
493
494 # droite y=kx+m
495 # renvoi (x,y(x))
496 def point_droite(k, m, x):
497     return Point(x, k*x + m)
498
499
500 def dessin(data, interieur=None):
501     global Voronoi, Sommets, cvs, dessin_on
502     dessin_on = True
503     root = Tk()
504     root.title('VORONOI')
505     root.geometry("1000x1000")
506     root.bind('<Escape>', lambda i: sys.exit(0))
507     cvs=Canvas(root, width=1000, height=1000, background="white")
508     cvs.grid(row = 0, column = 0)
509
510     if isinstance(data[0], tuple):
511         points = convertir_point(data)
512     else:
513         points=data
514
515     dessin_poly(points, cvs)
516
517     if interieur:
518         if isinstance(data[0], tuple):
519             inte_points=convertir_point(intérieur)
520         else:
521             inte_points=intérieur
522             dessin_poly(inte_points, cvs)
523             points=points+inte_points
524
525     for point in points:
526         dessin_point(point, 5, cvs)
527
528     fortune(points, cvs)
529
530     debug('Longueur résultat : ' + str(len(Voronoi)))
531
532     debug('Résultat avant compression :')
533     for arete in Voronoi:
534         if arete:
535             if debug_logs:
536                 debug(arete)
```

```
538
539     debug('Résultat après compression :')
540     for arete in Voronoi:
541         if arete:
542             if arete.fin==None:
543                 print(arete.__str__())
544             if debug_aretes:
545                 dessin_arete(arete, cvs)
546             if debug_logs:
547                 debug(arete)
548             if debug_grap:
549                 dessin_ori_arete(arete, cvs)
550
551     # Ce sont les points verts ceux de Voronoi
552     for sommet in Sommets:
553         dessin_point(sommet, 8, cvs, couleur="green")
554
555     root.bind('<Button 1>', coord_souris)
556     root.bind('<Motion>', mouvement)
557     root.mainloop()
558
```