



## Algorithme et structure de données :

Détection d'inclusion de polygones

ZOUGGARI Samy

CHELLAT Ryan

Encadré par Frédéric Wagner

2024



# Table des matières

Introduction .....	1
<b>I- Explication des algorithmes .....</b>	<b>1</b>
Explication de la méthode de Ray Casting .....	1
Algorithme pour trouver une intersection.....	2
<b>Algorithme (naïf) .....</b>	<b>3</b>
Complexité .....	4
Complexité du calcul de surfaces .....	4
Commentaire .....	5
<b>2e algorithme (tri de surface) .....</b>	<b>6</b>
Commentaire .....	7
Calculs de complexité .....	7
<b>3e algorithme (quadrants + tri) .....</b>	<b>8</b>
Construction des quadrants .....	8
Calculs de complexité pour les quadrants .....	9
En général .....	10
<b>II- Partie expérimentale : tests de ces algorithmes.....</b>	<b>10</b>
Générateur circnest/sqnest .....	12
Présentation .....	12
Observations .....	13
Commentaires .....	14
Complexité temporelle .....	14
Générateur circgrid/sqgrid/hexa.....	16
Générateurs sqgrid/circline.....	16
Générateur hexa.....	16
Observations .....	17
Commentaires .....	18
Complexité temporelle .....	19
Générateur Fakesqnest .....	20
Observations .....	20
Observations .....	21
Générateur Sierp/Sqfrac.....	22
Générateur Sierp.....	22
Générateur Sqfrac.....	23
Observations .....	23
Commentaires .....	25
<b>III - Critique des générateurs.....</b>	<b>26</b>
<b>IV - Synthèse .....</b>	<b>27</b>
<b>V- Propositions d'algorithmes .....</b>	<b>28</b>
1)Algorithme tri décroissant+quadrant .....	28
2) Algorithme new(tri+quadrant).....	29
Observations .....	29
Commentaires .....	30
Conclusion .....	30
<b>VI- Conclusion générale :.....</b>	<b>31</b>
<b>Annexe .....</b>	<b>32</b>

## Introduction :

Dans le domaine de l'informatique géométrique, la détection d'inclusion de polygones constitue un problème fondamental avec une large gamme d'applications. Ce projet vise à élaborer des algorithmes efficaces pour déterminer quel polygone est inclus dans quel autre à partir d'un ensemble de polygones. Pour ce faire, nous allons explorer des algorithmes de base de géométrie algorithmique tout en développant des techniques spécifiques pour résoudre ce défi.

Le processus de détection d'inclusion repose sur plusieurs aspects clés. Tout d'abord, nous devons mettre en œuvre des algorithmes pour déterminer si un point se trouve à l'intérieur d'un polygone, une étape cruciale pour la détection d'inclusion. Enfin, nous comparerons des paires de polygones pour identifier les relations d'inclusion.

Un aspect important de ce projet est l'optimisation des performances tout en garantissant la précision des résultats.

Ce projet offre une opportunité unique d'explorer des concepts avancés en géométrie algorithmique et de mettre en œuvre des solutions pratiques pour un problème concret.

## I- Explication des algorithmes

Tout d'abord, pour ces 3 algorithmes que je vais présenter. J'utilise la méthode de Ray-Casting pour détecter si un point est dans un polygone.

### Explication de la méthode de Ray Casting :

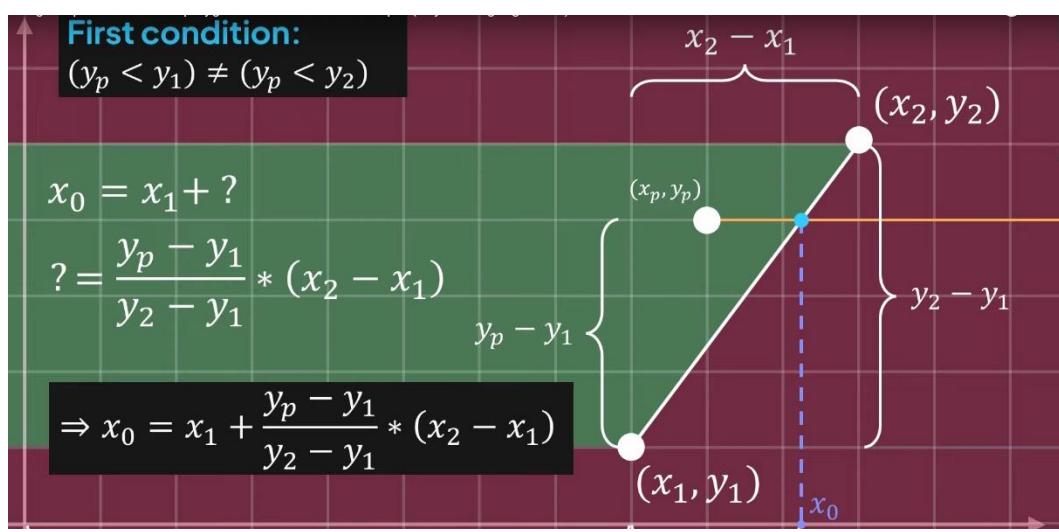


Figure 1 Méthode Ray Casting (1)

### Algorithme pour trouver une intersection:

Dans la figure (1). On considère un point  $(x_p; y_p)$  et un segment. On voit que si l'ordonnée  $y_p$  du point n'est pas inclus dans  $[y_1; y_2]$  (l'intervalle entre les ordonnées des deux bouts du segment), alors le rayon n'a pas de point d'intersection avec ce segment. De plus, si  $x_p < x_1$ , alors le rayon a forcément un point d'intersection avec le segment. Mais si,  $x_1 < x_p < x_2$ . Alors pour savoir si le rayon possède un point d'intersection avec le segment, il faut voir si ce point est à gauche du segment. Pour le savoir, on calcule l'éventuel point d'intersection  $x_0 = x_1 + \left[ \frac{(x_2-x_1)(y_p-y_1)}{(y_2-y_1)} \right]$ . Ainsi, si  $x_p < x_0$ . Alors, le rayon croise le segment en ce point  $x_0$ . Sinon, il n'y a pas d'intersection.

Maintenant, soit un polygone  $i$ , et un polygone  $j$  différent de  $i$ . On veut savoir si le polygone  $i$  est inclus dans le polygone  $j$ .

On trace imaginairement un trait horizontal partant d'un point du polygone  $i$  et allant vers la droite. On note ce trait **le rayon**.

Ensuite, on itère sur tous les segments du polygone  $j$  pour voir si ce segment possède un point d'intersection avec le rayon. Si celui-ci existe, on incrémente un compteur initialisé à zéro.

Ainsi, le compteur représente le nombre de fois que le rayon croise le polygone  $j$ .

La méthode de Ray Casting consiste à dire que si le rayon coupe le polygone  $j$  un nombre impair de fois, alors le polygone  $i$  est inclus dans le polygone  $j$ . Sinon, il n'est pas inclus dedans (figure 2).

Donc, lorsque l'itération se termine, on regarde la valeur du compteur, et on retourne True si la valeur du compteur est impair, False sinon.

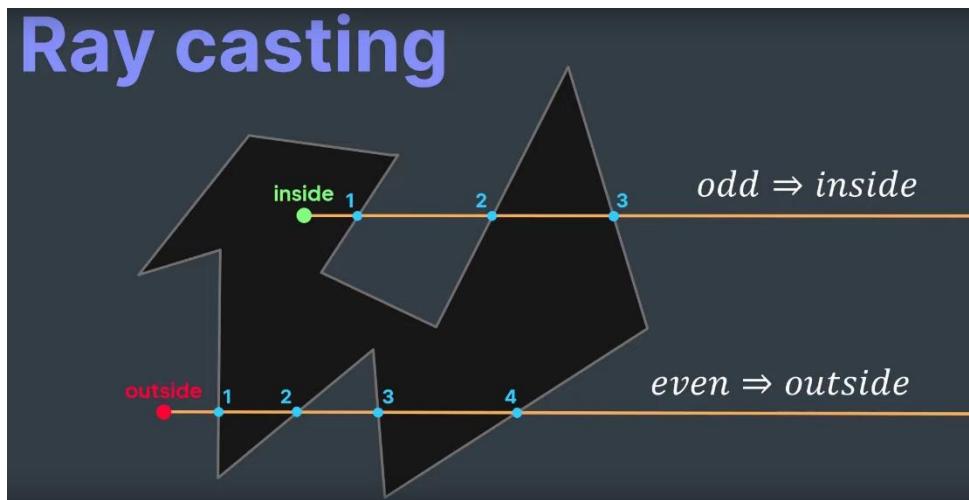


Figure 2 Méthode Ray Casting (2)

### Algorithme (naïf) :

On note  $n$  le nombre de polygones du fichier .poly

J'initialise d'abord un tableau de taille  $n$ , toutes les cases sont à -1. C'est le tableau qu'on veut retourner. (cf Annexe 1 pour voir l'algorithme)

Nous avons une fonction principale **detect\_inclusions** qui possède 2 boucles imbriquées.

La première boucle qui itère sur les  $n$  polygones. La deuxième boucle itère sur les  $n$  polygones pour chaque polygone  $i$ , (au final  $n^2$  tours de boucle).

Cette fonction teste l'inclusion du polygone  $i$  dans un polygone  $j$  à l'aide de 3 conditions:

1-  $i \neq j$ . Si ce n'est pas le cas, on passe au polygone  $j + 1$ , sinon, on regarde la deuxième condition.

2 -  $\text{Surface}(i) < \text{Surface}(j)$

Si la surface du polygone  $i$  est plus grande que celle du polygone  $j$  alors le polygone  $i$  ne peut être inclus dans le polygone  $j$ . Alors on passe directement au polygone  $j + 1$  directement. Sinon, on teste la deuxième condition.

3- `point_in_polygon(point,polygone j)` renvoie True

Ici, « point » est le premier point du polygone  $i$ .

On fait un appel à une fonction `point_in_polygon` qui renvoie True si le point est inclus dans le polygone  $j$  en utilisant la méthode de Ray Casting.

Lorsque la fonction `point_in_polygon` retourne True, cela veut dire que le polygone  $i$  est inclus dans le polygone  $j$ .

Alors on ajoute dans la liste **polygones\_inclus** (initialisée pour chaque polygone  $i$ , donc à chaque tour de la première boucle) le tuple  $(j, \text{abs}(\text{surface}(\text{polygone } j)))$ .

L'idée derrière cela est que, étant donné plusieurs polygones qui incluent le polygone  $i$ , le polygone qui l'inclue directement est celui dont la surface est la plus petite.

Alors l'objectif est de créer pour chaque polygone  $i$ , une liste de tuple contenant tous les polygones qui l'incluent.

Le polygone qui l'inclut directement est trouvé en effectuant une recherche sur le min des surfaces des polygones de la liste.

### Complexité :

On note  $n$  le nombre de polygones et  $N$  le nombre de points au total. On suppose ici que  $N = k \times n$  avec  $k$  une constante. C'est-à-dire que le nombre de points par polygone est une constante.

Pour simplifier les calculs (et ce sera souvent le cas), on suppose qu'il y a équirépartition du nombre de points par polygones. Par conséquent, le nombre de points par polygone vaut  $k$  (constante). Et le nombre de segments est égal au nombre de points, donc  $k$  segments par polygones.

Ces boucles itèrent dans tous les cas sur tous les polygones. C'est-à-dire que l'inclusion du polygone  $i$  va être testée avec tous les autres polygones. Il y a donc  $n^2$  tours de boucles.

### Complexité du calcul de surfaces :

On utilise une méthode cross\_product : 2 multiplications

Le calcul de surface se fait en itérant sur le nombre de points du polygons (divisé par 2), finalement on a  $k$  opérations pour chaque polygone  $i$ . Donc le nombre total d'opérations pour calculer la surface de tout le polygone est la somme des  $k$  ( $i$  variant de 1 à  $n$ ), ou alors le nombre total de points du polygone.

Si  $N$  est le nombre total de points de tous les polygones, alors le calcul de surface est en  $O(N)$ , donc en  $O(n)$ .

A chaque tour de boucles, on a 2 à 3 calculs de surface. Le calcul de surface se fait en  $O(k)$ .

Si on suppose que python calcule la surface à chaque fois, et ne stocke pas une valeur déjà calculée, on a  $n^2 \times 2k$  à  $3k \times n^2$  opérations pour le calcul des surfaces.

Ensuite, on a au maximum  $n \times (n - 1)$  appels à la fonction `point_in_polygon` (si tous les polygones sont de surface égale) qui est en  $O(k)$ . Au minimum on a  $\frac{n*(n-1)}{2}$  appels (si les polygones sont générés par ordre croissant ou décroissant de surface).

Ensuite on crée une liste pour chaque tour de la première boucle, donc il y a  $n$  créations de listes, où on met dans le meilleur des cas 1 polygones, et dans le pire des cas  $n-1$  polygones.

Le calcul du min se fait en  $O(n)$ .

Donc, pour toutes les hypothèses, la complexité est en  $O(n^2)$  dans le meilleur et le pire des cas.

### Commentaire :

- Cet algorithme fonctionne mais n'est pas efficace. On teste l'inclusion entre toutes les combinaisons possibles de polygone et cela est coûteux. De plus, on calcule un nombre de fois élevé les surfaces de polygones (suivant notre hypothèse). On essaiera d'améliorer ça en programmant un nouvel algorithme qui réduit la complexité.

## 2e algorithme (tri de surface) :

Cet algorithme (cf Annexe 2) trie tout d'abord les polygones par ordre croissant de surface (en valeur absolue).

La logique derrière cet algorithme est que, considérant un polygone i, il ne peut être inclus dans un polygone j de surface inférieure. De ce fait, on réduit considérablement le nombre de tests.

De plus, étant donné un polygone ayant la i-ème plus grande surface. En itérant sur les (i+1 à n) polygones. Le premier à inclure le polygone i est celui qui va l'inclure directement. On **break** donc la boucle j et on passe au polygone i+1. On réduit donc aussi considérablement le nombre de tests.

De ce fait. Au pire des cas, le premier polygone teste l'inclusion avec n-1 polygones. Le 2e polygone teste l'inclusion avec n-2 polygones ... L'avant-dernier teste l'inclusion avec le dernier polygone seulement. Quant au dernier, celui-ci n'est pas testé car il possède la surface la plus grande, il ne peut donc être inclus dans aucun autre polygone.

Dans le meilleur des cas, le premier polygone teste l'inclusion avec le polygone suivant seulement, etc. Le dernier polygone n'est toujours pas testé.

Il y a donc  $\frac{n*(n-1)}{2}$  opérations au pire des cas et n-1 opérations au meilleur des cas.

De plus, contrairement au premier algorithme naïf qui calcule la surface de deux polygones à chaque test. Celui-ci calcule les n surfaces correspondant aux n polygones et les stocke dans un tableau. Ce qui réduit considérablement le calcul de surface qui peut être coûteux.

La méthode pour voir si un polygone est inclus dans un autre ne change pas du 1er algorithme. On utilise toujours la méthode de Ray casting en appelant une fonction auxiliaire `point_in_polygon`.

**Commentaire :**

- $n$  calculs de surfaces contrairement à  $n^2 \times 2k$  à  $3k \times n^2$  pour le 1er algorithme (plus d'autres calculs de surface pour le min, et une recherche du min d'une liste).
- Beaucoup moins de tours de boucles :  $n$  tests au meilleur des cas et  $\frac{(n-2)(n-1)}{2}$  au pire des cas contrairement à  $n \times (n - 1)$  pour le premier algorithme, dans tous les cas.
- Il n'y a pas besoin de stocker les polygones qui incluent le polygone  $i$  dans une liste à chaque tour de boucle for. ( $n$  listes créées, donc  $n$  calculs de min réalisés donc moins de  $n \times (n - 1)$  calculs de surfaces réalisés pour trouver l'inclusion directe pour le premier algo, ce qui est coûteux).

**Calculs de complexité :**

Pour la fonction point\_in\_polygon. Complexité en  $O(k)$  encore.

Pour la fonction detect\_inclusion,  $n$  calculs de surfaces ( $O(n)$ ) + un tri de surface +  $n-1$  tours de boucles à  $n*(n-1)/2$  tours de boucles.

La fonction sort en python est en  $O(n\log(n))$ .

Donc au final, l'algorithme et en  **$O(n^2)$  au pire des cas** et en  **$O(n\log(n))$  au meilleur des cas** (la fonction (dans le meilleur des cas) est dominée par le tri de surface).

**Commentaire :**

- Même si dans le pire des cas la complexité temporelle du second algorithme est la même que celle du 1er. En réalité, il y a beaucoup moins d'opérations.
- Plus généralement si le nombre de points des polygones  $N = k \times (n^k)$ , alors on a une complexité en  $O(\log(n) \times n^k)$  à  $O(n^k + 1)$ . Mais c'est rarement le cas.

### 3e algorithme (quadrants + tri):

Nb : cet algorithme (cf Annexe 3) a passé le test 0 en 4040 ms, le test 1 en 68045 ms et le test 2 en 23042 ms.

Cet algorithme est une amélioration du précédent.

Dans cet algorithme on rajoute la notion de quadrant. Un quadrant est le rectangle minimal qui contient un segment/polygone donné.

L'idée derrière ce nouvel algorithme est d'utiliser les quadrants des polygones pour moins appeler la fonction point\_in\_polygon.

Le principal changement de cet algorithme avec le précédent est l'implémentation d'une nouvelle fonction "quadrant" qui crée une liste de liste à 3 éléments (indice du polygone, surface du polygone en valeur absolue, quadrant du polygone).

### Construction des quadrants :

La construction d'un quadrant pour un polygone s'effectue en prenant les points extrêmes de ce polygone. Les coordonnées des 4 points de ce quadrant sera  $(x_{min}, y_{min})$ ,  $(x_{min}, y_{max})$ ,  $(x_{max}, y_{min})$  et  $(x_{max}, y_{max})$ . Avec  $x_{min}$ ,  $x_{max}$ ,  $y_{min}$ ,  $y_{max}$  les 4 coordonnées extrêmes du polygone.

Cette fonction sera appelée une seule fois par la fonction principale detect\_inclusion.

Ensuite dans la fonction detect\_inclusion. Pour chaque itération sur un polygone, on récupère son quadrant correspondant.

### La différence fondamentale avec l'algorithme précédent est celle-ci :

En gardant la même logique que précédemment. Au lieu de tester l'inclusion directe. On teste d'abord si les quadrants des polygones i et j se touchent grâce à la méthode intersect de la classe Quadrant.

Si ces derniers ne se touchent pas, alors on peut directement conclure que le polygone  $i$  n'est pas inclus dans le polygone  $j$ , et l'on n'a pas besoin d'appeler la fonction `point_in_polygon` pour conclure.

On a donc 2 conditions :

- 1- Test si le quadrant du polygone  $i$  et celui du polygone  $j$  se touchent (méthode `intersect`)
- 2- Si condition 2 vérifiée, appel à la fonction `point_in_polygon`

Ainsi, à chaque itération, on n'appelle pas tout le temps la fonction `point_in_polygon`. Ce qui peut faire gagner du temps (pas tout le temps, mais le plus souvent).

Le reste de l'algorithme suit la même logique que le précédent.

Calculs de complexité pour les quadrants :

- Création de  $n$  quadrants. La complexité de la méthode `bounding_quadrants` est en  $O(k)$ . Donc la complexité de la création des  $n$  quadrants est en  $O(n)$ .
- A chaque test d'intersection (méthode `intersect`), 2 opérations de comparaisons.

Différents cas :

1er cas : aucun polygone n'est inclus dans l'autre. Donc on ne vérifie jamais la condition

Donc complexité en  $O(n^2)$ . En effet, on va itérer sur tous les polygones  $\binom{(n-1)(n-2)}{2}$  car on ne passera pas par l'instruction `break`. Ce premier cas a la particularité d'avoir une complexité qui ne dépend pas du nombre de points par polygones.

Si  $N = k \times n^i$ , (ce qui est rare). Quel que soit  $i$ , la complexité sera en  $O(n^2)$

2e cas : meilleur cas de l'algorithme précédent.

Complexité équivalente au meilleur cas de l'algorithme précédent.

Cependant, on aura plus d'opérations. En effet par rapport à l'algorithme précédent, on a  $n$  constructions de quadrants, et  $n$  tests d'intersections entre quadrants. Ce qui rajoute des opérations. Finalement, complexité en  $(O(n\log(n)))$ .

Pire cas: si le dernier polygone est celui qui inclut tous les autres et que les quadrants de tous les polygones se touchent.

Dans ce cas on a  $\frac{n(n-1)}{2}$  tests d'intersections entre quadrants et le même nombre d'appels à la fonction point\_in\_polygon.

L'appel à la fonction point\_in\_polygon est plus coûteux que le test d'intersection. Donc la complexité de ce cas est la même que la complexité au pire cas de l'algorithme précédent  $O(n^2)$ .

### En général :

Ce qu'on perd à construire les quadrants et à effectuer les tests entre quadrants (peu coûteux). On le gagne à effectuer moins d'appels à la fonction point\_in\_polygon qui est plus coûteuse que les tests.

Donc même si pour certains cas l'algorithme 2 sera un peu plus rapide que celui-ci. Dans la plupart des cas, c'est celui-ci qui sera plus rapide.

Complexité temporelle : meilleur des cas  $O(n\log n)$ . Pire des cas ( $O(n^2)$ ).

## II- Partie expérimentale : tests de ces algorithmes

Nous disposons pour cette partie d'un jeu de plusieurs tests. Ce jeu a été récupéré sur gitlab. Malheureusement, la génération de polygones est un problème dont nous nous sommes pris trop tard. Nous avons passé beaucoup de temps sur les algorithmes car nous pensions que ceux-ci étaient peu efficaces. En effet, lorsqu'on les soumettait au test 0, aucun algorithme ne passait. Et nous avons cru pendant longtemps que c'était à cause de leur inefficacité. Finalement, cela s'est révélé être un simple oubli de shebang.

Nous avons préféré nous concentrer sur la rédaction du rapport plutôt que de générer soi-même des polygones, car le plus important est l'algorithme et le lien entre l'entrée et l'algorithme et pas la génération de polygone en elle-même.

Dans cette partie nous allons présenter ces tests un par un. Ensuite nous allons présenter les performances des 3 algorithmes sur ces tests, et commenter les performances, pour mettre en évidence un lien entre les algorithmes et les entrées.

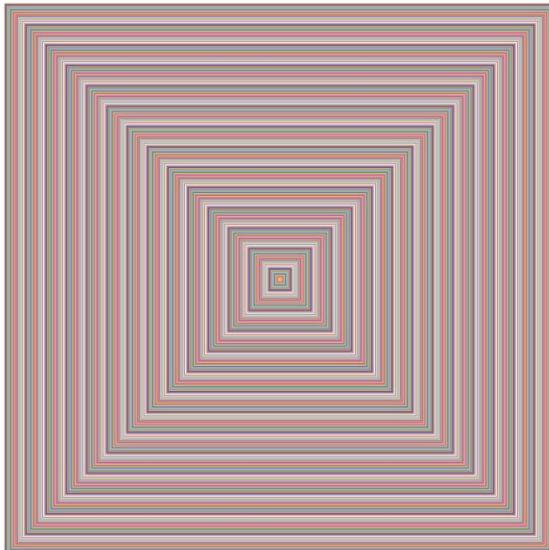
Le nombre d'opérations est dans le même ordre de grandeur que le nombre de tours de boucles. Comme il est difficile de théoriser complètement sur le nombre d'opérations. On théorise sur le nombre de tours de boucles et on en déduit la complexité.

**Remarque :** Comme le nombre de tours de boucles est le même pour l'algorithme tri et l'algorithme tri+quadrants. Au niveau des observations, nous avons décidé de mettre qu'un seul graphique représentant le nombre de tours de boucles en fonction du nombre de polygones pour les deux algorithmes.

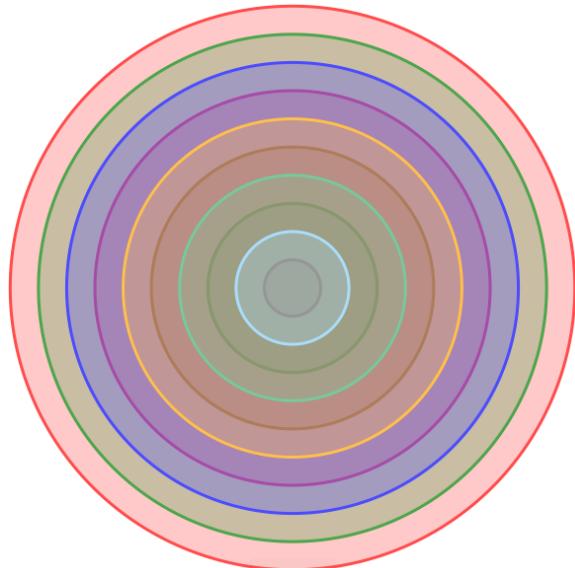
### Générateur circnest/sqnest :

#### Présentation :

Le générateur **sqnest** (respectivement circnest) construit des carrés (respectivement cercles) imbriqués. Le premier polygone est le plus grand. Le deuxième est plus petit, donc inclus dans le premier. Le troisième est plus petit que le deuxième (mais plus grand que le 4e) etc. Le dernier polygone est le plus petit.



500 carrés générés avec sqnest



10 cercles générés avec circnest

## Observations :

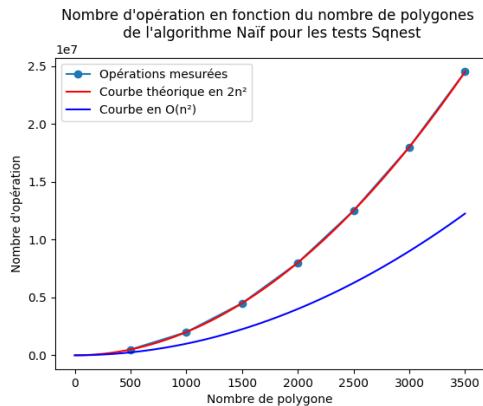


Figure 3 Naïf Sqnest

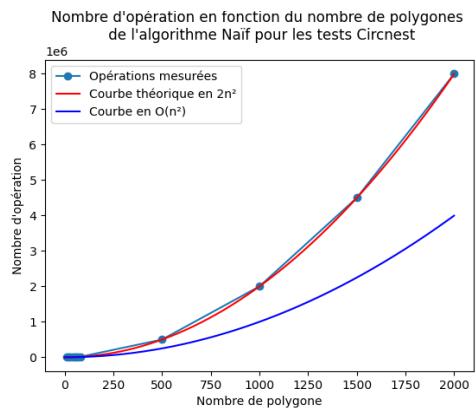


Figure 5 Naïf Circnest

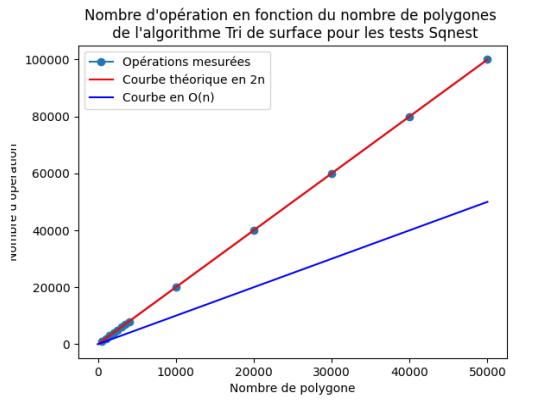


Figure 5 Tri de surface Sqnest

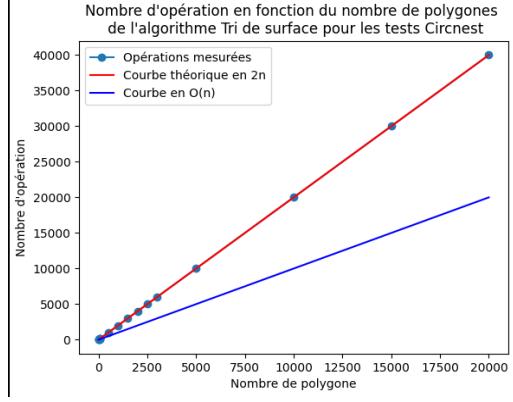


Figure 6 Tri de surface Circnest

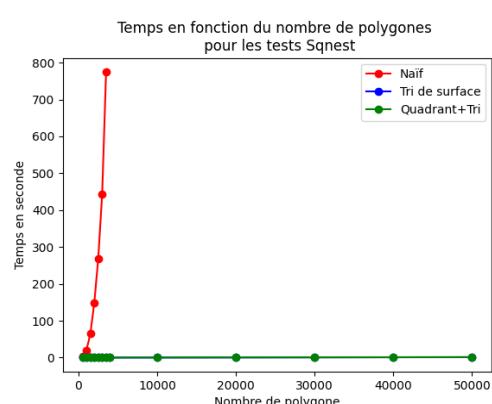


Figure 7 Temps Sqnest

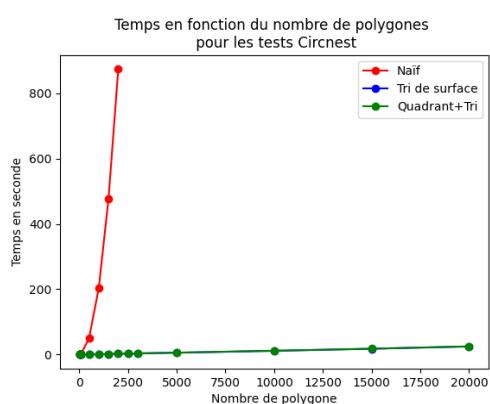


Figure 8 Temps Circnest

**Commentaires :**

On déduit des graphiques 3 et 4 que :

Il y a  $O(n^2)$  opérations dans l'algorithme naïf,  $O(n)$  opérations dans les deux algorithmes suivants.

Les graphiques du nombre d'opérations en fonction du polygone montrent que nous sommes bien dans le meilleur cas pour les deux derniers algorithmes ( $n$  tours de boucles). En effet, c'est logique. Car la configuration des polygones correspond exactement à la description du meilleur cas pour ces algorithmes.

Il y a le même nombre de tours de boucles dans les deux derniers algorithmes. Tandis qu'il y a  $n^2$  tours de boucles dans le premier.

Cela correspond à la théorie pour le nombre d'opérations.

**Complexité temporelle :**

Cela ne se voit pas vraiment dans les figures 7 et 8. Mais la courbe bleue et la courbe verte se superposent. Mais on observe une complexité temporelle un petit peu plus élevée dans le test circnest que dans le test sqnest. En effet, les carrés du test sqnest ont 4 points, alors que les cercles du test circnest ont 36 points. Ce qui fait que lors de l'appel à la fonction point\_in\_polygon. Il y a + d'opérations dans le test circnest que dans le test sqnest.

On voit donc que le nombre de points par polygone influe sur la complexité temporelle (mais cela ne change pas l'ordre de grandeur).

Donc, ici, l'algorithme (tri+quadrants) n'apporte aucune plus-value par rapport à l'algorithme de tri seulement. Ceci se voit car le nombre d'appels à la fonction point\_in\_polygon est le même pour les deux algorithmes (figures 9 et 10). Alors que la plus-value est censée se voir sur la réduction du nombre d'appels à cette fonction dans l'algorithme (tri+quadrants) par rapport à l'algorithme de tri. Or, ce n'est pas le cas ici.

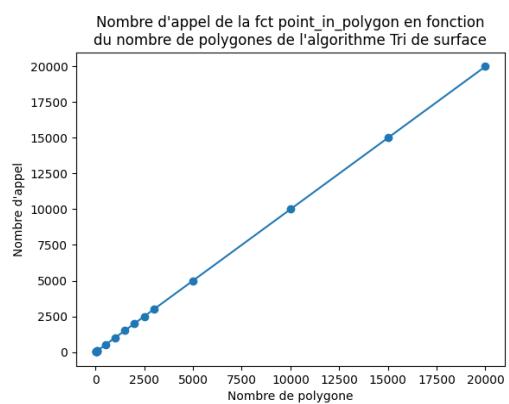


Figure 9 Nombre d'appel Tri surface (1)

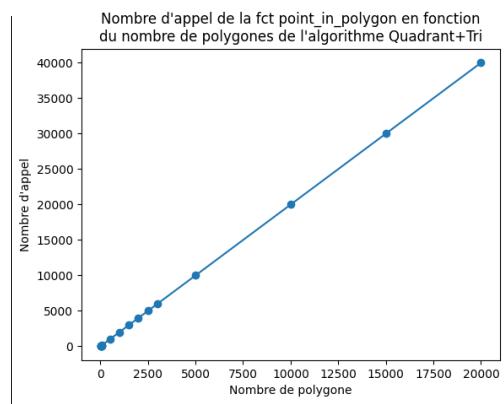
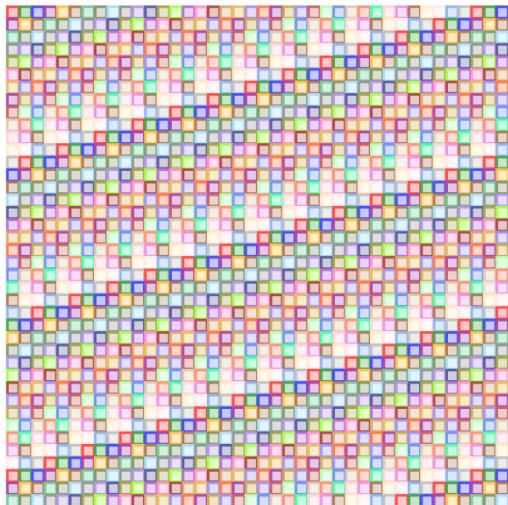


Figure 10 Nombre d'appel Quadrant+Tri (1)

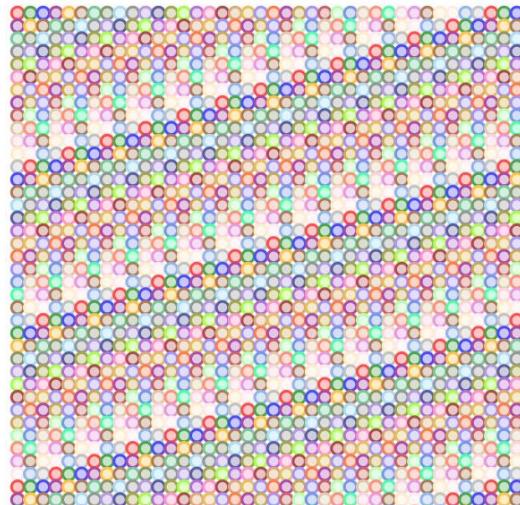
### Générateur circgrid/sqgrid/hexa:

#### Générateurs sqgrid/circline:

Le générateur sqgrid (respectivement circgrid) génère une grille de carrés (respectivement cercles). Ces polygones sont tous de même surface. Ainsi, aucun polygone n'est inclus dans l'autre.



1600 carrés générés avec sqgrid

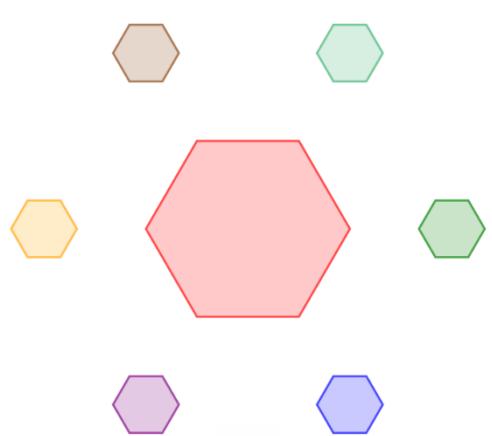


1600 cercles générés avec circgrid

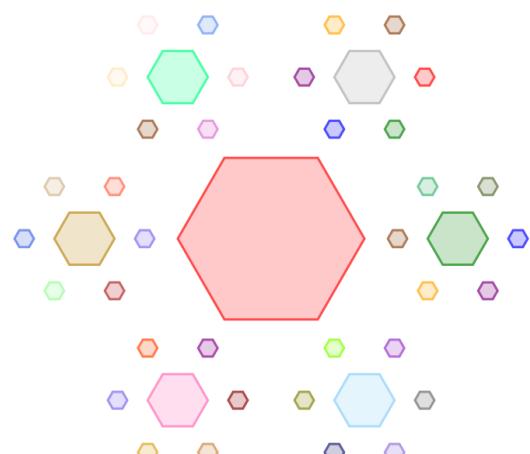
1600 carrés générés avec sqgrid

#### Générateur hexa:

Ce générateur génère une fractale d'hexagones indépendants.



Hexa récursivité niveau 2



Hexa récursivité niveau 3

A chaque niveau de récursivité, 6 petits hexagones sont créés autour de chaque petits polygones (les plus petits).

Le nombre de polygones dans ce générateur au niveau de récursivité  $n$  vaut  $1+6^n$ .

## Observations:

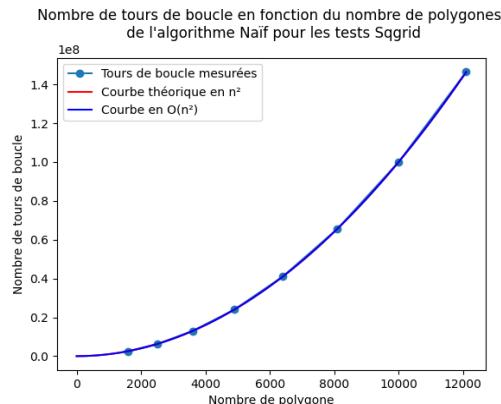


Figure 11 Naïf Sqgrid

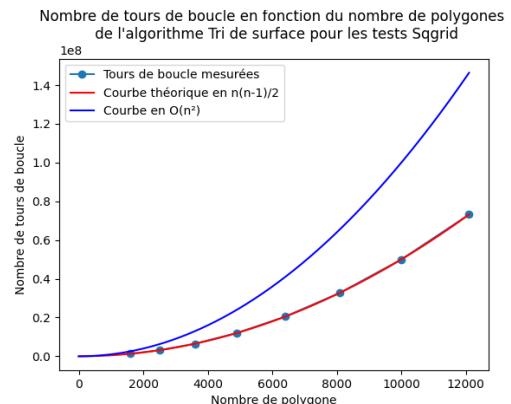


Figure 12 Tri de surface Sqgrid

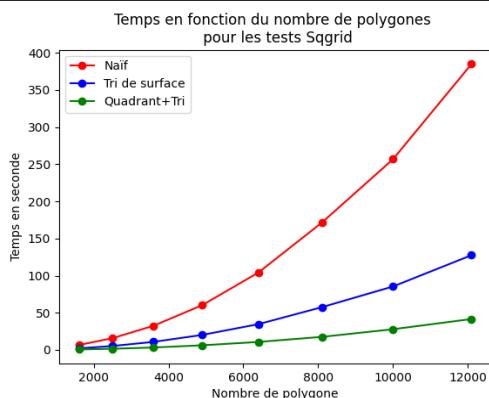


Figure 13 Temps Sqgrid

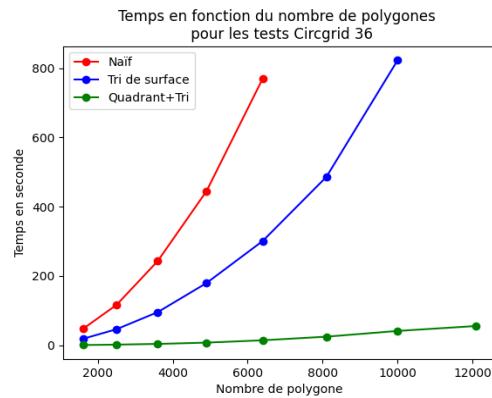


Figure 14 Temps Circgrid 36

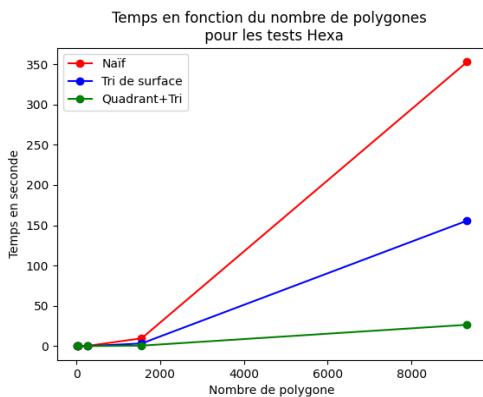


Figure 15 Temps Hexa

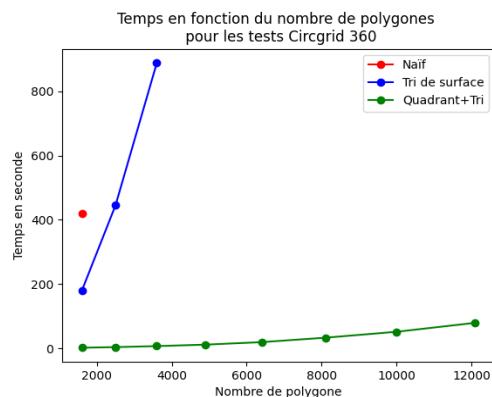
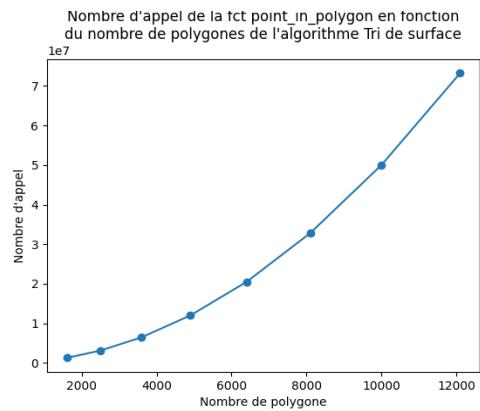
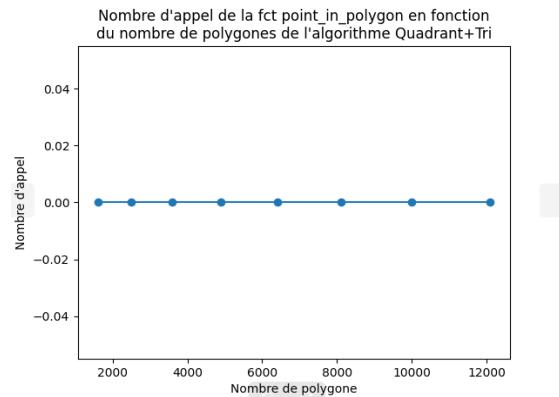


Figure 17 Temps Circgrid 360

Figure 17 Nombre d'appel *Tri de surface* (2)Figure 18 Nombre d'appel *Quadrant+Tri* (2)

### Commentaires :

Il y a exactement  $n^2$  tours de boucles pour le premier algo. Cependant, on voit qu'on a  $\frac{n*(n-1)}{2}$  tours de boucles (figure 12). Cela correspond au pire cas pour les algorithmes de tri et de tri+quadrants.

En effet, comme les polygones surface. L'arrangement par ordre croissant reste utile car il fait baisser le nombre de tours de boucles par rapport à l'algorithme naïf.

Comme les quadrants d'aucun polygones ne se touchent, la méthode qui rajoute les quadrants est utile ici. On la voit dans les graphiques sur la complexité temporelle.

### Complexité temporelle :

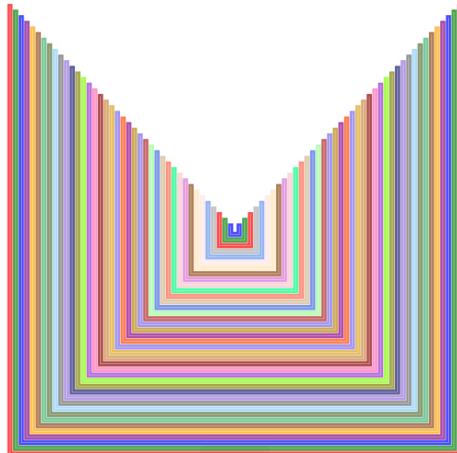
Les algorithmes tri et tri+quadrants sont nettement plus rapides que l'algorithme naïf encore une fois (figures 13 et 14). Mais on gagne beaucoup de temps dans l'algorithme (tri+quadrants) par rapport à l'algorithme de tri car on ne rentre jamais dans la fonction point\_in\_polygon (figures 17 et 18). On observe bien 0 appels de point\_in\_polygon pour l'algorithme tri+quadrants. Contrairement à un nombre d'appels en  $O(n^2)$  (exactement le nombre de tours de boucles, donc  $\frac{n*(n-1)}{2}$  pour l'algorithme de tri).

De plus, la différence entre les algorithmes de tri et tri+quadrants est d'autant plus marquée dans le test circgrid360 (360 points par cercles, figure 16) que dans le test circgrid36 (36 points par cercles, figure 14) que dans le test sqnest (4 points par carré, figure 13). Ceci est dû au coût de la fonction point\_in\_polygon, qui est croissant en fonction du nombre de points par polygones.

L'efficacité de l'algorithme tri+quadrants se manifeste très bien à travers ce type de générateurs.

### Générateur Fakesqnest :

Ce générateur génère des polygones en “U”. Ces polygones ne sont inclus dans aucun autre polygone, et c'est la différence avec le générateur sqnest. Le premier est le plus grand, tandis que le dernier est le plus petit.



500 polygones générés avec fakesqnest

### Observations:

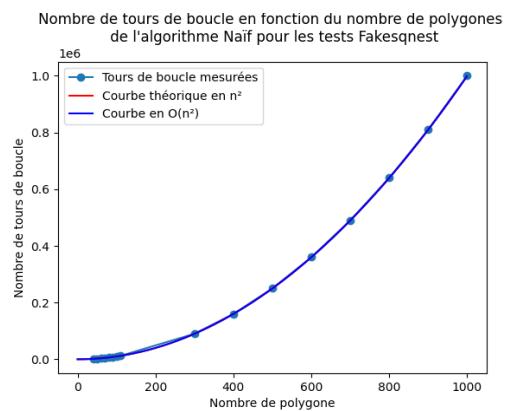


Figure 19 Naif Fakesqnest

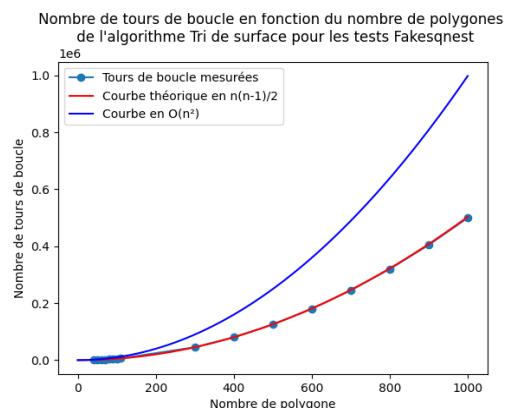


Figure 20 Tri de surface Fakesqnest

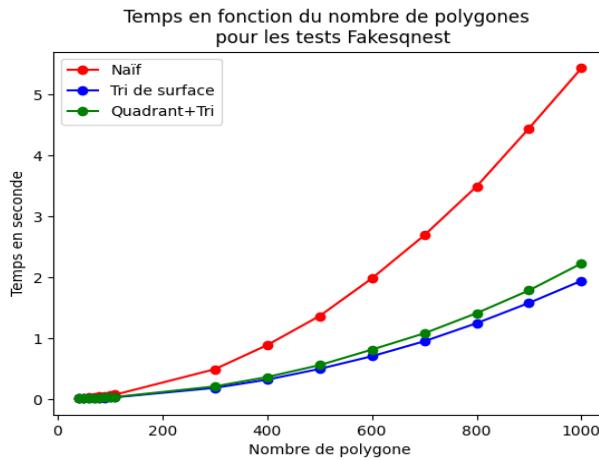


Figure 21 Temps Fakesqnest

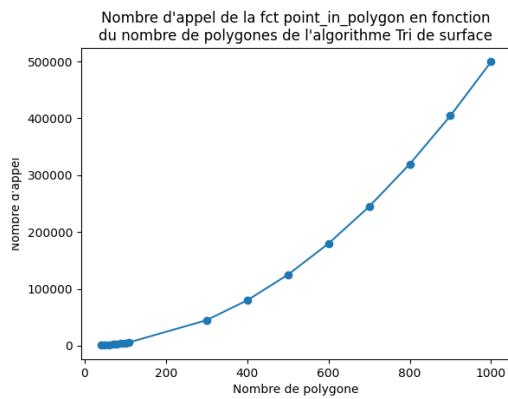


Figure 22 Nombre d'appel Tri de surface (3)

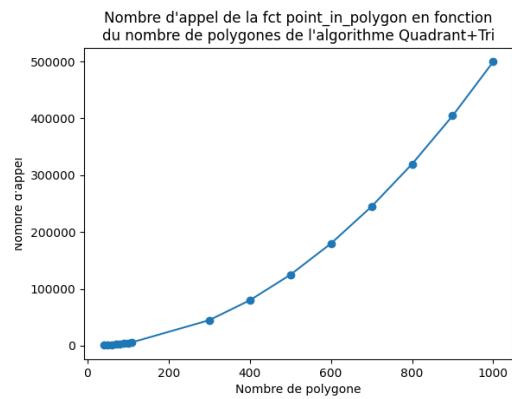


Figure 23 Nombre d'appel Quadrant+Tri

### Observations :

On observe (graphique 20) que les algorithmes tri et quadrant+tri effectuent  $n*(n-1)/2$  tours de boucles. Ce qui correspond au pire cas. En effet, on est bien dans le pire cas théorique car aucun polygone n'est inclus dans un autre.

De plus, on observe (figure 22 et 23) que le nombre d'appels à la fonction point\_in\_polygon est le même pour les algorithmes tri et tri+quadrant. Ceci s'explique par le fait que les quadrants de ces polygones sont imbriqués. Ainsi, l'algorithme tri+quadrant n'est pas plus efficace que l'algorithme de tri.

D'après le graphique 21, on voit même que l'algorithme de tri est un peu plus rapide que l'algorithme tri+quadrant. Ceci s'explique par le fait que dans l'algorithme quadrant+tri, du temps est perdu à créer des quadrants et à tester leurs intersections.

En conclusion, on est dans l'un des rares cas où l'algorithme de tri+quadrant est moins efficace (de peu) que l'algorithme de tri.

## Générateur Sierp/Sqfrac

### Générateur Sierp:

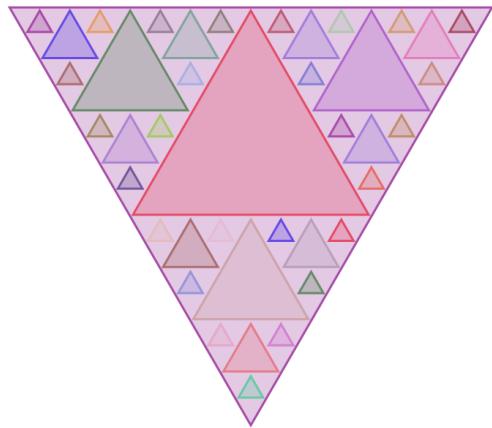
Ce générateur génère des triangles de Sierpinski, avec différents niveaux de récursivité.

Le premier niveau de récursivité est celui où il y a 1 polygone.

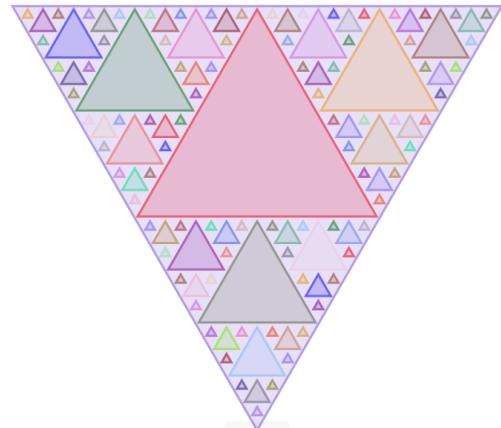
Pour le second, on multiplie par 3 le nombre de polygone et on ajoute 1, etc.

Au niveau de récursivité voulu, on ajoute 1 pour le triangle qui inclut tous les autres

Le test commence par le quatrième niveau de récursivité qui contient 41 polygones.



Sierp au niveau 4 de récursivité

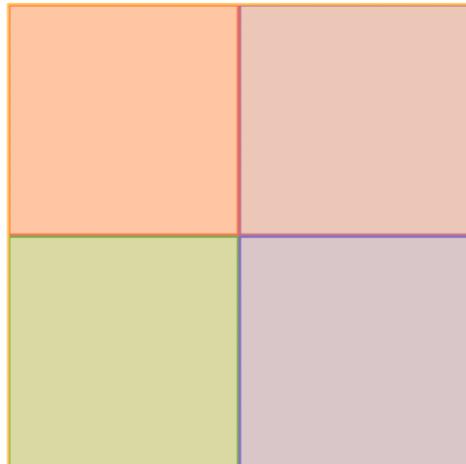


Sierp au niveau 5 de récursivité

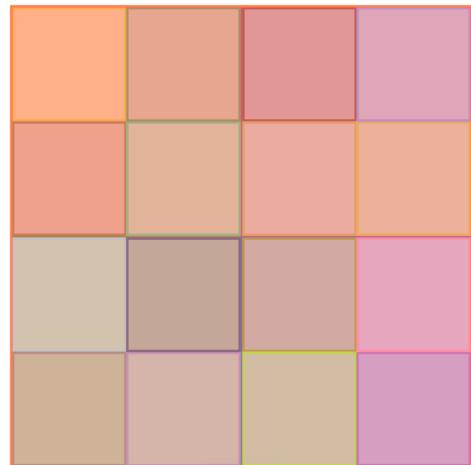
### Générateur Sqfrac:

Ce générateur génère des carrés avec différents niveau de récursivité.

Le premier contient un carré. Et pour le reste, le nombre de polygones au niveau de récursivité  $n$  vaut  $1 + 4^n$ .



Sqfrac au niveau 2 de récursivité



Sqfrac au niveau 3 de récursivité

Sqfrac au niveau 3 de récursivité

Dans ces deux générateurs, chaque polygone (à part le plus grand) est inclus dans exactement 1 polygone (le plus grand). Ces générateurs sont intéressants car on n'est pas dans le meilleur cas pour l'algorithme quadrant+tri, mais on est dans le pire cas non plus.

### Observations :

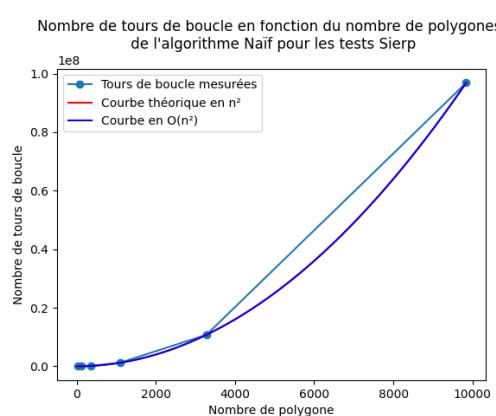


Figure 24 Naïf Sierp

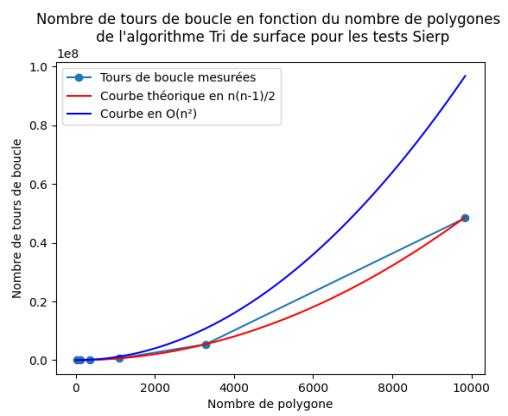


Figure 25 Tri de surface Sierp

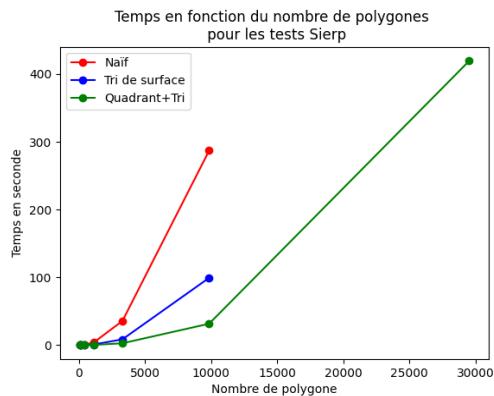


Figure 26 Temps Sierp

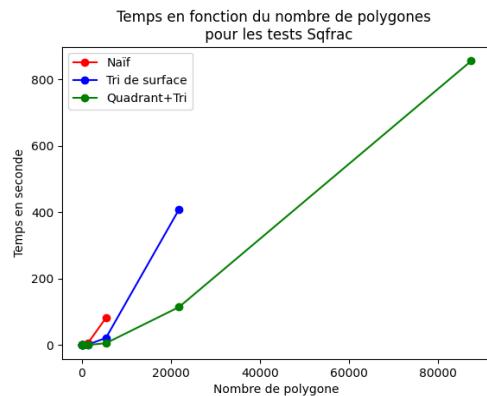


Figure 29 Temps Sqfrac

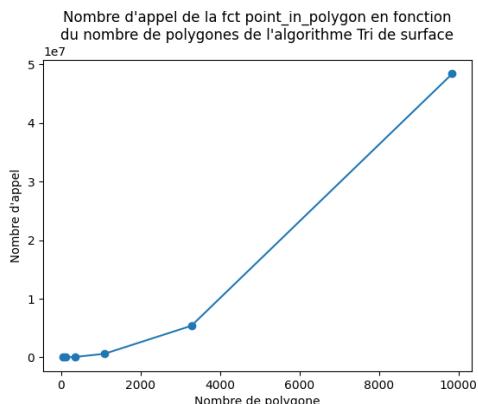


Figure 28 Nombre d'appel Tri de surface (4)

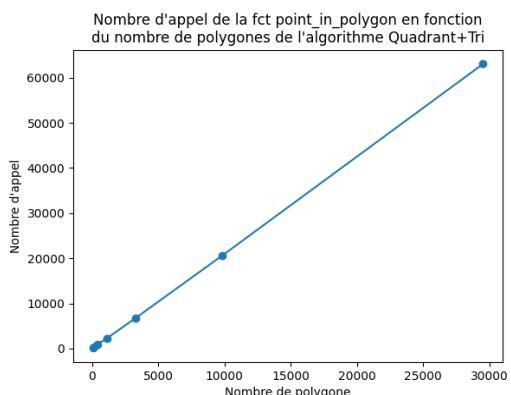


Figure 29 Nombre d'appel Quadrant+Tri

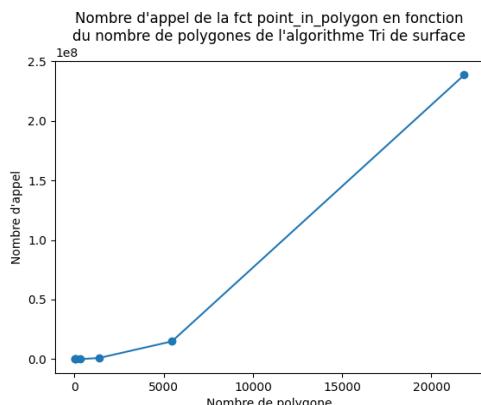


Figure 11 Nombre d'appel Tri de surface (5)

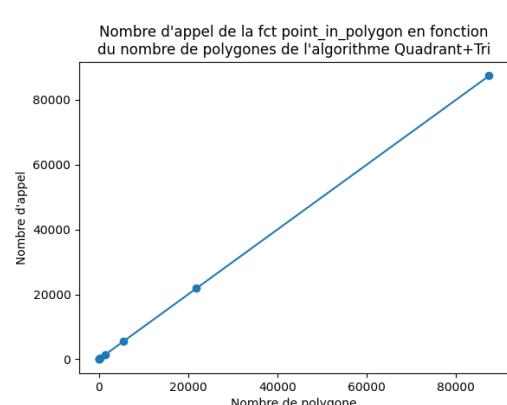


Figure 31 Nombre d'appel Quadrant+Tri (5)

## Commentaires

On observe que le nombre de tours de boucles pour l'algorithme naïf est de  $n^2$  (c'est toujours le cas). Cependant, le nombre de tours de boucles pour l'algorithme de tri est d'exactement  $\frac{n*(n-1)}{2}$  (figure 25).

Ce résultat est surprenant. Cependant, on voit que dans le fichier .poly des générateurs sierp et sqfrac que le plus grand polygone est toujours le dernier. Donc, à cause de la disposition des générateurs, on se retrouve quasiment dans le pire cas pour l'algorithme de tri de surface.

On voit (figures 29 et 31) que la fonction point\_in\_polygon est appelée de façon linéaire pour l'algorithme quadrant + tri. En effet, c'est logique. On voit dans les figures que les quadrants de chaque polygone à part le plus grand (qui n'est inclus dans aucun polygone) sont inclus exactement dans 1 polygone : le polygone le plus grand.

Dans les générateurs **Sierp**, Il y a environ 2n appels à la fonction point\_in\_polygon pour l'algorithme quadrant+tri (figure 29). En effet on voit dans la figure que le quadrant d'un polygone donné touche environ 2 à 3 quadrants d'autres polygones.

Dans les générateurs **Sqfrac**, il y a exactement n-1 appels à la fonction point\_in\_polygon pour l'algorithme quadrant+tri (figure 31). En effet, comme ici les polygones sont des carrés, les quadrants de ces polygones sont exactement ces polygones. Donc pour chaque polygone (à part le plus grand), comme il y a une seule inclusion (avec le plus grand). Il y a aussi une seule fois où le quadrant de ce polygone se touche avec le quadrant d'un autre polygone.

Dans les deux cas, le quadrant du polygone le plus grand touche les quadrants de tous les autres polygones. Mais comme c'est aussi le polygone de plus grande surface, on ne le teste jamais.

Ceci est à l'avantage de l'algorithme quadrant+tri qui profite du faible nombre d'intersection de quadrants entre polygones. Cela se voit dans le graphique sur la complexité temporelle (figures 25 et 26).

### III - Critique des générateurs:

Ces générateurs représentent bien les cas extrêmes de création de polygones. Cependant, aucun de ces générateurs n'a permis de créer des polygones de façon totalement aléatoire. Et c'est bien dommage ... Dans ce cas, on aurait pu avoir des résultats différents. L'algorithme le plus efficace serait l'algorithme quadrant+tri car il y aurait beaucoup de polygones qui n'ont pas d'intersection entre leurs quadrants et donc le rapport  $\frac{\text{nombre d'appels}}{n}$  (à la fonction point\_in\_polygon) serait petit devant n.

En conclusion, ces générateurs ont permis de valider les prévisions théoriques. De plus, avec toutes les informations générées par ces tests. Nous sommes en mesure de prévoir l'efficacité de ces algorithmes pour une génération de polygones totalement aléatoire. Donc ce n'est pas vraiment grave de ne pas avoir eu à disposition un générateur totalement aléatoire.

Au contraire, un générateur totalement aléatoire aurait pu nous fournir des résultats assez compliqués à interpréter. Et il aurait été difficile d'expérimenter un cas extrême pour un algorithme (pire cas ou meilleur cas) théorisé dans la partie I.

## IV - Synthèse :

Algorithme naïf	Algorithme tri	Algorithme tri+quadrant
Complexité en $O(n^2)$ dans tous les cas.  Donc peu efficace dans tous les cas.	Complexité variante entre $O(n \log(n))$ et $O(n^2)$ .  Beaucoup moins d'opérations que l'algorithme naïf. ( $n*(n-1)/2$ tours de boucles au maximum).  Complexité temporelle bien plus faible que l'algorithme naïf dans tous les cas.  Meilleur cas : polygones imbriqués. générés par ordre croissant de surface (ou décroissant).  Pire cas: polygones indépendants.	Même nombre de tours de boucles que l'algorithme tri.  En général, beaucoup moins d'appels à la fonction <code>point_in_polygon</code> (appel assez coûteux, surtout si le nombre de points par polygones est élevé).  Donc, en général, une complexité bien plus faible que l'algorithme tri.  Cependant, il y a certains cas où l'algorithme prend plus de temps que l'algorithme de tri. Mais ce sont de rares cas et le temps perdu est très faible. Ces cas sont lorsqu'on a des polygones imbriqués ou tels que leurs quadrants sont imbriqués.  Meilleur cas : polygones tels que leurs quadrants ne se touchent pas.  Pire cas: polygones imbriqués ou tels que leurs quadrants sont imbriqués

## V- Propositions d'algorithmes :

### 1) Algorithme tri décroissant+quadrant:

Cet algorithme (cf Annexe 4) trie la liste principale par ordre décroissant contrairement à l'algorithme tri+quadrant qui trie la liste par ordre croissant.

Nous avons voulu tester cet algorithme pour utiliser l'instruction pop.

Cette instruction permet de réduire la taille de la liste *surfaces\_triees* de la mémoire. Mais la question qu'on se pose est : réduire la complexité en mémoire a-t-il un impact sur la complexité temporelle ?

De plus, à travers les tests réalisés. On s'est aperçu qu'il y a souvent plusieurs polygones de même surface. On a donc rajouté deux instructions par rapport à l'algorithme.

La première, avant la première boucle for, est de retourner la liste résultat telle qu'elle (liste [-1]\*n) car on sait déjà que si tous les polygones ont la même surface, aucun polygone n'est inclus dans l'autre et donc il est inutile d'itérer.

La deuxième, est une condition rajoutée dans le if (ligne 64). Comme il y a plusieurs polygones qui peuvent être de même surface. Cette condition permet de passer directement au polygone j+1 lorsque les polygones i et j sont de même surface, car on sait déjà qu'il n'y aura pas inclusion.

Cet algorithme permet d'aller plus vite lorsqu'il y a plusieurs polygones de même surface. Et bien plus vite lorsque tous les polygones sont de même surface.

## 2) Algorithme new(tri+quadrant):

Cet algorithme (cf Annexe 5) est le même que le précédent. La seule différence réside dans le fait que le tri se fait par ordre croissant. L'instruction pop n'est donc plus utilisée. Ainsi, en comparant cet algorithme avec le précédent, nous pouvons directement voir l'influence de l'instruction pop sur les complexités temporelles.

Tout comme l'algorithme précédent, cet algorithme permet d'aller bien plus vite sur les fichiers où il y a des polygones de même surface.

### Observations :

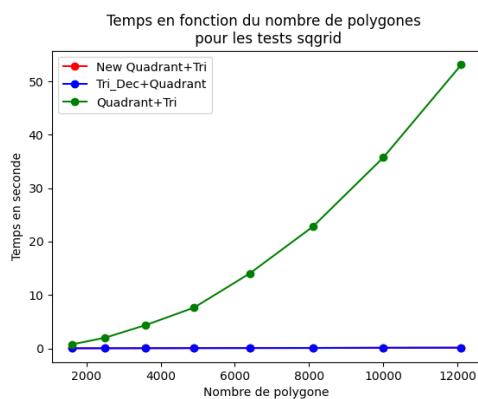


Figure 32 Temps Saarid (2)

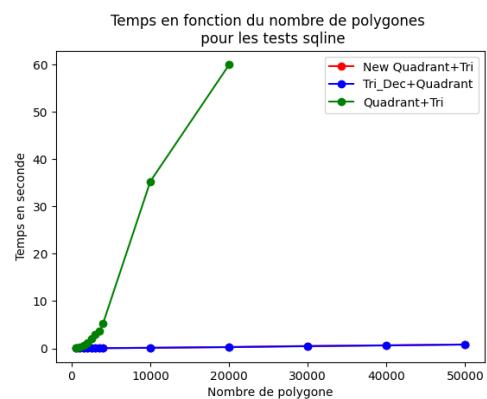


Figure 33 Temps Saline (2)

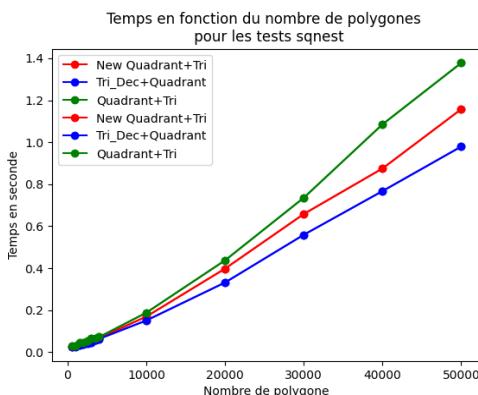


Figure 34 Temps Sqnest (2)

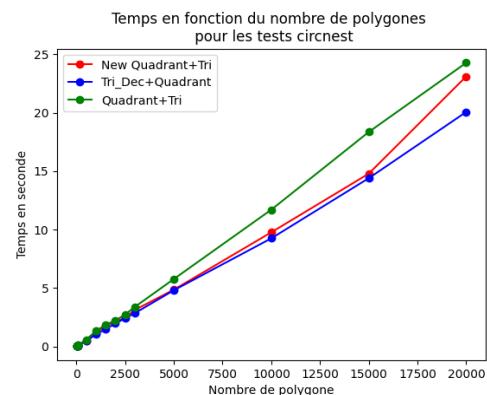


Figure 35 Temps Circnest (2)

**Commentaires :**

On note deux observations majeures :

- 1) On observe avec les figures 32 et 33 que les deux nouveaux algorithmes sont bien plus efficaces que l'algorithme quadrant+tri (les courbes bleue et rouge sont superposées). Ceci est dû au fait que les générateurs sqgrid et sqline génèrent des polygones de même taille. Ainsi, la fonction *detect\_inclusion* n'itère pas et renvoie directement la liste  $[-1] \times n$  comme ce qui était prévu.
  - 2) Les graphiques 34 et 35 montrent que l'algorithme utilisant l'instruction pop (tri\_dec+quadrant) est légèrement plus rapide que l'algorithme ne l'utilisant pas. Cependant, les courbes sont assez proches, il est donc difficile de conclure à la lecture de ce graphique. Mais après plusieurs essais, on observe toujours ce petit écart.
- L'instruction pop est en  $O(1)$ , ce qui explique les figures 34 et 35. De plus, on peut conclure que, bien que l'écart soit faible, le tri décroissant et l'instruction pop permettent d'être plus rapide (en plus d'être bénéfique pour la mémoire du PC).
  -

**Conclusion :**

Ces deux derniers algorithmes sont une amélioration de l'algorithme quadrant+tri que nous avons présenté dans la partie I. L'efficacité de ces deux algorithmes bonus se fait ressentir seulement pour certains types de tests (lorsqu'il y a plusieurs polygones de même surface, idéalement tous). En général, ils restent légèrement plus rapide que l'algorithme quadrant+tri et l'algorithme utilisant un tri décroissant est le plus rapide, et le moins coûteux en mémoire, ce qui en fait le meilleur compromis.

## **VI- Conclusion générale :**

Ce projet a été une exploration passionnante dans le domaine de l'informatique géométrique, où nous avons développé et évalué plusieurs algorithmes pour résoudre le problème de détection d'inclusion de polygones. Notre travail a impliqué la mise en œuvre d'algorithmes de base pour déterminer si un point est à l'intérieur d'un polygone, ainsi que la conception d'une méthode efficace pour déterminer les relations d'inclusion entre les polygones.

Pour visualiser et comprendre les performances de nos algorithmes, nous avons créé des graphes temporels, de nombre d'opérations, de nombre d'appels de fonction, etc. Ces graphiques ont été essentiels pour comparer les algorithmes et identifier les domaines où des améliorations peuvent être apportées.

Nous avons réalisé des tests expérimentaux approfondis en variant les générateurs de tests d'entrée pour évaluer les performances et les comportements des algorithmes dans différentes conditions. Ces tests nous ont permis de mieux comprendre les forces et les faiblesses de chaque algorithme dans des scénarios variés.

En parallèle, nous avons effectué des calculs théoriques de complexité pour évaluer la performance des algorithmes dans le pire des cas et comprendre leur comportement asymptotique. Cette analyse nous a permis de mieux comprendre les résultats des tests expérimentaux.

En somme, ce projet nous a permis d'explorer en profondeur le problème de détection d'inclusion de polygones, en mettant en œuvre une approche alliant tests expérimentaux, calculs théoriques de complexité et visualisation des performances. Les résultats obtenus constituent une contribution significative à notre compréhension de ce domaine.

## Annexe :

```

1 import sys
2 from geo.segment import Segment
3 from geo.point import Point
4 from tycat import read_instance
5
6
7 def point_in_polygon(point, polygon):
8     """
9         Check if a point is inside a polygon using the ray casting algorithm.
10    """
11    cpt = 0
12    x, y = point
13    segments = polygon.segments()
14    for segment in segments:
15        x1, y1 = segment.endpoints[0].coordinates
16        x2, y2 = segment.endpoints[1].coordinates
17
18        if min(y1, y2) < y <= max(y1, y2):
19            if x < min(x1, x2):
20                cpt += 1
21            else:
22                x_intersection = x1 + (y - y1) * (x2 - x1) / (y2 - y1)
23                if x < x_intersection:
24                    cpt += 1
25
26    return cpt % 2 == 1
27
28 def detect_inclusion(polygons):
29     parent_indices = [-1] * len(polygons)
30     for i, poly1 in enumerate(polygons):
31         polygones_inclus = []
32         for j, poly2 in enumerate(polygons):
33             if abs(poly1.area()) < abs(poly2.area()) and i != j and point_in_polygon(poly1.points[0].coordinates, poly2):
34                 polygones_inclus.append((j, abs(poly2.area())))
35         if polygones_inclus:
36             for k, tuples in enumerate(polygones_inclus):
37                 if tuples[1] == min([polygones_inclus[l][1] for l in range(len(polygones_inclus))]):
38                     parent_indices[i] = tuples[0]
39
40     return parent_indices
41
42 def main():
43     """
44         charge chaque fichier .poly donne
45         trouve les inclusions
46         affiche l'arbre en format texte
47     """
48     for fichier in sys.argv[1:]:
49         polygones = read_instance(fichier)
50         inclusions = detect_inclusion(polygones)
51         print(inclusions)
52
53 if __name__ == "__main__":
54     main()

```

Annexe 1 : Algorithme Naïf

```

3 from geo.segment import Segment
4 from geo.point import Point
5 from tycat import read_instance
6
7
8 def point_in_polygon(point, polygon):
9     """
10     Check if a point is inside a polygon using the ray casting algorithm.
11     """
12     cpt=0
13     x,y = point
14     segments=polygon.segments()
15     for segment in segments:
16         x1,y1= segment.endpoints[0].coordinates
17         x2,y2= segment.endpoints[1].coordinates
18
19         if min(y1,y2)<y<=max(y1,y2):
20             if x < min(x1, x2):
21                 cpt += 1
22             else:
23                 x_intersection = x1 + (y-y1)*(x2-x1)/(y2-y1)
24                 if x < x_intersection:
25                     cpt += 1
26
27     return cpt%2==1
28
29 def detect_inclusion(polygons):
30     compteur=0
31     surfaces=[(i,abs(poly.area())) for i,poly in enumerate(polygons)]
32     surfaces_triees=sorted(surfaces, key=lambda x:x[1])
33     parent_indices = [-1] * len(polygons)
34     for i,tuples in enumerate(surfaces_triees):
35         poly1=polygons[tuples[0]]
36         point= poly1.points[0].coordinates
37         for j in range(i+1,len(surfaces_triees)):
38             compteur +=1
39             poly2=polygons[surfaces_triees[j][0]]
40             if point_in_polygon(point, poly2):
41                 parent_indices[surfaces_triees[i][0]] = surfaces_triees[j][0]
42                 break
43
44 def main():
45     """
46     charge chaque fichier .poly donne
47     trouve les inclusions
48     affiche l'arbre en format texte
49     """
50     for fichier in sys.argv[1:]:
51         polygones = read_instance(fichier)
52         inclusions = detect_inclusion(polygones)
53         print(inclusions)
54
55 if __name__=="__main__":
56     main()

```

Annexe 2 : Algorithme Tri de surface

```

6 from geo.quadrant import Quadrant
7 from tycat import read_instance
8
9 def quadrants2(polygons): ##oui##
10    surfaces=[[i,abs(poly.area()),0] for i,poly in enumerate(polygons)]
11    for i,poly in enumerate(polygons):
12        q=poly.bounding_quadrant()
13        surfaces[i][2]=q
14
15    return surfaces
16
17 def point_in_polygon(point, polygon):
18    """
19        Check if a point is inside a polygon using the ray casting algorithm.
20    """
21    cpt=0
22    x,y = point
23    segments=polygon.segments()
24    for segment in segments:
25        x1,y1= segment.endpoints[0].coordinates
26        x2,y2= segment.endpoints[1].coordinates
27        if min(y1,y2)<y<=max(y1,y2):
28            if x < min(x1, x2):
29                cpt += 1
30            else:
31                x_intersection = x1 + (y-y1)*(x2-x1)/(y2-y1)
32                if x < x_intersection:
33                    cpt += 1
34    return cpt%2==1
35
36 def detect_inclusion(polygons):
37    parent_indices = [-1] * len(polygons)
38    list_surfaces_quadrants=quadrants2(polygons)
39    surfaces_triees=sorted(list_surfaces_quadrants, key=lambda x:x[1])
40    for i,tuples in enumerate(surfaces_triees):
41        poly1=polygons[tuples[0]]
42        quadrant1=tuples[2]
43        point=poly1.points[0].coordinates
44        for j in range(i+1,len(surfaces_triees)):
45            poly2=polygons[surfaces_triees[j][0]]
46            quadrant2=surfaces_triees[j][2]
47            if quadrant1.intersect(quadrant2) and point_in_polygon(point, poly2):
48                parent_indices[surfaces_triees[i][0]] = surfaces_triees[j][0]
49                break
50    return parent_indices
51
52 def main():
53    """
54        charge chaque fichier .poly donne
55        trouve les inclusions
56        affiche l'arbre en format texte
57    """
58    for fichier in sys.argv[1:]:
59        polygones = read_instance(fichier)

```

*Annexe 3 : Algorithme Quadrant+Tri*

```

9  def quadrants(polygons):
10     return surfaces
11
12
13 tabnine: test | explain | document | ask
14 def point_in_polygon(point, polygon):
15     """
16     Check if a point is inside a polygon using the ray casting algorithm.
17     """
18     cpt=0
19     x,y = point
20     segments=polygon.segments()
21     for segment in segments:
22         x1,y1= segment.endpoints[0].coordinates
23         x2,y2= segment.endpoints[1].coordinates
24         if min(y1,y2)<y<=max(y1,y2):
25             if x < min(x1, x2):
26                 cpt += 1
27             else:
28                 x_intersection = x1 + (y-y1)*(x2-x1)/(y2-y1)
29                 if x < x_intersection:
30                     cpt += 1
31
32     return cpt%2==1
33
34 tabnine: test | fix | explain | document | ask
35 def detect_inclusion(polygons):
36     parent_indices = [-1] * len(polygons)
37     list_surfaces_quadrants=quadrants(polygons)
38     surfaces_triees=sorted(list_surfaces_quadrants, key=lambda x:x[1], reverse=True)
39     n=len(surfaces_triees)
40     if min(surfaces_triees[i][1] for i in range(n))==max(surfaces_triees[i][1] for i in range(n)):
41         return parent_indices
42     for i in range(n-1,0,-1):
43         tuples=surfaces_triees.pop()
44         poly1=polygons[tuples[0]]
45         quadrant1=tuples[2]
46         point=poly1.points[0].coordinates
47         surface_1=tuples[1]
48         for j in range(0,i):
49             tuples_2=surfaces_triees[i-1-j]
50             poly2=polygons[tuples_2[0]]
51             quadrant2=tuples_2[2]
52             surface_2=tuples_2[1]
53             if surface_1<surface_2 and quadrant1.intersect(quadrant2) and point_in_polygon(point, poly2):
54                 parent_indices[tuples[0]] = tuples_2[0]
55                 break
56     return parent_indices
57
58 tabnine: test | explain | document | ask
59

```

Annexe 4 : Algorithme tri décroissant+quadrant

```

tabnine:test|explain|document|ask
def point_in_polygon(point, polygon):
    """
    Check if a point is inside a polygon using the ray casting algorithm.
    """
    cpt=0
    x,y = point
    segments=polygon.segments()
    for segment in segments:
        x1,y1= segment.endpoints[0].coordinates
        x2,y2= segment.endpoints[1].coordinates
        if min(y1,y2)<y<=max(y1,y2):
            if x < min(x1, x2):
                cpt += 1
            else:
                x_intersection = x1 + (y-y1)*(x2-x1)/(y2-y1)
                if x < x_intersection:
                    cpt += 1
    return cpt%2==1

tabnine:test|explain|document|ask
def detect_inclusion(polygons):
    parent_indices = [-1] * len(polygons)
    list_surfaces_quadrants=quadrants(polygons)
    surfaces_triees=sorted(list_surfaces_quadrants, key=lambda x:x[1])
    n=len(surfaces_triees)
    if min(surfaces_triees[i][1] for i in range(n))==max(surfaces_triees[i][1] for i in range(n)):
        return parent_indices
    for i,tuples in enumerate(surfaces_triees):
        poly1=polygons[tuples[0]]
        quadrant1=tuples[2]
        point=poly1.points[0].coordinates
        surface1=tuples[1]
        for j in range(i+1,len(surfaces_triees)):
            poly2=polygons[surfaces_triees[j][0]]
            quadrant2=surfaces_triees[j][2]
            surface2=surfaces_triees[j][1]
            if surface1<surface2 and quadrant1.intersect(quadrant2) and point_in_polygon(point, poly2):
                parent_indices[surfaces_triees[i][0]] = surfaces_triees[j][0]
                break
    return parent_indices

tabnine:test|explain|document|ask
def main():
    """
    charge chaque fichier .poly donne
    trouve les inclusions
    affiche les resultats
    """

```

Annexe 5 : Algorithme new(tri+quadrant)

Annexe 5 : Algorithme new(tri+quadrant)

