Projet Algo 2024

Antonio Mattar et Jayson Marest

Mars 2024

Sommaire

1 Introduction

L'objectif principal de ces algorithmes est de détecter des inclusions, où l'on cherche à déterminer si un polygone est inclus à l'intérieur d'un autre polygone. Cette problématique peut être abordée avec différentes approches algorithmiques, chacune présentant ses propres avantages et limitations.

Parmi les techniques les plus utilisées, on trouve des algorithmes basés sur le principe du "ray-casting", des approches basées sur le tri de polygones par aires décroissantes voire la triangulation des polygones. Chacune de ces méthodes offre des solutions pour résoudre le problème de détection d'inclusions dans des contextes spécifiques, en fonction des caractéristiques des polygones considérés et des contraintes de performance.

Dans cette revue, nous explorerons en détail plusieurs de ces algorithmes de détection de polygones dans les polygones, en examinant leurs principes de fonctionnement, leurs avantages, leurs limitations et leurs domaines d'application privilégiés. Nous mettrons également en évidence les défis et les opportunités associés à l'utilisation de ces algorithmes dans des applications réelles, ainsi que les perspectives d'amélioration et de développement futur de ces techniques.

2 Identification des inclusions de points dans polygone

On remarque que pour appliquer le ray-casting il faut que l'on teste l'intersection d'une ligne horizontale avec des segments (les cotés des polygones) et donc pour cela il faut que notre point (x, y) soit bien compris entre les y_{max}, y_{min} et les x_{max}, x_{min}.

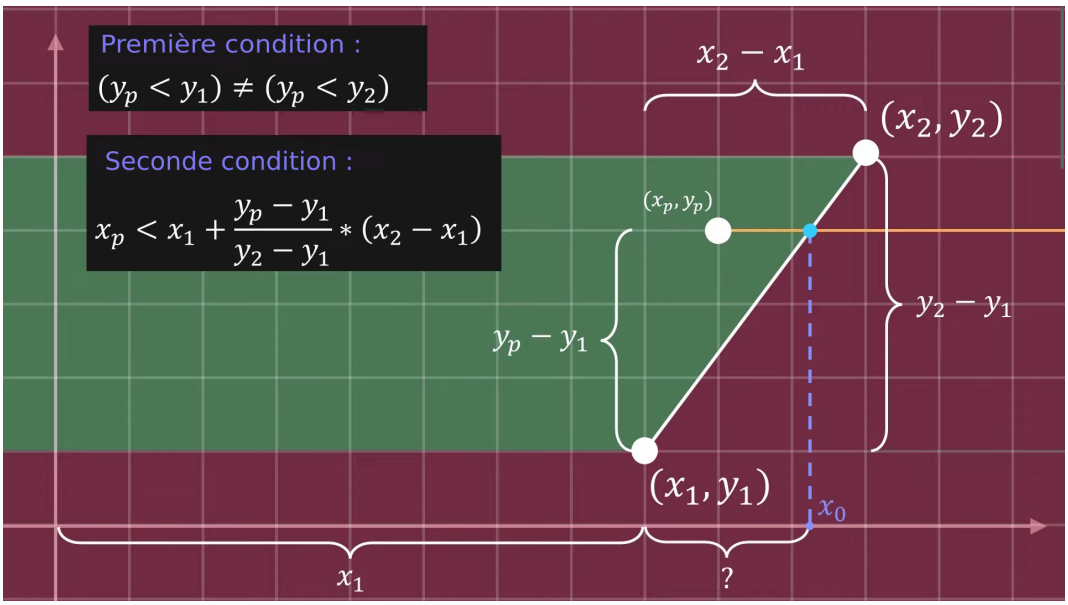


FIGURE 1 – Illustration de l'algorithme de détection d'intersection.
https ://www.youtube.com/watch ?v=w4Dosp2U74Y

Après il nous restera à compter le nombre d'intersection :

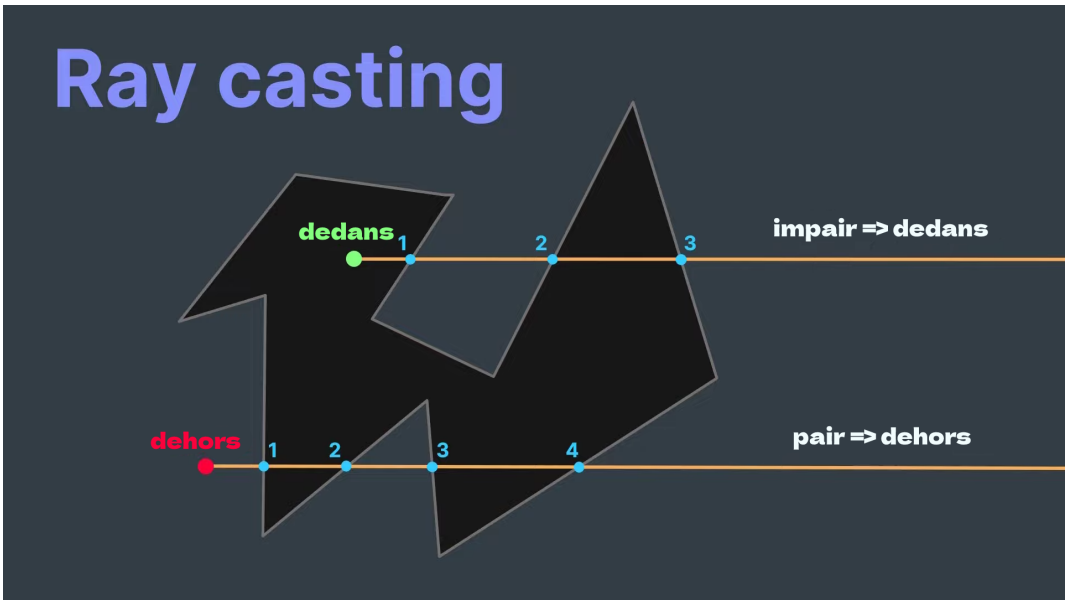


FIGURE 2 – Illustration de l'algorithme de détection d'intersection.
https ://www.youtube.com/watch ?v=w4Dosp2U74Y

Pourquoi cet algorithme ?

La première chose importante à noter dans notre cadre est que l'on peut se limiter à tester l'inclusion d'un seul point dans un polygone pour valider l'inclusion du polygone entière, et cela est dû au fait qu'il n'y a pas d'intersection entre les polygones.

Théorème 1. *Soit P_1 et P_2 deux polygones qui ne s'intersectent pas. Si un point de P_1 est inclus dans P_2, alors P_1 est inclus dans P_2.*

Démonstration. Soit P_1 et P_2 deux polygones qui ne s'intersectent pas. Supposons l'existence de M un point de P_1 inclus dans P_2. Par l'absurde supposons que P_1 n'est pas inclus dans P_2. Alors il existe un point M' non inclus dans P_2. Cela voudrait dire que nécessairement le segment MM' intersecterait P_2 et donc P_1 et P_2 s'intersecteraient, ce qui est absurde. □

Notre implémentation de cet algorithme est disponible dans le fichier `ray_casting.py`.

3 Calcul des aires des polygones

Par la suite, il sera très (très) utile de calculer les aires des polygones. Déjà, car nous allons utiliser par la suite des algorithmes basés sur les aires, mais également pour déterminer le plus petit père.

Le génial module `geo` qui nous est fourni contient déjà une méthode de classe pour les calculs d'aires de polygones : `area`. Naturellement, nous l'avons testé sur quelques polygones. Notre premier réflexe, a été de voir s'il n'y avait pas des méthodes, peut-être, moins couteuses.

Nous avons donc écrit, dans notre module `aire_polygone.py`, une implémentation de la formule des lacets, aussi connu comme formule des aires de Gauss dans d'autres langues. Cette formule a le

bon goût de pouvoir être implémenté en temps linéaire pour le nombre de points dans le polygône.

Bien sûr, la fonction utilisée par le module `geo` est aussi en temps linéaire car il s'agit aussi d'une implémentation de la formule des lacets. Cependant, après analyse, cette fonction utilise deux boucles distinctes sur le nombre de sommets, une pour créer les différents éléments à ajouter, puis une boucle avec la fonction native de Python `sum`. Notre implémentation fait la somme et le parcours des points simultanément.

Au final, notre implémentation présente des petits avantages de temps dont nous ne nous passerons pas pour la suite, chaque petite optimisation est bonne à prendre.

4 Génération de tests

Initialement, nous nous sommes basés sur des générateurs disponibles sur Internet. Ceux-là nous permettaient de nous donner une idée de la vitesse de nos algorithmes.

Nous avons cependant préféré laisser notre imagination être source de notre propre générateur de polygones. Celui-ci est disponible dans le fichier Python `gen_test.py`.

Notre générateur de tests crée une variété de polygones adaptés à nos algorithmes d'inclusion.
Il offre la possibilité de générer :

— des carrés
— des carrés imbriqués
— des carrés alignés en série ou combinant ces deux propriétés.
— des fractales de carrés
— des polygones aléatoirement générés imbriqués
— des polygones (3 à 5 cotés max) réguliers imbriqués

Les polygones imbriqués sont intentionnellement générés pour tester spécifiquement l'algorithme basé sur les aires, qui devrait être plus efficace face à ces structures par rapport à une autre entrée uniforme.

Concernant les polygones, deux approches distinctes sont employées : la première repose sur un maillage de points où trois points aléatoires forment un triangle, puis les points du maillage inclus dans ce triangle sont réexaminés jusqu'à atteindre la profondeur souhaitée ou jusqu'à ce que le nombre de points restants soit inférieur à trois. La seconde méthode utilise la génération de polygones à partir d'un cercle unité, en choisissant n sommets sur son périmètre. Ce processus est répété avec un rayon égal au rayon du cercle inscrit du polygone précédent.

En outre, une série de polygones aléatoires, créés manuellement, est intégrée au générateur. Ces polygones sont ensuite copiés et déplacés sur un plan étendu pour se donner un nombre élevé de polygones.

Finalement, tous les résultats de nos tests sur nos différents générateurs avec nos différents algorithmes sont présents dans le fichier `mesures`.

5 Algorithmes naïfs

Avant toute chose, notons que nous utiliserons tout le long de ce compte-rendu une liste `liste_inclusions`, renvoyé par l'algorithme et qui contient l'ensemble des indices des plus proches parents des polygones fournis dans la liste `polygones`.

Les algorithmes ci-dessous seront présents dans le fichier `naif.py`.

5.1 Algorithme naïf basique

5.1.1 Idée

L'idée est assez simple : pour chaque polygone, on vérifie s'il est dans un autre polygone. Et donc on teste toutes les inclusions possibles. Enfin, on vérifie quel parent de chaque polygone est d'aire minimale pour déterminer quel est son parent direct.

5.1.2 Description de l'algorithme

Algorithme 1: naïf

```
Données: Liste polygones
Résultat: Liste contenant pour l'indice de chaque polygone son plus proche parent
début
    liste_inclusions ⟵ ∅ ;
    dico_inclusions ⟵ ∅ ;
    pour polygone_1 dans polygones faire
        pour polygone_2 dans polygones \{polygone_1} faire
            si le premier point de polygone_1 est dans polygone_2 alors
                ajouter polygone_2 dans le dico_inclusions de polygone_1 ;
            fin
        fin
    fin
    pour polygone dans polygones faire
        trouver le plus petit parent de polygone et le mettre dans liste_inclusions ;
    fin
    retourner liste_inclusions
fin
```

5.1.3 Choix d'implémentations

La liste des inclusions est simplement la liste souhaitée en résultat, imposé par l'énoncé. Une table de hachage est utilisée pour stocker les relations parent-enfant entre les polygones. Pour chaque polygone, la table de hachage enregistre les indices des polygones dans lesquels il est inclus. Après un récent TD, je me rends compte qu'il aurait été beaucoup plus malin d'utiliser une liste de listes étant donné que l'on a besoin seulement des indices des polygones qui auraient servis d'indice de la liste...

5.1.4 Analyse de l'algorithme

La première implémentation a pour objectif d'être extrêmement naïve, par exemple, on calcule l'aire d'un polygone à chaque fois quand on compare lequel est le plus petit, donc si une aire est déjà calculée, on ne la prend pas en compte.

De plus il serait peut-être judicieux de mettre en place quelques vérifications préalables avant d'effectivement vérifier l'inclusion d'un polygone dans l'autre, en effet, c'est assez coûteux de vérifier l'inclusion d'un polygone dans un autre, il faudrait mettre en place un test très peu coûteux pour éviter de perdre du temps

Théoriquement, cet algorithme possède un coût en $\mathcal{O}(m^2 n)$ où m est le nombre de polygones dans le fichier et n est le nombre moyen de sommets dans un polygone. En effet, on fait m tour de boucles puis encore une fois m tour de boucles avec une opération en moyenne sur n sommets. (test d'inclusion en temps linéaire en le nombre de sommets).

Soumettons alors cet algorithme aux différents ensembles de polygones issus de notre générateur.

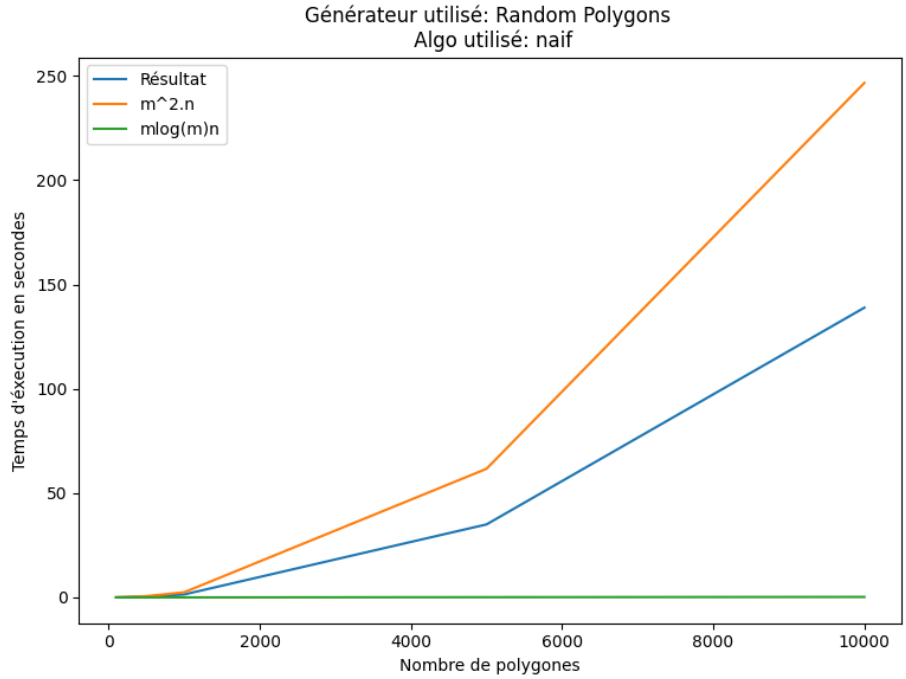


FIGURE 3 – Algorithme naïf sur des polygones aléatoires

5.1.5 Résultats

Un bon point de comparaison à notre échelle se trouve dans les high score du Git. Cet algorithme, notre premier, a passé le test 0 en 5040ms. Il ne passe pas le test 1.

5.1.6 Solutions

La deuxième implémentation de l'algorithme naïf présent dans le programme `naif.py` du compte-rendu est la même que la première à l'exception que l'on ajoute une liste qui garde en mémoire les aires. Nous nous épargnons un paragraphe complet sur cet algorithme puisque les gains de temps sont assez légers mais notons qu'ils sont présents, comme attendu.

Il reste à mettre en place une pré-condition simple.

5.2 Algorithme naïf à test de quadrant

5.2.1 Idée

L'idée reste la même que le premier algorithme. Seulement, on essaie de mettre en place une pré-condition très rapide à vérifier comme vu précédemment.

Notre attention s'est portée sur les quadrants fournis par le super module `geo`.

Définition 1 (Quadrant)**.** *Soit P un polygone. Le quadrant de P est le plus petit carré (en terme d'aire) qui contient P.*

Théorème 2 (Propriété simple sur les quadrants)**.** *Pour un polygone P, et un point M, si M n'est pas inclus dans le quadrant de P, alors M n'est pas inclus dans P.*

Preuve. Soit P un polygone et M un point. Par contraposée, si M est inclus dans P alors M est inclus dans le quadrant de P car il contient P. □

L'intérêt est donc de vérifier au préalable si l'on est dans le quadrant avant de vérifier si l'on est dans le polygone. On gagne un temps considérable, si l'on considère qu'il n'y a que peu de polygones dans lequel on est imbriqué en moyenne.

5.2.2 Description de l'algorithme

Algorithme 2: naïf quadrant

```
Données: Liste polygones
Résultat: Liste contenant pour l'indice de chaque polygone son plus proche parent
début
    liste_inclusions ⟵ ∅ ;
    dico_inclusions ⟵ ∅ ;
    liste_quadrants ⟵ {quadrants des polygones} ;
    pour polygone_1 dans polygones faire
        pour polygone_2 dans polygones \{polygone_1} faire
            si le premier point de polygone_1 est dans le quadrant de polygone_2 alors
                si le premier point de polygone_1 est dans polygone_2 alors
                    ajouter polygone_2 dans le dico_inclusions de polygone_1 ;
                fin
            fin
        fin
    fin
    pour polygone dans polygones faire
        trouver le plus petit parent de polygone et le mettre dans liste_inclusions ;
    fin
    retourner liste_inclusions
fin
```

5.2.3 Choix d'implémentations

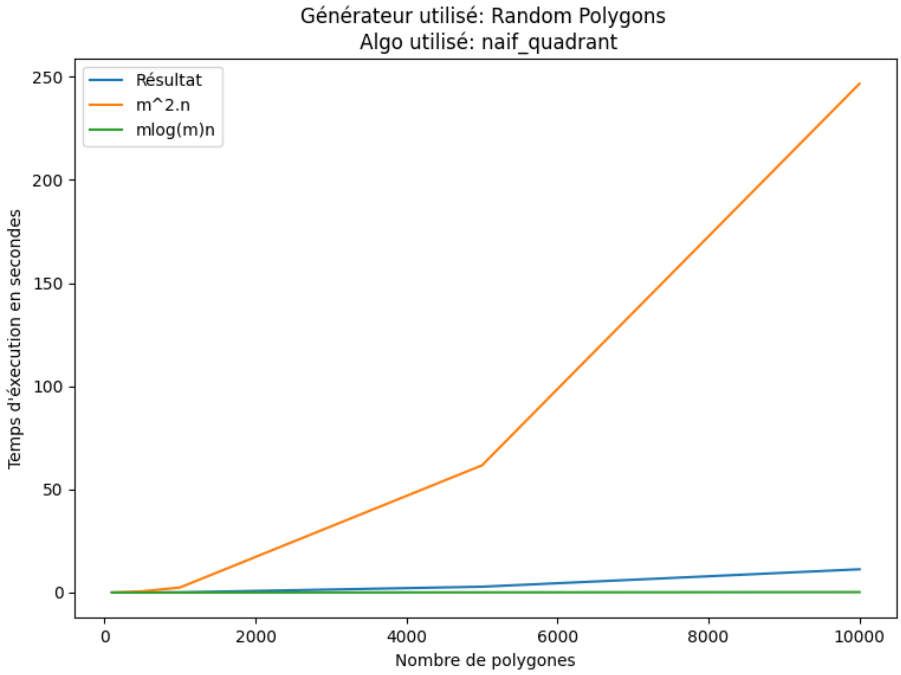
Tous les choix restent identiques au dernier algorithme. Seulement, on ajoute une liste de quadrants de polygones, en utilisant la méthode de stocker le quadrant du polygone i à l'indice i. On utilise bien le type `list` de Python.

5.2.4 Analyse de l'algorithme

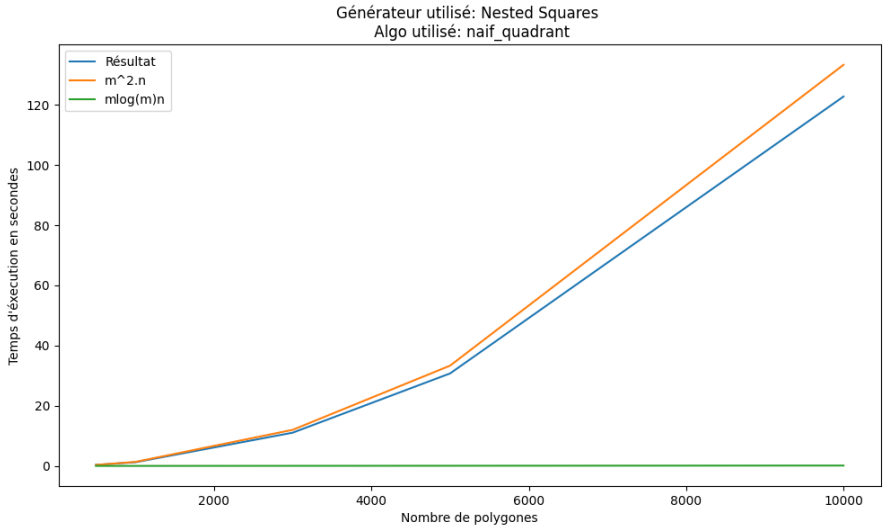
Bien sûr, cet algorithme présente une faiblesse, s'il y a énormément de "nids" de polygones (c'est-à-dire un nombre colossal de polygones imbriqués les uns dans les autres), la pré-condition ne sert quasiment à rien et on tendra à perdre plus de temps qu'autre chose.

Au contraire on gagne un temps considérable dans le cas contraire.

On peut alors tracer la complexité de l'algorithme de manière empirique à l'aide de notre générateur, la complexité étant assez difficile à calculer selon le cas.



(a) Vue générale sur des polygones aléatoires



(b) Vue sur les polygones imbriqués

FIGURE 4 – 2 cas différents

5.2.5 Résultats

Cet algorithme, notre avant-dernier, a passé le test 0 en 3040ms et le test 1 en 33.042ms. Il ne passe pas le test 2. Comme prévu, en le voyant sur le graphique "Nested Squares", s'il y'a beaucoup de nids, on se rapproche de l'algorithme naïf basique, voire pire.

5.3 Conclusion

Il est clair qu'il est possible d'aller beaucoup plus loin que des simples algorithmes naïfs, en établissant des pré-conditions plus robustes, ou des techniques pour éviter de visiter des polygones de manières inutiles. L'objectif à ce stade est de passer le test 2.

6 Algorithmes de tri par aire

Les algorithmes présents ci-dessous sont présents dans le fichier `tri_aire.py`.

6.1 Problèmes

Un problème majeur reste que, malgré les pré-conditions, l'on visite un bon nombre de polygones que l'on pourrait déterminer comme inéligible en tant que parent. C'est là qu'interviennent les aires.

Théorème 3 (Simple propriété sur les aires des polygones)**.** *Soit P_1 et P_2 deux polygones. Si l'aire de P_2 est plus petite que celle de P_1, alors P_1 ne peut être inclus dans P_2.*

Démonstration. Peut se montrer avec la formule des lacets, assez théorique alors que c'est plutôt clair. □

L'idée que l'on peut avoir consiste à analyser les polygones éligibles seulement, donc ceux d'aires inférieures. De plus, puisque l'on cherche uniquement à connaître le plus petit parent, on peut s'arrêter dès que l'on trouve une inclusion dans un polygone qu'on sait de plus petite aire possible.

6.2 Algorithme de tri par aire décroissante basique

6.2.1 Idée

L'idée est de trier la liste `polygones` par aire décroissante. On prend le polygone le plus à droite de la liste (d'aire la plus petite qui n'est pas évalué). Celui-ci ne peut-être inclus que dans les éléments plus à gauche que lui. On balaye de droite vers la gauche et on s'arrête dès que l'on trouve une inclusion car on sait que ce sera son parent le plus petit.

6.2.2 Description de l'algorithme

Algorithme 3: tri par aire décroissante

```
Données: Liste polygones
Résultat: Liste contenant pour l'indice de chaque polygone son plus proche parent
début
    trier polygones par aire décroissante ;
    liste_inclusions ⟵ ∅ ;
    pour polygone_1 dernier polygone dans polygones faire
        supprimer polygone_1 de polygones ;
        pour polygone_2 dans polygones à partir de la droite faire
            si le premier point de polygone_1 est dans polygone_2 alors
                définir polygone_2 comme parent de polygone_1 dans liste_inclusions ;
                sortir de cette boucle ;
            fin
        fin
    fin
fin
```

6.2.3 Choix d'implémentations

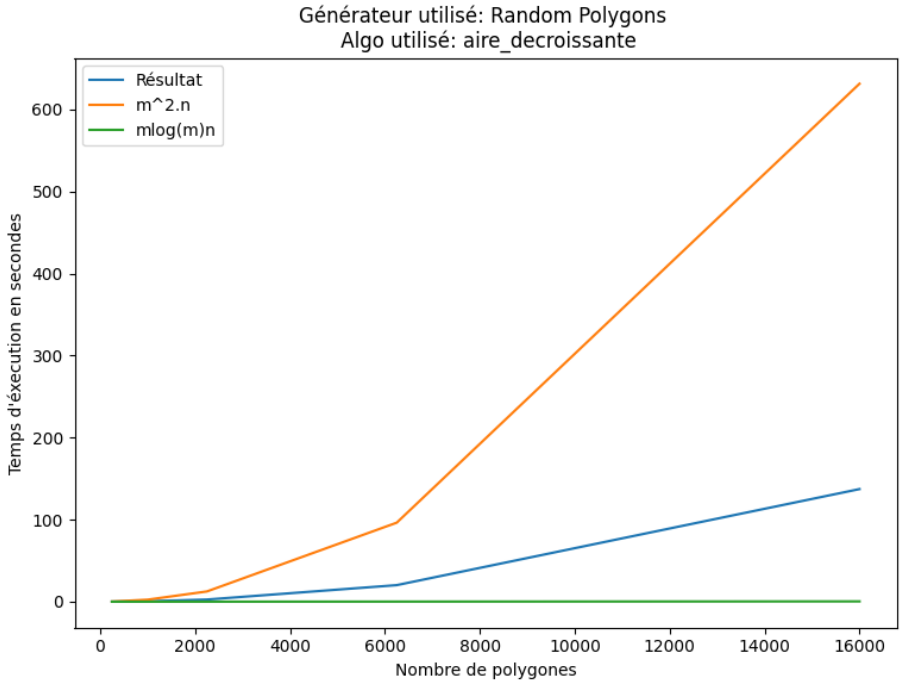
La liste des inclusions reste conforme aux demandes de l'énoncé. La liste des `polygones` est ici implémentée comme listes de tuples contenant les polygones en tant que tels, l'aire et l'indice des polygones, il est nécessaire de garder les indices car après le tri, on les perd.

Remarque : on utilise un tri par aire décroissante afin d'utiliser la méthode `pop` de Python, qui permet d'obtenir le dernier élément de la liste en temps constant et de le supprimer afin de gagner au fur et à mesure de la place en mémoire.

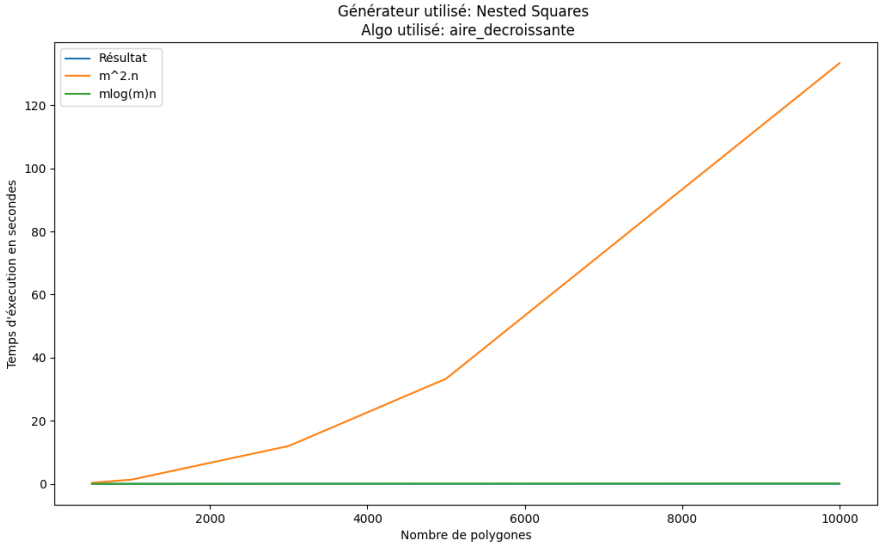
6.2.4 Analyse de l'algorithme

La complexité de l'algorithme est assez dur à calculer encore une fois, tout dépend de ce que l'on met en entrée. Par exemple, cet algorithme est très très efficace pour les ensemble de polygones avec énormément d'imbrications. Si jamais il n'y en a presque pas, on est équivalent au naïf.

Utilisons nos générateurs pour déterminer empiriquement la complexité de l'algorithme.



(a) Algorithme de tri par aire décroissante sur des polygones aléatoires



(b) Algorithme de tri par aire décroissante sur des polygones imbriqués

FIGURE 5 – 2 cas différents

6.2.5 Résultats

Cet algorithme, notre deuxième, passe le test 0 en 4040ms, il ne passe pas le test 1. Ce n'est certes pas très visible, mais pour les nids, on est confondu avec la courbe des opérations en $\mathcal{O}(mlog(m)n)$.

6.3 Tri par aire décroissante fusion

6.3.1 Idée

Si jamais nous avons plein de polygones de même aire, il y a un problème, cela ne permet pas d'éliminer des polygones que l'on aurait facilement pu éviter de ne serait-ce que de regarder, comme le veut l'idée du tri.

Une solution est de considérer tous les éléments de même aire comme un unique élément. La seule différence avec l'algorithme précédent est alors de trier puis de "fusionner" les éléments de la liste

6.3.2 Description de l'algorithme

L'algorithme ne change pas en soi, ce qui change est la manière dont on implémente la liste des polygones.

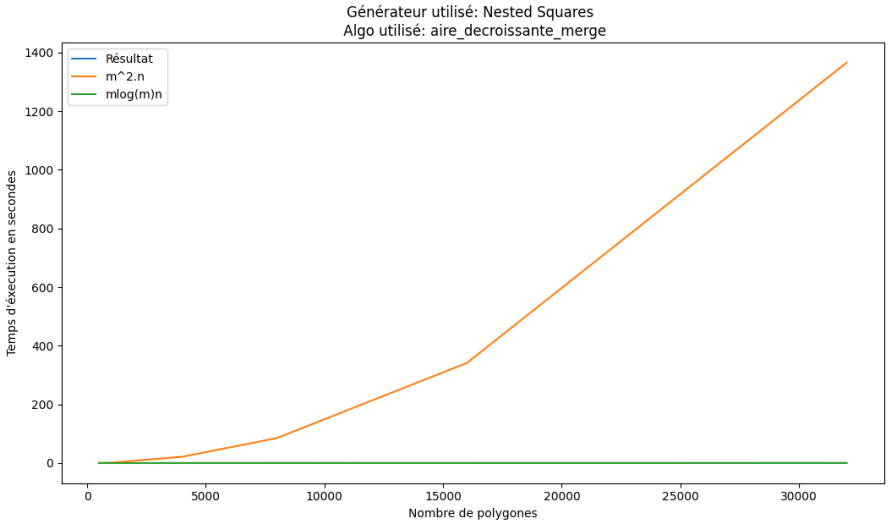
6.3.3 Choix d'implémentations

La liste `polygones` est ici implémentée comme listes de couple (aire, liste), la liste contenant tous les polygones de même aire.

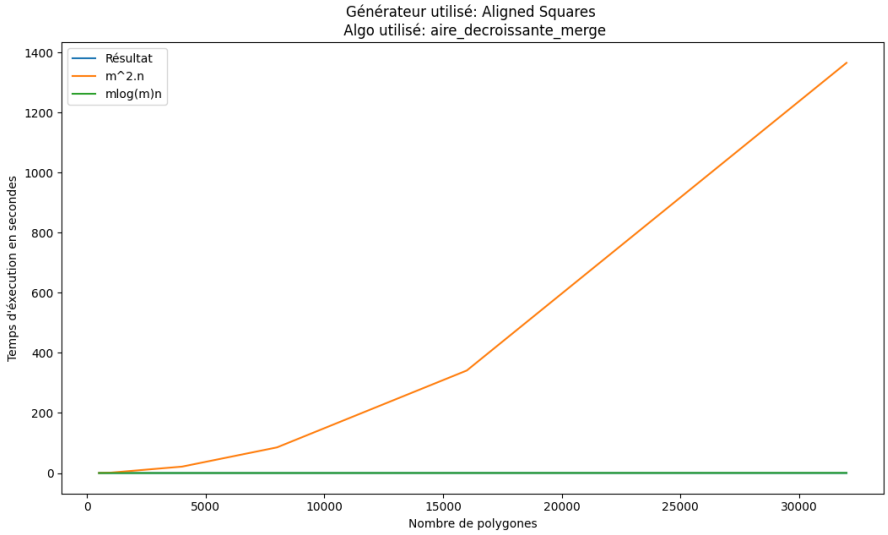
6.3.4 Analyse de l'algorithme

La complexité de l'algorithme est toujours difficile à déterminer.

Utilisons nos générateurs pour déterminer empiriquement la complexité de l'algorithme.



(a) Carrés imbriqués



(b) Carrés alignés

FIGURE 6 – 2 cas différentes, même efficacité

6.3.5 Résultats

Cet algorithme, notre deuxième, passe le test 0 en 4040ms, il ne passe pas le test 1. Ici, on gère autant les nids que les carrés de même aires alignés.

6.4 Tri par aire croissante et arbre

6.4.1 Idée

L'idée est de créer un arbre dont la relation est définie ainsi : si un polygone est fils d'un autre, alors c'est son parent le plus petit. Les fils de la racine n'ont pas de parent.

Ainsi, lorsque l'on trie par ordre décroissant par aire les polygones, on prend un polygone. On regarde au premier étage si l'on est inclus dans l'un des polygones. Si ce n'est pas le cas on le met à cet étage, sans parent, sinon, on fait le même processus récursivement avec le polygone dans lequel on est inclus. On s'arrête dès qu'on est inclus.

6.4.2 Description de l'algorithme

Algorithme 4: tri par aire croissante et arbre

Données: Liste `polygones`
Résultat: Liste contenant pour l'indice de chaque polygone son plus proche parent
début
  trier `polygones` par aire croissante ;
  `liste_inclusions` $\longleftarrow \emptyset$;
  `arbre_polygone` $\longleftarrow \emptyset$;
  pour polygone_1 **dernier polygone dans** `polygones` **faire**
    supprimer polygone_1 de `polygones` ;
    mettre polygone_1 dans sa place dans `arbre_polygone` et mettre à jour `liste_inclusions` ;
  fin
fin

6.4.3 Choix d'implémentation

Les arbres sont implémentées comme une classe `arbre` qui contient comme paramètres racine et feuille, racine est un indice de polygone (éventuellement -1) et feuille une liste d'arbres. Le reste ne change pas.

6.4.4 Analyse de l'algorithme

Comme précédemment, il est très compliqué de prévoir une complexité théorique précise pour cet algorithme car tout dépend de l'entrée. Son point faible est encore une fois les ensembles de polygones où tous les polygones sont disjoints, car on visitera tous les polygones à chaque fois pour vérifier.

L'implémentation comprend déjà l'idée de vérifier l'inclusion dans un quadrant pour gagner un peu de temps.

6.4.5 Résultats

Cet algorithme passe le test 0 en 3040ms, il ne passe pas le test 1.

6.5 Algorithme de tri par aire décroissante avec quadrant

6.5.1 Idée

L'idée est de simplement combiner l'algorithme naïf qui utilise des quadrants et l'algorithme de tri par aire décroissante.

6.5.2 Description de l'algorithme

Algorithme 5: tri par aire décroissante avec quadrant

```
Données: Liste polygones
Résultat: Liste contenant pour l'indice de chaque polygone son plus proche parent
début
    trier polygones par aire décroissante ;
    liste_inclusions ⟵ ∅ ;
    pour polygone_1 dernier polygone dans polygones faire
        supprimer polygone_1 de polygones ;
        pour polygone_2 dans polygones à partir de la droite faire
            si le premier point de polygone_1 est dans le quadrant de polygone_2 alors
                si le premier point de polygone_1 est dans polygone_2 alors
                    définir polygone_2 comme parent de polygone_1 dans liste_inclusions ;
                    sortir de cette boucle ;
                fin
            fin
        fin
    fin
fin
```

6.5.3 Choix d'implémentations

Même chose que pour l'algorithme 3.

6.5.4 Analyse de l'algorithme

Encore et toujours, on fait un calcul de complexité empirique, car cela dépend toujours des entrées.

Utilisons nos générateurs pour déterminer empiriquement la complexité de l'algorithme.

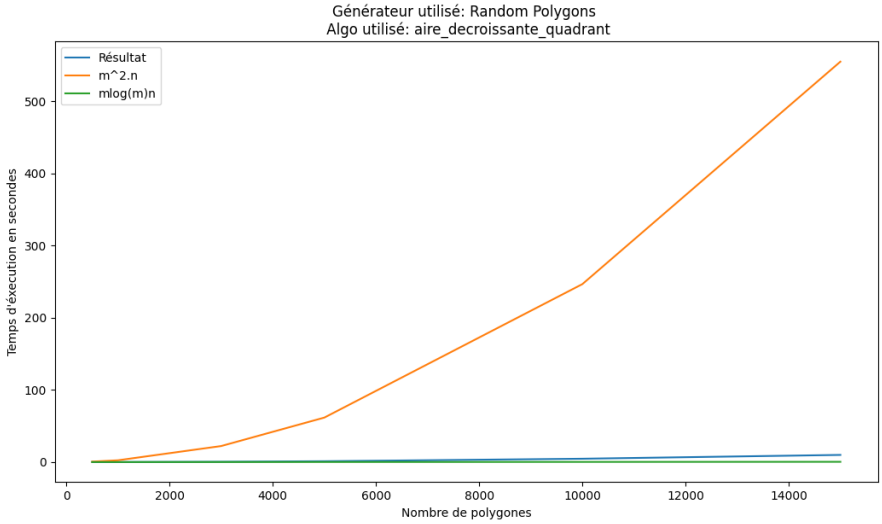


FIGURE 7 – Algorithme de tri par aire décroissante avec quadrant sur des polygones aléatoires

6.5.5 Résultats

Cet algorithme, notre dernier, passe le test 0 en 3040ms, le test 1 en 17.041ms et le test 2 en 7041ms. Hourra !

Nous avons bien conscience que ces résultats ne rentrent pas en compte pour l'évaluation mais il reste satisfaisant de voir que l'on traverse les épreuves.

7 Idées d'algorithmes non concluantes

7.1 Algorithme par triangulation

7.1.1 Motivation

Le TD sur la triangulation nous a donné envie d'essayer une nouvelle méthode. En effectuant des recherches sur Internet, nous sommes tombés sur des résultats fortement intéressants.

Définition 2 (Ligne monotone)**.** *Une ligne est dite monotone si l'ensemble des points possède des valeurs d'ordonnées monotones.*

Définition 3 (Polygone monotone)**.** *Un polygone est dit monotone s'il est la réunion de deux lignes monotones.*

Théorème 4. *Tout polygone quelconque peut être décomposé en polygones monotones. Si le polygone a n sommets, il existe un algorithme qui réalise cette action avec une complexité temporelle de* $\mathcal{O}(nlog(n))$.

Théorème 5. *Il existe un algorithme qui triangule n'importe quel polygone monotone avec une complexité de* $\mathcal{O}(n)$.

Démonstration. Voir Références 2 & 4. □

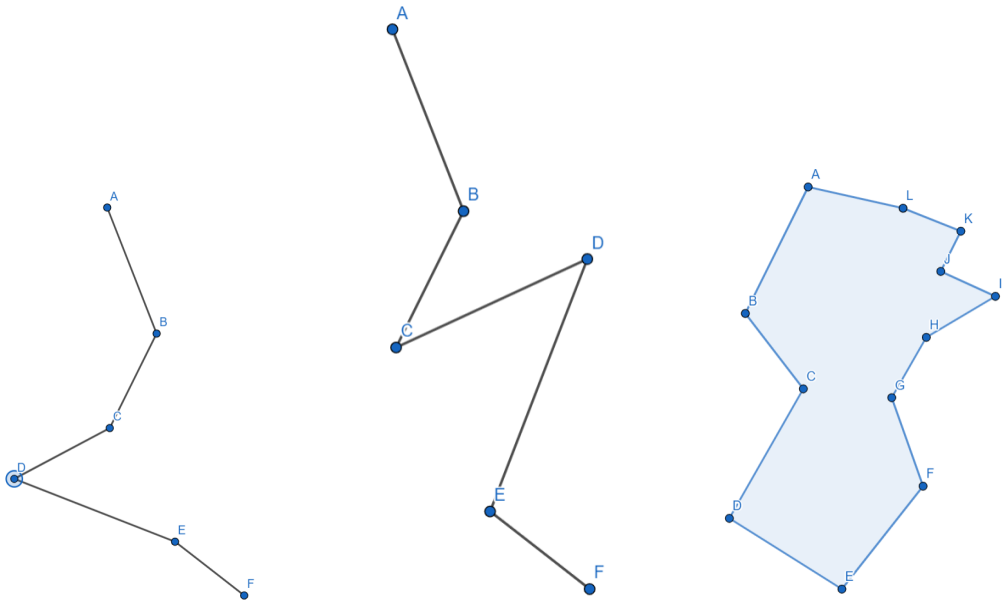


FIGURE 8 – De la gauche vers la droite : une ligne monotone, une ligne non monotone et un polygone monotone

Nous avons essayé de chercher des algorithmes à implémenter qui respectaient ces conditions. Mais ceux-ci étaient assez difficiles à conceptualiser, nous avions envie d'implémenter des choses que nous étions sûrs de comprendre.

Nous avions aussi essayé de nous tourner vers une méthode de triangulation dite de Delaunay.

Théorème 6. *L'algorithme de triangulation de Delaunay a une complexité temporelle en MOYENNE de* $\mathcal{O}(nlog(n))$.

Ici aussi, l'implémentation était trop compliquée. Nous avons essayé l'implémentation offerte par le module `scipy`.

7.1.2 Description de l'algorithme

Algorithme 6: triangulation

```
Données: Liste polygones
Résultat: Liste contenant pour l'indice de chaque polygone son plus proche parent
début
    liste_inclusions ⟵ ∅ ;
    dico_inclusions ⟵ ∅ ;
    liste_triangles ⟵ triangulation des polygones dans polygones ;
    pour polygone_1 dans polygones faire
        pour triangle dans liste_triangles faire
            si le premier point de polygone_1 est dans triangle alors
                ajouter le polygone correspondant au triangle dans le dico_inclusions de
                 polygone_1 ;
            fin
        fin
    fin
    pour polygone dans polygones faire
        trouver le plus petit parent de polygone et le mettre dans liste_inclusions ;
    fin
    retourner liste_inclusions
fin
```

7.1.3 Abandon et raison de l'abandon

La version fournie dans le fichier `algorithmes_non_concluants.py` ne fonctionne pas. Elle est présente pour montrer une idée de l'implémentation.

Le module `scipy` prenait un temps considérable à s'importer (1 seconde), il fallait aussi importer `numpy`, qui prenait aussi 1 seconde à s'importer. Déjà, on perdait énormément de temps.

Alors il a été préférable de s'arrêter là, car nous n'écrivions même pas nous-même l'algorithme de triangulation, ce qui était trop long et complexe.

7.2 Algorithme sans utiliser la détection de point

7.2.1 Motivation

L'idée est d'éviter de calculer des aires. Aussi, on essaie d'éviter d'utiliser l'algorithme de détection d'inclusion. L'idée est pour chaque premier point de chaque polygone, on stocke la hauteur, on stocke les abscisses des points sur segments des polygones qui passent à cette hauteur.

Prenons, un point quelconque, ainsi, si le nombre de points à la même hauteur à gauche de notre premier point est pair, on est hors d'un polygone, sinon on est dedans, et le plus petit parent est le polygone que l'on a traversé un nombre impair de fois.

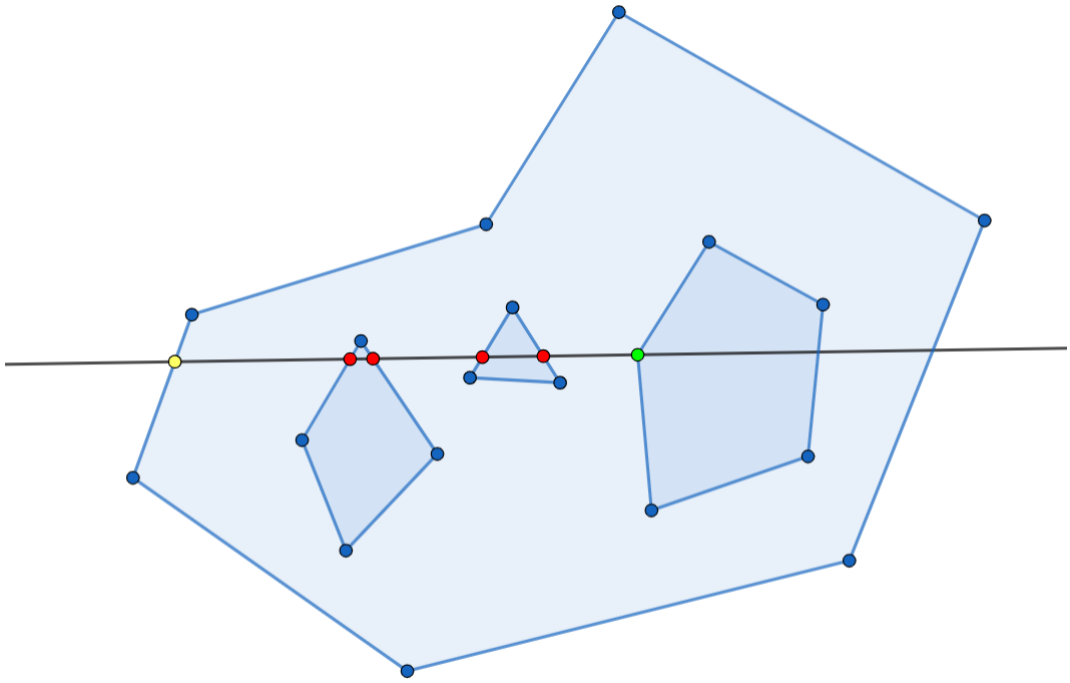


FIGURE 9 – En vert, le point étudié. En jaune le point sur le segment du polygone traversé un nombre impair de fois, en rouge les points des segments sur des polygones traversés un nombre pair de fois

7.2.2 Description de l'algorithme

Algorithme 7: sans détection

Données: Liste `polygones`
Résultat: Liste contenant pour l'indice de chaque polygone son plus proche parent
début
  `liste_inclusions` ⟵ ∅ ;
  `abscisses_hauteurs` ⟵ tous les points des segments pour chaque hauteur de chaque premier point de chaque polygone;
  pour polygone **dans** `polygones` **faire**
    (x,y) ⟵ premier point de polygone ;
    si *le nombre de points à gauche de x dans* `abscisses_hauteurs` *à la hauteur y est impair* **alors**
      mettre dans `liste_inclusions` le polygone qui apparaît un nombre impair de fois;
    sinon
      mettre -1 dans `liste_inclusions`;
    fin
  fin
fin

7.2.3 Abandon et raison de l'abandon

La version fournie dans le fichier `algorithmes_non_concluants.py` ne fonctionne pas non plus. Elle est présente pour montrer une vague idée de l'implémentation.

Plus on avançait dans l'implémentation, plus on se disait que la complexité temporelle serait trop élevée pour que l'algorithme soit intéressant.

Un manque de motivation est notamment à l'origine de l'abandon, nous voulions largement nous concentrer sur d'autres algorithmes, qui semblaient beaucoup plus efficaces, ainsi que sur les générateurs et les tracés de graphiques pour la complexité.

Chaque fois, il nous venait une nouvelle idée d'algorithme, qui était plus intéressante que celle-ci, il semblait plus judicieux de se concentrer sur ces idées plutôt que sur cet algorithme.

7.3 Algorithme en utilisant un quadrillage

7.3.1 Motivation

L'utilisation d'un quadrillage pour résoudre des problèmes géométriques peut simplifier les calculs et permettre une approche systématique de la résolution.

7.3.2 Description de l'algorithme

Dans cet algorithme, chaque point de l'espace est associé à une cellule de la grille. Une stratégie clé pour accélérer la détection d'inclusions de polygones consiste à utiliser un dictionnaire où chaque clé est un point et chaque valeur est la liste des polygones dans lesquels ce point est inclus. En construisant ce dictionnaire une fois pour tout l'ensemble de polygones, il est possible d'accéder rapidement aux polygones contenant un point donné.

Lorsqu'un point doit être testé pour son inclusion dans un polygone, l'algorithme commence par déterminer la cellule de la grille dans laquelle se trouve ce point. Ensuite, il consulte le dictionnaire pour obtenir la liste des polygones contenant ce point, en se limitant aux polygones des cellules actuelle et voisines. Cette approche réduit considérablement le nombre de polygones à considérer pour le test d'inclusion, ce qui peut entraîner une amélioration significative des performances de l'algorithme.

En utilisant ce dictionnaire, l'algorithme peut accélérer la détection d'inclusions de polygones en évitant de parcourir tous les polygones de l'ensemble pour chaque point testé. Au lieu de cela, il se concentre uniquement sur les polygones pertinents pour le point en question, ce qui permet une exécution plus rapide et plus efficace de l'algorithme.

7.3.3 Abandon et raison de l'abandon

L'algorithme utilisant un quadrillage a été abandonné principalement en raison de contraintes de temps et de difficultés rencontrées dans la mise en place du quadrillage pour chaque polygone. Malgré l'idée prometteuse d'utiliser un dictionnaire pour accélérer la détection d'inclusions de polygones, la complexité de la création de quadrillages distincts pour chaque polygone a posé des défis majeurs. En effet, il était difficile d'assurer que les mêmes points se retrouvent dans les mêmes cellules du quadrillage pour tous les polygones.

En raison de ces problèmes techniques et de contraintes de temps, il a été décidé d'abandonner cette approche au profit d'autres méthodes plus simples et plus fiables pour la détection d'inclusions de polygones.

8 Conclusion

Nous sommes restés sur des algorithmes assez simples. Les quelques algorithmes assez compliqués à implémenter ont étés assez robustes pour nous résister, et le manque de temps dû à notre investissement sur d'autres projets ne nous a pas permis de mener à bien ces implémentations.

Néanmoins, nous avons aboutis à des résultats satisfaisants sur les complexités temporelles observées depuis notre générateur. Le meilleur algorithme parmi ceux écrits est sans doute l'algorithme mélangeant l'utilisation de tri par aire décroissante et les quadrants.

Ce projet était très intéressant pour l'approche d'un problème assez ouvert, dans le développement d'un travail de recherche et pour le développement d'un compte-rendu.

9 Références

1 - https ://www.youtube.com/watch ?v=w4Dosp2U74Y
2 - https ://pagesperso.g-scop.grenoble-inp.fr/ ~lazarusf/Enseignement/triangulation.pdf
3 - https ://rosettacode.org/wiki/Ray-casting_algorithm
4 - https ://en.wikipedia.org/wiki/Polygon_triangulation