# Compte Rendu Projet Algo

Le Duc Baptiste - Nicolas Malo

Lien du dépôt GitLab :
*https://gitlab.ensimag.fr/algo2024/algo_leducb_nicolmal*

Année scolaire 2023-2024

# SOMMAIRE :

# 1. Introduction

## 1.1 Rappel objectif

Ce projet vise à identifier les diverses inclusions de polygones sans intersection, fournies en entrée dans un fichier.

## 1.2 Division du travail

Pour mener à bien ce projet, nous avons identifié deux axes d'études algorithmique :

- Les algorithmes, que nous désignerons sous le nom de "Point-in-Polygon" qui permettent de déterminer l'inclusion d'**un point** dans le polygone étudié.
- Les algorithmes de parcours d'inclusions qui, à partir d'une liste de polygones, effectuent des tests d'inclusion entre deux polygones en utilisant des algorithmes de type "Point-in-Polygon".

  ***Remarque :*** Les polygones ne s'intersectant pas, il nous suffit de tester un point du polygoneB avec un algorithme "Point-in-Polygon" sur un polygoneA afin de déterminer si polygoneB est inclus dans un polygoneA

L'objectif du projet étant d'optimiser ces différents algorithmes, nous étudierons plusieurs approches algorithmiques avec leur implémentation, évaluerons leur complexité respective et évoquerons des pistes d'amélioration possibles.

# 2. Algorithmes "Point in Polygon"

## 2.1 Algorithme de Ray-Casting classique

L'algorithme de Ray-Casting est une méthode "Point-in-Polygon" relativement simple. Pour déterminer l'inclusion d'un point P dans un polygone, on compte le nombre d'intersections entre un rayon partant du point P dans une direction fixe et les segments du polygone. En conséquence :

- Si le nombre d'intersections est pair, alors le point ne se trouve pas à l'intérieur du polygone.
- Si le nombre d'intersections est impair, alors le point est situé à l'intérieur du polygone.

**Implémentation :** En se basant sur les coordonnées des points formant les extrémités des segments (points d'extrémité), nous pouvons déterminer si la droite issue du point P intersecte les segments et de compter le nombre d'intersections.

**Limite :** Pour un point donné il faut parcourir l'**ensemble des segments** du polygone donné afin de déterminer le nombre d'intersections avec celui-ci.

**Complexité:** Pour un point donnée, on effectue $O(N_e)$ tests d'intersections où $N_e$ correspond au nombre de segments du polygone.

## 2.1 Algorithme de Grid Point-in-Polygon

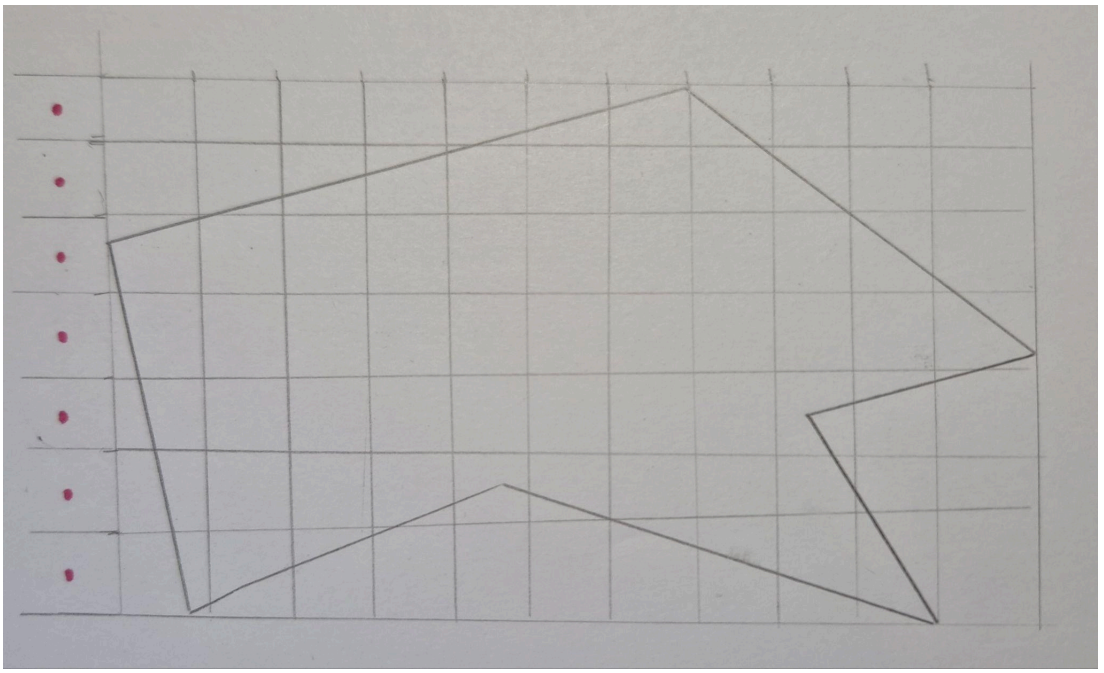
Cet algorithme issu d'un rapport de recherche, permet de tester l'inclusion d'un point dans un polygon via des grilles uniformes. Il consiste en trois étapes :

- **Etape 1 :** Construction de la grille uniforme

Cette phase implique la création d'une grille couvrant la zone englobant les dimensions maximales du polygone. Nous utiliserons ensuite les points centraux de chaque cellule de cette grille.

Ensuite, nous ajoutons une colonne à gauche. Les points centraux des cellules de cette colonne ont leur statut d'inclusion déterminé : ils ne sont pas inclus dans le polygone car ils ne se trouvent pas dans la zone délimitée par les dimensions maximales du polygone.

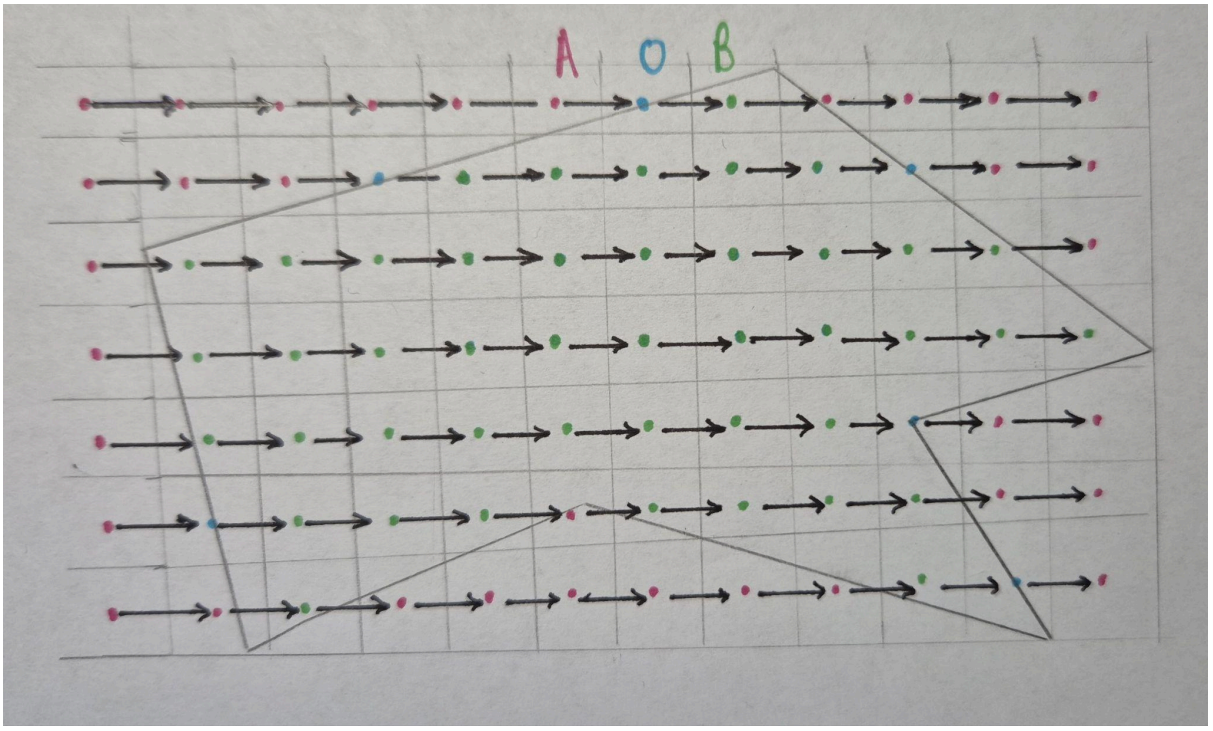
Pendant la construction, il est également nécessaire d'identifier les cellules traversées par les segments du polygone. Cela nous sera utile par la suite lorsque nous voudront appliquer un algorithme de Ray-Casting local. Pour réaliser cela on utilisera des algorithmes dits de "Line Generation" *(cf 3. Algorithme de Line-Generation)* permettant de tracer des approximations de segments.



- **Etape 2 :** Détermination de l'inclusion des points au centre des cellules de la grille

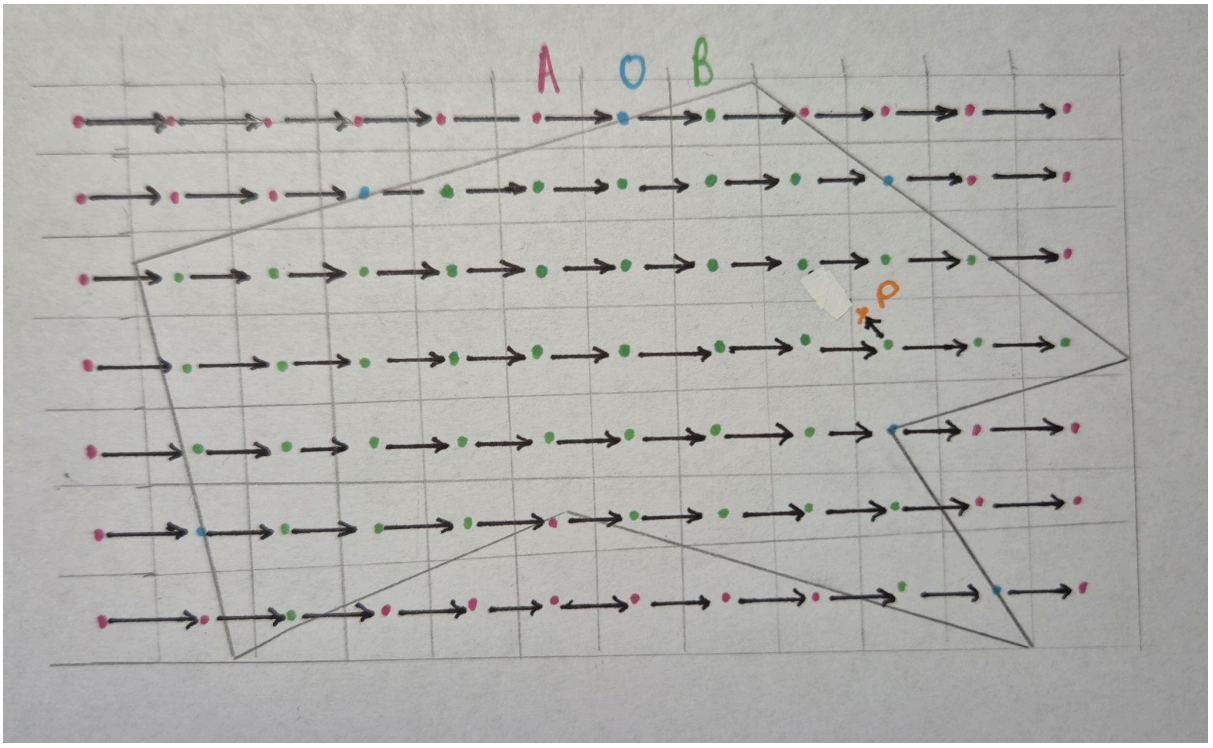
Dans cette phase, nous itérons à travers les lignes de la grille, en commençant par les points centraux qui ne sont pas inclus car appartenant à la colonne ajoutée. De manière itérative, nous établissons des segments entre les points de la ligne, deux par deux. Pour chaque paire de points centraux formant un segment, nous appliquons localement l'algorithme de Ray-Casting afin de déterminer l'inclusion du point à droite (l'inclusion du point à gauche étant déjà connue, puisque lors de l'itération 0, les propriétés d'inclusion des points de la colonne ajoutée sont établies).

Si un des points centraux (noté O sur le schéma) se situe sur un des segments du polygone alors nous le noterons comme étant "singular" et feront les tests d'inclusion du point B à l'aide du point A en étudiant les intersections du segment AB.




- **Etape 3 :** Test d'inclusion sur le point P

Enfin, pour déterminer si le point P est inclus, il nous suffit d'identifier la cellule de la grille à laquelle il appartient, puis d'appliquer localement l'algorithme de Ray-Casting sur le segment formé par P et le point central de cette cellule.




**Avantage comparé à la méthode de Ray-Casting classique :**
En utilisant cette méthode, on applique l'algorithme de Ray-Casting localement en parcourant non pas tous les segments du polygone mais seulement ceux qui ont traversé les cellules étudiées.

**Complexité :**

- **Construction de la grille :** $O(M)$ où $M$ représente le nombre de cellules dans la grille
- **Algorithme de Line Generation :** linéaire avec le nombre de cellules intersectées par les segments du polygone. En effet, si on suppose que un segment du polygone intersecte en moyenne $f_e$ cellules d'une grille de $M$ cellule alors une cellule se fait traverser par $f_e N_e / M$ segment en moyenne (où $N_e$ représente le nombre de segments du polygone étudié. La complexité pour cette étape est alors de $O(f_e N_e)$
- **Test d'inclusion des points centraux des cellules** : Un segment dans une cellule va être utilisé au plus 2 fois : une fois au moins pour déterminer l'inclusion du point central courant et une fois pour déterminer l'inclusion du point dans la cellule suivante. Ainsi, chaque segment va être visité $2f_e$ fois donc on a une complexité de $O(f_e N_e)$
- **Test d'inclusion pour un point P** :
    - Trouver la cellule contenant P : $O(1)$

    - Parcourir l'ensemble des segments intersectant cette cellule : comme précédemment dit, une cellule contient en moyenne $f_e N_e/M$ segments donc itérer sur ces segments et effectuer un test d'intersection sur chacun d'eux (en $O(1)$) nous donne une complexité de $O(f_e N_e/M)$
- **Au total :**
    - On obtient alors une complexité de $O(M + 2f_e N_e)$

## 3. Algorithmes de Line Generation

Afin d'utiliser l'algorithme de Grid Point-in-Polygon il est nécessaire d'identifier les cellules traversées par les segments du polygone. Pour cela, il nous faut approximer les segments via des Algorithmes de Line Generation.

### 3.1 Digital Differential Analyzer (DDA)

L'algorithme DDA interpole les valeurs entre les deux points $(x_i, y_i)$ notée d'un segments en utilisant ces relations :

- $x_i = x_{i-1} + 1$
- $y_i = y_{i-1} + m$ où m est la pente du segment : $m = \frac{y_{end} - y_{start}}{x_{end} - x_{start}}$

**Problème rencontré :** Cet algorithme approxime x en l'incrémentant de 1 ce qui est peu précis pour identifier des cellules dont la taille est largement inférieur à 1.

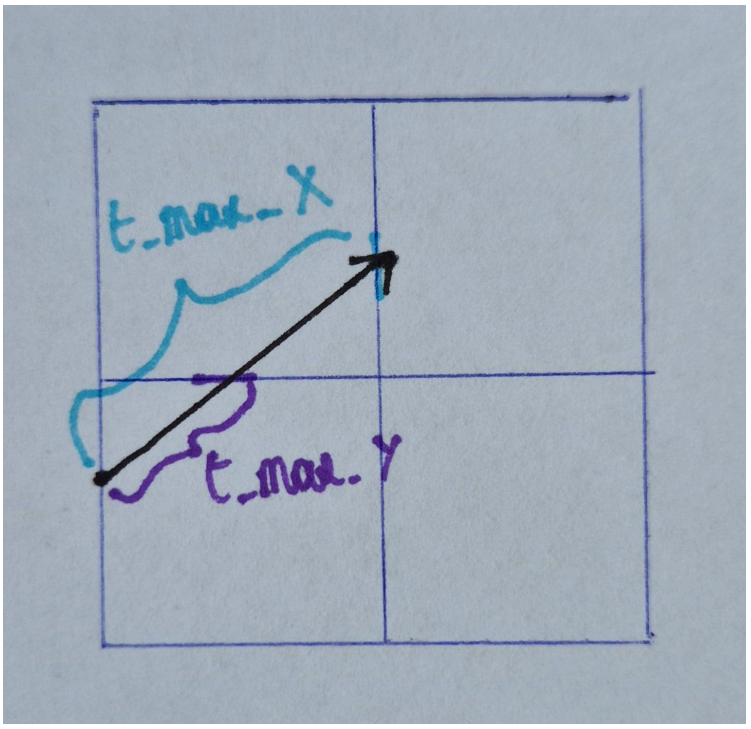
### 3.1 Fast Voxel Traversal Algorithm

Nous avons choisi d'implémenter cet algorithme issu d'un rapport de recherche (*cf. 6. Bibliographie*), car il offre la possibilité de définir de manière précise le pas d'incrémentation de x et de y. Contrairement à l'algorithme DDA, cela nous permet de mieux identifier les cellules traversées par un segment.

Il consiste en 2 étapes :

- **Initialisation**
    - Notons (x1, y1) les coordonnées d'une extrémité d'un segment.
      Dans cette étape nous déterminons dans quelle cellule de notre grille se trouve le point (x1, y1) : c'est à partir de cette cellule que nous allons approximer le segment.
    - `step_x` et `step_y` : Ces variables déterminent la direction dans laquelle le rayon se déplace le long des axes X et Y respectivement, en fonction de la pente du rayon par rapport à la grille.

- `t_max_x` et `t_max_y` : Ce sont les distances maximales que le rayon peut parcourir avant de rencontrer une limite d'une cellule dans les directions X et Y.




- `t_delta_x` et `t_delta_y` : correspondent à la distance depuis le point de départ nécessaire pour atteindre la cellule suivant d'après l'axe des abscisses et des ordonnées

- **Traversée par incrémentation des différents pas**

  **Boucle {**

  - À chaque itération, on détermine si le rayon devrait se déplacer dans la direction X ou Y en comparant les valeurs de `t_max_X` et `t_max_y`.
  - Si `t_max_x` est inférieur à `t_max_y`, le rayon se déplace dans la direction X en mettant à jour `t_max_x` et X.
  - Sinon, le rayon se déplace dans la direction Y en mettant à jour tMaxY et Y.
  - À chaque itération, on détermine dans quelle cellule se trouve (X,Y) et on met à jour `current_cell`

  **}**

# 4. Algorithmes de parcours d'inclusions

## 4.1 Algorithme naïf par tri d'air décroissante

Afin d'effectuer les différents tests d'inclusions sur notre liste de polygone en entrée, nous avons dans cet algorithme, procéder par un tri de cette liste de polygone en entrée par air décroissante. Comme on ne traite que les inclusions directes, il suffit de tester les inclusions deux à deux.

**Complexité :** Au pire des cas, pour un polygone on va effectuer tous les tests sur les polygones d'airs supérieur, on obtient donc une complexité en $O(n^2)$ où $n$ est le nombre de polygones.

## 4.2 Algorithme pour la détection des potentiels inclusions

Dans le but de réduire le nombre de tests à effectuer, on va déterminer une liste d'inclusion potentiel pour chaque polygone.

Nous allons nous aider d'une liste chaînée dont les cellules contiennent un polygone, couplée à un tableau dont l'élément i contient la cellule du polygone i de la liste. Nous allons parcourir le plan de gauche à droite en regardant les deux points d'abscisse minimale et maximale correspondant à l'entrée et la sortie du polygone. De ce fait, pour un point d'entrée, on ajoute en tête de liste le polygone correspondant. Pour un point de sortie, on considère les polygones enfants comme des polygones dans lesquels il est potentiellement inclus, dans l'ordre croissant, et on supprime le polygone de la liste. A la fin, pour chaque polygone, on obtient une liste de polygones avec qui tester l'inclusion.

Nous exécutons de nouveau le même procédé mais en parcourant de haut en bas le plan et effectuons l'intersection des deux résultats obtenus.

**Complexité :**

- L'ajout d'un élément à la liste est de coût constant
- La suppression d'un élément de la liste est de coût constant car au lieu de parcourir la liste pour trouver l'élément à supprimer, on le stocke dans le tableau associé
- Nous prenons en compte 2 points par polygones et parcourons l'ensemble des points

Nous obtenons donc une complexité temporelle en $O(2n)$ mais une complexité spatiale en $O(2n)$ aussi.

## 5. Tests et analyse des résultats obtenus

Passons maintenant à la partie tests et analyses. Notons premièrement que tous les algorithmes évoqués passent nos tests personnels. Néanmoins dans le cas de l'algorithme Grid Point-in-Polygon, nous ne passons pas les tests automatiques en ligne du tournoi. Le problème vient de l'algorithme fast voxel transversal, qui peut effectuer de trop grosses approximations dans des cas assez particuliers de polygones comme celui évoqué ci-dessous :

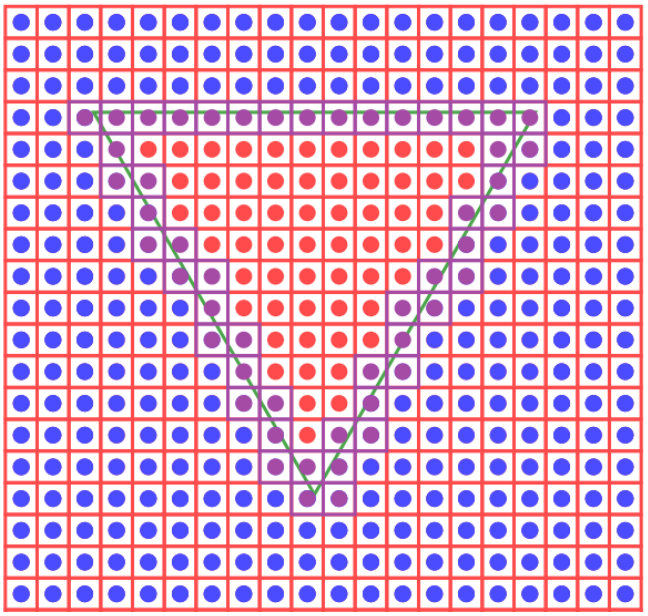


Figure 1

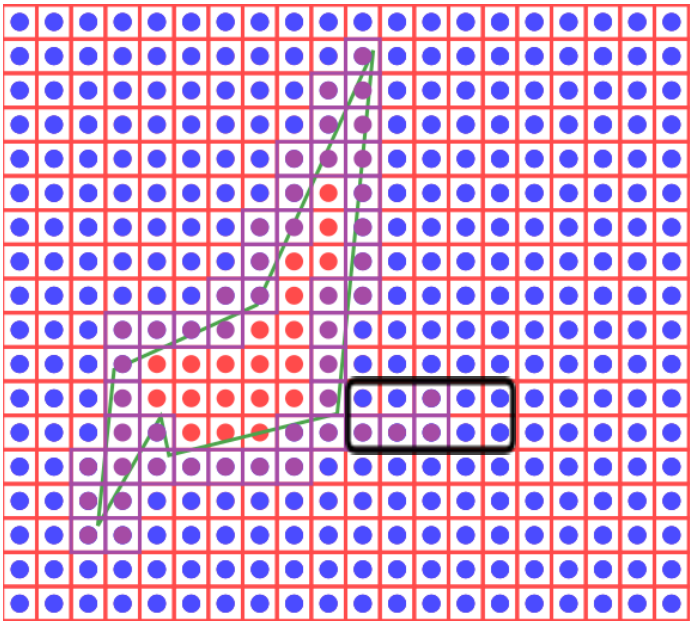


Figure 2

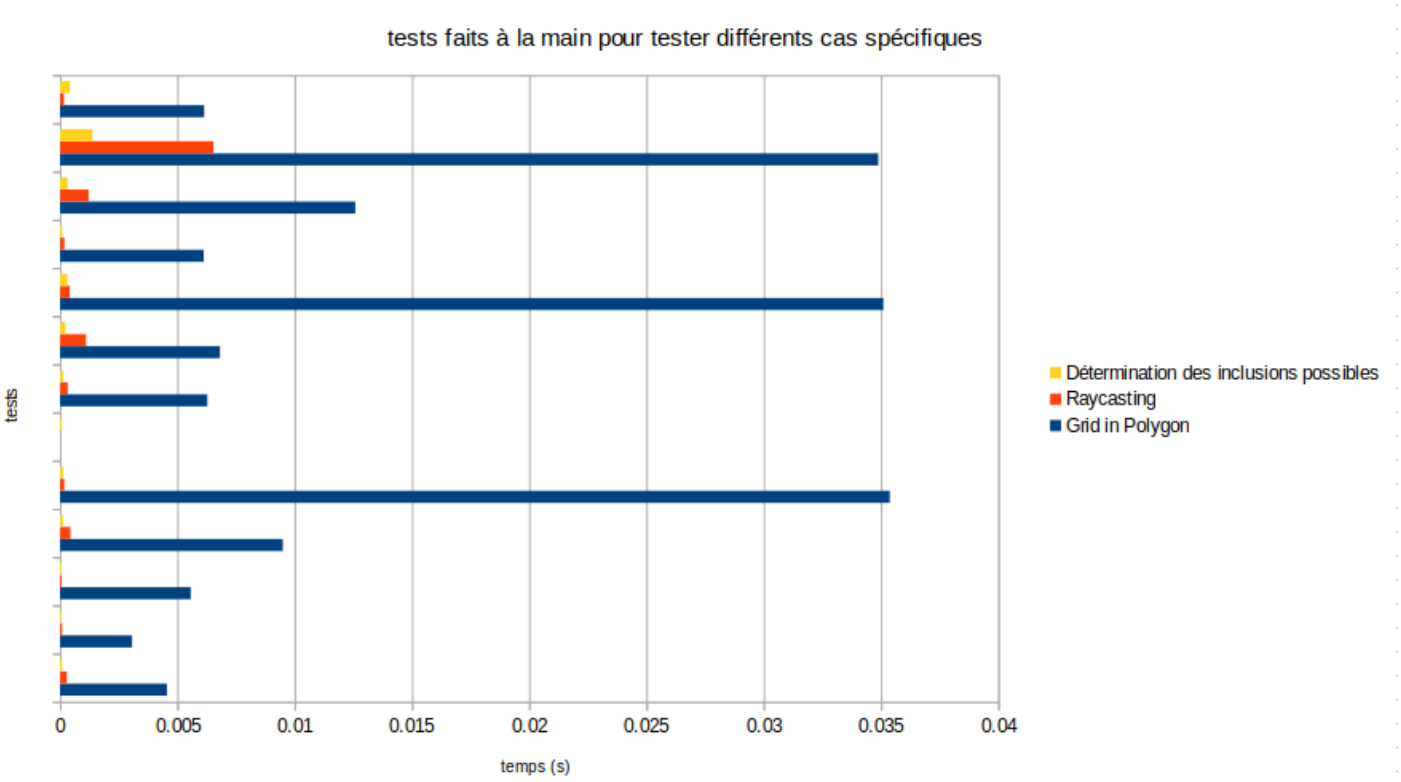
**Légende :**

- Cellules violettes: cellules (et points centraux associés) reconnus par le Fast-Voxel Algorithm comme étant traversées
- Points rouges : points centraux des cellules étant reconnus comme inclus dans le polygone par l'algorithme Grid Point-in-Polygon
- Points bleus : points centraux des cellules étant reconnus comme non-inclus dans le polygone par l'algorithme Grid Point-in-Polygon
- Carré noir : cellules violettes faussement considérées comme traversées par un segment par le Fast-Voxel Algorithm

En effet, dans la Figure 1, nous remarquons que le Fast-Voxel Algorithm a produit une approximation satisfaisante des cellules traversées par les segments du polygone. En revanche, dans la Figure 2, des cellules violettes sont visibles *(cf. carré noir sur Figure 2)*, bien qu'elles ne soient pas traversées par un segment du polygone. Ce phénomène est un problème du Fast-Voxel Algorithm que nous avons identifié,

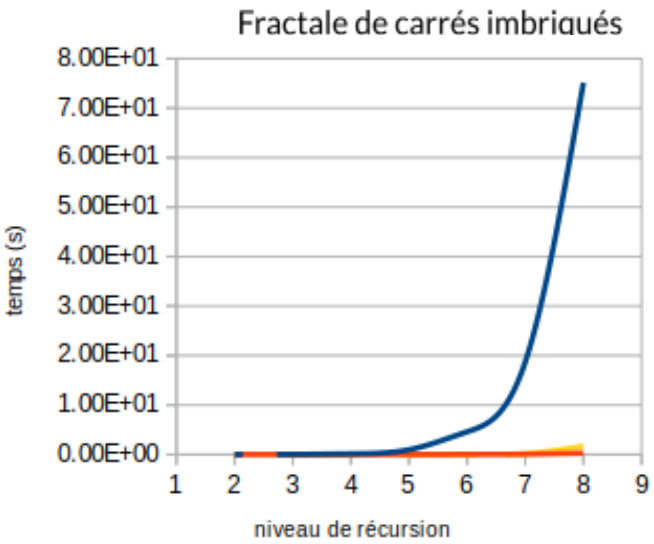
résultant des approximations arithmétiques pour les nombres flottants réalisées lors de son implémentation qui fait que pour de très petits polygones on peut voir des erreurs apparaître.

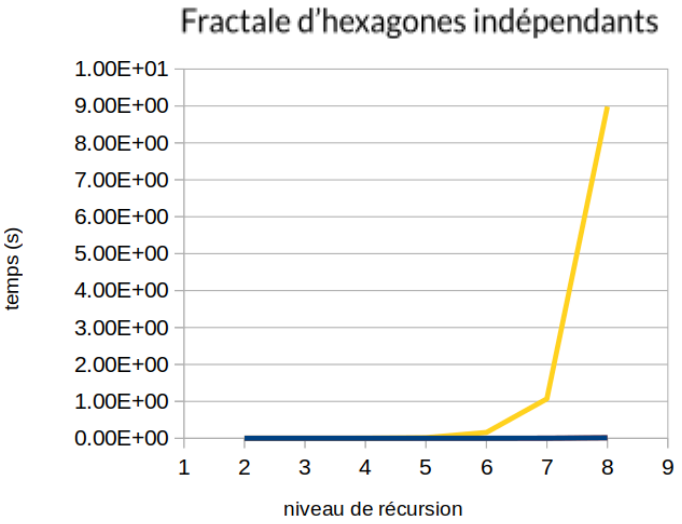
Cela correspond à un axe d'amélioration de notre algorithme Grid Point-in-Polygon.



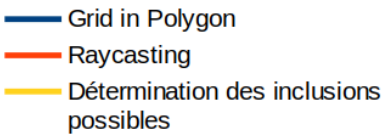
En ce qui concerne les tests avancés, nous avons utilisé une série d'exemples recueillis sur GitLab par un enseignant ou un élève de l'ENSIMAG [1]. Ces tests couvrent un éventail de paramètres d'entrée, tels que le nombre de polygones, le niveau de récursion, le nombre de carrés ou le nombre de lignes, à travers des tests manuels ainsi que des scénarios spécifiques impliquant des données d'entrée de taille croissante. Outre la validation du bon fonctionnement de nos algorithmes, cette méthode nous permet de discerner les cas favorables et défavorables.

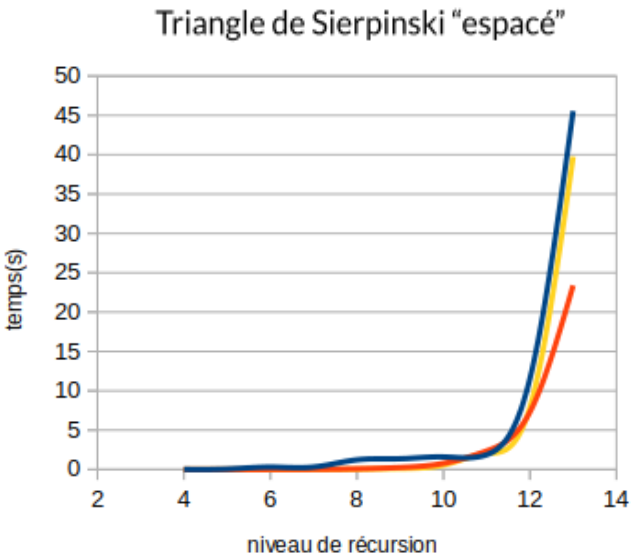
**Test sur le niveau de récursion :**



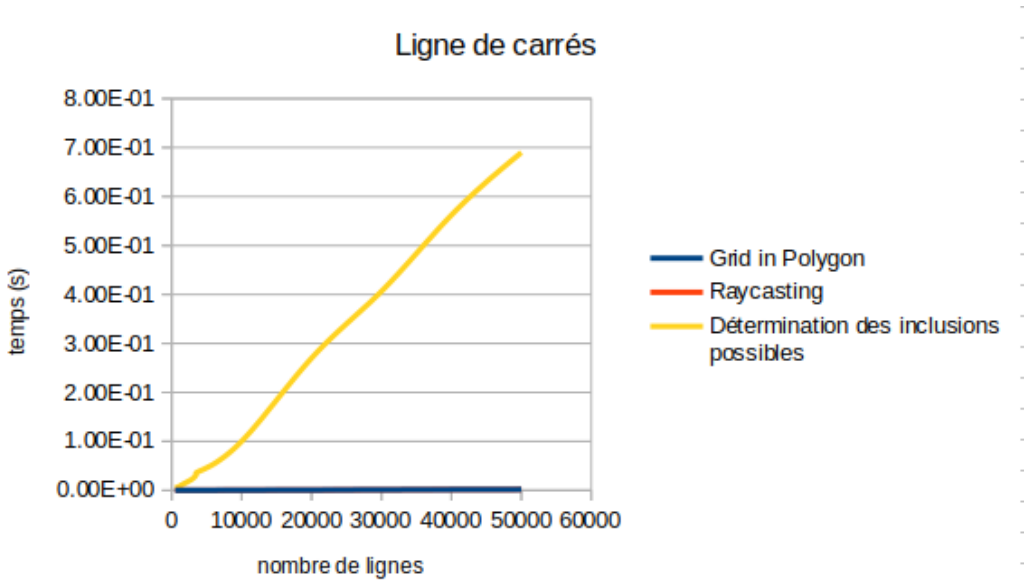




**Légende :**

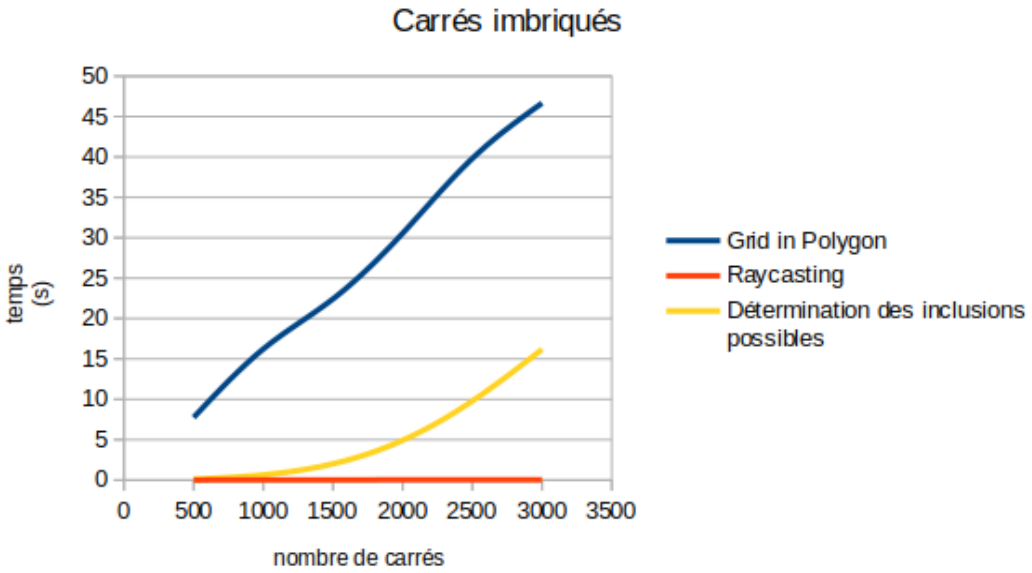


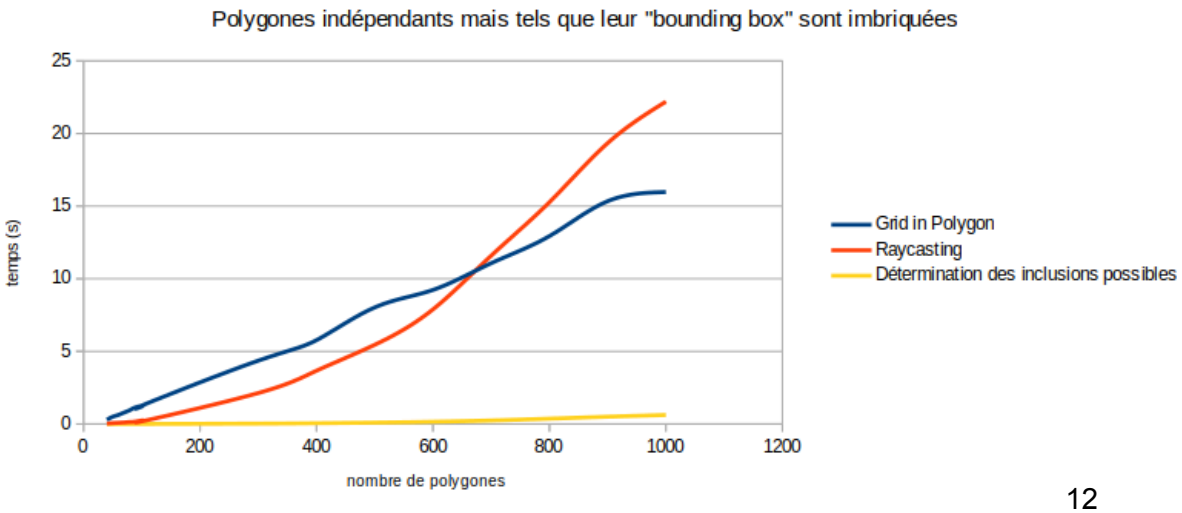
**Test sur le nombre de lignes :**




**Test sur le nombre de carrés :**




**Test sur le nombre de polygones :**



D'après les graphes ci-dessus, nous pouvons observer dans la plupart des cas une meilleure performance lors de l'utilisation du Ray-Casting. Cela s'explique par le fait que Grid Point-in-polygon trouve tout son intérêt lorsque la grille est déjà mise en place car on obtient un coût constant pour la détermination de l'inclusion. Or dans la plupart des cas présent, nous devons calculer beaucoup de grilles lorsque le polygone parent apparaît pour la première fois lors des tests inclusions. Cela se confirme dans le cas où le raycasting est moins performant. En effet dans ce cas-ci, on se trouve dans le pire cas pour le raycasting car on effectue les tests d'inclusions sur chaque polygone d'air supérieur.

## 6. Conclusion

Ainsi, nous avons examiné plusieurs algorithmes pour les deux étapes cruciales de la résolution de notre problème : l'étape de détermination d'inclusion d'un point dans un polygone et l'étape de parcours d'inclusion . Notre approche a consisté à chercher à optimiser chacune de ces étapes en ayant en tête de vouloir obtenir une vision plus locale du problème, en menant des recherches approfondies sur les avancées existantes dans le domaine. Cependant, lors de l'implémentation de l'algorithme Grid Point-in-Polygon, nous avons réalisé qu'un algorithme optimisé pour un cas spécifique du problème ne l'était pas nécessairement dans un contexte plus général. Nous avons aussi réfléchi de notre côté pour trouver des algorithmes plus personnels afin de résoudre notre problème.

Ce projet a été une expérience de travail en binôme nous offrant l'opportunité d'échanger nos compétences théoriques et pratiques sur un nouveau problème, tout en développant des compétences avec lesquelles nous étions moins à l'aise : le calcul de la complexité ou l'utilisation de Git par exemple. Nous avons également réalisé l'importance de produire un code clair et maintenable pour faciliter le travail en équipe et permettre une détection plus efficace des bug ainsi que de séparer notre travail dans plusieurs fichiers.

## 7. Bibliographie

[1] *https://gitlab.ensimag.fr/raffingu/polygon-adaptive-test*
[2] A Fast Voxel Traversal Algorithm For Ray Tracing John - Amanatides, Andrew Woo
[3] Point-in-Polygon Tests by Determining Grid Center Points in Advance - Jing Li, Wencheng Wang