Projet Algorithmique

Nolan Carouge et Sylvain Desbiolles

Année 2023 - 2024

Sommaire

1 Introduction

Dans ce compte-rendu de projet, nous détaillerons les différents algorithmes que nous avons pu développer ainsi que leurs fonctionnements. Nous expliquerons pourquoi certains algorithmes sont plus rapides sur certaines entrées et pourquoi ils sont plus longs sur d'autres. Nous aurons une partie sur notre générateur de polygones ainsi que les tests que nous avons pu réaliser avec ce dernier. Tout ceci sera accompagné de calculs de complexité (quand cela sera possible) ainsi que de graphiques pour visualiser clairement nos résultats.

2 Algorithmes et améliorations

Lors de la première séance, nous avons tout d'abord commencé par réfléchir à un algorithme qui permettrait de détecter des inclusions de polygones. Nous avons remarqué qu'un algorithme était proposé dans l'énoncé. Nous avons donc décidé de l'implémenter, au départ de manière très naïve. Cet algorithme, appelé algorithme de jeté de rayon ou Ray Casting en anglais, consiste à prendre un point et à tracer un trait (dans n'importe quelle direction) et à compter le nombre de fois où l'on intersecte un polygone. Si le nombre d'intersections est impair alors le point est inclus dans le polygone et inversement.

2.1 Algorithme n°1 :

Notre premier algorithme fonctionnait alors de la manière suivante : on prend le premier point d'un polygone et on parcourt l'ensemble des points de chaque polygone en comptant le nombre d'intersections. Pour compter le nombre d'intersections, il suffit lorsque l'on parcourt les polygones de prendre des couples de points et de déterminer la droite formée par ces derniers. Ensuite, il nous suffit de comparer les coordonnées de notre point initial avec celui correspondant sur la droite. Cependant, cet algorithme soulève plusieurs problèmes :

1). Il est extrêmement lent, en effet si par exemple le polygone A est inclus dans B et B dans C alors il est inutile de tester si le polygone A est inclus dans C de même pour C dans A. L'algorithme détecte les inclusions de manière plutôt brutale, mais cela permet de donner une bonne idée du temps d'exécution dans le cas où rien n'est optimisé.

2). Il ne fonctionne tout simplement pas. En effet, imaginons que nous sommes dans la situation suivante : nous avons pris le premier point du polygone A et ensuite, nous prenons les deux premiers points du polygone B sauf que ces deux points sont alignés verticalement avec le point du polygone A. Lorsque, l'on va générer la droite formée par les deux points, nous n'allons pas pouvoir compter le nombre d'intersections entre cette droite et le point de A, car les trois points seront sur la même droite.

2.2 Algorithme n°1 bis :

Nous avons donc par la suite décidé de rester sur cet algorithme, mais en premier lieu de le rendre fonctionnel. Pour ce faire, il fallait simplement ne plus prendre le premier point du polygone A mais prendre un point qui n'est pas aligné avec les deux points à tester. Nous avons donc créé une fonction qui trouve deux points consécutifs de A qui n'ont pas la même abscisse. Ensuite, nous prenons un point aléatoirement sur la droite

formée par ces deux points. Le fait de prendre ce point aléatoirement nous permet d'être sûr à presque 100% que les deux points verticaux de B et le point de A ne sont pas alignés par rapport à une même droite verticale.

Cet algorithme permet de passer nos différents tests et de passer un des trois tests en ligne (Test 0 : 7047 ms et pas de temps pour les autres). Comme nous avons pu le voir grâce à vos tests en ligne et les nôtres, l'algorithme est mauvais. En termes de complexité concernant cet algorithme: pour chaque polygone, on parcourt l'ensemble des polygones et l'ensemble des points de chacun de ces polygones. Ainsi, on a une complexité de l'ordre de $O(n^{2}*nbrptsmax)$ avec n le nombre de polygones et nbrptsmax le nombre de points du polygone ayant le plus de points. Nous devions donc l'améliorer.

2.3 Algorithme n°2 :

Nous avons donc procédé à une grosse amélioration. L'amélioration de l'algorithme consiste à ne pas tester l'ensemble des inclusions entre les différents polygones. Comme dit, précédemment, il est inutile de tester si le polygone i est inclus dans le polygone j, en sachant déjà que le polygone j est inclus dans le polygone i. Nous avons utilisé le fait que si le polygone i est inclus dans le polygone j, alors l'abscisse maximale de j est supérieure à celle de i, l'abscisse minimale de j est inférieure à celle de i, de même pour les ordonnées. Ainsi, avant de commencer à tester les inclusions, un tri est donc effectué sur les polygones par ordre croissant d'abscisse minimale. Pour le polygone P d'indice j, il n'est donc pas nécessaire de chercher une inclusion avec les polygones d'indice inférieur à j. De plus, comme l'on cherche le plus petit polygone dans lequel P est inclus, on cherche donc les inclusions avec les polygones d'indice supérieur à j et on s'arrête de tester dès que l'on a trouvé une inclusion. Cette méthode permet ainsi de diminuer fortement le nombre de tests d'inclusion à effectuer.

La complexité pour obtenir la liste des abscisses maximales triée est égale au maximum entre le nombre de points sur l'ensemble des polygones (pour obtenir toutes les abscisses minimales) est : O(nlog(n)); où n est le nombre de polygones (pour trier la liste). Concernant l'algorithme trouvant les inclusions, on va parcourir la liste, et pour chaque indice i, on va la reparcourir à partir de l'indice i, en effectuant un test d'inclusion quand les conditions énoncées précédemment sont respectées, et on arrête le parcours à la première inclusion trouvée. Ainsi, dans le pire des cas, il faudra tester les inclusions pour tous les polygones, on aura donc une complexité environ en $O(n^{2}*nbrptsmax)$; où n est le nombre de polygones et nbrptsmax est le nombre de points du polygone ayant le plus de points. Dans le meilleur des cas, le plus petit père de chaque polygone sera le polygone le suivant dans la liste, on aura donc une complexité environ en O(n*nbrptsmax). On peut donc conclure que cette version possède une meilleure complexité que la première.

2.4 Algorithme n°2 bis :

Après avoir réalisé l'algorithme 2, nous avons eu l'idée d'une potentielle optimisation. Par exemple dans le fichier 10x10.poly les points sont alignés, mais divisés en un sous-nombre de points élevé. Cela implique que de nombreux points sont inutiles. Un nouvel algorithme serait de d'abord traiter le fichier pour enlever les points qui ne sont pas utiles. Seulement, avant de réaliser cet ajout, nous ne savions pas si le fait de traiter le fichier nous ferait gagner du temps. L'algorithme de modification est très simple, on

dispose de trois pointeurs sur le fichier que l'on fait avancer suivant certaines conditions. Avec ces trois pointeurs, nous pouvons écrire les points importants du polygone. Notre algorithme de traitement de fichier ne peut traiter les points alignés que sur l'axe x et y. La gestion des points alignés sur tout autre axe nous parait bien trop compliquée et très peu optimisable. Nous verrons dans la section de mesure de performance si cet ajout a été bénéfique et sinon comment nous avons essayé de l'améliorer.

3 Générateur de polygones

Pour réaliser nos tests, nous avons développé un générateur de polygones. Nous avons pris un parti. Notre générateur de polygones ne va pas nous donner la liste des polygones inclus entre eux. En effet nous savons avec nos tests personnels que les algorithmes fonctionnent. Nous nous servons de ce générateur seulement pour mesurer les performances.

Notre algorithme de génération de polygones consiste à partir d'une liste vide qui au final contiendra la liste de points de chaque polygone. On va ensuite ajouter des polygones un par un en choisissant aléatoirement le nombre de points du polygone (entre 3 et 8). Pour ajouter un polygone, on va placer un par un les points en les choisissant de manière aléatoire. Pour chaque point aléatoirement choisi, on teste si le (ou les si le point est le dernier du polygone) segment(s) engendré par ce nouveau point croise un segment déjà existant sur l'image. Si un croisement a lieu, le nouveau point n'est pas ajouté et on recommence jusqu'à obtenir le nombre de points voulu. Nous avons remarqué que dans certains cas, il était très difficile (voire impossible) de placer un nouveau point, ainsi nous avons ajouté un timer qui permet, lorsque l'on a attendu trop longtemps (1 seconde), de supprimer le polygone en cours d'ajout et de repartir. Cela permet de "tomber" bien plus rapidement sur une solution. Ainsi, tous les points de ce générateur sont placés aléatoirement. Il s'agit d'un générateur assez naïf dont il est sûrement impossible de calculer la complexité. De plus les polygones générés sont totalement aléatoires, on ne pourra donc pas tester des cas particuliers. Il est aussi un petit peu lent, mais ce dernier a totalement rempli son rôle comme nous le verrons dans la section suivante.

4 Mesures de performance

Avec le générateur de polygones, nous avons pu réaliser des tests de performance. En effet, nous avons généré 10 fois des fichiers .poly allant de 25 à 500 polygones (de 3 à 8 côtés) par pas de 25. (Nous étions limités par la taille du rendu sur TEIDE, nous avons mis seulement les 3 premiers dossiers). Nous avons donc 200 fichiers .poly. Nous avons pu exécuter nos trois versions sur ces fichiers (V1 : 1 bis, V2 : 2, V3 : 2 bis) pour déterminer le temps d'exécution. Avec toutes ces données, nous avons réalisé une moyenne par nombre de polygones et tracé des graphes représentant le temps d'exécution.

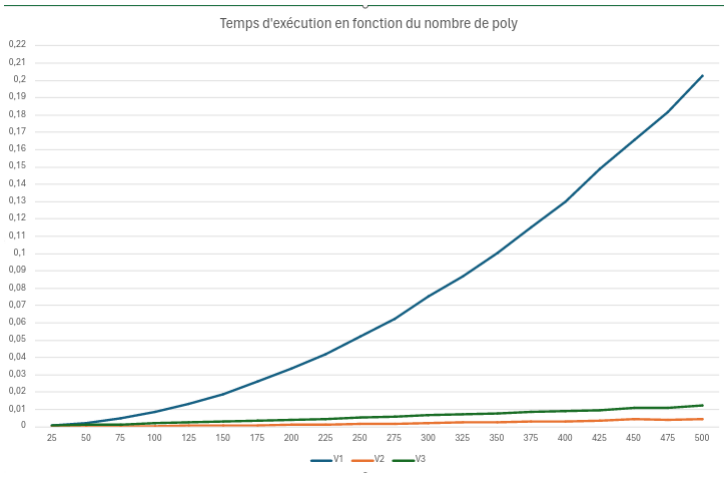

Figure 1: Temps d'exécution pour les 3 versions.

Sur cette figure, nous observons les temps d'exécution des trois versions. On voit bien, sur la Figure 1, que quand le nombre de polygones augmente le temps d'exécution explose pour la version 1. En effet comme dit précédemment nous avions trouvé que la complexité était de l'ordre de O(n^{2}*nbrptsmax). On remarque bien ce comportement non linéaire du temps d'exécution. Comme nous aurions pu nous en douter pour les deux autres versions, le comportement est un peu différent, le temps d'exécution ressemble plus à une droite linéaire. Pour rappel, on avait trouvé une complexité de l'ordre de O(n*nbrptsmax), le résultat était donc prévisible.

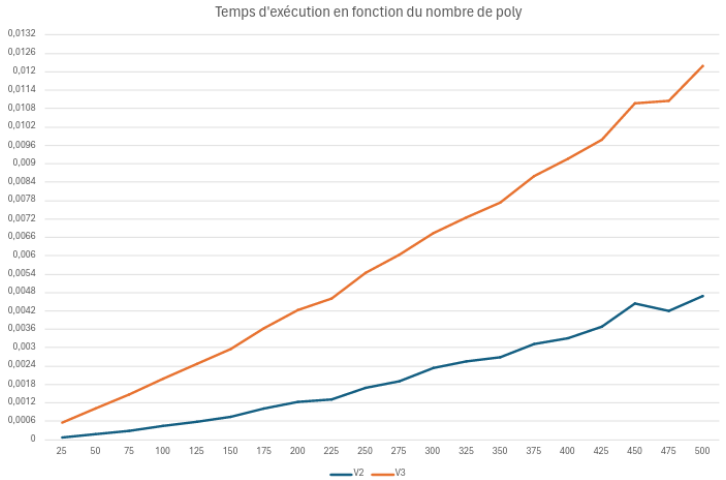

Figure 2: Zoom sur version 2 et 3.

Sur la figure 2, nous observons les temps d'exécution de la version 2 et de la version 3. Cependant, cette figure ne nous a pas surpris. En effet, notre générateur de polygone génère des polygones de manière totalement aléatoire, il est donc impossible que trois points se retrouvent alignés sur l'axe x ou y. Notre algorithme de traitement de fichier ne peut donc pas fonctionner sur des entrées réalisées par notre générateur. Cette différence de temps d'exécution représente donc le temps de traitement du fichier.

Ensuite, nous avons réalisé un dernier test. Ce test correspond au temps d'exécution pour le fichier "test/special.poly". Ce fichier correspond à une version spécifique de 10x10.poly où nous avons divisé chaque segment en un nombre très élevé de points. Au final, nous avions un fichier de presque 100 000 points. Malheureusement, les résultats nous ont un petit peu déçus. Effectivement, le temps d'exécution de la version 3 était un petit plus élevé que pour la version 2 (0.023078203201293945s contre 0.030479907989501953s).

Nous avons donc réalisé un test entre le fichier 10x10.poly (pour la v2) et le fichier tmp (pour la v3) crée lors de l'exécution de l'algorithme de traitement de fichier. Nous avons trouvé un temps d'exécution de 0.020756006240844727s pour la version 2 et 0.00020551681518554688s pour la version 3 avec le fichier déjà traité. Nous nous rendons compte que la version 3 est plus efficace si l'on optimise ce traitement de fichier. Seulement, au tout départ, dans la première version de l'algorithme de traitement de fichier, nous stockions l'entièreté du fichier dans un tableau, ce qui était très coûteux. Nous l'avons donc optimisé en réalisant un traitement à la volée et nous espérions avoir un temps d'exécution plus rapide, mais cela n'a pas été le cas. Nous sommes cependant convaincus que sur un fichier de plusieurs millions de points alignés sur l'axe x ou y, l'algorithme version 3 serait le plus efficace. Malheureusement ce cas a très peu de chances d'arriver.

5 Conclusion

Finalement, malgré nos tentatives d'améliorations, notre algorithme le plus performant restera le numéro 2. Cependant, en fonction des entrées, nous pouvons obtenir des résultats différents. Effectivement, dans la plupart des cas et pour la plupart des entrées, l'algorithme 2 sera le plus performant. Cependant, si l'entrée devient plus particulière, comme pour le fichier special.poly, un grand nombre (de l'ordre de 500 000) de points alignés rendra alors l'algorithme numéro 3 le plus efficace.

Ce projet nous aura permis de mettre en œuvre des connaissances que nous avons acquises au premier semestre ainsi qu'au deuxième. Nous avons appris à travailler en commun sur un même projet avec l'aide de git. Puis nous avons pu mener des tests de performance, ce qui nous a permis de conforter ou non nos calculs de complexité théorique. Ce fut un projet intéressant.