Rapport du projet de géométrie numérique

Guillaume Grosshenny Audric Wasmer

2018-02-26

Contents

1 Explicitation des choix

1.1 Recherche des plus proches voisins

Etant donné l'importance de la recherche des plus proches voisins, que ce soit avec le nuage de points donné en entrée lors de la création des plans ou bien avec les centres des plans lors de la création de la grille de voxels, il nous est apparu essentiel d'améliorer cette recherche afin de rendre l'algorithme utilisable en temps raisonnable. C'est donc pour cela que nous avons décidé d'utiliser un kd-tree avec $k = 2$, ce qui permet de rendre la recherche des plus proches voisins en $\theta(klog(n))$.

1.2 Construction des plans tangents

Afin d'accélérer le développement pour notre projet, nous avons décidé d'utiliser la librairie *Eigen* pour le calcul des *moindres carrés* et pour la *décomposition spectrale* de la matrice de covariance. Cependant, nous avons expérimenté plusieurs problèmes avec l'application des moindres carrés où nous obtenions des valeurs considérées comme inacceptables *(i.e : pour une entrée avec des points avec des valeurs de x allant de -1.0 à 1.0, nous obtenions des valeurs telles que x = 11 000).* Nous avons donc décidé de trouver le centroid en calculant la moyenne entre tous les voisins, ce qui a l'avantage d'être plus rapide que l'utilisation des moindre carrés, nous avons donc une complexité de $\theta(n)$ au lieu de $\theta(k^2 * n)$.

1.3 Orientation des plans

Pour faciliter la création de l'arbre couvrant minimal une fois les plans tangents mis sous forme de graphe, nous avons fait le choix d'utiliser la librairie Boost qui propose une implémentation de l'algorithme de Kruskal avec une complexité de $O(E * log(E))$ où E est le nombre d'arête du graphe d'entrée. Néanmoins, cela nécessite une conversion du graphe vers des structures de données différentes de celles que nous avons nous même implémenté, celles-ci ne pouvant être prises en entrée par l'algorithme Kruskal de Boost. Ladite conversion est en $\theta(n * k)$ avec n le nombre de noeuds (i.e. de plans) dans le graphe, et k le nombre de voisins. Quant à l'orientation des plans, nous avons implémenté une méthode fondée sur un parcours récursif de l'arbre couvrant minimum en $O(n)$, où n est le nombre de plans.

1.4 Création de la grille de voxels

La grille de voxels est une liste chaînée de cubes permettant d'avoir des trous dans la grille afin de réduire le nombre de cube visités par l'algorithme de marching cubes. Nous sommes restés proches de la version décrite dans l'algorithme.

1.5 Reconstruction de l'isosurface

Pour la reconstruction de l'isosurface, nous avons fait le choix d'implémenter la méthode des Marching Cubes de *Paul Bourke* [1].

[1] Polygonizing a Scalar Field - Paul Bourke

2 Explicitation des méthodes de test

2.1 Principe des tests mis en place

Les classes sont testées avec des tests unitaires afin de vérifier le comportement des méthodes publiques. Nous avons utilisé entre autres les méthodes *assert* de *c++* afin de réaliser ces tests. Il y a un jeu de test par librairie dans le projet, afin de tester chaque étape séparément et un test d'intégration a été réalisé en analysant le résultat de l'algorithme avec un nuage de point en entrée.

2.2 Ouverture

Afin de tester la pertinence des résultats obtenus grâce à cet algorithme, nous aurions pu utiliser un logiciel tiers tel que *metro* [2]. Le test prend en entrée un nuage de point provenant d'un maillage déjà créé que l'on passe dans l'algorithme. On récupère en sortie le maillage correspondant au nuage de point et on teste la différence entre les deux maillage, ce qui nous donne une idée de la précision de l'algorithme.

3 Complexité

Comme précisé précédemment, la complexité de la création des plans tangents est en $\theta(n+klog(n))$, où k est le nombre des plus proches voisins que l'on doit prendre en compte dans l'algorithme. Concernant l'orientation des plans, la complexité est majorée par l'algorithme de kruskal qui est en $\theta(n*k*log(n*k))$ où $n*k$ correspond au nombre de plans multiplié par le nombre de voisins de chaque plan (i.e. le nombre total d'arêtes du graphe) . Quant à l'étape de la création de la grille de voxel, la complexité ne va pas dépendre du nombre de point, mais du ratio entre l'étalement des points composant le maillage et de la taille des cubes que l'on donne en entrée. Si on prend $cx = d(minX, maxX)$, $cy = d(minY, maxY)$, $cz = d(minZ, maxZ)$ où $d(x, y)$ est la distance entre les deux valeurs et où minX|Y|Z et maxX|Y|Z sont les minimums et les maximaux des coordonnées X, Y et Z tous points confondus. On a donc un complexité telle que $\theta(\frac{cx*cy*cz}{cubeSize} * log(n))$ où cubeSize est définie en fonction de p qui est la densité du nuage de point et de δ qui est le bruit de l'échantillonnage et n le nombre de plans. Quant aux Marching Cubes, leur complexité est définie ici en fonction du nombre de cubes de la grille ($cx*cy*cz*cubeSize$), et est donc en $O(cx*cy*cz*cubeSize)$.

La complexité de l'algorithme est donc majorée par la complexité de la création de la grille de voxel, soit : $\theta(\frac{cx*cy*cz}{cubeSize} * log(n))$.

4 Discussions

4.1 Discussion de l'algorithme

L'algorithme *Surface Reconstruction from Unorganized Points* de *Hoppe et al.* est un algorithme dont l'intuition n'est pas compliquée à comprendre et dont l'implémentation n'est pas spécifiquement complexe. La vitesse d'exécution de l'algorithme est lente, mais reste toute de même acceptable.

[2] Metro: measuring error on simplifed surfaces.

Cependant, les paramètres de l'algorithme que l'utilisateur doit donner en entrée ont une importance trop grande vis à vis du résultat obtenu en sortie et de la vitesse d'exécution de l'algorithme. D'un côté, laisser une grande importance à l'interaction de l'utilisateur peut permettre d'affiner simplement les résultats en fonction de ce qu'il désire, mais de l'autre, cela demande à l'utilisateur d'être expérimenté dans le domaine ou bien, à défaut, de passer beaucoup de temps sur la recherche des paramètres qui donneront le meilleur résultat en fonction de ses attentes. Concernant la précision de l'algorithme, elle reste difficilement mesurable, puisqu'elle pourra changer en fonction du nombre des plus proches voisins à prendre en compte.

4.2 Discussion sur le projet

Concernant le projet en lui-même, bien qu'il soit long pour le temps donné, c'était très intéressant de pouvoir travailler sur l'implémentation d'un algorithme décrit dans un papier de recherche, surtout en ayant la possibilité d'utiliser des librairies telles que *Eigen* ou *Boost*, ce qui nous a permis de ne pas avoir besoin de passer du temps à implémenter des choses connues et de pouvoir se concentrer sur le coeur de l'algorithme.

Le principe de n'avoir qu'un article de recherche comme sujet est à la fois une bonne et une mauvaise chose. Cela permet à la fois de pratiquer un exercice que l'on pourrait rencontrer en entreprise, mais c'est aussi un exercice compliqué au vu du manque de description de certaines parties de l'algorithme dans l'article.

Quant à notre implémentation de l'algorithme, le fichier *OFF* de sortie nous apparaît correct par rapport à ce qui est attendu. Cependant, lorsque nous chargeons le fichier résultant dans un logiciel de modélisation (*Blender* dans notre cas), la surface 3D visualisée ne correspond absolument pas au fichier d'entrée (cf Figure 1). Bien que nous ayons analysé notre code pour trouver l'origine du problème, nous avons été incapables de la trouver.

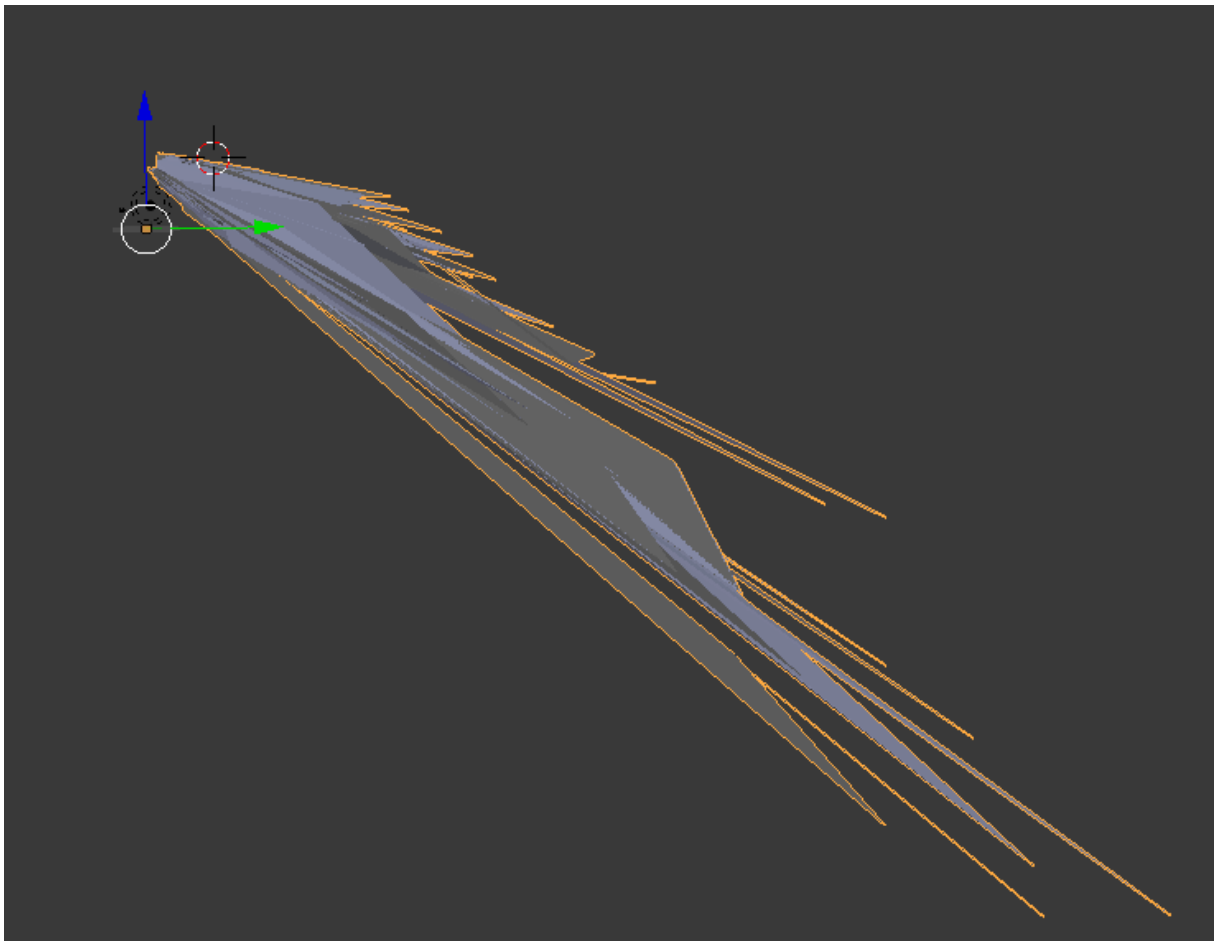

Figure 1: Résultat de l'algorithme appliqué au fichier fandisk.off visualisé sous Blender