X2012
12/01/2015

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

Détection de plans dans un nuage de points

INF552 : Analyse d'images et vision par ordinateur

Guillaume Endignoux, Gurvan L'Hostis

guillaume.endignoux@polytechnique.edu, gurvan.lhostis@polytechnique.edu

Abstract: Ce rapport décrit la méthode que nous avons développée pour détecter des plans dans des nuages de points en trois dimensions. Nous partons de nuages de points dont seules les coordonnées sont connues et nous renvoyons des sous-ensembles de points appartenant à un même plan.
Les points sont tous d'abord localisés via une structure d'*octree*, puis des plans sont recherchés localement au moyen de l'algorithme *RANSAC*. Ces plans détectés localement sont des morceaux de plans à l'échelle globale. L'idée de l'algorithme est donc de remonter la structure arborescente pour fusionner les morceaux qui correspondent au même plan. Nous utilisons une structure d'*Union-Find* pour fusionner efficacement les plans et une méthode de post-traitement pour affiner les résultats.

Keywords: Point Clouds • Plane Detection • RANSAC • Union-Find • Octree

1. Introduction

Avec le développement de technologies permettant de scanner un environnement en trois dimensions, de nouvelles perspectives s'ouvrent en matière de vision par ordinateur. Les méthodes de scan ne renvoient pas une surface mais un ensemble de points en 3D, de même qu'un appareil photo mesure des couleurs pour un ensemble de pixels discrets.

Le traitement de ces nuages de points ouvre donc de nombreuses perspectives algorithmiques. L'une d'entre elle est celle de la détection de plans. Résoudre ce problème permet en effet d'envisager le calcul d'un modèle tridimensionnel simple de l'objet observé, par exemple sous forme d'un maillage composé de faces.

Nous décrivons ici la méthode que nous avons développé pour détecter un ensemble de plans dans un nuage de points.

Entrée

Notre algorithme prend en entrée un nuage de points, sous forme de coordonnées spatiales. Ces points peuvent optionnellement avoir une couleur initiale pour faciliter la visualisation, mais cela n'a pas d'influence sur notre algorithme. Un exemple typique est un nuage de points répartis uniformément sur les façades des bâtiments d'une rue (figure 1). On envisage alors de détecter le sol de la rue, les façades, les toits, etc...

Figure 1. **Nuage de points typiquement utilisé comme entrée de l'algorithme.**

Sortie

La sortie est un nuage contenant les mêmes points mais dont on a modifié les couleurs selon le plan auxquels ils font partie. Nous donnons également la liste des plans trouvés, qui contient pour chaque plan son équation et quelques statistiques sur l'ensemble de points associés (centre, taille, épaisseur).

Aperçu de notre algorithme

Nous avons choisi de décomposer notre algorithme selon les étapes suivantes.

- Import des points et création d'une structure d'octree.
- Détection de petits morceaux de plans localement.
- Fusion de ces morceaux de plans pour obtenir des plans globaux.
- Affichage des résultats.

2. Outils utilisés par notre algorithme

Pour mettre en place cet algorithme de façon efficace, nous avons dû utiliser diverses structures de données et algorithmes. Nous présentons ici les différents outils qui composent notre méthode.

2.1. Structure d'*octree*

Effectuer des calculs sur l'ensemble du nuage de points est coûteux car les briques de base de notre algorithme ont souvent une complexité plus que linéaire. Nous avons donc choisi une approche du type diviser pour régner afin de simplifier les calculs. La division en sous-problèmes se fait en utilisant la localité des points et nous utilisons naturellement une structure d'*octree* pour gérer cette localité.

Un *octree* est un arbre qui découpe l'espace en régions parallélépipédiques nommés octants. Chaque nœud de cet arbre a huit fils car on divise en deux selon chaque direction de l'espace, soit huit sous-espaces possibles.
Nous utilisons un octree tel que tous les octants d'un même niveau de récursion ont la même dimension, c'est-à-dire que l'on coupe chaque parallélépipède selon son milieu sans tenir compte de la répartition des points.

2.2. Algorithme *RANSAC*

Les nuages de points que nous traitons sont issus du monde réel, généralement scannés par des instruments de mesure de précision limitée et/ou potentiellement basés sur des robots eux-mêmes imprécis dans leurs déplacements. Il y a également plusieurs plans différents dans le nuage, que l'on trouve y compris localement dans les coins des polyèdres. Ainsi, choisir un plan qui passe au plus près d'un sous-ensemble de points n'est pas une méthode fiable, notamment lorsqu'il y a plusieurs plans.
Nous utilisons donc la méthode du RANSAC[1], qui consiste à prendre un petit échantillon de points au hasard et tester si le plan qui lui correspond est un bon candidat pour l'ensemble des points.
La première étape consiste donc à tirer au hasard un petit échantillon et à trouver le plan qui optimise cet échantillon. Le second étape consiste à déterminer les points qui s'accordent au plan. En comptant la proportion de points s'accordant au plan, on peut décider si le plan obtenu est un plan valable. Enfin, on affine le résultat en recalculant l'équation de plan avec l'ensemble des points trouvés.
On réitère ces étapes un certains nombre de fois avec d'autres échantillons aléatoires et on choisit le meilleur plan. Ainsi, si plusieurs plans sont présents dans un sous-ensemble de points, nous avons une chance de trouver l'un d'eux.

2.3. Fusion des morceaux de plans

L'algorithme RANSAC cherche des plans dans des sous-espaces du nuage, ce qui donne une mosaïque de petits plans (figure 2).
Une étape cruciale de notre algorithme consiste donc à trouver quels sous-plans correspondent à un même plan global afin de les fusionner pour obtenir les plans complets. Pour ce faire, nous comparons les plans deux-à-deux selon trois critères : orientation, alignement et proximité.
Le premier critère est le plus évident : avant de recoller deux morceaux de plans, il faut vérifier qu'ils ont la même orientation, c'est-à-dire que leurs vecteurs normaux sont proches.
Il faut de plus que les deux morceaux soient alignés. Deux plans peuvent en effet avoir la même orientation mais être parallèles et donc décalés selon la normale. On va donc naturellement vérifier que la distance entre les deux plans est de l'ordre de leur épaisseur en calculant la distance entre un plan et le centre de l'autre plan.

[1] *Random Sampling Consensus*

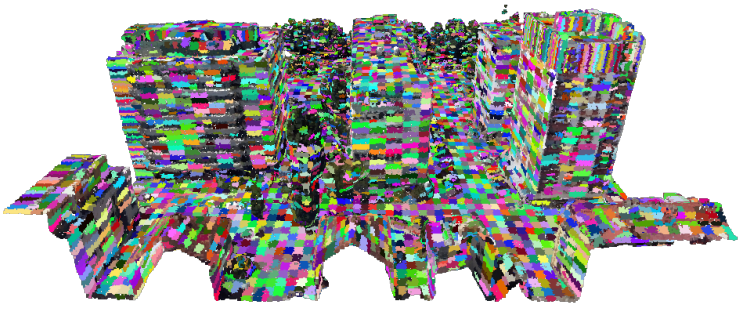

Figure 2. **Morceaux de plans détectés par le RANSAC.**

Le dernier critère, celui de proximité, consiste à vérifier que les surfaces que l'on cherche à recoller sont bien adjacentes. En effet, deux morceaux de plans peuvent être bien orientés et alignés sans former une face connexe. Par exemple, les façades de deux maisons alignées dans une rue ne doivent pas être recollées.

La méthode que nous employons pour vérifier ce critère consiste à considérer un rayon approximatif du nuage de points correspondant au morceau de plan. On compare alors l'éloignement des barycentres à la somme des deux rayons. Il est à noter qu'il s'agit d'une méthode approchée, il faudrait revenir aux coordonnées des points pour tester de façon sûre que les morceaux de plans se touchent, mais ce serait bien plus coûteux.

Nous avons également choisi d'effectuer la fusion des plans à chaque niveau de l'octree. En effet, il faut à chaque fois comparer les paires de morceaux de plans, ce qui donne une complexité quadratique. On a donc intérêt à fusionner localement pour avoir moins de plans à l'étape de fusion finale. De plus, des plans localement proches ont plus de chances d'être voisins et donc de fusionner.

2.4. Post-traitement

On ajoute à la détection par RANSAC et la fusion des plans une dernière étape pour affiner les résultats.

En effet la recherche puis fusion est efficace pour trouver les plans du nuage, mais de nombreux points sont laissés de côté à cause du caractère aléatoire du RANSAC. De plus, certains points sont dans le prolongement d'un plan trouvé dans un nœud voisin de l'octree mais n'étaient pas assez nombreux dans leur propre nœud pour constituer un plan.

L'étape de post-traitement consiste donc d'abord à parcourir l'ensemble des points encore libres pour voir s'ils ne correspondent pas à l'un des plans trouvés. Cela améliore également la stabilité de notre algorithme car nous augmentons le nombre de points dans les plans pertinents.

Nous avons aussi choisi de supprimer les plans contenant trop peu de points. En effet, les points ainsi libérés

peuvent alors être ajoutés à d'autres plans plus gros et plus pertinents. De plus, nous diminuons alors le nombre de plans et accélérons l'étape de fusion, qui est quadratique en le nombre de plans.

2.5. Calculs sur les plans

Les différents outils présentés ci-dessus nécessitent un certain nombre de calculs géométriques sur les plans.

2.5.1. Critère des moindres carrés

Un élément important de notre algorithme consiste à calculer le plan optimal passant au voisinage d'un sous-ensemble de points donné. Le critère d'optimalité que nous choisissons est celui des moindres carrés.

Pour un plan décrit par un vecteur normal unitaire $N = (a, b, c)$ et un point A du plan, on cherche à minimiser la somme des carrés des distances des points à ce plan. Ainsi, pour des points $X_i = (x_i, y_i, z_i)$, on cherche à minimiser $D = \sum_{i=1}^{n} ((X_i - A) \cdot N)^2$ sous la contrainte $||N|| = 1$. On remarque que le barycentre $A = \frac{1}{n} \sum_{i=1}^{n} X_i$ est un point de ce plan.

Si l'on note $Y_i = X_i - A$, on a alors :

$$D = N^T \left(\sum_{i=1}^{n} Y_i Y_i^T \right) N$$

N est donc un vecteur propre associé à la plus petite valeur propre de $S = \sum_{i=1}^{n} Y_i Y_i^T$.

Par ailleurs, on remarque que S vaut :

$$S = \left(\sum_{i=1}^{n} X_i X_i^T \right) - \frac{1}{n} \left(\sum_{i=1}^{n} X_i \right) \left(\sum_{i=1}^{n} X_i^T \right)$$

On choisit donc de stocker pour chaque plan associé à un ensemble de points $(X_i)_i$ la matrice M et le vecteur sum :

$$M = \sum_{i=1}^{n} \begin{pmatrix} x_i^2 & x_i y_i & x_i z_i \\ y_i x_i & y_i^2 & y_i z_i \\ z_i x_i & z_i y_i & z_i^2 \end{pmatrix} \quad sum = \sum_{i=1}^{n} \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$$

L'intérêt est que pour deux plans $\mathcal{P}_1$ et $\mathcal{P}_2$, et leur fusion $\mathcal{P}$ (au sens de l'ensemble des points associés), nous avons simplement $M = M_1 + M_2$ et $sum = sum_1 + sum_2$. Ceci nous permet de calculer rapidement les caractéristiques du plan $\mathcal{P}$, sans avoir à reparcourir tous les points.

2.5.2. Variance d'un ensemble de points

Une caractéristique intéressante est de connaître la variance d'un ensemble de points selon une direction donnée. Nous souhaitons en effet connaître l'épaisseur du plan et avoir une idée de son rayon, afin d'établir des critères de fusion de plans, ou d'ajouter des points isolés à un plan.

L'épaisseur est simplement l'écart-type de la distance selon la direction de la normale au plan. La variance vaut, pour un ensemble de points X et une direction N :

$$epaisseur = Var(X \cdot N) = Moy((X \cdot N)^2) - Moy(X \cdot N)^2 = \left(\frac{1}{n} N^T M N\right) - \left(\frac{1}{n} N^T sum\right)^2$$

Nous avons donc accès très facilement à des calculs de variance à partir de M et sum.
Pour estimer le rayon d'un ensemble de points, nous choisissons de calculer la variance selon les trois coordonnées de la base canonique et nous définissons : $rayon = \sqrt{Var(x) + Var(y) + Var(z)}$.

2.6. Structure d'*Union-Find*

Afin de gérer l'appartenance des points aux plans, nous avons choisi d'utiliser une structure d'*Union-Find*[2]. Cette structure de données permet de définir des classes d'équivalences (les plans) parmi un ensemble (les points), et de calculer efficacement la fusion de deux classes d'équivalences.
A contrario, stocker la liste des points dans chaque plan est coûteux car il faut déplacer tous les points d'un plan à un autre en cas de fusion.
Ainsi, avec la structure d'Union-Find, chaque point connaît son plan très efficacement, et ce malgré des fusions de plans successives. En revanche, on ne peut pas parcourir tous les points d'un plan (sauf en itérant sur tous les points du nuage et en testant l'appartenance à ce plan). C'est pourquoi nous utilisons la description synthétique de l'ensemble des points associés à un plan, décrite dans le paragraphe précédent.

3. Implémentation

Nous avons implémenté notre algorithme en C++. Nous avons notamment utilisé la bibliothèque OpenCV pour le calcul de vecteurs propres nécessaire dans l'optimisation d'un plan à un ensemble de points.
Le fichier `CMakeLists.txt` permet de compiler notre code avec `cmake`. Il faut alors exécuter le programme avec le fichier `Cloud.xyz` dans le même dossier. Divers fichiers `.ply` et `.planes` sont alors générés.
Nous présentons ici les classes que nous avons écrites.

3.1. Classes de base

Tout d'abord, nous avons défini quelques classes de base. Nous aurions pu prendre des classes équivalentes de OpenCV, mais nous pensions initialement tester notre méthode sans avoir besoin d'une bibliothèque extérieure comme OpenCV, afin de faciliter l'installation.

- `Vec3` : cette classe *template* définit un vecteur à trois coordonnées, ainsi que les opérations usuelles. Nous l'utilisons pour le type `double` avec la déclaration `typedef Vec3<double> Vec3d`.

[2] *http://fr.wikipedia.org/wiki/Union-Find*

- `RGB` : cette classe définit simplement une couleur par ses composantes rouge, verte et bleue.
- `Point` : initialement nous voulions attacher des propriétés aux points via une classe `Point` (par exemple la couleur), mais nous avons revu l'architecture du programme et nous définissons simplement un alias sur `Vec3d`.

3.2. Classe `PointCloud`

Nous définissons dans cette classe l'import et l'export d'un nuage de points dans un fichier. Nous utilisons notamment le format `.ply`, que nous pouvons visualiser avec MeshLab.
Cette classe permet également de calculer une *bounding box* qui est ensuite utilisée pour construire l'octree.

3.3. Classe `Octree`

Cette classe définit une structure d'*octree*. Chaque nœud de l'octree est découpé en huit parallélépipèdes de même dimensions et nous imposons une profondeur maximale d'insertion (en pratique 30), afin notamment d'éviter un récursion infinie si l'on insère deux fois le même point.
Cette classe implémente également la partie récursion de notre algorithme.

3.4. Classe `Plane`

La classe `Plane` définit un plan et enregistre les attributs suivants.

- L'équation du plan, via un vecteur normal unitaire N et un nombre réel d : $X \in \mathcal{P} \Leftrightarrow X \cdot N + d = 0$.
- Une matrice M et un vecteur sum calculés à partir des points associés à ce plan, et qui permettent de calculer l'équation du plan qui minimise le critère des moindres carrés.

$$M = \sum_{i=1}^{n} \begin{pmatrix} x_i^2 & x_i y_i & x_i z_i \\ y_i x_i & y_i^2 & y_i z_i \\ z_i x_i & z_i y_i & z_i^2 \end{pmatrix} \qquad sum = \sum_{i=1}^{n} \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$$

- Des statistiques sur l'ensemble de points associés au plan, calculées à partir de M et sum et utilisés comme critères de fusion de plans. $center = \frac{1}{n} sum$ est le barycentre des points. $thickness$ est l'écart-type de la projection selon l'axe normal au plan, et représente donc une approximation de l'épaisseur du plan. $radius$ est une approximation de la taille de l'ensemble de points, définie par $radius = \sqrt{var_x + var_y + var_z}$, où var_i est la variance selon la coordonnée i.

3.5. Classe `UnionFind`

Cette classe générique permet de définir une structure d'*Union-Find* sur l'ensemble des points. Ceci permet notamment de fusionner deux plans sans avoir à parcourir tous les points de l'un des plans.

Chaque point est alors associé à deux valeurs : sa couleur et un booléen qui indique sa présence ou non dans un plan. Ainsi, tous les points d'un même plan sont associés à la même paire de valeurs, c'est-à-dire à la même couleur.

Cette structure permet aussi de supprimer un plan, en associant un valeur invalide aux points de ce plan, qui reprennent alors leur valeur initiale après recherche dans la structure.

Ainsi, chaque point connaît sa couleur et sait s'il est dans un plan, mais nous n'avons pas besoin de stocker la liste des points dans chaque plan.

3.6. Classe `Ransac`

Cette classe définit une méthode statique qui applique l'algorithme RANSAC à un ensemble de points pour y trouver un plan. En pratique, nous l'appliquons à des ensembles d'au maximum 100 points, en partant de 10 points choisis au hasard et en cherchant un plan de minimum 30 points.

3.7. Classe `Test`

La classe `Test` permet de générer des formes géométriques afin d'élargir nos tests. Nous définissons notamment un cube, et des plans parallèles, et ce dans des directions aléatoires afin d'éviter que la géométrie ne coïncide avec les directions de l'octree.

3.8. Fichier `main.cpp`

Le fichier `main.cpp` automatise l'exécution de notre algorithme pour quelques nuages de points tests. Nous testons à la fois un nuage de points issu de données réelles représentant des immeubles et des rues, et des nuages de points artificiels représentant des formes géométriques simples (plans, cubes).

L'exécution attend un fichier `Cloud.xyz` contenant les coordonnées et les couleurs initiales du nuage de point réel. En sortie, pour chaque nuage de points, un fichier `.ply` est généré, où chaque plan trouvé a une couleur aléatoire, ainsi qu'un fichier `.planes` contenant la liste des plans.

4. Résultats

4.1. Influence des paramètres

Les paramètres qui influent sur le résultat sont de deux types.

- D'une part, les paramètres utilisés pour le *RANSAC* : taille du sous-ensemble dans lequel on détecte les points, nombre de points choisis pour initialiser un plan, nombre de points nécessaires dans un plan admissible. Nous avons choisi de prendre des sous-ensembles d'au plus 100 points, d'initialiser les plans avec 10 points et d'avoir au moins 30 points dans un plan. De plus, nous définissons un critère d'épaisseur (distance maximale des points qui sont acceptés dans le plan). Pour cela, afin d'avoir une méthode invariante

par changement d'échelle, nous estimons le rayon du sous-ensemble de points à traiter, et définissons une épaisseur proportionnelle (typiquement $e = 0.05r$).

- D'autre part, les paramètres de fusion et de post-traitement. Par exemple, la proximité entre deux plans s'exprime comme la distance entre leurs centres, et la limite de proximité est proportionnelle aux rayons des morceaux de plans, mais le coefficient de proportionnalité entre les deux peut être affiné. De même le critère de suppression des petits plans est une borne qui dépend du nombre de points dans le morceau le plus peuplé, mais cette fonction peut aussi être ajustée.

4.2. Nuage de points réel

Nous avons d'abord travaillé sur le fichier `Cloud.ply` fourni au début de notre projet. Nous obtenons le résultat suivant (figure 3) avec notre choix de paramètres, pour un temps d'exécution de l'ordre d'une minute.

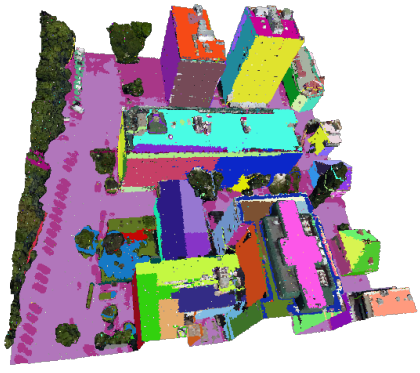

Figure 3. **Sortie de l'algorithme pour le nuage de points réel.**

On peut remarquer que la plupart des façades ont été reconnues et que les arbres à gauche n'appartiennent à aucun plan. Il reste quelques artefacts à certains endroit et il faudrait utiliser une méthode de post-traitement plus poussée pour les éliminer.

Nous avons également souhaité travailler sur d'autres nuages de points réels, mais après une recherche approfondie, il s'avère difficile de trouver des nuages intéressants. En effet, de nombreux fichiers ne contenaient pas de plan, par exemple ceux issus d'un scan tridimensionnel d'une œuvre d'art comme une sculpture. Une autre nécessité est d'avoir des points suffisamment denses, mais un nombre global de points raisonnable du point de vue du temps de calcul. Enfin, nous avons rencontré de nombreux fichiers aux formats binaires ou propriétaires, dont l'importation n'est pas triviale.

Nous avons donc choisi de tester notre algorithme sur des formes géométriques simples et connues, afin d'avoir

d'autres données pour valider notre méthode.

4.3. Formes géométriques

Le premier type de test géométrique est un ensemble de plans carrés composés de points disposés selon un quadrillage régulier. Ces plans ont des directions générées aléatoirement et sont centrés sur l'origine (ce qui implique de nombreuses intersections). Avec dix plans, nous obtenons effectivement dix plans principaux, ainsi que quelques groupes de points isolés qui n'ont pas été fusionnés, ce qui est plutôt bien étant données les nombreuses intersections à l'origine du repère (figure 4).

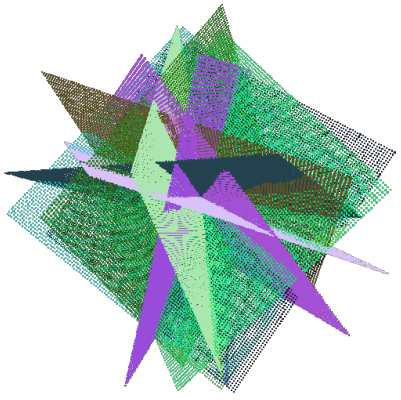

Figure 4. **Sortie de l'algorithme pour 10 plans aléatoires.**

Le second type de test est formé de cubes, dont les directions sont également aléatoires (notamment afin de tester la robustesse vis-à-vis de plans dont la direction ne suit pas l'orientation de l'octree). Nous avons par exemple généré trois cubes imbriqués de côtés respectifs 20, 50 et 80 et nous obtenons effectivement six plans par cube (figure 5).

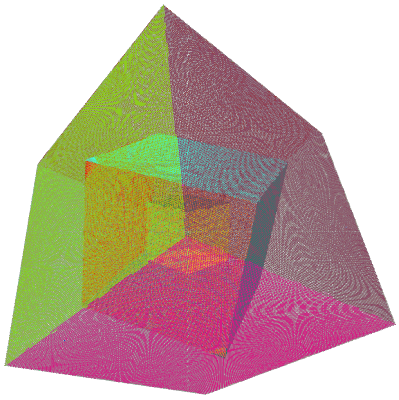

Figure 5. **Sortie de l'algorithme pour 3 cubes aléatoires.**

Le troisième type de test est composé de deux plans parallèles et proches, et nous obtenons également des résultats concluants.

5. Améliorations possibles

Nous présentons ici quelques axes d'amélioration de notre méthode.

5.1. Fusion des plans

Un point important de notre algorithme est qu'à chaque nœud de l'octree, nous cherchons à fusionner les plans trouvés récursivement. A partir d'une liste de plans obtenus sur les nœuds fils, nous comparons actuellement toutes les paires de plans, ce qui est relativement coûteux (quadratique selon le nombre de plans trouvés).
Or, l'une des conditions de fusionnabilité est l'angle entre les vecteurs normaux. Nous pourrions donc améliorer la recherche de paires de plans fusionnables en utilisant une structure de données adaptée qui permet, étant donné un vecteur normal, de trouver tous les plans dont le vecteur normal est proche, sans avoir à parcourir toute la liste.
Nous pouvons par exemple définir une distance à partir de l'angle entre deux directions, puis utiliser des algorithmes génériques de recherche de voisins dans un espace métrique.

5.2. Test de proximité probabiliste

Le test de proximité des plans que l'on effectue est approximé, car nous utilisons un critère global qui est le rayon du morceau de plan. Mais ce critère de proximité n'est pas toujours pertinent, par exemple dans le cas de rectangles beaucoup plus longs que larges (un autre plan situé dans le sens de la largeur risque de rentrer dans le

rayon sans pour autant être connexe au premier).

Un méthode exacte consisterait à parcourir l'ensemble des points des deux plans jusqu'à trouver des paires proches, ce qui serait calculatoirement inefficace.

Cependant, il serait possible de développer un critère probabiliste. En effet, deux morceaux de plans adjacents ont un nombre important de paires de points proches, et en tirant au hasard un certain nombre de paires de points on peut décider de façon probabiliste si deux plans sont adjacents ou non. On pourrait utiliser les propriétés de l'octree pour guider le tirage des points afin de trouver des paires proches.

5.3. Extraction de frontières de plans

Le calcul de frontières explicites pour les plans que nous détectons permettrait :

- d'évaluer la proximité entre deux morceaux de plans en calculant la distance entre les segments qui font la frontière, afin de décider si les morceaux sont fusionnables ;
- de donner des informations intéressantes sur le nuage de points en lui-même, par exemple pour créer un maillage a partir des plans.

Pour ce faire, on pourrait calculer l'enveloppe convexe des points (projetés sur le plan en question), par exemple en utilisant l'algorithme du parcours de Graham qui donne le résultat en temps $O(n \log n)$ où n est le nombre de points.

Cependant, nous avons fait le choix dans notre méthode de ne jamais en revenir à un parcours des points pour des raisons de temps de calcul, une telle application ne pourrait donc avoir lieu qu'en post-traitement final et serait sans-doute inefficace pour gérer les fusions de plans.