# Optimisation du traitement de nuages de points 3D organisés en maillage

Damien Marié

Automne 2013

Rapport TX n ° 4766 pour le laboratoire Heudiasyc

Supervisé par Gérald Dherbomez et Vincent Frémont

# Introduction

Ce document résume et analyse le travail que j'ai effectué durant le semestre d'Automne 2013 sur le projet TX « Optimisation du traitement de nuages de points 3D organisés en maillage » pour le laboratoire Heudiasyc de l'UTC. Le travail à consisté à traiter le nuage de points de sortie d'un scanner 3D, le Velodyne HDL64E, qui est utilisé sur les voitures intelligentes d'Heudiasyc.

Ce travail s'est précisé au fur et à mesure du projet pour au final se concentrer sur 3 points :

— Réduction non-uniforme de la densité du nuage
— Triangulation du nuage de points
— Détection de la surface correspondant au sol

# Remerciements

Je souhaite remercier Gérald Dherbomez et Vincent Frémont qui m'ont encadrés durant tout le projet.

Je souhaite aussi remercier le laboratoire Heudiasyc qui m'as chaleureusement acceuilli et aidé durant tout ce projet.

# Table des matières

# 1 Le projet

Le laboratoire Heudiasyc est le laboratoire d'informatique de l'UTC. Il développe une voiture qui à pour objectif d'être autonome. Dans ce cadre, ils ont acquis beaucoup de capteurs afin de receuillir des informations en continue sur l'environnement. Le plus imposant de ces capteurs est le Velodyne HDL64E, un capteur 3D omnidirectionnel qui scanne l'environnement.

Plusieurs tentatives ont déjà permis de commencer à utiliser ce capteur : Visualisation des données, re-calibrage des données en sortie et enregistrement dans un format standard.

L'objectif est donc de continuer les développements dans ce sens et ainsi rendre le capteur opérationnel et utilisable par les chercheurs du laboratoire.

**Le sujet** Un maillage est un ensemble de points, d'arêtes et de faces qui définissent la forme d'un objet. Cette représentation peut être utilisée à des fins de simulation ou de représentation graphique. Ces objets peuvent nécessiter des ressources importantes en termes de stockage ou présenter une complexité trop grande selon l'échelle à laquelle ils sont étudiés.

Les capteurs télémétriques laser fournissent un nuage de points 3D permettant de modéliser l'environnement d'un véhicule justement sous la forme de maillage en 3 dimensions. Le capteur Velodyne HDL64E utilisé par exemple par la société Google pour la navigation des ses véhicules autonomes "driverless Google car", fournit ce genre d'informations.

Bien souvent, ces informations sont très riches (le capteur Velodyne fournit 1,3 millions de points 3D à la seconde) et donc difficiles à traiter en temps réel. Ainsi, dans le cadre de ce sujet, nous proposons d'étudier une approche multi-résolution permettant de limiter la complexité du maillage 3D et ainsi réduire les temps de calculs. Plus précisément, les approches multi-résolutions consistent à représenter un maillage en niveaux de détails de complexité croissante, afin de pouvoir accéder facilement aux informations jusqu'à l'échelle souhaitée.

Dans le cadre de la représentation d'objets 3D, ce genre d'approche permet par exemple de ne pas considérer les détails d'un objet distant. L'objectif du projet consistera donc à appliquer une méthode multi-résolution sur

le nuage de points 3D fournit par le capteur Velodyne installé sur le toit du véhicule. La méthode sera en premier testée en post-traitement sur des données réelles enregistrées. Le portage en temps réel dans le véhicule sera un plus pour la réussite du projet.

**Concrètement** Le projet s'est axé sur 3 points :

- **Réduction non-uniforme de la densité du nuage :**
  Afin de pouvoir traiter le nuage de façon plus efficace sans sacrifier l'entropie des données, il nous faut réduire le nuage (comptabilisant plus de 100 000 points) à quelques dizaines de milliers de points. De plus, cette réduction, afin de garde un maximum de détails sur les zones importantes, doit pouvoir se faire de façon non-uniforme. Ainsi, une zone aura une densité supérieure à une autre par exemple.
- **Triangulation du nuage de points :**
  Un objectif du scanner 3D monté sur la voiture et de reconstituer son environnement à des fin de visualisation mais surtout pour pouvoir appliquer des algorithmes de navigation dans l'espace afin d'obtenir une voiture autonome par la suite.
- **Détection de la surface correspondant au sol :**
  La détection du sol permet d'ajouter à la triangulation l'information de l'emplacement de la route

Cependant, des travaux auxiliaires était aussi posés :

- **Intégration dans le framework PACPUS :**
  Afin que le travail soit pérenne, une intégration dans le framework PACPUS qui sert à centraliser tout les efforts de développement concernant la voiture intélligente aurait été un plus.
- **Traitement temps réel :**
  L'objectif final étant d'utiliser les algorithmes dans la voiture en temps réel, chacun de ceux soit doivent être le plus performant possible. Une latence supérieur à 100ms étant envisageable pour une intégration en environnement temps réel.
- **Documentation :**
  Toujours afin de pérenniser le travail, un objectif auxiliaire a été de documenter et de publier mon travail afin qu'il puisse être réutiliser, même au delà du laboratoire Heudiasyc.

# 2 Réduction non-uniforme de la densité du nuage

## 2.1 Approche

Le surplus de point venant essentiellement de groupes de points extrêmement proches entourés de vide. L'approche choisie à été de découper en grille 3D contenant autant de cubes que nous voulons le nuage filtré et simplifier chaque cube par son barycentre.

De façon plus détaillée, voilà comment la réduction de densité se fait :

1. On se donne une densité finale désirée qui correspond à la taille des cubes (appelés feuilles)
2. On calcul le barycentre de tout les points contenus dans ce cube
3. On remplace tout ces points par ce barycentre

L'avantage de cette méthode est d'abord sa rapidité qui la rend compatible avec un traitement temps réel (<10ms) mais surtout sa flexibilité pour un passage à la réduction non-uniforme.

Pour passer à la réduction non-uniforme de la densité, on peut envisager beaucoup de méthodes selon le résultat désiré :

- Procédé par division de chaque cube progressive : On obtient des cubes de plus en plus petits
- Avoir plusieurs grilles ayant chaque sa densité prédéfinie
- Avoir des grilles ne contenant par des cubes mais de parrallépipéde pour favoriser l'information en hauteur ou en largeur par exemple

Cela permet, par exemple, d'avoir une résolution qui s'adapte à la distance à la caméra ou encore par rapport à l'avant ou l'arriére de la voiture. Ou encore par rapport à la distance à la voiture.

D'autres approches possibles :

- Pour chaque point, rechercher tout les points qui sont inférieurs à une certaine distance du point et les supprimés.
  Ce procédé est très rapide mais peu fidèle. Il permet cependant l'application la plus demandée très simplement : L'ajustement de précision par rapport à la distance et la position.

— Prendre aléatoirement un nombre prédéfini de point et les supprimés. Ce procédé toujours très rapide (surtout sur des structures de données non ordonnées), permet de définir à l'avance le nombre de point voulu ou la densité désirée très simplement. Cependant elle est trés peu fidèle.

## 2.2 Implémentation

L'implémentation à été trés simple de par l'utilisation de la librairie PCL et du filtre VoxelGrid intégré, voici un exemple démontrant comment celui-ci à été utilisé dans le projet :

```
1  pcl::PointCloud<pcl::PointXYZ>::Ptr downsample_cloud(pcl::
       PointCloud<pcl::PointXYZ>::Ptr cloud){
2      pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::
           PointCloud<pcl::PointXYZ>());
3
4      //Downsample
5      pcl::VoxelGrid<pcl::PointXYZ> sor;
6      sor.setInputCloud (cloud);
7      sor.setLeafSize (0.1f, 0.1f, 0.1f);
8      sor.filter (*cloud_filtered);
9
10     //-> PointCloud before filtering: 133312 data points .
           PointCloud after filtering: 48027 data points.
11
12     return cloud_filtered;
13 }
```

Source 2.1 – widgetPCL.cpp

Comme on peut le voir, le filtre VoxelGrid de PCL permet d'avoir des voxel parralépipéde, ce qui permet de favoriser la conservation d'information dans un dimension. Le choix de 0.1cm est purement arbitraire, il est rapide tout en conservant une grande partie des points.
Voici un exemple d'exécution :

```
Points avant filtrage: 133312
Points après filtrage:  48027
```

On constate donc une réduction de **64%** de la taille du nuage. Mais surtout, on constatera ensuite une vitesse d'éxécution beaucoup plus rapide pour les algorithmes de triangulation et de détection de plan.

## 2.3 Améliorations possible

Tout d'abord, l'approche multi-résolution/non-uniforme n'as pas été implémentée par manque de temps. Celle-ci consisterait en un simple VoxelGrid modifié qui tiendrai en compte de la distance de la caméra.

De plus, aucun test d'intégration dans un environnement temps réel n'as été réalisé, afin de percevoir si les performances de l'implémentation permettent cette intégration.

Et pour finir, développer plusieurs approches serait préférable afin de savoir laquelle serait la plus performante, surtout dans le cas d'un besoin de densité/résolution non-uniforme.

# 3 Triangulation du nuage de points

## 3.1 Approche

La triangulation du nuage de point obtenu fut un point de blocage important durant le projet et beaucoup de recherches ont été faites afin d'obtenir des résultats utilisables. Cependant aucune approche concluante n'as put être trouvée.

Je développerai ici, deux grandes approches utilisées, mais d'autres encore ont été utilisées.

**Greedy projection** L'approche naïve consiste à utiliser la greedy projection. C'est une méthode très rapide mais très peu robuste qui va juste connecter les points ensemble dés qu'ils respectent une certaine distance.

**Surfaces de poisson** Cette fois-ci, une approximation est faite par des surfaces. C'est une méthode beaucoup plus lente mais avec laquelle ont obtiens des résultats très lisse.

Cependant chacune de ces méthodes utilisaient une **estimation des normales** préliminaires, celle-ci échouant forcément sur les données du velodyne qui sont très peu dense (entre les grandes courbes des lasers). Ce qui à rendu les résultats catastrophiques.

De plus, afin d'harmoniser la densité des données, une autre passes préliminaires à été essayée, le **MLF** (Moving Least Squares). Mais cela n'as que très peu affecté les résultats.

## 3.2 Implémentation

### 3.2.1 Greedy projection

On utilise directement la classe GreedyProjectionTriangulation de PCL et une estimation de normales par NormalEstimation, ce qui rend le travail très simple.

```
1  // Normal estimation
2  pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> n;
```

```
3   pcl::PointCloud<pcl::Normal>::Ptr normals (new pcl::PointCloud<
        pcl::Normal>);
4   pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::
        KdTree<pcl::PointXYZ>);
5   tree->setInputCloud (cloud);
6   n.setInputCloud (cloud);
7   n.setSearchMethod (tree);
8   n.setKSearch (20);
9   n.compute (*normals);
10
11  // Concatenate the XYZ and normal fields
12  pcl::PointCloud<pcl::PointNormal>::Ptr cloud_with_normals (new
        pcl::PointCloud<pcl::PointNormal>);
13  pcl::concatenateFields (*cloud, *normals, *cloud_with_normals);
14
15  // Create search tree
16  pcl::search::KdTree<pcl::PointNormal>::Ptr tree2 (new pcl::
        search::KdTree<pcl::PointNormal>);
17  tree2->setInputCloud (cloud_with_normals);
18
19  // Initialize objects
20  pcl::GreedyProjectionTriangulation<pcl::PointNormal> gp3;
21  pcl::PolygonMesh triangles;
22
23  // Set the maximum distance between connected points (maximum
        edge length)
24  gp3.setSearchRadius (1);
25
26  // Set typical values for the parameters
27  gp3.setMu (2.5);
28  gp3.setMaximumNearestNeighbors (10);
29  gp3.setMaximumSurfaceAngle(M_PI/4); // 45 degrees
30  gp3.setMinimumAngle(M_PI/18); // 10 degrees
31  gp3.setMaximumAngle(2*M_PI/3); // 120 degrees
32  gp3.setNormalConsistency(false);
33
34  // Get result
35  gp3.setInputCloud (cloud_with_normals);
36  gp3.setSearchMethod (tree2);
37  gp3.reconstruct (triangles);
```

Source 3.1 – test_meshing.cpp

### 3.2.2 Surfaces de poisson

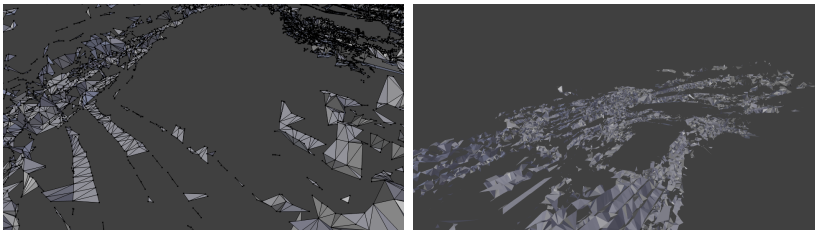
Ici, on utilise MovingLeastSquares, NormalEstimationOMP et Poisson afin de combiner Moving Least Squares et Poisson.

```
1   PointCloud<PointXYZ>::Ptr cloud (new PointCloud<PointXYZ>
        ());
2   io::loadPCDFile (argv[1], *cloud);
3
4   MovingLeastSquares<PointXYZ, PointXYZ> mls;
5   mls.setInputCloud (cloud);
6   mls.setSearchRadius (0.1);
7   mls.setPolynomialFit (true);
8   mls.setPolynomialOrder (2);
9   mls.setUpsamplingMethod (MovingLeastSquares<PointXYZ,
        PointXYZ>::SAMPLE_LOCAL_PLANE);
10  mls.setUpsamplingRadius (0.05);
11  mls.setUpsamplingStepSize (0.02);
12
13  PointCloud<PointXYZ>::Ptr cloud_smoothed (new PointCloud<
        PointXYZ> ());
14  mls.process (*cloud_smoothed);
15  NormalEstimationOMP<PointXYZ, Normal> ne;
16  ne.setNumberOfThreads (8);
17  ne.setInputCloud (cloud_smoothed);
18  ne.setRadiusSearch (0.03);
19  Eigen::Vector4f centroid;
20  compute3DCentroid (*cloud_smoothed, centroid);
21  ne.setViewPoint (centroid[0], centroid[1], centroid[2]);
22  PointCloud<Normal>::Ptr cloud_normals (new PointCloud<
        Normal> ());
23  ne.compute (*cloud_normals);
24  PointCloud<PointNormal>::Ptr cloud_smoothed_normals (new
        PointCloud<PointNormal> ());
25  concatenateFields (*cloud_smoothed, *cloud_normals, *
        cloud_smoothed_normals);
26
27  Poisson<PointNormal> poisson;
28  poisson.setDepth (9);
29  poisson.setInputCloud
30  (cloud_smoothed_normals);
31
32  PolygonMesh mesh;
33  poisson.reconstruct (mesh);
```

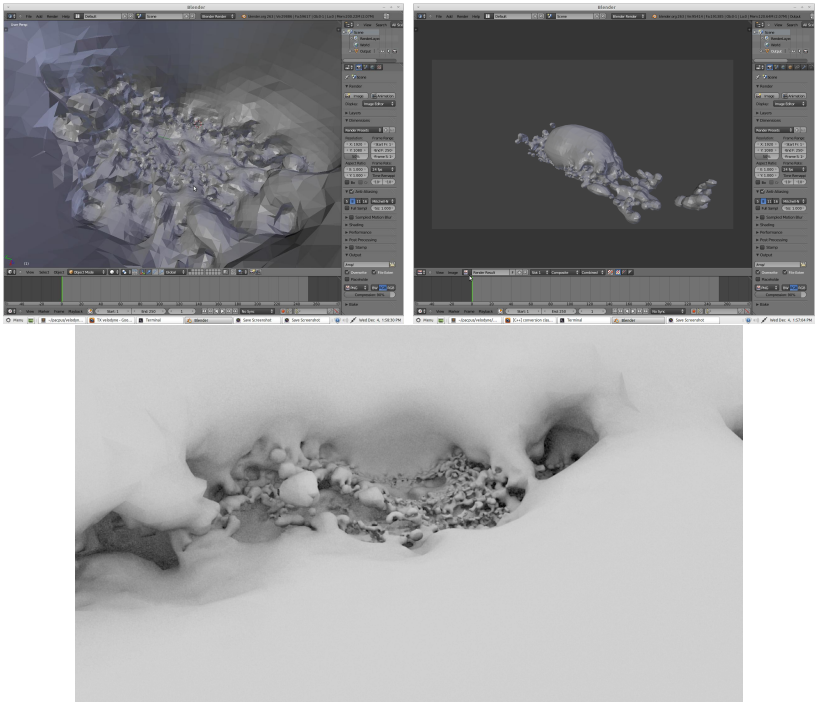
Source 3.2 – test_poisson.cpp

## 3.3 Illustration des résultats

### Greedy Projection



**Surfaces de Poisson**



## 3.4 Améliorations possible

Les données d'entrée semblent être le point bloquant. Il n'y a pas assez d'informations sur un seul nuage pour pouvoir trianguler.
La solution envisagée qui semble être la plus performante et facile à implémenter est d'ajouter au fur et à mesure des nuages, les concaténer ensemble, au fur et à mesure que la voiture avance. Ainsi, on obtient une meilleure répartition des données et la triangulation devrait être beaucoup plus facile.

Cela n'as pas put être fait par moi même du fait de la nécessité de recapturer ces données avec des données de positionnement (vitesse, odométrie, GPS,...). Mais cela est la voie à explorer pour quelqu'un souhaitant continuer

mon travail.

On peut aussi remarquer que peut importe la méthode, la triangulation prenant souvent plus de 20 secondes, celle-ci ne peut être fait en temps réel, même avec la réduction du nombre de points fait précédemment.

# 4 Détection de la surface correspondant au sol

## 4.1 Approche

Afin de détecter le sol, nous faisons l'approximation que celui-ci est une surface plane. Ainsi, nous détectons un plan plutôt qu'une courbe (comme cela à été démontré très difficile dans la partie triangulation).

Ensuite, on utilise une méthode très répandu pour découvrir des plans : La découverte aléatoire par prise d'échantillons. Voilà un fonctionnement simplifié :

1. On donne un seuil de tolérance pour l'adaptation au modèle
2. On fait ces étapes un nombre N (grand) de fois :
   (a) On prend 3 points aléatoirement dans le nuage
   (b) On a donc un modèle de plan : ax + bx + cx + d = 0
   (c) On regarde combien de points correspondent au modèle (en prenant en compte le seuil de tolérance)
   (d) Si ce nombre est supérieur à un certain pourcentage, il y a bien un plan et on a l'équation

## 4.2 Implémentation

L'implémentation fut tout aussi simple car PCL intégre un estimateur pour la détection de plan, SACSegmentation avec SACMODEL_PLANE (et SAC_RANSAC comme méthode).

```
1  pcl::ModelCoefficients::Ptr coefficients (new pcl::
       ModelCoefficients);
2  pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
3  pcl::SACSegmentation<pcl::PointXYZ> seg;
4  seg.setOptimizeCoefficients (true);
5  seg.setModelType (pcl::SACMODEL_PLANE);
6  seg.setMethodType (pcl::SAC_RANSAC);
```

```
7   seg.setDistanceThreshold (0.01);
8   seg.setInputCloud ((*cloud).makeShared ());
9   seg.segment (*inliers, *coefficients);
10  std::cout << "Model coefficients: " << coefficients->values[0]
        << "x +" << coefficients->values[1] << "y +" <<
        coefficients->values[2] << "z +" << coefficients->values[3]
        << " = 0 " << std::endl;
```

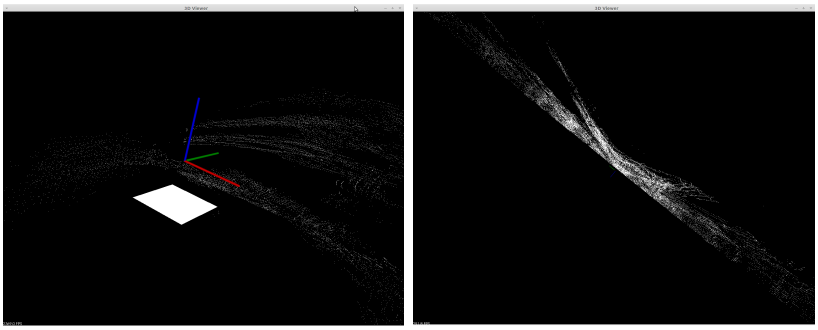
Source 4.1 – test_floor_detection.cpp

Voici un exemple d'éxécution :

```
Model coefficients: -0.0165034x + -0.0111194y + 0.999802z + 0.726204 = 0
Model inliers: 345
```

On constate aussi que l'estimation est plutôt rapide (<400ms) même si cela ne correspond toujours pas à une performance temps réel.

## 4.3 Illustration des résultats



## 4.4 Améliorations possible

L'estimation est assez grossière et plusieurs mesures permettraient d'en améliorer la qualité et la robustesse :

- Forcer les paramètres à certaines limites : Pas de plan vertical, pas d'inclinaison trop grande, .... Ainsi, un plan serait pas détecter lorsque l'on passe à côté d'un immeuble par exemple
- Éliminer les points aberrants et les points correspondants à la voiture, cela donnerait une bien meilleure qualité à notre approximation de plan
- Exécuter l'algorithme sur un sous-ensemble petit de point afin d'améliorer les performances et ainsi avoir une compatibilité temps réel

# 5 Travaux auxiliaires

**Intégration dans PACPUS** L'intégration dans PACPUS n'as pas été réalisée de par le faible apport de mes travaux mais aussi de par la transition que PACPUS fait aujourd'hui ce qui du code vite obsolète. J'ai cependant publié mes travaux sur GitHub afin qu'ils puissent être repris.

**Documentation** De même une documentation à été réalisé sur le projet publié afin de faciliter la reprise des travaux et la compréhension par une personne extérieur à Heudiasyc. Cette documentation ainsi que les codes source produits sont disponibles à l'adresse `https://github.com/MDamien/velodyne`

# 6 Points techniques

**Techniquement,** le framework développé par Heudiasyc, PACPUS, fut difficile à appréhender de part de manque de documentation mais surtout de stabilité. Il m'as été difficile de reprendre le code de l'élève précédent. Cela est dut au peu de temps et de moyens que l'UTC accorde à la maintenance de ses codes source, de l'intégration hâtive par les différents doctorants,...
J'ai cependant put très bien travaillé une fois que l'environnement été prêt. Mais il serait difficile de le reproduire sur une nouvelle machine.
Le passage en open source de PACPUS changera sûrement la donne avec des collaborations à longue distance et donc une documentation et surtout une maintenance du code qui sera forcément plus rigoureuse.
De plus, j'étais un des seul à développer sur Linux (la plupart étaient sur Windows), ce qui à entraîner une certaine difficulté à me faire aider par les autres personnes travaillant avec moi.
Enfin, PCL est une librairie très scientifique dont j'ai dut déchiffrer beaucoup de vocabulaire avant de pouvoir l'utiliser, mais elle a été très facile à utilisée ensuite.

# Conclusion

Les travaux produits sont, à mon avis, de très bonne qualité. Le développement sur Velodyne / PACPUS étant relativement difficile, il m'aurait fallu plus de temps et des objectifs très clair dés le début pour produire quelque chose de conséquent. C'est pourquoi je conseille plutôt de confier de tel travaux à des doctorants ou des projets de fin d'études afin d'avancer beaucoup plus efficacement.

J'ai appris beaucoup beaucoup techniquement (C++/PCL, algorithmes complexes, frameworks complexes) mais aussi du point de vue organisationnel. Voir comment un laboratoire tel qu'Heudiasyc travaille à été très enrichissant.