Travail de Bachelor LiDAR

He-Arc

Professeur Encadrant : François Tièche

Mandants : Fabien Droz, Christophe Pache, secteur « Time and Frequency » de la division « Systems » du CSEM

Auteur : Maxime Piergiovanni

Abstract

Ce rapport décrit l'élaboration d'une application de visionnement de données tridimensionnelles.

Ces données proviennent de caméras dites « LiDARs » (acronyme de Light Detection And Ranging) qui peuvent être comprises comme des sonars mais utilisant la lumière. De la même manière qu'un sonar va émettre une onde sonore puis attendre le retour de cette onde pour comprendre la topologie des lieux ou détecter un obstacle, le Lidar va émettre un flash de lumière et calculer le temps qu'il a fallu aux photons émis pour rebondir sur un obstacle et revenir au capteur.

Nous allons suivre dans ce rapport toutes les étapes qui ont été nécessaires à la visualisation de telles données. La structure du rapport suit les données dans leur cheminement à travers le programme.

Nous nous intéresserons à la lecture des données brutes, notamment les problèmes de mémoire qui sont soulevés lorsque les données sont de grande taille.

Nous passerons ensuite à une explication de la bibliothèque de traitement et de visualisation des données VTK, les données doivent en effet être transférées dans un format qui convient à la bibliothèque.

Toujours dans le cadre de cette bibliothèque, sera discuté la manière de traiter, filtrer, modifier les données pour influencer sur ce que l'utilisateur va voir.

Enfin, nous discuterons de la visualisation en elle-même, des images produites, ainsi que des divers outils qui permettent la navigation et l'interprétation de ces données de manière plus claire pour un utilisateur. Cette partie ouvrira une discussion sur l'interfaçage avec l'utilisateur et la difficulté d'ancrer des données abstraites dans le réel, de leur donner une signification.

Table des matières

Contents

Introduction

Contexte

Le projet a été proposé par le secteur « Time and Frequency » de la division « Systems » du CSEM. L'équipe cherchait à mettre en valeur les données obtenues par leur caméra LiDAR, notamment dans un but d'explication et de promotion de la caméra.En effet, il est compliqué d'expliquer ce que fait le capteur Lidar à une personne ne venant pas d'un milieu technique ou scientifique. Il était donc important de pouvoir présenter les données résultantes de manière visuellement attrayante, pour pouvoir illustrer les capacités de la caméra auprès de journalistes ou d'acheteurs potentiels.

Problématique

Les objectifs du projet étaient de proposer de belles visualisations des données fournies par le lidar. Pour cela, nous nous sommes mis d'accord sur une visualisation à base de nuages de points. L'un des premiers objectifs était de produire un nuage de points à partir d'une image de profondeur 2D. Ci-dessous une illustration du passage de la carte de profondeur à une représentation tridimensionnelle.

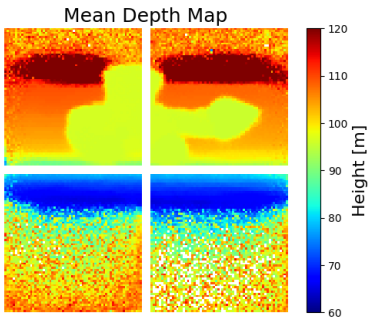

Figure 1 : Image de profondeur 2D

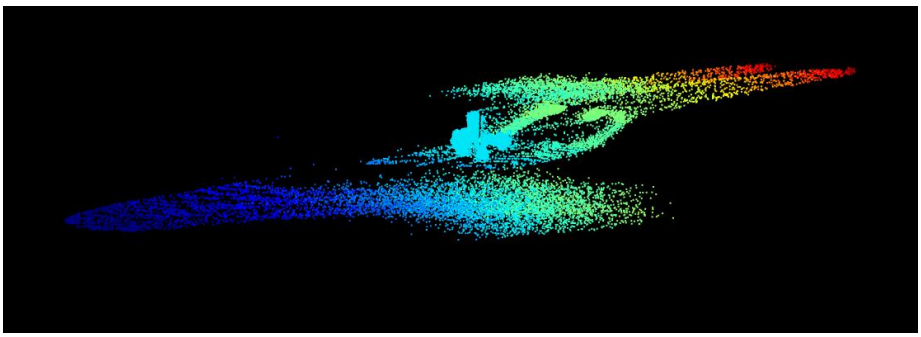

Figure 2 ; Nuage de point en 3D

Dans l'optique d'obtenir une meilleure visualisation, il était nécessaire de donner à l'utilisateur des outils qui lui permettraient de nettoyer ses données, de filtrer les indésirables, afin de clarifier le nuage de point et accéder aux données ayant le plus de sens.

Il était aussi important de pouvoir colorer les points pour leur assigner plusieurs informations. Une fois placés dans l'espace, nous voulions pouvoir avoir une indication quant à d'autres informations liées à ce point dans l'espace.

D'autres points importants du programme étaient primordiaux :

- Pouvoir naviguer de manière intuitive autour des données.
- Pouvoir facilement exporter ce que l'on visionne dans un format d'objets 3D standard.

Méthodologie

Pour attaquer ce problème, il a fallu dès le départ chercher des outils appropriés à la tâche. Voici un bref retour sur ces choix, qui seront discutés plus en profondeur plus tard.

Le programme est écrit en langage Python. Ce langage s'est imposé de par le format dans lesquelles nous étaient fournies les données, des archives Numpy, ainsi que la compétence préalable de l'équipe du CSEM à utiliser Python. Il est en effet plus facile pour eux de reprendre le projet une fois terminé s'il est écrit dans ce langages qu'ils maitrisent.

Le traitement des données et la visualisation se fait avec la bibliothèque VTK en utilisant sont wrapper Python. VTK est une bibliothèque bien établie, très utilisée dans le monde académique et très rapide car écrite en C++ avec OpenGL.

L'interface Utilisateur se fait avec PyQt car le duo VTK / Qt est très facile à intégrer, Qt est par la même la bibliothèque de GUI la plus aboutie pour Python.

Structure du rapport

Le rapport va dans sa structure suivre le même chemin que les données que nous devons traiter dans le cadre de ce projet. Nous allons ainsi commencer par nous intéresser à leur lecture, avant de passer à leur traitement et leur filtrage. Il sera ensuite question de la meilleure manière de les mettre en valeur et de donner à l'utilisateur des outils qui lui permettront de donner du sens à ce qu'il voit. Enfin, nous parlerons de l'intégration de l'outil de visualisation à une interface graphique permettant d'influer en temps réel sur les différents filtres et les éléments de visualisation.

Analyse

L'avant-projet

Les anciens outils

Les outils de visualisations utilisés avant ce projet étaient divers, notamment des images 2D représentant diverses informations sur les données. Ci-dessous des exemples :

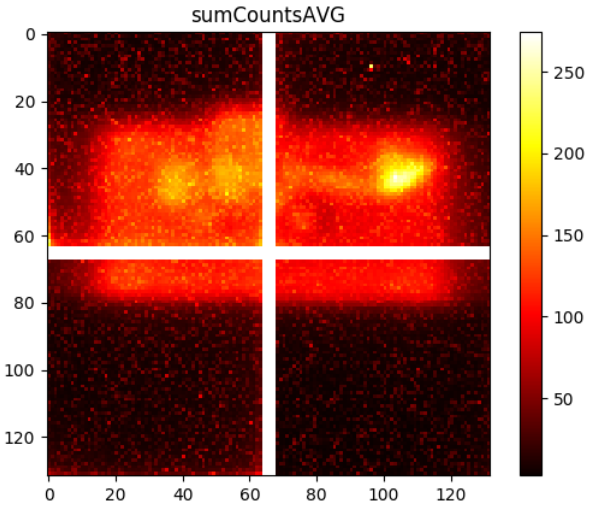

Figure 4 : Photons reçus par capteurs

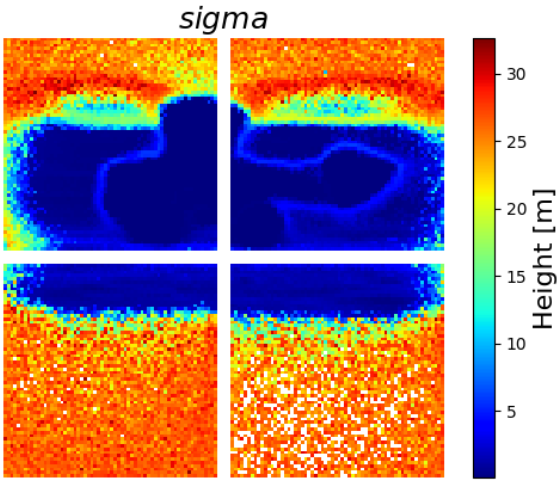

Figure 3 : sigma calculé sur N images

Les données desquelles sont tirées ces images sont celles d'une tractopelle dans un champ. Sur la Figure 1, on a interprété les données pour essayer de trouver à quelle profondeur se trouvait l'obstacle qui a renvoyer le plus de photons. A gauche on indique le nombre total de photons qu'un capteur (pixel) a reçu. A droite, on fait la différence moyenne sur une centaine d'images de profondeurs prises à la suite.

Sans entrer dans l'interprétation de ces résultats, on peut en effet comprendre qu'il est difficile de se faire une représentation tridimensionnelle de la tractopelle à partir de ces images.

L'équipe du CSEM utilisait aussi un logiciel de visionnement 3D nommé mayaVi, néanmoins, ils étaient insatisfaits du manque de possibilité quant au traitement des données en temps réel. Il n'est en effet pas commode de traiter ses données dans un programme, puis de visualiser les données dans un autre programme.

Format des données

Le capteur Lidar fonctionne comme suit : dans un premier temps, il émet un flash de lumière. Ensuite, les capteurs vont compter le nombre de photons reçus dans un certain lapse de temps (de l'ordre de quelques dixièmes de nano secondes dans notre cas). La caméra Lidar est composée de capteur organisé en grilles : Une grille contient 64x64 capteurs. Afin d'augmenter la résolution la capture au-delà de 64x64 pixels, la caméra contient 4 grilles, ce qui donne une caméra de résolution 128x128. Les 4 grilles étant légèrement espacées, on compte un écart de 4 pixels entre chaque grille. Au final, les données fournies sont de 132x132, mais avec des pixels ne contenant aucune information.

Individuellement, chaque capteur va donc renvoyer, pour chaque lapse de temps, le nombre de photons reçus. Ce qui donne des données ressemblant à ceci :

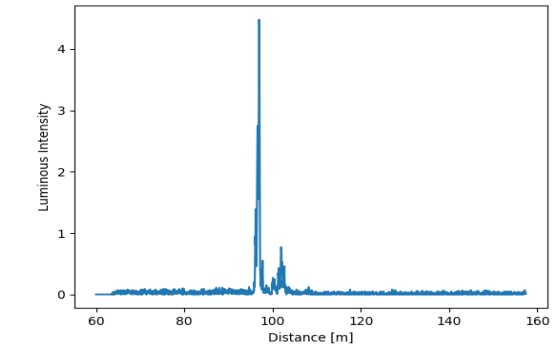

Figure 6 : pic clair

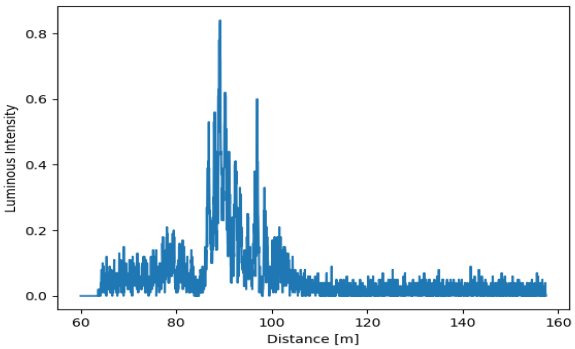

Figure 5 : pic plus bruité

La vitesse de propagation de la lumière étant constante, on parlera ici indifféremment d'une mesure de temps et d'une mesure de distance.

Les capteurs sont réglés à l'avance pour enregistrer l'information à partir d'un certain temps et jusqu'à un certain temps. Ceci permet de réduire le volume de données capturées en précisant dans quelle étendue se trouve notre cible (Ici sur les figures 3 et 4, entre 60 et 120 mètres).

Les données fournies par la caméra sont interprétées par un ordinateur et stockées dans des archives au format d'archives Numpy. Numpy est une bibliothèque Python permettant d'accélérer grandement les calculs sur des données de très grandes dimensions.

Au final, les données que le programme lit sont des archives Numpy. Dans une archive sont contenues des matrices de dimensions 138x138xN avec N le nombre d'observations des capteurs. Chaque entrée de la matrice représente donc l'intensité lumineuse récoltée au temps t par le capteur situé à l'indice x, y de la grille de capteurs.

Solutions proposées

Choix des outils

Lors du premier contact avec le projet, le premier réflexe a été de proposer une solution basée sur OpenGL et donc écrite en C++. Etant donné la nature du projet et le besoin d'une application permettant de traiter potentiellement de très nombreux objets, le choix d'une application écrite directement en OpenGL me semblait la plus évidente. Ce choix aurait sûrement mené à un projet moins abouti, tant il aurait été long ou fastidieux de réimplémenter de nombreux algorithmes ou des interactions utilisateurs.

Le langage C++ est devenu problématique dès la prise en main des données. Les données sont stockées sous forme d'archives Numpy, qui requièrent donc le langage Python pour être chargées en mémoires. Il aurait été possible de lancer un programme Python depuis un programme écrit en C++ pour aller lire les données nécessaires, mais déjà le doute était installé.

De nombreuses bibliothèques sont écrites en C++ avec OpenGL et offrent des bindings pour le langage Python. De Nombreuses bibliothèques sont néanmoins de très haut niveau, et ne

permettent pas une assez grande liberté, elles ne sont pas assez généralistes. C'est notamment le cas de MayaVi, l'application utilisée par l'équipe du CSEM. Néanmoins MayaVi est elle-même construite sur une autre bibliothèque qui permet cette généralité : VTK.

Après investigation, il est apparu que VTK joue un rôle important dans le monde académique, possède une solide communauté et est solide depuis de nombreuses années. Après avoir installé VTK et essayé quelques exemples, la simplicité d'utilisation et de la puissance de cet outil en ont fait le choix de prédilection.

Une fois le couple Python / VTK solidement ancré comme choix pour avancer dans le projet. Il restait à choisir une manière d'interfacer la visualisation faite sur VTK. Le choix se séparait, avec un binding Python, entre les bibliothèques Qt et Tk / TkInter. Des deux choix possible, Qt est le plus largement utilisé, et le plus sophistiqué des deux bibliothèques, étant donné le besoin de créer des Widgets personnalisés pour les besoins de l'application, Qt s'est imposé comme le meilleur choix.

Séparabilité, réécriture

Une force de cette application provient de la grande séparabilité de ses différents composants. La classe PointCloudVisualisator, permettant la visualisation du nuage de points et toutes les interactions avec celui-ci est totalement cloisonnée, absolument indépendante de l'interface utilisateur. Le code a été pensé de manière à ce que l'on puisse retirer totalement la partie GUI, le réécrire avec une bibliothèque différente, puis reconnecter les appels de méthodes sans que cela ne perturbe le fonctionnement de la classe de visualisation.

En termes de réécriture du code, la classe utilitaire permettant d'aller lire les données brutes est très facile à étendre. L'équipe du CSEM modifie sa manière de lire dans les données continuellement, il est important de pouvoir leur donner les outils pour qu'ils puissent étendre cette classe et implémenter de nouvelles manières de lire les données brutes.

Lecture des données

Nous avons déjà discuté de la forme que prennent les données. Discutons alors de la manière employée pour les lire. La classe LidarDataInterpreter s'occupe de la lecture et de l'interprétation des données. L'utilisation est simple, on instancie un objet « interpréteur » en lui indiquant un répertoire contenant nos données. On peut ensuite lui demander différentes manières d'interpréter ces données selon nos envies.

Chacune des méthode getXXX va lire les données d'une manière unique, nous reviendrons plus tard sur les deux lectures différentes proposées par le programme.

Il est important de préciser que ces lecteurs de données implémentent le pattern « Generator », ce pattern consiste à offrir les éléments uniquement lorsqu'ils sont demandés. Au contraire d'un itérateur sur une liste qui garderait en mémoire tous ses éléments, le générateur ne les calcule que lorsqu'ils sont demandés. Cette différence est particulièrement importante lorsque l'on traite de très grandes données. En effet, en évitant de passer par une structure intermédiaire qui prendrait inutilement de la place en mémoire, cette technique permet de ne placer les données que dans une structure, celle de traitement et d'affichage.

FullCube vs DepthMap

Illustrons ce problème de lecture de donnée en opposant les deux modes de lectures présents à ce jour dans l'application. Sachant de quelles formes sont nos données, des histogrammes contenant l'information d'intensité lumineuse pour chaque observation donnée d'un capteur, on oppose deux manières de représenter les données :

La première consiste à construire une image de profondeur : pour chacun des capteurs, on va chercher dans l'histogramme le pic qui indiquera le mieux la profondeur à laquelle se trouve un obstacle. Le désavantage de cette méthode est que certains histogrammes ressemblent à ceci :

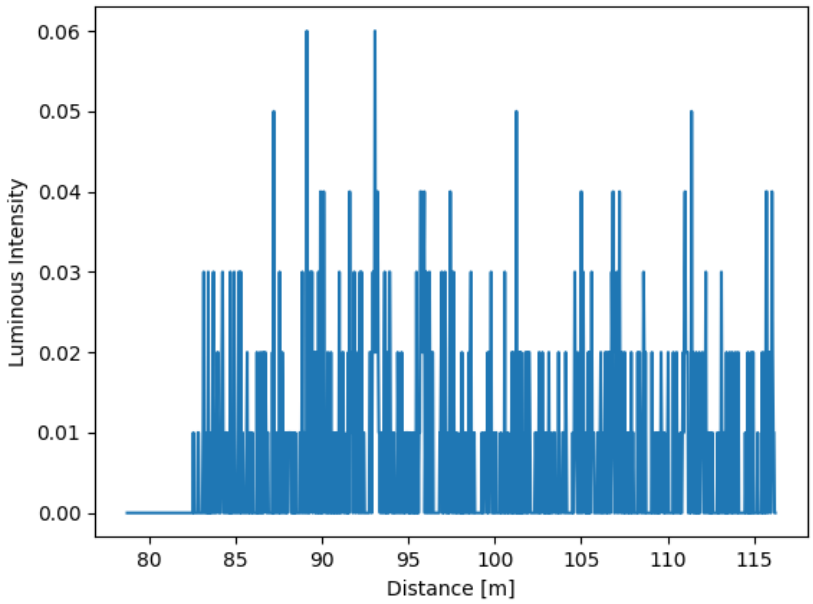

Figure 7 : Capteur très peu illuminé

Difficile donc de déterminer quel est le bon pic. Cette méthode mène à des images passablement bruitées, on saura qu'on ne pourra pas faire confiance à la plupart des points présents dans notre scène.

La deuxième méthode consiste à placer un point dans notre scène pour chacune des observations présente dans notre scène. La manière de voir quelque chose est alors de jouer sur la transparence des points. On considère que plus une observation d'un histogramme est forte, plus le point est opaque, à contrario si l'on a une intensité presque nulle, le point sera totalement transparent. Cette méthode a pour désavantage de consommer énormément de ressources à la machine, tant en termes de mémoire qu'en calcul de rendu. Elle a comme avantage de permettre de visualiser toutes les données d'un seul coup, sans réelle perte d'information.

Observons un exemple de chacune des techniques :

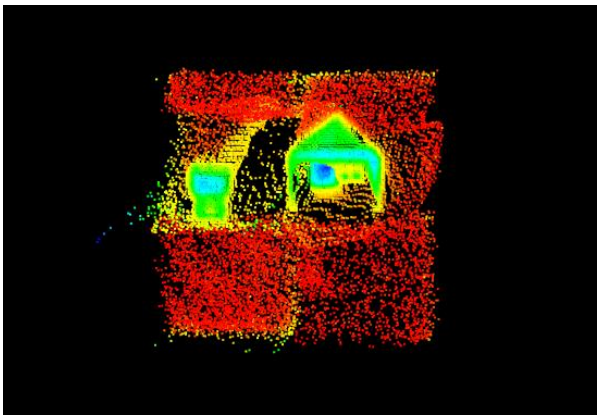

Figure 10 : Construit avec la DepthMap

Figure 9 : mode Full Cube

Sur l'image de gauche, les points rouges sont ceux de faible intensité lumineuse. Sur l'image de droite, la densité de points est gigantesque, mais les points de faibles intensités sont transparents

Filtrage

La très grande force de ce programme est de pouvoir influer sur notre nuage de points au travers de différents filtres. Nous allons, sans rentrer dans les détails d'implémentation, lister les différents filtres qui permettent de clarifier les données et d'obtenir des images plus pures.

Il est important de noter que chaque point dans notre scène porte avec lui deux informations : son intensité relative aux autres points ainsi qu'une mesure sigma. La mesure sigma indique l'écart type d'un point sur plusieurs mesures prises. En effet il est fréquent de prendre plusieurs captures d'une scène sans bouger la caméra LiDAR. Il est alors possible de calculer l'écart type que chaque point a eu avec ses précédentes itérations.

Les points de notre scène passent par ces filtres dans l'ordre qu'ils seront décrits, ceux qui ne sont pas mis de côté sont affichés à l'écran :

1. Filtrés par l'utilisateur : l'utilisateur peut décider de passer en mode de sélection de points, tous les points qui auront été sélectionnés ne passeront plus ce filtre.
2. Filtre par profondeur : on peut réduire l'étendue sur laquelle les points seront affichés. On précise une distance minimum et maximum, tous les points sortants de cet écart seront filtrés.
3. Filtre par Intensité relative : On considère que les points ayant une trop petite intensité ne sont pas dignes d'intérêts, on fait donc sortir tous les points ayant une intensité plus petite que le seuil décidé.

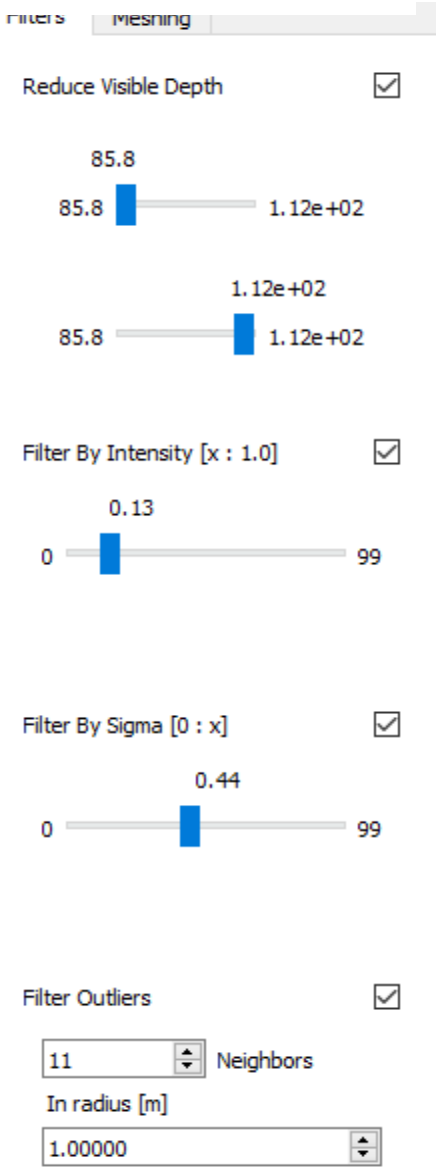

Figure 8 : Widget de contrôle des filtres

4. Filtre par Ecart-Type relatif : On considère que les points qui bougent trop d'une observation à l'autre ne sont pas dignes d'intérêts. Ce filtre permet de garder les points qui bougent le moins.
5. Filtre des cas particuliers : On considère que les points qui sont trop isolés sont le résultat d'une erreur de mesure. Ce filtre va calculer pour chaque point s'il possède un certain nombre de voisins dans un certain rayon autour de lui. Si le point n'a pas assez de voisins, il est filtré.

Observons les effets des filtres sur nos données, avec un avant / après :

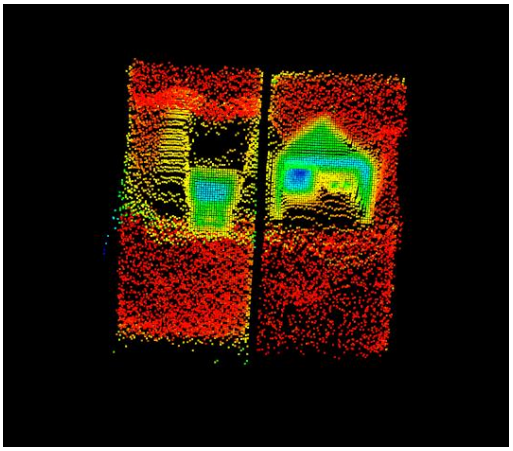

Figure 12 : Avant Filtres

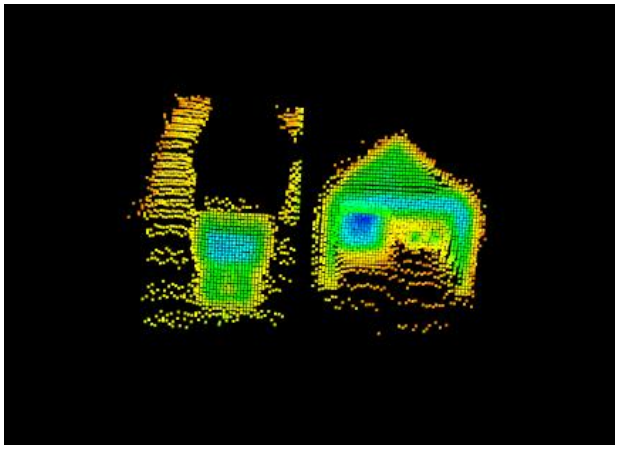

Figure 11 : Après filtres

Les filtres nous permettent de mieux comprendre ce qui, dans nos données, relève probablement du bruit.

Création de Meshs

Une fois notre nuage de points bien nettoyé de son bruit, nous pouvons créer une surface à partir des points restants. A la sortie de tous ces filtres, nous pouvons transformer nos données, un nuage de point, en une surface qui va combler les vides entre les points via une triangulation de Delaunay. Cette triangulation s'apparente à étendre un drap selon l'axe de profondeur sur nos points, chaque point agissant comme un sommet pour le drap d'élasticité infinie.

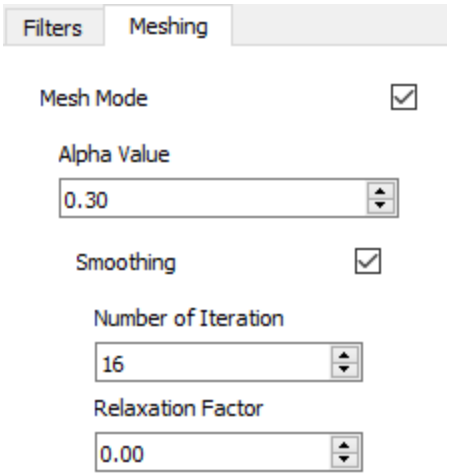

Figure 13 : Widget de contrôle du meshing

Le paramètre Alpha permet de régler l'élasticité de ce drap. La valeur 0 est une valeur spéciale, elle correspond à une élasticité infinie. Sinon, lorsqu'on s'approche de 0, le taux d'élasticité est très bas et le drap se rompt. Plus alpha grandit, plus le drap peut s'étendre loin. Illustrons ceci avec des exemples :

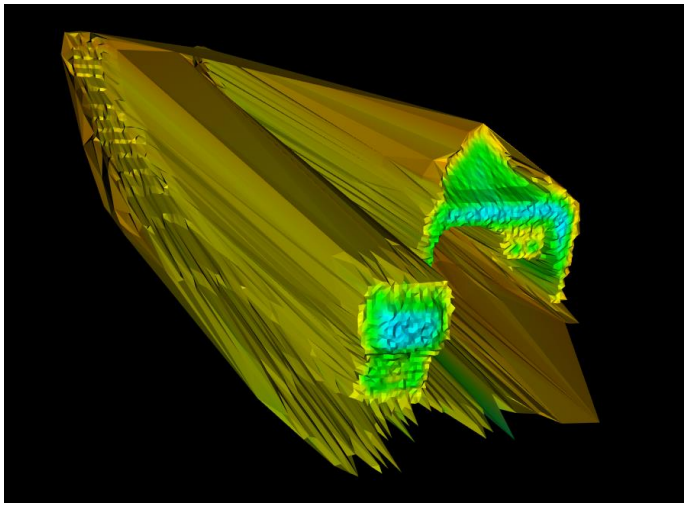

Figure 14 : alpha = 0

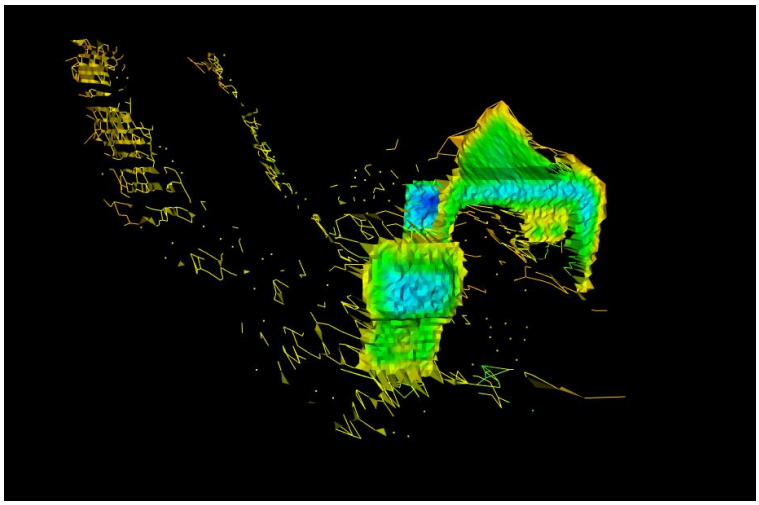

Figure 15 : alpha = 0.3

Une fois le mesh créé, on peut faire en sorte d'arrondir les angles passant notre surface par un lissage. Voici ce que donnent les exemples ci-dessus passés au lissage :

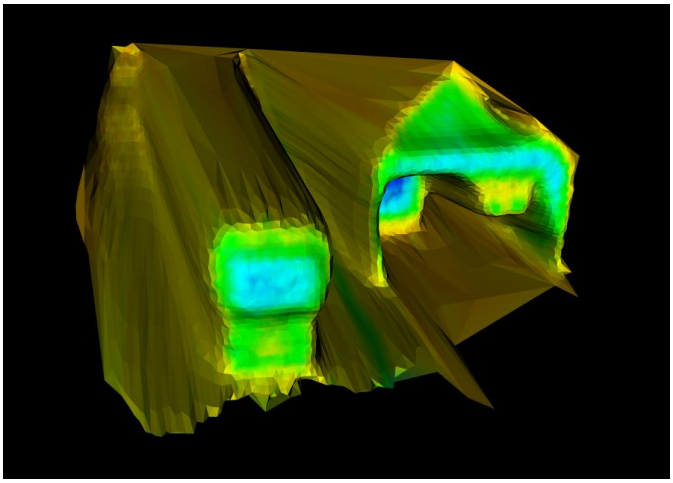

Figure 17 : alpha = 0, lissé

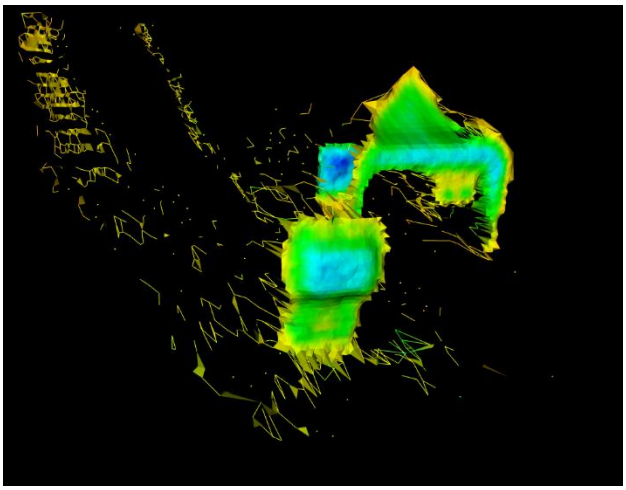

Figure 16 : alpha = 0.3, lissé

Coloris

Rappelons-nous que nos données, nos points dans l'espace, portent avec eux des données, leur intensité relative, et une notion de constance d'un point par rapport à d'autres mesures. Ces données étant porteuse de sens, le programme permet de colorier le nuage de points selon plusieurs critères

- Coloration selon sigma, l'écart type
- Coloration selon l'intensité
- Coloration selon la profondeur du point

La colorisation de ces points se fait au moyen d'une échelle de couleur. Cette échelle de couleur est modifiable et permet de changer la gamme de ton. Il est possible d'afficher l'échelle de couleur sur le côté gauche de la visualisation. Voici un aperçu des échelles implémentées :

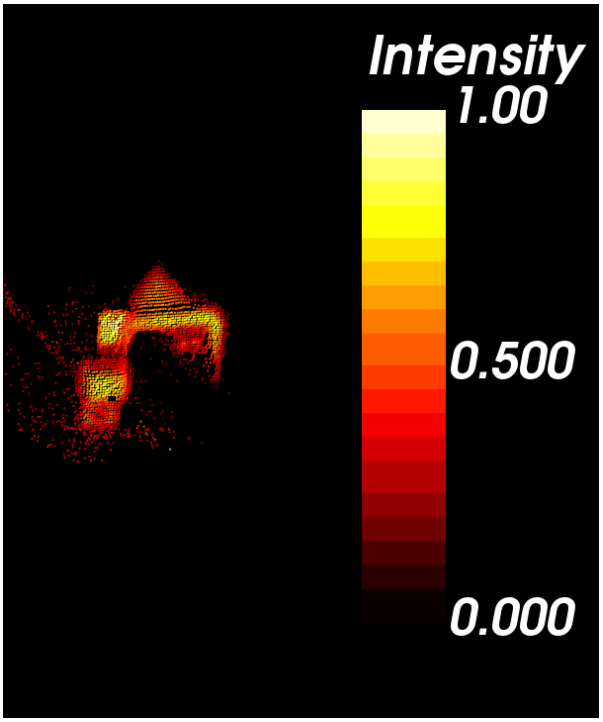

Figure 20 : échelle "Hot"

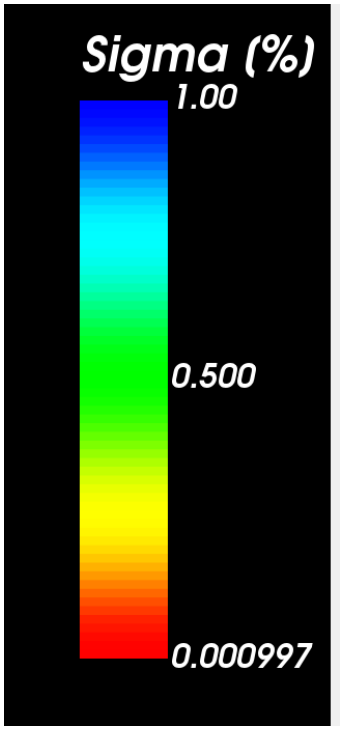

Figure 18 : échelle personnalisée

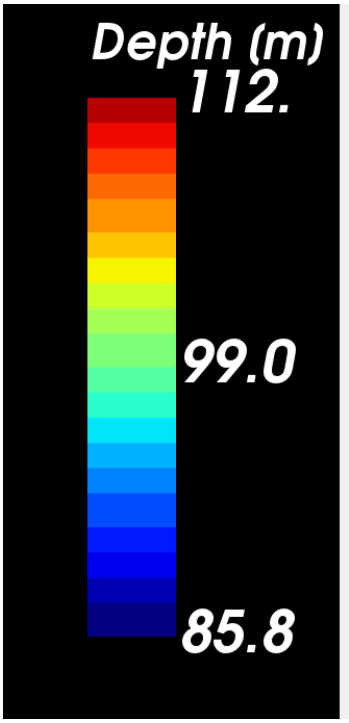

Figure 19 : échelle "Jet"

Une échelle de couleur indique à quelle donnée on s'intéresse, puis indique la correspondance des couleurs aux dimensions liées aux données.

L'implémentation d'une nouvelle échelle de couleur est extrêmement simple, il suffit de construire une liste des couleurs par lesquelles on veut passer. La classe PointCloudVisualisator implémente une méthode qui construira une échelle de couleur linéaire passant par les couleurs de la liste passée en argument.

Une autre fonctionnalité utile est la possibilité de recalculer les échelles de couleur en fonction soit de tous les points, soit uniquement des points filtrés. Cette fonctionnalité est très utile une fois que les points ont été filtrés, si l'on veut obtenir plus de nuance de couleur dans les points restants, on peut recalculer l'échelle en fonction des points survivants.

Aide à la visualisation

Il est parfois difficile de comprendre dans quelles sens est situé le nuage de points, il est possible de perdre le Nord en tournant autour du nuage de point. A cette fin, l'application implémente des indicateurs qui permettent de mieux comprendre ce que l'on voit.

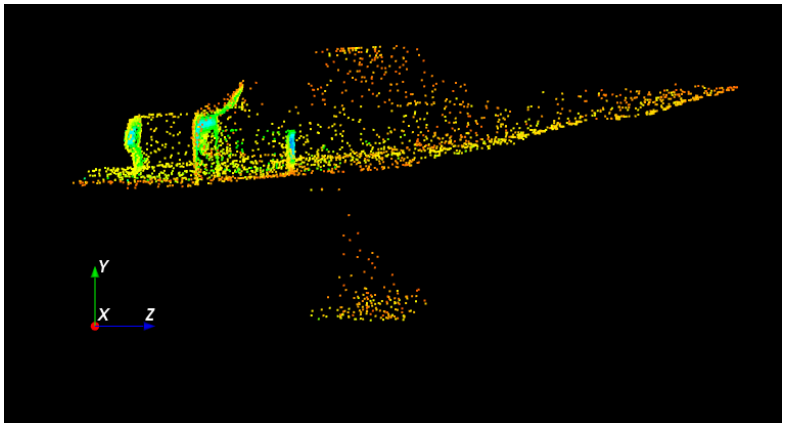

Figure 21 : Axes

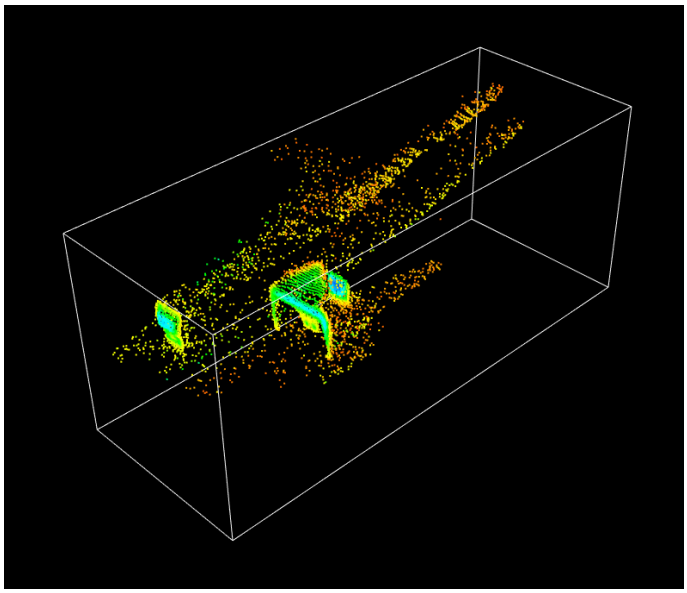

Figure 22 : rectangle englobant

Le premier indicateur se présente sous la forme de trois petits axes nichés dans le coin inférieur droit de l'écran. Ces axes correspondent aux axes x, y, z de l'espace Euclidien. Ainsi, il est facile de comprendre comment est orientée notre caméra. Sur l'illustration ci-dessous, nous regardons dans la direction de l'axe des c'est donc une vue de côté.

Il est parfois difficile de comprendre quel espace occupent vraiment les points, étant donné que nous voyons le nuage au travers d'une caméra. Dans le but de mieux comprendre la place prise par notre nuage de point, une boîte entourant les données pernet d'appréhender le volume général de manière plus facile.

Enfin, dans un souci de comprendre la taille réelle de la scène que nous voyons, nous avons besoin d'un repère qui indique la distance réelle. Dans l'exemple ci-dessous, on comprend que notre cible et notre tente se trouvaient à 87 mètres de la caméra, et que la prise de vue s'étendait jusqu'à 112 mètres.

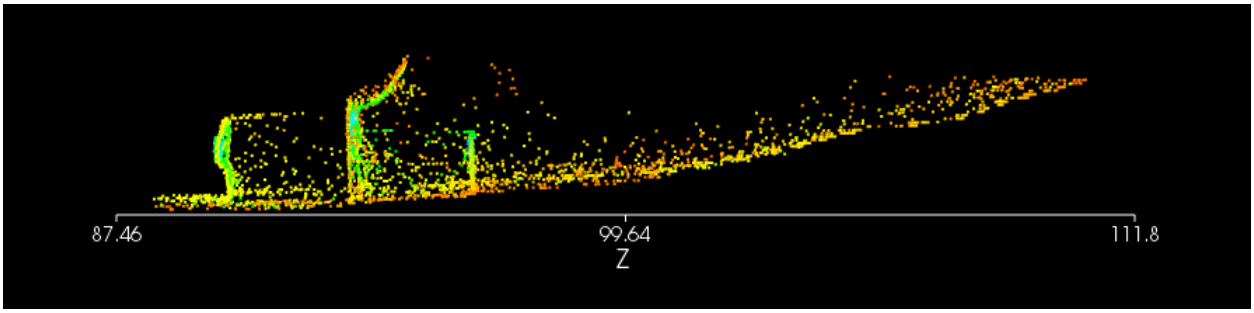

Figure 23 ; Indicateur de profondeur

Toutes ces aides visuelles peuvent être cachées par l'utilisateur s'il trouve qu'elles sont gênantes.

Toujours dans l'optique de ne pas se perdre dans notre visualisation, le programme permet de recentrer la caméra selon plusieurs angles prédéfinis. Ainsi, on peut demander de placer la caméra en vue de face (regarde dans le sens de l'axe Z), en vue de côté droit (regarde dans le sens de l'axe X), en vue du dessus (regarde dans le sens inverse de l'axe Y) ou encore en vue Isométrique à 45° selon les trois axes.

Implémentation

Lecture des données

La lecture de données est faite dans la classe LidarDataInterpreter. Cette classe est assez simple à comprendre, elle correspond au pattern de fabrique, qui va nous donner, selon ce que l'on souhaite, différents lecteurs de données. La fabrique va créer un générateur de point. Ce générateur de points garde une référence sur les données de la fabrique par le processus de closure.

Pour expliquer le mécanisme de closure, lorsque la fabrique renvoie un générateur, il est empaqueté avec les références vers les données qui lui sont nécessaires dans la fabrique.

Nos générateurs créent des objets InterestPoints, ce sont de simples objets à qui l'on attribue une position, une information sur l'intensité du point d'intérêt, et de son écart-type (Sigma).

Un générateur a cela de différent avec un autre objet itérable qu'il ne fait les calculs nécessaires à la génération de l'élément suivant que quand on lui demande explicitement. En soi, la fabrique ne prend aucun temps de calcul.

Cette manière de créer des générateurs est très simple à mettre en place, il est très important qu'une personne reprenant le projet puisse créer son propre générateur de donnée. C'est sur cette flexibilité que repose une bonne partie de la force de l'application.

Bibliothèque VTK

Présentation

VTK est une bibliothèque de visualisation de données écrite en C++. Elle met en place un système de chaîne d'algorithme pour traiter les données sur lequel nous reviendrons. VTK possède plusieurs bindings qui lui permettent d'être interfacée avec d'autre langages de programmation (Java, Python, Tcl), de plus, elle est intégrable à plusieurs bibliothèques d'Interfaces Utilisateurs (Qt, Swing, Tk). Cette bibliothèque est une surcouche d'OpenGL, ce qui en fait un outil à très large spectre, multiplateforme et utilisant toute la puissance d'OpenGL pour accélérer les calculs de rendu.

Un de ses défaut est d'être compliquée à installer, il faut en effet la recompiler à partir des sources. Un guide d'installation pour Windows sera mis en annexe de ce document.

Architecture

L'architecture de VTK gravite autour de son pipeline graphique. Le pipeline est, comme son nom l'indique, une suite de tuyaux que l'on peut arranger à sa guide, certains tuyaux prenant

plusieurs entrées, ou menant à plusieurs sorties, il est possible de réarranger le pipeline comme bon nous semble. Cette architecture très flexible permet tous les traitements de données imaginables, à condition de ne pas perdre de vue ce que l'on manipule.

Un pipeline VTK aura tout de même structure de base à respecter, certains blocs sont immuables :

- La source, contient les données. Les données peuvent être générées programmatique ment, être lues depuis un fichier ou provenir des classes déjà implémentées par VTK (sphères, rectangles, cônes…)
- Les filtres, c'est ici que le pipeline peut devenir aussi long et tordu que voulu, chaque filtre ajouté est un algorithme qui va modifier les données reçues. Par exemple, on peut filtrer des points selon leur position, selon des données attachées aux points, faire fusionner des sources différentes. Une plongée dans la documentation VTK est nécessaire à ce stade pour trouver le bon outil correspondant à vos désirs
- Le mapper, cet objet essentiel va transformer vos données en primitives OpenGL.
- L'acteur, c'est l'objet 3D VTK en lui-même, cet acteur servira à positionner et orienter notre objet, le colorer, ou changer des propriétés graphiques.
- Le renderer, auquel on ajoute des acteurs, s'occupe du rendu desdits acteurs
- La fenêtre de rendu, connectée au renderer, sera la fenêtre en elle-même
- L'interacteur, s'attache à la fenêtre, lance la boucle événementielle, et s'occupe des interactions avec l'utilisateur

Il est vivement conseillé d'aller lire des exemples pour comprendre la mise en place du pipeline et les connections faites entre ces différents objets. De nombreux exemples sont disponibles sur le site web de VTK et ce dans les différents bindings :

http://www.vtk.org/Wiki/VTK/Examples

Filtrage des données

Maintenant que l'on s'est familiarisé avec le principe de base de VTK, voyons comment nous avons mis cette architecture à profit dans ce projet.

Initialisation du pipeline

La première chose à faire est de copier les points d'intérêts qui nous viennent du générateur dans une structure de donnée VTK, donc par extension une structure de donnée qui sera utilisée par OpenGL. Au sortir de notre lecture de points, nous avons une structure utilisée par VTK pour représenter les données non ordonnées : un vtkPolyDAta.

Nos données sont prêtes à entreprendre leur périple à travers le pipeline graphique VTK, mais encore faut-il le mettre en place.

Notre application a besoin d'un pipeline modulable, il est nécessaire de pouvoir ajouter et enlever des filtres à notre guise. Pour cela, nous créons un squelette composé de plusieurs tuyaux qui ne font que faire passer l'information. Ces tuyaux, des vtkPassThrough, seront les points fixes de notre pipeline auxquels viendront se greffer les filtres mobiles.

Un autre point fixe de notre pipeline est notre premier filtre. Le filtre qui permet à l'utilisateur de choisir des points dans le nuage pour les bannir. Ce filtre est actif en tout temps et ne peut pas être retiré du pipeline.

A ce stade, une fois les connexions faites, nos données passent à travers le filtres des points bannis par l'utilisateur avant de traverser tous les filtres « «passthrough » pour terminer au niveau du mapper, et donc être affichés à l'écran.

Activation / Désactivation d'un filtre

Toute les méthodes qui permettent d'ajouter ou d'enlever un filtre fonctionnent d'une manière extrêmement similaire, bien que chaque filtre soit initialisé d'une manière particulière. Les filtres du pipeline sont placés dans un ordre précis, de l'algorithme le plus rapide à l'algorithme le plus lent. Chaque filtre ayant une place précise, son activation signifie uniquement que l'on créé le filtre, on le connecte aux vtkPassThrough correspondants à sa place, puis on modifie les réglages du filtre en fonction de ce que l'utilisateur a demandé.

La désactivation d'un filtre est très simple aussi, on déconnecte le filtre, on reconnecte les vtkPassThrough correspondant.

Le pipeline VTK va automatiquement détecter le changement opéré et faire repasser les données.

Filtres et classes VTK correspondantes

VTK implémentant de très nombreux algorithmes, le plus difficile n'est pas souvent de mettre le filtre en place, mais souvent de trouver le bon dans la documentation. Nous allons donc rapidement indiquer pour chacun des filtres son utilisation et les pièges qui pourraient subvenir.

Certains filtres nécessites que nos points aient une certaine de leur caractéristique (Intensité, sigma) soit « active ». Dans le monde VTK, être la caractéristique active veut dire que pour un objet PolyData, cette information est dite « Scalars »

- Filtre de points sélectionnés par l'utilisateur : il s'agit ici de plusieurs objets qui vont interagir entre eux, Il est nécessaire de bien initialiser chacun des objets. Le filtre en lui-même est un vtkExtractSelection., il a besoin qu'on lui assigne un vtkSelection. Lequel vtkSelection est construit à bade de nœuds appelés vtkSelectionNode. Une fois les objets mis dans le bon ordre, on pourra modifier l'objet vtkSelectionNode en lui assignant des points qui ne passeront plus le filtre.
- Filtre de profondeur : le filtre est un vtkExtractPolyDataGeometry, il faut le paramétrer avec un objet vtkBox, tout ce qui sera en dehors de cette boîte sera filtré.
- Filtre par Intensité : classe vtkThresholdPoints. En plus de le paramétrer avec un seuil, il est important de s'assurer que les PolyData aient le champ « Intensity » marqué comme étant « scalar ». Pour ceci nous utilisons un objet vtkAssignAttribute qui va modifier l'attribut qui est considéré comme « scalar »
- Filtre par Sigma : Idem que pour le filtre par intensité, mais on doit s'assurer que « Sigma » soit considéré comme « scalar »
- Filtrer par cas particulier : objet vtkRadiusOutlierRemoval, uniquement le paramétrer et le connecter.
- Filtre « Mesh » : objet vtkDelaunay2D, uniquement le paramétrer et le connecter

- Filtre « MeshSmooting » : objet vtkSmoothPolyDataFilter, uniquement le paramétrer et le connecter.

Coloris

Créer une échelle de couleur est très simple grâce à la classe vtkLookupTable. Quelle que soient les couleurs désirées dans l'échelles, il est possible de les ajouter dans l'ordre à la lookupTable, puis de la construire.

La colorisation des points, par contre, pose des problèmes relativement corsés lorsque nos données sont sujettes à être coloriées selon des critères différents (Ici Intensité, Sigma, Profondeur). Il faut comprendre que le mapper VTK va colorier les points en fonction de leur attribut dit « scalar ». Le mapper va dériver les couleurs sur sa lookupTable en fonction de cet attribut. Pour colorier les points de la bonne manière, on utilise donc des vtkAssignAttribute aux endroits stratégiques pour changer l'attribut qui sera « scalar » et donc colorer les points selon nos désirs.

Interfaçage avec PyQt

Il est à noter que notre classe principale, PointCloudVisualiser, hérite de la classe QVTKRenderWindowInteractor. C'est cette parenté qui permet l'intégration de ce widget VTK dans Qt. Il s'agit de l'unique dépendant de cette classe avec la bibliothèque Qt, si un futur développeur voulait changer de librairie. Il devrait changer le parent de cette classe pour le vtkRenderWindowInteractor adapté. En voici la liste tirée de la documentation :

- vtkAndroidRenderWindowInteractor
- vtkCocoaRenderWindowInteractor
- vtkGenericRenderWindowInteractor
- vtkIOSRenderWindowInteractor
- vtkOculusRenderWindowInteractor
- vtkOpenVRRenderWindowInteractor
- vtkWin32RenderWindowInteractor
- vtkXRenderWindowTclInteracto

Nous avons là une illustration de la force de VTK, qui garde une couche d'abstraction qui lui permet de s'interfacer avec n'importe quelle librairie de GUI.

L'interface Qt n'est sûrement pas le meilleur point de ce projet. Bien que très fonctionnelle, elle pourrait être rendue plus belle. On peut noter qu'elle a en partie été créé par l'outil QtDesigner, qui permet de gagner énormément de temps, et est très facile d'usage même avec Python.

Conclusion

Améliorations, suites du projet

Le projet mérite très certainement être continué, de nombreuses fonctionnalités ont été discutées puis abandonnées, faute de temps. Un des aspects qui n'a malheureusement pas été retenu est une exploration des différentes manières dont les données initiales peuvent être lues. Je garde espoir que ces meilleures lectures des données pourront être poursuivies par l'équipe du CSEM.

Le mot de la fin

Lors des différentes présentations faites au CSEM, les mandants semblaient très satisfaits du résultat de cette application, qui permet de bien explorer les données sous des angles qui étaient difficilement visibles avant. En l'état, le code est encore un peu brouillon par endroits et mériterait d'être vraiment mis en ordre pour être digne d'un logiciel non plus en phase de Debug, mais de Production.