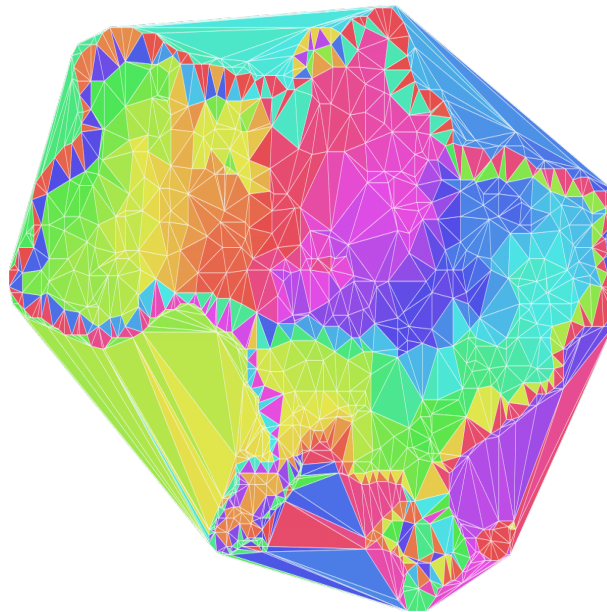


Application de triangulation de points lidar sur le canton de Genève



Thèse de Bachelor présentée par

Jérôme CHÉTELAT

pour l'obtention du titre Bachelor of Science HES-SO en

**Ingénierie des technologies de l'information avec orientation en
logiciels et systèmes complexes**

Septembre 2020

Professeur-e HES responsable

Orestis MALASPINAS

Légende et source de l'illustration de couverture : Une triangulation de Delaunay des points aléatoires. Source : tiré de <https://github.com/mapbox/delaunator>

TABLE DES MATIÈRES

Remerciements	vi
Énoncé du sujet	vii
Liste des acronymes	ix
Liste des illustrations	x
Liste des tableaux	xi
Introduction	1
1 Chapitre 1 : Données et format de fichier	3
1.1 Nuage de point	3
a Format LIDAR	3
b En-tête public	4
c Enregistrements à longueur variable	5
d Données des points	5
e Classification des points	7
1.2 Fichier Stéréoligraphique	7
a Format ASCII	8
b Format binaire	8
2 Chapitre 2 : Algorithmes	10
2.1 Nettoyage	10
a Nettoyage par classe de point	10
b Nettoyage par point pondéré	10
c Filtre par point moyen	11
2.2 Triangulation de Delaunay	12
a Bowyer-Watson	12
b Lee & Scachter	14
2.3 Fusion de maillage	14
3 Chapitre 3 : Implémentations	17
3.1 Rust	17
3.2 Architecture	17
3.3 Bibliothèque	17
a Nettoyage	19
b Triangulation	19
c Fusion de triangulation	19
3.4 Utilitaires	21
a mesh-filter	22
b mesh-merge	22
3.5 Stack Web	22
a Serveur	22

b	Client	22
4	Chapitre 4 : Résultats	23
4.1	Triangulation	23
4.2	Nettoyage	23
4.3	Stack Web	23
a	Serveur Web	23
b	Client Web	23
	Conclusion	24
	Annexe 1 : Tableau des champs de l'en-tête de fichier LAS	25
	Annexe 2	26
	Annexe 3	29
	Bibliographie	30

REMERCIEMENTS

En premier lieu, je tiens à remercier monsieur Orestis Malaspinas, qui m'a encadré tout au long de ce travail de bachelor et qui m'a fait partager ses intuitions. Qu'il soit aussi remercié pour sa gentillesse, sa disponibilité et pour ses nombreux encouragements.

Je remercie monsieur Gabriel Strano pour son aide dans la rédaction du code et la relecture de mon mémoire ainsi que ses précieux conseils pour la rédaction du mémoire.

Je tiens à remercier également madame Emmanuelle Moreau pour la relecture de mon mémoire ainsi que pour son soutien et ses nombreux encouragements qu'elle m'a prodiguée tout au long de mon travail.

APPLICATION DE TRIANGULATION DE POINTS LIDAR SUR LE CANTON DE GENÈVE

ORIENTATION : LOGICIELS ET SYSTÈMES COMPLEXES

Descriptif :

Les nuages de point lidar aériens sont une technologie de récolte de topographie qui a émergé durant les vingt dernières années. Ces données sont difficiles d'utilisation sans un traitement préalable. Nous proposons ici plusieurs outils de traitement de données lidars ainsi que de maillage. Dans un premier temps, les outils devront être implémentés ensuite un client web permettant de parcourir les fichiers et de les visualiser dans un navigateur web devra être implémenté.

Travail demandé :

Dans les limites du temps imparti, les tâches suivantes seront réalisées :

- Explorer des manières de sous-échantillonner, filtrer des données lidar et les implémenter.
- Explorer des manières efficaces de créer un maillage à la volée à partir de données lidar.
- Explorer des manières de combiner plusieurs maillages, de les décimer ainsi que de les sous-échantillonner. Par la suite, les implémenter.
- Création d'un client web permettant de parcourir et de charger des maillages ou des données lidar dans un navigateur web.

Candidat :

CHÉTELAT JÉRÔME

Filière d'études : ITI

Professeur responsable :

ORESTIS MALASPINAS

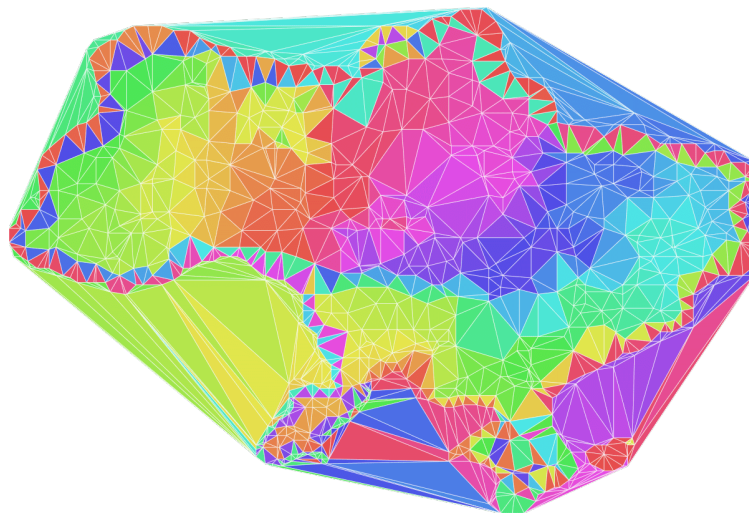
En collaboration avec :

Travail de bachelor soumis à une convention de stage en entreprise : non

Travail de bachelor soumis à un contrat de confidentialité : non

RÉSUMÉ

Depuis l'antiquité à nos jours, les cartes sont le reflet des premières formes de données récoltées par l'Homme. La cartographie a aidé l'humanité à définir des chemins à travers le monde et à naviguer. Cependant avec les fortes avancées technologiques des dernières décennies, les données récoltées ont augmenté massivement et notamment dans la cartographie avec les données LIDAR. Ces dernières représentent une des informations collectées des plus massives. Elles sont utilisées dans l'étude topographique de régions, dans les géosciences, dans la science environnementale ou bien encore elles viennent en aide au guidage automatique de véhicule terrestre. Un problème fréquent lié à ces données est le traitement réalisé avant leurs utilisations dans des applications, qui est souvent nécessaire afin de filtrer tout ce qui ne présente aucune utilité et pourrait même fausser les résultats. Il reste cependant difficile de les traiter manuellement due à la quantité de données importante. Pour ce faire, des méthodes analytiques sont employées afin d'accélérer ce processus au travers de systèmes automatisés. Un autre problème présent est leurs utilisations pour reconstruire des surfaces. Les méthodes automatisées n'étant pas à 100% fiables, il est nécessaire de vérifier la cohérence des données ainsi que la qualité des maillages produits. Ce travail se focalise principalement sur la création d'outils de transformation et de nettoyage de données LIDAR et l'affichage de maillages produit par les transformations au travers d'un client web. Il est également focalisé sur la conception d'outils de traitement de maillage résultant d'un traitement de données LIDAR.



Candidat-e :

JÉRÔME CHÉTELAT

Filière d'études : ITI

Professeur-e(s) responsable(s) :

ORESTIS MALASPINAS

Travail de bachelor soumis à une convention de stage
en entreprise : non

Travail soumis à un contrat de confidentialité : non

LISTE DES ACRONYMES

API Application Programming Interface. 22

ASPRS American Society for Photogrammetry and Remote Sensing. 3

CAO Conception Assistée par Ordinateur. 8

GPS Global Positioning System. 3, 7

LAS Laser Format. xi, 3, 4, 6

LIDAR Light Detection And Ranging. 3, 10, 22, 24

REST Representational State Transfer. 17

SITG Système d'information du territoire à Genève. 1

STL Stéréolitographie. 7, 17, 20, 23

WASM WebAssembly. 17, 22, 24

LISTE DES ILLUSTRATIONS

1.1	Schéma représentant un instrument de mesure LIDAR. Source : tiré de réf URL02	6
2.1	Surface plane avant le passage par un filtre. Source : réalisé par Jérôme Chételat	11
2.2	Surface plane après le passage par un filtre. Source : réalisé par Jérôme Chételat	11
2.3	Deux triangulations du même jeu de données. Source : tiré de ref URL01 . . .	12
2.4	Maillage résultant d'une triangulation de delaunay. Source : réalisé par Jérôme Chételat	13
2.5	Création de la base LR entre les deux maillages. Source : réalisé par Jérôme Chételat	14
2.6	Candidat final du maillage M_L . Source : réalisé par Jérôme Chételat	15
2.7	Candidat final du maillage M_R . Source : réalisé par Jérôme Chételat	16
3.1	Structure de fichier présent dans le module de la librairie de l'application. Source : réalisé par Jérôme Chételat	18

Références des URL

- URL01 <https://www.ti.inf.ethz.ch/ew/Lehre/CG13/lecture/Chapter%206.pdf>
- URL02 <https://hackaday.io/project/20628-open-simple-lidar>

LISTE DES TABLEAUX

1.1	Sections composant un fichier au format LAS	4
1.2	Type de données de la définition du format LAS	4
1.3	Champs de l'en-tête des enregistrements de longueur variable. Source : adapté de LAS Specification Version 1.2 (2008, p. 6)	5
1.4	Information d'un point présent dans un fichier Laser Format (LAS) , Source : adapté de LAS Specification Version 1.2 (2008, p. 6)	6
1.5	Représentation des bits dans l'octet de classification. Source : adapté de LAS Specification Version 1.2 (2008, p. 8)	7
1.6	Signification des valeurs de classification. Source : adapté de LAS Specification Version 1.2 (2008, p. 8)	7
4.1	Champs de l'en-tête d'un fichier LAS	25

INTRODUCTION

Contexte

Depuis l'antiquité à nos jours, les cartes sont le reflet des premières formes de données récoltées par l'Homme. La cartographie a aidé l'humanité à définir des chemins à travers le monde et à naviguer. Cependant avec les fortes avancées technologiques des dernières décennies, les données récoltées ont augmenté massivement et notamment dans la cartographie avec les données LIDAR. Ces dernières représentent un nuage de points dense qui sont actuellement une des informations collectées des plus massives. Elles sont utilisées dans l'étude topographique de régions, dans les géosciences, dans la science environnementale ou bien encore elles viennent en aide au guidage automatique de véhicule terrestre. Un problème fréquent lié à ces données est le traitement réalisé avant leurs utilisations dans des applications, qui est souvent nécessaire afin de nettoyer tout ce qui ne présente aucune utilité et pourrait même fausser les résultats. Il reste cependant difficile de les traiter manuellement due à la quantité de données importante. Pour ce faire, des méthodes analytiques sont employées afin d'accélérer ce processus au travers de systèmes automatisés. Un autre problème présent est leurs utilisations pour reconstruire des surfaces. Les méthodes automatisées n'étant pas totalement fiables, il est nécessaire de vérifier la cohérence des données ainsi que la qualité des reconstructions de surface.

Ce travail de bachelor est une continuation d'un projet de semestre effectué sur la reconstruction de surface à partir de données lidar.

Les données lidar sont mises à disposition par le [Système d'information du territoire à Genève \(SITG\)](#) et sont utilisés avec la triangulation de Delaunay pour créer des maillages reconstruisant les surfaces des régions.

Buts

Le but principal de ce travail est de créer une panoplie d'outils de traitement de fichiers lidar ainsi que de fichier stl. Ils doivent permettre à un utilisateur de reconstruire des surfaces grâce à une triangulation sur un nuage de points, nettoyer des données lidar provenant d'un aéronef ainsi que d'afficher un fichier stl dans un navigateur web. Un objectif supplémentaire est de mettre en oeuvre ces outils grâce au langage de programmation Rust.

Méthodologie

Dans un premier temps nous avons recherché des algorithmes de traitement de données lidar. Nous avons par la suite implémenté ceux qui nous paraissaient utile dans l'application. Enfin nous avons testé ces algorithmes sur un jeu de données et collecté des résultats.

Dans ce mémoire, nous allons en un premier détailler les formats de fichier rencontrer dans l'application. Ensuite nous détaillerons les méthodes algorithmiques misent en oeuvre pour réaliser des traitements sur ces fichiers. Enfin nous allons voir comment ces dernières ont été implémenté dans l'application et comparer leurs résultats. Les sources des implémentation sont disponible à l'adresse : <https://githepia.hesge.ch/jerome.chetelat/meshtools>

CHAPITRE 1 : DONNÉES ET FORMAT DE FICHIER

Dans ce chapitre nous allons aborder les différents formats de données qui seront et utilisés dans la suite de ce document.

1.1. NUAGE DE POINT

Un nuage de points est un ensemble de points de données dans un système de coordonnées à trois dimensions. Il est généralement produit à partir d'un instrument de [Light Detection And Ranging \(LIDAR\)](#) ou en français détection et estimation de la distance par la lumière. Il s'agit d'une technique de mesure de distance où la lumière réalise un aller-retour entre sa source et un objet. On chronomètre le temps entre le moment où une impulsion de lumière est émise par un laser et son retour vers un capteur. En connaissant la vitesse de propagation de la lumière dans l'environnement où elle évolue, il est possible de déterminer la distance qui sépare l'objet du capteur. Plus précisément, il s'agit de la distance séparant le dispositif de mesure et un point de l'objet frappé par la lumière émise. Cette opération effectuée à de multiples reprises en changeant par petit pas l'angle du laser, créer des points qui une fois réunis, constituent un nuage de points profilant l'environnement autour de l'instrument. Si l'instrument de mesure est placé sur un aéronef et que la source de lumière est dirigée vers le bas, il est ainsi possible de capturer, sous forme d'un nuage de points, une région du globe. Ce cas nécessite de connaître la position [Global Positioning System \(GPS\)](#) du véhicule au moment de la capture d'un point afin de mettre en relation tous les points à la fin de la mesure. Il existe différentes versions de la spécification et ce document va se concentrer sur la version un point deux.

a. Format LIDAR

Le format de stockage des nuages de points de données [LIDAR](#), est un format binaire dont les spécifications sont définies par l'[American Society for Photogrammetry and Remote Sensing \(ASPRS\)](#). Le format [LAS](#) est composé de trois sections, respectivement : L'en-tête de fichier, les enregistrements à longueur variable et les données liées aux points récoltés.

En-tête
Enregistrements à longueur variable
Données des points

TABLEAU 1.1 – Sections composant un fichier au format LAS, Source : adapté de LAS Specification Version 1.2 (2008, p. 2)

Dans toutes les sections, on définit les types de données dans le tableau 1.2

Type de donnée	Nombre d'octets
char	1
unsigned char	1
short	2
unsigned short	2
long	4
unsigned long	4
double ¹	8

TABLEAU 1.2 – Type de données de la définition du format LAS. Source : adapté de LAS Specification Version 1.2 (2008, p. 2-3)

b. En-tête public

L'en-tête public comporte les métadonnées du fichier. Il est composé de tous les champs du tableau 4.1 présent dans l'annexe 1. Ils sont tous encodés en mode little endian. Tous les champs inutilisés doivent être remplis par des bits nuls. La signature de fichier doit obligatoirement contenir les quatre caractères "LASF" et celle-ci est requise par la spécification. Un programme lisant le fichier doit vérifier cette signature pour déterminer qu'il s'agisse bien d'un fichier LAS.

Parmi les champs, les facteurs X, Y et Z scale sont utilisés avec les valeurs X, Y et Z offset pour obtenir des coordonnées cartésiennes des axes. Ces valeurs sont globales au fichier. Voici la formule pour chaque coordonnée :

$$X_{coordinate} = (X_{record} * X_{scale}) + X_{offset}$$

$$Y_{coordinate} = (Y_{record} * Y_{scale}) + Y_{offset}$$

$$Z_{coordinate} = (Z_{record} * Z_{scale}) + Z_{offset}$$

Les champs min et max X, Y, Z sont les valeurs minimales et maximales non mises à

1. selon le standard IEEE 754

l'échelle des données des points présents dans la dernière section du fichier.

c. Enregistrements à longueur variable

Ce bloc suit directement l'en-tête de fichier et peut avoir une ou plusieurs entrées. Le nombre total d'entrées est spécifié dans le champ "Number of Variable Length Records" de l'en-tête de fichier. Ce bloc doit être lu de manière séquentielle, car chaque entrée est de taille variable. Chaque entrée est débutée par un en-tête de 54 octets selon le tableau 1.3

Champ	Type de donnée	Taille (Octets)	Requis
Réservé	unsigned short	2	
User ID	char[16]	16	*
Record ID	unsigned short	2	*
Record Length After Header	unsigned short	2	*
Description	char[32]	32	

TABLEAU 1.3 – Champs de l'en-tête des enregistrements de longueur variable. Source : adapté de LAS Specification Version 1.2 (2008, p. 6)

Les enregistrements peuvent provenir de sources différentes. Elles sont identifiées par le champ "User ID" qui est une chaîne de caractères unique et enregistrée auprès d'un organe d'identification des producteurs de données lidar afin de prévenir que deux sources différentes aient la même séquence.

Chaque utilisateur dispose de 65536 enregistrements au maximum qu'il gère de manière indépendante. Les identifiants des enregistrements sont renseignés dans le champ "Record ID" de l'en-tête. Il peut également, s'il le souhaite, publier une signification liée aux identifiants choisis.

"Record Length After Header" : il s'agit d'un offset en nombre d'octets qui indique l'endroit où commencent les données après l'en-tête de 54 octets. La source des données peut renseigner des informations supplémentaires dans l'en-tête, mais elles ne suivront pas le standard de la spécification.

d. Données des points

Les données des points commencent à l'adresse du champ "Offset to Point Data" de l'en-tête de fichier. Il existe quatre formats pour les entrées de point.

Le format zéro contient les champs communs à tous. Les informations présentées sont dans

le tableau 1.4

Champ	Format de donnée	Taille	Requis
X	long	4 octets	*
Y	long	4 octets	*
Z	long	4 octets	*
Intensity	unsigned short	2 octets	
Return number	3 bits (bits 0, 1, 2)	3 bits	*
Number of returns (given pulse)	3 bits (bits 3, 4, 5)	3 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 octet	*
Scan Angle Rank (-90 to +90) – Left side	char	1 octet	*
User Data	unsigned char	1 octet	
Point Source ID	unsigned short	2 octets	*

TABLEAU 1.4 – Information d'un point présent dans un fichier **LAS**, Source : adapté de LAS Specification Version 1.2 (2008, p. 6)

Les points sont collectés à partir des impulsions de laser envoyées et détectées par un capteur le tout depuis un aéronef.

Un exemple d'appareils est celui de la figure 1.1

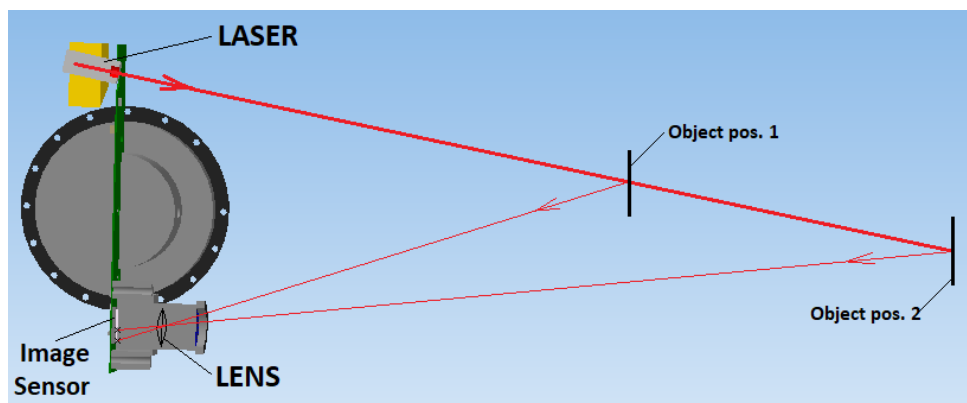


ILLUSTRATION 1.1 – Schéma représentant un instrument de mesure LIDAR. Source : tiré de réf URL02

On peut considérer une entrée dans le fichier comme une impulsion de lumière émise lors de la mesure des points. Chaque impulsion peut recevoir plusieurs retours d'où le champ "Number of returns (given pulse)". Il est aussi nécessaire de connaître l'angle de l'appareil par rapport à l'horizon afin de corriger la position du point récolté, car seul l'angle de l'instrument de mesure ne suffit pas. Cet angle est en degré et allant de -90° à $+90^{\circ}$ à partir de l'aile gauche de l'aéronef dans le sens du vol.

Le format un ajoute simplement des informations concernant le temps GPS à laquelle le point a été enregistré. Il est de même des autres format cependant ces informations ne sont pas utiles pour ce travail.

e. Classification des points

Les points collectés sont généralement classés selon le type de sol rencontré. Les valeurs se trouvent au tableau 1.6. Toutes les informations concernant le classement sont contenues en un octet et en quatre sections. Elles sont expliquées dans le tableau 1.5.

Les cinq premiers bits correspondent à une table standard des classifications. Si le bit cinq à un, le point a été créé par un logiciel de manière synthétique et si le bit sept est aussi à un, le point ne doit pas être pris en compte. Il est considéré comme étant supprimé.

0	1	2	3	4	5	6	7
N° de classification					Synthétique	Point clé	Retenu

1 octet

TABLEAU 1.5 – Représentation des bits dans l’octet de classification. Source : adapté de LAS Specification Version 1.2 (2008, p. 8)

N° de classification (bit 0 à 4)	Signification
0	Créé, jamais classé
1	Non classé
2	Sol
3	Faible végétation
4	Végétation moyenne
5	Végétation dense
6	Bâtiment
7	Bruit
8	Point de masse
9	Eau
10-11	Réservé
12	Point superposé
13-31	Réservé

TABLEAU 1.6 – Signification des valeurs de classification. Source : adapté de LAS Specification Version 1.2 (2008, p. 8)

1.2. FICHIER STÉRÉOLIGRAPHIQUE

Le format de Stéréolitographie (STL) est conçu par la société 3D System pour contenir des modèles d’objets tridimensionnels. Il a été pensé pour permettre un prototypage rapide dans les

logiciels de **Conception Assistée par Ordinateur (CAO)**. Ce format ne décrit que la géométrie de la surface d'un modèle. Il existe un format binaire et un format ASCII, le dernier laissant une empreinte dans la mémoire morte plus conséquente.

On y stocke les triangles composant le modèle où chaque sommet du triangle est décrit par ses coordonnées cartésiennes (x, y, z) . Chaque triangle partage, sans exception, deux sommets avec un triangle voisin. La coordonnée dans l'axe z est considérée comme l'axe vertical, ceci peut être gênant pour les programmes considérant l'axe y comme l'axe vertical.

a. Format ASCII

Le format commence par la séquence de caractères : "solid *nom*" où *nom* est une séquence correspondant au nom du modèle qui est facultatif. Si le nom est vide,

l'espace après "solid" est obligatoire. Un triangle s'écrit de la manière suivante :

```
facet normal   $n_x$   $n_y$   $n_z$ 

    outer loop

        vertex  $v1_x$   $v1_y$   $v1_z$ 
        vertex  $v2_x$   $v2_y$   $v2_z$ 
        vertex  $v3_x$   $v3_y$   $v3_z$ 

    endloop
endfacet
```

Les sommets sont définis à l'aide du mot clé "vertex" qui sont contenus dans un bloc loop. Les n et les v sont des nombres à virgule flottante. Ils s'écrivent dans le format "signe-mantisse-e-sign-exposant", par exemple "6.248000e-003". La fin du fichier ASCII est définie par la séquence : "end solid".

b. Format binaire

Le format binaire présente un avantage considérable par rapport au format ASCII, il est beaucoup moins volumineux.

Les 80 premiers octets sont un commentaire. Les quatre suivants sont un entier 32 bits qui indique le nombre de triangles présent dans le fichier. Chaque triangle est encodé sur 50 octets. Comme le format ASCII, les nombres sont encodés conformément à la spécification IEEE-754 et en mode little endian.

Le format dans le fichier est le suivant :

```
unsigned char[80] – header
unsigned int – total number of triangles

foreach triangle :
    float – normal vector
    float – vertex 1
    float – vertex 2
    float – vertex 3
unsigned short – control word
end
```

CHAPITRE 2 : ALGORITHMES

Ce chapitre contient les différents algorithmes rencontrés lors de l'application

2.1. NETTOYAGE

L'utilité de réduire la quantité de données sauvegardées de nos jours au vu des moyens de stockage moderne employé n'est plus une occupation majeure. Cependant, certains systèmes restent limités en mémoire et c'est à ce moment qu'intervient le filtrage de données. Le nettoyage des données **LIDAR** est utile sur des systèmes plus puissants pour accélérer l'utilisation ultérieure des fichiers. Les méthodes présentées ici peuvent être utilisées indépendamment ou bien combinées.

a. Nettoyage par classe de point

Une première méthode pour filtrer les données est de se baser sur la classification des points du chapitre 1.1. En connaissant les informations qui nous intéressent, le filtrage se fait donc lors de la lecture des données. Étant donné que la lecture se déroule de manière séquentielle, il suffit de poser une ou plusieurs conditions vérifiant la classe du point. Si la ou les conditions sont respectées, le point est collecté dans une liste de points à garder, le cas échéant étant simplement de l'ignorer et de passer au suivant. Une fois la lecture terminée, on réécrit la liste des points gardés dans un nouveau fichier **LIDAR**. Le fichier original peut être supprimé ou conservé.

b. Nettoyage par point pondéré

Une autre manière de réduire la quantité de données est de remplacer plusieurs points par un seul où l'élévation de celui-ci équivaut à la moyenne des élévations des voisins. En d'autres termes

$$P_y = \frac{1}{n} \sum_{i=0}^n p_{iy}$$

où n est le nombre de voisins d'un point P . Le choix de l'endroit où l'on créera le point peut être déterminé à l'aide d'une condition, p. ex.

$$\sum_{i=0}^n dist(P, p_i) \geq \epsilon$$

Un effet de bord de cette méthode est le lissage des surfaces plates. Un exemple serait les illustrations 2.1 et 2.2 où l'on peut observer qu'après une reconstruction de surface, on aperçoit que les points ont une tendance à moins osciller dans l'axe vertical.

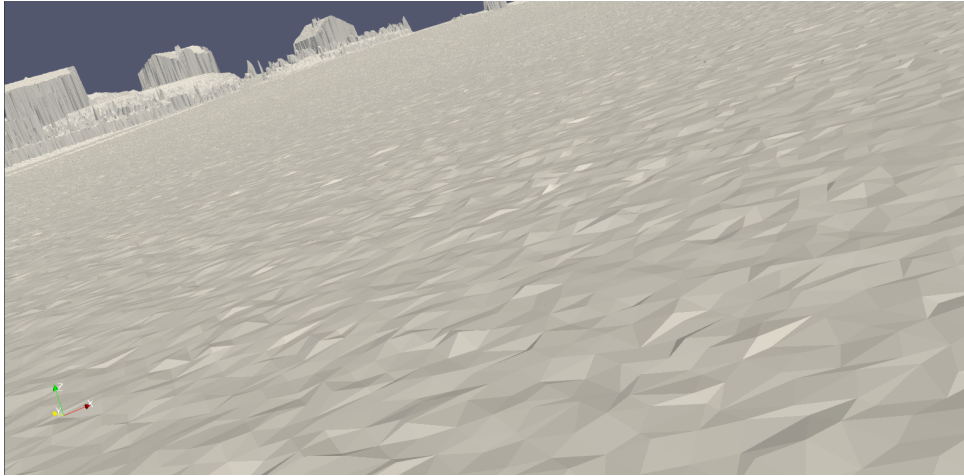


ILLUSTRATION 2.1 – Surface plane avant le passage par un filtre. Source : réalisé par Jérôme Chételat



ILLUSTRATION 2.2 – Surface plane après le passage par un filtre. Source : réalisé par Jérôme Chételat

c. Filtre par point moyen

Ce filtre se base sur l'utilisation de l'intensité de retour d'une impulsion. Les points à retour multiple peuvent être nombreux en présence d'arbres. Au lieu d'avoir plusieurs points de retour, on les remplace par un seul point qui prend une moyenne de toutes les positions de retour de la

même impulsion. En terme mathématique, on a donc :

$$P = \begin{pmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{pmatrix} = \frac{1}{n} \sum_{i=0}^n \begin{pmatrix} p_{ix} \\ p_{iy} \\ p_{iz} \end{pmatrix}$$

Où n désigne le nombre de retour d'une impulsion et p les points de l'impulsion.

2.2. TRIANGULATION DE DELAUNAY

La triangulation de Delaunay est une manière de créer des triangles entre des points dispersés dans l'espace. Cette manière de trianguler les points permet de créer un résultat plus homogène que d'autres méthodes.

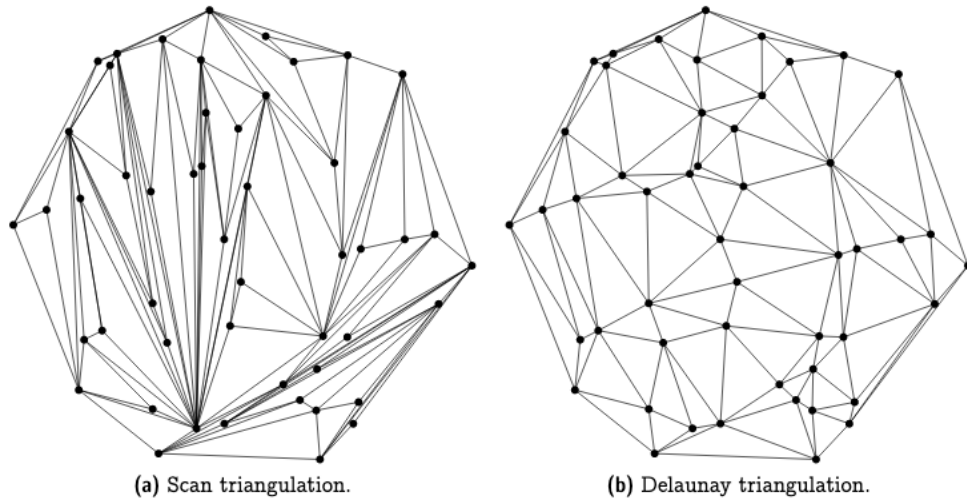


ILLUSTRATION 2.3 – Deux triangulations du même jeu de données. Source : tiré de ref URL01

a. Bowyer-Watson

Il existe plusieurs algorithmes pour recréer des triangles entre des points. Il s'agit de l'algorithme de Bowyer-Watson, un algorithme itératif découvert par Adrian Bowyer et David Watson. Le principe de ce procédé se repose sur l'ajout progressif de points dans la triangulation. Après chaque ajout, de nouveaux triangles sont formés à partir du point ajouté et des sommets du triangle contenant le point.

Soit $\mathcal{P} \subset \mathbb{R}^2$ l'ensemble des points à trianguler et \mathcal{T} les triangles appartenant à la triangulation. On construit dans un premier temps, un triangle $S \in \mathcal{T}$ tel que $\mathcal{P} \subset S$. On nomme

ce triangle le "super-triangle". Il doit contenir en son sein l'ensemble des points \mathcal{P} . On ajoute donc trois points, les sommets de S dans \mathcal{P} .

On prend ensuite un point de \mathcal{P} puis on détermine dans quel triangle il se situe. Pour connaître l'appartenance d'un point à un triangle, on détermine si un point est contenu dans le cercle circonscrit d'un triangle. La condition qui permet de déterminer si un point D est contenu dans le cercle-circonscris de $\triangle ABC$ est la suivante :

$$\begin{vmatrix} A_x - D_x & A_y - D_y & (A_x - D_x)^2 + (A_y - D_y)^2 \\ B_x - D_x & B_y - D_y & (B_x - D_x)^2 + (B_y - D_y)^2 \\ C_x - D_x & C_y - D_y & (C_x - D_x)^2 + (C_y - D_y)^2 \end{vmatrix} \geq 0$$

Si la condition est vraie, le point est relié au sommet du triangle qui le contient. Les triangles formés désormais appartiennent à \mathcal{T} .

On répète les opérations d'ajout de points et de création de triangles jusqu'à ce qu'on ait parcouru tous les points de \mathcal{P}

Pour finir, on supprime les sommets du super-triangle ainsi que ses arrêtes de \mathcal{P} et de \mathcal{T} .

Le résultat est un ensemble de triangles \mathcal{T} représentant la géométrie des surfaces appelées triangulation de Delaunay. Ce résultat est unique pour un ensemble de points \mathcal{P} donné. En voici un exemple dans la figure 2.4.

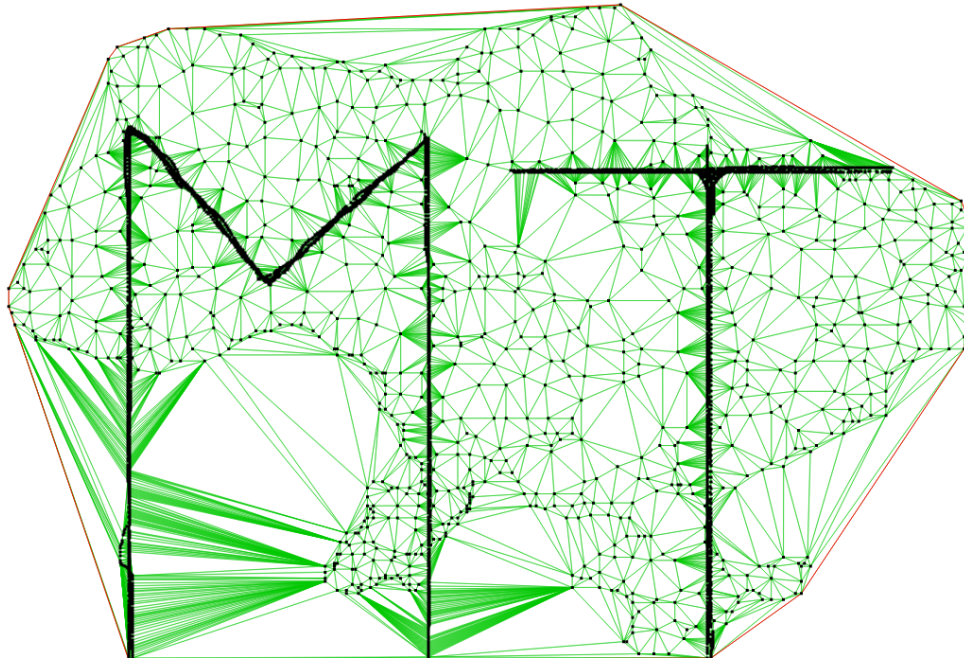


ILLUSTRATION 2.4 – Maillage résultant d'une triangulation de delaunay. Source : réalisé par Jérôme Chételat

b. Lee & Scachter

L'algorithme de Lee et Scachter est basé sur le principe de "Divide and conquer". Le principe consiste à former des petites triangulations à partir d'un espace divisé selon un axe. On applique une triangulation pour la première itération. Ensuite, on fusionne les triangulations entre elles et l'on répète ces étapes jusqu'à ne plus pouvoir en fusionner. Le résultat final est un maillage considéré comme étant triangulation de Delaunay.

2.3. FUSION DE MAILLAGE

L'algorithme présenté dans cette section concerne la fusion de maillage. Dans l'application du travail de bachelor, il se peut que l'on ait déjà des triangulations existantes et que l'on souhaite afficher un résultat de la fusion des deux triangulations. Il s'agit en réalité que d'une partie de l'algorithme de triangulation par Lee et Scachter basé sur le principe de "Divide and conquer".

Soit deux triangulations M_L et M_R linéairement séparable par une droite g . On cherche en premier les sites aux extrémités inférieures selon l'axe y et on les relie par une arête coupant la droite g . L'arête créée est nommée "base LR". On a donc une situation similaire à l'illustration 2.5

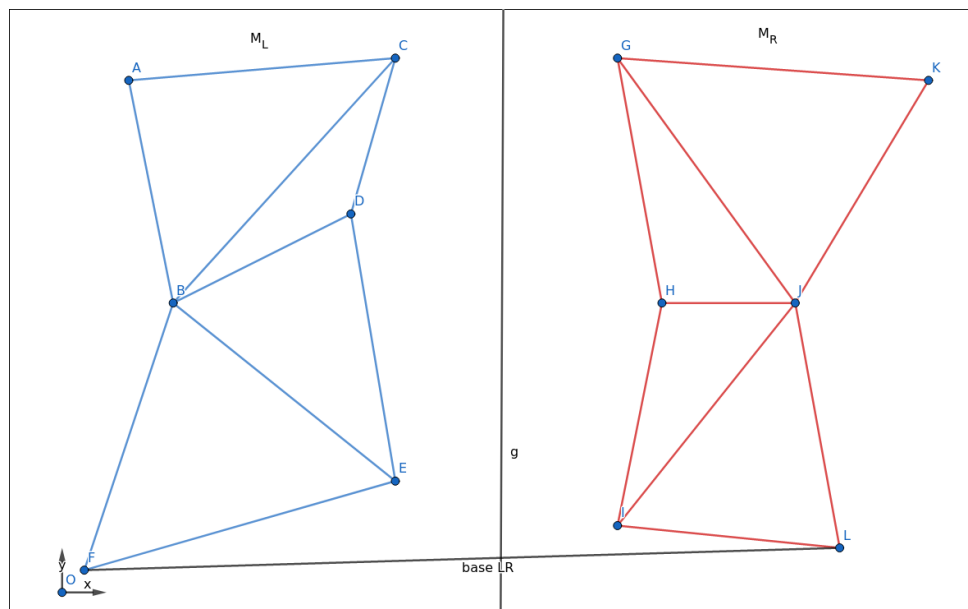


ILLUSTRATION 2.5 – Création de la base LR entre les deux maillages. Source : réalisé par Jérôme Chételat

On poursuit par la recherche d'un site candidat voisin d'une des extrémités de la base LR

qui respecte les conditions suivantes :

1. L'angle ,dans le sens antihoraire pour un site de M_R et dans le sens horaire pour un site de M_L , par rapport à la base LR est inférieur à 180° .
2. Le cercle circonscrit défini par les deux extrémités de la base et l'une des extrémités avec un candidat ne contient pas un prochain candidat potentiel en son sein.

Si les deux conditions sont satisfaites, le site devient un candidat final pour le maillage en question. Si la première condition n'est pas respectée, aucun candidat pour le maillage n'est choisi et l'arrête reliant le candidat actuel avec la base est supprimé de la triangulation.

Le processus de choix d'un candidat se poursuit tant qu'aucun candidat final ne soit choisi ou qu'on détermine qu'aucun candidat ne peut être choisi.

Les illustrations 2.6 et 2.7 représente le choix d'un candidat final pour chaque maillage. Cela ne représente que la première itération.

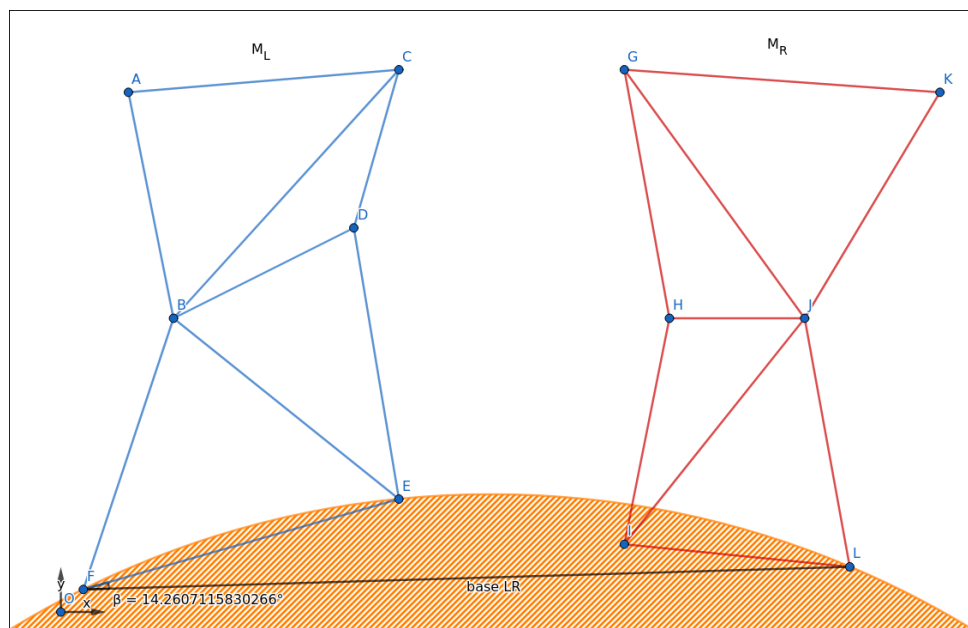


ILLUSTRATION 2.6 – Candidat final du maillage M_L . Source : réalisé par Jérôme Chételat

Une fois que les candidats pour chaque maillage ont été déterminés, si un seul candidat final est présent, on lie le candidat par une arête avec le site de la base du maillage opposé. Dans le cas où deux candidats sont choisis, on détermine un dernier test. On vérifie qu'un candidat final n'est pas contenu dans le cercle circonscrit de l'autre. Si c'est le cas, le candidat qui contient le site dans le cercle circonscrit est alors abandonné et celui qui n'en contient pas est utilisé pour créer une nouvelle base. On peut observer que dans la figure 2.6 le site I est contenu dans le

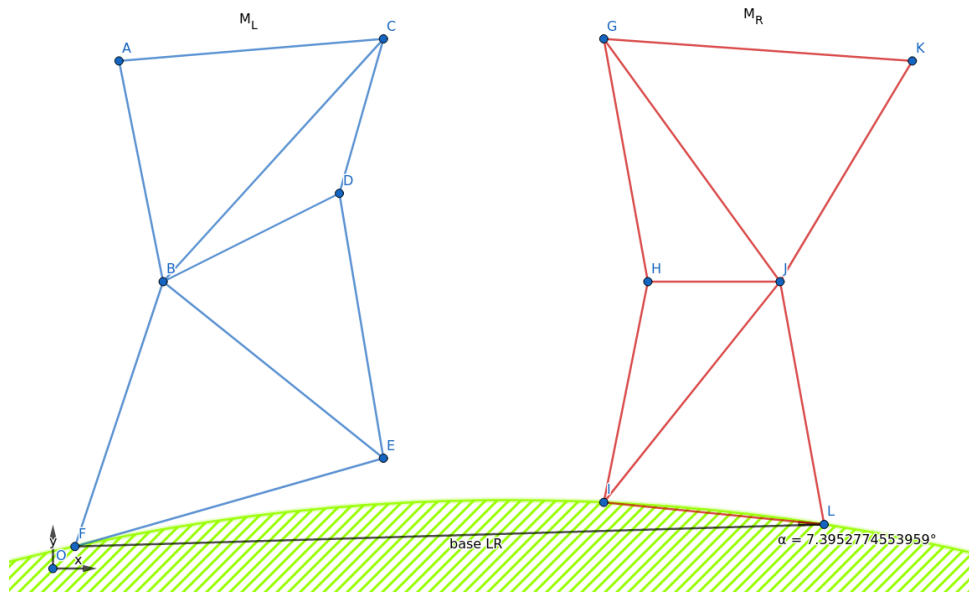


ILLUSTRATION 2.7 – Candidat final du maillage M_R . Source : réalisé par Jérôme Chételat

cercle circonscrit formé par les points F , E et L . Dans ce cas, le site E n'est pas maintenu en tant que candidat final et une arête entre I et F est créée. Celle-ci devient alors la nouvelle base.

Les opérations de recherche de candidat et de création d'arêtes continuent jusqu'à ce que plus aucun candidat final ne soit obtenu.

CHAPITRE 3 : IMPLÉMENTATIONS

Dans ce chapitre, on détaillera l'architecture du projet ainsi que les technologies et les bibliothèques utilisées pour développer les différentes parties de l'application.

3.1. RUST

Rust est un langage de programmation système à usage général². C'est un langage moderne et mutiparadigme. Grâce au principe de *ownership* ou propriété des données, Rust empêche un grand nombre de bogues lors de l'écriture du code. Dans la nomenclature de Rust, le mot "crate" désigne un paquet ou une bibliothèque externe. Une particularité de Rust est le **ownership** ou la propriété des données.

Ce langage permet une compilation vers les plateforme Windows, MacOS et Linux ainsi que le web grâce au [WebAssembly \(WASM\)](#). :

3.2. ARCHITECTURE

L'application est composée de quatre modules principaux.

- Une bibliothèque regroupant des fonctions et des structures communes.
- Des utilitaires, ce sont les outils en ligne de commande qui implémentes les pipelines
- Un serveur web, celui-ci expose des fichiers pour le client web à travers une api [Representational State Transfer \(REST\)](#)
- Un client web permettant à un utilisateur de visualiser des maillages dans un navigateur web.

En ce qui concerne le module *utilitaire*, l'architecture en pipeline est utilisé pour tout les binaires. Il est composé de 4 étapes.

Chacun des utilitaires adapte ce pipeline pour traiter des données spécifique.

3.3. BIBLIOTHÈQUE

La bibliothèque contient les outils de lecture et d'écriture de fichiers pour les formats du chapitre 2.

Elle a comme dépendance principale les crates *stl-io* qui implémentes les structures pour écrire les fichiers *STL* et *las-rs* qui expose les structures et méthodes liées aux données lidar.

2. Peut être utilisé dans l'écriture de kernels ou d'applications

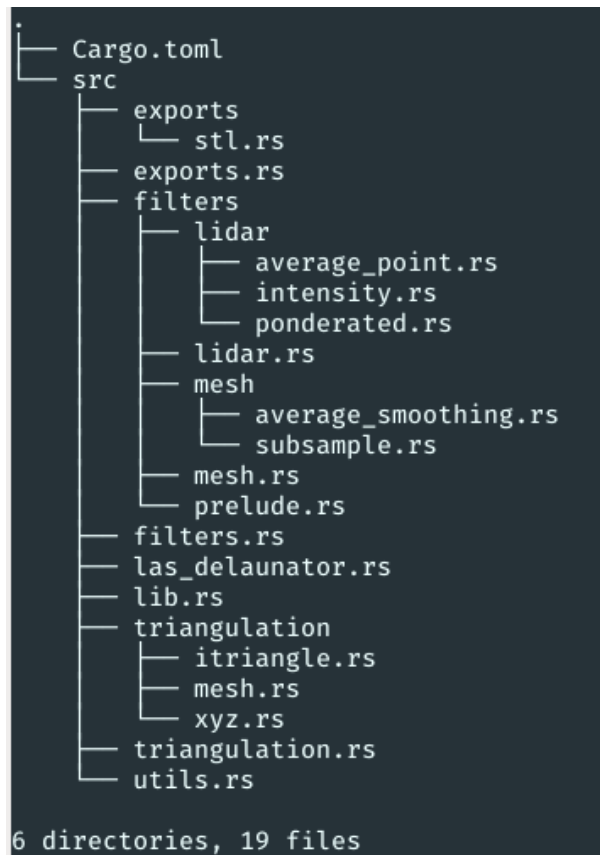


ILLUSTRATION 3.1 – Structure de fichier présent dans le module de la librairie de l’application.
Source : réalisé par Jérôme Chételat

a. Nettoyage

Les implémentations des filtres sont dérivées du trait **MeshFilter** ou du trait **LidarFilter**. Leurs définitions sont les suivantes :

```

1 pub trait MeshFilter {
2     fn apply(&self, mesh: IndexedMesh, epsilon: f32) -> IndexedMesh;
3 }
4
5 pub trait LidarFilter {
6     fn apply(&self, points: Vec<Point>) -> Vec<Point>;
7 }

```

Les traits sont ensuite spécialisés dans le type de filtre voulu.

b. Triangulation

La librairie contient un module de triangulation. On y trouve les implémentations des algorithmes de triangulation réalisés lors du projet de semestre, mais également un algorithme de fusion de triangulation.

c. Fusion de triangulation

Pour ce qui est de la fusion, une structure propre à la librairie était nécessaire en plus des structures de données offertes par la crates stl-io. Elle est définie de la manière suivante :

```

1 pub struct Mesh {
2     /// Vertices composing the mesh
3     pub vertices: Vec<XYZ>,
4     /// Indexed triangles of the mesh
5     pub faces: Vec<usize>,
6     /// Indexed vertices part of the convex hull of the mesh
7     pub hull: Option<Vec<usize>>,
8     /// The neighbourhood of each indexed point
9     pub neighbourhood: Option<Vec<HashSet<usize>>>,
10 }

```

La particularité de cette structure est qu'on y trouve un champ pour le voisinage du maillage qui est "neighbourhood" et second champ pour l'enveloppe convexe "hull". Ils sont optionnels pour maintenir une compatibilité avec les crates utilisées dans la librairie. Une des raisons vient du fait que lors de la lecture d'un fichier **STL**, on ne retrouve pas ces champs.

Une méthode utile pour s'assurer de la présence d'un voisinage est d'appeler la fonction `neighbourhood`. Elle renvoie le voisinage d'un maillage s'il est déjà présent, le cas échéant est de calculer les voisins.

Son implémentation est disponible dans l'annexe 3. Elle parcourt les différents triangles du maillage et pour chaque point présent du triangle, on ajoute ses deux voisins dans une liste.

Cette dernière peut cependant être améliorée car actuellement, une copie du voisinage est renvoyée à l'appelant. Ce qui a pour effet de créer des données dupliquées. Ce choix a été fait car l'utilisation du voisinage dans un algorithme de fusion nécessite de muter certaine partie. Une solution pour éviter ces duplications serait de créer une structure indiquant les parties qui ont mutés et de rendre la structure `Mesh` immutable. Lors d'un prochain appel de la fonction, cette dernière pourrait renvoyer un voisinage ayant les changements appliqués à partir de la structure précédemment décrite.

Concernant le champ `hull`, aucune méthode n'est disponible dans la librairie pour en calculé. Il est possible de le calculé au travers de la crate "ncollide" qui expose une version de `quickhull`, un algorithme de calcul d'enveloppe convexe.

Une fonction intéressante est la recherche d'un candidat final dans un maillage. Voici la signature de la fonction :

```
1 fn get_side_candidate(
2     mesh: &mut Mesh,
3     lr_indexed_point: usize,
4     other_lr_point: &XYZ,
5 ) -> Option<usize>;
```

Elle prend en paramètre un point indexé par un entier non signé dans le maillage passé en argument et une structure de point pour effectuer la recherche et retourne éventuellement un index pour le site choisi.

Une grosse partie de la fusion est réalisée dans la méthode **`merge`** avec comme signature :

```
1 /// Returns a Ok(Mesh) containing the fusion of two meshes together.
```

```
2 /// It needs the convex hull of each mesh.
3 pub fn merge(&mut self, other: &mut Mesh) -> Result<Mesh, &'static str>;
```

La méthode renvoie un maillage dans le cas où toute la fusion se serait bien déroulée sinon elle renvoie un message d'erreur.

Une partie intéressante de l'implémentation est la création des arêtes entre la base et un candidat.

Un point intéressant est l'utilisation de pattern matching provenant de langage de programmation fonctionnel pour vérifier les conditions énumérées dans le chapitre 2.3. Si un candidat final est choisi, il est ajouté à une liste d'arête **edges** et à la fin de l'opération, cette liste est utilisée afin de construire les nouveaux triangles entre les deux maillages.

3.4. UTILITAIRES

Ce module contient les binaires en ligne de commande pour traiter les données du chapitre 1. Chaque binaire utilise l'architecture en pipeline car c'est ce qui s'applique le mieux pour l'utilisation des outils.

Chaque utilitaire utilise une crate pour vérifier et extraire les arguments donnés à ce dernier. La crate se nomme *clap* et grâce à une macro de la crate, il est possible de définir simplement les arguments acceptés par l'utilitaire.

Voici la liste des commandes disponibles :

- mesh-merge
- las-delaunator
- las-geometry
- las-filtering
- las-

a. mesh-filter

b. mesh-merge

3.5. STACK WEB

a. Serveur

b. Client

Le client web a été écrit en Rust et compilé en [WASM](#). Les rendu des maillages se fait par l'api WebGL. WebPack, un packager d'application web, a été utilisé pour lier le HTML, le JavaScript et les modules [WASM](#).

Cette partie utilise grandement une crate appelée "web-sys" qui met à disposition des bindings pour le JavaScript. On utilise également la crate "nalgebra" qui propose des fonctions pour manipuler des matrices 4x4 utilisées dans les [Application Programming Interface \(API\)](#) graphique.

Une contrainte survient en écrivant du code pour le web avec Rust est que la plupart des fonctionnalités pour améliorer le fonctionnement des modules sont à ce jour encore expérimental et de ce fait, pas suffisamment stable pour créer une application complète.

N'étant pas assez optimisé, l'affichage d'une région entière d'un fichier [LIDAR](#) ou sa version triangulée est impossible malgré une implémentation en [WASM](#) afin de bénéficier d'une performance supplémentaire par rapport à une version qui serait écrite en JavaScript.

CHAPITRE 4 : RÉSULTATS

Dans ce chapitre, nous allons détailler les résultats obtenus par les différentes parties de l'application.

4.1. TRIANGULATION

La triangulation implémentée nous provient en partie du projet de semestre.

4.2. NETTOYAGE

4.3. STACK WEB

a. Serveur Web

Le serveur actuel permet les téléchargements de fichiers [STL](#) présent

b. Client Web

Les tests suivants ont été effectués sur le navigateur Firefox version 81. Le client web permet actuellement de faire un rendu d'un maillage dans un navigateur web. Un fichier [STL](#) est téléchargé à l'aide de l'api *fetch* du navigateur au chargement de la page web.

Amélioration possible

Le chargement des vertex du maillage se faisant dans un seul file d'exécution, la page lors du chargement reste figée et peut ne plus répondre. Il serait possible dans un futur proche de faire ces opérations dans des WebWorkers. Au moment de l'écriture de ce mémoire, les possibilités d'utiliser des WebWorkers en Rust est encore expérimental.

CONCLUSION

Ce travail avait pour objectif de créer et de mettre à disposition des outils pour manipuler les données **LIDAR** ainsi qu'une interface web pour les afficher. Les outils devaient pouvoir filtrer des données LIDAR et les trianguler de manière efficace, de filtrer et fusionner des maillages et pour cela, un travail de recherche sur les formats de données a été réalisé pour tirer la meilleure partie des informations à disposition. Les fonctionnalités ont été implémentées dans le langage Rust et grâce à cela, les différents outils bénéficient d'une garantie de ne pas avoir de fuites mémoires. L'interface web a été implémentée aussi en Rust et compilée en **WASM** afin de bénéficier de performances supplémentaires pour l'affichage des jeux de données. La réalisation de ce travail, m'a permis, en plus de mettre en oeuvre l'enseignement reçu lors de ma formation, mais encore d'avoir pu enrichir mes connaissances dans le domaine de la topographie et du langage de programmation Rust que je compte désormais approfondir encore plus. Un challenge particulier lors de ce travail était le confinement dû au covid19. Mon environnement de travail n'était pas propice à la productivité due aux bruits des personnes vivant sous le même toit me gênant dans ma concentration. Ce fut un gros challenge de devoir travailler uniquement à la maison sans pouvoir avoir du calme.

Le travail n'étant pas complètement terminé, notamment au niveau du client web, un futur travail pourrait reprendre le projet et continuer de l'améliorer avec des fonctionnalités stables. Des outils supplémentaires peuvent être ajoutés pour compléter la panoplie de l'application notamment au travers de services web.

ANNEXE 1 : TABLEAU DES CHAMPS DE L'EN-TÊTE DE FICHIER LAS

Champ	Type de donnée	Taille (Octets)	Requis
File Signature ("LASF")	char[4]	4	*
File Source ID	unsigned short	2	*
Global Encoding	unsigned short	2	
Project ID - GUID data 1	unsigned long	4	
Project ID - GUID data 2	unsigned short	2	
Project ID - GUID data 3	unsigned short	2	
Project ID - GUID data 4	unsigned char[8]	8	
Version Major	unsigned char	1	*
Version Minor	unsigned char	1	*
System Identifier	char[32]	32	*
Generating Software	char[32]	32	*
File Creation Day of Year	unsigned short	2	
File Creation Year	unsigned short	2	
Header Size	unsigned short	2	*
Offset to point data	unsigned long	4	*
Number of Variable Length Records	unsigned long	4	*
Point Data Format ID (0-99 for spec)	unsigned char	1	*
Point Data Record Length	unsigned short	2	*
Number of point records	unsigned long	4	*
Number of points by return	unsigned long[5]	20	*
X scale factor	double	8	*
Y scale factor	double	8	*
Z scale factor	double	8	*
X offset	double	8	*
Y offset	double	8	*
Z offset	double	8	*
Max X	double	8	*
Min X	double	8	*
Max Y	double	8	*
Min Y	double	8	*
Max Z	double	8	*
Min Z	double	8	*

TABLEAU 4.1 – Champs de l'en-tête d'un fichier LAS

ANNEXE 2

```

1 loop {
2     let candidate_right = Mesh::get_side_candidate(
3         other,
4         r_bs,
5         &self.vertices[l_bs]);
6     let candidate_left = Mesh::get_side_candidate(
7         self,
8         l_bs,
9         &other.vertices[r_bs]);
10    let candidates = (candidate_left, candidate_right);
11    match candidates {
12        (Some(l), Some(r)) => {
13            // Check that one of these does not contain the other one
14            let left_candidate_inside = self
15                .vertices[l]
16                .inside_circum_circle((
17                    &self.vertices[l_bs],
18                    &other.vertices[r_bs],
19                    &other.vertices[r],
20                ));
21            let right_candidate_inside = other
22                .vertices[r]
23                .inside_circum_circle((
24                    &self.vertices[l],
25                    &self.vertices[l_bs],
26                    &other.vertices[r_bs],
27                ));
28            match (left_candidate_inside, right_candidate_inside) {
29                // If both are contained in each other

```

```

30         // it is a degenerate case and there
31         // is no possible outcome
32         // to have a good Delaunay triangulation
33         (false, false) =>
34             return Err(
35                 "Edge_case_found_about_4_co-circular_points"
36             ),
37         (true, false) => {
38             edges.push(l);
39             edges.push(r_bs);
40             l_bs = l;
41         }
42         (false, true) => {
43             edges.push(l_bs);
44             edges.push(r);
45             r_bs = r;
46         }
47         _ => break,
48     }
49 }
50 (Some(l), None) => {
51     // Connect it to the lr base,
52     // push the new triangle and move the l_bs reference
53     edges.push(l);
54     edges.push(r_bs);
55     l_bs = l;
56 }
57 (None, Some(r)) => {
58     // Connect it to the lr base,
59     // push the new triangle and move the r_bs reference
60     edges.push(l_bs);

```

```
61         edges.push(r);
62         r_bs = r;
63     }
64     _ => {
65         // The merging is done if it doesn't exists anymore candidates
66         break;
67     }
68 }
69 }
```

ANNEXE 3

```

1 pub fn neighbourhood(&mut self) -> Option<Vec<HashSet<usize>>> {
2     if self.neighbourhood.is_some() {
3         return self.neighbourhood.clone();
4     }
5     let mut dict: Vec<HashSet<usize>> =
6         vec![HashSet::new(); self.vertices.len()];
7
8     // Iterating over all triangles in the mesh
9     self.faces
10        .iter()
11        .enumerate()
12        .step_by(3)
13        .for_each(|(i, _): (usize, &usize)| {
14            // Adding the neighbours of the first vertex
15            dict[self.faces[i]].insert(self.faces[i + 1]);
16            dict[self.faces[i]].insert(self.faces[i + 2]);
17
18            // Adding the neighbours of the second vertex
19            dict[self.faces[i + 1]].insert(self.faces[i]);
20            dict[self.faces[i + 1]].insert(self.faces[i + 2]);
21
22            // Adding the neighbours of the third vertex
23            dict[self.faces[i + 2]].insert(self.faces[i]);
24            dict[self.faces[i + 2]].insert(self.faces[i + 1]);
25        });
26
27     self.neighbourhood = Some(dict.clone());
28     Some(dict)
29 }

```


BIBLIOGRAPHIE

- COMPUTING CONSTRAINED DELAUNAY TRIANGULATIONS IN THE PLANE*, [p. d.] [en ligne] [visité le 2020-06-16]. Disp. à l'adr. : http://www.geom.uiuc.edu/~samuelp/del_project.html.
- DESGOGUES, Pascal, 1996. *Triangulations et quadriques* [en ligne] [visité le 2020-04-06]. Disp. à l'adr. : <https://tel.archives-ouvertes.fr/tel-00771335>. Thèse de doct. Université Nice Sophia Antipolis.
- LEMAIRE, Christophe, [p. d.]. Triangulation de Delaunay et arbres multidimensionnels, p. 227.
- LIDAR Filters*, 2018 [en ligne] [visité le 2020-04-23]. Disp. à l'adr. : <https://www.alluxa.com/optical-filter-applications/lidar-filters/>. Library Catalog : www.alluxa.com Section : Learning Center.
- MÄKELÄ, I, [p. d.]. Some Efficient Procedures for Correcting Triangulated Models, p. 9.
- Streaming Delaunay : I/O and memory efficient Computation of Delaunay Triangulations*, [p. d.] [en ligne] [visité le 2020-05-22]. Disp. à l'adr. : <http://www.cs.unc.edu/~isenburg/sd/>.
- TATE, Nicholas ; BRUNSDON, Chris ; CHARLTON, Martin ; FOTHERINGHAM, Alexander ; JARVIS, Claire, 2005. Smoothing/filtering LiDAR digital surface models. Experiments with loess regression and discrete wavelets. *Journal of Geographical Systems*. T. 7, p. 273-290. Disp. à l'adr. DOI : 10.1007/s10109-005-0007-4.