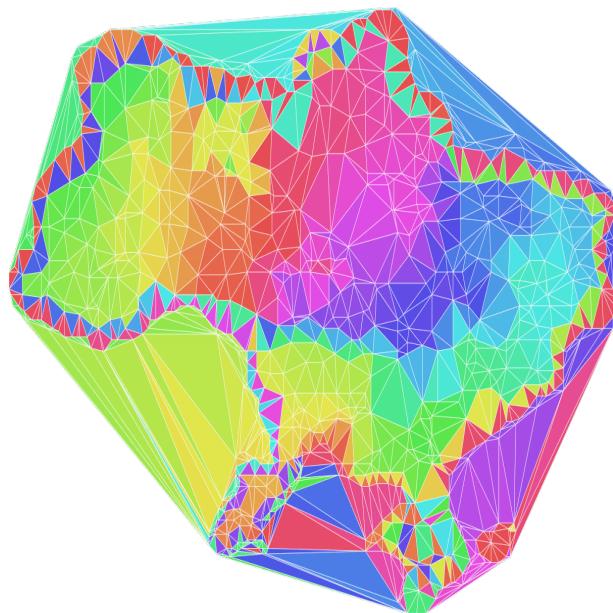


Application de triangulation de points lidar sur le canton de Genève



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	xi
Liste des tableaux	xii
Liste des annexes	xiii
Introduction	1
0.1 Nuage de points	3
a Format LIDAR	3
b En-tête public	4
c Enregistrements à longueur variable	4
d Données des points	5
e Classification des points	7
0.2 Maillage	7
a Fichier Stéréoligraphique	7
b Format ASCII	8
c Format binaire	9
1 Chapitre 2 : Algorithmes	10
1.1 Nettoyage	10
a Lidar	10
b Maillage	12
1.2 Triangulation de Delaunay	13
a Bowyer-Watson	14
b Lee & Schachter	16
1.3 Fusion de maillage	17
2 Chapitre 3 : Implémentations	20
2.1 Rust	20
a Rust is the new JavaScript	20
2.2 Architecture	20
2.3 Bibliothèque	21
a Nettoyage	22
b Triangulation	22
2.4 Utilitaires	25
a mesh-smoothing	25
b mesh-subsample	26
c mesh-merge	26
2.5 Stack Web	26

a	Serveur	26
b	Client	27
3	Chapitre 4 : Résultats	29
3.1	Triangulation	29
a	Algorithme de Bowyer-Watson	29
b	Algorithme Delaunator	29
c	Fusion de triangulation	30
3.2	Nettoyage	30
a	Lidar	31
b	Lissage de maillage	31
c	Décimation de maillage	33
3.3	Stack Web	33
a	Serveur web	33
b	Client web	33
Conclusion	35	
Bibliographie	45	

REMERCIEMENTS

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

Je remercie aussi M. Gabriel Strano pour son aide dans la rédaction du code et la relecture de mon mémoire ainsi que ses précieux conseils pour la rédaction de celui-ci.

Je remercie M. Sébastien Olquist-Cartier pour la relecture de mon mémoire et les corrections apportées à celui-ci.

Je tiens à remercier également Emmanuelle Moreau pour la relecture de mon mémoire ainsi que pour son soutien et ses nombreux encouragements qu'elle m'a prodigués 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

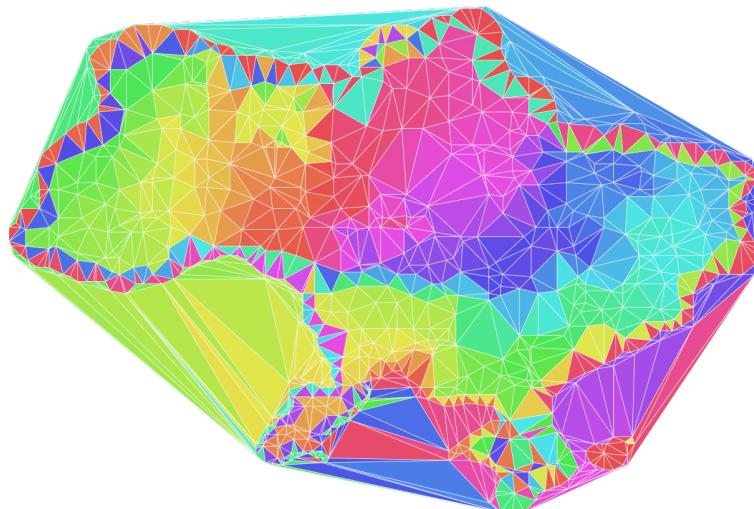
En collaboration avec :

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 naviguer et à définir des chemins à travers le monde. 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 les 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 véhicules terrestres. Un problème fréquent lié à ces données est le traitement réalisé avant leur utilisation 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 du 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 leur utilisation 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 produits par les transformations au travers d'un client web. Il est également focalisé sur la conception d'outils de traitement de maillage résultante 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. 27

ASPRS American Society for Photogrammetry and Remote Sensing. 3

CAO Conception Assistée par Ordinateur. 7

CORS Cross-Origin Resource Source. 27, 33

GPS Global Positioning System. 3, 7

LAS Laser Format. xii, 3, 4, 6

LIDAR Light Detection And Ranging. 3, 10, 35

REST Representational State Transfer. 20, 27

SITG Système d'information du territoire à Genève. 1, 29

STL Stéréolitographie. 7, 21, 23, 27, 31, 33, 35

WASM WebAssembly. 20, 27

LISTE DES ILLUSTRATIONS

1	Schéma représentant un instrument de mesure LIDAR. Source : tiré de réf URL02	6
1.1	Bruit sur des données lidar. Source : tiré de ref URL01	10
1.2	Retours d'une impulsion de lidar. Source : tiré de ref URL04	11
1.3	Surface plane avant le nettoyage. Source : réalisé par Jérôme Chételat	12
1.4	Surface plane après le nettoyage. Source : réalisé par Jérôme Chételat	13
1.5	Deux triangulations appliquées sur même jeu de données. Source : tiré de ref URL03	13
1.6	Création du super-triangle englobant l'ensemble de points \mathcal{P} . Source : réalisé par Jérôme Chételat	14
1.7	Ajout du premier points de dans la triangulation. Source : réalisé par Jérôme Chételat	15
1.8	Ajout des arrêtes reliant les sommets du super-triangle au point ajouté dans la triangulation. Source : réalisé par Jérôme Chételat	15
1.9	Identification des arrêtes liées aux sommets du super-triangle en rouge . Source : réalisé par Jérôme Chételat	16
1.10	Maillage résultant d'une triangulation de delaunay. Source : réalisé par Jérôme Chételat	17
1.11	Création de la base LR entre les deux maillages. Source : réalisé par Jérôme Chételat	18
1.12	Candidat final du maillage M_L . Source : réalisé par Jérôme Chételat	18
1.13	Candidat final du maillage M_R . Source : réalisé par Jérôme Chételat	19
2.1	Structure de fichier présent dans le module de la librairie de l'application. Source : réalisé par Jérôme Chételat	21
2.2	Structure de fichier présent dans le module de la librairie de l'application. Source : réalisé par Jérôme Chételat	21
3.1	Flamegraph du binaire mesh-merge. Source : réalisé par Jérôme Chételat . . .	30
3.2	Maillage bruité avec une route avant lissage. Source : réalisé par Jérôme Chételat	32
3.3	Maillage avec une route après lissage. Source : réalisé par Jérôme Chételat . . .	32

3.4 Différence d'élévation des maillages après lissage. Vert : maillage de base. Source : réalisé par Jérôme Chételat	33
4.1 Configuration de l'ordinateur de tests. Source : réalisé par Jérôme Chételat	42
4.2 Fichier lidars brute avec les batiments de l'HEPIA . Source : réalisé par Jérôme Chételat	43
4.3 Fichier lidars avec HEPIA après nettoyage par point moyen . Source : réalisé par Jérôme Chételat	43
4.4 Fichier lidars avec HEPIA après nettoyage par point pondéré . Source : réalisé par Jérôme Chételat	44
4.5 Résultat du rendu d'un maillage par le client web. Source : réalisé par Jérôme Chételat	45

Références des URL

- URL01 https://www.researchgate.net/figure/a-Measurement-errors-from-LiDAR-sensor-and-b-noise-removed-with-pre-processing-step_fig2_323946774
- URL02 <https://hackaday.io/project/20628-open-simple-lidar>
- URL03 <https://www.ti.inf.ethz.ch/ew/Lehre/CG13/lecture/Chapter%206.pdf>
- URL04 <https://www.vd.ch/themes/territoire-et-construction/informations-sur-le-territoire/geodonnees/altimetrie-lidar/technologie-lidar/>

LISTE DES TABLEAUX

1	Sections composant un fichier au format LAS	3
2	Type de données de la définition du format LAS	4
3	Champs de l'en-tête des enregistrements de longueur variable. Source : adapté de LAS Specification Version 1.2 (2008, p. 6)	5
4	Information d'un point présent dans un fichier Laser Format (LAS), Source : adapté de LAS Specification Version 1.2 (2008, p. 6)	6
5	Représentation des bits dans l'octet de classification. Source : adapté de LAS Specification Version 1.2 (2008, p. 8)	7
6	Signification des valeurs de classification. Source : adapté de LAS Specification Version 1.2 (2008, p. 8)	8
3.1	Résultats de la triangulation avec l'algorithme Bowyer-Watson	29
3.2	Résultats de la triangulation avec l'algorithme delaunator	30
3.3	Résultats du nettoyage par point moyen	31
3.4	Résultats du nettoyage par point pondéré	31
4.1	Champs de l'en-tête d'un fichier LAS	37

LISTE DES ANNEXES

4 Annexes	36
Annexe 1 : Tableau des champs de l'en-tête de fichier LAS	37
Annexe 2 : Implémentation de la recherche de site candidat pour la fusion	38
Annexe 3 : Implémentation du voisinage d'un maillage	41
Annexe 4 : Configuration de l'ordinateur de tests	42
Annexe 5 : Fichiers lidars avant et après nettoyage	43
Annexe 5 : Résultat d'un rendu de maillage dans un navigateur avec le client web . .	45

INTRODUCTION

Contexte

De 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 est actuellement une des informations collectées les plus massives. Ces informations sont utilisées dans l'étude topographique de régions, dans les géosciences, dans la science environnementale ou bien encore, viennent en aide au guidage automatique de véhicules terrestres. Un problème fréquent lié à ces données est le traitement réalisé avant leur utilisation 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 du 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 leur utilisation pour reconstruire des surfaces. Les méthodes automatisées n'étant pas encore 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 utilisées 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 sur un stack d'application entier (un seul langage pour le serveur et le client).

Méthodologie

Dans un premier temps nous avons rechercher des algorithmes de traitement de données lidar. Nous allons par la suite implémenter les algorithmes que nous considérons comme judicieux au sein de notre application. Enfin nous allons tester ces algorithmes sur un jeu de données et en ressortir des mesures de performance.

Dans ce mémoire, nous allons en un premier détailler les formats de fichier rencontrés dans l'application. Ensuite nous détaillerons les méthodes algorithmiques mises en oeuvre pour réaliser des traitements sur lesdits fichiers. Enfin nous verrons comment ces dernières ont été implémentées dans l'application puis nous comparerons leurs résultats. Les sources des implantations sont disponibles à l'adresse : <https://githepia.hesge.ch/jerome.chetelat/meshtools>

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

0.1. NUAGE DE POINTS

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 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 mais ce document va se concentrer sur la version 1.2 uniquement.

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 – 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 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 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 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 public 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. Il

1. selon le standard IEEE 754

doit être lu de manière séquentielle pour cause, la taille variable des données l'impose. Chaque entrée est débutée par un en-tête de 54 octets selon le tableau 3

Champ	Type de donnée	Taille (Octets)	Requis
Réserve	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 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 organisme 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 55 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 4

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

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

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

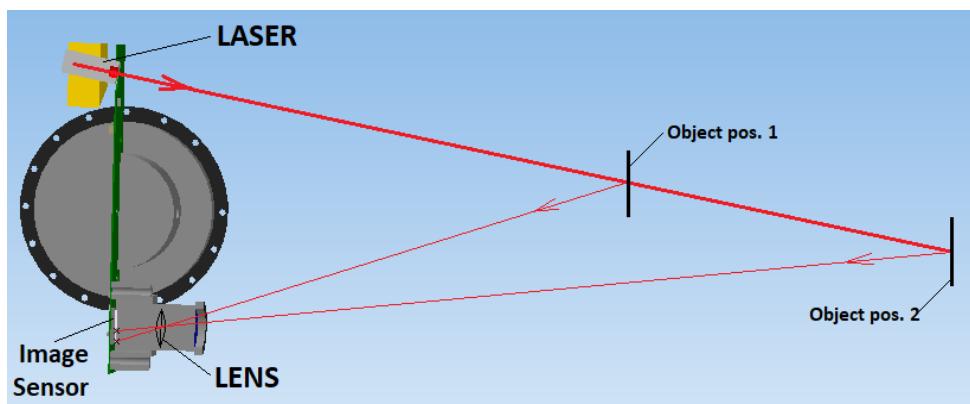


ILLUSTRATION 1 – Schéma représentant un instrument de mesure LIDAR. Source : tiré de réf URL02

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 1 ajoute des informations concernant le temps [GPS](#) durant lequel le point a été enregistré. Les autres formats ajoutent d'autres champs, 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 6. Toutes les informations concernant le classement sont contenues en un octet et en quatre sections. Elles sont expliquées dans le tableau 5.

Les cinq premiers bits correspondent à une table standard des classifications. Si le bit 5 est à un, le point a été créé par un logiciel de manière synthétique et si le bit 7 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 5 – Représentation des bits dans l'octet de classification. Source : adapté de LAS Specification Version 1.2 (2008, p. 8)

0.2. MAILLAGE

Les maillages sont une modélisation géométrique d'un domaine spatial continu par des éléments proportionnés finis. Les maillages simplifient donc un système par un modèle représentant ce système ou son environnement. Ils permettent notamment de réaliser des simulations ou bien d'avoir une représentation graphique numérique.

a. Fichier Stéréolithographique

Le format de [Stéréolithographie \(STL\)](#) est conçu par la société 3D System pour contenir des maillages. 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

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

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 axe vertical.

b. 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 v1x v1y v1z
    vertex v2x v2y v2z
    vertex v3x v3y v3z
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-signe-exposant", par exemple "6.248000e-003". La fin du fichier ASCII est définie par la séquence : "end solid".

c. 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 présents 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 que nous avons décidé d'implémenter dans notre application. Nous allons premièrement expliquer comment nettoyer les lidars à l'aide de l'algorithme de nettoyage par point pondéré et de nettoyage par point moyen, ensuite nous allons parler du fonctionnement de l'algorithme de triangulation de Delaunay et pour terminer nous vous expliquerons l'algorithme de fusion de maillage que l'on a implémenté.

1.1. NETTOYAGE

L'utilité de réduire la quantité de données sauvegardées de nos jours au vu des moyens de stockage modernes employés n'est plus une occupation majeure. Cependant, certains systèmes d'information restent limités en mémoire et en puissance de calcul notamment les smartphones ou encore les ordinateurs portables. Dans ces cas, le nettoyage des données **LIDAR** est aussi utile sur des systèmes plus puissants pour accélérer l'utilisation ultérieure des fichiers par la quantité réduite d'information. Les méthodes présentées ici ne sont pas exhaustives et peuvent être utilisées de manière indépendante ou bien combinées ensemble.

a. Lidar

Les données brutes provenant des instruments **LIDAR** contiennent souvent des informations bruitées et inutiles. Ces dernières peuvent fausser les résultats finaux et prennent de la mémoire de stockage supplémentaire.

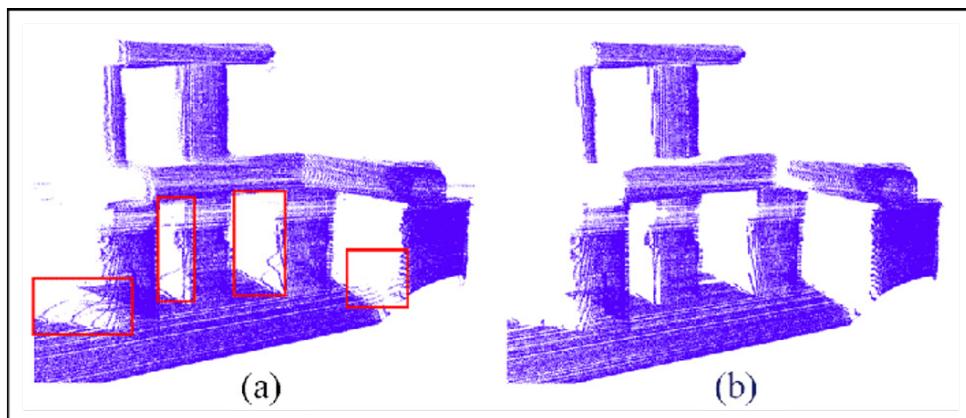


ILLUSTRATION 1.1 – Bruit sur des données lidar. Source : tiré de ref URL01

Les méthodes de nettoyage présentées ici se concentrent sur les impulsions à retour mul-

tiples vues au chapitre 1.

Une impulsion peut donc avoir plusieurs retours. Cela est dû au fait que la lumière frappe des objets superposés verticalement. Comme on peut le voir sur la figure 1.2

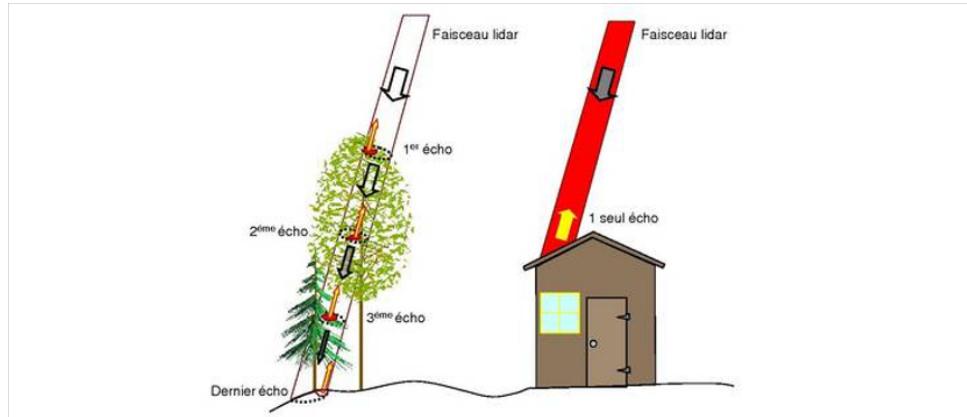


ILLUSTRATION 1.2 – Retours d'une impulsion de lidar. Source : tiré de ref URL04

a.1. Nettoyage par point moyen

Ce filtre se base sur la moyenne de position des points d'une impulsion. Ces impulsions à retour multiple peuvent être nombreuses en présence d'arbres. À la place d'avoir plusieurs échos, on les remplace par un seul point qui est la position moyenne des retours.

En termes mathématiques, 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.

a.2. Nettoyage par point pondéré

Une autre méthode similaire au point moyen consiste à refaire la moyenne des positions des voisins, où chaque élément de la moyenne est pondérée par l'intensité du retour.

a.3. Nettoyage par classe de points lidar

Une méthode différente pour nettoyer les données est de se baser sur la classification des points présentée au chapitre 1.1. La classification est donnée sur chaque point. En connaissant le type de points qui nous intéresse, il est possible de charger en mémoire uniquement le ou les types de points voulus.

b. Maillage

b.1. Nettoyage de maillage

Dans cette partie, on nettoie le bruit sur les grandes surfaces planes. Plus concrètement on lisse les surfaces planes en observant les changements d'élévation entre les sites afin de déterminer si ce changement est du bruit ou bien un réel changement de géométrie.

On procède de la manière suivante. Si la différence de hauteur entre la moyenne des voisins et le site actuel est supérieure à un ε on applique un changement de hauteur sur le site actuel. En d'autres termes :

$$P_z = \begin{cases} \bar{V}_z \text{ si } \bar{V}_z - P_z < \varepsilon \\ P_z \end{cases}$$

Où \bar{V}_z est l'élévation moyenne des voisins du site P

On peut observer le résultat de cette méthode dans les illustrations 1.3 et 1.4. On aperçoit que les points ont tendance à moins osciller dans l'axe vertical.

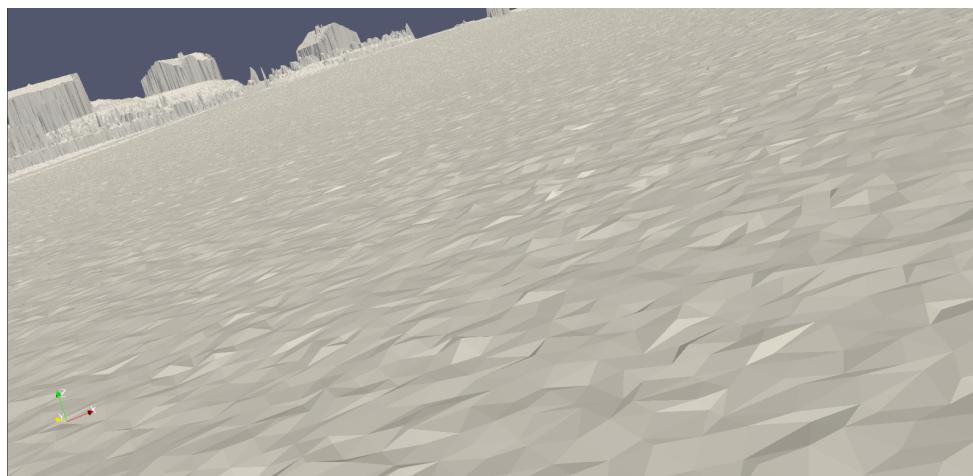


ILLUSTRATION 1.3 – Surface plane avant le nettoyage. Source : réalisé par Jérôme Chételat



ILLUSTRATION 1.4 – Surface plane après le nettoyage. Source : réalisé par Jérôme Chételat

1.2. TRIANGULATION DE DELAUNAY

Dans cette section, on détaille les algorithmes de reconstruction de surface. Il existe plusieurs manières de créer le même résultat, mais une méthode utilisée est la 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. On peut observer des résultats différents dans la figure 1.5.

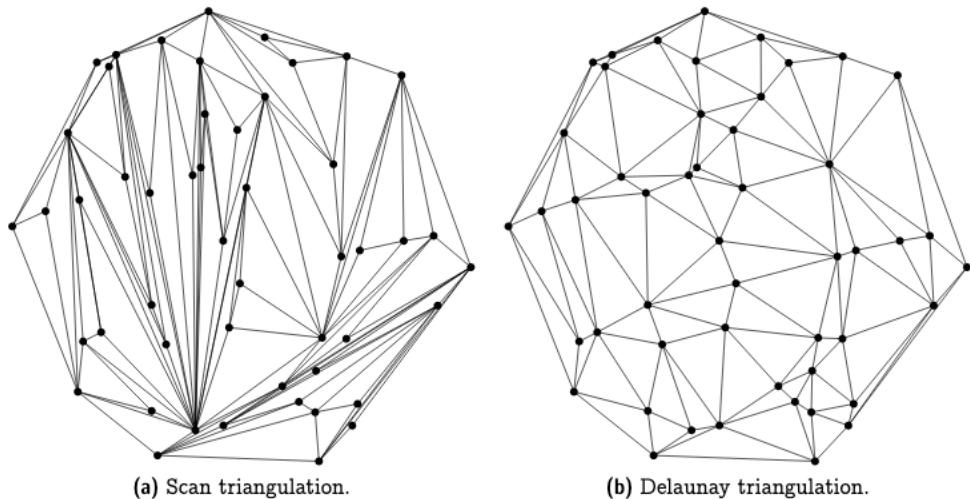


ILLUSTRATION 1.5 – Deux triangulations appliquées sur même jeu de données. Source : tiré de ref URL03

Une triangulation est de Delaunay si et seulement si la condition de Delaunay est vraie. Elle est la suivante : tous les cercles circonscrits des triangles du réseau sont *vides*.

Un triangle est vide si et seulement si aucun sommet autre que les siens réside strictement

dans le cercle circonscrit. Donc d'autres points peuvent être sur le périmètre du cercle circonscrit.

a. Bowyer-Watson

L'algorithme de Bowyer-Watson est conceptuellement simple. C'est un algorithme itératif découvert par Adrian Bowyer et David Watson. Le principe de ce dernier 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 de \mathcal{P} . On ajoute donc trois points, les sommets de S à \mathcal{P} .

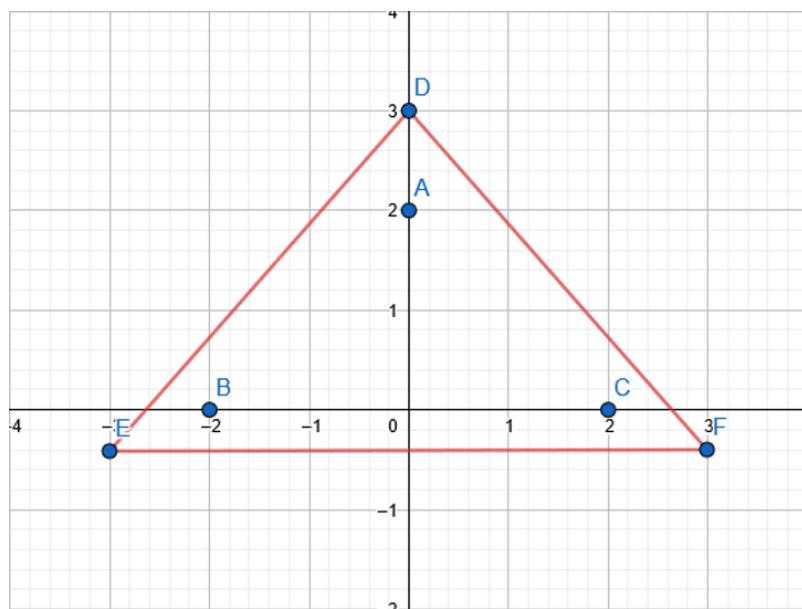


ILLUSTRATION 1.6 – Création du super-triangle englobant l'ensemble de points \mathcal{P} . Source : réalisé par Jérôme Chételat

On prend ensuite un point de \mathcal{P} puis on vérifie la condition de Delaunay.

La condition qui permet de déterminer si un point D est contenu dans le cercle circonscrit de $\triangle ABC$ est la suivante :

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

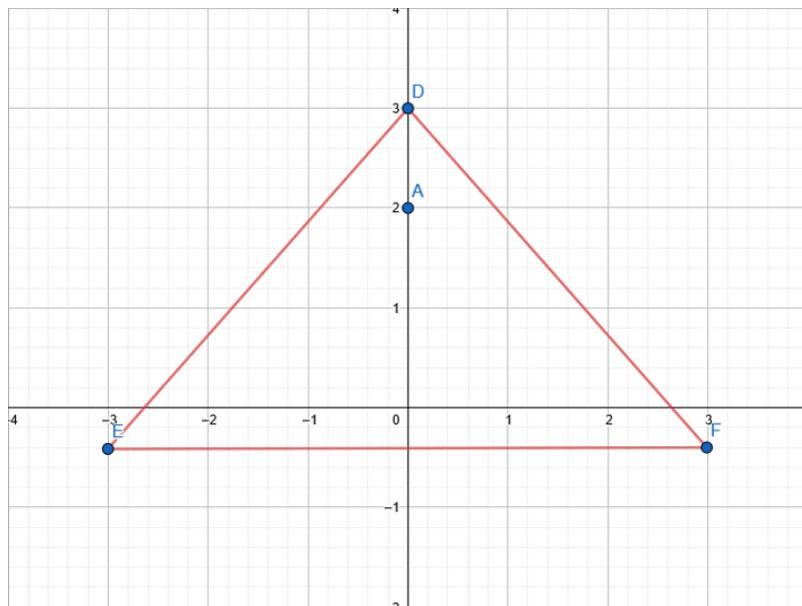


ILLUSTRATION 1.7 – Ajout du premier points de dans la triangulation. Source : réalisé par Jérôme Chételat

Si la condition est vraie, le point est relié au sommet du triangle qui le contient. Les arrêtes créées forment de nouveaux triangles appartenant désormais à la triangulation \mathcal{T} . Dans le cas de la première itération, on a le résultat intermédiaire dans la figure 1.8.

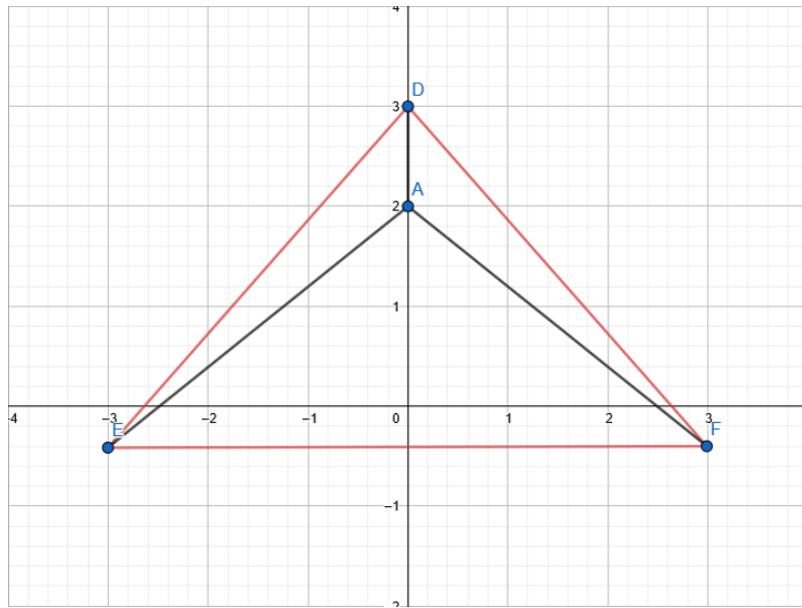


ILLUSTRATION 1.8 – Ajout des arrêtes reliant les sommets du super-triangle au point ajouté dans la triangulation. Source : réalisé par Jérôme Chételat

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

Pour finir, on identifie les arrêtes reliées aux sommets du super-triangle puis on les supprime de la triangulation \mathcal{T} . On supprime également les sommets du super-triangle ainsi que ses arrêtes de \mathcal{P} et de \mathcal{T} . Exemple de l'identification des arrêtes dans la figure 1.9

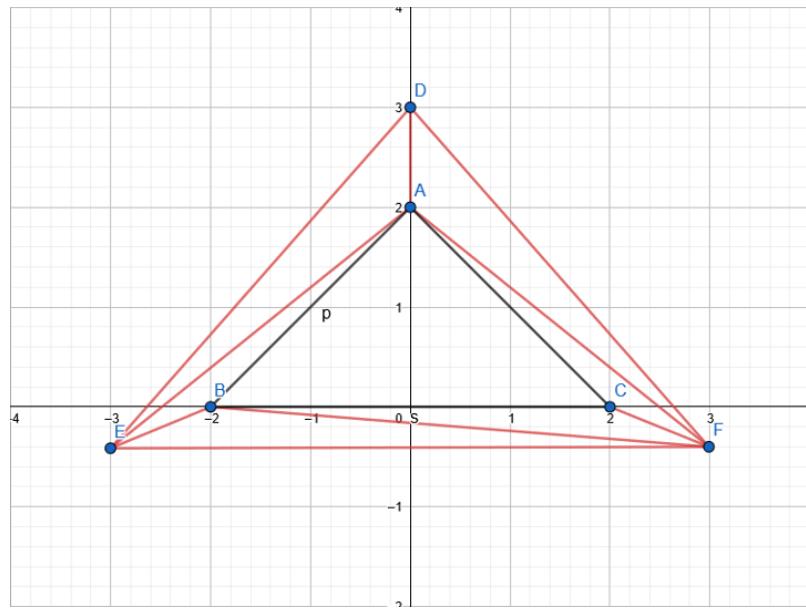


ILLUSTRATION 1.9 – Identification des arrêtes liées aux sommets du super-triangle en rouge .
Source : réalisé par Jérôme Chételat

Le résultat est un ensemble de triangles \mathcal{T} représentant la géométrie des surfaces. Cette dernière est appelée triangulation de Delaunay. Ce résultat est unique pour un ensemble de points \mathcal{P} donné. On peut observer le résultat de la triangulation dans la figure 1.10.

b. Lee & Schachter

L'algorithme de Lee et Schachter est basé sur le principe appelé "Diviser pour reigner". 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 triangulation de Delaunay.

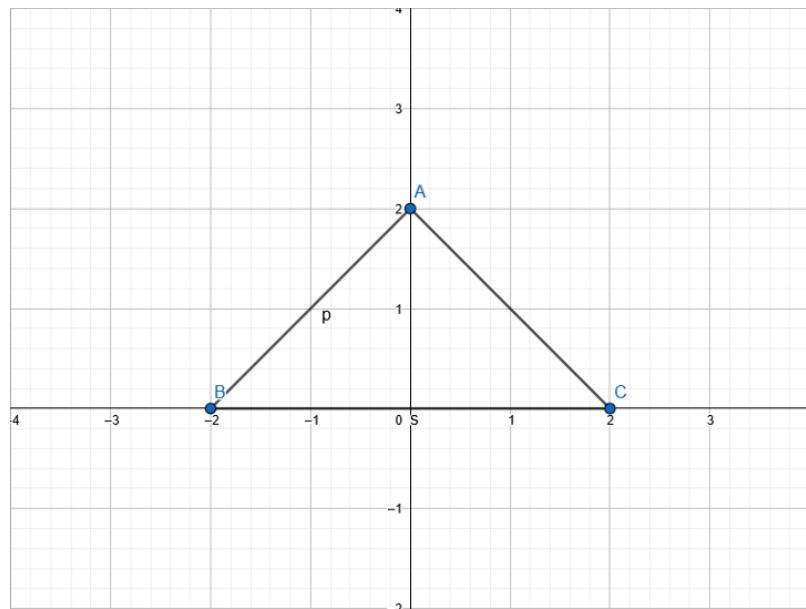


ILLUSTRATION 1.10 – Maillage résultant d'une triangulation de delaunay. Source : réalisé par Jérôme Chételat

1.3. 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 Schachter 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 arrête coupant la droite g . L'arrête créée est nommée "base LR". On a donc une situation similaire à l'illustration 1.11

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 trigonométrique 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

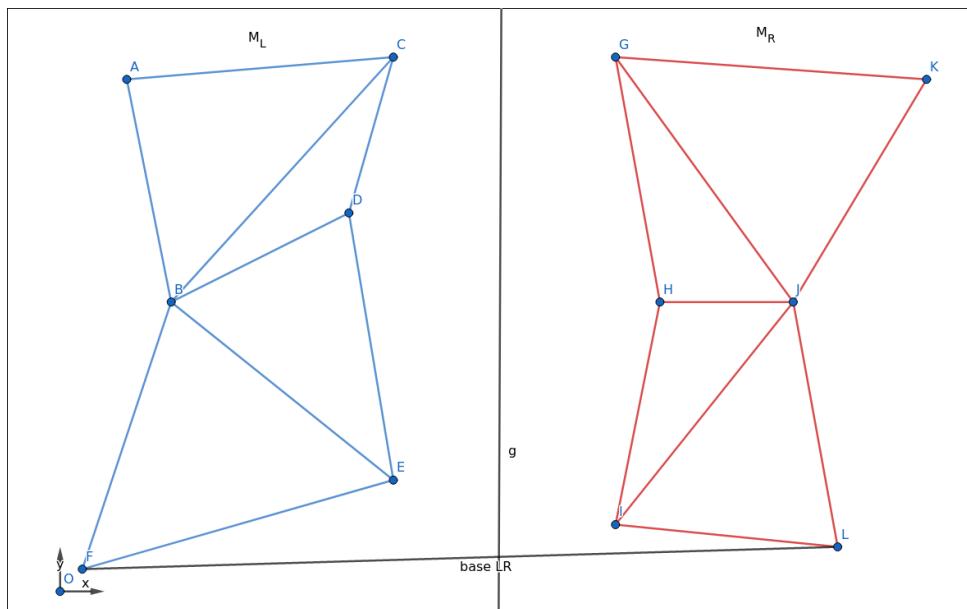


ILLUSTRATION 1.11 – Création de la base LR entre les deux maillages. Source : réalisé par Jérôme Chételat

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 n'est choisi ou que l'on détermine qu'aucun candidat ne peut être choisi.

Les illustrations 1.12 et 1.13 représentent le choix d'un candidat final pour chaque maillage. Cela ne représente que la première itération.

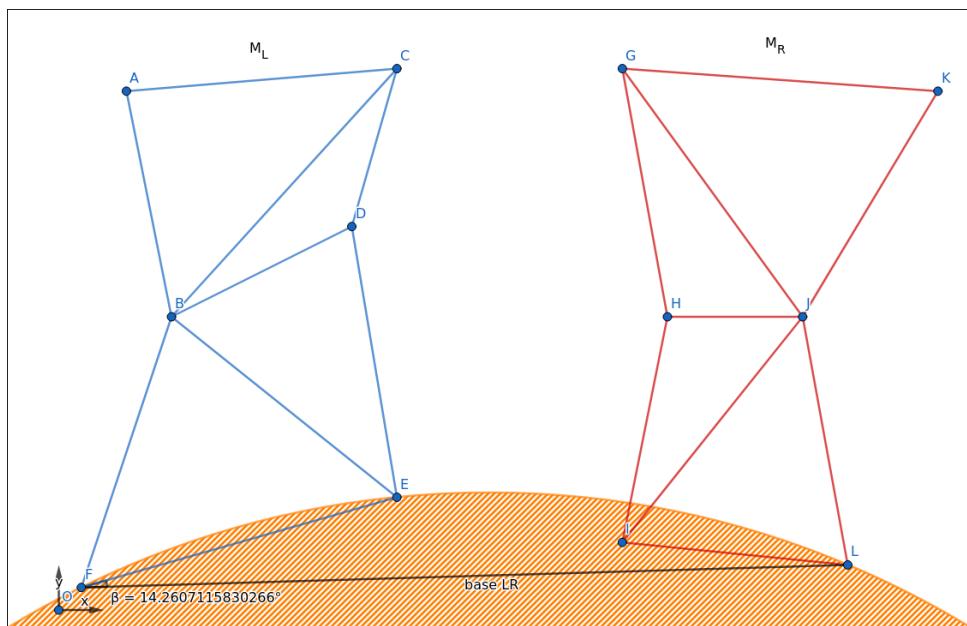


ILLUSTRATION 1.12 – Candidat final du maillage M_L . Source : réalisé par Jérôme Chételat

Une fois que les candidats pour chaque maillage ont été déterminés, si un seul candidat final

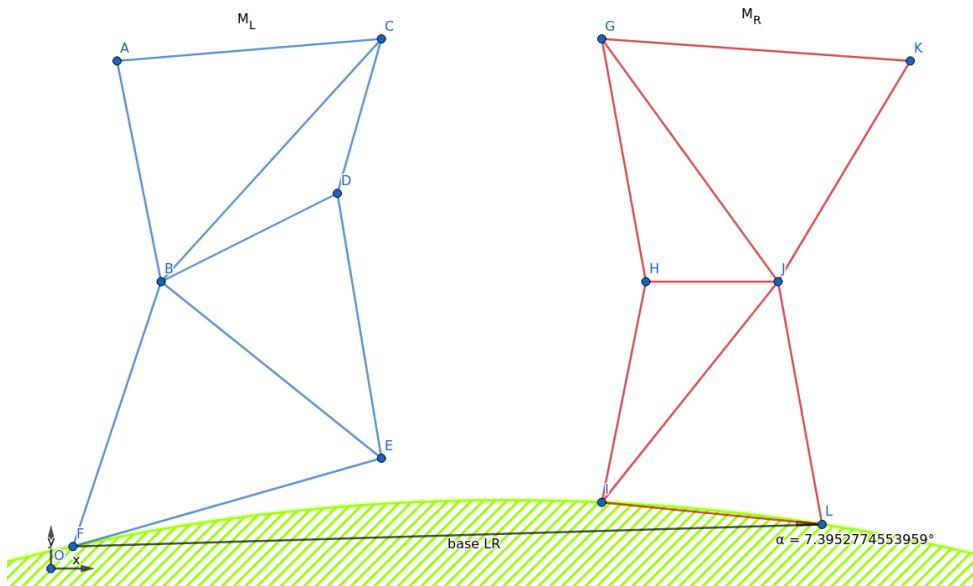


ILLUSTRATION 1.13 – Candidat final du maillage M_R . Source : réalisé par Jérôme Chételat

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 1.12 le site I est contenu dans le 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 arrête 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.

2.1. RUST

Rust est un langage de programmation système à usage général². C'est un langage moderne et multiparadigme. 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 plateformes Windows, MacOS et Linux.

a. Rust is the new JavaScript

Le choix du langage Rust n'est pas anodin. Un des buts de ce projet était d'expérimenter un stack web applicatif utilisant un seul langage et jusqu'à présent un langage régnait dans ce domaine. Il s'agit de JavaScript, car depuis la naissance des sites web dynamique, il n'avait pas de concurrent. Cependant depuis l'apparition du **WebAssembly (WASM)** dans les navigateurs, il n'est plus le seul. D'autres langages tels que le Rust compilent désormais en **WASM**. Cependant le support de cette plateforme, au moment de l'écriture de ce mémoire, reste encore expérimental.

2.2. ARCHITECTURE

L'application est composée de quatre modules principaux.

- Une bibliothèque regroupant des fonctions et des structures communes.
- Des utilitaires qui sont les outils en ligne de commande qui implémentent les pipelines
- Un serveur web qui 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 utilisée pour tous les

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

binaires. Il est composé de 4 étapes.



ILLUSTRATION 2.1 – Structure de fichier présent dans le module de la librairie de l’application.
Source : réalisé par Jérôme Chételat

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

2.3. 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 implémentent les structures pour écrire les fichiers **STL** et *las-rs* qui exposent les structures et méthodes liées aux données lidar.

```
.
├── Cargo.toml
└── src
    ├── exports
    │   └── stl.rs
    ├── exports.rs
    ├── filters
    │   ├── lidar
    │   │   ├── average_point.rs
    │   │   ├── intensity.rs
    │   │   └── ponderated.rs
    │   └── lidar.rs
    ├── mesh
    │   ├── average_smoothing.rs
    │   └── subsample.rs
    └── mesh.rs
    └── prelude.rs
    ├── filters.rs
    ├── las_delaunator.rs
    ├── lib.rs
    ├── triangulation
    │   ├── itriangle.rs
    │   └── mesh.rs
    └── xyz.rs
    └── triangulation.rs
    └── utils.rs

```

6 directories, 19 files

ILLUSTRATION 2.2 – 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.

b.1. Triangulation de Bowyer-Watson

L'implémentation a été faite lors du projet de semestre. Actuellement l'algorithme a une complexité quadratique ($\mathcal{O}(n^2)$) à cause de la recherche d'appartenance d'un point à un triangle qui itère à travers tous les triangles de la triangulation. Ceci est fait jusqu'à trouver le triangle contenant le point.

Dans ce travail, l'optimisation de cet algorithme n'était pas demandée. Cependant la complexité pourrait théoriquement être réduite à $\mathcal{O}(n \log(n))$ en utilisant une structure de données telle qu'un arbre ou un dictionnaire qui enregistrerait la position des centres des cercles-circonscrits de chaque triangle créé et parcouru.

b.2. Fusion de triangulation

Pour 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<u32>,
6      /// Indexed vertices part of the convex hull of the mesh
7      pub hull: Option<Vec<u32>>,
8      /// The neighbourhood of each indexed point
9      pub neighbourhood: Option<Vec<HashSet<u32>>>,
10 }

```

Une particularité de cette structure est que l'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 le calculer.

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

Le set permet de prévenir les duplications de même voisin pour un site. Ce set est ensuite indexé selon le site actuel.

Une fois tout les triangles parcouru, on assigne au maillage le voisinage et l'on renvoie une copie.

Cette dernière peut cependant être améliorée, car actuellement, une copie du voisinage est renvoyée à l'appelant. Ce qui à pour effet de créer des données dupliquées dans la mémoire. Le choix de la copie a été fait, car l'utilisation du voisinage dans un algorithme de fusion nécessite de muter certaines parties. Une solution pour éviter ces duplications serait de créer une structure

indiquant les parties qui ont muté 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 bibliothèque pour en calculer. Il est possible de le calculer au travers de la crate "ncollide" qui expose une version de quickhull, un algorithme de calcule 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 paramètres un point indexé par un entier non signé dans le maillage passé en argument et une structure de points pour effectuer la recherche puis 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 langages de programmation fonctionnels pour vérifier les conditions énumérées dans le chapitre 2.3. Si un candidat final est choisi, il est ajouté à une liste d'arêtes **edges** et à la fin de l'opération, cette liste est utilisée afin de construire les nouveaux triangles entre les deux maillages.

2.4. 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 se sert d'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 :

- las-delaunator : Triangulation par l'algorithme quickhull
- las-geometry : Triangulation par l'algorithme de Bowyer-Watson
- las-filtering : Nettoyage de points lidar
- las-sample : Prendre les n premiers points d'un fichier lidar
- las-map-geometry : Triangulation d'un ou plusieurs fichiers simultanément
- mesh-merge : Fusion de maillage
- mesh-smoothing : Nettoyage de maillage
- mesh-subsample : Prendre qu'une partie des triangles du maillage
- to-txt : Transforme un fichier lidar en simples points ASCII dans le format $x\ y\ z$

a. mesh-smoothing

Le binaire ici produit permet de nettoyer le bruit qui est généré par les points lidars après une triangulation. Il agit sur les triangles d'un maillage.

La commande prend en argument un fichier d'entrée et un fichier de sortie. Il prend également en option un paramètre ϵ qui sert de critère pour appliquer la moyenne d'un point par rapport à ses voisins.

Pour réaliser le nettoyage, on itère à travers la liste des sommets du maillage. Pour chaque sommet, on vérifie si la différence entre la hauteur moyenne des voisins et le sommet actuel est inférieure au ϵ . Si la condition est vraie, dans ce cas on assigne la hauteur moyenne des voisins au sommet actuel. Le cas échéant, le sommet est simplement ignoré.

L'itération se fait sur une copie des sommets du maillage pour éviter d'avoir des incohérences de hauteurs avec les points voisins dans le cas où un sommet voisin d'un autre est modifié avant ce dernier.

b. mesh-subsample

La commande permet de décimer un maillage. Elle prend en argument un fichier STL d'entrée et un fichier STL de sortie. Elle prend encore en paramètres optionnels un type de filtre à appliquer.

Le paramètre du choix de la méthode de décimation est un nombre.

1. Décimation par rapport à la distance
2. Décimation aléatoire

Par défaut, si aucune méthode n'est choisie, on applique la décimation selon la distance.

On crée un itérateur sur les sommets présents dans le fichier STL. Ensuite on calcule la distance euclidienne par rapport au voisin de chaque sommet. Si elle se trouve être inférieure à la valeur de ε , on supprime le sommet. Après avoir déterminé les sommets qui sont à garder, on applique à nouveau une triangulation quickhull de la crate *delaunator-rs*. Enfin on retourne le nouveau maillage contenant moins de sommets. Une autre manière de choisir quels points seront supprimés est de le faire aléatoirement.

c. mesh-merge

Cette commande permet à un utilisateur de créer une triangulation à partir de la fusion de deux triangulations existantes. Cet outil est particulièrement utile lors de lorsque l'on a déjà une partie des points lidar sous forme de maillage.

2.5. STACK WEB

Cette section présente la partie concernant le web. On y détaille l'implémentation de deux modules : le client et le serveur. Ces derniers forment le *stack web*.

a. Serveur

Le serveur consiste en un binaire Rust utilisant le framework actix-web. Le choix de l'utilisation de ce framework a été fait pour déléguer la gestion des requêtes reçues de bas niveau ainsi que l'utilisation simple de fonctions asynchrones pour gérer ces dernières. Il a aussi été choisi, car il est le seul pour l'instant à utiliser que des fonctionnalités stables du langage de programmation.

Le serveur est une **Application Programming Interface (API)** REST comprenant une route pour chaque fichier d'un dossier. Elle expose au web, les fichiers appartenant au dossier nommé *regions*. Ce dernier pouvant être donné en argument au serveur, à son lancement.

Concernant les mécanismes de **Cross-Origin Resource Source (CORS)**, on utilise une crate de actix s'appelant *actix_cors*. Il s'agit d'un middleware actix s'ajoutant sur une ou plusieurs routes pour activer les mécanismes **CORS**.

La lecture et le découpage en paquet d'un fichier sont réalisés par un autre module de actix.

b. Client

Le client web a été écrit en Rust et compilé en **WASM**. Il permet une visualisation de maillage au travers d'un navigateur web grâce à la bibliothèque graphique WebGL.

Cette partie utilise grandement une crate appelée "web-sys" qui expose des bindings pour le JavaScript. On utilise également la crate "nalgebra" qui propose des fonctions pour manipuler des matrices 4x4 utilisées dans WebGL.

Lorsqu'on écrit du code Rust ciblant les plateformes web, plusieurs contraintes doivent être respectées. La première est que **WASM** étant un format sur 32 bits, la mémoire adressable par le programme n'est que de quatre gigaoctets. Une seconde contrainte est que les threads n'existent pas pour une application web et sont remplacés par les WebWorkers³.

Lors du chargement de la page web, la première étape est de récupérer le fichier qui doit être affiché. Ceci est fait à partir de l'**API fetch** des navigateurs. On forge un objet requête en donnant le lien de téléchargement du fichier. Une fois téléchargé, on transforme la réponse en *blob*. Ce dernier est ensuite casté en tableau Rust à travers la crate *JsCast*. La lecture du fichier **STL** en binaire est déléguée à la crate *stl-io*.

On récupère les vertex du maillage ainsi que les indices des sommets qui forment les triangles. On récupère aussi les normales des triangles. Ils nous permettront de faire le calcul des ombrages.

On crée ensuite un vertex buffer object pour chaque objet récupéré du maillage, puis l'on transfère les données dans les buffers respectifs. Cette dernière opération en Rust est dans un bloc *unsafe*, car on déréférence des pointeurs dans la mémoire du navigateur. Enfin on crée les matrices de projection, de vue et de transformation puis on les transfère également dans des

3. Technologie des navigateurs web permettant l'exécution de code dans un processus en arrière-plan de manière indépendante

zones mémoires graphiques. Enfin on ordonne de faire un rendu à l'écran.

CHAPITRE 4 : RÉSULTATS

Dans ce chapitre, nous allons voir les résultats obtenus par les différentes parties de l'application. On va aussi comparer les différentes métriques collectées des binaires.

Les tests suivants ont été réalisés principalement sur la configuration d'ordinateur se trouvant dans la figure 4.1 en annexe 4.

3.1. TRIANGULATION

La triangulation de Delaunay provient en grande partie du projet de semestre. Seule la partie fusion de maillage a été implémentée dans ce travail.

Pour la triangulation, on utilise deux fichiers :

- Le fichier 2501500_1112000.las du jeu de données des SITG contenant 9'000'987 points
- Un fichier contenant que les 100'000 premiers points du fichier précédent.

a. Algorithme de Bowyer-Watson

	Nb Points	Temps moyen [s]	Nb Trig	Taille de sortie [Mb]
2501500_1112000_sample.las	100'000	6,646	199928	9.6 Mb
2501500_1112000.las	9'000'987	N/A	N/A	N/A

TABLEAU 3.1 – Résultats de la triangulation avec l'algorithme Bowyer-Watson

Dans le tableau 3.1, on peut observer que sur un fichier de plusieurs millions de points aucune valeur n'est présente. L'implémentation actuelle n'arrivant pas à exécuter en un temps raisonnable, les valeurs ici manquent.

b. Algorithme Delaunator

Cette partie utilise une implémentation de la triangulation de Delaunay existante. Il s'agit de l'algorithme delaunator implémentant des idées de S-hull et de sweepline triangulation.

Les résultats suivants montrent l'efficacité de cette implémentation par rapport à l'implémentation de Bowyer-Watson.

	Nb Points	Temps moyen [s]	Nb Trig	Taille de sortie [Mb]
2501500_1112000_sample.las	100'000	0,0513	199'928	9.6
2501500_1112000.las	9'000'987	9,548	18'001'310	859

TABLEAU 3.2 – Résultats de la triangulation avec l’algorithme delaunator

c. Fusion de triangulation

Les résultats de la fusion ne sont pas disponibles, car le binaire actuel ne soumet pas de fichiers de sortie.

On peut observer le flamegraph de la figure 3.1 du binaire ou le programme reste bloqué dans la méthode *get_side_candidate* avant que ce dernier soit interrompu.

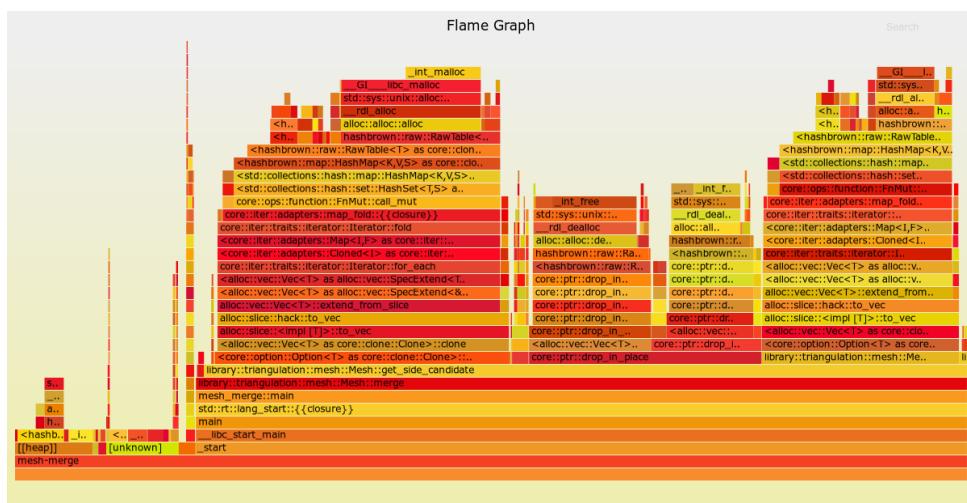


ILLUSTRATION 3.1 – Flamegraph du binaire mesh-merge. Source : réalisé par Jérôme Chételat

Cependant les tests unitaires valident l’algorithme pour un jeu de données simple.

Améliorations possibles

L’implémentation actuelle gère uniquement les maillages linéairement séparables. L’idée serait de pouvoir aussi gérer le cas où les maillages peuvent se superposer.

On a aussi le problème que l’algorithme ne s’applique que dans une direction pour créer des arrêtés. Le principe serait d’implémenter un choix de direction pour appliquer la fusion.

3.2. NETTOYAGE

Le nettoyage de points est effectué sur deux fichiers lidars :

1. 2488500_1111500.las

2. 2499000_1118000.las

Pour les maillages on effectue les tests sur deux fichiers **STL**, résultant des triangulations des fichiers précédent.

a. Lidar

Voici les résultats obtenus :

Point moyen

	Nb Points	Temps	Points supprimés	Taille originale	Taille de sortie
2488500_1111500.las	8'570'137	4.67 s	1'147'625	229 Mo	199 Mo
2499000_1118000.las	9'336'496	4.96 s	1'000'136	250 Mo	223 Mo

TABLEAU 3.3 – Résultats du nettoyage par point moyen

Point pondéré

	Nb Points	Temps	Points supprimés	Taille originale	Taille de sortie
2488500_1111500.las	8'570'137	4.87 s	1'147'625	229 Mo	199 Mo
2499000_1118000.las	9'336'496	5.03 s	1'000'136	250 Mo	223 Mo

TABLEAU 3.4 – Résultats du nettoyage par point pondéré

On observe aucune différence de taille de sortie ni de nombre de points supprimés entre les deux méthodes. Cela s'explique par le choix de la méthode de nettoyage. Ici le nombre d'impulsions à retour multiple reste constant et donc le nombre de points à supprimer ne varie pas.

Cependant, on observe des différences de position des points lidars dans les figures de l'annexe 5.

b. Lissage de maillage

Le lissage a été effectué avec un epsilon de 2.0. Voici les résultats obtenus sur un maillage comportant une route :

On observe dans les images précédentes que le lissage rend les surfaces planes et que les pics ne perdent pas d'élévation au-dessus d'un certain ε .

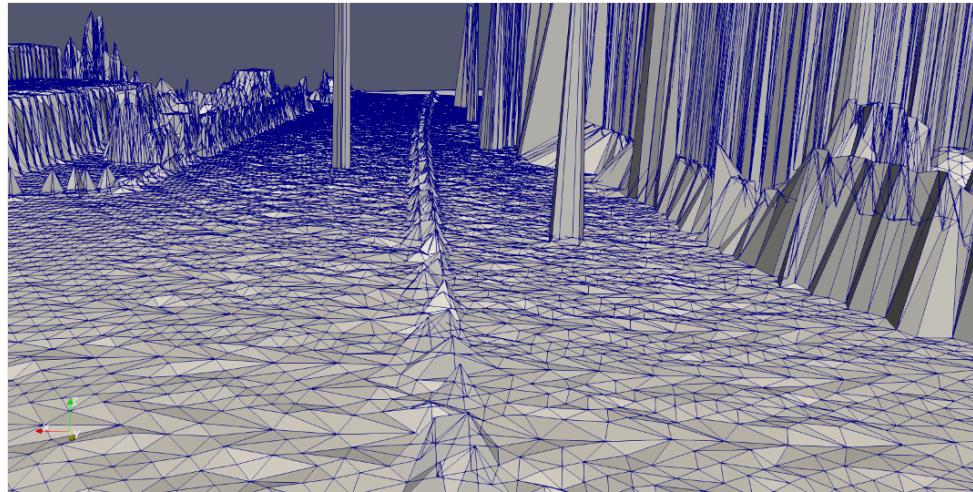


ILLUSTRATION 3.2 – Maillage bruité avec une route avant lissage. Source : réalisé par Jérôme Chételat

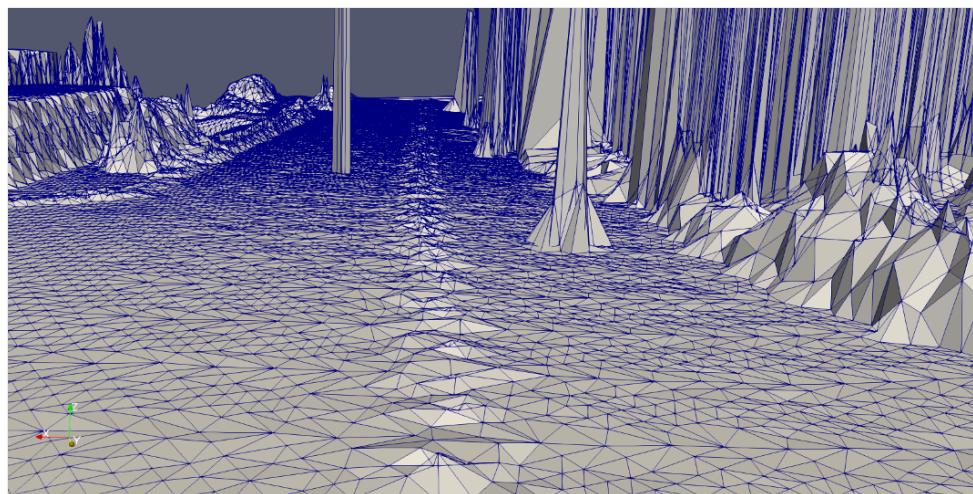


ILLUSTRATION 3.3 – Maillage avec une route après lissage. Source : réalisé par Jérôme Chételat

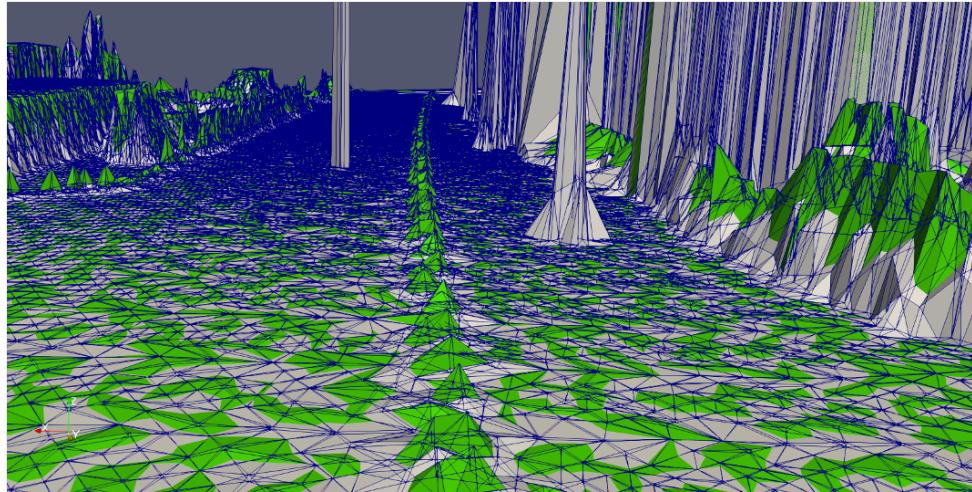


ILLUSTRATION 3.4 – Différence d’élévation des maillages après lissage. Vert : maillage de base. Source : réalisé par Jérôme Chételat

c. Décimation de maillage

3.3. STACK WEB

a. Serveur web

Le serveur actuel permet le téléchargement de fichiers STL présents dans un dossier spécifique. Il arrive à servir des fichiers avec les mécaniques CORS.

b. Client web

Les tests suivants ont été effectués sur le navigateur Firefox version 80.0.1. 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.

On peut observer dans la figure 4.5 en annexe 6 qu'une grande partie de la page est blanche. Ce phénomène est normal, car le rendu se fait sans ombres. De ce fait, cela rend l'observation des changements d'élévation des surfaces difficile. Cependant, on peut voir que le contour de la partie blanche laisse imaginer qu'il y a bien un rendu de maillage présent.

On y observe aussi que dans un tableau de *Float32* se trouve 300'000 valeurs qui correspondent aux 100'000 sommets du maillage (une valeur par composante de la position d'un point).

Améliorations possibles

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 (dépend de la puissance de l'hôte) . Il serait possible dans un futur proche de faire ces opérations dans des WebWorkers. Au moment de l'écriture de ce mémoire, l'utilisation des WebWorkers en Rust est encore expérimentale.

Une autre amélioration à faire serait de réaliser le calcul des ombres afin d'afficher les reliefs du maillage. Ou encore de donner des contrôles à l'utilisateur pour changer le point de vue du maillage (angle et position de la caméra dans la scène).

CONCLUSION

TODO 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 afficher certaines données. Les outils devaient pouvoir nettoyer des données **LIDAR** et les trianguler de manière efficace, de filtrer et fusionner des maillages. Pour cela, on a dû en premier explorer les formats de fichier **LIDAR** et **STL**. Ensuite nous avons explorer les algorithmes qui allait être implémentées dans l'application comme l'algorithme de Bowyer-Watson qui permet de faire une triangulation de Delaunay et l'algorithme de Lee & Schachter pour la fusion de triangulation. Nous avons aussi vu les différentes méthodes de nettoyage de données pour réduire la taille des fichiers utilisés et éviter de fausser les résultat dans leur utilisation ultérieur. Par la suite nous avons implémenté ces méthodes dans des outils en ligne de commande qui ont été implémentées dans le langage Rust. Ce dernier permet de compiler pour des plateformes inhabituelles pour un langage système tel que le web et grâce à cela, nous avons réaliser un serveur de fichier et un client web pour afficher des maillages. Nous avons ensuite testé nos outils et comparé les résultats obtenus et vu que les méthodes de nettoyage donnait des résultats numériques similaire mais visuels différents. L'algorithme de Bowyer-Watson donnait un résultat pour un nombre de points limité mais lorsqu'on passait à un jeu de donnée complet, aucun résultat n'a été trouvé au vu du temps de calcul trop important. On avait vu que le serveur de fichier servait des fichiers au client web qui les affiche à l'aide de WebGL. Certains outils comme le client web sont encore inachevés et nécessitent des corrections pour fonctionner correctement. La réalisation de ce travail m'a permis de mettre en oeuvre l'enseignement reçu lors de ma formation ainsi que d'enrichir mes connaissances dans le domaine de la topographie numérique. J'ai pu aussi utiliser le langage Rust que je compte désormais utiliser de manière régulière pour mes projets personnels, car l'écosystème, malgré son manque de maturité, rend le développement de logiciels facile et amusant. Ce travail était aussi un challenge, non seulement à cause du covid19, mais aussi de par l'ampleur du projet qui dépasse tout ce que j'ai pu rencontrer auparavant.

Le travail étant inachevé, un futur travail pourrait reprendre le projet et terminer ce qui n'est pas encore complet, mais aussi rajouter de nouvelles fonctionnalités comme la gestion de nouveaux formats de données de maillages ou encore l'optimisation de certains algorithmes implémentés afin de pouvoir véritablement gagner en performance.

ANNEXES

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 : IMPLÉMENTATION DE LA RECHERCHE DE SITE

CANDIDAT POUR LA FUSION

```

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
30         // If both are contained in each other
31         // it is a degenerate case and there
32         // is no possible outcome
33         // to have a good Delaunay triangulation
34         (false, false) =>
35             return Err(
36                 "Edge_case_found_about_4_co-circular_points"
37             ),
38
39         (true, false) => {
40             edges.push(l);
41             edges.push(r_bs);
42             l_bs = l;
43         }
44
45         (false, true) => {
46             edges.push(l_bs);
47             edges.push(r);
48             r_bs = r;
49         }
50
51         _ => break,
52     }
53
54     (Some(l), None) => {
55
56         // Connect it to the lr base,
57         // push the new triangle and move the l_bs reference
58         edges.push(l);
59         edges.push(r_bs);
60         l_bs = l;
61     }
62
63     (None, Some(r)) => {
64
65         // 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 : IMPLÉMENTATION DU VOISINAGE D'UN MAILLAGE

```

1  pub fn neighbourhood(&mut self) -> Option<Vec<HashSet<usize>>> {
2
3      if self.neighbourhood.is_some() {
4
5          return self.neighbourhood.clone();
6
7
8      // Iterating over all triangles in the mesh
9
10     self.faces
11
12         .iter()
13
14         .enumerate()
15
16         .step_by(3)
17
18         .for_each(|(i, _): (usize, &usize)| {
19
20             // Adding the neibours of the first vertex
21
22             dict[self.faces[i]].insert(self.faces[i + 1]);
23
24             dict[self.faces[i]].insert(self.faces[i + 2]);
25
26
27             // Adding the neibours of the second vertex
28
29             dict[self.faces[i + 1]].insert(self.faces[i]);
30
31             dict[self.faces[i + 1]].insert(self.faces[i + 2]);
32
33             // Adding the neibours of the third vertex
34
35             dict[self.faces[i + 2]].insert(self.faces[i]);
36
37             dict[self.faces[i + 2]].insert(self.faces[i + 1]);
38
39         });
40
41
42     self.neighbourhood = Some(dict.clone());
43
44     Some(dict)
45
46 }

```

ANNEXE 4 : CONFIGURATION DE L'ORDINATEUR SUR LEQUEL LES TESTS ONT ÉTÉ EFFECTUÉS

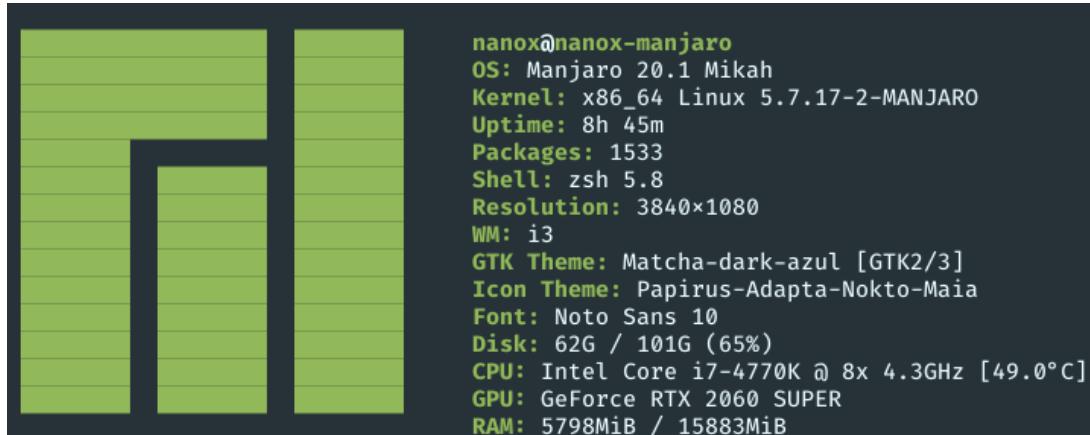


ILLUSTRATION 4.1 – Configuration de l'ordinateur de tests. Source : réalisé par Jérôme Chételat

ANNEXE 5 : FICHIERS LIDARS AVANT ET APRÈS NETTOYAGE

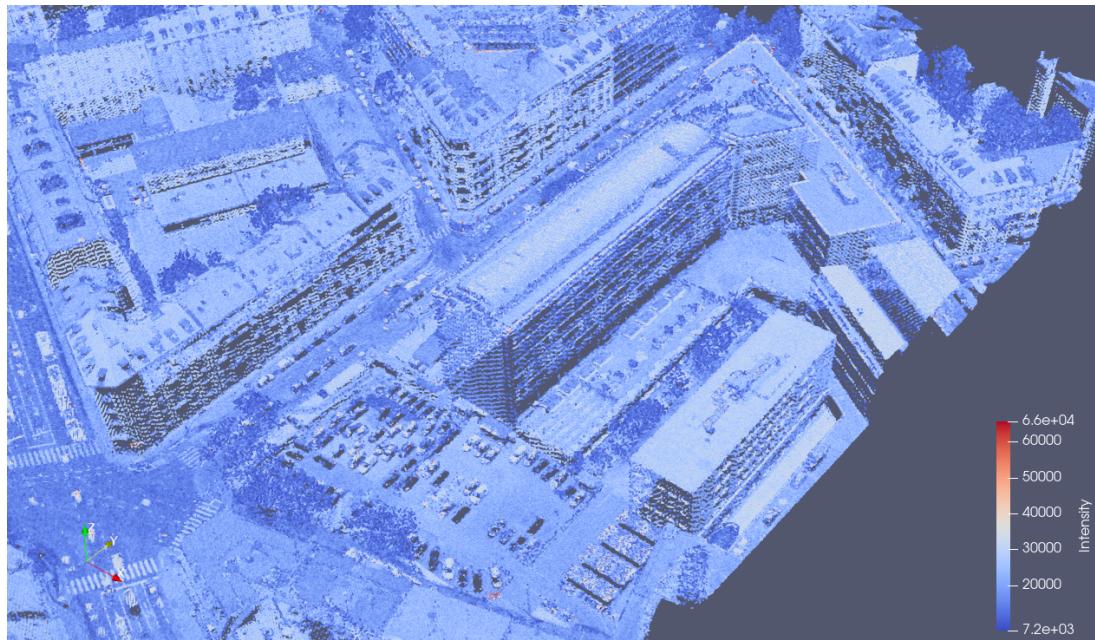


ILLUSTRATION 4.2 – Fichier lidars brute avec les batiments de l'HEPIA . Source : réalisé par Jérôme Chételat

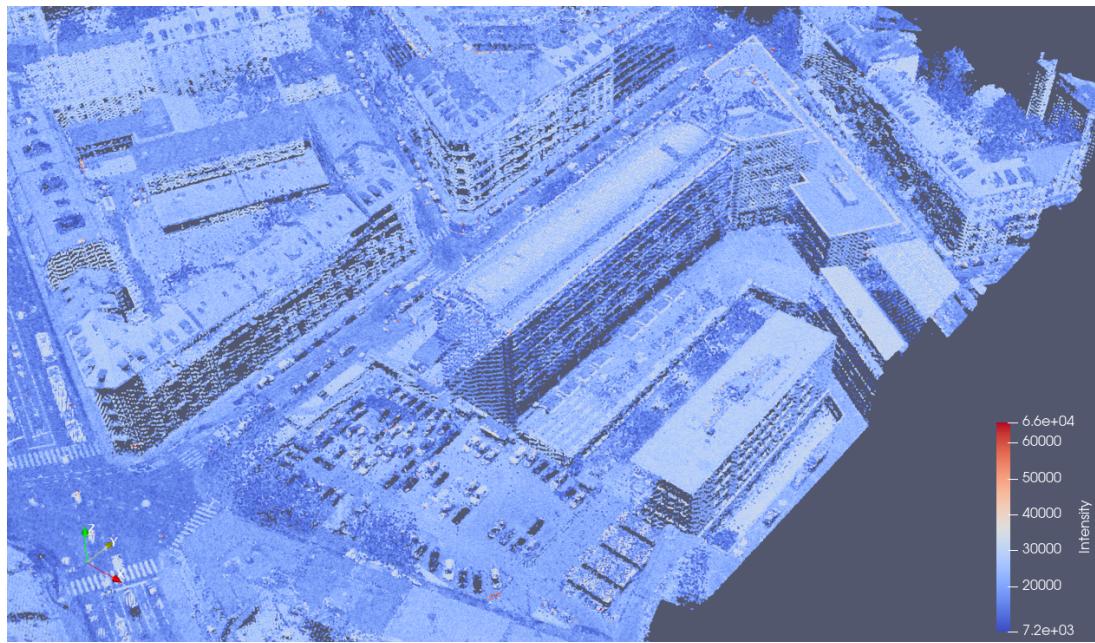


ILLUSTRATION 4.3 – Fichier lidars avec HEPIA après nettoyage par point moyen . Source : réalisé par Jérôme Chételat

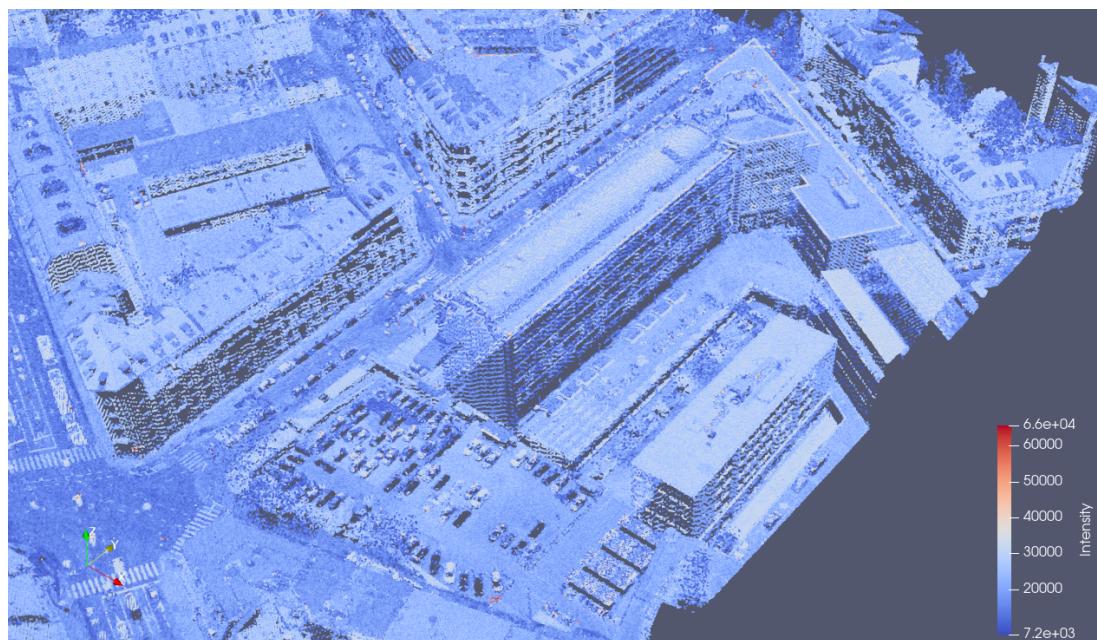


ILLUSTRATION 4.4 – Fichier lidars avec HEPIA après nettoyage par point pondéré . Source : réalisé par Jérôme Chételat

ANNEXE 6 : RÉSULTAT D'UN RENDU DE MAILLAGE DANS UN NAVIGATEUR AVEC LE CLIENT WEB

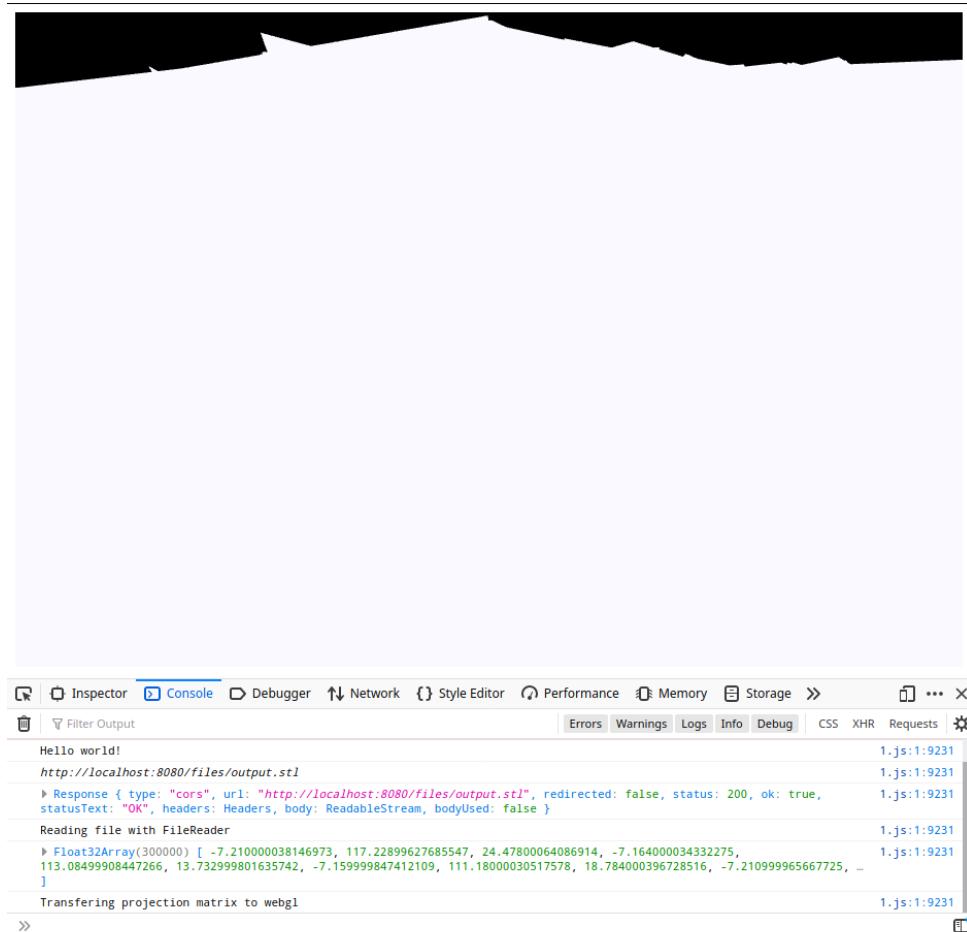


ILLUSTRATION 4.5 – Résultat du rendu d'un maillage par le client web. Source : réalisé par Jérôme Chételat

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.
- DESNOGUES, 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](https://doi.org/10.1007/s10109-005-0007-4).