P1RV : ALGORITHME DE REDUCTION DE MAILLAGE

REALISE PAR :

PAUL BERTRAND

SALWA ABERKANE

ENCADRE PAR :

M. ETIENNE PEILLARD

Table des matières

TABLE DES FIGURES

INTRODUCTION

Pour l'affichage d'images en temps réel, il est nécessaire d'avoir en permanence plusieurs versions des objets 3D affichés qui correspondent à plusieurs niveaux de détail.

À partir d'un objet 3D initial, il est possible d'obtenir une version allégée de cet objet (avec un niveau de détail plus faible) en pratiquant la réduction de maillage. Celle-ci représente donc un enjeu majeur pour l'affichage en temps réel. Cependant, c'est un processus manuel souvent fastidieux. Il est donc synonyme de perte de temps et donc d'argent pour les entreprises.

L'objectif de notre projet a donc été de trouver un moyen d'automatiser le processus de réduction de maillage.

Dans ce rapport, nous exposerons d'abord le projet et ses objectifs, puis nous présenterons l'application que nous avons développée pour répondre aux problématiques du sujet. Ensuite, nous nous pencherons sur les choix que nous avons réalisés pour le développement et comment nous avons implémenté l'algorithme de réduction de maillage. Enfin, nous mettrons en avant les difficultés rencontrées et les pistes d'amélioration avant de conclure.

PRESENTATION DE PROJET

Le but du projet est d'implémenter un algorithme de réduction de polygones qui permet de créer une version plus légère d'un modèle 3D.

L'objectif est de proposer une interface qui permet de guider cette décimation.

L'objectif final est que cette interface puisse permettre de personnaliser la décimation afin que le processus de réduction de maillage ne soit pas entièrement automatisé, mais que l'utilisateur ait la possibilité de choisir quelle partie de l'objet il veut réduire.

SOLUTION PROPOSEE

L'application que nous avons réalisée est développée en C++ sur Linux et présente les fonctionnalités suivantes :

- Extraction de données depuis un fichier *stl* binaire.
- Affichage des informations relatives au fichier chargé :
    - Nombre de nœuds, triangles, et en-tête du fichier *stl* chargé
    - Liste des nœuds
    - Liste des triangles
    - Liste des normales
- Affichage de l'objet avec OpenGL
- Réduction pas à pas de l'objet :
    - Suppression d'un nœud
    - Suppression de 5% à 45% des nœuds tant que les critères de suppression que nous avons fixés sont valides
    - Suppression de tous les nœuds qui vérifient les critères de suppression

Il est notable que l'application que nous proposons ne répond pas entièrement au Cahier des Charges exposé dans la présentation du projet car l'utilisateur, lorsqu'il réalise une réduction, ne peut pas choisir quel partie de l'objet sera réduite. Nous avons malheureusement manqué de temps pour mettre en œuvre cette fonctionnalité.

UTILISATION DE L'APPLICATION

A. LANCEMENT DE L'APPLICATION

Celle-ci se lance à partir du terminal en lui donnant en argument le fichier *stl* à traiter :

```
./MeshReductor moto.stl
```

Si aucun argument n'est donné, c'est le fichier *Box1x1x1.stl* qui est chargé. Le fichier stl donné en argument doit être un ***stl binaire.*** L'extraction des données de ce fichier est ensuite effectuée. Celle-ci peut prendre un certain temps. Son évolution est affichée en pourcentage (Figure 1).

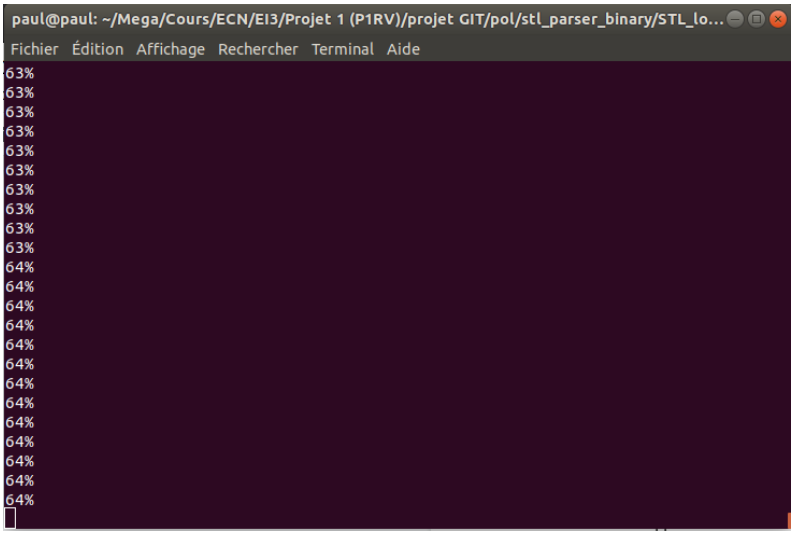


Figure 1: l'évolution de l'extraction en pourcentage

Si le fichier n'est pas un fichier *stl* binaire, un message d'erreur s'affiche (Figure 2) et le fichier par défaut est chargé. Il en est de même si trop d'arguments sont fournis.

```
paul@paul:~/EI3/STL_loader$ ./MeshReductor porte\ clef\ normal.stl
ERROR: File is not a binary stl but ASCII. Please convert it.
```

Figure 2: message d'erreur

Une fois l'application lancée, un menu s'affiche dans le terminal. Ce menu permet à l'utilisateur de réaliser plusieurs actions détaillées dans la partie suivante.

B. ACTIONS DE L'UTILISATEUR DANS LE TERMINAL

Le menu affiché est le suivant (Figure 3) :

```
Que voulez-vous faire?
1. Afficher les informations du fichier
2. Afficher la liste des vertices
3. Afficher la liste des normales
4. Afficher la liste des triangles du fichier
5. Afficher le fichier chargé avec openGL
6. Créer un fichier stl à partir des infos chargées
7. Supprimer 1 vertex (le meilleur candidat)
0. Quitter
```

Figure 3: le menu

1. Afficher les informations du fichier :

Le header du fichier est affiché, ainsi que le nombre de nœuds et de triangles, et le nombre de nœuds candidats pour être supprimés lors d'une réduction.

Dans le cas du cube par défaut, le résultat obtenu est le suivant (Figure 4) :

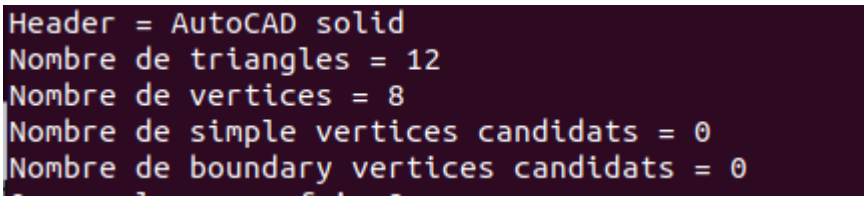


Figure 4: les informations du fichier

2. Afficher la liste des nœuds :

La liste des nœuds est affichée, avec leurs coordonnées suivies de leur type (s pour *simple*, c pour *complex*, b pour *boundary*), comme le montre la Figure 5. Un décompte du nombre de nœuds de chaque type est également affiché. Les différents types de nœud sont explicités dans la partie Choix de l'algorithme (Figure 5).

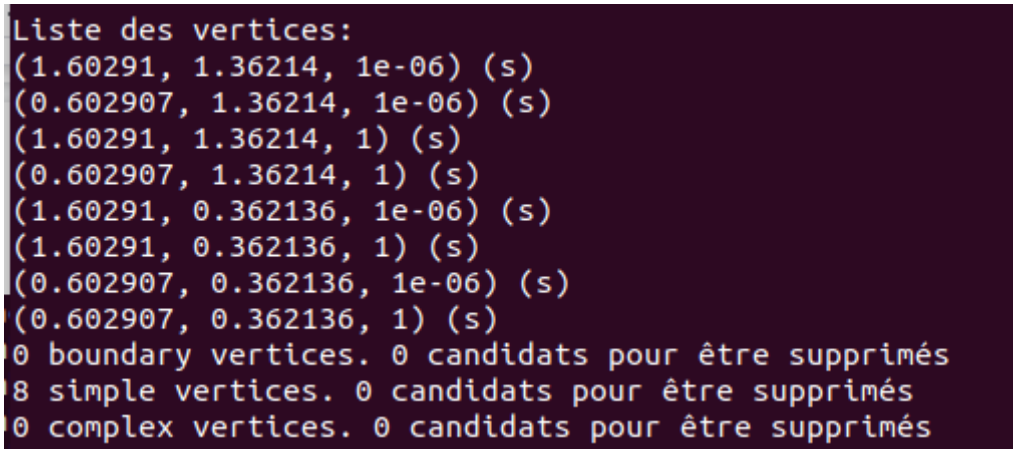


Figure 5: Liste des nœuds

3. Afficher les normales :

De la même manière qu'en b), la liste des normales est affichée.

4. Afficher la liste des triangles du fichier :

Tous les triangles du fichier sont affichés, avec en premier les coordonnées de la normale puis les coordonnées des nœuds comme on peut le voir sur la Figure 6.

```
---- TRIANGLE ----
(0, -0, 1)
(0.602907, 1.36214, 1)
(1.60291, 0.362136, 1)
(1.60291, 1.36214, 1)

---- TRIANGLE ----
(0, -0, -1)
(0.602907, 1.36214, 1e-06)
(1.60291, 1.36214, 1e-06)
(0.602907, 0.362136, 1e-06)

---- TRIANGLE ----
(0, -0, -1)
(0.602907, 0.362136, 1e-06)
(1.60291, 1.36214, 1e-06)
(1.60291, 0.362136, 1e-06)
```

Figure 6: liste des triangles

5. Affichage du fichier chargé avec OpenGL :

Cette commande ouvre une fenêtre pour afficher l'objet (Figure 7). Les commandes disponibles alors sont listée dans la partie « SOLUTION PROPOSEE - C ».

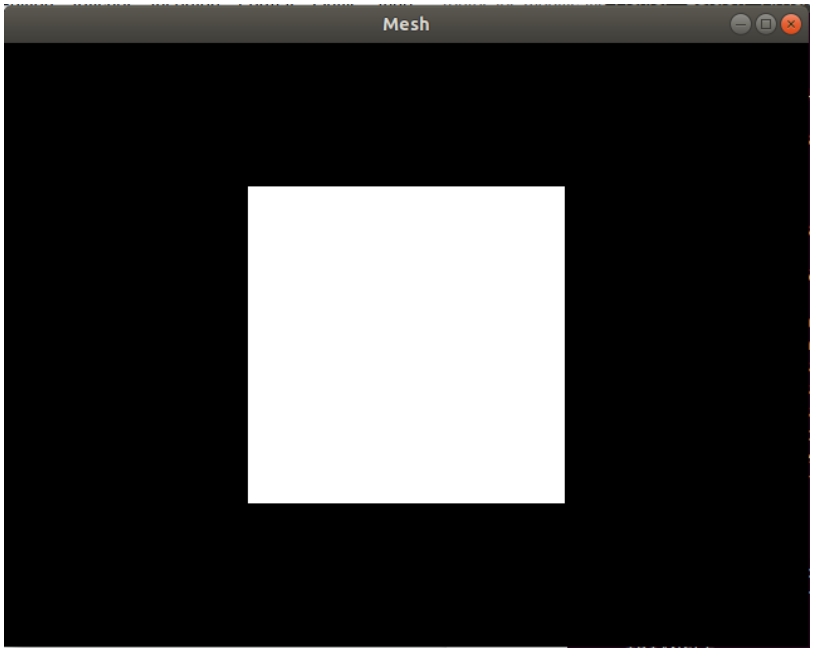


Figure 7: fenêtre d'affichage

6. Créer un fichier *stl* à partir des infos chargées :

Cela permet de faire une sauvegarde du fichier en cours en générant un fichier *stl* à partir des données stockées. Le fichier est créé dans le dossier de l'application, avec le nom *created_file.stl*.

7. Supprimer un vertex (le meilleur candidat) :

Le nœud qui est le meilleur candidat pour la réduction de maillage est supprimé. Le choix du nœud est détaillé dans la partie Choix de l'algorithme.

0. Quitter :

Permet de fermer l'application

C. Actions de l'utilisateur dans la fenetre OpenGL

Options d'affichage :

- 'p': Afficher l'objet en triangles pleins
- 'f': Afficher l'objet en fil de fer (seulement des arêtes)
- 'p': Afficher seulement les sommets
- '+': Augmenter la taille des sommets
- '-': Diminuer la taille des sommets

Options de réduction de maillage :

Lorsqu'une réduction est effectuée sur l'objet, les informations de celui-ci sont à nouveau affichées sur le terminal,

- 'r': Supprimer un Nœud (le meilleur candidat)
- '*': Suppression de tous les nœuds candidats
- Nombres N entre 1 et 9 : suppression de N*5% des nœuds de l'objet. Seuls les nœuds candidats sont supprimés. Si le nombre de nœuds que l'utilisateur souhaite supprimer est supérieur au nombre de nœuds candidat, la réduction s'arrête.

Autres options :

- 'e'/'E': Sauvegarde du fichier sous format *stl* (équivalent de la commande 6 du menu du terminal)
- Échap/'q': Quitter

Choix realises

Choix technique

Choix du langage de programmation

Nous avons choisi de réaliser une application grâce au langage C++ car c'est celui avec lequel nous somme le plus à l'aise et nous l'avons abondamment pratiqué pendant la première partie de l'année au cours de différents Travaux Pratiques. De plus, nous avons été formés à l'affichage avec la librairie OpenGL dans les cours d'IMAGRV, et cela nous a permis de mettre en place la partie affichage de l'objet en temps réel. Nous avons opté pour une Programmation Orientée Objet pour la grande majorité du code car les données contenues dans un fichier *stl* s'y prêtent bien (nœuds, triangles, normales).

Choix du type de fichier STL traite

Le type de fichier *stl* qui fonctionne avec l'application est le *stl* binaire. Celui-ci présente l'avantage d'être moins volumineux que le type ASCII et est donc généralement plus utilisé. Nous avons fait le choix de ne traiter que ce type de fichier et non les fichiers ASCII car nous ne voulions pas nous

éterniser sur la partie d'extraction des données du fichier et que nous avons eu rapidement une version fonctionnelle de l'extraction des données d'un fichier *stl* binaire.

Choix realises pour l'extraction des donnees du fichier stl

L'extraction des données depuis le fichier *stl* est nécessaire au développement de l'application. Cependant, elle n'en est pas l'objectif principal. C'est pourquoi nous avons choisi de ne pas passer beaucoup de temps à la mettre en place. Nous nous sommes donc directement inspirés du code de *dillonhuff* que nous avons bien entendu adapté à notre vision de l'application. La licence correspondant à ce code est fournie dans notre dossier git et nous permet bien de copier, modifier, distribuer tant que nous fournissons la licence et mentionnons le Copyright.

Choix des outils de developpement

Nous avons choisi d'utiliser git afin de pouvoir travailler tous les deux simultanément sur le code.

Choix de l'algorithme

Pour avoir une idée des algorithmes de décimation qui sont utilisés aujourd'hui, nous avons d'abord fait quelques recherches sur internet et synthétisé le fonctionnement de 3 algorithmes qui revenaient souvent dans nos recherches. Le document de synthèse est fourni en pièce jointe. Nous avons opté pour l'algorithme vtkDecimate de Visual Toolkit car il nous a semblé le plus abordable.

Fonctionnement de l'algorithme

Les grandes étapes de l'algorithme consistent à :

- Classifier les nœuds selon leur géométrie locale et leur topologie
- Définir les critères de suppression de chaque type de nœud
- Supprimer les nœuds candidats et remailler le trou généré

On commence par la 1ère étape :

A. Classification des nœuds

On a établi une méthode de la classe « Vertex » qui s'appelle vertexType qui retourne un caractère pour caractériser la topologie d'un nœud.

« s » pour un *simple vertex*

« b » pour *boundary vertex*

« c » pour *complex vertex*

Le traitement consiste à parcourir l'ensemble des nœuds connectés (en bleu dans la Figure 8) au nœud en question (en noir dans la Figure 8) et compter le nombre de triangles qu'ils ont en commun. En comparant ce nombre pour chaque nœud connecté, on peut déduire si le nœud d'intérêt est simple, complexe, ou frontière.

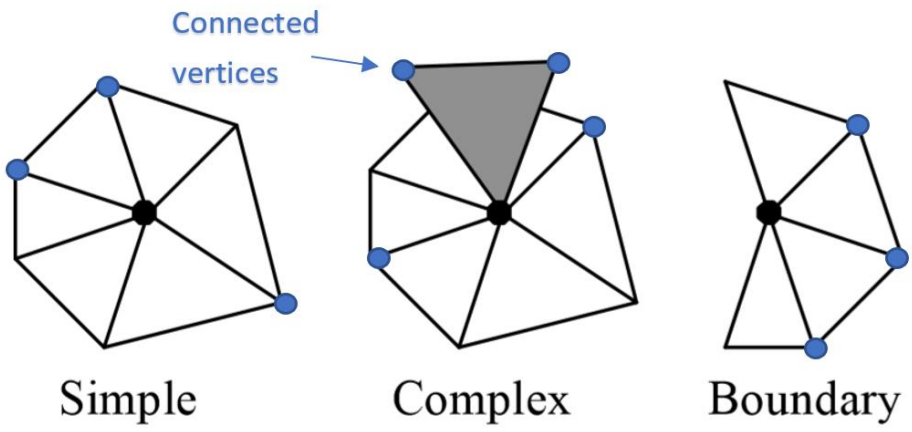


Figure 8: Classification des nœuds

B. CRITERE DE SUPPRESSION

Pour les nœuds de type simple ou frontière, une distance est calculée :

- Pour les nœuds simples, elle correspond à la distance entre le nœud en question et le nœud moyen calculé pour les nœuds connectés.
- Pour les nœuds frontières, c'est la distance entre le nœud en question et la droite liant les deux nœuds limite du cycle formé par les nœuds connectés. (Voir Figure 9)

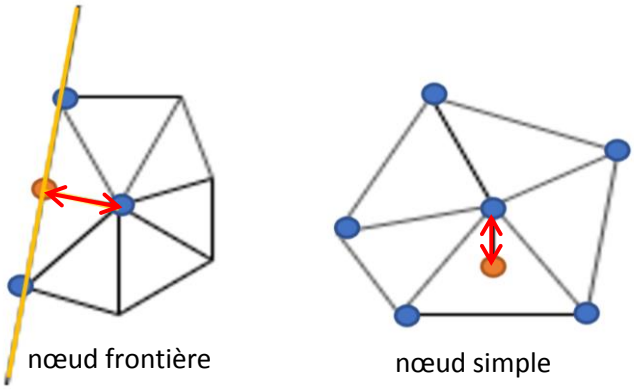


Figure 9: critères de suppression

La distance calculée est comparée à une distance limite fixée. Si celle-ci est inférieure à la limite, le nœud d'intérêt est alors candidat pour la suppression. Nous choisissons en priorité pour la suppression les nœuds candidats simples, comme le préconise l'algorithme vtkDecimate. Plus la distance calculée est faible, meilleur est le candidat.

C. SUPPRIMER LE NŒUD CANDIDAT ET TRIANGULARISER LE TROU

Une fois le choix de nœud effectué, on le supprime ainsi que les triangles qui y sont connectés. Ensuite, on fait la triangularisation du trou formé par la suppression de nœud.

La démarche qu'on a suivie pour la triangularisation est simple : on fixe un nœud connecté arbitrairement et le connecte à tous les autres nœuds du cycle, comme le montre la Figure 10.

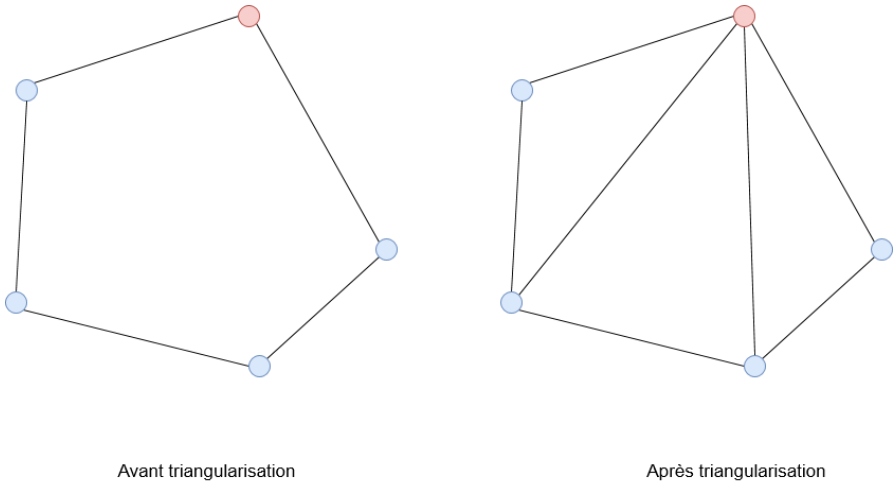


Figure 10: triangularisation d'un trou

IMPLEMENTATION DE L'ALGORITHME

CHOIX DES STRUCTURES

Nous avons défini trois classes afin d'implémenter l'algorithme de réduction de polygones. Les diagrammes de classe correspondants est exposé Figure 11.

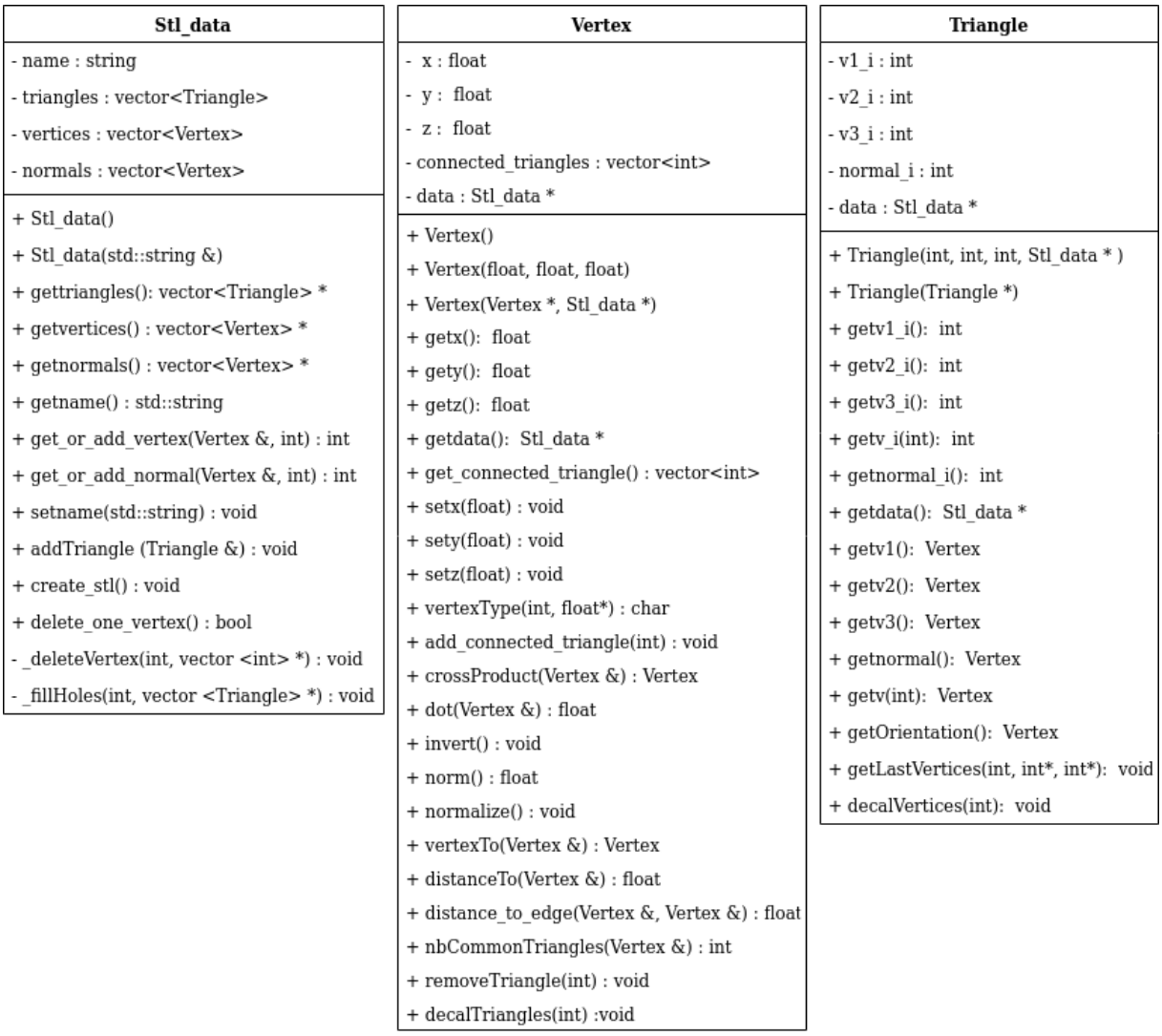


Figure 11: les structures

i. CLASSE STL_DATA

Cette classe permet de stocker toutes les informations issues d'un fichier *stl*. Lors de sa première instanciation, elle extrait toutes les informations du fichier *stl* que l'on souhaite traiter. Elle contient donc en attribut une liste de triangles, de nœuds, et de normales.

Toutes les opérations que nous réalisons sur l'objet sont dans cette classe, à savoir la réduction de maillage et la construction d'un fichier *stl* à partir des données modifiées.

ii. CLASSE VERTEX

Les attributs de cette classe représentent les éléments caractérisant un nœud ainsi qu'un pointeur de type *Stl_data* pour référencer l'objet auquel le nœud est associé.

Cette classe implémente une méthode qui permet, à partir de la géométrie locale du nœud, de donner le type de celui-ci (simple, complexe, ou frontière) et de calculer la distance d'intérêt définie précédemment. D'autres méthodes utiles à la manipulation des nœuds sont également

implémentées, comme le calcul de la distance par rapport à un autre nœud, le produit scalaire et vectoriel (un vecteur pouvant être représenté par un nœud).

iii. Classe Triangle

Cette classe est implémentée pour faciliter tout traitement précédemment évoqué. Chaque triangle est caractérisé par trois entiers correspondant aux indices de ses trois sommets la une liste des nœuds ainsi qu'un pointeur vers l'objet *Stl_data* auquel il est associé.

Implementation des differentes fonctionnalites de l'application

Les étapes de construction de cet algorithme sont faites dans l'ordre suivant :

- Importer un fichier *stl* :

On a fait le choix d'importer un fichier binaire *stl* parce qu'il a une structure standardisée et donc il est plutôt simple à lire.

Nous nous sommes renseignés sur internet pour connaitre la structure des fichiers *stl* binaires. Celle-ci est résumée ci-dessous :

- Les 80 premiers octets sont un commentaire. Il est représenté dans notre code par un attribut de la classe *Stl_data* « name » de type chaîne de caractères.
- Les 4 octets suivants représentent un entier contenant le nombre de triangles présents dans le fichier.

Ensuite, la décomposition suivante se reproduira autant de fois que le nombre de triangles indiqué en-dessus :

- 3 fois 4 octets, chaque paquet de 4 octets représente un nombre à virgule flottante correspondant respectivement aux coordonnées (x, y, z) de la direction normale au triangle.
- 3 paquets de 3 fois 4 octets, chaque groupe de 4 octets représente un nombre à virgule flottante correspondant respectivement aux coordonnées (x, y, z) de chacun des sommets du triangle.
- Deux octets représentant un mot 16 bits de contrôle.

Dans la classe *Stl_data*, nous avons défini un constructeur qui prend en argument le chemin vers le fichier *stl*. L'extraction des données de fichier *stl* se fait à l'aide d'autres fonctions définies dans un fichier « tools.cpp » dont les fonctions sont listées dans la Figure 12.

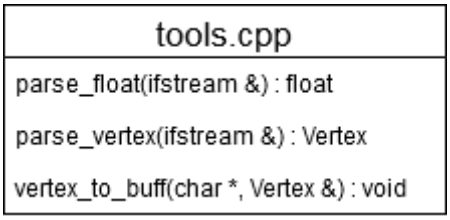


Figure 12: fonctions définies dans le fichier tools.cpp

La fonction *parse_float* permet de récupérer l'une des trois coordonnées d'un nœud/normale codé en binaire sur 4 octets et le convertir en un flottant.

La fonction *parse_vertex* extrait chaque coordonnée à l'aide de la fonction précédente et retourne un nœud.

La fonction *vertex_to_buff* réalise l'opération inverse de *parse_float* : elle convertit un élément de type Vertex en binaire sur 4 octets.

- Structurer les données :

On a organisé les données extraites à l'aide des classes précédemment mentionnées et les relations d'agrégation établies entre elles.

- Un nœud donné est associé à un objet *Stl_data* d'où l'existence d'un attribut qui pointe sur un objet *Stl_data* dans la classe Vertex. Le nœud est stocké parmi d'autres dans une liste des nœuds définie comme attribut dans la classe *Stl_data*.
- Un triangle est caractérisé par trois nœuds, chacun défini comme un attribut de type entier qui correspond à l'indice de nœud dans la liste des nœuds. On a accès à cette dernière liste grâce au pointeur de type *Stl_data* défini comme un attribut dans la classe Triangle.

- Classification des nœuds :

Dans la classe Vertex, on a implémenté une méthode *vertexType* qui permet de retourner un caractère pour distinguer les différents types de nœuds (simple, complexe, ou frontière).

Une valeur par défaut qui vaut « s » est attribuée au caractère retourné par la méthode. Le traitement se fait sur le nœud dont l'indice est donné en argument en et utilise tous les nœuds connectés à celui-ci. On a défini plusieurs cas qui nous laisse soit changer la valeur du caractère à « b » ou « c » soit la laisser à sa valeur par défaut « s » et ceci est en fonction de nombre de triangles communs entre les nœuds connectés et le nœud en question.

- Critères de suppression :

A la même fonction « vertexType », on lui a passé en argument (passage par adresse) un flottant qui correspond au critère de suppression, il s'agit d'une distance calculée en deux façons selon le type de nœud.

Pour un nœud simple : la distance est égale à celle entre le nœud en question et le nœud de coordonnées moyennes calculées par les nœuds connectés.

Pour un nœud frontière : la distance est la projection de nœud sur la droite liant les deux nœuds frontière de cycle formé par les nœuds connectés.

Ces deux distances sont expliquées dans la Figure 9.

- Suppression de nœud et triangularisation :

On a implémenté cet algorithme dans la classe *Stl_data*, il s'agit de la méthode *delete_one_vertex*() qui permet de choisir le meilleur candidat pour chaque type de nœud (best_candidate_simple/best_candidate_boundary). Entre les deux candidats, on priorise le nœud candidat simple s'il existe. Une fois le meilleur nœud candidat choisi, nous procédons à la suppression.

Nous récupérons d'abord les triangles connectés à ce nœud.

A ce niveau, on a introduit deux autres méthodes définies en privé dans la classe et qui prennent en argument le nœud candidat et un vecteur de triangles connectés à ce nœud. Elles sont appelées dans la méthode « delete_one_vertex » pour supprimer le nœud candidat et triangulariser le trou généré.

- *_fillholes* est la fonction qui permet la triangularisation, elle est appelée en premier pour remplir le cercle formé par la suppression de nœud. Le principe est comme suit, un nœud est choisi au hasard pour qu'il soit commun à tous les triangles créés.
- *_deletevertex* est la fonction qui supprime le nœud candidat et les triangles associés. On fait appel à d'autres méthodes définies dans la classe Vertex et la classe Triangle.

Comme les références utilisées dans nos classes sont sous forme d'indice dans une liste (entier) et non d'adresse, nous avons dû être vigilants lors de la suppression d'un élément dans une liste : tous les éléments de la liste où a eu lieu la suppression qui ont un indice supérieur à celui de l'élément supprimé voient leur indice décrémenté. C'est pourquoi nous avons implémenté les 3 méthodes suivantes :

- *decalTriangles* de la classe Vertex et qui permet de décaler de un les indices de triangles dans la liste d'entiers *connected_triangles* en cas de suppression d'un triangle,
- *decalVertices* de la classe Triangle pour décaler de un les indices des nœuds d'un triangle au moment de suppression d'un nœud.

À la suppression d'un triangle, il faut également le supprimer de la liste des triangles connectés des nœuds auquel il est connecté. Nous avons donc créé une méthode *removeTriangle* de la classe Vertex et qui enlève le triangle dont l'indice est passé en argument de la liste des triangles connectés.

- Création du fichier *stl* :

Une fois que tout l'algorithme de réduction est appliqué sur le fichier *.stl* donné en entrée, nous avons implémenté une autre méthode de la classe *Stl_data* qui permet de créer un fichier *stl* à partir des données contenues dans l'instance de *Stl_data* sur laquelle nous travaillons.

Pour chaque triangle, les trois sommets sont convertis en un buffer de 12 octets contenant les trois coordonnées du nœud courant à l'aide de la fonction *vertex_to_buff* définie dans le fichier *tools.cpp* (voir Figure 12).

AFFICHAGE AVEC OPENGL

Pour visualiser nos résultats, nous avons utilisé les fonctions d'OpenGL pour construire un algorithme qui permet l'affichage en 3D.

Pour cela, nous utilisons la fonction *opengl_display* qui prend en argument un pointeur sur un objet *Stl_data*. Elle reprend tous les éléments nécessaires à l'affichage (liste des triangles avec leurs nœuds et leur normale). En premier lieu, elle calcule le barycentre de l'objet à afficher afin de le centrer. Ensuite, nous réalisons la configuration de la fenêtre comme nous l'avions fait lors des TP d'IMAGRV. Enfin, toutes les callbacks nécessaires au bon fonctionnement de la fenêtre (clavier, souris, etc.) sont définies. Les callbacks sont définis comme suit :

- *glutDisplayFunc(display)* : définition de la fonction d'affichage d'OpenGL. Cette fonction est appelée à l'affichage de chaque frame.
- *glutKeyboardFunc(clavier)* : la fonction clavier permet l'interaction entre le clavier et l'utilisateur en temps réel. Comme il a été décrit dans la partie solution proposée, l'utilisateur est capable de réduire le nombre de polygones en temps réel en appuyant sur les touches de claviers mentionnées précédemment.
- *glutMouseFunc(souris)* : elle permet d'actualiser la position de la souris dans la scène, utile pour réaliser des rotations sur l'objet affiché avec la souris.
- *glutMotionFunc(deplacementSouris)* : elle permet de modifier les angles de rotations de l'objet dans la scène en fonction de la dernière position de la souris et de sa position actuelle.
- *glutReshapeFunc(redimensionner)* : pour redimensionner la fenêtre.
- *glutSpecialUpFunc(releaseSpecialKey)* : Les *SpecialKeys* correspondent aux flèches du clavier. Elles permettent de faire reculer ou avancer la caméra selon la touche pressée.

RESULTATS OBTENUS

Sont exposés ci-dessous les résultats obtenus lorsque notre application est utilisée sur un fichier *stl*. Les tests sont effectuée avec le fichier *rabbit.stl* qui se trouve dans le répertoire git.

Les Figure 13 et Figure 14 montrent la différence entre un fichier *stl* non réduit et un fichier *.stl* réduit par l'algorithme à un pourcentage de 30% de nombre de nœuds et un fichier *stl* réduit au maximum possible.

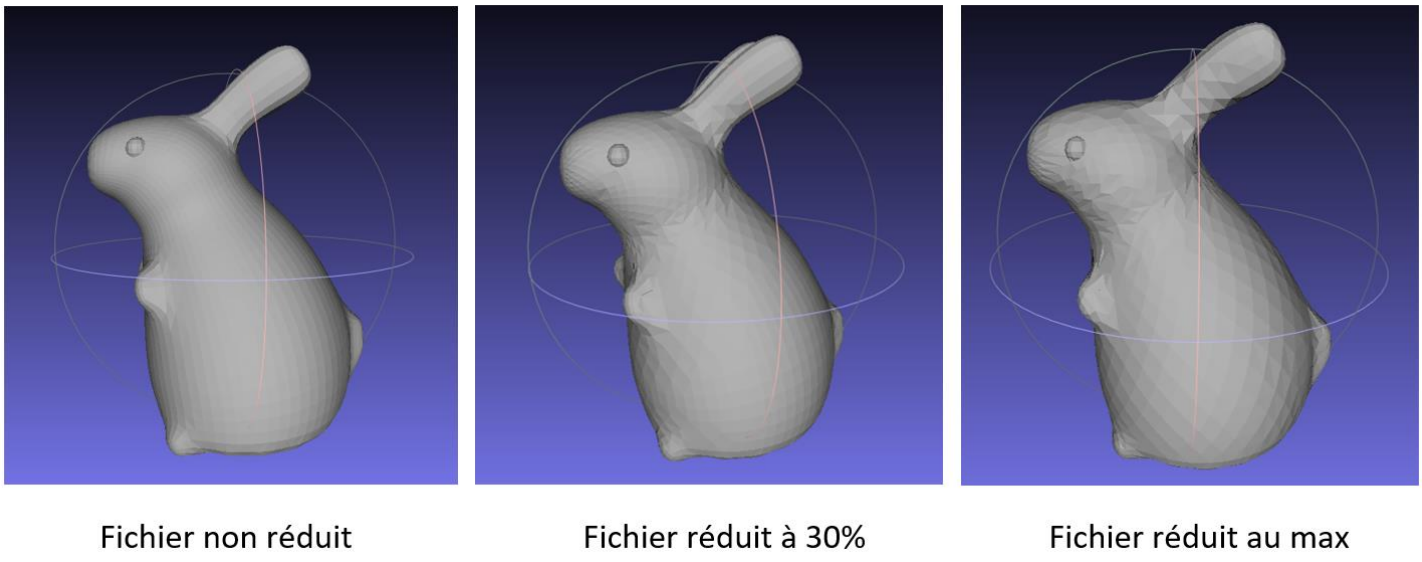
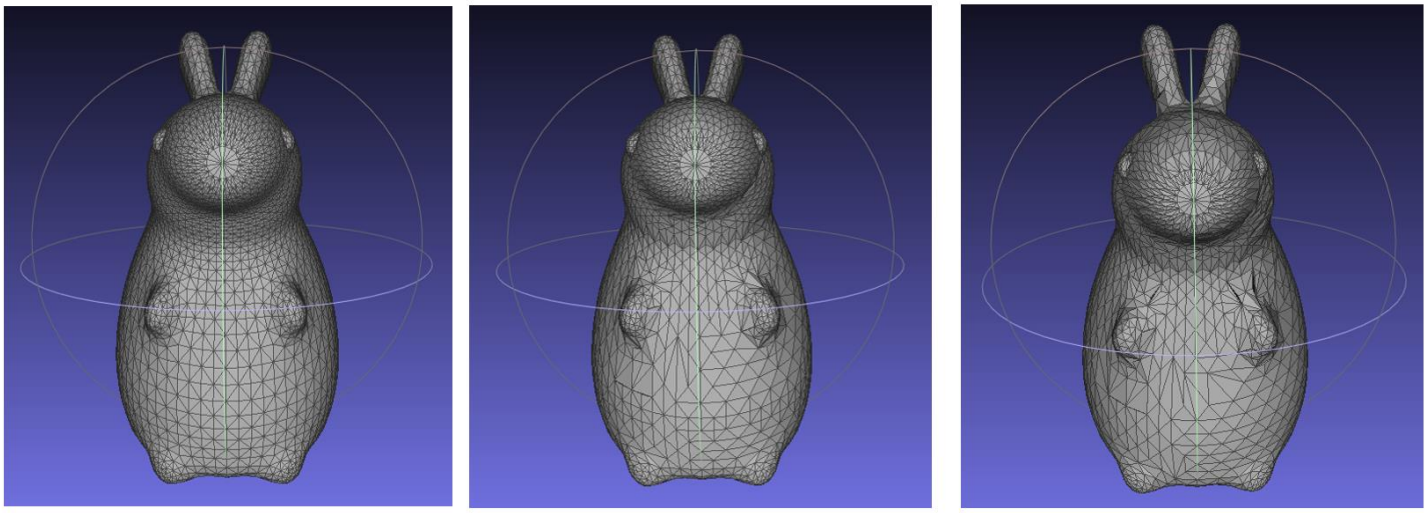


Figure 13: comparaison 1 des résultats



Fichier non réduit | Fichier réduit à 30% | Fichier réduit au max

Figure 14: comparaison 2 des résultats

DIFFICULTES RENCONTREES

CHOIX DES STRUCTURES DE STOCKAGE DES DONNEES EXTRAITES DU FICHIER STL

La première difficulté que nous avons rencontrée a été de rendre exploitables les données extraites du fichier *stl*. En effet, lorsque nous avons implémentés l'extraction des fichiers *stl*, les données résultantes étaient une liste de triangles, avec pour chaque triangle ses nœuds et sa normale.

Le diagramme de la classe Triangle était donc celui de la Figure 15.

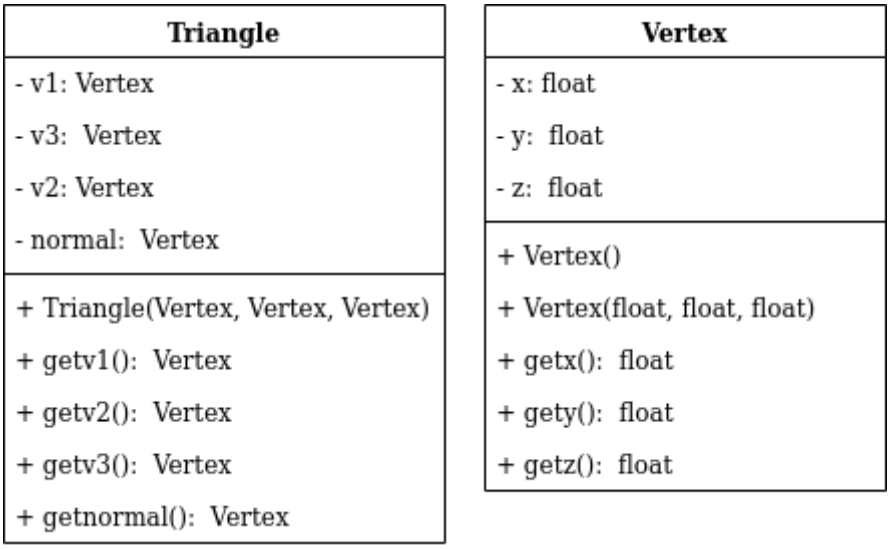


Figure 15: Structure choisie initialement

Les données, sous cette forme, nous permettaient d'afficher toutes les informations générales concernant le fichier : liste des triangles, nombre de triangles, contenu des triangles, et également de reconstruire l'objet sous OpenGL par exemple. Seulement, plusieurs informations manquaient et demandaient un traitement un peu complexe pour y accéder :

- Pour un nœud donné, les triangles qui y sont connectés
- Le nombre total de nœuds distincts

Ces informations ne sont pas fournies dans le fichier *stl*, mais peuvent être déduites d'un traitement. Comme ces informations sont utile à l'application de l'algorithme de réduction que nous avons choisi d'implémenter, nous avons choisi de les déduire en amont (avant l'application de l'algorithme) une fois pour toute de manière à ce que l'application de la réduction ne soit pas trop chronophage (puisque les informations dont il a besoin seront déjà extraites).

Nous avons donc modifié nos structures de données comme suit :

- Les nœuds et normales sont stockées dans deux listes à part
- Les listes de normales et nœuds ne contiennent pas de doublons
- Chaque nœud ou normale contient la liste des triangles qui y sont connectés (sous forme d'indice)
- Chaque triangle stocke les nœuds et normale sous forme d'indice (nombre entier) correspondant à la position du nœud/de la normale dans sa liste.

Les diagrammes de classe correspondants sont ceux exposés dans la Figure 11.

Ainsi, les données extraites sous cette forme (Figure 11) étaient plus facilement exploitables. Le seul bémol de cette modification est le temps d'extraction qui augmente. Nous avons considéré que c'était un mal pour un bien, car le traitement doit dans tous les cas être fait, et il vaut donc mieux qu'il soit fait une seule fois au lancement de l'application plutôt qu'à chaque opération de réduction de l'utilisateur.

DIFFICULTE TECHNIQUE CONCERNANT LES POINTEURS ET LES VECTEURS

Comme précisé plus haut, nous avons dû changer nos structures de données afin de garder un lien entre les nœuds et les triangles auquel ils sont connectés, et inversement. Ces deux éléments sont stockés sous forme de liste. Nous avons donc voulu, pour chaque triangle, stocker l'adresse de ses nœuds. Or nous nous sommes heurtés à la difficulté suivante : il est impossible d'obtenir l'adresse d'un élément d'un vecteur, ou du moins nous n'avons trouvé aucun moyen : l'instruction "&my_vector.at(i)" ne permet pas cela. Nous avons donc essayé de stocker un itérateur plutôt qu'une adresse, mais cela n'a pas donné de meilleur résultat.

Ce problème s'est posé pour tous les cas où il nous fallait créer un lien vers un élément d'un vecteur :

- Stockage des triangles connectés à un nœud,
- Stockage des nœuds et normales d'un triangle.

C'est pourquoi nous avons choisi de stocker simplement l'indice du nœud dans sa liste, sous forme d'entier.

PISTES D'AMELIORATION

L'application que nous proposons ne répond pas totalement au Cahier Des Charges du projet puisqu'elle ne permet pas de guider la décimation, mais seulement de choisir la quantité de nœuds que l'on veut supprimer. Une piste d'amélioration serait donc de mettre en place un outil de sélection d'un ensemble de nœuds sur lesquels on effectue la décimation.

Il y a également moyen d'améliorer le temps d'exécution du programme : lorsque l'on charge un fichier *stl* d'environ 10 000 triangles, le lancement prend 50 secondes (le test est effectué dans nos conditions de travail, sur le fichier *moto.stl* fourni dans le dossier git). La création d'un fichier *stl* à partir des données modifiées est quant à elle rapide (quelques millisecondes pour le même fichier).

Le critère utilisé pour la suppression ou non d'un nœud peut être également amélioré : nous n'avons traité que 3 types de nœuds (simple, complexe, et frontière) tandis que l'algorithme vtkDecimate prend en compte deux autres types de nœud : intérieur et extérieur. De plus, lors du calcul de la distance au plan dans le cas d'un nœud "simple", nous ne calculons pas la distance au plan moyen mais la distance au point moyen. Le critère peut donc être biaisé, comme dans l'atteste la Figure 16: dans les deux situations qu'elle présente, la distance du point rouge au plan moyen (en vert) formé par les points bleus est la même, tandis que la distance au point moyen (double flèche rouge) est différente (Figure 16).

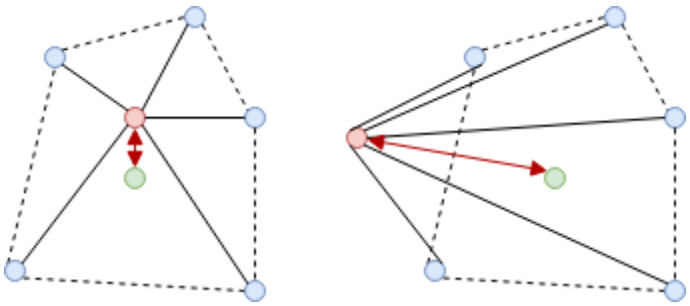


Figure 16 : fiabilité de critère de la distance au point moyen

CONCLUSION

Ce projet nous a permis d'améliorer nos compétences en programmation C++, notamment en programmation avec OpenGL. Nous avons dû nous organiser avec rigueur pour remplir nos tâches et respecter le délai. Au niveau scolaire, nous avons pu acquérir des nouvelles connaissances et les mettre en pratique à savoir :

- Créer un algorithme de réduction de maillage d'un objet 3D et comprendre la démarche mise en place
- Mettre en œuvre nos compétences en programmation et les développer
- Apprendre à gérer les différentes versions de code avec git

Au niveau professionnel, ce projet nous a donné l'opportunité de travailler en binôme et donc nous avons dû apprendre à faire des compromis et à s'organiser pour mener à bien notre réalisation.

Nous tenons à remercier M. PEILLARD pour l'encadrement et l'accompagnement qu'il nous a montré pendant cette période de projet.