

---

Université de Montpellier  
Faculté des Sciences

Année universitaire 2024–2025

## Traitement de géométrie hors mémoire (Out-of-Core)

HAI823I – Travail d'Etude et de Recherche

**Hugo Vaillant, Donovann Zassot, Mateusz Birembaut**

Groupe Orcoeur, Master Informatique, Parcours IMAGINE

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Contexte et Définitions</b>	<b>4</b>
2.1	Qu'est-ce qu'un maillage 3D ? . . . . .	4
2.2	Pourquoi simplifier un maillage ? . . . . .	4
2.3	Simplification Out-of-Core . . . . .	5
2.4	État de l'art . . . . .	5
2.4.1	Simplification par partitionnement . . . . .	5
2.4.2	Simplification par opérations locales . . . . .	6
<b>3</b>	<b>OoCSx</b>	<b>7</b>
3.1	Notations . . . . .	7
3.2	Pipeline OoCSx . . . . .	7
3.3	Tri externe . . . . .	8
3.4	Types de maillages supportés . . . . .	9
3.5	Déréférencement des triangles . . . . .	9
3.6	Découpage du maillage . . . . .	10
3.6.1	Calculer le volume englobant . . . . .	10
3.6.2	Une méthode pour calculer l'indice de cellule d'un sommet . . . . .	10
3.6.3	Résultat . . . . .	10
3.7	Écriture des données intermédiaires . . . . .	11
3.8	Calcul des représentants . . . . .	13
3.9	Déréférencement des clusters . . . . .	15
3.10	Résultats . . . . .	16
3.10.1	Comparaison qualitative des résultats . . . . .	16
<b>4</b>	<b>Partie Adaptative</b>	<b>17</b>
4.1	Phase 1 : Quantification du maillage . . . . .	17
4.2	Phase 2 : Construction d'un arbre de partition adaptatif . . . . .	19
4.2.1	Initialisation du nœud racine et file de priorité . . . . .	19
4.2.2	Boucle de subdivision adaptative . . . . .	19
4.2.3	Résultats : Visualisation des feuilles adaptatives . . . . .	20
4.3	Phase 3 : Regroupement des sommets et reconstruction du maillage . . . . .	21
4.3.1	Comparaison qualitative des résultats . . . . .	21
<b>5</b>	<b>Partie Streaming</b>	<b>22</b>
5.1	Bordures de notre tampon . . . . .	22
5.1.1	Structure de demi-arêtes . . . . .	23
5.2	Phase 1 : La lecture . . . . .	24
5.3	Phase 2 : La décimation . . . . .	24
5.4	Phase 3 : L'écriture . . . . .	26
5.5	Explication générale de la méthode . . . . .	27
5.6	Problème rencontré . . . . .	28
<b>6</b>	<b>Résultats et comparaisons</b>	<b>30</b>
6.1	Comparaison visuelle de la simplification . . . . .	30
6.2	Distance d'Hausdorff . . . . .	30
6.2.1	Définition de la distance de Hausdorff . . . . .	30
<b>7</b>	<b>Discussion</b>	<b>32</b>
7.1	Gantt . . . . .	32
7.2	Méthode de correction des erreurs du stream . . . . .	32
7.3	Conclusion . . . . .	34



# 1 Introduction

La complexité des maillages géométriques ne cesse de croître, notamment en raison des progrès des techniques de numérisation 3D et de simulation. Les modèles obtenus peuvent contenir plusieurs millions, voire milliards de triangles, dépassant largement les capacités de mémoire vive des machines standards.

Les premières méthodes de simplification supposent que l'ensemble du maillage peut être chargé en mémoire, ce qui limite leur utilisation à des modèles de taille modérée. Pour les maillages de grande taille, cette hypothèse devient irréaliste : la mémoire disponible ne permet plus de stocker l'ensemble de la géométrie ni d'effectuer des opérations globales sur sa structure.

Dans ce contexte, des approches dites *Out-of-Core* ont été développées. Elles reposent sur l'idée de traiter les données par blocs, en ne maintenant en mémoire qu'une partie du maillage à chaque instant. Ce paradigme permet de s'affranchir des limites matérielles, mais impose de nouvelles contraintes algorithmiques, notamment en ce qui concerne la gestion locale de la topologie et la cohérence du résultat.

## 2 Contexte et Définitions

### 2.1 Qu'est-ce qu'un maillage 3D ?

Un **maillage 3D** (ou *mesh*) est une structure géométrique utilisée pour représenter la forme d'un objet tridimensionnel. Il est composé :

- **de sommets** (*ou vertices*), qui définissent des points dans l'espace,
- **d'arêtes**, qui relient deux sommets,
- **et de faces**, souvent triangulaires ou quadrilatérales, qui délimitent des surfaces.

Dans ce travail d'étude, nous nous concentrons sur les **maillages triangulaires**, dans une forme particulière de représentation : la *soupe de triangles* (*triangle soup*). Il s'agit d'un ensemble de triangles définis chacun par trois sommets, sans structure topologique explicite : les triangles sont stockés indépendamment et ne partagent pas forcément leurs sommets ou arêtes, même s'ils sont géométriquement adjacents. Cette structure est simple à manipuler, facile à stocker, et particulièrement adaptée à un traitement Out-of-Core.

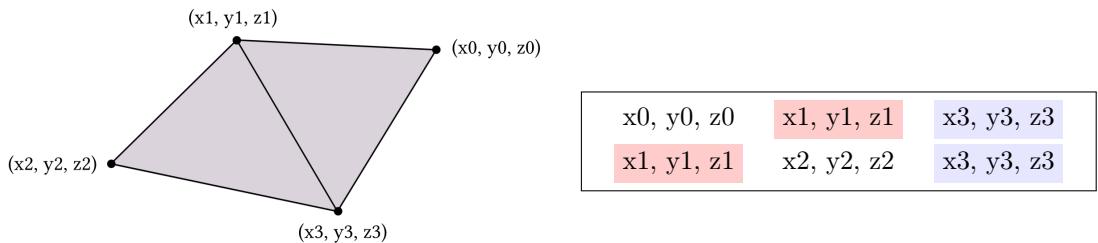


Figure 1: Exemple de soupe de triangles : à gauche, le maillage ; à droite, deux triangles partageant des sommets (en couleur) sans connectivité explicite.

Bien que nos algorithmes soient conçus pour fonctionner avec des triangles, ils s'appliquent également aux maillages quadrangulaires : chaque face quadrilatérale peut être décomposée en deux triangles avant traitement.

### 2.2 Pourquoi simplifier un maillage ?

Les maillages complexes, notamment issus de la numérisation 3D ou de la modélisation détaillée, peuvent contenir des millions, voire des milliards de faces. Leur manipulation nécessite alors une mémoire importante, une puissance de calcul élevée et peut devenir un goulot d'étranglement pour le rendu en temps réel ou les simulations.

La simplification de maillage vise à réduire le nombre d'éléments (sommets, arêtes, faces) tout en conservant autant que possible l'apparence et la structure de l'objet initial. L'objectif est d'obtenir un compromis acceptable entre qualité visuelle et complexité du maillage.

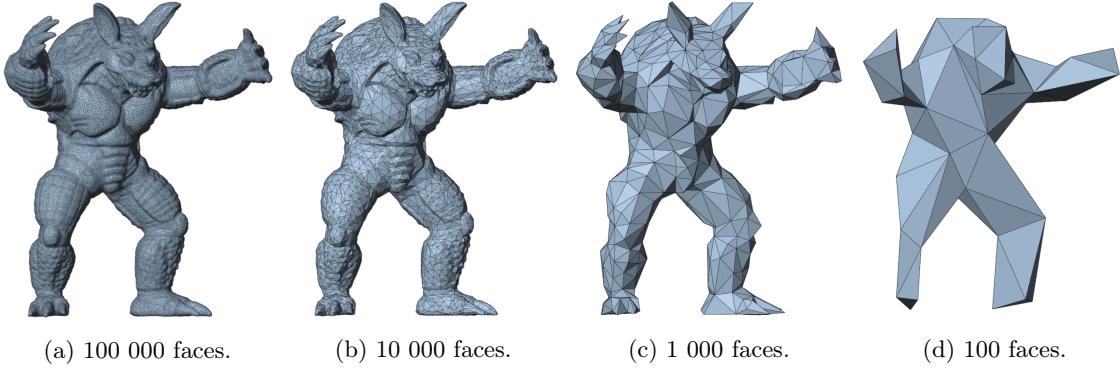


Figure 2: Illustration de la simplification progressive du modèle *Armadillo*.

### 2.3 Simplification Out-of-Core

La plupart des algorithmes de simplification traditionnels supposent que l'ensemble du maillage peut être chargé en mémoire (on parle alors d'approche *in-core*). Cette hypothèse devient cependant irréaliste face à des maillages de grande taille. Pour traiter ces volumes de données, on adopte une approche dite **Out-of-Core**, qui regroupe un ensemble de techniques permettant de manipuler des maillages trop volumineux pour tenir en mémoire vive. Ces méthodes reposent sur des modèles de calcul conçus pour minimiser l'utilisation de la RAM, en s'appuyant notamment sur :

- le *traitement par blocs*, qui divise le maillage en sous-ensembles plus petits ;
- l'utilisation de structures de données externes, telles que des fichiers sur disque ou des arbres spatiaux persistants ;
- des algorithmes de *streaming*, dans lesquels les triangles sont lus et traités séquentiellement, sans nécessiter un chargement global.

Travailler en Out-of-Core impose des contraintes supplémentaires : comme l'algorithme ne dispose que d'une vision partielle du maillage à chaque instant, il doit être conçu pour préserver la cohérence globale (topologie, continuité des surfaces, absence de trous) tout en fonctionnant localement.

### 2.4 État de l'art

La simplification de maillages massifs, en particulier dans un contexte *out-of-core*, a donné lieu à de nombreuses approches. Ces méthodes de **simplification** se basent généralement sur deux grandes stratégies : le **partitionnement spatial** et les **opérations locales de décimation**.

#### 2.4.1 Simplification par partitionnement

Cette méthode divise le maillage en zones indépendantes afin de simplifier localement chaque partie. Une technique répandue, le clustering spatial, regroupe les sommets proches selon une grille régulière. Cette approche, simple et rapide, convient particulièrement aux environnements à mémoire limitée, puisqu'elle ne nécessite pas le chargement complet du modèle.

Rossignac et Borrel [3] proposent en 1993 de remplacer tous les sommets d'une cellule de la grille par un seul point. Malgré sa rapidité, cette méthode donne un résultat de qualité moyenne à cause du positionnement approximatif des sommets.

Lindstrom propose en 2000 l'algorithme **OoCS** [2], qui améliore la qualité du résultat à l'aide des *quadric error metrics*. Il permet de traiter de grands modèles, mais reste limité par la mémoire requise pour stocker le résultat. Cette contrainte est levée dans **OoCSX** [1], une version incrémentale qui écrit directement les données sur disque. D'autres variantes rendent la subdivision spatiale

adaptative. C'est le cas de l'approche de Shaffer et Garland [4], qui utilise un arbre BSP pour s'ajuster à la complexité géométrique locale.

#### 2.4.2 Simplification par opérations locales

Cette approche réduit progressivement le maillage en appliquant des opérations successives, typiquement des **contractions d'arêtes**, fusionnant deux sommets et supprimant les faces dégénérées. L'approche suit généralement une stratégie gloutonne, où chaque contraction est choisie pour minimiser une erreur basée sur les *quadric error metrics*. Cette méthode génère des résultats de qualité, mais se heurte à des difficultés en *out-of-core* en raison de son coût mémoire et des accès non localisés. Pour surmonter ces limitations, Wu et Kobbelt [5] proposent en 2003 une méthode de décimation **en streaming**, dans laquelle le maillage est traité en une seule passe séquentielle à l'aide d'un tampon mémoire de taille limitée.

Dans notre travail d'étude, nous nous sommes concentrés sur trois algorithmes : l'approche **OoCSX** de Lindstrom et Silva, la version adaptative basée sur la subdivision spatiale de Shaffer et Garland, ainsi que l'algorithme de décimation en streaming proposé par Wu et Kobbelt.

### 3 OoCSx

Dans cette première partie, nous allons présenter notre implémentation de l'algorithme de simplification de maillage OoCSx décrit dans l'article *A Memory Insensitive Technique for Large Model Simplification* [1]

#### 3.1 Notations

$G_x$  Indice de la cellule du sommet  $x$ .

$x_G$  Sommet représentant (optimal) de la cellule  $G$ .

$\bar{n}_t$  Normale du plan formé par le triangle  $t$ .

$\bar{n}_t^T$  Transposé de la normale  $\bar{n}_t$ .

#### 3.2 Pipeline OoCSx

- **Déréférencement des triangles**

Chargement d'un fichier OBJ ou PLY, que nous convertissons en une « soupe de triangles » binaire.

- **Découpage du maillage et écriture des informations intermédiaires**

Pour chaque triangle de notre maillage, on va calculer l'indice de cellule de ses trois sommets. Puis nous écrivons dans deux fichiers binaires différents les informations suivantes :

- Fichier **plane\_equation.bin**, on écrit pour chaque sommet un couple  $(G_x, \bar{n}_t)$
- Fichier **triangle\_cluster.bin** pour les triangles non dégénérés ( $G(v_1) \neq G(v_2) \neq G(v_3)$ ), on enregistre les trois indices de cellule de ses sommets :  $(G(v_1), G(v_2), G(v_3))$ .

Enfin le fichier **plane\_equation.bin** est trié par l'indice de la cellule  $G$  via un tri externe afin de les accumuler avec un simple balayage séquentiel lors de l'étape suivante.

- **Calcul des représentants**

On cumule tous les  $\bar{n}_t \bar{n}_t^T$  pour chaque cellule  $G$  afin de calculer la quadrique de cette cellule. Le représentant optimal  $x_G$  est ensuite déterminé grâce à la fonction **findOptimalVertex** qui retourne une position minimisant l'erreur quadrique. On écrit ensuite dans un nouveau fichier nommé **representatives.bin**, les paires  $(G, x_G)$ .

- **Déréférencement de triangle\_cluster.bin**

On remplace dans le fichier **triangle\_cluster.bin** les indices de cellules  $G_x$  par les coordonnées des représentants des cellules. Le résultat est un fichier **simplified.bin** listant, pour chaque triangle, trois positions 3D des représentants au lieu des indices de grille.

### 3.3 Tri externe

Pour trier un fichier tout en limitant l'utilisation de la mémoire, nous avons implémenté un tri externe en deux étapes. Ce tri sera utilisé lors du déréférencement des triangles, après la création du fichier `plane_equation.bin` et lors de l'étape finale de déréférencement du fichier `triangle_cluster.bin`.

#### Étape 1 : création des runs

1. Lire séquentiellement des blocs (« chunks ») de taille paramétrable depuis le fichier d'origine.
2. Trier chaque bloc en mémoire avec `std::sort`.
3. Écrire le bloc trié dans un fichier temporaire `run_i`.

#### Étape 2 : fusion k-voies

1. Ouvrir chaque fichier `run_i` et en lire un élément initial.
2. Maintenir un tas-min (min-heap) contenant la tête de chaque run.
3. Répéter :
  - Extraire du tas l'élément le plus petit et l'écrire dans le fichier de sortie.
  - Remplacer dans le tas cet élément par le prochain élément de la même run (lecture séquentielle).

#### Complexité

- **Temps** :  $O(n \log n)$  pour trier chaque bloc +  $O(n \log k)$  pour fusionner  $k$  runs.
- **Mémoire** :  $O(\text{CHUNK\_SIZE})$  en mémoire vive.

### 3.4 Types de maillages supportés

Notre implémentation d’OoCSx accepte tout maillage triangulaire basique aux formats OBJ et PLY.

### 3.5 Déréférencement des triangles

Après avoir chargé le maillage au format OBJ ou PLY, nous convertissons le maillage en une « soupe de triangles » binaire, où chaque triangle est représenté par les coordonnées de ses trois points. Pour minimiser la mémoire utilisée, nous appliquons une méthode Out-Of-Core inspirée de l’appendice de l’article, elle est organisée en trois passes successives et repose sur des fichiers temporaires.

#### Séparation initiale

Pour tout exemple de fichiers, le numéro avant "|" représente le numéro de la ligne dans le fichier.

Listing 1: Fichier des sommets (x y z)

```
1 | 0.0 0.0 0.0
2 | 1.0 0.0 0.0
3 | 0.0 1.0 0.0
4 | 0.0 0.0 1.0
5 | 1.0 1.0 0.0
```

Listing 2: Fichier des triangles

```
1 | 4 5 1
2 | 1 2 3
```

#### Pass 1 : tri sur v1 et remplacement

1. Tri externe du fichier des triangles sur le premier indice v1 :
2. Lecture séquentielle simultanée du fichier des triangles trié sur v1 et du fichier des sommets :
3. Les indices surlignés en jaune sont remplacés par les coordonnées des sommets.

```
1 | 1 2 3 -> 0.0 0.0 0.0 2 3
2 | 4 5 1 -> 0.0 0.0 1.0 5 1
```

#### Pass 2 : tri sur v2 et remplacement

```
1 | 0.0 0.0 0.0 2 3 -> 0.0 0.0 0.0 1.0 0.0 0.0 3
2 | 0.0 0.0 1.0 5 1 -> 0.0 0.0 1.0 1.0 1.0 0.0 1
```

#### Pass 3 : tri sur v3 et remplacement

```
1 | 0.0 0.0 1.0 1.0 1.0 0.0 1 -> 0.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0
2 | 0.0 0.0 0.0 1.0 0.0 0.0 3 -> 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0
```

Au terme de ces trois passes, chaque triangle est défini par ses trois coordonnées 3D, dans un fichier binaire nommé **simplified.bin**. Cette organisation nous permettra de traiter notre maillage avec un simple balayage séquentiel, triangle par triangle, sans avoir à chercher dans le fichier les coordonnées de ses sommets et minimisera l’utilisation de mémoire.

### 3.6 Découpage du maillage

Pour découper notre maillage, nous allons d'abord devoir créer notre grille 3D uniforme et associer un identifiant unique à chaque cellule. Cela permettra de stocker pour chaque cellule les plans la traversant et plus tard de calculer la position minimisant l'erreur quadrique. Pour ce faire, nous avons besoin de deux éléments.

#### 3.6.1 Calculer le volume englobant

Le volume englobant ou *bounding box* est le plus petit parallélépipède rectangle englobant le maillage. Cette étape ne nécessite qu'un parcours linéaire du fichier contenant le maillage. On stocke simplement les valeurs minimales et maximales sur chaque axe lors de ce parcours.

$$\text{coin inférieur gauche} = (x_{\min}, y_{\min}, z_{\min}) \quad \text{coin supérieur droit} = (x_{\max}, y_{\max}, z_{\max})$$

Une fois calculé on peut utiliser ce volume englobant ainsi que la résolution choisie par l'utilisateur pour calculer des indices uniques à chaque cellule selon la méthode suivante.

#### 3.6.2 Une méthode pour calculer l'indice de cellule d'un sommet

Soit  $r$  la résolution de la grille uniforme 3D, Pour chaque sommet  $(x, y, z)$  on peut :

1. Calculer les coordonnées de cellule sur chaque axe ( $indice_x, indice_y, indice_z$ ) :

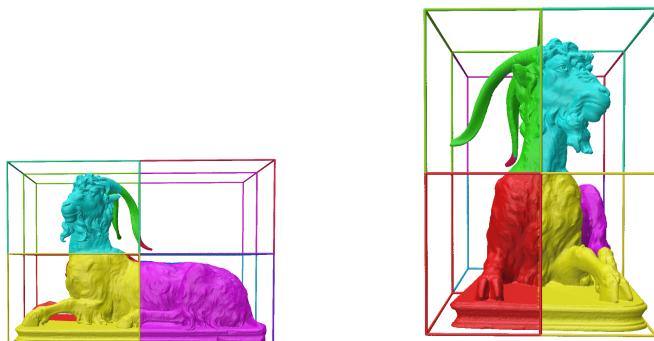
$$indice_x = \left\lfloor \frac{x - x_{\min}}{x_{\max} - x_{\min}} \times r \right\rfloor, \quad indice_y = \left\lfloor \frac{y - y_{\min}}{y_{\max} - y_{\min}} \times r \right\rfloor, \quad indice_z = \left\lfloor \frac{z - z_{\min}}{z_{\max} - z_{\min}} \times r \right\rfloor.$$

2. Utiliser ce triplet pour obtenir un identifiant de cellule unique noté  $G$  :

$$G = indice_x (r^2) + indice_y (r) + indice_z.$$

De cette manière, on peut découper notre maillage en cellules regroupant les sommets. C'est cette fusion des sommets en cellules qui permet la simplification et garantit que l'on aura au maximum  $r^3$  sommets.

#### 3.6.3 Résultat



(a) Visualisation du volume englobant d'un maillage

(b) Visualisation de l'assignation des triangles à une cellule pour une grille uniforme de résolution 2

(c) Visualisation de l'assignation des triangles à une cellule pour une grille uniforme de résolution 2

Figure 3: Visualisation des étapes de découpe du maillage *Statue of Resting Goat*.

### 3.7 Écriture des données intermédiaires

**Quadriques simples.** Pour mesurer l'erreur d'un point par rapport à un ensemble de plans, on représente chaque plan  $t$  par sa normale homogène.

$$\bar{n}_t = (\hat{n}_x, \hat{n}_y, \hat{n}_z, d) \implies Q_t = \bar{n}_t \bar{n}_t^T \in \mathbb{R}^{4 \times 4}.$$

où  $Q_t$  représente la matrice quadrique d'un plan.

**Pourquoi des quadriques pour un maillage ?** Dans un maillage triangulaire, chaque triangle définit un plan, et la quadrique globale encode la forme locale de la surface en cumulant les quadriques de chaque triangle.

La matrice quadrique globale d'une cellule  $Q_G$  est définie comme l'accumulation des matrices des différentes quadriques

$$Q_G = \sum_{t \in G} Q_t = \begin{pmatrix} A & -b \\ -b^T & c \end{pmatrix},$$

**Qu'est-ce qu'elle permet de calculer ?** Elle permet de calculer l'erreur quadrique (ou la distance) d'un point par rapport à la surface encodée par la matrice.

Cette métrique aussi appelée QEM pour « Quadric Error Metric » est un standard pour la simplification de maillage lorsqu'on peut stocker les matrices localement.

Cette matrice permet aussi d'obtenir un point « optimal » qui minimisera cette erreur quadrique afin de préserver au mieux les détails de la surface passant dans cette cellule. Ce point sera détaillé dans la prochaine partie.

Après avoir calculé la boîte englobante, nous parcourons une seconde fois tous les triangles ( $v_1, v_2, v_3$ ) et pour chacun d'eux :

- On calcule la normale du plan formée par triangle avec la formule suivante :

$$\bar{n}_t = \begin{pmatrix} v_1 \times v_2 + v_2 \times v_3 + v_3 \times v_1 \\ -[v_1, v_2, v_3] \end{pmatrix}$$

- $x \times y$  représente un produit vectoriel
- $[v_1, v_2, v_3]$  représente un produit scalaire triple

En détaillant le calcul :

$$\tilde{\mathbf{n}}_t = \begin{pmatrix} y_{v_1}z_{v_2} - z_{v_1}y_{v_2} + y_{v_2}z_{v_3} - z_{v_2}y_{v_3} + y_{v_3}z_{v_1} - z_{v_3}y_{v_1} \\ z_{v_1}x_{v_2} - x_{v_1}z_{v_2} + z_{v_2}x_{v_3} - x_{v_2}z_{v_3} + z_{v_3}x_{v_1} - x_{v_3}z_{v_1} \\ x_{v_1}y_{v_2} - y_{v_1}x_{v_2} + x_{v_2}y_{v_3} - y_{v_2}x_{v_3} + x_{v_3}y_{v_1} - y_{v_3}x_{v_1} \\ -(x_{v_1}(y_{v_2}z_{v_3} - z_{v_2}y_{v_3}) - y_{v_1}(x_{v_2}z_{v_3} - z_{v_2}x_{v_3}) + z_{v_1}(x_{v_2}y_{v_3} - y_{v_2}x_{v_3})) \end{pmatrix}$$

- Pour chacun des trois sommets  $v_i$  de ce triangle :

- On détermine son indice de cellule  $G(v_i)$  via la méthode précédente.
  - On écrit dans le fichier `plane_equation.bin` le couple  $(G(v_i), \bar{n}_t)$ .
- On vérifie aussi  $G(v_1) \neq G(v_2) \neq G(v_3)$  (triangle non dégénéré) et si c'est le cas, on écrit dans le fichier `triangle_cluster.bin` le triplet  $(G(v_1), G(v_2), G(v_3))$ . Les triangles dégénérés (devenus des segments ou points) ne sont pas enregistrés.

Toutes ces écritures sont faites séquentiellement, triangle par triangle, ce qui minimise l'utilisation de la mémoire.

<b>Plane\_Equation.bin :</b> G(v <sub>1</sub> ) $\bar{n}_t$ G(v <sub>2</sub> ) $\bar{n}_t$ G(v <sub>3</sub> ) $\bar{n}_t$ ...	<b>TriangleCluster.bin :</b> G(v <sub>1</sub> ) G(v <sub>2</sub> ) G(v <sub>3</sub> ) G(v <sub>1</sub> ) G(v <sub>2</sub> ) G(v <sub>4</sub> ) G(v <sub>4</sub> ) G(v <sub>5</sub> ) G(v <sub>6</sub> ) ...
---	---

Figure 4: Fichiers intermédiaires

#### 4. Tri de `plane_equation.bin`

Une fois tous les triangles traités, on lance un tri externe (tri fusion) sur `plane_equation.bin` pour trier les entrées par cellule  $G$ .

Cette organisation en deux fichiers et ce tri permettent de limiter l'utilisation de la mémoire en dissociant la géométrie (plans passant dans la cellule) de la topologie. On pourra ainsi lors des étapes suivantes :

- (a) Traiter d'abord la géométrie en parcourant séquentiellement `plane_equation.bin` et calculer les représentants de chaque cellule sans charger la connectivité complète et sans avoir en mémoire les matrices quadriques des autres cellules.
- (b) Réintégrer ensuite la topologie en remplaçant, dans `triangle_cluster.bin`, les identifiants de cellule par les coordonnées des représentants calculés.

De plus, n'écrire que la normale du plan  $\bar{n}_t$  (quatre valeurs) plutôt que la matrice  $Q_t \in \mathbb{R}^{4 \times 4}$  de chaque triangle limite fortement l'espace disque et la mémoire utilisée. On peut seulement stocker  $\bar{n}_t$  puisque comme énoncé dans l'encadré,

$$Q_t = \bar{n}_t \bar{n}_t^T$$

On peut donc facilement calculer la matrice quadrique seulement à partir de  $\bar{n}_t$  et économiser de la mémoire et du stockage.

### 3.8 Calcul des représentants

**Quadriques simples.** Soit

$$Q_G = \sum_{t \in G} \bar{n}_t \bar{n}_t^T = \begin{pmatrix} A & -b \\ -b^T & c \end{pmatrix},$$

où

$$A = \sum_{t \in G} n_t n_t^T \in \mathbb{R}^{3 \times 3}, \quad b = \sum_{t \in G} d_t n_t \in \mathbb{R}^3,$$

avec pour chaque plan  $t$ ,  $n_t = (\hat{n}_x, \hat{n}_y, \hat{n}_z)^T$  sa normale et  $d_t$  son décalage au plan.

**Interprétation de  $A$  et  $b$  :**

- $A$  cumule tous les termes quadratiques  $n_t n_t^T$ , reflétant la “rigidité” directionnelle de la cellule.
- $b$  cumule les normales pondérées par le décalage  $d_t$ , fournissant le terme linéaire orientant la position optimale.

**Optimisation.** Pour trouver le sommet représentatif  $x$  qui minimise l’erreur quadratique, on résout simplement le système linéaire

$$Ax = b,$$

ce que réalise la fonction `findOptimalVertex`.

Le fichier `plane_equation.bin` étant trié par  $G$  lors de l’étape précédente, on peut simplement accumuler les  $\bar{n}_t \bar{n}_t^T$  séquentiellement pour former la quadrique et définir un sommet représentatif pour chaque cellule.

#### Algorithme pour la lecture et l’accumulation des équations de plans

Listing 3: Lecture + Accumulation des équations de plans et Écriture des représentants

```
Quadric Q;
int currentG = -1;
for each record (G, nbar):
    if G != currentG:
        if currentG >= 0:
            x = findOptimalVertex(Q, currentG, grid);
            write(representatives.bin, currentG, x);
            currentG = G;
        Q += nbar * nbar.transpose();
# traiter dernier groupe
x = findOptimalVertex(Q);
write(representatives.bin, currentG, x);
```

## FindOptimalVertex

Pour chaque cellule  $G$  de la grille, on calcule ses bornes  $\min, \max \in \mathbb{R}^3$ , qui serviront de limite pour la position optimale. À partir de la quadrique  $\mathbf{Q}$ , on construit les matrices  $A$  et le vecteur  $b$  pour résoudre le système

$$A \mathbf{x} = b.$$

Nous utilisons ensuite la bibliothèque Eigen pour vérifier si  $A$  est inversible (FullPivLU). Si ce n'est pas le cas, on effectue une décomposition LDLT et on vérifie l'absence de NaN. Si tout est valide, on constraint la solution  $\mathbf{x}$  aux intervalles  $[\min, \max]$ . En cas d'échec (matrice singulière ou NaN), on retourne le centre de la cellule :

$$\mathbf{x} = \frac{1}{2}(\min + \max).$$

Cette contrainte, qui borne le représentant, évite notamment, dans le cas où la quadrique d'une cellule ne contient qu'un seul plan (par exemple, une cellule avec un unique triangle), que le représentant ne puisse être trop éloigné de ses deux sommets voisins. Sans cette étape, la minimisation d'une quadrique plane admet une infinité de solutions, et le sommet pourrait être positionné n'importe où afin de minimiser l'erreur.

## Exemple de simplification avec les représentants non bornés / bornés

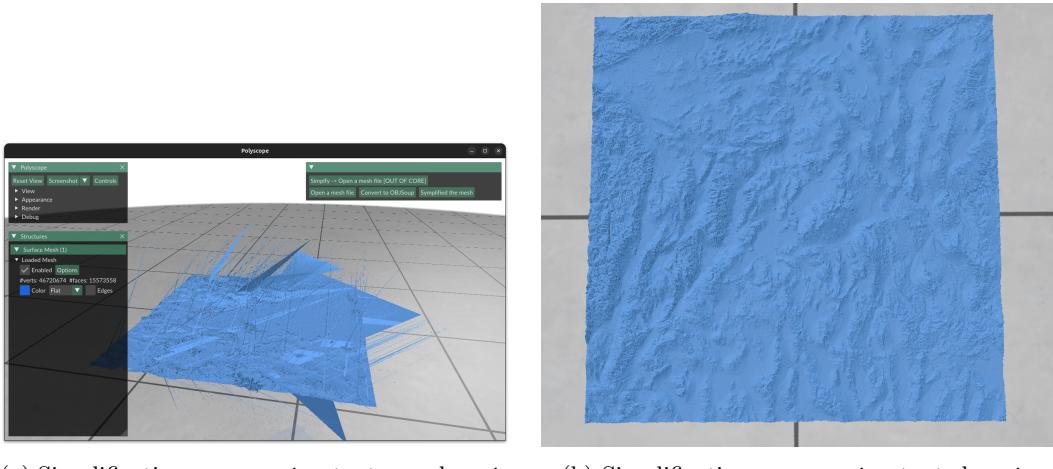


Figure 5: Exemple de points de représentants bornés / non bornées pour un même nombre de faces final

Listing 4: Fichier `representatives.bin` : (G, coordonnées du représentant)

```
1 | 1 0.0 0.0 0.0
2 | 2 0.3 0.35 0.26
...

```

### 3.9 Déréférencement des clusters

Maintenant que nous avons les coordonnées de tous les représentants, nous pouvons parcourir et remplacer dans le fichier `triangle_cluster.bin` les indices de cellules par leurs coordonnées. On effectue cette étape en 3 passes pour limiter l'utilisation de mémoire.

#### Séparation initiale

Listing 5: Fichier des représentants

1		1	0.0	0.0	0.0
2		2	0.3	0.35	0.26
3		3	0.64	0.68	0.64
4		4	0.8	0.7	0.75
5		5	1.0	1.0	1.0

Listing 6: Fichier des triangle clusters

1		1	5	3
2		2	1	4

#### Pass 1 : tri sur $G_1$ et remplacement

1. Tri externe du fichier des triangles clusters sur le premier indice de cellule  $G_1$  :
2. Lecture séquentielle simultanée du fichier des triangles cluster trié sur  $G_1$  et du fichier des représentants :
3. Les indices surlignés en jaune sont remplacés par les coordonnées des représentants.

1		1	5	3	->	0.0	0.0	0.0	2	3
2		2	1	4	->	0.3	0.35	0.26	5	1

#### Pass 2 : tri sur v2 et remplacement

1		0.3	0.35	0.26	1	4	->	0.3	0.35	0.26	0.0	0.0	0.0	4
2		0.0	0.0	0.0	5	3	->	0.0	0.0	0.0	1.0	1.0	1.0	3

#### Pass 3 : tri sur v3 et remplacement

1		0.0	0.0	0.0	1.0	1.0	1.0	3	->	0.0	0.0	0.0	1.0	1.0	1.0	3	0.64	0.68	0.64
2		0.3	0.35	0.26	0.0	0.0	0.0	4	->	0.3	0.35	0.26	0.0	0.0	0.0	4	0.8	0.7	0.75

Au terme de ces trois passes, chaque triangle est défini par ses trois coordonnées de représentant, dans un fichier binaire nommé `simplified.bin` qui représente notre maillage simplifié. Il est ensuite reconvertis en fichier .OBJ.

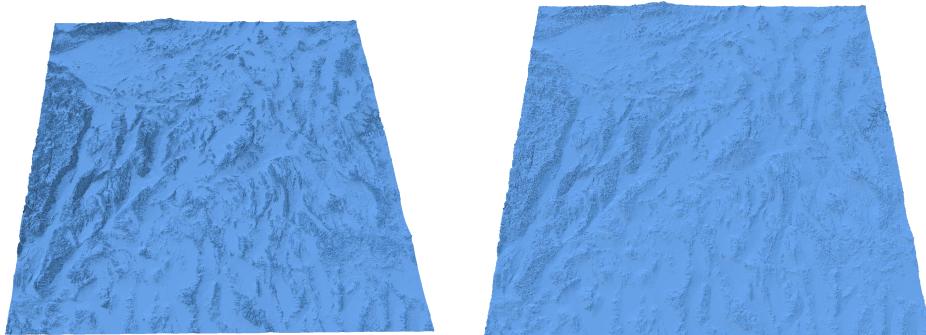
### 3.10 Résultats

#### 3.10.1 Comparaison qualitative des résultats



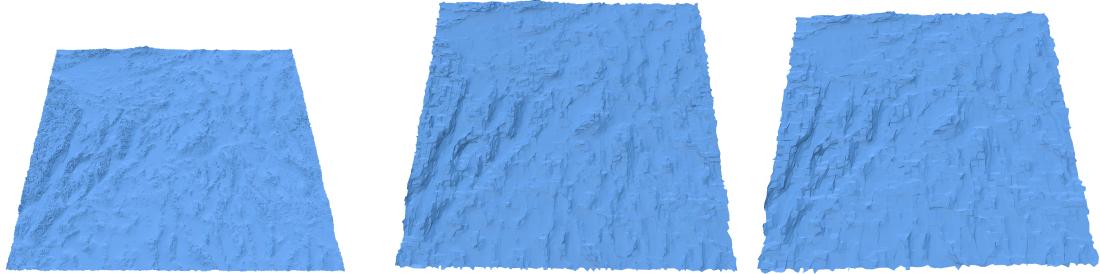
(a) Modèle original contenant 222 114 faces.  
(b) Modèle simplifié à 115 724 faces avec notre implémentation d'OoCSx.  
(c) Modèle simplifié à 7 580 faces avec notre implémentation d'OoCSx.

Figure 6: Comparaison des différentes simplifications du modèle *Statue of Resting Goat*.



(a) Modèle original à 112 000 000 faces      (b) Modèle simplifié à 20 000 000 faces

Figure 7: Comparaison des différentes simplifications d'un maillage de terrain.

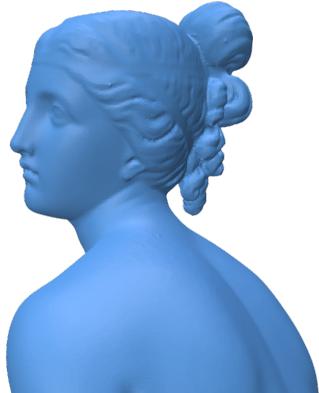


(a) Modèle simplifié à 6 600 000 faces      (b) Modèle simplifié à 191 000 faces      (c) Modèle simplifié à 85 000 faces

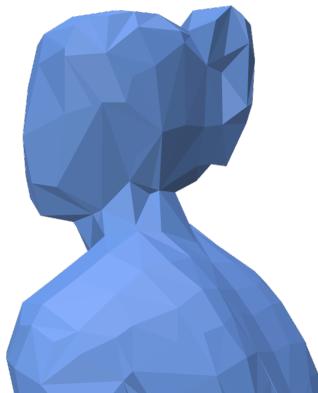
Figure 8: Comparaison des différentes simplifications d'un maillage de terrain.

## 4 Partie Adaptative

Dans la première version de la simplification *Out Of Core* que nous avons implémentée et présentée dans la partie précédente, nous utilisons une grille régulière pour découper l'espace. Bien que cette méthode permette de réduire significativement le nombre de faces du maillage, elle entraîne une perte importante de détails visuels. Cela se remarque par exemple dans la figure 9, où les traits du visage ainsi que les détails de la chevelure ne sont plus visibles après simplification.



(a) Modèle original contenant 533 439 faces.



(b) Modèle simplifié à 7 306 faces avec la version initiale de notre algorithme.

Figure 9: Comparaison entre la tête du modèle *Nymph Preparing for the Bath* avant et après simplification.

Dans cette partie, notre objectif est d'améliorer cet algorithme afin qu'il soit capable de repérer et de préserver les zones détaillées du modèle. Pour cela, nous nous appuyons sur l'approche proposée dans l'article *Efficient Adaptive Simplification of Massive Meshes* d'Eric Shaffer et Michael Garland [4]. Leur méthode se divise en trois phases principales :

- **Phase 1 : Quantification du maillage**

Le maillage d'entrée est d'abord projeté sur une grille uniforme pour regrouper les sommets proches dans des cellules régulières.

- **Phase 2 : Construction d'un arbre de partition adaptatif**

À partir des données issues de la grille, on construit un BSP Tree adaptatif (*Binary Space Partitioning Tree*), qui organise hiérarchiquement les sommets selon leur répartition dans l'espace.

- **Phase 3 : Regroupement des sommets et simplification**

Enfin, les sommets sont regroupés en fonction des feuilles du BSP Tree, puis remplacés par des représentants calculés pour générer une approximation adaptative du maillage.

### 4.1 Phase 1 : Quantification du maillage

Cette première phase consiste à partitionner l'espace englobant le maillage en une grille régulière de cellules, afin de regrouper spatialement les sommets proches. Pour chaque cellule, nous accumulons deux types de quadriques complémentaires : les quadriques simples et les quadriques duales (voir l'encadré 6.2 pour la définition).

Les *quadriques duales* servent à décrire la répartition spatiale d'un nuage de points. Contrairement à la quadrique simple, qui mesure l'erreur d'un point par rapport à un ensemble de plans, la quadrique duale mesure l'erreur d'un **plan** par rapport à un ensemble de **points**.

Pour chaque sommet  $v \in \mathbb{R}^3$ , on définit une quadrique duale sous la forme :

$$D = v v^T, \quad e = v, \quad f = 1.$$

Les quadriques sont **agrégées** par simple sommation des triplets  $(D, e, f)$  sur tous les sommets contenus dans une cellule. Cette agrégation permet de représenter globalement la distribution spatiale des points d'une cellule par une quadrique duale unique. La matrice obtenue, calculée par :

$$Z = D - \frac{e e^T}{f}$$

est équivalente à une **matrice de covariance centrée**, qui mesure la dispersion du nuage de points autour de sa moyenne.

Rappel : la matrice de covariance est une matrice symétrique dont les vecteurs propres donnent les directions principales (axes) de variation des points, et les valeurs propres quantifient l'étalement (variance) le long de ces directions.

Les deux types de quadriques sont utilisées conjointement car elles se complètent : la quadrique simple permet de minimiser l'erreur géométrique d'un point par rapport aux plans locaux, tandis que la quadrique duale capture la distribution spatiale globale des points dans la cellule, facilitant ainsi une analyse plus structurelle du maillage.

**1 - Délimiter l'espace du modèle** Nous commençons par analyser les coordonnées extrêmes des sommets afin de calculer la *bounding box*, c'est-à-dire le plus petit parallélépipède rectangle englobant le modèle. Cette étape ne nécessite qu'un parcours linéaire du fichier contenant le maillage.

**2 - Créer une grille régulière** À partir de la bounding box et d'un paramètre de résolution fourni par l'utilisateur, on peut définir la grille. Elle subdivise l'espace en cellules de taille constante, indépendamment de la densité du maillage. Chaque cellule correspond à un bloc de l'espace dans lequel les données vont être accumulées.

**3 - Projeter les sommets et accumuler les quadriques** Nous parcourons ensuite le maillage une seconde fois. Pour chaque sommet  $v$ , nous identifions la cellule dans laquelle il tombe, et nous y ajoutons la *quadrique simple* associée au plan de la face ainsi que la *quadrique duale*. Par ailleurs, nous enregistrons une table de correspondance entre chaque sommet d'origine et sa cellule, afin de pouvoir reconstruire les connexions entre triangles.

**4 - Calculer un point représentatif par cellule** Une fois les quadriques agrégées, chaque cellule possède suffisamment d'information pour définir un point représentatif. Ce point est choisi comme celui qui minimise l'erreur quadrique simple associée à la cellule. Il jouera le rôle de substitut pour tous les sommets qu'elle contient.

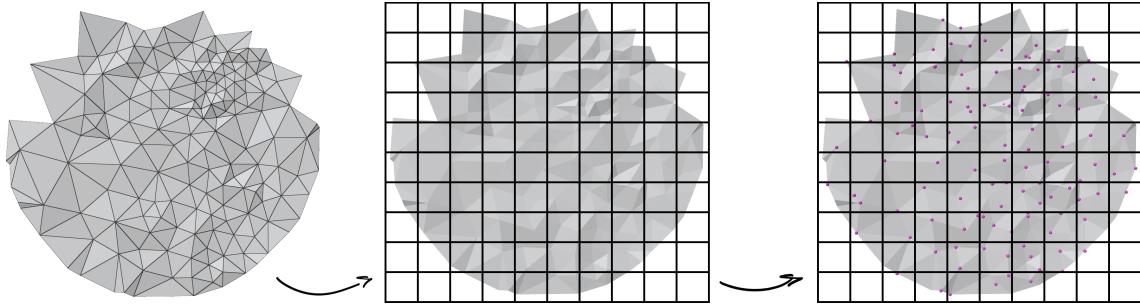


Figure 10: Illustration de la quantification du maillage : (gauche) maillage initial, (centre) projection sur une grille régulière, (droite) calcul des points représentatifs dans chaque cellule.

## 4.2 Phase 2 : Construction d'un arbre de partition adaptatif

Dans cette deuxième phase, nous utilisons les quadriques agrégées pour construire un BSP Tree, une structure binaire de découpage de l'espace. Celui-ci va nous servir à regrouper les cellules de la grille selon leur importance géométrique, en concentrant la finesse de découpe là où l'erreur de représentation est la plus élevée.

### 4.2.1 Initialisation du nœud racine et file de priorité

Nous commençons par regrouper l'ensemble des cellules occupées dans un unique nœud racine. Pour ce nœud, nous cumulons la quadrique simple et la quadrique duale associées à chaque cellule. Nous créons aussi une file de priorité où chaque élément est un noeud de l'arbre associé à son erreur quadrique simple. La clé de tri est précisément cette erreur, de sorte que le noeud ayant la plus grande erreur se trouve toujours en tête.

### 4.2.2 Boucle de subdivision adaptive

Tant que le nombre de feuilles (nœuds sans enfant) n'atteint pas l'objectif fixé par l'utilisateur :

1. on retire de la file le nœud présentant la plus grande erreur ;
2. on effectue sa division en deux enfants selon la procédure décrite ci-après ;
3. on insère ces deux nouveaux nœuds dans la file avec leurs erreurs recalculées.

**Division d'un nœud** Pour diviser un noeud, nous utilisons sa quadrique duale agrégée  $\{D, e, f\}$  pour déterminer un plan de coupe optimisé. Nous commençons par construire la matrice de covariance

$$Z = D - \frac{e e^T}{f}$$

qui caractérise la dispersion spatiale des points représentés par le nœud. Nous résolvons ensuite l'équation aux valeurs propres  $Zv = \lambda v$ , obtenant trois valeurs propres ordonnées  $\lambda_1 \geq \lambda_2 \geq \lambda_3$  et leurs vecteurs propres correspondants  $v_1, v_2, v_3$ . Ces résultats quantifient respectivement l'étalement maximal et minimal du nuage de points.

Si le rapport  $\lambda_1/\lambda_3 \geq 2$ , l'étalement est cohérent (similaire à une surface plane) ; nous choisissons comme normale du plan de coupe la direction de plus forte variation  $v_1$ . Sinon, l'étalement est considéré comme incohérent et la normale est prise selon la direction de plus faible variation  $v_3$ .

Nous plaçons ce plan de sorte qu'il passe par le **centre de gravité** des points du nœud, donné par  $\bar{p} = e/f$ . Cette découpe génère deux nouveaux nœuds, qui sont à leur tour insérés dans la file de priorité.

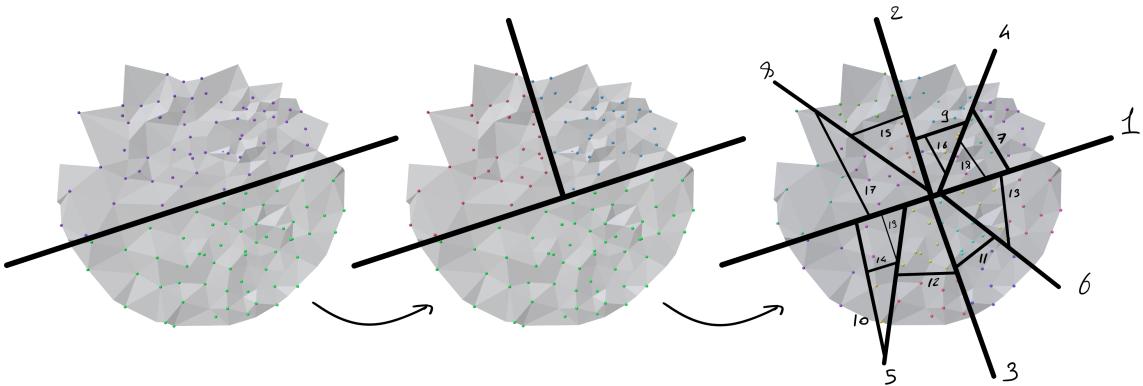


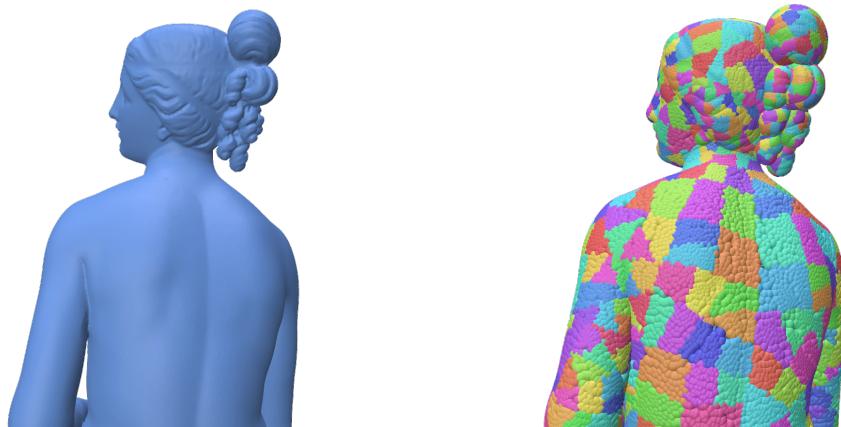
Figure 11: Construction du BSP Tree : (gauche) première division de l'arbre, (centre) subdivision suivante, ... (droite) partition complète avec 20 feuilles.

#### 4.2.3 Résultats : Visualisation des feuilles adaptatives

Pour montrer la distribution des cellules au sein des feuilles du BSP Tree, la figure 12 présente :

- en sous-figure à gauche, le modèle d'origine ;
- en sous-figure à droite, le nuage de points des feuilles du BSP Tree.

Dans cette visualisation, chaque point coloré représente un point représentant de cellule, et chaque couleur regroupe les points associés à une même feuille de l'arbre.



(a) Modèle original de la statue.

(b) Nuage de points coloré : chaque couleur représente une feuille du BSP Tree.

Figure 12: Visualisation de la subdivision adaptative : comparaison entre le modèle initial (a) et la répartition spatiale des feuilles (b).

On observe que certaines régions, comme le dos de la statue, sont couvertes par des feuilles couvrant une large surface et contenant un grand nombre de cellules (jusqu'à 150 environ). Cette faible densité de subdivision traduit une géométrie locale relativement plane et peu détaillée, qui peut être représentée de manière satisfaisante avec peu de points.

À l'inverse, dans des zones plus complexes comme les cheveux, les feuilles sont plus petites et contiennent moins de cellules (entre 30 et 50 en moyenne). Cela montre que l'algorithme adapte le niveau de subdivision à la richesse géométrique locale, en créant plus de feuilles là où les détails sont nombreux. Ce comportement correspond bien à notre objectif, il permet de mieux préserver les formes fines et les reliefs du modèle d'origine.

### 4.3 Phase 3 : Regroupement des sommets et reconstruction du maillage

Cette phase finale consiste à générer un maillage simplifié en réaffectant les sommets d'origine aux feuilles du BSP Tree, puis en remplaçant chaque feuille par un point représentant unique.

#### Première passe : association des sommets aux feuilles et accumulation des quadriques

Nous parcourons dans un premier temps l'ensemble des faces du maillage original. Pour chaque sommet d'une face, nous identifions la feuille du BSP Tree à laquelle il appartient en traversant l'arbre depuis la racine. Une fois les trois feuilles associées aux sommets d'une face déterminées, nous attribuons la quadrique simple définie par cette face à chacune de ces feuilles.

**Calcul des représentants de feuilles** Une fois les quadriques accumulées, nous calculons, pour chaque feuille, un point représentant unique. Celui-ci est défini comme le point minimisant l'erreur quadrique simple agrégée dans la feuille. Il remplacera tous les sommets initiaux associés à cette feuille dans le maillage simplifié.

**Deuxième passe : reconstruction des faces** Dans une seconde itération sur les faces du maillage initial, nous remplaçons les sommets par le représentant de leurs feuilles. Si les trois sommets d'une face appartiennent à des feuilles distinctes, nous construisons une nouvelle face entre les trois représentants. En revanche, si deux sommets ou plus appartiennent à la même feuille, la face devient dégénérée et est donc supprimée du maillage simplifié.

#### 4.3.1 Comparaison qualitative des résultats

Comme le montre la figure 13, la simplification adaptative conserve bien mieux les détails visuels dans les régions à forte complexité géométrique. Contrairement à la version basée sur une grille régulière, les traits du visage, comme le nez ou les contours des joues, restent bien définis. La chevelure, qui apparaissait comme un bloc uniforme dans la première simplification (figure 13b), retrouve ici une structure bien plus détaillée (figure 13c).

Cependant, cette meilleure restitution des zones complexes se fait parfois au détriment des régions plus simples. Par exemple, sur le dos de la statue, certains volumes comme les omoplates deviennent moins visibles, voire disparaissent complètement.

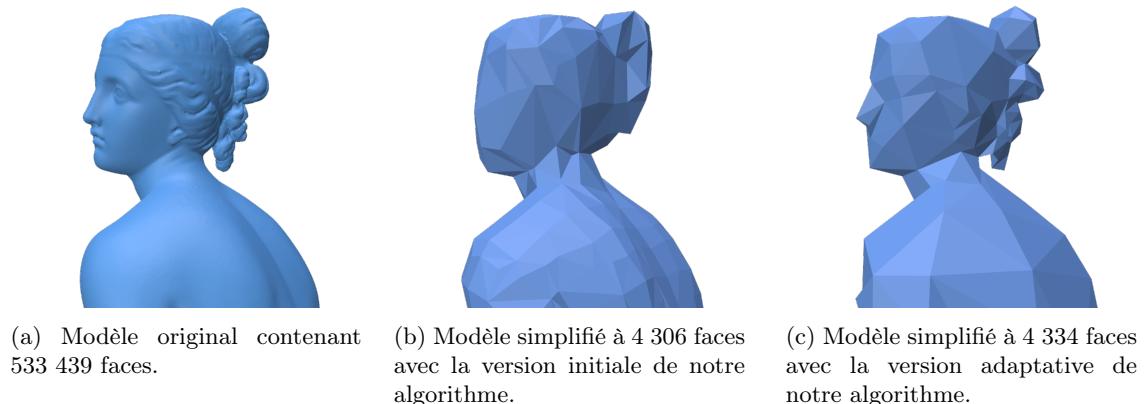


Figure 13: Comparaison des différentes simplifications du modèle *Nymph Preparing for the Bath*.

## 5 Partie Streaming

Les deux algorithmes que nous avons étudiés jusqu'à maintenant reposent principalement sur le principe d'écrire les résultats intermédiaires sur disque pour pouvoir traiter l'ensemble du maillage. Ce processus est long et nécessite plusieurs passages dans le maillage complet. Ce dernier article que nous avons étudié a pour but de pallier ces deux problèmes en proposant une méthode qui permet de réaliser la simplification en un seul passage et sans nécessiter d'écrire nos calculs dans des fichiers.

L'approche proposée par Jianhua Wu et Leif Kobbelt [5] peut se résumer en trois phases principales.

- **Phase 1 : La lecture** Les triangles sont chargés par petits lots, selon la limite mémoire fixée par l'utilisateur. (Triangles rouges dans la figure 14)
- **Phase 2 : La décimation** Les triangles en mémoire sont décimés pour atteindre le pourcentage de réduction souhaité. (Les triangles en rouge et en vert dans la figure 14 ont été simplifiés)
- **Phase 3 : L'écriture** Lorsqu'un triangle est simplifié, il peut être écrit en sortie pour libérer de la place. (Triangles verts dans la figure 14)

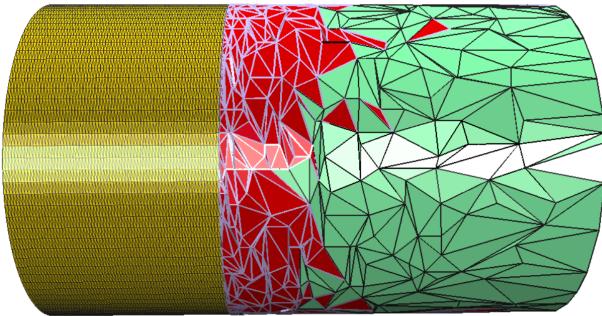


Figure 14: Illustration du traitement en flux. **Jaune** : non traité, **Rouge** : en cours de simplification, **Vert** : simplifié.

### 5.1 Bordures de notre tampon

Avant d'expliquer comment nous allons réaliser les trois phases de la décimation en flux, nous allons vous expliquer le principal point sur lequel repose toute la méthode, qui est la gestion des bordures de notre tampon de triangles en mémoire. En effet, comme nous réalisons un traitement au fur et à mesure, il est très important de connaître les sommets qui sont disponibles pour réaliser une décimation. Et pour les connaître, on peut séparer les sommets en trois catégories (figure 15) :

- Les sommets qui n'ont pas tous leurs triangles lus. Ces sommets représentent la frontière (nommée  $P_{AB}$  sur notre schéma) entre le maillage en mémoire (B) et le maillage hors mémoire (A) qui reste à lire.
- Les sommets dont au moins un triangle a été écrit, qui représentent la frontière (nommée  $P_{BC}$  sur notre schéma) entre le maillage en mémoire (B) et le maillage écrit (C).
- Enfin, les sommets entièrement connus (B) (tous leurs triangles ont été lus et sont actuellement présent en mémoire) mais non encore impliqués dans une écriture, sont considérés comme stables : ils peuvent être utilisés pour la décimation sans risque d'incohérence

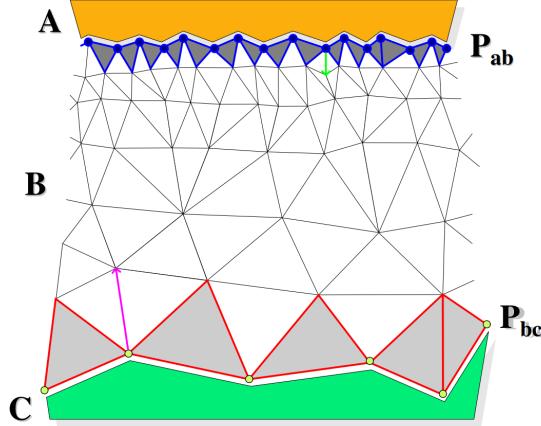


Figure 15: Illustration des trois parties — partie A : non lues, partie B : actuellement en mémoire, partie C : déjà écrites

### 5.1.1 Structure de demi-arêtes

Pour pouvoir catégoriser correctement les sommets, les auteurs de l'article recommandent d'utiliser une structure de demi-arêtes, un concept fondamental en modélisation de maillages. Cette structure permet de représenter la topologie d'un maillage 2-manifold, (c'est-à-dire que pour chaque arête, on a seulement deux faces associées) et d'avoir un accès constant et cohérent à la connectivité locale sans stocker explicitement toutes les arêtes et les faces. Elle repose sur les principes suivants :

- Chaque **arête** est dédoublée en deux **demi-arêtes** opposées, orientées l'une vers l'autre.
- Une **demi-arête** stocke :
  - son *origine* (sommets de départ),
  - un pointeur vers la demi-arête *suivante* dans la même face,
  - un pointeur vers sa *demi-arête opposée*,
  - un lien vers la *face* à laquelle elle appartient, si elle possède une face (demi-arêtes bleues sur la figure 16), sinon un pointeur vers null pour une demi-arête en bordure (demi-arêtes rouges sur la figure 16)
- Chaque **sommet** dispose d'une référence vers l'une de ses demi-arêtes sortantes, facilitant le parcours de ses voisins.
- Chaque **face** conserve une demi-arête de référence, permettant de récupérer toutes ses arêtes par chaînage des demi-arêtes *suivantes*.

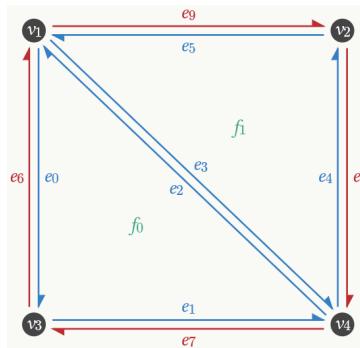


Figure 16: Schéma qui nous montre l'utilisation des demi-arêtes

Plutôt que de réimplémenter toute la structure classique de demi-arêtes, assez lourde et complexe, nous avons défini un ensemble de tables et listes qui jouent un rôle similaire. Pour mieux comprendre le but de chaque tables et listes, l'explication de la structure va se faire en même temps qu'on explique les trois phases principales de l'article (la structure est disponible en annexe figure 26) En combinant ces tables, nous simulons la double occurrence des demi-arêtes (première et seconde rencontre) et la notion de bordure d'écriture, tout en restant sur une structure de données plus simple.

## 5.2 Phase 1 : La lecture

L'article définit la lecture comme un appel simple à la fonction **READ(k)**, qui lit les  $k$  prochains triangles ayant été triés au préalable selon un axe, afin de mieux gérer le système de frontières. Cette fonction met également à jour la structure représentant le maillage actuellement en mémoire. À chaque lecture d'un nouveau triangle, on réalise les étapes suivantes :

- Ajouter ce nouveau triangle en l'associant à un id unique et calculer sa quadrique.
- Comme on lit une soupe de triangles, il faut associer chacune des coordonnées de sommets à un id unique grâce à une table. Cela permet aussi de manipuler uniquement des ids au lieu de coordonnées, ce qui serait moins pratique.
- Une fois qu'on a nos ids pour chaque sommet :
  - Mise à jour des listes de voisins et ajout, si nécessaire, du sommet dans la liste des sommets hors bordure, en respectant les cas expliqués plus loin.
  - Ajout de la quadrique du triangle à chacun des trois sommets.
  - Ajout de l'id unique du triangle dans la liste des triangles des sommets.

C'est à partir des **listes de voisins** que la topologie est suivie, remplaçant la structure de demi-arêtes. Pour le moment, on se concentre ici sur deux d'entre elles :

- **Liste de première rencontre** : sert à stocker la première fois que l'on rencontre un voisin de notre sommet.
- **Liste de seconde rencontre** : sert à stocker la deuxième fois que l'on rencontre le même voisin. On déplace le sommet depuis la première liste vers la deuxième. Cela nous permet donc de définir qu'entre ces deux sommets, on a une arête interne.

Lors de la lecture, les deux listes doivent être correctement mises à jour, en maintenant les listes dans un ordre croissant des ids, et en respectant les trois cas qui peuvent se présenter :

- **Cas 1** : Pour la lecture d'un triangle qui n'avait aucun voisin, ajout des nouveaux voisins dans les listes de première rencontre (figure 17a).
- **Cas 2** : Lorsque le triangle possède une ou deux arêtes en commun, déplacement des voisins concernés dans les listes de seconde rencontre (figure 17b).
- **Cas 3** : Lecture du dernier triangle associé à un sommet, modification des deux listes ; si la première liste de sommets est vide après ces modifications, ajout du sommet dans la liste des sommets hors bordure (figure 17c).

## 5.3 Phase 2 : La décimation

La fonction **DECIMATE(k)** effectue  $k$  contractions d'arêtes successives. Le choix de l'arête qui va être contractée s'effectue entre 5 ou 15 arêtes choisies aléatoirement parmi les sommets présents hors bordure, et on gardera celle qui minimisera le calcul de l'erreur quadratique. Grâce à ce critère de sélection, on obtient un maillage de bonne qualité.

Pour nous aider à mieux visualiser ce qui se passe lors d'une contraction d'arête, analysons la figure 18 :

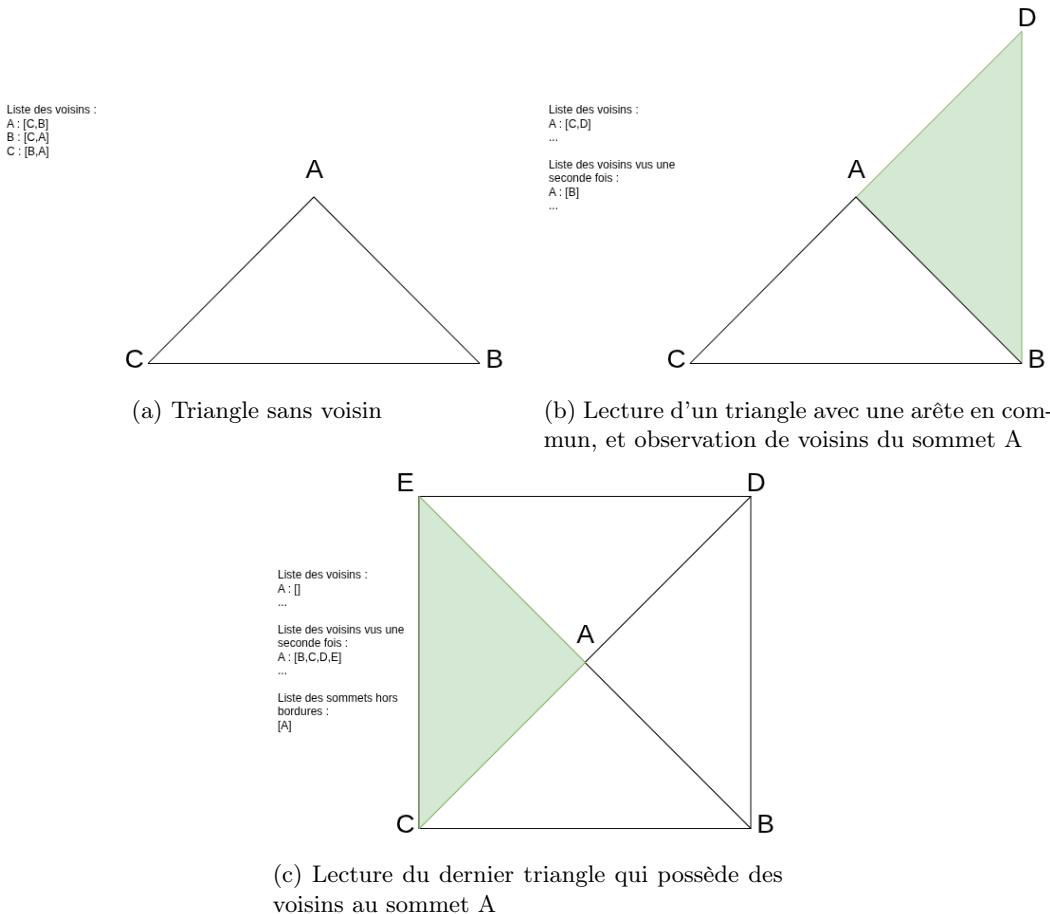


Figure 17: Trois principaux cas de lecture d'un triangle

On peut remarquer facilement sur le schéma que les voisins des deux sommets ( $v_s$  et  $v_t$ ) de l'arête vont se retrouver voisins du sommet résultant de la contraction ( $v_n$ ), et que, par contre, les deux triangles qui étaient liés à l'arête vont être supprimés. C'est avec cela à l'esprit que l'on va procéder ainsi pour chaque contraction d'arêtes :

- Récupération du plus petit indice des deux sommets, qui représentera le **nouvel indice**, et de la **nouvelle coordonnée** issue du calcul de la quadrique.
- Fusion des deux listes de sommets voisins de  $(v_s, v_t)$ .
- Ensuite, pour chaque sommet voisin de cette liste fusionnée :
  - Remplacement de l'ancien indice par le **nouvel indice** dans leur liste de seconde rencontre, tout en maintenant le tri de la liste.
  - Pour les sommets qui étaient en commun, on va supprimer les triangles qui sont dégénérés suite à la contraction.
- Pour chaque triangle (privé des triangles dégénérés), on va mettre à jour la coordonnée du sommet avec la **nouvelle coordonnée**.
- Mise à jour de la quadrique associée à ce nouveau sommet par l'addition des deux quadriques, en accord avec l'article  $(Q_{v_s} + Q_{v_t})$ .
- Et pour finir, on va marquer ce sommet comme simplifié pour la suite de l'algorithme.

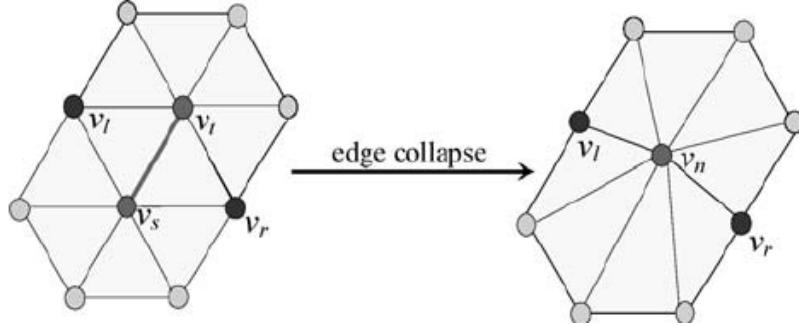


Figure 18: Illustration d'une contraction d'arête

Petite réflexion sur la complexité : pour éviter une complexité trop élevée lors des recherches des sommets voisins ou des triangles en commun, on maintient les listes triées, ce qui permet de trouver, en maximum  $O(n)$ , les indices recherchés, où  $n$  est la taille de la plus petite liste des voisins.

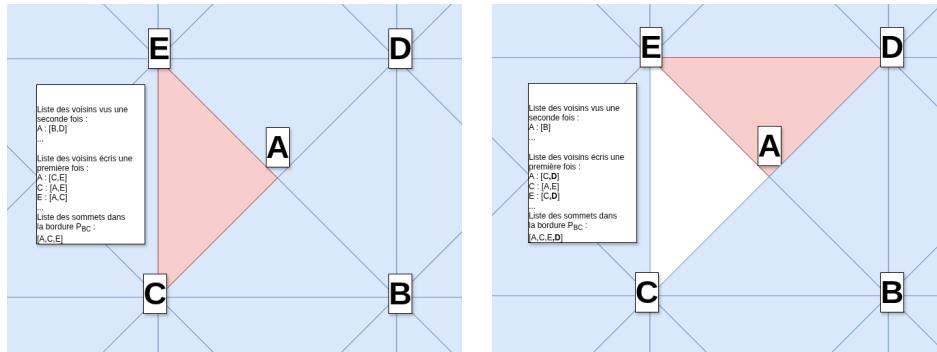
#### 5.4 Phase 3 : L'écriture

La fonction **WRITE(k)** effectue  $k$  écritures des triangles qui maximisent l'erreur quadratique associée au triangle. Pour l'écriture, la sélection va se faire, soit depuis les sommets qui sont sur la bordure d'écriture ( $P_{BC}$ ), soit parmi n'importe quels sommets qui ont été marqués comme simplifiés. On va faire également une sélection aléatoire parmi plusieurs triangles potentiels, et on gardera celui dont l'erreur quadratique associée est la plus élevée. Pour trouver cette erreur, on additionne les quadrics des trois sommets du triangle, et on calcule l'erreur par rapport au centre du triangle.

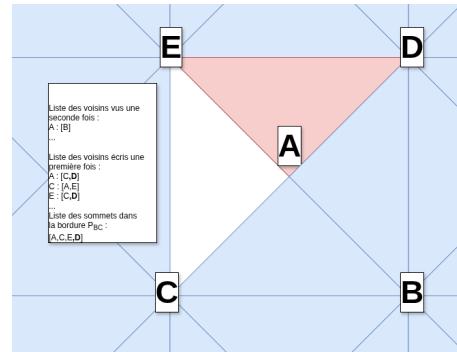
Tout comme lors de la lecture, lors de l'écriture d'un triangle, on va vouloir garder une trace des sommets qui sont sur la bordure d'écriture ( $P_{BC}$ ) pour deux raisons. La première c'est qu'on ne peut plus simplifier un sommet qui est sur cette bordure, et la deuxième c'est pour permettre une sélection prioritaire sur les triangles qui ont minimum un sommet présent sur la bordure d'écriture. Ce choix prioritaire pour les triangles sur la bordure d'écriture, permet d'éviter de réduire trop rapidement les sommets simplifiables. Cela s'explique simplement par le fait que si nous choisissons un sommet hors bordure, nous réduirons, de trois en trois, le nombre de sommets simplifiables. Contre une réduction de maximum de deux sommets, pour un triangle qui aurait seulement un sommet présent sur la bordure, et dans le meilleur cas, d'un seul sommet.

Pour faire donc cela, on a rajouté, la liste qui contient les sommets présents en bordure BC, et une troisième table qui sert à stocker les voisins de chaque sommets dont on a écrits le premier triangle qui les reliait avant. Ces deux ajouts nous permet de sélectionner un sommet en bordure BC, et choisir en priorité un de ses voisins dont on a écrits un triangle. C'est pour les différentes raisons citées au dessus qu'on doit également veiller à bien mettre à jour les différentes listes, en respectant les différents cas :

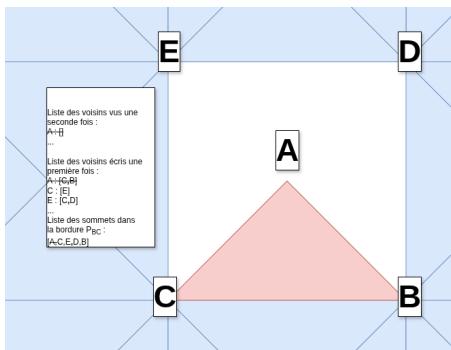
- **Cas 1 :** Au cas où, il n'y a pas de meilleur choix, écriture d'un triangle dont le sommet était hors bordure,(au milieu des triangles présent dans le maillage): déplacement des sommets dans la liste des voisins écrits une première fois, et ajout des sommets correspondant dans la bordure  $P_{BC}$  (figure 19a).
- **Cas 2 :** Écriture d'un triangle qui possède deux sommets ayant déjà été écrit une première fois (présent en bordure BC): suppression du lien entre les deux sommets dans la liste (les deux triangles qui liaient ces sommets ont été écrits). Mise à jour des listes de chaque sommets pour correspondre à la nouvelle topologie. (figure 19b).
- **Cas 3 :** Écriture du dernier triangle associé à un sommet : suppression des listes associées à ce sommet et suppression du sommet de la bordure. (figure 19c).



(a) Écriture du triangle rouge, qui était entouré de triangle en mémoire (en bleu)



(b) Écriture du triangle rouge qui possérait deux sommets ayant déjà été écrit une première fois (E et A), modification des listes: suppression des liens de voisinages entre A et E, et ajout de D en tant que sommets sur une bordure



(c) Écriture du dernier triangle rouge associé au sommet A, mise à jour des listes des voisins et suppression des références à ce sommet A

Figure 19: Trois principaux cas d'écriture d'un triangle

## 5.5 Explication générale de la méthode

Pour réaliser correctement l'alternance entre les trois étapes expliquées précédemment, nous nous sommes basés sur le squelette de l'algorithme fourni dans l'article. On va maintenant l'illustrer avec un exemple concret.

Supposons que l'on souhaite simplifier 1000 triangles, avec un tampon mémoire de taille maximale  $N_{\max} = 200$ , et un taux de décimation fixé à  $p = 0.5$  (donc 50% des triangles doivent être supprimés).

### Initialisation du tampon

La première étape sert à initialiser le tampon sans faire d'écriture, uniquement avec des phases de lecture et de décimation. Pour cela, on doit d'abord choisir un entier  $n$  qui respecte la condition suivante :

$$\frac{1}{n} \geq p > \frac{1}{n+1}$$

Dans notre cas, avec  $p = 0.5$ , on obtient  $n = 2$  car  $\frac{1}{2} \geq 0.5 > \frac{1}{3}$ .

On exécute ensuite les instructions suivantes :

```
READ (N_max / 2)           // Lecture de 100 triangles
repeat (n - 1) times :
    READ (N_max / 2)       // Lecture de 100 triangles
```

```
DECIMATE (N_max / 4) // Decimation de 25 arêtes donc 50 triangles
```

À la fin de cette étape, on a environ lu 300 triangles de notre maillage et il en reste 150 triangles dans le tampon, ce qui correspond bien à la résolution  $1/n = 0.5$  demandée.

### Boucle principale

On lance ensuite la boucle principale qui s'exécute tant qu'il reste des triangles à lire dans le fichier d'entrée.

```
while input not empty :
    READ (N_max / 2)           // Lecture de 100 triangles
    DECIMATE ((1 - p) * N_max / 4) // Decimation de 25 triangles
    WRITE (p * N_max / 2)        // Ecriture de 50 triangles
```

Cette boucle maintient à chaque fois le tampon autour de 150 triangles : on en lit 100, on en décime 25, et on en écrit 50.

### Finalisation

Une fois que tous les triangles ont été lus, il reste encore une dernière phase pour vider correctement le tampon. On applique :

```
DECIMATE ((1 - n * p) * N_max / 4) // Ici : (1 - 1) * 50 = 0
WRITE ((n * p * N_max) / 2)       // Ecriture des 100 derniers triangles
```

Cette étape permet d'écrire les derniers triangles restants dans le tampon.

Ce schéma garantit d'obtenir la bonne résolution demandée par l'utilisateur, sans jamais dépasser  $N_{max}$  en mémoire.

## 5.6 Problème rencontré

Nous n'avons malheureusement pas réussi à terminer l'implémentation complète de cet article. L'article était plus complexe et plus long à comprendre en comparaison aux autres articles. Nous avons de plus pris beaucoup de temps à théoriser la structure que nous possédons actuellement. Et malgré la théorie simplifiée expliquée dans ce rapport, en pratique, maintenir la structure s'est révélé plus complexe qu'il n'y paraissait. Notamment dans la mise à jour des listes de voisins de chaque arêtes, en rédigeant le rapport, nous avons remarqué que certains aspects de l'écriture ont potentiellement mal était pris en compte dans notre code (exemple: écriture d'un triangle qui n'aurait qu'un seul sommet sur la bordure d'écriture  $P_{BC}$ ), ceci pourrait provoquer des fuites de mémoire, avec par exemple des triangles qui ont été écrit mais dont la liste des voisins de leur sommets ont été mal mise à jour.

Le dernier vrai problème rencontré concerne une mauvaise mise à jour sur la liste des voisins déjà écrits et la liste de voisins déjà vue, un sommet se retrouve avec un sommet voisins dans chacune des deux listes, ce qui provoque des erreurs lors de recherche des voisins qui sont en commun.

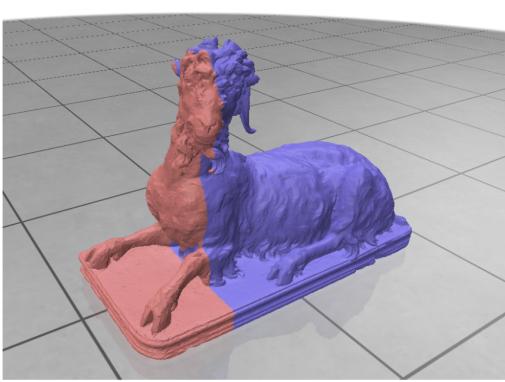
Si nous avions eu plus de temps, nous aurions pu corriger ces quelques erreurs qui provoque une topologie corrompue, et principalement des fuites de mémoires.

Ou peut-être explorer **une autre structure** discutée avec notre encadrant, qui consiste en une structure d'arêtes à la place des trois tables pour les voisins. La structure serait plus proche des demi-arêtes et serait composée d'un sommet source, d'un sommet final et des faces associées.

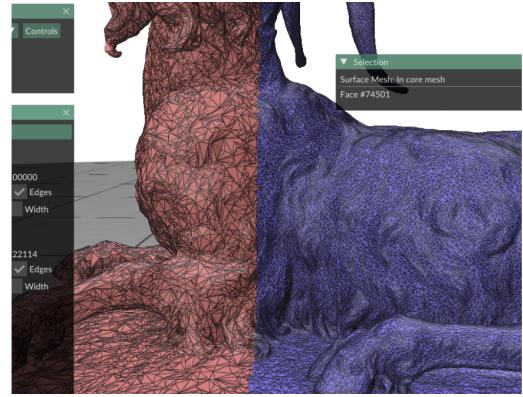
- Une table de hachage permettrait d'associer les sommets à la même structure d'arêtes (elle devrait donner le même sommet source et le même sommet final, peu importe si on donne d'abord un sommet A puis B, ou B puis A).
- Et quand on aura associé deux faces pour une même arête, cela voudra dire que l'arête est interne. (Si une seule face, l'arête est externe).

Nous n'avons pas exploré cette structure plus en détail. Mais avec les connaissances que nous avons acquises grâce à l'exploration de la structure actuelle, cette approche nous aurait sûrement permis de corriger les quelques erreurs qu'il nous reste.

Malgré les erreurs de fuites de mémoires nous avons obtenus ces résultats, sur deux maillages donnés (figure 21 et 20).

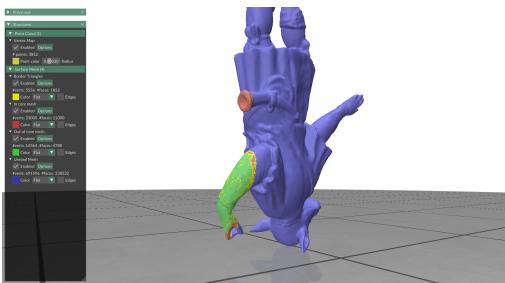


(a) Simplification en cours avec la partie rouge pour les triangles simplifiés et la partie violette non simplifiés

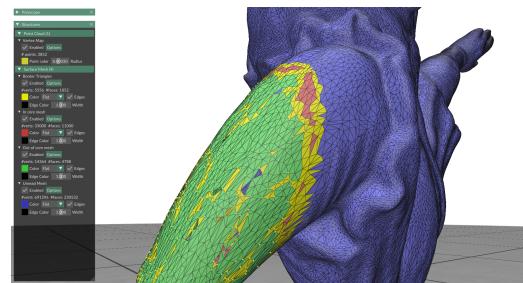


(b) Vue zoomée avec affichage des arêtes

Figure 20: Affichage d'un maillage traités en flux avec de la simplification par contraction d'arêtes sans écriture



(a) Simplification du maillage en cours avec les parties rouges pour les triangles simplifiées, la partie verte pour les triangles écrivent dans le fichier de sortie, la partie violette pour les triangles non simplifiées et les parties jaunes pour les triangles sur une des bordures



(b) Vue zoomée sur le bras avec affichage des arêtes (les trous représentent les triangles qui viennent d'être écrits)

Figure 21: Affichage du début du traitement en flux du maillage *Actaeon* composé de 236 532 faces

## 6 Résultats et comparaisons

Nous évaluons ici la qualité et la fidélité géométrique de notre implémentation OoCSx par rapport à la méthode Adaptative, en utilisant comme référence un scan 3D du retable de Sainte-Anne (Anna Retable).

### 6.1 Comparaison visuelle de la simplification

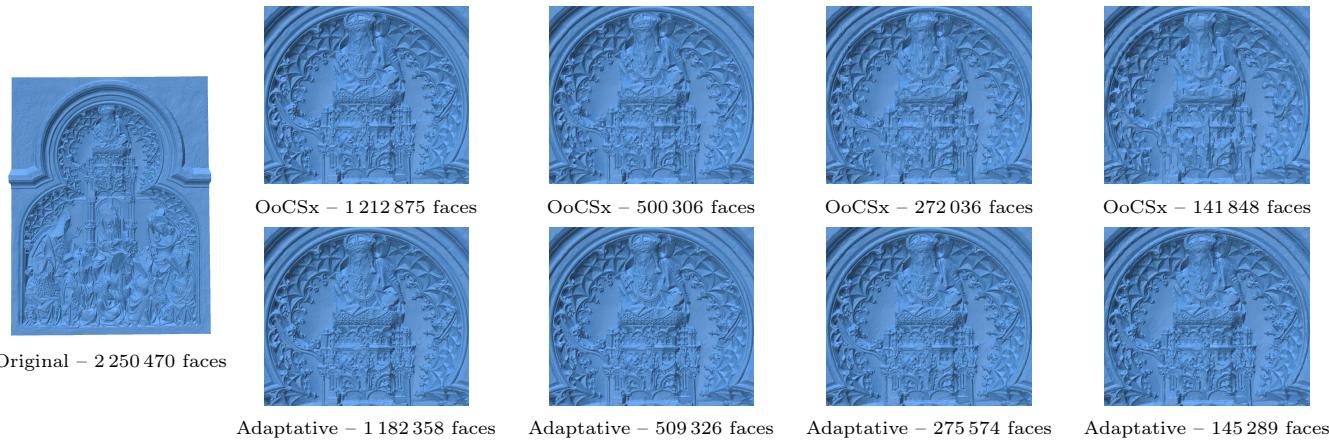


Figure 22: Comparaison visuelle des maillages simplifiés par OoCSx (en haut) et Adaptative (en bas) à différents niveaux de réduction.

### 6.2 Distance d’Hausdorff

Pour évaluer quantitativement la qualité géométrique des maillages simplifiés, nous mesurons la distance de Hausdorff entre le modèle simplifié et le modèle de référence. La Table 1 compare ces distances pour les méthodes OoCSx et Adaptative à différents niveaux de simplification.

#### 6.2.1 Définition de la distance de Hausdorff

Soient  $A$  et  $B$  deux ensembles de points dans  $\mathbb{R}^3$ . La distance de Hausdorff  $d_H(A, B)$  est définie par

$$d_H(A, B) = \max\left\{\sup_{a \in A} \inf_{b \in B} \|a - b\|, \sup_{b \in B} \inf_{a \in A} \|b - a\|\right\}.$$

Autrement dit, on mesure pour chaque point de l’un des ensembles sa distance au plus proche point de l’autre ensemble, puis on prend le maximum de ces deux valeurs.

Table 1: Comparaison des distances d’Hausdorff pour les méthodes de simplification OoCSx et Adaptative

Méthode	#Faces	#Échant.	$d_{H,\min}$	$d_{H,\max}$	$d_{H,\text{moy}}$	$d_{H,\text{RMS}}$	$d_{H,\max}/D$	$d_{H,\text{moy}}/D$	$d_{H,\text{RMS}}/D$
OoCSx	1 212 875	2 250 470	0.0000	11.7187	0.0810	0.1806	0.003157	0.000022	0.000049
	500 306	1 500 714	0.0000	31.3172	0.2810	0.6439	0.008423	0.000076	0.000173
	272 036	815 949	0.0000	40.5459	0.4772	1.1098	0.010914	0.000128	0.000299
	141 848	425 470	0.0000	47.6542	0.7761	1.7551	0.012836	0.000209	0.000473
Adaptative	1 182 358	2 250 470	0.0000	7.0911	0.0524	0.0915	0.001908	0.000014	0.000025
	509 326	1 527 978	0.0000	78.0413	0.1487	0.3131	0.021019	0.000040	0.000084
	275 574	826 708	0.0000	30.1374	0.2304	0.3522	0.008077	0.000062	0.000094
	145 289	435 894	0.0000	102.8518	0.3625	0.9126	0.027422	0.000097	0.000243

#### Légende des paramètres :

**#Faces** Nombre total de faces dans le maillage simplifié.

**#Échant.** Nombre de points échantillonnes sur le maillage pour le calcul de la distance.

$d_{H,\min}$  Distance de Hausdorff minimale observée.

$d_{H,\max}$  Distance de Hausdorff maximale observée.

$d_{H,\text{moy}}$  Valeur moyenne de la distance de Hausdorff.

$d_{H,\text{RMS}}$  Racine carrée de la moyenne des carrés (RMS) des distances.

$D$  Longueur de la diagonale de la boîte englobante du modèle de référence.

$d_{H,\max}/D$ ,  $d_{H,\text{moy}}/D$ ,  $d_{H,\text{RMS}}/D$  Distances normalisées par  $D$ .

D’après la Table 1, plusieurs observations :

- Malgré des distances maximales observées plus grandes pour Adaptative, elle reste globalement plus fidèle que OoCSx
- Plus la simplification est élevée (le nombre final de faces est faible), plus l’écart entre les deux méthodes se creuse, montrant la capacité d’Adaptative à mieux préserver les détails.

## 7 Discussion

### 7.1 Gantt

Nous avons quasiment réussi à implémenter les articles qui étaient prévus dans le Gantt (figure 25). Mais comme expliqué plus tôt, l'implémentation de l'article de stream nous a retardé et empêché de réaliser de véritables comparaisons entre les différents algorithmes.

### 7.2 Méthode de correction des erreurs du stream

Pour pouvoir corriger les erreurs présentes dans notre implémentation de l'article du stream, nous avons mis en place différentes méthodes qui nous ont permis de mieux comprendre, et de repérer rapidement les erreurs.

- Un affichage des triangles selon si ils sont en bordures ou non. (figure 23)
- Affichage dans le terminal, de toutes les listes présentes dans notre structure avec un code couleurs pour une meilleure visibilité.
- Et utilisation d'un nuage de point pour avoir les véritables id des sommets (figure 24), permet un repérage et une traçabilité grâce au terminal des triangles et sommets problématiques.

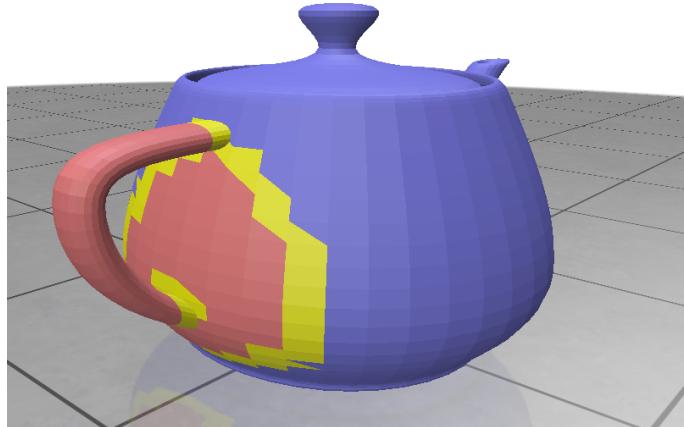


Figure 23: Affichage des triangles en fonction de leurs apparteneances: En rouge, les triangles qui sont actuellement en mémoire, en jaune les triangles qui possèdent un ou plusieurs sommets sur une bordure, et en bleu affichage du reste du maillage qui n'a pas encore été mis dans le tampon de traitement. Ce dernier affichage est possible car nous faisons nos tests sur des maillages qui rentrent en mémoire.

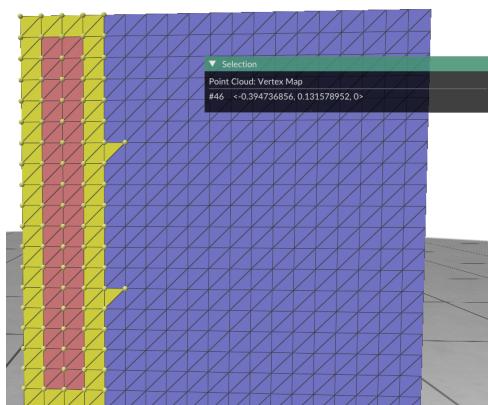


Figure 24: Nuages de points pour repérage des erreurs



Figure 25: Gantt

### 7.3 Conclusion

En conclusion, ce semestre de TER nous a permis d'explorer en profondeur les enjeux et les techniques de simplification de maillages hors mémoire. À travers l'étude et l'implémentation de trois algorithmes majeurs : OoCSX de Lindstrom et Silva, la méthode adaptative de Shaffer et Garland, et la décimation en flux de Wu et Kobbelt.

La première partie, consacrée à OoCSX, nous a familiarisés avec les quadric error metrics et les méthodes de tri externe pour traiter des modèles gigantesques sans surcharger la mémoire vive. L'approche adaptative a ensuite montré l'intérêt de concentrer la subdivision là où la géométrie est la plus complexe, au prix d'une logique d'arbres BSP plus sophistiquée. Enfin, l'algorithme de streaming nous a confrontés à la gestion fine des frontières du tampon et aux défis de la cohérence topologique en une seule passe.

## References

- [1] P. Lindstrom and C. T. Silva. A memory insensitive technique for large model simplification. In *Proceedings Visualization, 2001. VIS '01*, pages 121–550, San Diego, CA, USA, 2001. IEEE.
- [2] Peter Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00*, page 259–262, USA, 2000. ACM Press/Addison-Wesley Publishing Co.
- [3] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In B. Falcidieno and T. L. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465, Genova, Italy, June 1993. Springer-Verlag. Proceedings of the international conference, June 28–July 2, 1993.
- [4] Eric Shaffer and Michael Garland. Efficient adaptive simplification of massive meshes. In *Proceedings Visualization, 2001. VIS '01*, pages 127–534, San Diego, CA, USA, 2001. IEEE.
- [5] Jianhua Wu and Leif Kobbelt. A stream algorithm for the decimation of massive meshes. In *Graphics Interface Proceedings*, volume 3, pages 185–192, 04 2003.

## 8 Annexes

```
struct StreamMeshData {  
  
    // Associe à chaque coordonnée 3D un identifiant unique de sommet.  
    // Cet identifiant sert de clé dans toutes les autres structures, à la place des pointeurs de demi-arêtes.  
    std::map<Vertex, int> vertexMap;  
  
    // Liste des voisins immédiats d'un sommet, construite lors de la lecture  
    // de la première arête qui lie ces deux sommets.  
    std::map<int, std::vector<int>> adjacencyList;  
  
    // Liste des voisins "déjà rencontrés" pour chaque sommet, équivalente à la détection  
    // d'une seconde demi-arête opposé. C'est sur cette table que repose la détection des  
    // arêtes internes et la qualification des sommets "décimables".  
    std::map<int, std::vector<int>> adjacencyListAlreadySeen;  
  
    // Liste des voisins pour lesquels un triangle a déjà été émis en sortie,  
    // remplaçant la notion de demi-arête en bordure d'écriture.  
    std::map<int, std::vector<int>> adjacencyListAlreadyWritten;  
  
    // Pour chaque sommet (id), liste des indices de triangles où il apparaît.  
    // Similaire au chainage des demi-arêtes autour d'une face, mais en mode liste brute.  
    std::map<int, std::vector<int>> triangleList;  
  
    // Ensemble des sommets dont adjacencyList est vide, c'est-à-dire prêts à être décimés  
    // (équivalent à un sommet dont toutes les demi-arêtes opposées ont été vues).  
    std::vector<VertexAlreadyBeSimplify> vertexNotInBorder;  
  
    // Sommets qui ont commencé à écrire un triangle (bordure d'écriture $P_{(BC)}$),  
    // équivalent aux demi-arêtes dont la face opposé n'existe pas encore en sortie.  
    std::vector<int> vertexInBorderBC;  
  
    // Buffer en mémoire des coordonnées des triangles actifs.  
    std::map<int, TriangleCoordinates> inCoreTriangleBuffer;  
  
    // Stocke pour chaque sommet son quadric error metric, utilisé lors des contractions d'arêtes.  
    std::map<int, Quadric> triangleQuadricMap;  
  
    // Permettent de générer des identifiants croissants pour sommets et triangles.  
    int unique_triangle_index = 0;  
    int actual_unique_id = 0;  
};
```

Figure 26: Structure utilisé pour l'article de stream