

Etudiant : Nathan Dahy

Année Universitaire 2022-2023

Projet de fin de Master.

Formation suivie : Master Informatique et Mobilités (2ème année)

à l'Université de Haute-Alsace (UHA)

# Fusion convexe de polyèdres



Tuteur : Mr Julien LEPAGNOT, professeur d'informatique

# Remerciements

Je souhaite remercier mon tuteur, Mr Lepagnot, pour m'avoir offert l'opportunité de ce projet, ainsi que pour son suivi. Vu que c'est pour moi la fin de mes études et que j'en ai effectué quatre ans au sein de l'UHA, je souhaite remercier l'ensemble des enseignants que j'ai eus, pour leur pédagogie et leur passion de l'informatique.

# Sommaire

<b>Remerciements</b>	<b>2</b>
<b>Sommaire</b>	<b>3</b>
<b>I. Introduction</b>	<b>4</b>
<b>II.Contexte et sujet</b>	<b>4</b>
<b>III.Réalisation</b>	<b>5</b>
<b>IV.Problèmes rencontrés et solutions apportées</b>	<b>10</b>
<b>IV.Conclusion</b>	<b>13</b>
<b>IV.Bibliographie:</b>	<b>14</b>

N.B: Toutes les images utilisées dans ce rapport sont des captures d'écrans que j'ai réalisées ou dans le cas contraire leurs sources sont indiquées.

# I. Introduction

Ayant arrêté mon stage pour des raisons médicales, j'ai eu l'opportunité de réaliser à la place un projet, sous l'encadrement de Mr Julien Lepagnot.

Le sujet consiste au développement en C++ d'un algorithme capable de trouver, pour un ensemble de polyèdres donnés, une fusion en un certain nombre de polyèdres convexes. La solution optimale consiste à obtenir un minimum de polyèdres.

Le travail s'est effectué sur 6 semaines environ, à distance, ainsi que par échange de mail pour le suivi par le tuteur.

## II. Contexte et sujet

L'Université de Haute-Alsace (UHA) possède un département informatique (IRIMAS – INSTITUT DE RECHERCHE EN INFORMATIQUE, MATHÉMATIQUES, AUTOMATIQUE ET SIGNAL) dont l'équipe OMeGA - Équipe Optimisation par Métaheuristiques, alGorithmique et ModélisAtion, effectue notamment des recherches sur des problèmes géométriques.



Figure 1: Logo de l'IRIMAS. Source : <https://www.irimas.uha.fr/>

Dans ce cadre, et bien qu'il ne s'agit pas d'un sujet de recherche, vient donc notre problème de polyèdres.

Un polyèdre est un solide géométrique. Lorsqu'il est dit convexe, cela signifie que tout segment qui relie n'importe quel de ses points reste à l'intérieur du polyèdre. ("de façon plus visuelle il est "plein").

Le sujet consiste à réduire le nombre de polyèdres mais en gardant bien des polyèdres convexes, garant de l'intégrité du modèle 3D obtenu. Son domaine d'application est toute l'imagerie de synthèse / jeux vidéo / tout domaine nécessitant de réduire le nombre de modèles 3D pour être plus optimisé.

Une première équipe a développé une version répondant à ce cahier des charges. Elle n'est cependant pas totalement fonctionnelle. L'objectif est donc de développer une solution fonctionnelle en corrigeant cette version ou en recréant un projet.

J'ai préféré repartir de zéro car cela me semblait plus sûr que d'aller chercher l'erreur dans un code qu'il faut d'abord comprendre. Cependant lors de phases de recherches et de blocage je suis allé voir comment cela avait été fait dans cette précédente version.

### III. Réalisation

Le projet a donc été réalisé en C++ avec l'IDE CodeBlocks, un Workflow classique et très utilisé à l'UHA. (et de manière générale dans la recherche et dans le développement de jeux vidéo).

Pour les données d'entrée et la visualisation des polyèdre, le format imposé est l'OBJ. Celui-ci donne de manière succincte toutes les informations nécessaires :

- Liste des points (X, Y, Z)
- Liste des polyèdres ainsi que pour chaque polyèdre:
- Liste des faces.

En plus de cela, le fichier OBJ permet de définir d'autres informations qui ne nous sont pas toutes utiles, telles que:

- Les textures de chaque point
- Les normales de chaque faces (finalement non utilisés ici)
- Les noms des objets
- Des commentaires avec un # (ici par exemple la version de Blender utilisée)

```
# cube.obj
#

o cube

v 0.0 0.0 0.0
v 0.0 0.0 1.0
v 0.0 1.0 0.0
v 0.0 1.0 1.0
v 1.0 0.0 0.0
v 1.0 0.0 1.0
v 1.0 1.0 0.0
v 1.0 1.0 1.0

f 1 4 3
f 1 2 4
f 3 8 7
f 3 4 8
f 5 7 8
```

Figure 2: exemple de définition d'un cube dans un fichier .OBJ

```
vn 1.0000 -0.0000 -0.0000
vn -0.0000 -0.0000 1.0000
vn -0.0000 -1.0000 -0.0000
vn 0.5774 0.5774 -0.5774
vt 0.375000 0.000000
vt 0.625000 0.000000
vt 0.625000 0.250000
vt 0.375000 0.250000
vt 0.625000 0.500000
vt 0.375000 0.500000
vt 0.625000 0.750000
```

Figure 3: Normales et textures dans un fichier OBJ

Pour visualiser les résultats (ce qui est important car l'humain ne sait pas "lire" un fichier OBJ, et pour un résultat naturel), on utilise une visionneuse 3D; il y en a plusieurs possibles. Mon choix s'est porté sur Blender; en effet je peux ainsi modifier dans le logiciel les polyèdres et j'ai pu donc créer plusieurs dataset de données d'entrées; j'ai pu aussi tester facilement des modifications; cela m'a été utile pour le débogage.

Il est facile d'importer et d'exporter des fichiers .OBJ sous blender.

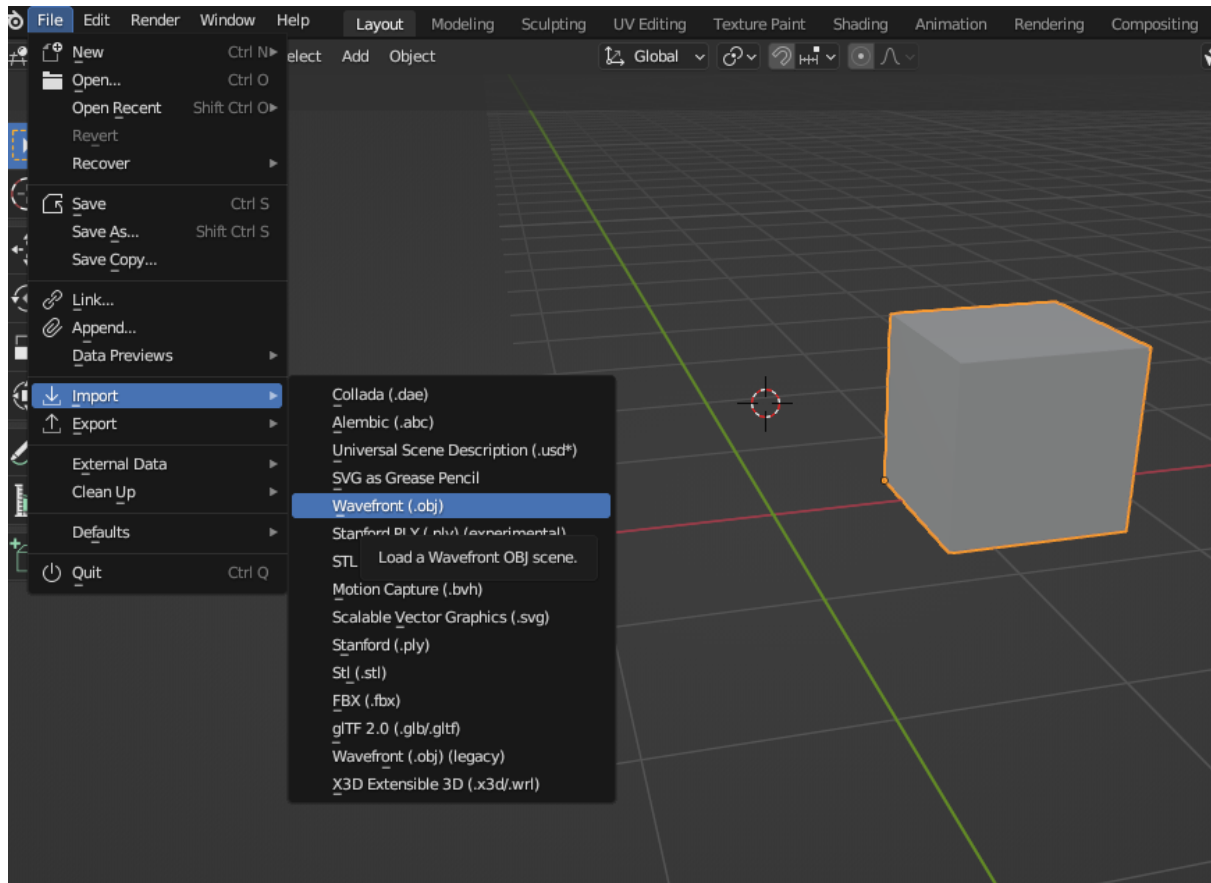


Figure 4: Logiciel Blender; importation et exportation en OBJ de ce cube.

## Lecture, écriture et structure des données

Afin de travailler sur les données contenues dans le fichier .OBJ, il a fallu les transférer vers nos structures de données (programmation objet).

Après avoir exporté les données via Blender, en ayant créé un dataSet intéressant, le programme se charge de passer de ces données textuelles à notre structure de données, qui est la suivante:

### Classe Point

Contient les données du point (coordonnées X, Y, Z).

### Classe Face

Contient les différents points de cette Face (donc minimum trois) sous forme de vecteur (et non vecteur de pointeurs)

### Classe Polyèdre

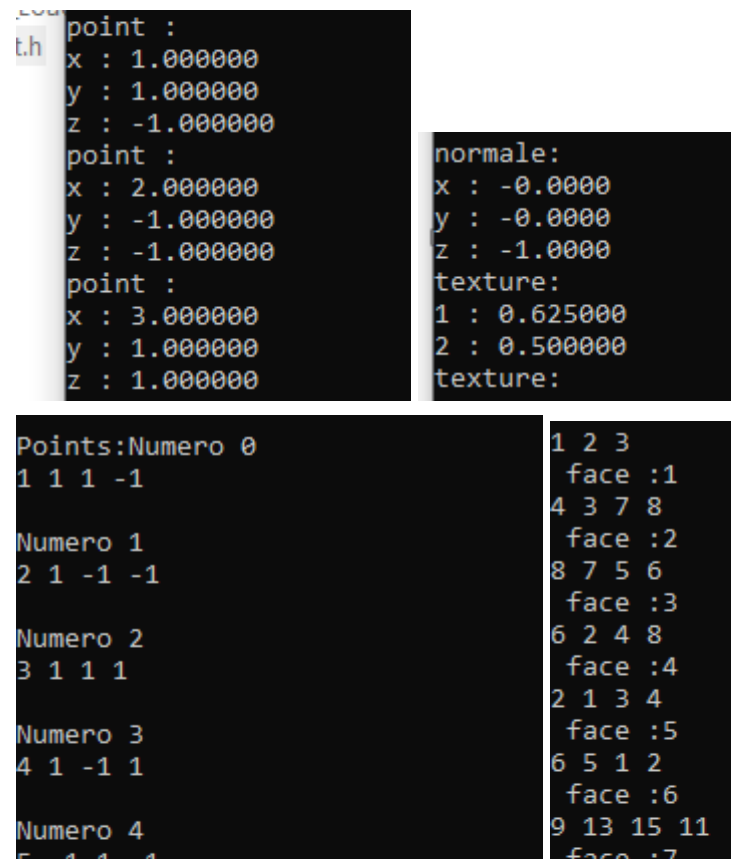
Contient les différentes faces constituant un polyèdre, sous forme de vecteur (et non vecteur de pointeurs)

## Classe Plan

Permet notamment de facilement construire un plan à partir de trois points, d'obtenir facilement les vecteurs directeurs et normaux, ce qui est utile pour calculer la position d'un point par rapport à un plan (cela nous sera nécessaire pour le calcul de la convexité)

## Fichier Main

Centralise et ordonne les tâches (lecture, algorithme, sortie)



```

point :
x : 1.000000
y : 1.000000
z : -1.000000
point :
x : 2.000000
y : -1.000000
z : -1.000000
point :
x : 3.000000
y : 1.000000
z : 1.000000
normale:
x : -0.0000
y : -0.0000
z : -1.0000
texture:
1 : 0.625000
2 : 0.500000
texture:

Points:Numero 0
1 1 1 -1
Numero 1
2 1 -1 -1
Numero 2
3 1 1 1
Numero 3
4 1 -1 1
Numero 4
5 1 1 1
1 2 3
face :1
4 3 7 8
face :2
8 7 5 6
face :3
6 2 4 8
face :4
2 1 3 4
face :5
6 5 1 2
face :6
9 13 15 11
face :7

```

Figure 5: Diverses captures d'écran de la lecture de données



## Algorithme principal

Voici en brut l'algorithme utilisé :

Etapes à répéter pour tous les ordres possibles de polyèdres:

Etapes a repeter tant que des fusions sont encore possibles (fusion totale)

En allant du 1er au N-ieme polyèdre:

Fusionner le polyedre en cours avec le suivant

Si la fusion ne donne pas un polyedre convexe

ne pas fusionner et laisser le polyedre en cours en stockage,

puis partir de ce polyedre non ajoutable en tant que polyedre en cours

Si solution trouvee donne moins de polyedres finaux que le minimum actuel

stocker ce minimum et l'ordre par lequel on y est parvenu

Mais ce n'est que peu lisible. Je vais tâcher de l'expliquer plus en détails:

-Pour fusionner les polyèdres en un nombre de polyèdres convexes minimal, il faudra d'abord trouver un moyen de vérifier la convexité obtenue lors d'une fusion.

On utilise pour cela une propriété des polyèdres convexe:

En prenant n'importe quelle face du polyèdre, tous les points du polyèdre doivent se trouver du même côté par rapport au plan formé par cette face. Si cette règle n'est pas respectée, on en déduit que le polyèdre n'est pas convexe.

-En prenant les polyèdres les uns après les autres, on les fusionne ainsi en les ajoutant autant que possible au précédent; si la fusion ne donne pas de polyèdre convexe, on abandonne la fusion; le polyèdre précédent est stocké et l'on continue la fusion sur les polyèdres suivants. Cela donnera un nombre de polygones réduits mais pas forcément minimaux.

-En effet un second passage (puis un n-ième) fusionnant les ensembles précédemment obtenus permet d'obtenir un nombre plus petit. On continuera à fusionner autant que c'est possible jusqu'à arriver à un nombre irréductible.

-L'ordre de traitements de ces polygones influe sur le nombre de polygones finaux.

Cela étant dit, le but de l'algorithme est de trouver une solution optimale, soit un nombre de polyèdres minimaux. Pour cela il faut soit utiliser une méta-heuristique ou autre algorithmique, soit y aller en force brute et tester toutes les possibilités. Ce sera cette dernière solution ici. Pour un set de polyèdres, il y a un certain nombre de permutations possibles, chacune donnant un résultat propre. Cela limitera cependant le nombre de données en entrée puisque pour N polyèdres on a N! permutations possibles.

```

Poly :2
13 14 15 16 17
53 54 55 56 57

Poly :3
93 94 95 96 97
125 126 127 128
155 156 157 158

0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
1 0 2 3
1 0 3 2

```

Figure 6: permutations possibles obtenues de l'ordre des polygones

## Sortie des données

Une fois que nous avons modifié nos données il faudra les remettre dans un fichier OBJ pour pouvoir de nouveau les visualiser. On itère sur les vecteurs de données obtenus afin de sortir toutes les données, en passant par les polyèdres et en descendant dans la hiérarchie des objets.

L'utilisation d'une fonction m'a permis de faire une sortie de données à volonté, par exemple chaque fois qu'une fusion plus optimisée était trouvée.

```

str << minimumPolygones;

str << "Polygones.obj";

string nomFichier = str.str();

ecrireFichier(listePoints, listePoly, nomFichier);

```

Figure 7: Capture de code: la fonction `ecrireFichier` écrit dans l'OBJ

## IV.Problèmes rencontrés et solutions apportées

### Usage d'une bibliothèque externe

Pour la lecture des données, j'ai pensé utiliser une bibliothèque afin d'avancer plus vite. J'en ai trouvé une fonctionnelle, me donnant les points et les polyèdres, mais ne gérant pas les faces (et apparemment faites particulièrement pour développer avec de l'OpenGL).

Or je ne pouvais faire sans cette condition. J'ai donc dû abandonner cette option et développer une solution "à la main". Cela m'a pris, avec la sortie de données, beaucoup de temps même si ce n'était pas la partie la plus difficile.

```
- find materials in: untitled.mtl
- Cube | vertices > 8 | texcoords > 14 | normals > 6 |
- Cube.001 | vertices > 16 | texcoords > 28 | normals
Mesh 0: Cube
Vertices:
V0: P(1, 1, -1) N(-0, 1, -0)
V1: P(-1, 1, -1) N(-0, 1, -0)
V2: P(-1, 1, 1) N(-0, 1, -0)
V3: P(1, 1, 1) N(-0, 1, -0)
V4: P(1, -1, 1) N(-0, -0, 1)
V5: P(1, 1, 1) N(-0, -0, 1)
V6: P(-1, 1, 1) N(-0, -0, 1)
V7: P(-1, -1, 1) N(-0, -0, 1)
V8: P(-1, -1, 1) N(-1, -0, -0)
V9: P(-1, 1, 1) N(-1, -0, -0)
V10: P(-1, 1, -1) N(-1, -0, -0)
V11: P(-1, -1, -1) N(-1, -0, -0)
V12: P(-1, -1, -1) N(-0, -1, -0)
V13: P(1, -1, -1) N(-0, -1, -0)
V14: P(1, -1, 1) N(-0, -1, -0)
V15: P(-1, -1, 1) N(-0, -1, -0)
```

Figure 8: Lecture de données que j'avais obtenue avec la bibliothèque

## Reprise du travail précédent

Dès que j'étais bloqué, je m'inspirais de ce qui avait été fait sur le travail précédent. J'ai cependant tout repris et revérifié. Au final, j'ai réutilisé:

- La structure de données (mais pas les indices, et pas les deque qui sont remplacés par des vectors).

- Les fonctions du plan et de la convexité.

## Classe plan

Je pensais pouvoir me passer d'une classe plan et faire directement les calculs de convexité.

Il s'est avéré plus pratique à l'utilisation d'avoir cette classe Plan.

## Pointeurs ou copie

Le travail précédent utilisait des indices pour gérer les relations entre objets. J'ai pour ma part hésité entre passer des valeurs par copie ou références (pointeurs). J'ai finalement tout fait par copie en veillant bien au bon fonctionnement et en testant.

## Vector

J'ai préféré utiliser les vecteurs car je connais bien cette structure de mon parcours à l'UHA. J'ai dû bien penser à les vider. Après coup je me rends compte que des opérateurs m'aurait évité le parcours des vecteurs pour les copier/ tester leur égalités

## Structure de l'OBJ

Le fait qu'on puisse mélanger les déclarations de points et de faces/objets m'ont posé problèmes. J'ai découvert qu'on pouvait définir tous les points dès le début. Cela permet de les mutualiser mais aussi d'éviter des problèmes d'ordres.

```
o Polyedre0
f 1 5 7 3
f 4 3 7 8
f 8 7 5 6
f 6 2 4 8
f 2 1 3 4
f 6 5 1 2
o Polyedre1
f 9 13 15 11
f 12 11 15 16
f 16 15 13 14
f 14 10 12 16
```

Figure 9: Structure de données OBJ avec définitions des polyèdres/faces après les points

A noter que j'ai créé un objet par polyèdre. Cela permet de bien différencier la fusion et de, sous Blender, bien voir quels objets ont été fusionnés.

## Erreur sur le calcul de convexité

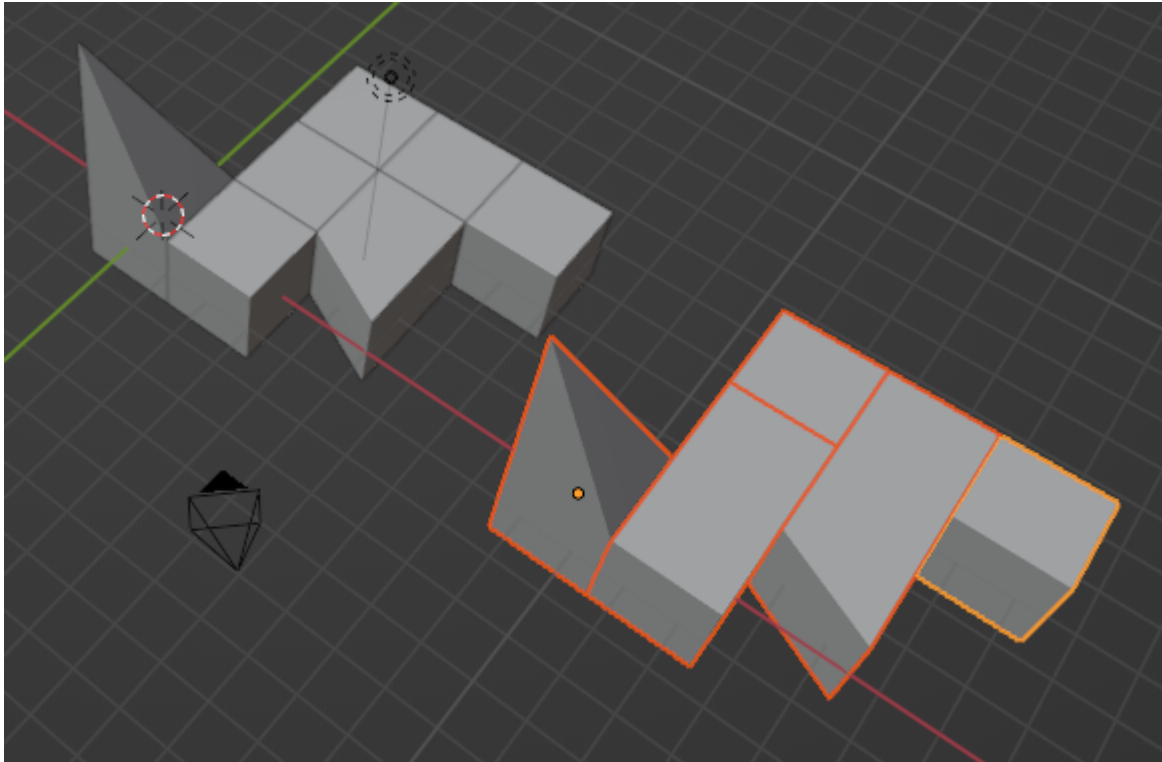
J'ai eu de mauvais résultats dans le calcul de convexité; des polygones étaient marqués non convexes alors qu'ils l'étaient. L'erreur venait du fait que je considérais la face commune (à supprimer évidemment) dans ce calcul.

## IV. Conclusion

Après avoir tout testé et réglé tous les problèmes algorithmiques, j'ai cependant (à l'heure où j'écris ces lignes) été coincé sur l'aspect suivant:

-répétition tant que la fusion n'est pas optimale.

L'algorithme fonctionne cependant; voici un exemple avec 7 polyèdres que j'ai conçu:



*Figure 10: A gauche les polyèdres AVANT, à droite APRES. Il s'agit d'une solution obtenue par l'une des permutations. Les deux cubes modifiés sont là pour mettre en avant l'aspect convexité nécessaire à la fusion: le premier n'est pas fusionné, le deuxième n'était pas fusionnable avec un des deux adjacents.*

J'obtiens donc un résultat imparfait.

Je suis néanmoins plutôt satisfait mais j'aurais aimé aller jusqu'au bout.

La partie lecture écriture de données m'a pris beaucoup plus de temps que prévu, (quasiment la moitié du temps) d'où le fait que j'aurais voulu utiliser une bibliothèque.

L'aspect visualisation du résultat permet de bien voir ce que fait l'algorithme.

Le sujet était intéressant, je me suis aussi renseigné sur la partie méta-heuristique, et il aurait été effectivement trop court pour la traiter.

## V.Bibliographie:

### **Documentation pour l'aspect géométrique :**

<https://lexique.netmath.ca/polyedre-convexe/>

<https://fr.wikipedia.org/wiki/Poly%C3%A8dre>

<https://www.kartable.fr/ressources/mathematiques/methode/montrer-quun-vecteur-est-normal-a-un-plan/4536>

<https://stackoverflow.com/questions/471962/how-do-i-efficiently-determine-if-a-polygon-is-convex-non-convex-or-complex>

### **Pour l'aspect mathématique:**

[https://www.geeksforgeeks.org/stdnext\\_permutation-prev\\_permutation-c/](https://www.geeksforgeeks.org/stdnext_permutation-prev_permutation-c/)

### **Pour l'aspect programmation:**

<https://www.digitalocean.com/community/tutorials/string-concatenation-in-c-plus-plus>

<https://iq.opengenus.org/ways-to-remove-elements-from-vector-cpp/>

### **Autres :**

<https://www.irimas.uha.fr/>

**Résumé du rapport :**

Dans le cadre d'un projet de fin de master, on m'a confié la reprise/réalisation d'un logiciel. On cherche dans le milieu de la synthèse d'image et du jeu vidéo à optimiser le nombre de polygones représentant les modèles 3D. Pour ne pas perdre l'intégrité de la forme 3D ou du niveau de détails, on s'intéresse ici à réduire les polyèdres à un nombre minimal de polyèdres dits convexes; c'est-à-dire sans cavité, "pleins". Pour cela, à l'aide du format OBJ, nous allons traiter les données via un logiciel en C++ qui va réduire une suite de polyèdre en les parcourant. L'ordre étant important dans cet algorithme, on essaiera de traiter toutes les possibilités (force brute), grâce aux permutations, ce qui réduit cependant la taille de données d'entrée possible. Une fois appliqué, on obtient une possibilité de réduction.

**Mots-clés :**

polyèdres, convexe, .obj, permutation, optimisation, 3D