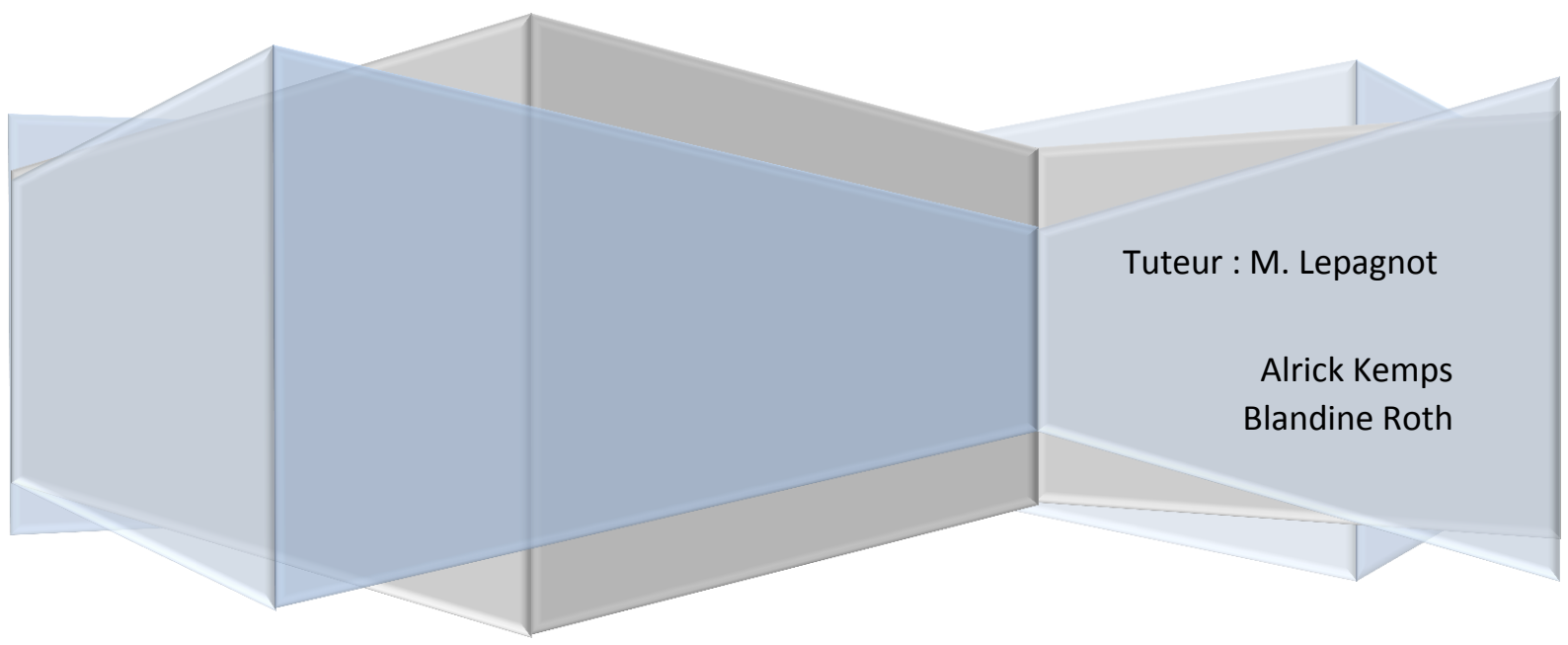


Rapport de projet tutoré
2015-2016

Fusion de polyèdres convexes



Tuteur : M. Lepagnot

Alrick Kemps
Blandine Roth

Sommaire

I.	Présentation du sujet	2
II.	Présentation de l'équipe	2
III.	Outils utilisés	2
A.	Source des données	2
B.	Développement	2
C.	Organisation des tâches	2
IV.	Structure du programme	3
A.	Classes	3
1.	Vertex	3
2.	Face	3
3.	Polyhedron	3
4.	Plane	3
5.	Program	3
6.	Utils	4
B.	Fichier main	4
V.	Organisation de l'algorithme	4
A.	Acquisition des données	4
B.	Stockage des données	4
1.	Structures de données de base	4
2.	Structures de données temporaires	4
C.	Etapas de l'algorithme	5
1.	Traitement de tous les cas de fusion possibles	5
2.	Remplissage des matrices après chaque permutation	5
3.	Test de fusion	5
4.	Fusion des formes de base vers tableau de polyèdres temporaire	5
5.	Détermination de la convexité d'un polyèdre	5
6.	Recherche du meilleur cas de fusion	6
7.	Traitement du résultat	7
8.	Exécution	7
VI.	Problèmes rencontrés et solutions apportées	9
A.	Le choix des structures	9
B.	Manipulation des structures deque	9
C.	Erreurs de programmation	9
D.	Liste de polyèdres de départ trop volumineuse	10
E.	Liste de faces mal pensée	10
F.	Fusion défectueuse	10
VII.	Conclusion	11
VIII.	Webographie	11

I. Présentation du sujet

Le sujet de notre projet se nomme « Fusion de polyèdres convexes ».

Il s'agit de créer un algorithme permettant de fusionner des polyèdres donnés entre eux et ce, deux par deux, à la condition que le résultat de leur fusion soit convexe, pour aboutir à un nombre minimal de polyèdres, c'est-à-dire les polyèdres finaux sur lesquels plus aucune fusion supplémentaire n'est possible.

Dans le cas de la liste de polyèdres fournie comme base de travail, ce nombre minimal de polyèdres obtenu sera de deux.

Il a été demandé de réaliser un algorithme de type « Brute Force » qui teste toutes les combinaisons possibles sur la liste des polyèdres fournie.

Bien que des structures de stockage soient proposées lors de la réunion préliminaire avec le tuteur du projet, M. Lepagnot, le choix est laissé aux étudiants.

II. Présentation de l'équipe

L'équipe est composée d'un tuteur de projet, M. Lepagnot, fournissant les informations nécessaires à la mise en route du projet et assurant un suivi et un support tout au long du déroulement de celui-ci, ainsi que de 2 étudiants à qui revient la tâche de développer l'algorithme, Alrick Kemps et Blandine Roth.

III. Outils utilisés

A. Source des données

Les données à récupérer se trouvent dans un fichier d'extension « .obj » fourni au début du projet. Il s'agit des coordonnées des points constituant des faces à l'aide de leur identifiant, elles-mêmes formant des objets qui représenteront les polyèdres sur lesquels nous effectuerons l'algorithme.

La visualisation des fichiers d'extension « .obj » se fait via le logiciel GLCPlayer.

B. Développement

Le développement de l'algorithme est réalisé par le biais de l'IDE Visual Studio 2013, en langage C++.

C. Organisation des tâches

La répartition du travail se fait à partir d'une Todo List fréquemment remise à jour. Chacun peut ainsi effectuer les tâches qu'il souhaite tout en suivant un ordre de priorité d'exécution pour éviter toute inertie au niveau du développement.

IV. Structure du programme

A. Classes

1. Vertex

La classe Vertex représente un point dans l'espace tridimensionnel. Un vertex est constitué de ses coordonnées x, y et z ainsi que d'un identifiant unique et se construit à partir de ces mêmes données récupérées dans le fichier « .obj ».

Il dispose de surcharge des opérateurs de comparaison == et != et de l'opérateur de sortie. Ce dernier servira à écrire dans les fichiers « .obj » obtenus après application de l'algorithme. Il comporte également des getters vers ses attributs.

2. Face

La classe Face représente une surface délimitée par un ensemble de sommets (ou Vertex). Une face est constituée d'un tableau de points 3D ainsi que d'un identifiant unique à partir desquelles elle sera construite.

Elle possède une fonction permettant de tester si elle est coplanaire avec une autre face, des getters vers ses attributs ainsi qu'une surcharge de son opérateur d'égalité ainsi que de son opérateur de sortie.

3. Polyhedron

La classe Polyhedron représente un polyèdre délimité par un certain nombre de faces. Un polyèdre possède un tableau de faces ainsi qu'un identifiant qui lui est propre et à partir duquel on peut le construire.

Il dispose de fonctions de modification de son tableau de faces (ajout et suppression) et d'opérations permettant de déterminer sa convexité et d'obtenir un tableau de numéros de faces partagées avec un autre polyèdre.

Il contient également des getters vers ses attributs ainsi qu'une surcharge de son opérateur de sortie.

4. Plane

La classe Plane représente un plan dans l'espace tridimensionnel.

Un plan est constitué des coefficients a, b, c et d que l'on retrouve dans son équation, « $ax + by + cz = d$ ». Il se construit à partir de 3 points de l'espace permettant de trouver ces mêmes coefficients.

Un plan possède une fonction permettant de trouver la position d'un point quelconque par rapport à lui-même à partir de son équation.

5. Program

La classe Program intègre l'acquisition des données ainsi que les étapes de l'algorithme. Le programme a pour attributs 3 listes d'éléments géométriques de base (points, faces et polyèdres) ainsi que de 2 matrices, l'une contenant les listes des numéros de faces partagées entre 2 polyèdres, l'autre indiquant à l'aide de booléens si la fusion entre 2 polyèdres est réalisable.

Il possède des fonctions de chargement et de sauvegarde de fichier « .obj » permettant respectivement d'exploiter les polyèdres d'un fichier de base et de fournir un fichier contenant les nouveaux polyèdres convexes résultant des fusions.

Il comporte également des fonctions permettant de remplir ses 2 matrices, des getters sur ses attributs et des fonctions d'opérations sur les structures pour gérer la fusion entre 2 polyèdres et les permutations au sein de la liste de polyèdres pour effectuer tous les tests de fusion possibles.

Enfin, il contient la procédure de traitement effectuant le chargement du fichier « .obj », le premier remplissage des matrices puis exécutant la procédure de permutations qui effectue la fusion et la sauvegarde des résultats.

6. [Utils](#)

La class Utils contient l'espace de noms « maths » où l'on retrouve des fonctions mathématiques personnalisées, à savoir le calcul d'un vecteur dans l'espace tridimensionnel à partir de deux points ainsi que la coplanarité (ou non) de 3 vecteurs.

B. [Fichier main](#)

Le fichier main.cpp contient la fonction permettant l'exécution de l'algorithme.

V. Organisation de l'algorithme

A. [Acquisition des données](#)

Le fichier « .obj » est lu par le programme et les données sont récupérées via une fonction de chargement au sein de la classe Program. Elles pourront ainsi être stockées dans les structures de données associées.

B. [Stockage des données](#)

Différentes données seront stockées tout au long du déroulement de l'algorithme. Les structures choisies pour cela permettront d'optimiser l'accès aux données, leur traitement ou encore leur modification.

1. [Structures de données de base](#)

Ces structures contiennent des informations qui seront récupérées à partir du fichier « .obj » mais qui ne seront jamais modifiées au cours de l'exécution de l'algorithme.

Les listes de points, de face et de polyèdres seront représentées par des deque.

2. [Structures de données temporaires](#)

Ces structures contiennent des données qui seront fréquemment modifiées lors de l'exécution de l'algorithme puisque pour chaque nouvelle itération il faudra les actualiser :

- Matrice des faces partagées : `vector< vector< vector<int>>>`
- Matrice des fusions : `vector< vector<bool>>`
- Un deque de polyèdres temporaire, basé sur une permutation de la liste chargée depuis le fichier, et sur lequel le programme va travailler

C. Etapes de l'algorithme

1. Traitement de tous les cas de fusion possibles

Pour permettre à l'algorithme de traiter tous les cas possibles, nous utilisons l'algorithme de Heap qui génère toutes les permutations possibles des éléments d'un tableau.

Le concept de cet algorithme est de générer chaque nouvelle permutation à l'aide de la précédente en intervertissant une seule paire d'élément et ce, de manière récursive. Les autres éléments, quant à eux, ne seront pas déplacés.

2. Remplissage des matrices après chaque permutation

Les matrices étant propres à une même permutation et à un niveau de fusion, il faudra les actualiser systématiquement après chaque nouvelle permutation et opération sur la liste de polyèdres.

3. Test de fusion

Un test de fusion sera effectué pour déterminer si deux polyèdres peuvent être fusionnés, c'est-à-dire s'ils ont une face en commun. Pour cela nous comparerons le nombre de sommets de chaque face d'un polyèdre avec le nombre de chaque face d'un autre polyèdre ainsi que les coordonnées des vertex qui les composent afin d'obtenir les faces communes.

4. Fusion des formes de base vers tableau de polyèdres temporaire

Pour ne pas porter atteinte à l'intégrité des données récupérées à partir du fichier « .obj » contenant les polyèdres de base, nous stockerons les polyèdres résultant des fusions dans un tableau temporaire, sur lequel de nouvelles opérations seront effectuées jusqu'à arriver au nombre minimal.

5. Détermination de la convexité d'un polyèdre

Pour déterminer la convexité d'un polyèdre, nous utilisons la propriété suivante : en considérant une des faces extérieures d'un polyèdre convexe comme un plan, les points constituant la face appartenant ainsi à ce même plan, tous les autres points du polyèdre seront placés du même côté dans l'espace par rapport au plan.

En revanche dans le cas d'un ou plusieurs points ne respectant pas cette condition, le polyèdre considéré ne serait pas convexe.

Pour mieux visualiser, voici quelques exemples avec deux polyèdres que nous savons convexes, un cube et une pyramide, ainsi qu'avec un polyèdre non convexe représenté par un cube creusé au niveau d'une de ses faces.

- Cube : Ici la face ABCD est confondue avec le plan P_{ABC} et tous les autres points du cube sont effectivement situés au-dessus du plan. Il est facilement remarquable que pour chaque autre face, la propriété sera respectée de la même manière. Le cube est donc convexe. (cf : Figure 1)

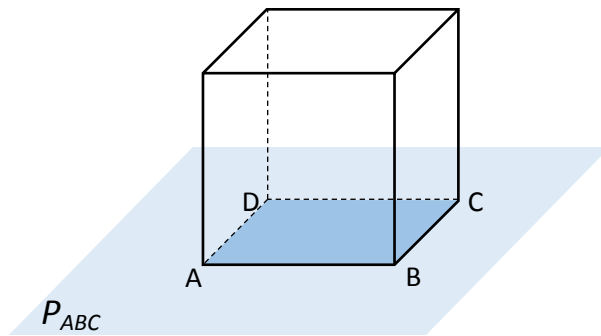


Figure 1 : Cube

- Pyramide : De même, pour chaque face nous pouvons constater que les autres points du polyèdre seront toujours situés du même côté dans l'espace. La pyramide est bien convexe. (cf : Figure 2)

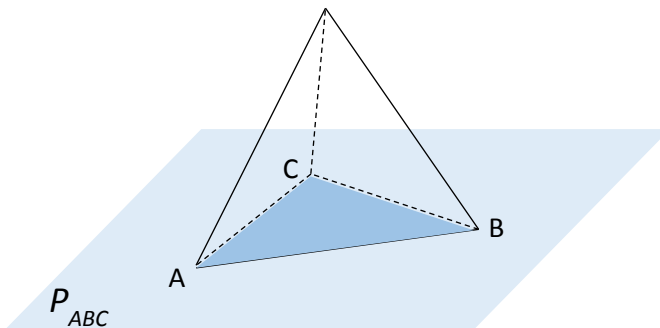


Figure 2 : Pyramide

- Cube creusé au niveau de sa face supérieure : En considérant le plan P_{ABE} obtenu à partir de la face ABE, on obtient que les points C et D sont situés au-dessus du plan tandis que les points F, G, H et I sont en-dessous. Le polyèdre n'est donc pas convexe. (cf : Figure 3)

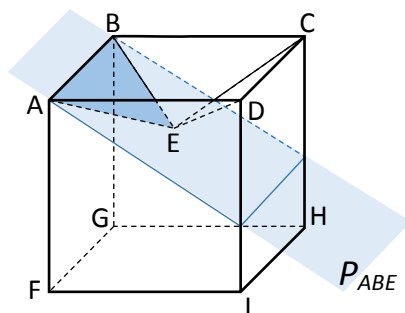


Figure 3 : Cube creusé

6. Recherche du meilleur cas de fusion

Pour chaque permutation, nous allons comparer la solution courante à la dernière solution stockée. Nous garderons celle avec le plus petit nombre de polyèdres, ou bien la dernière solution stockée dans le cas d'une égalité.

Lorsque toutes les permutations et tous les cas possibles auront été traités, nous disposerons d'une des meilleures solutions possibles (en fait, la première de celles-ci).

7. Traitement du résultat

A partir des données stockées en mémoire relatives à la meilleure solution, nous pourrions produire un fichier « .obj » contenant les coordonnées des éléments des polyèdres finaux sur le même modèle que le fichier « .obj » de départ et ainsi permettre à l'utilisateur de visualiser le résultat des fusions et de récupérer les données utiles.

8. Exécution

Pour finir, voici un exemple d'exécution pour illustrer le cheminement de l'algorithme.

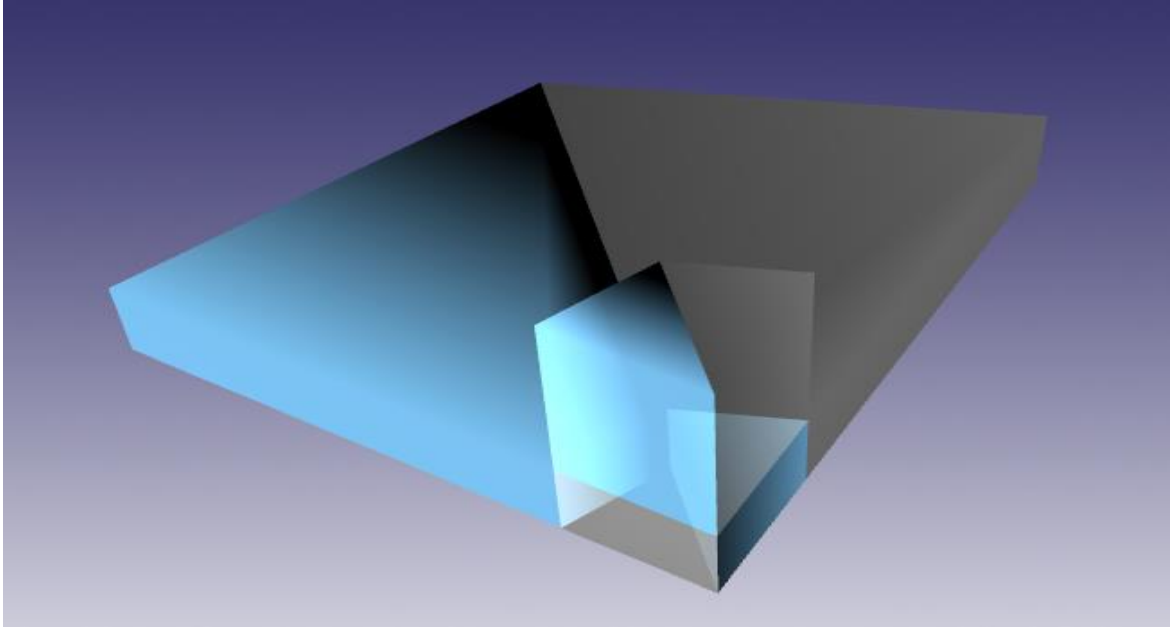


Figure 1 : L'ensemble de base est composé de 6 polyèdres.

Lors du premier passage, les polyèdres vont être fusionnés comme montré dans les figures 2, 3 et 4

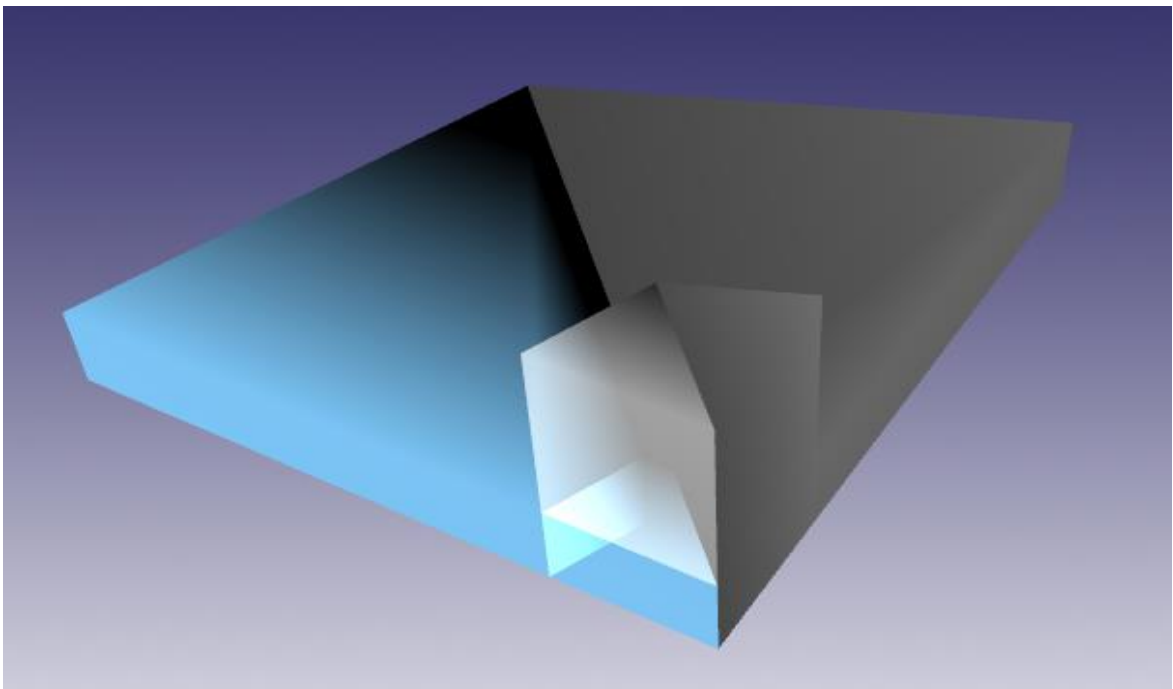


Figure 2 : Première fusion

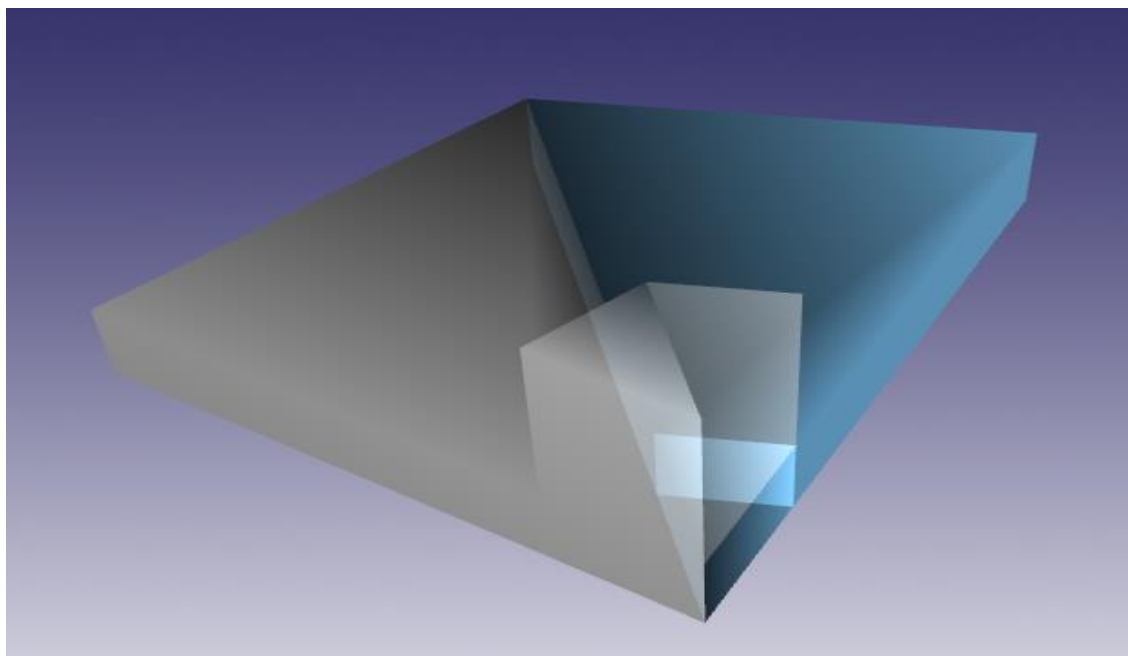


Figure 3 : Seconde fusion

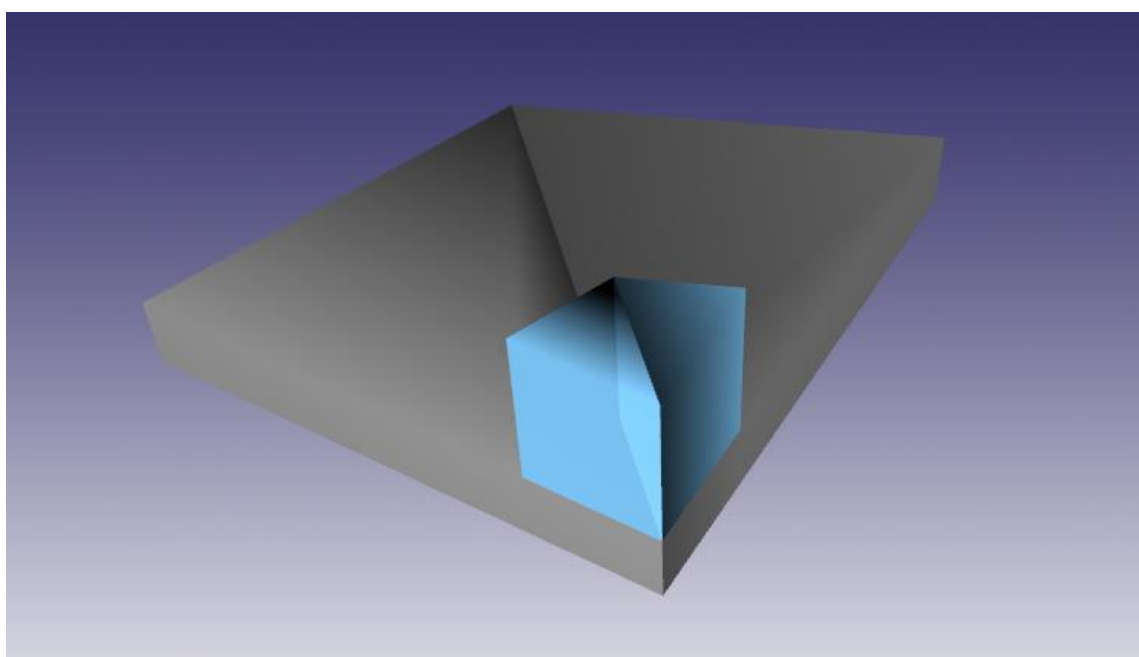


Figure 4 : Dernière fusion

Pour finir, lors du second passage (qui sera aussi le dernier pour cet exemple), les polyèdres résultant des fusions que l'on peut voir en figure 2 et 3 seront fusionnés et le résultat de cette permutation stocké. On recommencera ces opérations sur une nouvelle permutation.

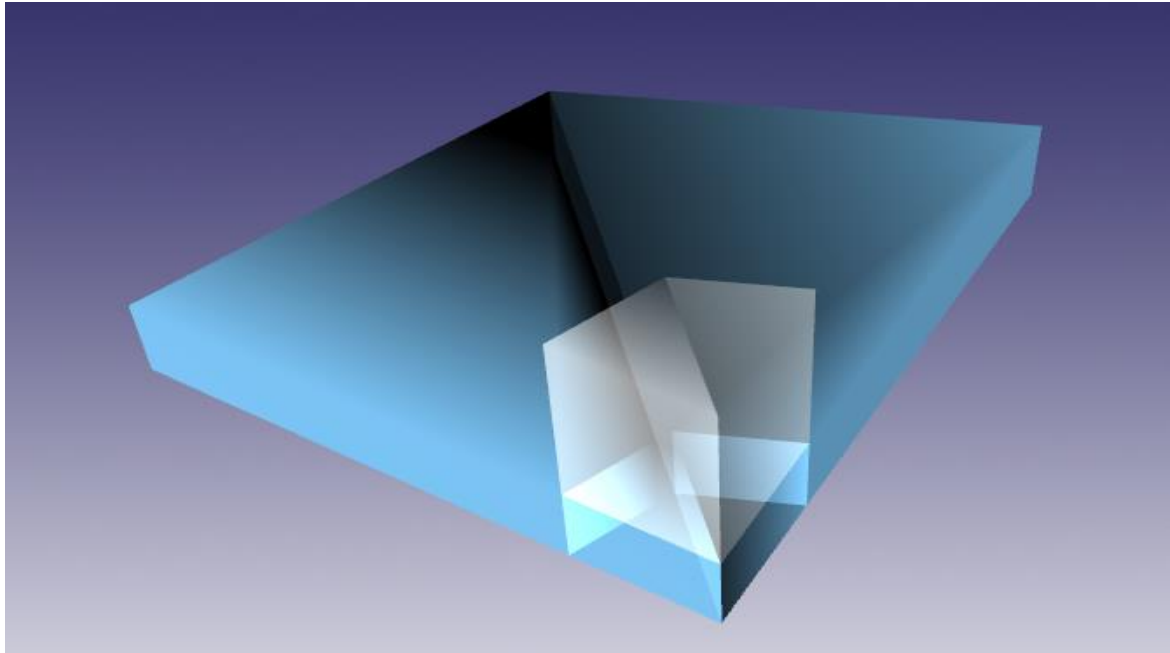


Figure 8 : Cette solution sera stockée puis comparée aux solutions suivantes pour conserver la meilleure d'entre elle

VI. Problèmes rencontrés et solutions apportées

Au cours du projet, nous avons fait face à divers problèmes qui ont entravé la bonne marche du développement à différents degrés. Fort heureusement, dans la grande majorité des cas, des solutions ont pu être mises en œuvre dans des délais raisonnables.

A. Le choix des structures

Certaines structures que nous pensions utiliser en premier lieu s'étant avérées peu pratiques en regard des traitements auxquelles elles étaient destinées, le choix des structures s'est finalement fait suite à la consultation d'un site regroupant un ensemble de tests sur différentes structures de stockage afin de juger de leur efficacité vis-à-vis des différentes opérations de base (insertion, suppression, affectation, etc...).

B. Manipulation des structures deque

N'ayant jamais utilisé de deque en C++, nous avons fait face, lors de la compilation, à des erreurs que nous n'arrivions pas à identifier en premier lieu. Après diverses recherches il est apparu que celles-ci étaient dues à une absence d'initialisation systématique que nous ne savions pas nécessaire lors de l'utilisation de ces structures en statique.

C. Erreurs de programmation

Des phases de tests fréquentes ont été réalisées après la mise en place de nouvelles fonctions dans le but d'éviter toute accumulation d'erreurs ainsi qu'une répercussion sur les fonctions développées par la suite.

Cela nous a par exemple permis de mettre en lumière un problème au niveau du calcul d'un des coefficient de l'équation du plan, que nous avons pu rectifier aisément grâce à un outil de calcul proposé sur le site que nous avons pris comme référence.

D. Liste de polyèdres de départ trop volumineuse

La liste de polyèdres de départ fournie par notre tuteur étant trop volumineuse (35 polyèdres) pour un résultat optimal de 3 polyèdres finaux, la phase de permutations à elle seule prenait beaucoup trop de temps et compliquait donc la mise en place de notre algorithme. Notre tuteur nous a ainsi proposé une liste raccourcie, ne possédant cette fois que 9 polyèdres avec un nombre optimal de 2 polyèdres finaux issus de fusions.

E. Liste de faces mal pensée

Au départ nous partions du principe que la liste de faces au sein de chaque polyèdre était constituée d'éléments Face ce qui causait des problèmes au niveau de l'utilisation des matrices de données temporaires nécessaires aux itérations successives de l'algorithme.

Après la lecture du 1^{er} jet de notre rapport, une mise au point a été effectuée avec notre tuteur qui nous a indiqué la marche à suivre au sujet de la liste de faces qui devenait alors une liste d'entiers correspondants à l'identifiant unique de chaque face, ce qui a réglé les problèmes sous-jacents.

F. Fusion défectueuse

La fusion ne se passe pas comme prévu puisqu'elle s'applique à des polyèdres qui ne devraient pas être fusionnés deux à deux. De plus, lors de la visualisation du résultat final, nous pouvons remarquer la disparition pure et simple de plusieurs polyèdres.

Testées une à une sur des tronçons de résultats, les différentes fonctions concernées se comportent bien comme elles sont censées le faire ce qui ne facilite pas la localisation du problème.

A l'heure où nous finalisons le rapport avant envoi, nous n'avons toujours pas identifié la cause de ce problème mais nous continuons à essayer de le résoudre.

VII. Conclusion

Le résultat obtenu étant encourageant mais pas satisfaisant en regard de ce qui nous était demandé dans le sujet du projet, nous avons tout de même commencé à réfléchir aux optimisations possibles qui pourraient être apportées à un tel algorithme, sous réserve bien sûr d'en avoir les ressources nécessaires, ici le temps.

Ainsi, la fusion pourrait être améliorée de sorte à, en plus de celle des faces communes, intégrer la fusion des faces qui se trouvent dans le prolongement les unes des autres. Cela permettrait d'alléger les tests lors de la recherche de convexité puisque le nombre de faces serait ainsi considérablement réduit.

Pour ce faire, nous avons pensé à deux méthodes distinctes, la première intégrant un test de colinéarité entre des vecteurs formés à l'aide de plusieurs points de deux faces distinctes (chacune appartenant à un polyèdre différent) puis à une recherche de 2 points partagés entre ces deux faces. La seconde en revanche, utiliserait la coplanarité entre les 2 faces grâce à la classe Plane déjà existante ainsi que la recherche de 2 points partagés.

De plus, si à l'heure actuelle nous sommes encore obligés de mettre à jour les matrices à chaque itération de l'algorithme, cela ne devait plus être le cas à terme, réduisant ainsi le nombre d'opérations secondaires de manière significative.

VIII. Webographie

Choix des structures de stockage :

<http://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html>

Algorithme de Heap :

https://en.wikipedia.org/wiki/Heap%27s_algorithm

Calcul de la position d'un point par rapport à un plan :

<http://www.had2know.com/academics/equation-plane-through-3-points.html>