# Implantation Fonctionnelle d'Algorithmes Géométriques en 3D
Quickhull

Titouan Laurent

## 1 Introduction

**Preuve assistée par ordinateur** Prouver la correction d'un algorithme, c'est à dire démontrer formellement qu'il répond à des spécifications précises est une tâche qui se veut essentielle dans de nombreux domaines de l'informatique. Cela permet de s'assurer de sa robustesse et donc de celle des implantations qui en découlent.

Cependant, il peut-être fastidieux de chercher à faire une démonstration formelle à la main, d'autant plus qu'en informatique l'écart entre une implantation donnée et sa description formelle représente une limite. Dans le cas des algorithmes géométriques, il y a l'introduction de difficultés supplémentaires, puisque l'on se retrouvera généralement à travailler aussi bien sur des notions topologiques qu'arithmétiques.

À partir de là, les outils outils de preuve automatique se présentent comme une solution viable. Ils permettent de démontrer des propriétés dans une implantation donnée d'un algorithme, et donc d'en prouver la correction. La contrainte étant que l'implantation doit être faite dans un style de programmation fonctionnel. Leur intérêt réside aussi dans le fait qu'on peut en extraire un programme certifié.

**Application au calcul de l'enveloppe convexe** Le calcul de l'enveloppe convexe d'un ensemble fini de points est un problème bien connu en géométrie et en informatique. Un certain nombre d'algorithmes existent pour le résoudre, et parmi ceux-ci Quickhull se veut être inspiré de Quicksort en proposant une approche de type diviser pour régner, lui assurant une complexité en $O(n \log n)$.

Par le passé, une implantation de Quickhull restreinte aux ensembles de points à 2 dimensions a été proposée et prouvée formellement. Seulement, la généralisation de Quickhull aux ensembles à 3 dimensions et plus, à partir de la version précédente, n'est pas une tâche triviale à cause des contraintes supplémentaires de visibilité (qu'on ne retrouve pas en 2 dimensions). Il faut également ajouter que les implantations en 3 dimensions existantes sont le pour la grande majorité faites dans un style de programmation impératif, et donc pas directement adapté à la preuve automatique.

Mon travail de recherche a donc pour objectif d'explorer ce qui est envisageable en terme d'implantation en style de programmation fonctionnel, de preuve mais aussi d'aide à la visualisation de l'algorithme Quickhull en 3 dimensions.

## 2 Choix d'un langage de programmation

Pour ce sujet le langage de programmation retenu est celui de Coq. Coq est un assistant à la construction de preuves automatiques, codées avec une syntaxe qui lui est propre, en paradigme fonctionnel.

Seulement, ici il présente au moins quatre limitations importantes :

— C'est un langage plutôt austère à prendre en main, sa syntaxe est loin d'être évidente et donc il n'est que peu pratique pour développer rapidement et aisément.

— Il ne permet nativement pas de faire du dessin graphique, ce qui serait pourtant très pratique pour vérifier visuellement le bon fonctionnement des programmes développés.

— Il en est de même pour la lecture et l'écriture de fichiers (pour stocker l'enveloppe convexe par exemple).
— Coq n'est pas un environnement dédié à l'exécution de programmes. En effet il n'est que très peu performant en temps et en capacités de calcul (même lorsqu'il s'agit de faire des opérations arithmétiques sur de simples entiers non signés).

J'ai donc porté mon choix sur le JavaScript, et ce pour les raisons suivantes :
— Mis à part Coq, je n'ai jamais utilisé de langage purement fonctionnel jusque-là.
— JavaScript est **multi-paradigme**, et donc il permet nativement de développer dans un style de **programmation fonctionnel**.
— C'est un langage que je connais particulièrement bien.
— JavaScript est adapté à la lecture/écriture de fichiers.
— JavaScript intègre nativement des outils d'**affichage graphique** (WebGL et dessin dans un Canvas avec des instructions simples).

# 3 Implantation de Quickhull en 2D

Avant de commencer à directement travailler sur Quickhull en 3 dimensions, il m'est bien plus évident de déjà comprendre le cas restreint (et donc plus simple) aux ensembles à deux dimensions, ne serait-ce que pour en cerner les caractéristiques principales.

L'algorithme Quickhull en 2 dimensions pourrait être qualifié de "cas d'école" du *diviser pour régner*. Pour s'en convaincre on peut reprendre l'exemple suivant, illustrant trois étapes successives de construction de l'enveloppe (en rouge).

On identifie ici clairement en vert les sous-ensembles indépendants de point sur lesquels on va récursivement opérer (opération *diviser*). Un sous-problème est ici de ne garder que les points de notre sous-ensemble qui appartiennent à l'enveloppe.

L'enveloppe convexe est alors construite par les opérations de combinaison des données de retour des sous-appels récursifs (il s'agit basiquement d'une liste ordonnée de points appartenant à ladite enveloppe).

**Note :** Il est important de noter ici que l'opération de division est possible pour la simple et bonne raison que lorsque l'on sélectionne un point extrême, les sous-ensembles de points obtenus sont strictement indépendant (leur union est vide). Ce détail est d'une importance cruciale pour comprendre le passage de la 2D à la 3D comme nous allons le voir.

# 4 Implantation de Quickhull en 3D

## 4.1 Données d'entrées

Un *vec*3 permet de modéliser aussi bien un point qu'un vecteur dans l'espace 3D. Les composantes sont données dans l'ordre usuel $x$, $y$ et $z$ (avec $x, y, z \in \mathbf{R}$).
Dans notre implantation, on considèrera une simple liste ordonnée des $n$ points (représentée par la variable globale $GLOBAL\_V\_LIST$) comme étant les données d'entrée du problème à résoudre.

## 4.2 Stockage et manipulation de l'enveloppe convexe en mémoire

La structure en liste de demi-arêtes orientées (doubly connected edge list, abrégé DCEL) est parmi les plus communes qu'il soit dès lors que l'on souhaite modéliser un maillage topologique.
Ici on implante une version restreinte de cette dernière, puisqu'elle permet de créer maillages triangulaires uniquement — ce qui n'empêche pas au programme d'être fonctionnel comme nous le verrons ensuite.

**Demi-arête** — *he*

Une *he* (abrévation de halfedge, ou demi-arête) est une structure de données disposant d'informations sur elle-même et son voisinage topologique. Celles-ci sont les suivantes :

— $index \in \mathbf{Z}$, un identifiant unique pour la demi-arête, avec :

$$\begin{cases} index = -1 & \text{est une demi-arête nulle} \\ index \geq 0 & \text{sinon} \end{cases}$$

— $opposite \in \mathbf{Z}$, l'identifiant de la demi-arête opposée, avec :

$$\begin{cases} opposite = -1 & \text{est une demi-arête du bord} \\ opposite \geq 0 & \text{sinon} \end{cases}$$

— $vertex \in \mathbf{N}$, l'indice du sommet incident dont elle est issue.

**Maillage triangulaire** — *dcel*

Une *dcel* permet la modélisation d'un maillage topologique triangulaire. Elle dispose des attributs suivants :

— *he_list*, une liste de *he*.

— $available\_he\_index \in \mathbf{N}$, le prochain indice non attribué à une *he* de la liste.

**Exemple simple en 2D**

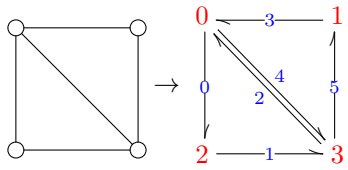


FIGURE 1 – Un maillage simple et un indiçage correspondant.

| index | opposite | vertex |
|---|---|---|
| 0 | -1 | 0 |
| 1 | -1 | 2 |
| 2 | 4 | 3 |
| 3 | -1 | 1 |
| 4 | 2 | 0 |
| 5 | -1 | 3 |

| Indice | Coordonnées |
|---|---|
| 0 | (0,0) |
| 1 | (1,0) |
| 2 | (1,1) |
| 3 | (0,1) |

*available_he_index* **:** 6

FIGURE 2 – Structure *dcel* à gauche et la table des sommets à droite.
Note : *avaiable_he_index* est un indice disponible pour la prochaine demi-arête qui pourrait être ajoutée à la *dcel*.

Considérons le maillage de la Figure 1. Une structure *dcel* associée valide serait alors telle que celle présentée Figure 2.

Note : si les numéros d'indices choisis pour les demi-arêtes sont ici plus ou moins arbitraires, la structure *dcel* impose tout de même des contraintes comme nous allons le voir.

#### 4.2.1 Ajout et retrait d'une face dans une *dcel*

### 4.3 Stockage et manipulation de l'ensemble de points extérieurs associés aux faces

**Sous-ensemble de points extérieurs** — *osubset*

Un sous-ensemble de points extérieurs (outside subset) est une structure de données à deux champs :
— $face\_index \in \mathbf{N}$, l'identifiant unique d'une face.
— $list \in \mathbf{N}$, l'ensemble des points extérieurs à la face $face\_index$, représentés par leur indice dans $GLOBAL\_V\_LIST$.

**Ensemble de points extérieurs** — *oset*

L'ensemble de points extérieurs (outside set) n'est rien d'autre qu'une liste d'*osubset*, chacune associée à une unique face (et son ensemble de points extérieurs associé).

## 5 Évaluation des performances

Lors d'une itération dans l'algorithme de calcul d'enveloppe convexe 3D, deux instructions successives amènent à devoir choisir un élément parmi un ensemble — sachant que tous conduisent à la terminaison du programme et à la construction d'une enveloppe convexe valide. Dès lors l'utilisation d'heuristiques est envisageable, et cette partie porte sur la comparaison des performances en temps de calcul en fonction des méthodes employées.

À la section 5 on s'intéresse à la sélection du prochain sous-ensemble de points non-vide (et donc implicitement d'une face du polygone convexe en cours de construction). Tandis qu'à la section 5 on s'intéresse à la sélection du point parmi ceux du sous-ensemble choisit précédemment.

Les tests ont été conduits en moyennant le temps de calcul pour 50 échantillons de points générés aléatoirement (répartition spatiale aléatoire uniforme).

## Heuristique sur le choix du point

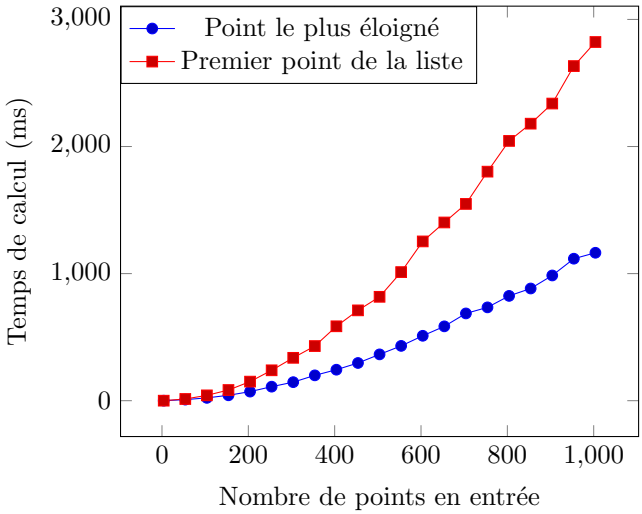


FIGURE 3 – La courbe bleue montre ici que la stratégie consistant à prendre le point le plus éloigné du plan courant offre de meilleures performances en temps que celle où l'on prend naïvement le premier point de la liste (courbe rouge).

## Heuristique sur le choix du sous-ensemble

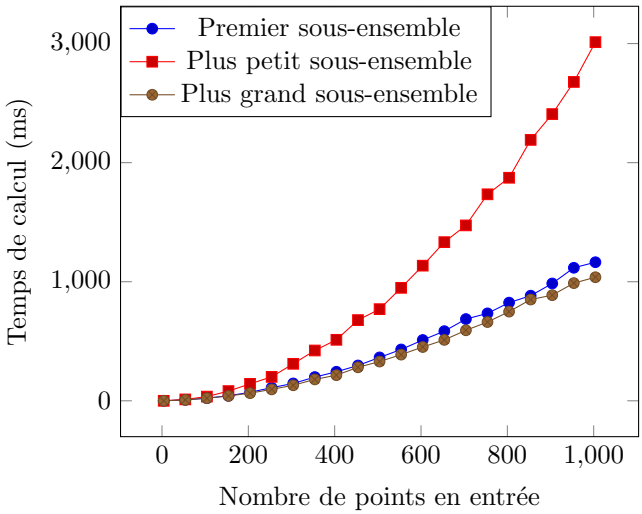


FIGURE 4 – De manière prévisible choisir le plus petit sous-ensemble à chaque itération nuit aux performances de l'algorithme (courbe rouge). En effet en ne prenant à chaque fois que un plus petit sous-ensemble de $n$ points (avec $n > 0$), on se restreint alors à retirer de l'ensemble total $n$ points au mieux à cette itération.
Il est intéressant de noter ici que prendre le plus grand sous-ensemble (courbe brune) est plus performant que de choisir le premier de la liste (courbe bleue), mais le gain n'est pas aussi significatif que ce qu'on a montré précédemment.