Sorbonne Université

Compte rendu écrit du projet

Initiation au C++ 4M016

Découpe d'un cube en tétraèdre et visualisation

David Lam
Alan Balendran

Décembre 2018

Remerciements

Nous tenons particulièrement à remercier notre chargé de TD en algorithme et compléxité, Razvan Barbulescu, pour nous avoir donner une condition nécessaire et suffisante pour que deux tétraèdres s'intersectent exactement en une face.
Enfin nous remercions également Frédéric Hecht et Xavier Claeys pour nous avoir accompagné et avoir partagé avec nous votre passion pour la programmation.

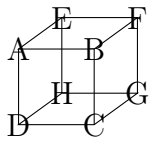
Plan

Le compte rendu se décompose en 3 parties. Dans la première, on modélise la situation, donne des définitions, remarques et quelques propriétés qui nous serviront plus tard. Dans la deuxième partie, on explique l'algorithme attendu du projet. Dans la troisième partie, on explique pas à pas les méthodes que nous avons utiliser pour implémenter l'algorithme en C++.

Partie 1

Dans tout le projet, nous travailleront dans le plan R^3. Le cube sera représenté par ses huit sommets. Chaque sommet sera représenté sous forme cartésienne (x,y,z) pour ses 3 coordonées. Un tétraèdre du cube sera représenté par ses 4 sommets du cube qui le compose. Ainsi deux tétraèdres sont égaux si les ensembles de leurs sommets qui les composent sont égaux. De même, deux faces quelconques d'un cube, respectivement deux faces quelconques d'un tétraèdres sont égaux, si les ensembles des sommets du cube, respectivement les ensembles des sommets du tétraèdre qui les composent sont égaux. Enfin, de faççon analogue, deux découpes seront égales si les tétraèdres qui les composent sont égaux.
De plus, nous travailleront uniquement sur le cube noté c, formés des sommets A(0,1,0), B(1,1,0), C(1,0,0), D(0,0,0), E(0,1,1), F(1,1,1), G(1,0,1),H(0,0,1) car tout cube dans R^3 peut se rammener à c par translations et hotométies.

Visuellement, cela donne ceci :



Definition 1. *On dit qu'un tétraèdre T du cube est un tétraèdre d'un coin, s'il a 3 faces incluses dans les faces du cube. Il y en a 8 au total qui sont (A,B,D,E), (A,E,F,H), (E,F,B,G), (A,D,H,C), (D,C,G,B), (C,G,H,F), (G,H,D,E) et (A,B,F,C).*

Definition 2. *On dit que deux tétraèdres T1 et T2 sont voisins s'ils s'intersectent exactement en une seule face F et on dit que F est leurs face commmune.*

Proposition 1. *Soient A,B,C,D quatre sommets distincts du cube, alors ou bien ils forment un carré, ou bien ils forment un rectangle, ou bien ils forment un tétraèdre.*

Proposition 2. *Soient A,B,C,D quatre sommets distincts du cube. Notons (xA,yA,zA), (xB,yB,zB), (xC,yC,zC), (xD,yD,zD) leurs coordonées. Alors ils sont inclus dans un même plan ssi*

$$\begin{vmatrix} xB - xA & xC - xA & xD - xA \\ yB - yA & yC - yA & yD - yA \\ zB - zA & zC - zA & zD - zA \end{vmatrix} = 0$$

Proposition 3. *Soient T1 et T2 deux tétraèdres d'un même cube,on suppose que T1 et T2 ont exactement 3 sommets en commun. Notons S1(x1,y1,z1), S2(x2,y2,z2), S3(x3,y3,z3) ces sommets communs et notons ST1(xt1, yt1, zt1) et ST2 (xt2, yt2, zt2) les sommets distincts de T1 et T2. Alors T1 et T2 sont voisins en la face commune S1,S2,S3 ssi*

$$\begin{vmatrix} x1 & x2 & x3 & xt1 \\ y1 & y2 & y3 & yt1 \\ z1 & z2 & z3 & zt1 \\ 1 & 1 & 1 & 1 \end{vmatrix} \begin{vmatrix} x1 & x2 & x3 & xt2 \\ y1 & y2 & y3 & yt2 \\ z1 & z2 & z3 & zt2 \\ 1 & 1 & 1 & 1 \end{vmatrix} < 0$$

Remarques

Les deux premières propositions assurent qu'il y a 58 tétraèdres différents. En effet on parcourt l'ensemble des $\binom{8}{4} = 70$ combinaisons de sommets puis on applique les conditions issues des propriétés.

De plus, ces 58 tétraèdres sont subdivisés en 2 groupes : ceux de volume égal à 1/3 et ceux de volume égal à 1/6. Le plus étonnant, c'est qu'il n'y a que deux tétraèdres de volume 1/3 et que ces deux tétraèdres déterminent chacun, une unique découpe, dans le sens où si on note D1 et D2 ces découpes, toute autre découpe contenant un de ces deux tétraèdres est forcément égal à D1 ou D2. Il ne reste donc qu'à trouver toutes les découpes issues des tétraèdres de volume 1/6.

Partie 2

Voici la façon dont nous allons procéder. Commençons par nous fixer T, un des 56 tétraèdres de volume 1/6. Notre but est de déterminer toutes les découpes qui contiennet T, chaque tétraèdre de ces découpes doit avoir un volume égal à 1/6, sinon, on revient aux cas déjà traités précédement.
Pour celà, nous allons modéliser un arbre A de tétraèdres, dont tous les chemins depuis la racine aux feuilles représenteront toutes les découpes qui contiennent T. L'arbre A sera initialement réduit à la racine = T et nous allons construire A de la manière suivante :

-On réalise une première opération en ajoutant les fils de T qui sont les tétraèdres qui vérifient les conditions de compatibilités avec T et qui sont voisins avec T. Remarquons qu'il s'agit là des tétraèdres voisins avec T.
-On réalise une deuxième opération : on parcourt la liste des feuilles fi, et pour chaque fi, on ajoute les fils de fi qui sont les tétraèdres qui vérifient les conditions de compatibilités avec ses ancêtres (ici fi et T) et qui sont de plus voisins avec au moins un de leurs ancêtres.
-On réitère la deuxième opération jusqu'à obtenir un arbre de hauteur 6.(Il faut 6 tétraèdres de volume égal à 1/6 pour engendrer le cube)

Par construction, on aura bien l'Arbre A voulu. On appelera les opérations ci-dessus, **les différentes extensions du tétraèdre T.** (il y en a 5)

On a remarqué empiriquement que toutes les découpes contenaient au moins un tétraèdre d'un coin. C'est l'idée clé de notre Projet ! Non seulement les découpes des tétraèdres des coins "engendrent" toutes les découpes possibles mais surtout, les tétraèdres des coins possèdent des premières extensions assez faciles à construire. Nous allons donc appliquer l'algorithme uniquement aux tétraèdres des coins. Par suite, on aura toutes les découpes éventuellement en plusieurs fois.

Partie 3

Notre but maintenant est d'implémenter ces extensions en C++. A la base, on voulait créer une classe arborescence pour stocker les découpes, on a vite renoncé pour la raison suivante : Pour le faire proprement, on devait munir cette structure d'une fonction parent. Cette fonction prend en argument un tétraèdre et renvoie son parent. Le problème étant, que dans une arborescence, tous les noeuds doivent être distincts, sinon un noeud aurait deux parents différents. Dans notre cas, c'est possible (deux découpes obtenues à partir d'un tétraèdre du coin peuvent avoir un ou plusieurs tétraèdres en commun).
A la place, nous représenteront les découpes sous la forme d'un vector de vectors de tétraèdres : vector<vector<Tetra» V où la taille de V est égale le nombre de chemins obtenus (i.e découpes) obtenus à partir d'un tétraèdre du coin T et la taille de V[i] sera égale à 6 : pour les 6 tétraèdres qui forment une découpe. On privilégie les vectors aux tableaux pour une raison importante, on aura souvent besoin de connaître précisement la taille du vector.

La première étape est donc de construire une classe R3 pour représenter les sommets du cube et tétraèdre. Pour cela, on réutilise celui qu'on avait fait en TP puis on la complète.

Explications de R3.hpp

Comme annoncé tout au début, les sommets sont représentés par 3 doubles : x,y,z. dans private.

-On commence avec des constructeurs : un par défaut, un autre à partir de 3 doubles et un par copie.

-On y ajoute une méthode d'affichage.

-On y ajoute une des méthodes pour avoir accès aux composantes.

-On y ajoute opérations + et - vues comme lois internes de l'espace vectoriel R3 et * comme sa loi de externe.

-On y ajoute enfin deux comparateurs pour savoir si deux sommets sont égaux ou non.

Hors de la classe, on y ajoute la fonction **distance** qui prend en argument deux sommets et qui renvoie la distance entre eux.
On ajoute aussi un opérateur booléen **AppartientPas** qui prend en arguments un sommet A et une vector de sommet V et qui renvoie true si A n'est pas dans V. Pour celà, on parcourt le vector V et dès que A y apparaît, on return false, sinon on return true à la fin.

Une fois R3.hpp créée, on peut commencer à construire la classe cube puis Tetra.

Explications de cube.hpp

-On y définit la classe Cube, elle sera reprensentée par ses 8 sommets dans private. On fera attention que l'ordre des sommets est le même que celui annoncé dans la partie 1.
-Dans public, on y ajoute deux contructors : un par défaut et l'autre à partir de huit sommets. On ajoute aussi un moyen pour avoir un accès au i-ème sommet.

Celà peut sembler court comme classe, mais cela nous suffit amplement pour le projet car on a déjà fixé le cube qu'on va utilisé dans la partie 1. Maintenant, on peut construire la classe des tétraèdres.

Explications de Tetra.hpp

Cette classe comporte énormément de lignes de codes, on y a mis beaucoup de fonctions hors de la classe qui nous serviront plus tard.

On représente un tétraèdre sous la forme de quatre R3 dans private.

De manière analogue à R3, on y construit des constructeurs, des opérateurs booléens pour savoir si deux tétraèdres sont égaux et une méthode d'affichage. Attention, on construit les opérateurs booléens de sorte qu'ils vérifient la définition de la partie 1.

Deux tétraèdres sont égaux si leurs sommets qui les composent sont égaux à permutation près. Pour celà on crée deux tableaux de tétraèdres T1 et T2 pour les sommets des tétraèdres. Pour un élément de T1, on regarde bien s'il apparait une et une seule fois dans T2. Le compteur p compte le nombre de fois où un élément de T1 apparait dans T2. Pour que deux tétraèdres soient identiques, il faut que p =4. Remarquons que ce raisonnement ne marche que si les tétraèdes possèdent des sommets tous distincts et ce sera toujours la cas.
Maintenant, hors de la classe, on y ajoute plusieurs fonctions.

La fonction **intersect** prend en argument deux tétraèdes et renvoie un vector de R3 composé des sommets en commun de T1 et T2. Pour celà on parcourt T1 et T2 et on se sert des

opérators booléens de la classe R3.

-La fonction **volume** calcule le volume d'un tétraèdre T1.
-La fonction **volumetotal** calcule le volume total de tous les tétraèdes d'un vecteur de tétraèdre.

La fonction **interface** est une fonction booléenne qui prend en argument deux tétraèdres T1 et T2 et qui renvoie true s'ils sont voisins. On regarde d'abord s'ils ont bien 3 sommets en communs avec la fonction intersect, puis on vérifie la condition de la Proposition 3. Le calcul des déterminants sont faits avec la formule de Sarrus. La première boucle for permet d'avoir les deux sommets distincts de T1 et T2.

La fonction **interuneface** prend en argument un tétraèdre T1 et un vector<Tetra> briques (il représente les briques "élémentaires" pour construire toutes les découpes : les 56 tétraèdres de volume 1/6.) On parcourt le vector briques et on regarde si le ième tétraèdre de briques est voisin avec T1. Pour cela, on utilise la fonction précédente. Avec cette fonction, on pourra avoir les découpes des tétraèdres obtenues avec les tétraèdres des volume 1/3.

La fonction **AppartienPas** est une analogie de celle dans R3 mais cette fois-ci avec des tétraèdres.

La fonction **Sommets** prend en paramètre un vector<tetra> V et renvoie le vector<R3> de tous les sommets de tous les tétras de V.

La fonction **Sommetsdispo** Voici une des fonctions qui nous servira à construire les extensions. Elle prend en paramètres un vector<tetra> V ainsi qu'un cube Cu et renvoie un vector <R3> de tous les sommets de Cu privés de tous les sommets de tous les tétraèdres de V. Pour chaque sommet de Cu, Cu[i], on regarde le nombre de fois ou Cu[i] apparait dans Sommets(V) et on compte avec p. Si p =0, on le garde.

La fonction **interavecuncube** est un booléen qui prend en arguments un tétraèdre T1 et un vector<Tetra> Carre (pour les faces du cube) et qui renvoie true si T1 s'intersecte avec un élément de Carre. Pour chaque élément de Carre noté face, on regarde si face a 3 sommets en commun avec T1.

La fonction **memechemin** est un booléen, qui, comme son nom l'indique, détermine si deux chemins d'une extenion, ie deux découpes sont égales. Là aussi on prend garde, deux découpes sont égales si les tétraèdes qui les composent sont égaux à permuration près. En conséquence, on peut construire une fonction **cheminsansdoublon** qui prend un vector<vector<Tetra» V et qui enlève tous les doublons. On construit initialement avec vector<vector<tetra» un chemin initialement égal à la première découpe de V, on ajoute un par un les élements de V s'il n'apparaissent pas dans chemin, pour cela, on les compte avec p.

Nous avons maintenant tous les outils pour construire la première extension d'un tétraèdre d'un coin. La fonction **Extension1** prend en paramètre un tétra T d'un coin ainsi qu'un vector <tetra> briques (pour les briques élémentaires des 56 tétraèdres de volume 1/6) et renvoie un vector<vector<Tetra» V représentant la première extension de T. On remarquera que chaque chemin sera pour l'instant de longueur 2. Pour cela, On commence par construire t, tous les tétraèdres voisins avec T à l'aide d'interuneface et briques. Puis on met dans V tous les vector<tetra> = (T,test[i]). Cela nous donne la première extension.

Pour construire les prochaines extensions, on devra manipuler les faces des tétraèdres, c'est pourquoi nous construisons une classe face.

Explications de face.hpp

On représente une face par trois R3 (on peut la voir comme un triangle) dans private. Comme pour les classes précédentes, on commence avec des constructeurs : un par défaut, un à partir de trois R3 et un autre par copie. On ajoute aussi l'operator"()", qui crée la i-ème face d'un tétraèdre. On ajoute aussi des moyens d'accéder au i-ème sommet d'une face. Enfin, on ajoute une méthode d'affichage et de façon analogue, un opérator booléeen qui compare deux faces pour savoir si elles sont égales à permutation près des sommets.

Hors de la classe, on ajoute beaucoup de fonctions intermédiaires pour créer les dernières extensions.

La fonction **intersecfaces** prend en arguments deux faces et renvoie un vector de R3 qui correspond aux sommets communs des deux faces. Pour ça, on utilise le booléen de R3 associé.

La fonction **creeTetra** renvoie le tétraèdre formé à partir d'une face et d'un sommet. En pratique, on fera attention à ce que le tétraèdre est dans les briques élémentaires.

Les fonctions **Appartient** et **AppartientPas** vérifient si oui ou on une face F est dans un vector de faces.

La fonction **PasDansUnCarre** prend en arguments une face F et un vector de tétraèdre = Carre , qui représente les faces du cube et renvoie true si F est incluse dans les faces du cubes. On considère deux compteurs. Le premier, p qui compte le nombre de sommets communs entres les sommets de F et les sommets pour chaque élément de Carre. q compte le nombre de fois où p vaut 3 (ie où le nombre de fois où la face est incluse dans une face du cube).

La fonction **facetetra** renvoie un vector face qui représente toutes les faces d'un tétraèdre T.

La fonction **facestetras** renvoie un vector face qui représente toutes les faces de tous les tétraèdres d'un vector<Tetra>

La fonction **inters** renvoie la face commune de deux tétraèdres voisins. On prendra garde de l'utiliser qu'avec des tétraèdres voisins.

La fonction **allintersection** prend en arguments un vecteur de tétraèdres T et renvoie un vector<Face> de toutes les faces communes de tous les tétras voisins de T. On se sert du booléen interface défini dans la Tetra.hpp.

La fonction **facedispo** prend en arguments un vector <Tetra> T et un vector <Tetra> Carre (pour les faces du cube) et renvoie un vector <Face> des faces "utilisables" (cf les prochaines extensions). Moralement, les faces "utilisables" seront les faces Fi de tous les tétraèdres de T qui vérifient les conditions suivantes :
-Les Fi ne sont pas des faces inclusent dans les faces du cube.
-Les Fi ne sont pas des faces communes de deux tétraèdres voisins de T.

On a maintenant tout ce qu'on a besoin pour attaquer les dernières extensions.

La fonction **Extension2** prend en paramètres un vector<vector<Tetra» Extension (en pratique ce sera le résultat d'Extension1 d'un tétra du coin T), un vector <Tetra> carre pour les faces du cube et enfin le cube Cu, et renvoie l'extension2 de T. Il faut imaginer qu'à ce moment, on a fixé un tétraèdre du coin, auquel on lui a ajouté un de ses voisins. On cherche maintenant de savoir comment ajouter un autre tétraèdre à cette découpe. Ici, il suffit juste de prendre une face "utilisable" de toutes les faces des deux tétraèdres ainsi qu'un sommet disponible. Un sommet disponible sera un sommet de cube privé des sommets formant les deux tétraèdres. Ce raisonnement ne marche uniquement car on est parti d'un tétraèdre du coin. Voilà comment faire la fonction extension2.

Après avoir appliqué extension1 à notre tétraèdre du coin pour obtenir tous les différents tétraèdres constructibles à partir du premier, on applique extension2 afin de créer pour chaque chemin de extension1 un nouveau tétraèdre. Pour cela on regarde pour chaque chemin de extension1,les faces et sommets disponible grâce aux fonctions "facedispo" et "Sommetsdispo", et pour chaque nouveau tétraèdre qui peut être construit à partir d'une face et d'un sommet possible on va crée un nouveau chemin qui reprendra le précedent chemin et va ajouter le nouveau tétraèdre.
A la fin de extension2 on se retrouve avec uniquement des chemins de longueur 3.

De même que extension2, **Extension3** va ajouter un nouveau tétraèdre avec en plus comme condtion que le nouveau ne s'intersecte pas avec ceux déja crée, condition vérifié avec la fonction interface. A la fin de extension3,nous nous retrouvons avec des chemins de longueur 4.

On s'attaque maintenant à **Extension4** :

Ajouter un 5-ième tétraèdre est plus délicat, c'est pourquoi dans extension4 il faut distinguer plusieurs cas en fonction des chemins obtenus avec extension3 :

cas1 : il nous reste 2 faces et 1 sommet, on a alors le choix pour construire différents types de tétraèdre :

1.1-un tétraèdre formé par une des faces et le sommet pour cela on applique la fonction "creeTetra".
1.2-un tétraèdre formé par les deux faces, on cherche alors les 2 points d'intersection des faces ainsi que les 2 points distincts et on crée le tétra à partir de ces points.
1.3-un tétraèdre coin, pour cela on prend les 2 points distincts des deux faces, à cela on ajoute le sommet disponible et on complète par un des deux points d'intersection des deux faces, le choix du points se fera en comparant les 2 tétraèdres obtenus avec tetracoin.

cas2 : il nous reste 3 faces de disponible et un sommet, cela implique qu'on peut construire un tétraèdre du coin et c'est ce qu'on fait car c'est le tétraèdre le plus simple à construire. Pour cela on crée les 3 trois tétraèdres qui partent chacuns d'une des faces et qui ont pour sommet le dernier sommet et on utilise "Appartient" pour déterminer lequel est un tétraèdre coin. Il ne nous manque plus qu'un tétraèdre à construire.

Finalement extension5 va nous permettre de construire le dernier tétraèdre.Il ny'a que 3 cas possible :

cas1 : il nous reste 1 face et 1 sommets donc on construit le tétraèdre grâce à la fonction creeTetra.
cas2 : 2 faces restantes,on contruit le tétraèdre induit par ces 2 faces, de la même facon que dans extension4 on prend les deux points d'intersection et on complète avec les deux points distincts et on obtient notre dernier tétraèdre.
cas3 : 3 faces restantes donc forcément le dernier tétraèdre contient ces 3 faces.Pour le construire on prend une face au hasard (dans notre cas la 1ere face) et on trouve le dernier point en regardant lequel des deux points d'intersection des deux faces restantes n'appartient à la 1ere face.
Nous avons donc au final obtenu toutes les découpes possible à partir d'un tétraèdres du coin

La fonction Decoupe permet d'appliquer tout les extensions à un tétraèdre afin d'obtenir tout les découpes à partir de ce dernier.

On a testé la fonction decoupe pour tous les tétraèdres des coins. A la toute fin, on obtient que 68 découpes (sans compter celles engendrées par les tétraèdres de volume 1/3). Il y a deux hypothèses : la plus probable, une ou plusieurs de nos extensions sont fausses ou soit les tétraèdres des coins n'engendrent pas toutes les découpes. Nous n'avons malheureusement pas le temps de corriger. Par ailleurs, nous ne savons pas si toutes nos découpes sont correctes.

Explications de MainFinal.cpp

Tout d'abord, le lecteur s'est peut-être demandé qu'est-ce que le fichier combinaisons.txt ? Nous y répondons maintenant. Nous n'avons pas réussi à génerer les $\binom{8}{4} = 70$ combinaisons pour générer nos tétraèdres à partir des sommets du cube. A la place, nous sommes allés sur le site https ://www.dcode.fr/combinaisons qui générait directement les combinaisons qu'on voulait, on a ensuite copier-coller les résultats sur ce fichier.txt et nous allons l'utiliser pour générer nos tétraèdres.
Soit N la N-eme decoupe voulu, on demandera au lecteur de l'entrer après avoir compiler ce fichier puis exécuté.
On construit les sommets de notre cube, puis le cube lui-même comme définis dans la partie 1. On génère ensuite les 70 quadruplets distincs de sommets du Cube en lisant le fichier combinaison.txt. Le lecteur s'assurera de bien lire ce fichier lors de la compilation. On peut maintenant séparer les 58 tétraèdres des carrés et des rectangles en vertu de la proposition2. On en profite aussi pour créer les deux tétraèdres de volume 1/3. En conséquence, on obtient les 56 tétraèdres de volume 1/6.

On génère maintenant les tétraèdres des coins, puis on les parcourt tous en leur appliquant la fonction Decoupe. On obtient ainsi un vector<vecto<Tetra» de toutes les découpes obtenues à partir des tétraèdres des coins avec des doublons. Enfin, on enlève les doublons et on rajoute les découpes obtenues avec les tétraèdres de volume 1/3.

On va réprésenter la N-eme découpe dans un fichier.txt pour l'afficher avec gnuplot. On affichera aussi les tétraèdres dans le terminal pour comparer.