|  |  |
| --- | --- |
| C:\Users\JMG\Desktop\Divers\Logos ISEN\logo_isen_alpha.png  **Institut Supérieur de l’Électronique**  **et du Numérique**  *Tél : 00.33.2.98.03.84.00*  *Fax : 00.33.2.98.03.84.10*  20, Rue du Cuirassé Bretagne  29200 Brest - FRANCE | N4 **Année universitaire 2010-2011** |

**Amélioration des performances d’un modeleur**

**géométrique basé sur le logiciel ROOT**

par

Jean-Marie GUYADER

[jean-marie.guyader@isen.fr](mailto:jean-marie.guyader@isen.fr)

réalisé au

**CERN**

*Organisation Européenne pour la Recherche Nucléaire*

[](http://upload.wikimedia.org/wikipedia/en/a/ae/CERN_logo.svg)

du

7 juin and 2 septembre 2011

Travail supervisé par : M. Andrei GHEATA, Bureau № 12/1-007

CERN

CH-1211 Genève 23 – SUISSE

[andrei.gheata@cern.ch](mailto:andrei.gheata@cern.ch)

Contact ISEN : Mme Dominique MARATRAY

ISEN Brest

20, Rue du Cuirassé Bretagne

29200 Brest – FRANCE

**Sommaire**

[1. Introduction 5](#_Toc305622559)

[2. Opérations booléennes sur des solides - Justification du projet 7](#_Toc305622560)

[3. Présentation de la librairie USolids 8](#_Toc305622561)

[3. 1. Généralités 8](#_Toc305622562)

[3. 2. Présentation de la classe UMultiUnion 8](#_Toc305622563)

[3. 3. Fonctions de base de la librairie 9](#_Toc305622564)

[3. 4. Analyse fonctionnelle pour les solides 10](#_Toc305622565)

[3. 5. Autres classes 12](#_Toc305622566)

[3. 6. Notion de complexité d’un algorithme 13](#_Toc305622567)

[4. Voxélisation d’une instance de la classe UMultiUnion 14](#_Toc305622568)

[4. 1. Utilité de la voxélisation 14](#_Toc305622569)

[4. 2. Analyse fonctionnelle pour la voxélisation 15](#_Toc305622570)

[4. 3. Procédure de voxélisation 16](#_Toc305622571)

[4. 4. Détermination des nœuds candidats dans un voxel donné 23](#_Toc305622572)

[5. Opérations sur une instance de UMultiUnion 25](#_Toc305622573)

[5. 1. Description des principales méthodes implémentées 25](#_Toc305622574)

[5. 1. 1. Méthode *Inside* 25](#_Toc305622575)

[5. 1. 2. Méthode SafetyFromInside 26](#_Toc305622576)

[5. 1. 3. Méthode SafetyFromOutside 26](#_Toc305622577)

[5. 1. 4. Méthode DistanceToOut 27](#_Toc305622578)

[5. 1. 5. Méthode Normal 28](#_Toc305622579)

[5. 2. Test des algorithmes 28](#_Toc305622580)

[5. 2. 1. Test d’efficacité de la voxélisation, méthode Inside 28](#_Toc305622581)

[5. 2. 2. Autres résultats 31](#_Toc305622582)

[6. Conclusion 32](#_Toc305622583)

[7. Références bibliographiques 33](#_Toc305622584)

[8. Crédits photographiques 33](#_Toc305622585)

[9. Annexes 33](#_Toc305622586)

**Remerciements**

Je voudrais adresser mes remerciements sincères à mon superviseur, M. Andrei GHEATA pour sa grande disponibilité, ses nombreux conseils toujours avisés, ainsi que pour l’aide qu’il a su m’apporter pendant ce stage.

Merci également à M. John APOSTOLAKIS, M. Gabriele COSMO, M. Marek GAYER et Mme Tatiana NIKITINA, membres du groupe de travail USolids, pour leurs indications très utiles ainsi que pour leur appui durant ce stage.

Ma reconnaissance va enfin vers M. Marc FAUDEIL et M. Jean-Yves MULOT, qui ont eu l’amabilité de soutenir ma candidature au programme d’été du CERN.

\*

\* \*

**Sigles et abréviations**

→ *CERN* : Organisation européenne pour la recherche nucléaire ;

→ *LHC* : Large Hadron Collider ;

→ *ALICE* : A Large Ion Collider Experiment ;

→ *GEANT* : GEometry And Tracking ;

→ *CSG* : Construction Solid Geometry ;

→ *ATLAS* : A Toroidal LHC ApparatuS;

→ *CMS* : Compact Muon Solenoid;

→ *LHCb* : Large Hadron Collider beauty experiment;

→ *SPS* : Super Proton Synchrotron.

1. Introduction

L’Organisation européenne pour la recherche nucléaire, plus souvent connue sous l’acronyme CERN, est le plus grand laboratoire de physique des particules au monde. Le CERN voit le jour au sortir de la seconde guerre mondiale, sur une idée du scientifique français Louis de Broglie et par la volonté de 12 pays européens : la République Fédérale d’Allemagne, le Royaume de Belgique, le Royaume du Danemark, la République française, Le Royaume-Uni de Grande-Bretagne et d’Irlande du Nord, la République hellénique, la République italienne, le Royaume de Norvège, le Royaume des Pays-Bas, le Royaume de Suède, la Confédération suisse et la République fédérative socialiste de Yougoslavie. À l’heure actuelle, le CERN compte 20 états membres. Ces derniers contribuent financièrement au budget et participent au processus de prise de décision de l’Organisation. Le directeur général du CERN est actuellement l’allemand Rolf Heuer.

Par ailleurs, en plus des états membres, six pays possèdent le titre d’observateurs. Il s’agit de la République de Turquie, de l’État d’Israël, de la Fédération de Russie, du Japon, des États-Unis d’Amérique et de la République d’Inde. L’UNESCO ainsi que la Commission Européenne sont également observateurs au sein de l’organisation.

Le CERN emploie environ 2400 personnes. De plus, un nombre très important de scientifiques du monde entier participent d’une façon ou d’une autre aux travaux menés au CERN.



Figure 1 Pays fondateurs du CERN (1954)

[](http://upload.wikimedia.org/wikipedia/commons/f/f4/CERN1999.png)

Figure 2 Pays membres (2011)

En 1954, la convention mettant en place le CERN stipule que : *l’Organisation assure la collaboration entre États européens pour les recherches nucléaires de caractère purement scientifique et fondamental.* Par ailleurs, il est précisé que : *l’Organisation s’abstient de toute activité à fins militaires et les résultats de ses travaux expérimentaux et théoriques sont publiés ou de toute autre façon rendus généralement accessibles*.

Le laboratoire européen pour la physique des particules, dont le siège est localisé dans la commune suisse de Meyrin, s’étale à la frontière franco-suisse, entre la République et Canton de Genève (CH) d’une part, et le département de l’Ain (F) d’autre part. Le CERN, pour partie, doit sa renommée au LHC (*Large Hadron Collider*ou Grand collisionneur de hadrons). Il s’agit du plus puissant accélérateur de particules jamais construit au monde. D’une circonférence de 27 kilomètres, il est capable d’accélérer très fortement (jusqu’à environ 7 TeV) deux faisceaux de particules dans des directions opposées, à 99,9 % de la vitesse de la lumière.

La carte ci-dessous présente la localisation des équipements du CERN dans la région genevoise :



**8**

**5**

**2**

**1**

**CERN - *Site de Meyrin* (CH)**

**CERN - *Site de Prévessin* (F)**

**GENÈVE (CH)**

**Meyrin (CH)**

**Saint-Genis-Pouilly (F)**

**SPS**

**LHC**

Figure 3 Installations du CERN dans la région genevoise

Les collisions mises en œuvre au sein du LHC induisent la création de nouvelles particules. Par conséquent, afin d’étudier les phénomènes en jeu lors des expérimentations, il est nécessaire d’utiliser des dispositifs de détection. Quatre grands détecteurs sont actuellement en service sur le parcours du LHC (cf. points jaunes sur la carte). Il s’agit de :

\* point 1 - ATLAS ;

\* point 2 - ALICE ;

\* point 5 - CMS ;

\* point 8 - LHCb.

 L’expérience à laquelle est rattachée la présente étude est ALICE (*A Large Ion Collider Experiment*, ou en français, une expérience sur le grand collisionneur d’ions). Le but de cette expérience est de mieux comprendre certains aspects et phénomènes physiques fondamentaux encore mal appréhendés aujourd'hui. Le détecteur ALICE est gigantesque et extrêmement complexe : il mesure 16 mètres de hauteur, 26 mètres de longueur et pèse environ 10 000 tonnes.

Figure 4 Vue extérieure du détecteur ALICE

Au sein de l’expérience ALICE, le groupe ALICE-Offline a pour tâche de développer la structure logicielle permettant la simulation, la reconstruction, la visualisation, le traitement et l’analyse des grandes quantités de données brutes générées par les expérimentations de physique des particules. C’est à cet effet que, depuis 1994, le logiciel ROOT a été développé au CERN. D’autres chercheurs du CERN développent une plateforme permettant de simuler le passage de particules à travers la matière : il s’agit du logiciel GEANT4.

Parmi la nombreuse gamme d’outils offerts par ces deux logiciels, des instruments analogues peuvent être trouvés. GEANT4 et ROOT possèdent en effet tous les deux des paquetages géométriques permettant des calculs complexes sur des volumes en trois dimensions. De ce fait, il a été décidé de regrouper les fonctionnalités similaires au sein d’un seul et même paquetage commun appelé USolids (pour Universal Solids).

Ce document rend compte des travaux effectués pendant trois mois au sein du groupe de recherche USolids. Le but du projet est de créer un nouveau type de solide, correspondant à l’union booléenne de plusieurs sous-solides, puis d’implémenter des outils permettant une analyse performante de la géométrie créée, au sein desquels les techniques de voxélisation auront une place de choix. Les travaux sont réalisés en langage C++.

1. Opérations booléennes sur des solides - Justification du projet

Les paquetages géométriques utilisés par ROOT et GEANT4 permettent de pratiquer des opérations booléennes entre des solides. Les trois opérations booléennes de base sont l’union, l’intersection et la différence. On réalise ce que l’on appelle une géométrie de construction de solides (en anglais, *Constructive Solid Geometry*).

En guise d’exemple, considérons une sphère et un cube, formes auxquelles on applique les trois opérations booléennes de base :

Union

Soustraction

Intersection

cube  sphère

cube – sphère

cube  sphère







N.-B. : On remarquera que dans les cas de l’union et de l’intersection, il n’est pas important de savoir de quels côtés de l’opérateur se situent les solides, ce qui n’est pas vrai pour la soustraction.

Pour la création d’un solide complexe, les modeleurs de GEANT4 et ROOT construisent un arbre binaire de nœuds booléens. Néanmoins, la navigation au sein d’une telle structure devient rapidement prohibitive lorsque le nombre de nœuds devient important. Les travaux présentés dans ce document se focalisent donc, sur la création d’un solide représentant l’union booléenne de plusieurs autres solides appelés nœuds. L’intérêt d’une telle structure est justifié par la flexibilité qu’elle offrirait dans la description des détecteurs (ALICE, notamment).

1. Présentation de la librairie USolids
   1. Généralités

Un des intérêts premiers de la librairie USolids est permettre la création de formes 3D. Chaque sorte de solides (e.g. parallélépipède, sphère, polyèdre) doit dériver d’une classe mère appelée *VUSolid*. Les contributeurs de la librairie USolid avaient déjà, avant le début du projet dont il est rendu compte dans ce document, créé cette classe *VUSolid*, ainsi que la classe fille *UBox*, permettant l’instanciation de solides ayant la forme d’un parallélépipède rectangle. En outre, un certain nombre d’autres classes à visée plus utilitaire étaient aussi implémentées. Ces dernières permettent notamment de définir des translations, des rotations, des points, des vecteurs dans l’espace en trois dimensions, ou encore d’utiliser des applications mathématiques courantes.

La classe permettant d’utiliser une géométrie basée sur l’union booléenne de plusieurs solides est appelée *UMultiUnion*. Une instance de cette dernière classe sera bâtie par l’ajout de plusieurs sous-solides appelés nœuds. Compte tenu de l’état d’avancement du projet, seuls des nœuds sous forme de boîte (*UBox*) pourront, dans un premier temps, être utilisés pour former un objet de classe *UMultiUnion*.



Figure 5 Héritage depuis la classe mère VUSolid

* 1. Présentation de la classe UMultiUnion

Une instance de la classe *UMultiUnion* est caractérisée par un tableau de nœuds. Chacun de ces nœuds est construit à partir d’un solide intrinsèque issu d’une classe fille de *VUSolid*, ainsi que par une instance de la classe *UTransform3D* (translations / rotations). Par conséquent, il est envisageable de construire une structure de classe *UMultiUnion* en utilisant des solides *UBox*, *UTube*, mais également des instances de *UMultiUnion* (structures imbriquées). Voici donc une représentation de l’architecture envisagée :



Figure 6 Diagramme synoptique de la classe UMultiUnion

* 1. Fonctions de base de la librairie

La librairie *USolids* doit être en mesure d’effectuer des traitements complexes sur les structures 3D définies en son sein. On en présente ici un aperçu concis en deux dimensions.

La description des méthodes ci-dessous est simplifiée :



1) DistanceToOut :

Cette méthode doit, quand on lui donne un point et une direction, déterminer la distance entre le point et la surface du solide suivant la direction fixée ;

2) DistanceToIn :

Idem, mais avec un point situé au dehors du solide ;

3) SafetyFromInside / SafetyFromOutside :

On peut comprendre les méthodes safety comme étant analogues aux méthodes DistanceToIn et DistanceToOut, à ceci près qu’on ne précise aucune direction particulière ;

4) Extent :

Détermine l’extension du solide sous la forme d’une boîte ;

5) Normal :

Cette méthode a pour objectif de déterminer la normale à un point.

Figure 7 Illustration des fonctions de base de la librairie

En outre, une méthode Inside déterminant si un point se trouve dans, au dehors ou sur la surface de la structure devra être implémentée.

Toutes ces méthodes devront naturellement être applicables à une instance de la classe *UMultiUnion*. Ci-dessous, on prend l'exemple d’un objet comportant trois sous-solides numérotés de 0 à 2 :

Figure 8 Application à un exemple de structure UMultiUnion

Extent suivant x

DistanceToIn

x

y

**0**

**2**

**1**

Extent suivant y

Normal

**eOutside**

**eInside**

**eSurface**

* 1. Analyse fonctionnelle pour les solides

Ci-dessous, on présente un diagramme UML simplifié relatif aux classes utilisées pour construire des solides. Les prototypes et noms des méthodes avaient déjà été précédemment fixés par le groupe USolids.

|  |
| --- |
| **VUSolid** |
| Attributs :  => *Privés :*  **UString** fName // nom du solide  **UBBox** \*fBBox // pointeur vers la bounding box du solide  => *Protégés :*  **static** **double** fgTolerance |
| Méthodes :  => *Protégées :*  **virtual** **void** ComputeBBox(**UBBox** \*aBox, **bool** aStore)  // calcule la bounding box associée au solide passé en argument  => *Publiques :*  **virtual** **EnumInside** Inside(**const** UVector3 &aPoint)  **virtual** **double** SafetyFromInside(**const** UVector3 aPoint,  **bool** aAccurate)  **virtual** **double** SafetyFromOutside(**const** UVector3 aPoint,  **bool** aAccurate)  **virtual** **double** DistanceToIn(**const** UVector3 &aPoint,  **const** UVector3 &aDirection,  **double** aPstep)  **virtual** **double** DistanceToOut(**const** UVector3 &aPoint,  **const** UVector3 &aDirection,  UVector3 &aNormalVector,  **bool** convex,  **double** aPstep)  **virtual** **bool** Normal(**const** UVector3 &aPoint, UVector3 &aNormal)  **virtual** **void** Extent(**EAxisType** aAxis, **double** &aMin, **double** &aMax)  **virtual** **void** Extent(**double** aMin[3], **double** aMax[3])  **virtual** UGeometryType GetEntityType()  N.-B. : bien vouloir se référer aux schémas des pages précédentes pour avoir une description du comportement des méthodes ci-dessus |

*HÉRITAGE*

|  |
| --- |
| **UBox** |
| Attributs :  => *Privés :*  **double** fDx // demi-longueur suivant x  **double** fDy // demi-longueur suivant y  cf. (a) – page suivante  **double** fDz // demi-longueur suivant z |
| Méthodes :  => *Publiques :*  UBox()  UBox(**const** **char** \*name, **double** dx,  **double** dy,  **double** dz) |

(a)

|  |
| --- |
| **UMultiUnion** |
| Attributs :  => *Privés :*  **vector**<UNode\*> \*fNodes // tableau contenant les pointeurs vers  // chacun des sous-solides  UVoxelFinder \*fVoxels  // pointeur vers la structure voxélisée  **class** UNode  {  VUSolid \*fSolid ;  UTransform3D \*fTransform ;  }  // classe permettant de stocker des nœuds (solides intrinsèques + transformations (translation / rotation. À un nœud correspond un sous-solide d’une instance de la classe UMultiUnion |
| Méthodes :  => *Publiques :*  **void** Voxelize()  // méthode permettant la voxélisation de l’instance UMultiUnion  **int** GetNumNodes()  // méthode retournant le nombre de nœuds qui composent l’instance de UMultiUnion  VUSolid\* GetSolid(**int** index)  // méthode qui retourne le pointeur vers le solide intrinsèque du index i-ème nœud  UTransform3D\* GetTransform(**int** index)  // méthode qui retourne le pointeur vers la transformation du index-ième nœud  **void** SetVoxelFinder(UVoxelFinder\* finder)  // méthode permettant de mémoriser la structure voxélisée dans l’instance courante de la classe UMultiUnion  **void** Voxelize()  // permet la voxélisation de la structure en utilisant la classe UVoxelFinder |

Dans le cas de la classe *UBox*, les méthodes virtuelles de la classe mère *VUSolid* avaient déjà été implémentées par le groupe de travail USolids. Un des buts du projet est donc d’implémenter ces mêmes fonctions, mais dans le cas de la classe *UMultiUnion*. Voici plus en détail le fonctionnement de ces méthodes dans ce dernier cas, avec l’exemple à trois nœuds donné à la page 9 :

**void** Extent(EAxisType aAxis, **doble** &aMin, **double** &aMax)

Méthode permettant de déterminer l’extension maximale du solide suivant l’axe *aAxis*. Renvoi des valeurs calculées par référence

**void** Extent(**double** aMin[3], **double** aMax[3])

Idem, mais sur les trois axes

EnumInside Inside(**const** UVector3 &aPoint) **const**

En fonction de la localisation de *aPoint* par rapport à la structure de type *UMultiUnion*, la méthode Inside retourne eInside (cf. croix oranges), eOutside (cf. croix violettes) ou eSurface (cf. croix roses)

**bool** Normal(**const** UVector3 &aPoint, UVector3 &aNormal)

Détermination de la normale à un point par rapport au solide de classe *UMultiUnion*. Le vecteur normal est retourné sous la forme d’un vecteur unitaire. Dans le cas où le point n’est pas localisé sur une surface, on trouve le solide le plus proche. L’appel de la fonction *Normal* pour ce solide et le point considéré donne le résultat voulu. La fonction renvoie *true* si le point considéré est assez proche d’une surface (cf. croix et flèche noire)

**double** DistanceToIn(**const** UVector3 &aPoint,

**const** UVector3 &aDirection,

**double** aPstep)

Détermination de la distance depuis un point (normalement situé en dehors du solide) jusqu’à la surface la plus proche, en suivant la direction donnée. Sur le schéma ci-dessus, le point de départ est le point vert et la direction imposée est la flèche verte. La distance retournée elle celle allant du point à la fin des pointillés verts

**double** DistanceToOut(**const** UVector3 &aPoint,

**const** UVector3 &aDirection,

UVector3 &aNormalVector,

**bool** convex,

**double** aPstep)

Détermination de la distance à partir d’un point (normalement situé dans le solide) jusqu’à la surface la plus proche, en suivant la direction donnée. La normale au point de croisement est renvoyée par référence

**double** SafetyFromInside(**const** UVector3 &aPoint

**bool** aAccurate) **const**

Estimation de la distance d’un point contenu dans la structure de type *UMultiUnion* à la surface du solide. La quantité retournée est, en général, sous-évaluée par rapport à la valeur réelle (rapidité d’exécution). aAccurate permet de sélectionner un mode lent mais efficace (true) ou un mode rapide mais plus approximatif (false)

**double** SafetyFromOutside(**const** UVector3 &aPoint

**bool** aAccurate) **const**

Estimation de la distance d’un point situé en dehors de la structure de type *UMultiUnion* à la surface du solide le plus proche

* 1. Autres classes

D’autres classes ont par ailleurs été définies. Il s’agit de :

\* *UUtils*: dans cette classe sont définies des constantes mathématiques ainsi qu’un assortiment de fonctions utiles pour l’implémentation des autres classes. Il peut s’agir aussi bien des fonctions trigonométriques (cosinus, sinus, tangente), que de la valeur absolue ou encore des fonctions retournant un maximum et un minimum ;

\* *UVector3*: cette classe permet de manipuler facilement des vecteurs contenant trois nombres décimaux (assignation, normalisation, rotations, etc.) ;

\* *UTransform3D*: cette classe permet de définir des transformations caractérisées par une translation et une rotation. Cela est notamment utile pour placer les solides en des positions différentes de l’espace. De façon commode, cette classe permet de définir les translations par un couple de trois nombres décimaux, et les rotations par les trois angles d’Euler.

La figure ci-contre présente comment s’effectue une rotation en utilisant les angles d’Euler. Trois angles *ψ*, *φ* et *θ* suffisent à définir complètement la rotation.

x

y

z

u

v

w

z’

x’

y’

**2**

**1**

**3**

*ψ*

*ψ*

*φ*

*φ*

*θ*

*θ*

On attribue souvent un nom à ces trois angles :

\* *ψ* : précession ;

\* *φ* : nutation ;

\* *θ* : rotation propre.

Figure 9 Angles d’Euler

Pendant la phase de développement, il doit être possible de tester le comportement de la librairie *USolids* avec les classes *TGeoShape* de ROOT et *G4USolid* de GEANT4. On a l’architecture suivante.

Figure 10 Utilisation de bridge classes

VUSolid

UBox

UTube

UMultiUnion

TGeoShape

TGeoBBox

TGeoTube

TGeoUShape

G4

G4Box

G4USolid

**ROOT**

**GEANT4**

**USolids**

Comme on peut le voir sur le schéma ci-dessus, pour tester le comportement des objets des classes dérivées de *VUSolid*, on passe par des classes faisant le pont (*Bridge Classes*, en anglais) entre les objets liés à ROOT ou GEANT4 d’une part, et *VUSolid*, d’autre part. Il s’agit des classes *TGeoUShape* et *G4USolid*.

* 1. Notion de complexité d’un algorithme

Dans de nombreuses applications, il est crucial que les algorithmes mis en œuvre aient une exécution rapide. C’est le cas dans notre cadre d’étude.

Afin de pouvoir apprécier l’efficacité d’un algorithme, on définit souvent la notion de complexité. Cette dernière rend compte de la durée d’exécution d’un algorithme, en fonction d’une quantité caractéristique. Cela permet de comparer l’algorithme étudié avec un ou plusieurs autres réalisant des fonctionnalités analogues.

On parle souvent de plusieurs types de complexité : constante, logarithmique, linéaire, quadratique, exponentielle, etc. La complexité est, en général, détériorée par l’utilisation d’un nombre trop important de boucles dans l’algorithme, ou à leur utilisation non judicieuse. Dans notre cas, il faudrait tendre à obtenir une complexité de type logarithmique. En effet, dans ce cas, la durée d’exécution varie légèrement lorsqu’augmente la quantité caractéristique. Dans le cas d’une complexité linéaire, la durée d’exécution est directement proportionnelle à la quantité caractéristique.

σ(log(n))

σ(1)



n

*Complexité constante*

σ(n)



n



n

*Complexité logarithmique*

σ(n²)



n

*Complexité linéaire Complexité quadratique*

1. Voxélisation d’une instance de la classe UMultiUnion
   1. Utilité de la voxélisation

Dans un souci d’optimisation, les algorithmes mis en œuvre dans le cadre du projet *USolids* doivent être performants pour ce qui est de la rapidité d’exécution. Pour ce faire, il a été envisagé d’utiliser des techniques de voxélisation. Ces dernières permettent notamment de connaître quels sous-solides sont localisés dans une partie délimitée de l’espace.

En considérant un point déterminé par ses coordonnées x, y et z, on est à même de savoir dans quel voxel il se trouve. Au préalable, on aura mémorisé quels nœuds sont situés dans chacun des voxels. Au final, cela permet d’effectuer des traitements sur un nombre plus faible de nœuds dans la mesure où l’on n’a plus besoin d’analyser tous les nœuds de la structure, mais seulement des nœuds candidats.

* 1. Analyse fonctionnelle pour la voxélisation

On donne maintenant le diagramme des classes qui a été créé dans le cadre de la voxélisation :

|  |
| --- |
| **UVoxelFinder** |
| Attributs :  => *Privés :*  UMultiUnion \*fMultiUnion // Solide devant être voxélisé  **double** \*fBoxes // Tableau contenant les limites des bounding  // boxes de chaque noeud  **double** \*fBoundaries // Tableau contenant le quadrillage non  // ordonné et non trié, issu des bounding  // boxes  **double** \*fXSortedBoundaries // Quadrillage trié suivant x  **int** fXNumBound // Nombre de limites suivant x  **double** \*fYSortedBoundaries // Quadrillage trié suivant y  **int** fYNumBound // Nombre de limites suivant y  **double** \*fZSortedBoundaries // Quadrillage trié suivant z  **int** fZNumBound // Nombre de limites suivant z  **int** \*fMemoryX // Ces trois tableaux permettent de stocker  **int** \*fMemoryY // quels sont les nœuds qui sont situés  **int** \*fMemoryZ // dans une tranche suivant x, y ou z  **int** fNx, fNy, fNz // Nombre de nœuds dans chaque tranche  **double** fTolerance // Distance minimale permettant de discriminer  // deux limites proches mais non identiques |
| Méthodes :  => *Privées* :  **void** GetCandidatesAsString()  => *Publiques* :  **void** Voxelize() // Méthode regroupant les 4 suivantes  **void** BuildVoxelLimits() // Stockage des coordonnées de l’origine  // ainsi que des demi-longueurs de  // chaque noeud  **void** CreateBoundaries() // Le but de cette méthode est de  // déterminer les limites issues des  // bounding boxes déterminées par  // BuildVoxelLimits. Ces limites sont  // stockées dans le tableau fBoundaries  **void** SortBoundaries() // Cette méthode ordonne les limites  // suivant chaque axe (ordre croissant) et  // supprime les limites excédentaires  // quand plusieurs sont trop rapprochées.  // Les limites triées sont stockées dans  // les tableaux fXSortedBoundaries,  // fYSortedBoundaries, fZSortedBoundaries  **void** BuildListNodes() // Cette methode permet de stocker dans  // les tableaux fMemoryX, fMemoryY et  // fMemoryZ les solides présents dans  // chaque tranche suivant l’axe considéré  vector<**int**> GetCandidatesVoxelArray(UVector3 point)  // Cette méthode permet de déterminer quels solides sont présents dans le voxel caractérisé par les trois index passés en arguments (candidats)  vector<**int**> Intersect(**char**\* mask)  // Cette méthode retourne une liste de nœuds correspondant au tableau de bits passé  **double** GetBoxes() // Méthode retournant le pointeur du tableau  // contenant les caractéristiques des  // bounding boxes associées à chaque nœud |

* 1. Procédure de voxélisation

Voici les grandes étapes qui ont été suivies pour voxéliser une instance de la classe *UMultiUnion*. On utilise, pour l’exemple un cas en deux dimensions seulement.

La voxélisation est opérée par la méthode Voxelize (cf. **annexe p. 34**), cette dernière étant décomposée en plusieurs sous fonctions, que l’on s’attache à décrire dans cette partie.

*1) Structure de base :*

À titre d’exemple on considère une structure comportant huit nœuds, ces derniers étant de forme circulaire :



Figure 11 Disposition initiale des nœuds

Les opérations de voxélisation sont réalisées au sein de la classe *UVoxelFinder*. Il est avant tout nécessaire de stocker l’adresse (pointeur) du solide à voxéliser parmi les données membres d’une instance de la classe *UVoxelFinder* : il s’agit de *fMultiUnion* (de type *UMultiUnion*).

*2) Calcul, dans le repère global, des coordonnées des bounding boxes de chaque nœud (coordonnées du centre et demi-longueurs associées à chaque axe) :*

→ cf. **annexe p. 34**, méthode BuildVoxelLimits

Ci-dessous, les carrés verts représentent les bounding boxes associées à chacun des nœuds de la structure.



Figure 12 Bounding boxes (en vert) associées à chaque noeud

La méthode *BuildVoxelLimits* permet de mémoriser les coordonnées suivantes pour chacun des nœuds : - les coordonnées dans le repère global du centre de la bounding box ;

- les coordonnées des trois demi-longueurs associées.

Cela fait donc six quantités à stocker pour chaque nœud, ces dernières étant mémorisées dans le tableau *fBoxes* (tableau de double de dimension six fois le nombre de nœuds).

*3) Détermination des limites sur chacun des axes, à partir des nœuds :*

→ cf. **annexe p. 34**, méthode CreateBoundaries

À partir des grandeurs stockées dans le tableau *fBoxes*, il est possible de stocker dans un autre tableau les coordonnées de chacune des extrémités des bounding boxes. En utilisant la méthode CreateBoundaries, on obtient dans le tableau *fBoundaries*, une sorte de quadrillage irrégulier. On utilise l’algorithme suivant pour chacun des nœuds (*iIndex* représente le numéro du sous-solide considéré) :

// Tranches suivant x :

fBoundaries[2\*iIndex] = fBoxes[6\*iIndex+3]-fBoxes[6\*iIndex];

fBoundaries[2\*iIndex+1] = fBoxes[6\*iIndex+3]+fBoxes[6\*iIndex];

// Tranches suivant y :

fBoundaries[2\*iIndex+2\*carNodes] = fBoxes[6\*iIndex+4]-fBoxes[6\*iIndex+1];

fBoundaries[2\*iIndex+2\*carNodes+1] = fBoxes[6\*iIndex+4]+fBoxes[6\*iIndex+1];

// Tranches suivant z :

fBoundaries[2\*iIndex+4\*carNodes] = fBoxes[6\*iIndex+5]-fBoxes[6\*iIndex+2];

fBoundaries[2\*iIndex+4\*carNodes+1] = fBoxes[6\*iIndex+5]+fBoxes[6\*iIndex+2];

Suivant les axes x et y, on obtient donc la forme suivante :



Figure 13 Quadrillage obtenu grâce aux premières opérations de voxélisation

*4) Tri et suppression des limites redondantes parce que trop proches les unes des autres :*

→ cf. **annexe p. 34**, méthode SortBoundaries

La méthode *SortBoundaries* considère successivement les trois axes. Pour chacun d’entre eux, on effectue les traitements suivants :

- on trie dans un ordre croissant les limites contenues dans le tableau *fBoundaries*, pour l’axe considéré. La fonction de tri sort un tableau contenant les index des éléments triés ;

- on vérifie que deux limites successives ne sont pas plus rapprochées qu’un coefficient de tolérance (*fTolerance*) et ce, afin d’éviter une éventuelle profusion de limites trop proches les unes des autres, ce qui pourrait être préjudiciable en terme de performances ;

- on place les coordonnées des limites triées et écumées dans un tableau de taille ad-hoc : *fXSortedBoundaries*, *fYSortedBoundaries* et *fZSortedBoundaries*.

Visuellement, on conserve les limites violettes tandis que celles demeurant en traits pointillés sont éliminées car trop proches d’une limite déjà mémorisée.

*5) On obtient la structure voxélisée :*



Figure 14 Structure voxélisée

*6) Enfin, on mémorise les solides contenus dans chaque voxel :*

→ cf. **annexe p. 36**, méthode BuildListNodes

Chacun des nœuds contenu dans une tranche (une tranche définie comme étant l’espace entre deux limites successives) est ensuite enregistré comme étant situé dans celle-ci, ainsi qu’on peut le voir sur le schéma ci-dessous. C’est la méthode BuildListNodes qui permet de le réaliser.



Figure 15 Mémorisation des noeuds

Dans les faits, la solution envisagée pour la mémorisation des nœuds utilise des opérations binaires. Les sous-solides seront consignés dans des octets en utilisant un tableau de caractères. Il est avant tout nécessaire de connaître le nombre d’octets, soit dans la majorité des systèmes, le nombre de caractères nécessaires au stockage des nœuds intégrés à l’instance de classe UMultiUnion. On peut vérifier que la formule suivante donne le résultat escompté :

**int** nperslice = 1+(carNodes-1)/8;

La quantité carNodes représente le nombre total de nœuds. Évaluer nperslice sur des valeurs données de carNodes nous permet de dresser le tableau suivant, puis de constater qu’on trouve bien les valeurs attendues :

|  |  |
| --- | --- |
| carNodes | nperslice |
| 0 | 0 |
| 1 | 8 |
| 2 | 16 |
| 3 | 24 |

Sur certaines machines, il arrive cependant qu’un caractère ne soit pas seulement stocké sur un octet. Afin de tenir compte de cette subtilité, la ligne de code précédente doit être remplacée par celle qui suit :

**int** nperslice = 1+(carNodes-1)/(8\***sizeof**(**char**));

N.B. : la quantité **sizeof**(**char**) permet de déterminer la place que prend en mémoire une variable de type **char** (en nombre d’octets).

Par ailleurs, il est possible, à partir de la quantité carNodes, de déterminer le nombre maximal de tranches pouvant exister le long d’un axe. Le cas donnant le plus grand nombre de tranches est celui pour lequel les nœuds ne se chevauchent pas et sont séparés d’une distance plus grande que la tolérance. L’exemple ci-dessous se présente sous cette configuration. La structure comporte cinq nœuds.



Suivant l’axe des abscisses x, on dénombre au total onze tranches. D’une façon générale, le nombre maximal de tranches nmaxslices est lié à la quantité carNodes par la relation suivante :

**int** nmaxslices = 2\*carNodes+1;

On donne maintenant les grandes lignes de l’algorithme mis en œuvre pour stocker les nœuds contenus dans chaque tranche pour l’axe x :

Figure 16 Fonctionnement simplifié de la méthode UVoxelFinder::BuildListNodes

On détermine :

carNodes – nombre de nœuds de la structure UMultiUnion

nperslice – nombre d’octets de mémoire par tranche

nmaxslices – nombre maximal de tranches sur x

xNumslices – nombre de tranches suivant x

BuildListNodes

Détermination des positions minimale et maximale du nœud courant sur l’axe x

Le nœud courant

est-il dans la

tranche courante ?

Incrémenter le compteur des nœuds pour la tranche courante

(tableau fNumNodesSliceX)

Stocker le nombre du nœud

pour la tranche courante

(au bon endroit dans le tableau fMemoryX)

i < xNumslices ?

j <

carNodes ?

Axe suivant

j++

i++

OUI

OUI

NON

NON

OUI

NON

On explicite à présent plus en détail la partie de l’algorithme qui permet de stocker le nombre du nœud courant dans la mémoire de la tranche courante. Tout d’abord, l’indice current permet de repérer l’indice du premier octet de la mémoire attribuée à la tranche courante dans le tableau storage. Ce tableau aura, au préalable, été alloué dynamiquement et pourra contenir nmaxslices\*nperslice caractères.

Dans cette partie, on utilise des masques et des opérations binaires pour effectuer le stockage des nœuds. Reprenons l’exemple précédent pour rendre le propos plus intelligible :



Figure 17 Structure voxélisée

En suivant l’algorithme exposé à la page 20, on a :

carNodes = 8

nperslice = 1+(carNodes-1)/(8\***sizeof**(**char**)) = 1+(8-1)/8 = 1

nmaxslices = 17

xNumslices = 11

On alloue ensuite un tableau de char de taille : nperslice\*nmaxslices = 17. Il s’agit du tableau storage. On commence par en mettre tous les éléments à la valeur nulle.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

N.B. : On utilise pour des commodités de présentation, la base hexadécimale pour présenter les résultats dans le tableau storage.

Chacun des caractères du tableau permet de stocker huit informations binaires, entre les nombres 00000000 et 11111111. Comme on l’a déjà indiqué, dans le cas où la structure définie comporte plus de huit nœuds, il devient nécessaire d’allouer plusieurs caractères par tranche. Le schéma ci-dessus montre qu’il existe 11 tranches suivant l’axe x, et 7 suivant l’axe y. Cela signifie qu’on n’utilisera pas toute les cases du tableau ci-dessus. En revanche, il pourra être réutilisé et n’aura pas besoin d’être réalloué pour le traitement d’autres axes.

Considérons l’axe x. Remplissons maintenant le tableau storage en suivant l’algorithme présenté à la page précédente :

→ *Tranche 0 :*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

|  |
| --- |
| 00000000 |
| 0 |

On boucle sur tous les solides et on teste si le solide courant est contenu dans la tranche courante. La tranche d’indice zéro suivant x ne comporte aucun nœud ; on ne modifie donc pas le caractère courant du tableau storage.

→ *Tranche 1 :*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

00000001 = x01

OU logique

|  |
| --- |
| 00000000 |
| 1 |

|  |
| --- |
| 00000001 |
|  |

|  |
| --- |
| 00000001 |
| 1 |

**|**

**=**

1 << 0

La tranche 1 comporte, quant à elle, un solide en son sein. On utilise alors le numéro de ce dernier afin de former un masque binaire en décalant un 1 binaire un nombre de fois correspondant à l’index du solide contenu. Dans notre exemple, il s’agit du solide d’indice 0. On effectue ensuite le OU logique.

→ *Tranche 2 :* les solides 1 et 2 appartiennent à cette tranche :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x00 | x01 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

00000110 = x06

|  |
| --- |
| 00000000 |
| 2 |

|  |
| --- |
| 00000010 |
| 1 << 1 |

|  |
| --- |
| 00000010 |
| 2 |

**|**

**=**

|  |
| --- |
| 00000010 |
| 2 |

|  |
| --- |
| 00000100 |
| 1 << 2 |

|  |
| --- |
| 00000110 |
| 2 |

**|**

**=**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x00 | x01 | x06 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 | x00 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

On poursuit de la même manière pour les autres tranches et l’on trouve :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x00 | x01 | x06 | x00 | x18 | x00 | x20 | x00 | x40 | xC0 | x00 | x00 | x00 | x00 | x00 | x00 | x00 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

Lorsque le nombre de nœuds est supérieur à 8, on rappelle que le stockage des nœuds doit s’effectuer sur plusieurs octets (plusieurs caractères). Le traitement est alors similaire, mais un peu plus complexe. Par exemple, si l’on avait 20 solides, il faudrait utiliser un tableau de stockage de 3 caractères par tranche, comme on peut le voir ci-dessous :

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  | ... |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | jusqu’à 50 |

tranche 3

tranche 2

tranche 1

tranche 0

L’utilisation de masques et d’opérations binaires est, dans notre cas, très avantageuse. Ces techniques permettent en effet d’utiliser les opérations binaires de bases (opérateurs ET, OU, OU EXCLUSIF, etc.) d’un usage utile pour effectuer des traitements sur la géométrie considérée.

* 1. Détermination des nœuds candidats dans un voxel donné

→ cf. **annexe p. 35**, méthode GetCandidatesVoxelArray

Les candidats stockés précédemment doivent à présent servir aux traitements qui seront réalisés en aval (e.g. fonction Inside, SafetyFromInside, etc.). L’idée est simple ; il faut, à partir des coordonnées d’un point, déterminer les candidats qui lui correspondent.

Considérons un exemple en deux dimensions. Soit la structure ci-dessous, composée de trois nœuds cubiques :

x

y

*0 1 2 3 4*

*2*

*1*

*0*

**0**

**2**

**1**

Figure 18 Structure voxélisée à analyser

Le comportement décrit ci-dessus est opéré par la méthode *GetCandidatesVoxelArray* (bien vouloir se reporter au diagramme des classes). Voici, de façon simplifiée comment se déroule l’algorithme pour le point orange ci-contre :

- les candidats étant stockés sur chaque axe par tranches, il est nécessaire d’obtenir l’index de la tranche en fonction des coordonnées du point. Dans le cas présent, on doit, à partir des coordonnées du point orange, trouver le couple (3,1). Pour cela, il est utilisé un outil de recherche utilisant la dichotomie comme principe de base : la fonction *BinarySearch* permet de renvoyer l’indice de la tranche immédiatement inférieure à la localisation du point sur l’axe considéré ;

- dans le cas présenté ci-dessus, on a l’état des mémoires suivant :

→ suivant l’axe x :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x04 | x05 | x06 | x06 | x02 | x00 | x00 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

→ suivant l’axe y :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x03 | x07 | x04 | x00 | x00 | x00 | x00 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

- on utilise ensuite les listes de candidats stockés, aux bons indices. Dans le cas considéré, on aura :

→ suivant l’axe x :

\* tranche 3 : 00000110

ET binaire : 00000110

→ suivant l’axe y :

\* tranche 1 : 00000111

Le masque rouge ci-dessus est, par la suite, décodé par la méthode *Intersect*. Cette dernière renvoie les candidats dans un conteneur de type vector. En l’occurrence, ce dernier contiendra ici les index 0 et 1.

Des traitements plus complexes doivent être mis en œuvre dans certains cas, par exemple lorsque le point considéré est localisé sur la limite d’un voxel. Prenons le cas du point violet. Ce dernier est situé à la limite entre deux voxels. La recherche binaire donnerait (3,0). Dans ce qui a été décrit jusqu’à présent, nous aurions :

→ suivant l’axe x :

\* tranche 3 : 00000010

ET binaire : 00000010

→ suivant l’axe y :

\* tranche 0 : 00000011

Or, le solide 1 doit également être considéré comme étant un candidat possible pour le point violet. Cet écueil est résolu en considérant les candidats de la tranche précédente, lorsque le point passé est localisé sur la limite entre deux tranches. On effectue alors les opérations suivantes :

→ suivant l’axe x :

\* tranche 3 : 00000010

\* tranche 2 : 00000011

OU binaire : 00000011

ET binaire : 00000011

→ suivant l’axe y :

\* tranche 0 : 00000011

De cette façon, le décodage renverra les solides 0 et 1.

L’étape venant après la recherche binaire est celle où l’on effectue les opérations sur les tableaux de bits. Dans la façon qui était initialement implémentée, les opérations binaires étaient réalisées caractère par caractère. En utilisant un transtypage, il est possible de diviser le nombre d’opérations à réaliser. Sur la plupart des machines, une variable du type char a une taille de 1 octet, contre 4 octets pour une variable *unsigned int*. En utilisant cette astuce, on arrive donc à diviser par 4 le nombre de fois qu’on utilise le OU binaire.

1. Opérations sur une instance de UMultiUnion

Comme on l’a déjà évoqué, le but des opérations de voxélisation est de pouvoir écrire des algorithmes performants en terme de rapidité d’exécution. La classe *UMultiUnion* comprend un certain nombre de méthode permettant d’effectuer des opérations sur le solide. Il s’agit notamment de : *Inside*, *SafetyFromInside*, *SafetyFromOutside*, *DistanceToIn*, *DistanceToOut*, *Normal* et *Extent*. Ces méthodes utilisent, autant que faire ce peut, les possibilités offertes par la voxélisation. La méthode *DistanceToIn* présentée dans les annexes est fonctionnelle, mais n’a pu être que très peu optimisée.

On rappelle que l’intérêt majeur de la voxélisation est de pouvoir se concentrer sur les nœuds candidats correspondant à un point donné de l’espace. On limite donc, du même effet, les boucles sur l’ensemble des nœuds, ce qui est très avantageux pour des structures en comportant un grand nombre. Somme toute, dans la réalisation des méthodes listées au-dessus, il s’est parfois avéré nécessaire de préférer une solution approchée mais rapide et performante, à une solution exacte mais coûteuse en temps.

On donnera, dans cette partie, les organigrammes relatifs aux principales méthodes qui ont été mises en place.

* 1. Description des principales méthodes implémentées

→ cf. **annexe p. 38**, méthode Inside

* + 1. Méthode *Inside*

Figure 19 Fonctionnement de la méthode UMultiUnion::Inside

Détermination de la liste des candidats pour le point passé

Inside

Boucle sur les candidats

Conversion du point au système d’axe local pour le solide candidat considéré

Application de la fonction *Inside* pour le nœud considéré et le point converti

return **eInside**

**eInside** ?

boolSurface = false

**eSurface** ?

boolSurface = true

plus de candidats

boolSurface = true ?

return **eSurface**

NON

OUI

return **eOutside**

* + 1. Méthode SafetyFromInside

→ cf. **annexe p. 39**, méthode SafetyFromInside

La méthode SafetyFromInside permet, on le rappelle, d’obtenir une estimation de la distance du point considéré jusqu’à la surface du solide. En suivant l’algorithme ci-dessous, on n’aboutit pas, dans le cas général, à un résultat exact, mais on obtient une valeur sous-estimée de la véritable quantité safety.

Figure 20 Fonctionnement de la méthode UMultiUnion::SafetyFromInside

Détermination de la liste des candidats pour le point passé

SafetyFromInside

Boucle sur les candidats

Conversion du point au système d’axe local pour le solide candidat considéré

safetyMax = 0

Le point converti est-il dans le candidat ?

(eInside)

NON

OUI

Application de la fonction *SafetyFromInside* pour le nœud considéré et le point converti. On en tire la variable *safetyTemp*

safetyTemp > safetyMax ?

OUI

NON

safetyMax = safetyTemp

return safetyMax

plus de candidats

→ cf. **annexe p. 39**, méthode SafetyFromOutside

* + 1. Méthode SafetyFromOutside

L’implémentation de cette méthode résulte de la transposition d’une méthode déjà implémentée dans le logiciel ROOT. Son implémentation ne sera donc pas ici détaillée.

* + 1. Méthode DistanceToOut

→ cf. **annexe p. 38**, méthode DistanceToOut

Figure 21 Fonctionnement de la méthode UMultiUnion::DistanceToOut

Détermination de la liste des candidats pour le point passé

DistanceToOut

Liste vide ?

Boucle sur les candidats

OUI

NON

forbiddenNode = -1

dist = 0

candidat = forbiddenNode ?

OUI

Conversion du point au système d’axe local pour le solide candidat considéré

Application de la fonction *Inside* pour le nœud considéré et le point converti

FAIRE

**eOutside** ?

**eInside** ou **eSurface** ?

forbiddenNode = nœud courant

Application de la fonction *DistanceToOut* pour le nœud considéré, le point et la direction convertis. On en tire la variable *tempDist*

dist += tempDist ; mise à jour de la normale

Mise à jour du point courant suivant la direction donnée

Le nouveau point

est-il Inside de la structure globale ?

NON

return dist

Nouvelle liste de candidats

OUI

plus de candidats

Tant que SafetyFromInside pour le point courant est supérieur à 1E-5

return dist

Mise à jour de la normale

NON

Utilisation d’une tolérance pour vérifier s’il existe ou non des candidats dans une zone très proche autour du point initial.

Si on trouve un point possédant une liste non vide, on utilise ce dernier pour effectuer l’algorithme qui commence à NON

* + 1. Méthode Normal

→ cf. **annexe p. 39**, méthode Normal

Bien vouloir se référer aux annexes pour la description de cette méthode.

* 1. Test des algorithmes
     1. Test d’efficacité de la voxélisation, méthode Inside

Dans le but de connaître l’efficacité de la voxélisation, on a implémenté les fonctions *Inside* et  *InsideDummy*, qui déterminent le statut d’un point (ie. dedans, dehors ou sur une surface) vis-à-vis de la structure UMultiUnion. Ces deux méthodes ont, par conséquent, le même but, mais n’y arrivent pas de la même manière ; en effet, *Inside* utilise les résultats issus de la voxélisation, tandis que *InsideDummy* ne le fait pas.

On ne détaille pas ici l’implémentation de ces méthodes. Pour en savoir plus, bien vouloir se référer aux annexes.

On commence par créer une structure comportant un nombre défini de nœuds cubiques identiques disposés régulièrement dans l’espace à trois dimensions. Par la suite, on génère aléatoirement un million de points, ces derniers étant situés sur les trois axes entre les coordonnées -2000 et 2000. Avec une fonction faisant office de chronomètre, on teste la méthode Inside() pour un nombre de nœuds allant de 1 à 8000. Enfin, on affiche les points situés à l’intérieur de la structure de type UMultiUnion, ce qui permet aussi d’avoir un aperçu en 3D de la structure.





Figure 22 Structure à 500 nœuds testée à l’aide de la fonction Inside

Sur les deux images précédents, les points rouges sont les points pour lesquels la méthode Inside() a renvoyé *eInside*, ce qui signifie que le point passé en argument est localisé à l’intérieur d’un nœud de la structure. La méthode Inside semble bien fonctionner dans la mesure où aucun point rouge n’est situé en dehors des cubes.

Ce test visuel ayant été effectué, regardons ce qu’il en est pour la durée d’exécution en fonction du nombre de nœuds. On obtient les graphiques suivants :



zoom



Analyse des résultats, bilan :

D’après les courbes ci-dessus, on constate que l’algorithme utilisant des techniques de voxélisation est plus rapide que l’algorithme élémentaire, pourvu que la structure UMultiUnion comporte au moins environ 25 nœuds. Les deux tracés obtenus semblent être assimilables à des droites dont la différence majeure est la pente ; dans le cas d’une structure voxélisée, on obtient une pente beaucoup plus faible que dans le cas basique. Cela semble confirmer l’utilité de la voxélisation en terme de rapidité.

Les courbes sont valables pour un nombre de points déterminé, dans le cas présent, un millions. Malgré tout, on peut affirmer que la voxélisation est très avantageuse par rapport à l’algorithme de base (en effet, pour 8000 nœuds, l’algorithme simple prend environ vingt fois plus de temps que celui utilisant les techniques de voxélisation).

Par ailleurs, il est possible d’utiliser un logiciel permettant de connaître le temps relatif que prend chacune des méthodes utilisées afin de connaître les méthodes à optimiser dans le code. On utilise ici les outils *Callgrind* et *kCachegrind*.

Voici les résultats obtenus dans le cas où l’on considère l’exemple précédent avec 100 nœuds et un million de points générés aléatoirement, le temps d’exécution est réparti (en pourcentage du temps total d’exécution) de la façon présentée sur les graphes ci-dessous :

a) implémentation de base - 100 noeuds :



b) implémentation utilisant la voxélisation - 100 nœuds :



Les deux cadres ci-dessus présentent la distribution du temps d’exécution en fonction des méthodes implémentées. Le programme utilisé pour tester la fonction *Inside()* est appelé *TestMultiUnion*, c’est pourquoi il apparaît comme le cadre englobant tous les autres. En réalité, l’exécution du programme nécessite aussi des traitements internes, voilà pourquoi les pourcentages associés à *TestMultiUnion* ne sont pas égaux à 100 %.

Plus précisément, à supposer que les traitements internes qu’on a juste évoqués soient d’une durée d’exécution constante dans les trois cas, une diminution du quota lié à *TestMultiUnion* suggère une plus grande rapidité de l’algorithme dans le cas où l’on utilise la voxélisation, et plus encore lorsqu’on utilise une ROM.

Par ailleurs, en faisant passer, dans le second cas, le nombre de nœuds de 100 à 500, on obtient le graphique suivant :

c) implémentation utilisant la voxélisation - 500 noeuds :



On constate que la différence majeure entre les graphiques b) et c) provient de la partie non séparée de la méthode *GetCandidatesVoxelArray*, c’est-à-dire à la détermination des masques. Les résultats obtenus ont été considérés comme suffisants, c’est pourquoi une nouvelle optimisation à ce niveau a été reportée à plus tard.

* + 1. Autres résultats

On présente dans cette partie les tests effectués, ainsi que les résultats obtenus sur les méthodes implémentées au sein de la classe UMultiUnion.

Les tests directs des méthodes par des affichages, par exemple dans un terminal, sont complétés par l’utilisation de tests beaucoup plus élaborés et déjà mis en œuvre pour le logiciel ROOT. En utilisant la *bridge class* (cf. p. 13), on peut utiliser ces tests pour la classe UMultiUnion.

|  |  |  |  |
| --- | --- | --- | --- |
| № test | Méthode(s) concernée(s) | Nature du (des) test(s) | Résultats |
| 1 | Extent | Affichage dans un terminal des limites inférieures et supérieures | Les limites affichées correspondent bien au placement des nœuds dans l'espace |
| 2 | SafetyFromInside | Affichage dans un terminal des *safeties*, pour des points et des directions donnés | Une distance cohérente est retrouvée dans les cas testés |
| 3 | SafetyFromOutside | Affichage dans un terminal des *safeties*, pour des points et des directions donnés | Une distance cohérente est retrouvée dans les cas testés |
| 4 | SafetyFromInside  SafetyFromOutside | Utilisation du de la *bridge class* pour les *safety* | Le test renvoie un résultat valide |
| 5 | DistanceToIn | Affichage, dans un terminal, de la distance vers l’intérieur, à partir d’un point et d’une direction donnés | Les tests renvoient des distances concordantes avec des calculs effectués « à la main » |
| 6 | DistanceToOut | Affichage, dans un terminal, de la distance vers l’extérieur, à partir d’un point et d’une direction donnés | Les tests renvoient des distances concordantes avec des calculs effectués « à la main » |
| 7 | DistanceToIn  DistanceToOut | Utilisation du de la *bridge class* pour les *distances* | Le test renvoie un résultat valide |
| 8 | Normal | Détermination de la normale correspondant à un point, même non localisé sur une surface | Pour la majorité des cas testés, la normale retournée est cohérente |
| 9 | Normal | Utilisation du de la *bridge class* pour les *normals* | cf (\*) |

(\*) Dans le cas du test de la fonction Normal avec les bridge tests de ROOT, le test arrive bien à son terme (100 %), mais des messages d’erreur sont affichés au cours de l’exécution. Apparemment, ces messages proviennent d’un souci d’implémentation au sein des tests eux-mêmes. Nous n’en tiendrons donc pas compte ici.

Dans l’ensemble, les résultats obtenus montrent que les fonctions implémentées permettent d’obtenir les comportements décrits dans la description des méthodes (cf. p. 11).

Malgré tout, la méthode DistanceToIn demeure à perfectionner. D’autre part, la complexité dans l’implémentation des autres méthodes ci-dessus a empêché la réalisation de tests de vitesse poussés.

1. Conclusion

L’objectif initial du projet visant à l’amélioration des performances d’un modeleur géométrique basé sur le logiciel ROOT était de créer une classe d’objets représentant l’union booléenne de plusieurs solides. À cet effet, la classe UMultiUnion a été implémentée. Au sein d’une telle structure, les différents solides placés dans l’espace sont définis comme des nœuds ; à chacun d’entre eux correspond donc une forme intrinsèque (dans l’état d’avancement du projet USolids, il s’agissait seulement de parallélépipèdes rectangles) ainsi qu’une transformation permettant le placement et la rotation du sous-solide dans l’espace à trois dimensions.

Par la suite ont été mises en œuvre, au sein de la classe UVoxelFinder, des techniques de voxélisation permettant de créer un quadrillage virtuel non régulier utilisant les positions extrémales de chaque nœud sur les trois axes. Une fois le quadrillage déterminé sur un axe, le stockage des nœuds localisés dans chaque tranche a été réalisé. À partir de la connaissance des trois index d’un voxel à considérer, et en utilisant des opérations logiques pour manipuler les mémoires créées pour les candidats sur chacun des trois axes, il est possible de dresser une liste des candidats présents à l’intérieur du voxel. Des manipulations subséquentes ont permis de pouvoir obtenir la liste de candidats à partir des coordonnées d’un point, et non plus seulement à partir des index du voxel. Le cas de points situés à la limite de deux voxels a rendu la tâche plus complexe que prévue initialement. En outre, les algorithmes de voxélisation ne se sont pas seulement contentés de répondre au fonctionnement voulu. Ils ont, en effet, été optimisés de façon à rendre plus rapide l’exécution.

La voxélisation a ensuite été utilisée pour implémenter la méthode Inside de la classe UMultiUnion. Cela a rendu possible une comparaison avec une méthode de fonctionnement similaire mais n’utilisant pas la voxélisation. Il a alors été observé que la voxélisation apportait un gain de temps considérable lorsque le nombre de nœuds de la structure devient important. Enfin, d’autres méthodes d’analyse d’une structure UMultiUnion ont été mises en place.

\*

\* \*

1. Références bibliographiques

[1] Site internet de ROOT, <http://root.cern.ch>

[2] Site internet de GEANT4, <http://geant4.cern.ch>

[3] Claude Delannoy, *C++ pour les programmeurs C*, Eyrolles, 2007

[4] <http://www.cplusplus.com>

[5] The ROOT team, *Users Guide 5.26*, 2009

[6] CERN Communication Group, *The Large Hadron Collider*, 2010

\*

\* \*

1. Crédits photographiques

[1] p. 1, *Logo de l’ISEN Brest*, ISEN

[2] p. 1, *Logo du CERN*, CERN

[3] p. 1, *Logo de ALICE*, ALICE - CERN

[4] p. 5, *Pays fondateurs du CERN - 1954*, <http://en.wikipedia.org/wiki/CERN>

[5] p. 5, *Pays membres du CERN - 2011*, <http://en.wikipedia.org/wiki/CERN>

[6] p. 6, *Vue aérienne des installations du CERN* (image modifiée), banque d’images du CERN

[7] p. 6, *Vue extérieure du détecteur ALICE*, Mona Schweizer

\*

\* \*

1. Annexes

Les pages qui suivent présentent les parties les plus importantes du code source relatif au projet. Il s’agira successivement des méthodes de la classe *UVoxelFinder*, puis de la classe *UMultiUnion*.

**UVoxelFinder -** *page 1 de 3*





**UVoxelFinder -** *page 2 de 3*



**UVoxelFinder -** *page 3 de 3*



**UMultiUnion -** *page 1 de 4*



**UMultiUnion -** *page 2 de 4*



**UMultiUnion -** *page 3 de 4*



**UMultiUnion -** *page 4 de 4*