Université Lumière Lyon 2

Année 2000 N°

THÈSE

pour obtenir le grade de

Docteur

en

Informatique

présentée et soutenue publiquement par

David SARRUT

le 25 Janvier 2000

Recalage multimodal et plate-forme d'imagerie médicale à accès distant

préparée au sein du laboratoire ERIC
sous la direction de
Serge Miguet

COMPOSITION DU JURY

Mme.	Jocelyne Troccaz	Rapporteur	(Directeur de Recherche CNRS)
M.	Christian Roux	Rapporteur	(Professeur)
Mme.	Isabelle Magnin	Examinateur	(Directeur de Recherche INSERM)
M.	Ehoud Ahronovitz	Examinateur	(Maître de conférences)
M.	Bernard Tourancheau	Examinateur	(Professeur)
M.	Serge Miguet	Directeur de thèse	(Professeur)

Table des matières

Introduction **1**

I Recalage d'images. Application au positionnement de patient **5**

1 Recalage d'images **7**

1.1 Domaines d'application 8
1.2 Classification des techniques de recalage 10
1.3 Les méthodes iconiques 20
1.4 Conclusion 22

2 Mesures de similarité **23**

2.1 Principes de base 24
2.2 Classification des mesures 28
2.3 Expérimentations 42
2.4 Conclusion 57

3 Interpolations **61**

3.1 Introduction 62
3.2 Étude de procédures d'interpolation 64
3.3 Autres procédures d'interpolation 69
3.4 Expérimentations 72
3.5 Conclusion 82

4 Transformations géométriques d'images **83**

4.1 Transformations d'images 84
4.2 Principe de la nouvelle méthode 86
4.3 Modèle d'exécution de l'algorithme 90
4.4 Expérimentations 93
4.5 Discussion et conclusion 96

5 Positionnement de patients **97**

5.1 Description de la problématique 98
5.2 Travaux préalables 99
5.3 Principe 101
5.4 Justification géométrique 105
5.5 Évaluations expérimentales 109

5.6 Conclusion 120

II Santé et calculs haute-performance **121**

6 ARAMIS : une plate-forme d'imagerie médicale à accès distant **123**

6.1 Contexte 124

6.2 Principes généraux 127

6.3 Coeur du système 134

6.4 Conclusion 136

7 Approche surface **137**

7.1 Extraction de surface triangulée 138

7.2 Détermination de la relation d'adjacence 140

7.3 Complexité et tests expérimentaux 145

7.4 Conclusion 149

Conclusion générale **151**

III Annexes **167**

A Transformations affines et modèle sténopé **169**

A.1 Matrices des transformations affines 169

A.2 Modèle sténopé 170

B Modalités d'acquisition d'images médicales **173**

B.1 Modalités d'acquisition 173

B.2 Nomenclature des données 175

C Index des auteurs cités **177**

Deuxième partie

Santé et calculs haute-performance

6

ARAMIS : une plate-forme d'imagerie médicale à accès distant

Sommaire

6.1 Contexte . . . 124
6.1.1 Motivations . . . 124
6.1.2 Applications distribuées . . . 124
6.1.2.1 Description du domaine . . . 124
6.1.2.2 Caractéristiques . . . 125
6.1.2.3 Étude de l'existant . . . 126
6.2 Principes généraux . . . 127
6.2.1 Ressources . . . 128
6.2.2 Objectifs . . . 128
6.2.3 Principes . . . 129
6.2.4 Intégration dans un domaine hospitalier . . . 131
6.2.5 Flux de données . . . 131
6.3 Coeur du système . . . 134
6.3.1 Partie client . . . 134
6.3.2 Partie cachée . . . 134
6.3.3 Ajout de librairies . . . 136
6.4 Conclusion . . . 136

6.1 Contexte

6.1.1 Motivations

Dans le cadre d'un projet financé par la région Rhône-Alpes et nommé *Santé et Calculs Haute-Performance*, nous avons été amenés à étudier la conception d'une plate-forme d'imagerie médicale à accès distant. Cette étude a débuté à la suite du constat suivant.

Dans un environnement hospitalier, les bases de données d'images médicales sont généralement distribuées entre différents services (cardiologie, radiologie), et sont *potentiellement* accessibles de partout à travers un réseau local. Ces images souvent 3D représentent un volume de données considérable et doivent être accédées, visualisées et traitées par les médecins à l'aide d'outils informatiques spécifiques de haute-performance.

Dans ce domaine et depuis plusieurs années maintenant, la communauté de recherche en imagerie parallèle a développé un grand nombre d'algorithmes et de méthodes d'optimisation pour le traitement et l'analyse d'images. Ces techniques sont généralement dédiées aux architectures MIMD (*Multiple Instructions Multiple Data*), pour des types de parallélisme allant de grain moyen à gros grain, sur des configurations matérielles gérant d'une centaine de processeurs à quelques machines seulement. Néanmoins, l'efficacité de ces outils se fait souvent au détriment de leur accessibilité. De plus, comme les machines parallèles sont très coûteuses, peu d'entre elles peuvent être présentes sur un même site hospitalier.

Nous avons donc essayé d'étudier comment il est possible de permettre dans un environnement hospitalier l'accès *à distance* et *à partir de postes de travail banalisés*, à un ensemble de ressources matérielles et logicielles pour le traitement d'images médicales haute-performance.

Dans ce contexte là, nous nous sommes intéressés aux techniques dites de *Metacomputing* qui permettent à une même application d'accéder et d'utiliser une grande variété de ressources distantes (section 6.1.2). Nous présentons ensuite dans la section 6.2 les ressources à notre disposition et les objectifs initiaux du projet. Enfin, la section 6.3 décrit plus précisément le cœur d'ARAMIS, le prototype existant.

6.1.2 Applications distribuées

Cette section a pour but de positionner notre approche par rapport aux technologies existantes. Les techniques usuelles sont décrites et nos choix et contraintes sont indiqués par un alinéa en caractères italiques.

6.1.2.1 Description du domaine

De manière générale, la notion de *Metacomputing* fait référence aux techniques concernant l'utilisation de plusieurs *ressources* au sein d'une même application. Le parallélisme traditionnel peut être vu comme un sous-ensemble de ce domaine où les ressources sont uniquement composées de processeurs et de leur mémoire correspondante. Plus précisément, on distingue la notion d'*accès* aux ressources et celle de *collaboration*.

Lorsqu'une application cherche à *accéder* à des ressources distantes, le spectre couvert par cette notion de *ressources* est très large. D'un point de vue *matériel*, les ressources sont non seulement des ordinateurs (processeurs, mémoires, disque de sto-

ckage, périphériques), mais également des instruments divers : appareils d'acquisition d'images médicales, capteurs spécifiques. D'un point de vue *logiciel*, les ressources peuvent être des *bases de données*, mais aussi des *applications* ou des librairies de calcul spécifiques. En ce qui concerne ce dernier point, une part importante des techniques de Metacomputing vise à proposer l'accès et l'intégration d'*anciennes applications*. La difficulté est alors de définir des protocoles de communication entre langages, systèmes d'exploitation et processeurs différents. Généralement cela ce fait à travers des schémas de conception de haut-niveau, typiquement l'encapsulation d'objets.

> *Dans notre cas, nous cherchons à accéder à deux types de ressources : des machines (serveur de calcul, base de données) et des librairies de calcul déjà développées.*

D'un autre côté, la *collaboration* en vue d'une tâche s'effectue entre *opérateurs* (par exemple des praticiens travaillant de manière simultanée sur une même image d'un patient afin de confronter leur point de vue) ou entre *entités logicielles* (noeuds, processus, processus légers [1], objets).

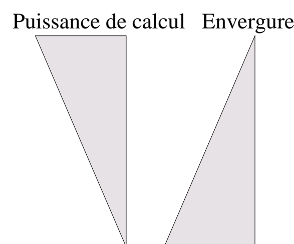
> *Les aspects collaboratifs auxquels nous avons affaire concernent uniquement la gestion de plusieurs clients accédant aux ressources, nous ne décrirons pas le parallélisme des librairies de calcul utilisées.*

Les *apports* de ces techniques sont multiples. En plus des gains de temps inhérents aux calculs parallèles, l'accès simultané par de multiples utilisateurs à des ressources partagées permet des gains d'investissement conséquents. De plus, savoir réutiliser tout ou partie d'un parc d'applications spécifiques lorsqu'une évolution des matériels s'impose, est également un facteur réducteur de coût. Enfin, dissocier l'*interface* des traitements, la première est locale et les seconds distants, permet de faire évoluer les techniques matérielles ou logicielles sans avoir à bouleverser complètement les habitudes des utilisateurs finaux.

6.1.2.2 Caractéristiques

La première notion permettant de caractériser une application distribuée est le *protocole de communication* employé. Celui-ci dépend dès sa conception du contexte de l'application. Ainsi, la classe de réseaux et donc le type de parallélisme à laquelle s'adresse l'application, détermine son envergure. Nous distinguons ici plusieurs niveaux :

- véritable machine parallèle (SIMD, MIMD)
- pile de PCs (dédiés)
  ou grappes de stations
- réseaux local, intranet
- Internet



Le schéma décrit une perte de puissance de calcul au fur et à mesure de l'augmentation de l'envergure du réseau (perte de temps due aux communications distantes), cela s'entend évidemment à nombre de processeurs équivalent. Le premier niveau est celui qui conduit généralement aux meilleures performances en termes de calcul brut. Cependant, cette approche a pris du recul ces dernières années en raison de l'absence manifeste de pérennité des machines parallèles, pour une bonne part due à

1. *threads*

leur prix de revient très élevé. Ce recul a donné lieu au développement des *piles de PCs* et autres *grappes de stations* (voir section 6.2.4). D'une envergure plus large, l'ensemble des machines appartenant à un site défini forme un réseau de type *intranet*. Enfin, le dernier niveau est le plus large et comprend potentiellement *toutes* les machines ayant accès au réseau Internet. Plusieurs projets se placent dans cette voie (voir section 6.1.2.3), et cherchent à exploiter la puissance de calcul potentielle de milliers de machines la plupart du temps sous-utilisées.

> *Notre application utilise deux types de réseaux : les traitements sont effectués sur des machines parallèles (ou grappes de stations) et les accès entre clients et librairies de calcul se font à travers un réseau local (intranet).*

Si, sur de véritables machines parallèles, c'est au niveau des *instructions* même que la concurrence a lieu, des objets de plus haut-niveau sont généralement utilisés pour les autres classes de réseaux. L'entité la plus fine est le *processus léger* (ou *thread*). Il s'agit d'un flot d'instructions à l'intérieur d'un *processus* (Unix), possédant une pile d'exécution propre mais pour lequel peu de ressources sont associées. Plusieurs dizaines de processus légers peuvent s'exécuter au sein d'un même processus. Quant aux différents modèles de communication, il est possible de les séparer en deux classes : les modèles par passages de messages (*message passing*, tels PVM ou MPI) et les modèles orientés-objets, agissant par réification des appels de méthodes, c'est-à-dire en les transformant en objets encapsulant les détails de communications.

> *Le premier protocole (passage de messages) est présent dans pratiquement toutes les librairies que nous manipulons, voir section 6.2.1. Quant au protocole d'accès à ces outils décrit section 6.3, il utilise la seconde approche.*

Bien entendu, les contraintes de conception varient sensiblement suivant les niveaux. En particulier, un point important concerne l'*hétérogénéité des plates-formes* (on considère généralement que le terme *plate-forme* désigne un couple processeur-système d'exploitation). Dans ce cas, différentes problématiques viennent s'ajouter, entre autres en termes d'équilibrage des charges ou de protocoles de communication. Au niveau le plus fin, une autre contrainte réside dans l'éventuelle *hétérogénéité des langages* utilisés. Cet aspect concerne généralement les applications liées à la réutilisation de codes existants.

> *Les machines clients doivent être des postes de travail banalisés, il s'agit donc par définition d'un ensemble de machines hétérogènes. De plus, différents langages sont utilisés pour les outils de traitements d'images et l'application cliente.*

6.1.2.3 Étude de l'existant

Un certain nombre de projets de grande envergure ont été étudiés pour nous permettre de proposer une approche adaptée à notre problématique. Parmi ceux-ci, nous citons ainsi un projet *typique* du Metacomputing (Globus), puis déclinons ceux relatifs aux spécificités de notre approche : accès à un ensemble de *librairies de calcul* (Netsolve), problématique *image* (CEV), et enfin projets *médicaux* en milieu hospitalier.

Dans le système Globus (voir par exemple [FK98, CFKK98]), le but des auteurs est de concevoir une boite à outils fournissant un ensemble de fonctionnalités et d'interfaces de base pour construire des applications distribuées de grande envergure. Cette boite à outils est composée d'une collection de modules définissant une interface utilisée par les composants de plus haut niveau. La partie *librairie de communication*

au sein de Globus est nommée Nexus, [FTT97]. Les auteurs veulent montrer qu'un unique ensemble de mécanismes relativement bas-niveau peut servir de base à la construction d'applications efficaces sur de multiples plates-formes.

Plus spécifique, le projet Netsolve initié par CASANOVA, DONGARRA *et al.* [BCD96, CD97] est un environnement permettant d'utiliser un ensemble de ressources de calcul pour résoudre des problèmes scientifiques. Dans cette approche plus orientée *client/serveur* que la précédente, l'utilisateur soumet une tâche par l'intermédiaire de différentes interfaces (Matlab, Shell, programmes en C ou Fortran) et charge est donnée au système de résoudre le problème de la manière la plus efficace possible. Cela signifie : déterminer la taille et le type du problème, trouver les librairies de calcul correspondantes et finalement les machines les plus adaptées (les moins chargées, les plus proches). Le calcul est alors exécuté selon le plan défini par l'étape précédente et le résultat transmis à l'utilisateur. Du côté des serveurs, le système agit comme un dépôt de stockage de librairies de calcul, et comme un gestionnaire de machines susceptibles de proposer du temps de calcul. L'esprit de ce projet est similaire au nôtre (accès à des librairies distantes), mais la différence essentielle avec nos objectifs concerne la nature des données et donc des traitements.

> *Ainsi, en ce qui nous concerne, l'accès se fait vers des librairies de traitement d'images et les machines de calcul sont limitées à une machine parallèle locale (nous n'avons pas à déterminer le serveur de calcul). Enfin et surtout, les volumes de données que nous traitons (images 3D) sont beaucoup plus lourds.*

En particulier, ces volumes peuvent difficilement être transmis efficacement sur le réseau à un nombre d'utilisateurs importants. Dans le projet CEV pour *Collaborative Environment for Visualization* [BRF98], les auteurs étudient les aspects spécifiques aux données *images* dans un environnement collaboratif. En séparant les composants 2D et 3D des processus de visualisation, ils permettent aux utilisateurs de visualiser le résultat de traitements lourds effectués sur une machine distante.

> *C'est une approche similaire à la nôtre (voir section 6.2.5) qui a été développée à peu près à la même époque (notre première publication date également de 1998).*

Au sein d'un environnement hospitalier, les applications distribuées se retrouvent potentiellement à de nombreux niveaux. Ainsi, GRAVES *et al.* décrivent dans [GTDK97] un système de neurochirurgie à distance. De plus large envergure, THOMPSON *et al.* présentent dans [TJG+97] un système distribué pour l'accès à des informations relatives à la santé de patients : des images, mais également des informations textuelles. Enfin, en utilisant le système Globus précédemment évoqué, LASZEWSKI *et al.* décrivent dans [vLSI+99] une architecture logicielle utilisant des réseaux haut débit et des super-calculateurs pour permettre entre autres la visualisation et l'analyse d'images de MicroTomographie 3D (images acquises par rayonnement synchrotron, permettant des résolutions de l'ordre du micro-mètre).

6.2 Principes généraux

Dans cette section nous décrivons plus précisément l'ensemble des ressources que nous considérons ainsi que les tâches que doit couvrir l'application. À partir de cette liste, les principes généraux du système sont présentés.

6.2.1 Ressources

Nous avons à notre disposition un ensemble de librairies parallèles de traitements d'images. Ces outils ont été développés depuis quelques années déjà, durant les activités de recherches de Serge Miguet et représentent les travaux de nombreux auteurs. Ces outils sont destinés à des architectures MIMD, pour un parallélisme de type grain moyen. Une bonne partie de ces algorithmes repose sur l'utilisation d'une librairie de communication parallèle nommée PPCM (*Parallel Portable Communication Module*) développée au LIP (*Laboratoire d'Informatique du Parallélisme*), voir [CBF+92]. Cette librairie permet de conserver l'efficacité d'un code C optimisé, tout en apportant une certaine portabilité. Ainsi l'environnement PPCM permet de compiler un *même* code pour un *ensemble* de plates-formes cibles (soit logicielles comme PVM ou MPI, soit matérielles comme les machines CrayT3E, Intel Paragon) et pour différents types *d'architectures* parallèles (réseaux de processeurs en anneau, en grille, en étoile, *etc*).

L'éventail des traitements couverts par ces développements concerne des tâches extrêmement consommatrices de temps et de ressources, non seulement en termes de capacité brute de calcul, mais également en capacité mémoire. Outre le gain de temps dû à la multiplication des processeurs, c'est en effet un des grands apports des techniques parallèles que la capacité à traiter d'importants volumes de données. Ainsi, parmi ces outils et techniques d'imagerie médicale parallèles, nous pouvons citer :

S. Miguet *et al.* librairie de gestion d'images 3D nommée *Voxcube* [Mig92] et utilisée tout au long de nos développements ; plusieurs algorithmes d'équilibrage de charges [MP96, MR91, MP95].

J.J. Li *et al.* rendu volumique parallèle [LM92]. Nous avons déjà cité cet algorithme utilisé afin de produire des images DRR pour le positionnement de patient (chapitre 5).

H.P. Charles *et al.* un algorithme parallèle avec équilibrage de charge de visualisation du type Z-buffer, [CLM95]. Cette technique est utilisée pour produire les images de rendus surfaciques de la page 148.

F. Feschet *et al.* structure de données pour un équilibrage des charges dynamique, nommée *Parlist*, [FMP98], (voir section 6.3.2).

S. Contassot *et al.* techniques parallèles de déformation d'images, [CV98].

J.M. Nicod *et al.* extraction parallèle de surface par l'algorithme des Marching Cubes [MN95, Nic97]. Cette technique sera décrite dans le chapitre 7, car elle est à la base de l'algorithme proposé dans ce même chapitre.

Dans son Habilitation à Diriger des Recherches [Mig95], S. Miguet proposait l'intégration de cet ensemble de technologies pour l'imagerie médicale au sein d'une même plate-forme. La section suivante présente le cahier des charges que nous nous sommes fixé et la première ébauche de ce projet.

6.2.2 Objectifs

Le premier objectif consiste à proposer un *accès distant*, à partir de machines banalisées, aux traitements effectués par les outils précédemment décrits. En outre, un certain nombre de contraintes doivent être prises en compte dans cette plate-forme. Ainsi, il est primordial d'essayer de définir des protocoles de communication permettant de *réutiliser* au maximum l'ensemble des codes déjà développés, il ne doit pas

être question de réécrire ces bibliothèques. Cependant, les aspects haute-performance des développements doivent être maintenus. La *maintenance et l'intégration d'outils* à la plate-forme doit être simple et nécessiter le moins de code supplémentaire possible. Enfin, l'*intégration* potentielle d'une telle plate-forme au sein d'un hôpital doit être envisagée, en particulier *minimiser le trafic réseau* est primordial.

Ainsi, le système doit permettre d'une part d'intégrer des librairies d'outils parallèles de traitement d'images et d'autre part de concevoir des applications clientes présentant une interface graphique conviviale à l'exécution distante de ces traitements.

6.2.3 Principes

Dans cette section, nous présentons et motivons les choix qui ont servi de base au développement d'ARAMIS. Cet acronyme signifie *A Remote Access Medical Imaging System* et désigne le prototype que nous avons commencé à implémenter depuis 1997. Basiquement, il s'agit d'une approche *clients/serveurs* dont la partie client est écrite en langage Java et communique avec un serveur de calcul ayant accès aux librairies de traitement d'images ainsi qu'au serveur de base de données d'images médicales.

Java Créé par la compagnie *Sun Microsystem* dans les années 1995 (début du projet vers 1991), le langage Java comporte plusieurs caractéristiques intéressantes pour notre problématique. Java est en fait un *système de programmation*, relativement novateur bien que regroupant un ensemble d'idées anciennes. Le système Java est composé d'un *langage haut-niveau* totalement orienté objet, d'un *compilateur* générant un *code bas-niveau*, proche d'un code machine classique et nommé *byte-code*, mais s'exécutant dans une application particulière nommée machine virtuelle Java (JVM pour *Java Virtual Machine*). Les JVM étant distribuées gratuitement pour la plupart des plates-formes existantes, un *byte-code* Java bénéficie d'une grande portabilité.

Ainsi, l'insertion de JVM au sein des navigateurs Internet classiques, a permis la création d'un nouveau type d'application, les *applets*. Il s'agit d'application Java, dont le code compilé pour la JVM est inséré dans une page HTML classique, à la manière d'une image. Lors du chargement de la page HTML, cette application est ainsi téléchargée à travers le réseau depuis le poste serveur et s'exécute ensuite *sur la machine client*, quelle que soit sa plate-forme, grâce à la JVM du navigateur. Une description plus détaillée de ce principe peut être trouvée dans [NP97], ou sur le site :

`http://java.sun.com`

Ce choix du langage Java pour la partie client de notre plate-forme d'imagerie présente plusieurs avantages. Comme l'application s'exécute dans un navigateur Internet, aucune installation n'est à effectuer sur les postes utilisateurs, le code est unique pour toutes les plates-formes et la maintenance du logiciel est automatique puisqu'il est chargé depuis le serveur à chaque exécution. Néanmoins, quelques inconvénients sont à prendre en considération. Ainsi, le concept d'applet pose potentiellement un problème de sécurité (n'oublions pas qu'il s'agit d'une application exécutée *automatiquement* et *sur le poste client* lors de la connexion à la page HTML). Cela a conduit les concepteurs à introduire différents mécanismes de sécurité, limitant par exemple l'accès à partir de l'applet à toutes autres machines que celle qui l'a servie, ainsi qu'à l'ensemble des fichiers du client autres que ceux situés dans un répertoire particulier (*sandbox*). D'autre part, le langage Java est actuellement réputé pour une certaine lenteur d'exécution. La figure 6.1 présente la partie client d'ARAMIS, toutes

les fenêtres présentées font parties de l'applet.

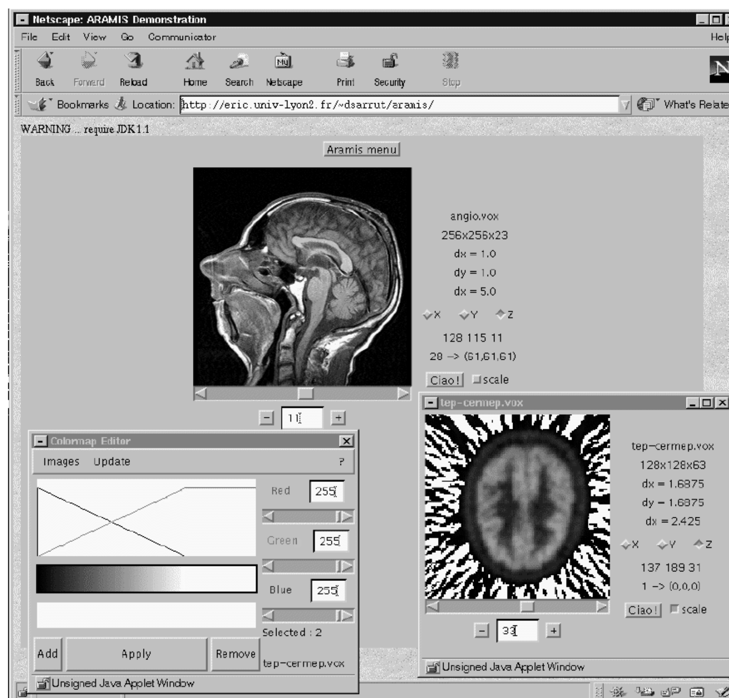


FIG. 6.1 – *L'applet ARAMIS dans un navigateur*

Serveurs Du côté des serveurs, nous distinguons trois niveaux de services :

- un *serveur Internet* classique permet de gérer et délivrer les applets Java à la demande. Il a pour charge de démarrer des sessions de travail sur la machine suivante.
- un *serveur de calcul* composé d'une (ou plusieurs) machine parallèle sur laquelle résident les librairies, permet de lancer un traitement sur une requête de l'utilisateur transmise par l'applet Java. Il s'agit de la ressource de calcul principale.
- un *serveur de base de données d'images* permettant la gestion et l'accès aux images médicales. Ce serveur englobe les mécanismes de sécurité et de recherche dans les données.

Il s'agit d'un découpage *logique*, qui ne correspond pas nécessairement à exactement trois machines *physiques*. Ainsi, le serveur de base de données peut être distribué et gérer plusieurs sites distants. De même, le serveur Web peut être regroupé sur la machine parallèle.

Trafic réseau Dans le but d'éviter de surcharger le réseau, les volumes de données importants (typiquement les images 3D) restent sur les serveurs de base de données et de calcul. Les clients reçoivent *uniquement* des images 2D, résultantes de traitements lourds, effectués par la machine parallèle. Pratiquement aucune ressource n'est ainsi demandée sur les machines clients, ni en puissance de calcul, ni en mémoire, voir section 6.2.5.

6.2.4 Intégration dans un domaine hospitalier

Ce projet de plate-forme d'imagerie médicale est au départ destiné à une intégration au sein d'un hôpital. Dans ce cadre là, afin de réutiliser au maximum les ressources existantes, nous proposons le schéma suivant.

Le serveur de calcul accède fréquemment et régulièrement aux images fournies par le serveur de données. Cela impose que les deux sites soit proches et/ou liés par un réseau haute-performance. Nous définissons ainsi deux niveaux de réseaux, le premier (haut-débit) se situe entre le serveur de calcul et celui de données, il supporte le transport d'importants volumes de données. Le second, entre les clients et le serveur de calcul, peut quant à lui être un réseau local classique (Ethernet) car il ne transporte que trois types de messages : des applets, des images 2D du serveur vers le client, et des requêtes de calcul dans l'autre sens.

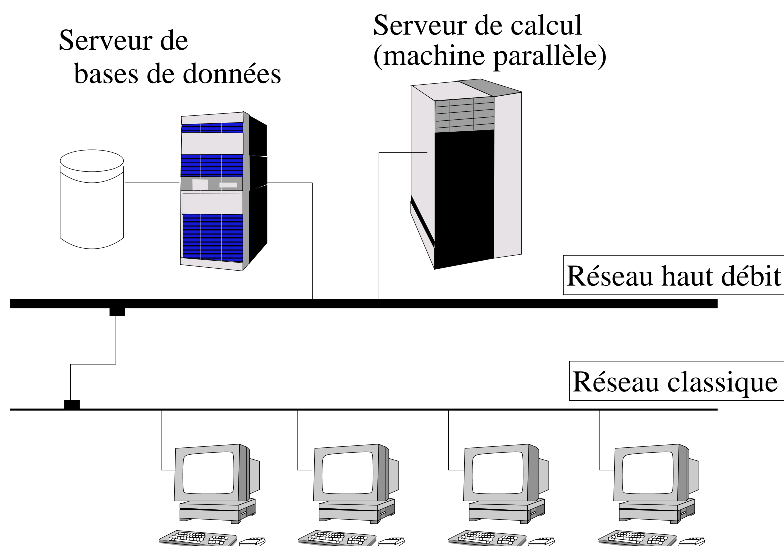


Fig. 6.2 – *Configuration clients/serveurs avec deux niveaux de réseaux*

Dans une optique de réduction de coût de développement, des solutions existent pour les deux parties potentiellement coûteuses de notre schéma, le réseau haute-performance et la machine parallèle. Ainsi, les récentes recherches sur les réseaux haut débit (ATM, Myrinet, IPv6, Fast Ethernet) les ont rendus accessibles à un moindre coût. Typiquement, la redéfinition uniquement logicielle des protocoles de communication bas-niveaux a permis à Pryli *et al.* [PT97] d'atteindre des vitesse de l'ordre du GigaBit par seconde sur un simple réseau Myrinet. Très liés aux progrès réseaux, la communauté de recherche en parallélisme a récemment axé ses développements sur des machines parallèles composées d'une *pile de PCs*, c'est-à-dire de l'intégration au sein d'une même architecture de composants usuels et bon marché.

6.2.5 Flux de données

Nous avons déjà évoqué le fait que les gros volumes de données ne sont jamais transférés sur le poste client. Par exemple, l'outil le plus simple est celui permettant d'afficher une coupe d'un volume 3D selon un des 3 axes. Dans ce cas, l'image est chargée depuis le serveur de base de données vers le serveur de calcul et ce sont uniquement des coupes 2D qui sont envoyées au client sur demande. Un tel procédé est assez rapide sur un réseau local pour permettre un affichage interactif des coupes

(voir figure 6.3).

FIG. 6.3 – *Visualiseurs de volumes*

Cependant, toutes les tâches ne rentrent pas dans ce cadre là. Nous considérons ainsi trois types de tâches :

Premier niveau : les tâches faiblement consommatrices de ressources (typiquement s'appliquant à des images 2D) sont effectuées à travers l'applet Java par la machine locale. Par exemple, la figure 6.4 montre le panneau éditeur de couleur, permettant de créer et d'appliquer interactivement des changements d'intensités et de facteur de transparence.

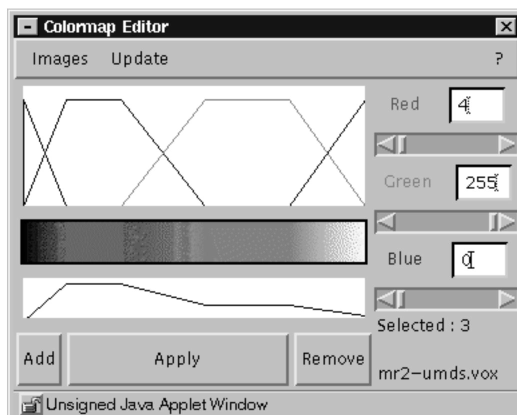


FIG. 6.4 – *Éditeur de couleur (exécution* locale*)*

Deuxième niveau : les tâches lourdes, extrêmement consommatrices de ressources (temps CPU *et* mémoire) sont effectuées de manière distante, par le serveur de calcul. Typiquement, cela concerne les processus qui utilisent au moins plusieurs minutes ou dizaines de minutes de calcul sur une station de travail banalisée. Grâce à ce système ces tâches sont accomplies dans des temps de calcul de l'ordre de quelques secondes seulement, sans que le médecin ait à changer son environnement de travail. Un exemple typique est la technique de rendu volumique incluant des effets de transparence [LM92] (voir figure 6.6). Nous classons en fait dans cette partie les processus dont le surcoût de temps engendré par le transfert des images sur le réseau est négligeable par rapport au

temps d'exécution de l'algorithme.

Troisième niveau : d'un autre côté, certains processus interactifs, tels que la visualisation de surfaces composées de millions de polygones, ne pourraient s'effectuer qu'à un taux de rafraîchissement faible compte tenu du temps supplémentaire engendré par le trafic réseau (cela surchargerait également le réseau). Notre approche ne permet pas de gérer ce type de tâches, le système est destiné à ramener les tâches les plus lourdes vers des temps de calcul raisonnables, de l'ordre de quelques secondes. Il n'est actuellement pas envisageable de fournir sans investissements très lourds, un serveur de calcul assez puissant pour délivrer des images en temps réel à plusieurs utilisateurs simultanés.

Ainsi, dans le but de rendre néanmoins ce dernier type de tâches accessible pour des machines de faibles capacités, nous utilisons plutôt une méthode en deux passes. La première étape consiste a fournir à l'utilisateur un modèle simplifié de la surface à visualiser : un cube ou quelques facettes seulement, voir figure 6.5. Le choix du point de vue de l'objet est alors effectué *localement* à l'aide de cette interface et le rendu en pleine résolution est ensuite activé de manière distante une fois la position choisie.

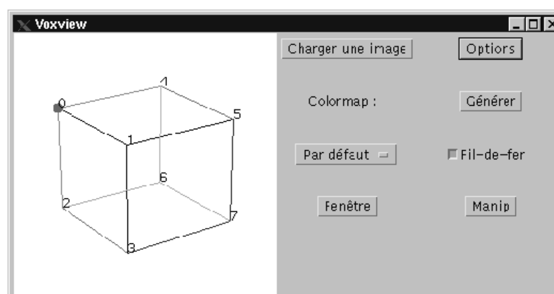


FIG. 6.5 – *Interface simplifiée (car l'exécution est* locale*) pour une technique de rendu volumique*

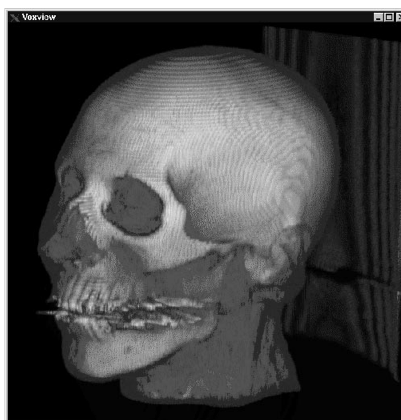


FIG. 6.6 – *Résultat d'un rendu volumique, activé localement et exécuté de manière distante*

6.3 Coeur du système

Cette section décrit l'ensemble des mécanismes de communication entre l'application cliente en Java et les librairies parallèles distantes. Nous désirons fournir un système permettant à un praticien, utilisateur final, de disposer d'une application conviviale permettant d'activer des traitements distants. Notre système se positionne ainsi entre *deux types de concepteurs* : ceux écrivant des outils parallèles haute-performance et ceux concevant des applications clientes, écrites en Java (programmation haut-niveau). Ainsi, du côté serveur, le protocole de communication doit permettre l'intégration d'outils de manière conviviale, idéalement en décrivant simplement les fonctionnalités des librairies. À l'autre bout de la chaîne, notre but est de fournir une API (*Application Programming Interface*), c'est-à-dire une interface de programmation haut-niveau, masquant les aspects distants de l'utilisation des outils afin de permettre la création d'applications d'imagerie médicale haute-performance.

6.3.1 Partie client

L'API finale fournit ainsi trois classes d'objets. Sans entrer dans les détails, le premier type d'objet désigne les *serveurs* ; ces objets servent uniquement pour la phase d'initialisation d'une session de travail. Ensuite, la deuxième classe d'objets concerne les *données distantes*. Il s'agit ici des images médicales dont le chargement sur le serveur de calcul est automatiquement géré. Enfin, la dernière classe d'objets représente les *outils distants*, correspondant aux librairies de traitement d'images situées sur le serveur. À l'aide de simples appels de méthodes, le calcul distant est alors effectué de manière totalement transparente. La création d'applications finales s'apparente alors à l'association de briques de base, les objets, et à la conception d'interface visuelle aux outils.

6.3.2 Partie cachée

Pour initier une session de travail, une phase de connexion est d'abord nécessaire. Ainsi, sur le serveur délivrant l'applet Java, un processus (dit *daemon*) se charge d'accueillir les clients et de démarrer sur leur demande une session de travail sur la machine parallèle. Une fois la connexion établie, le nouveau processus est prêt à recevoir des ordres de l'application Java connectée. La figure 6.7 décrit le schéma d'activation d'une commande distante à partir d'un simple appel de méthode. Il s'agit du principe connu sous le nom de souche-squelette ou *stub-skeleton* en anglais.

Lors d'un appel de la méthode, celui-ci est *réifié*, c'est-à-dire qu'il est intercepté et transformé en objet (partie *stub*). Cet objet *"appel de méthode"* doit alors effectuer plusieurs opérations pour créer une requête distante : récupération des arguments, détermination du serveur et de l'identificateur de la requête distante. Celle-ci est ensuite envoyée à travers le réseau et l'objet se met en attente. Du côté du serveur, la requête reçue est analysée par un autre objet (partie *skeleton*) afin de déclencher la bonne procédure avec les bons paramètres, en particulier l'adresse de l'image sur laquelle s'effectue le traitement. Le traitement s'effectue en parallèle et le résultat du calcul poursuit finalement le chemin inverse.

Points spécifiques Plusieurs points font l'objet d'attentions spécifiques. En particulier l'*attente* mentionnée du côté client peut être *asynchrone* dans le sens où elle s'effectue dans un *processus léger* particulier, évitant ainsi à l'application locale de rester bloquée en attente de l'accomplissement du calcul. Le second point concerne la

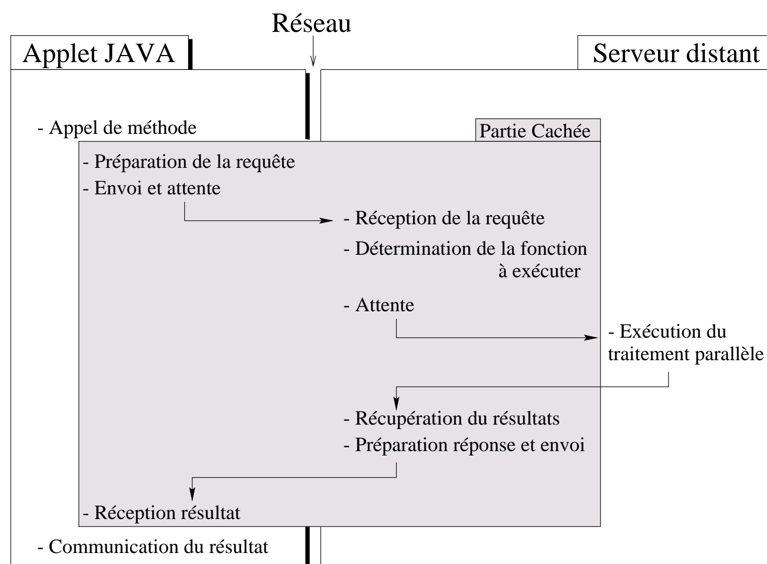


FIG. 6.7 – *Décomposition d'une requête*

gestion de multiples utilisateurs au sein de la machine parallèle. La stratégie adoptée consiste à *réserver* un certain nombre de processeurs de la machine parallèle (ou *noeuds*) par utilisateur. La détermination de ce nombre est actuellement effectuée sur choix de l'utilisateur. Bien entendu, la prochaine étape consiste à allouer ces processeurs de manière dynamique, en fonction du nombre d'utilisateurs connectés.

Une particularité des traitements parallèles réside dans la gestion des données. Bien souvent en imagerie, l'image est découpée suivant une méthode spécifique, typiquement afin de répartir de la manière la plus équitable possible les temps de calcul sur chaque processeur (voir par exemple l'algorithme *élastique* développé par MIGUET dans [MP96, MR91] pour des partitionements rectilinéaires). Ainsi, à la suite d'un traitement parallèle, les données images sont réparties sur chaque processeur. Si le traitement suivant est déclenché, une étape de répartition des données doit avoir lieu. Nous prévoyons dans une évolution future du système d'effectuer cette étape de manière optimale, c'est-à-dire en minimisant les transferts des données grâce à l'algorithme *ParList* décrit dans [FMP98].

Pourquoi ces choix ? D'autres techniques auraient pu se substituer à notre protocole de communication. C'est le cas par exemple de l'utilisation du système RMI (*Remote Method Invocation*) du langage Java même. Celui-ci est néanmoins limité au langage Java (nos librairies de calcul ainsi que le serveur de calcul sont écris en langage C) et l'utilisation de librairies telles que JNI (*Java Native Interface*) aurait alourdi le code et nécessité une adaptation *ad-hoc* à chaque librairie. De plus cette approche, encore à ses débuts, a plusieurs fois fait preuve d'une certaine inefficacité [TTK98]. D'un autre côté, la technologie Corba [GGM97] parait être la solution future de tels développements. Néanmoins, tous les navigateurs n'intègrent pas encore un ORB [2] adéquat.

2. *Object Request Broker*, cœur des spécifications du système Corba permettant d'établir les relations client-serveurs entre différents objets

6.3.3 Ajout de librairies

Le processus serveur, un par client, gère un ensemble d'outils correspondant à autant de points d'entrée dans les librairies de calcul. Lors de l'ajout d'une librairie dans le système, ces points d'entrée sont actuellement extraits de manière interactive. Selon une syntaxe prédéfinie, le concepteur d'une librairie crée alors un simple fichier de liaison décrivant les signatures des méthodes qu'il veut rendre accessibles de manière distante. Cette étape est actuellement manuelle mais pourrait faire l'objet d'un traitement automatique ou semi-automatique à travers un langage de définition d'interface simplifié, un IDL ou *Interface Definition Language*).

Ainsi, le processus distribuant les *requêtes* permet de gérer les outils de manière *semi-dynamique*. Dans une première approche [Sar98], les outils étaient totalement dynamiques dans le sens où ils étaient activés sur demande : il n'était ainsi pas nécessaire de redémarrer ou de compiler le serveur à chaque ajout d'outils. Cependant, nous avons constaté que ce n'était pas une approche efficace, en particulier les temps de chargement des processus sont relativement importants et la gestion de la mémoire devient délicate. De ce fait, dans notre dernière implémentation, les outils ne sont plus dynamiques. Cependant, leur intégration au serveur ne demande *aucune modification* de celui-ci, car au moment de la compilation, le serveur intègre lui-même une liste d'outils définie à travers les *fichiers de liaison*. Ce système permet de gérer la liste des outils de manière totalement indépendante du serveur, sans modification de code à chaque ajout de librairie.

6.4 Conclusion

Dans ce chapitre, nous avons décrit un système qui a été développé durant cette thèse suite à un projet nommé *Santé et Calculs haute-performance* et à la volonté de réutiliser les développements antérieurs effectués au sein du laboratoire. Ce système est composé d'une applet Java permettant à un utilisateur sur une machine de faible ressource, de déclencher à travers une interface conviviale des traitements d'images distants effectués sur une machine parallèle. Du côté serveur, un mécanisme permet d'intégrer de nouvelles librairies de calcul. Différentes parties de ce travail ont été publiées dans [Sar98] et [SM99a].

C'est un projet relativement important (plusieurs milliers de lignes de codes ont été écrites) et encore inachevé, mais qui a donné lieu à un prototype opérationnel. Les prochaines étapes consisteront à intégrer l'ensemble des outils à notre disposition, car tous ne le sont pas encore, et à développer la partie concernant l'accès aux bases de données images.

7

Approche surface

Sommaire

7.1 Extraction de surface triangulée 138

7.1.1 Contexte 138

7.1.2 L'algorithme des Marching Cubes 138

7.1.2.1 Notations 138

7.1.2.2 Principe 138

7.1.2.3 Remarque 140

7.1.3 Objectifs 140

7.2 Détermination de la relation d'adjacence 140

7.2.1 Données de départ 140

7.2.2 Algorithmes naïf 141

7.2.3 Un algorithme linéaire 143

7.3 Complexité et tests expérimentaux 145

7.3.1 Analyse de la complexité 145

7.3.2 Expérimentations et résultats 146

7.3.3 Introduction à la décimation 148

7.4 Conclusion 149

7.1 Extraction de surface triangulée

7.1.1 Contexte

La première partie de ce document a mis l'accent sur une approche volumique, sans segmentation des données. Bien entendu, un système de traitements d'images médicales ne consiste pas uniquement en ce type d'approches, notamment de nombreux traitements s'opèrent sur des surfaces extraites des volumes de données. Plus précisément, nous avons travaillé sur les approches surfaciques obtenues à l'aide de l'algorithme des Marching-Cubes. L'étude de ces surfaces a fait l'objet d'une attention particulière dans notre équipe. Historiquement, les travaux ont été initiés par J.M. Nicod [Nic97], puis nous les avons poursuivis avec lui à travers ceux présentés dans ce chapitre [MNS97]. Par la suite, nous avons collaboré avec D. Coeurjolly afin d'utiliser les travaux présentés dans les sections suivantes comme base pour un algorithme de décimation de surface [CST99].

La section suivante résume la technique des Marching-Cubes, puis nous présentons section 7.2 un algorithme original permettant de calculer en temps linéaire (complexité optimale) la relation d'adjacence entre les éléments de la surface triangulée. Enfin, nous terminons le chapitre en présentant succinctement la technique de décimation de surface faisant suite à ces travaux.

7.1.2 L'algorithme des Marching Cubes

L'algorithme original a été proposé par Lorensen *et al.* en 1987 dans [LC87]. Il s'agit d'une technique de reconstruction de surface à partir d'un volume discret 3D, la surface produite étant représentée sous forme de facettes triangulaires. Elle est générée à partir d'une valeur d'intensité *seuil*, séparant les voxels ayant une valeur supérieure au seuil des voxels ayant une valeur inférieure. On parle de reconstruction d'*iso-densité*.

7.1.2.1 Notations

Nous donnons quelques définitions (voir illustrations[1] figure 7.1) :

- une **coupe** k est l'ensemble des voxels (échantillons de l'image 3D, voir nomenclature annexe B.2) de même altitude, c'est-à-dire ayant la même valeur selon l'axe Oz dans l'image 3D.
- une **cellule** (i,j,k) est la réunion de 8 voxels placés aux sommets d'un cube, de coordonnées $(i+\varepsilon_1,j+\varepsilon_2,k+\varepsilon_3)$ avec $\varepsilon_1,\varepsilon_2,\varepsilon_3 \in [0;1]$. On la représente par un graphe qui associe un sommet à chaque voxel et dont les arcs relient deux voxels différents de plus ou moins un suivant une coordonnée seulement.
- une **couche** k est constituée de deux coupes consécutives k et $k+1$, c'est également l'ensemble des cellules de même altitude k.

7.1.2.2 Principe

L'algorithme traite toutes les cellules de la matrice image, en générant une partie de l'iso-surface (une ou plusieurs facettes) si celle-ci intersecte la cellule. En considérant une arête d'une cellule, un sommet d'une facette de l'iso-surface est généré

1. Merci au grand Jean-Marc pour cette figure ...

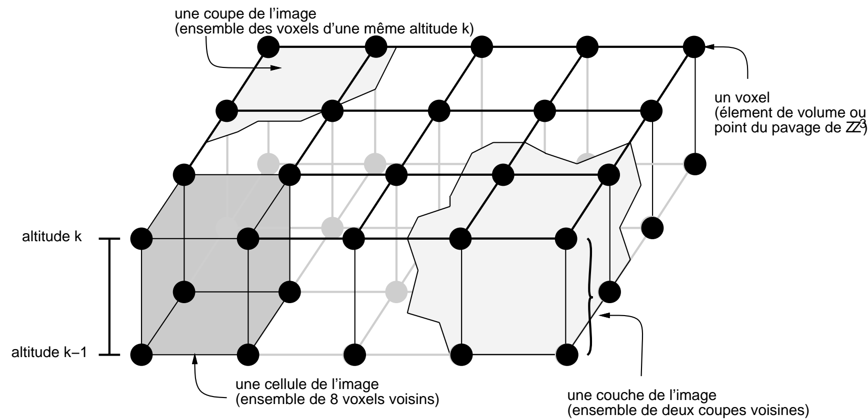


FIG. 7.1 – *Définition d'une* cellule*, d'une* coupe *et d'une* couche *d'une image 3D*

sur cette arête si et seulement si les valeurs associées aux deux voxels extrémités de l'arête sont de part et d'autre du seuil choisi. Les sommets générés sont ensuite reliés pour former une ou plusieurs facettes. Si les sommets d'une cellule ont *tous* une valeur inférieure au seuil (voxels *extérieurs* à la surface) ou supérieure (voxels *intérieurs*), il n'y a pas d'intersection avec l'iso-surface et aucune facette n'est générée.

Comme chaque voxel ne peut avoir que deux positions par rapport au seuil (inférieur ou supérieur), il y a 256 configurations différentes. En assignant un bit à chaque sommet, 1 s'il est de valeur supérieure au seuil et 0 sinon, il est possible de coder ces configurations avec un *index* de 8 bits. De plus, les auteurs ont montré que 14 configurations standards décrivent les 256 autres, ces dernières étant déduites des précédentes par symétries, rotations et/ou complément (intérieur-extérieur). La figure 7.2 donne les 14 configurations de base, les points noirs • représentent les voxels de la cellule ayant une valeur supérieure au seuil.

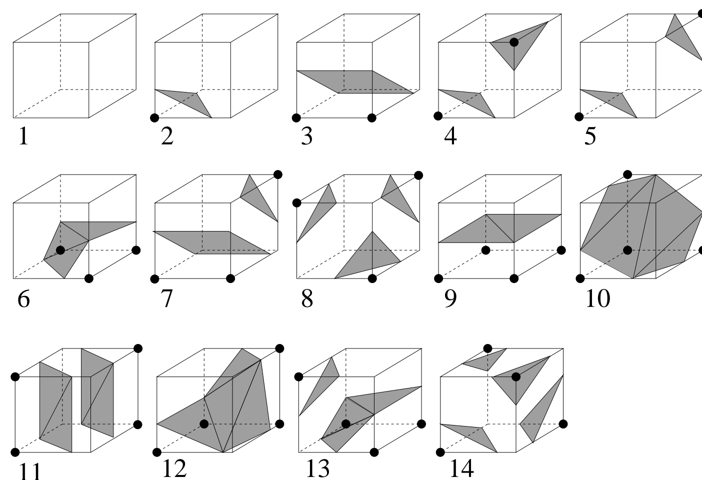


FIG. 7.2 – *Triangulations des 14 configurations standard*

De manière à générer une iso-surface précise, la position d'un sommet est calculée par interpolation linéaire des valeurs des extrémités de l'arête à laquelle il appartient. De plus, lorsque le but est de visualiser la surface par une technique de rendu classique avec ombrage, il est indispensable de connaître la normale à la surface en chaque sommet de la triangulation. On peut ainsi interpoler linéairement sur

l'arête le gradient centré en les voxels pour avoir une approximation de la normale en chaque sommet de la triangulation.

L'algorithme des Marching Cubes est une méthode efficace, conduisant à des surfaces précises, mais qui utilise un important volume mémoire. Dans sa thèse, NICOD [Nic97] propose une étude de la complexité de la surface extraite afin de réaliser une version parallèle avec équilibrage de charges.

7.1.2.3 Remarque

Dans la description originale de l'algorithme, il subsiste des anomalies topologiques dans la surface générée. Ainsi, certains auteurs ont remarqué que lorsque l'iso-surface coupe les quatre côtés d'une des faces d'une cellule, il existe une ambiguïté lors de l'appariement des sommets (fig. 7.3).

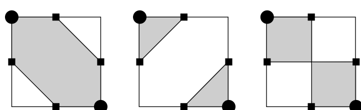


FIG. 7.3 – *Trois possibilités d'appariement différentes (face du dessus du cas n°10 de la figure 7.2)*

Des solutions *ad-hoc* visant à diriger l'algorithme vers l'une ou l'autre des configurations ont été proposées, mais d'autres méthodes plus simples fournissent des surfaces topologiquement correctes à partir de configurations de départ spécifiques. Ainsi, LACHAUD [Lac96] propose trois tables différentes, conduisant à des surfaces topologiquement correctes adaptées à une connexité choisie. C'est cette version améliorée à partir de laquelle nous avons travaillé.

7.1.3 Objectifs

Une surface extraite d'une image volumique est généralement le point de départ à de nombreux traitements (simple affichage, calcul de volume, recalage). Cependant, une information primordiale est absente : une procédure de recherche est nécessaire pour connaître les triangles voisins de chaque élément de surface. Nous montrons dans la section suivante qu'une telle procédure peut conduire à des temps de calcul impraticables si elle est effectuée de manière naïve et proposons une technique pour y remédier.

7.2 Détermination de la relation d'adjacence

7.2.1 Données de départ

La surface générée par les Marching-Cubes à partir de laquelle nous travaillons est décrite par une structure de données non redondante [MN95, ZN94]. Celle-ci est constituée de deux listes (à droite figure 7.4), une liste de sommets et une liste de facettes. Chaque sommet est représenté par ses trois coordonnées de position et les trois coordonnées de sa normale. Une facette est représentée par un triplet d'entiers correspondant aux trois indices de sommets dans la première liste. Les triangles sont ainsi ordonnés par cellules et par couches.

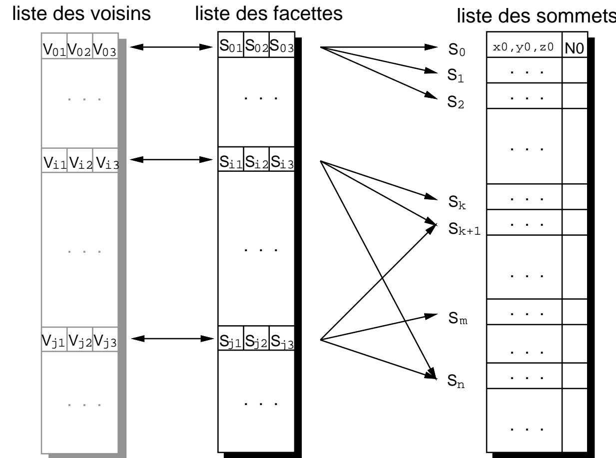


FIG. 7.4 – *Structure de données de la surface extraite*

Notre objectif est de construire la relation d'adjacence de la surface $\mathcal{R}$ définie comme suit : deux facettes distinctes f et f' sont en relation par $\mathcal{R}$ si et seulement si elles ont une arête en commun. Comme nous utilisons une version légèrement modifiée de l'algorithme original des Marching-Cubes, dans laquelle il est prouvé [Lac96] que la surface générée est simple, orientée et fermée, une arête est partagée par *exactement* deux facettes. Une facette donnée possède donc *exactement* trois voisins. La relation d'adjacence est donc stockée en ajoutant à chaque facette les indices de ses trois voisins (tableau à gauche en gris figure 7.4).

Dans la section suivante, nous introduisons deux algorithmes différents ne requérant aucune mémoire supplémentaire que celle nécessaire au stockage de la relation elle-même. En effet, il aurait été possible d'utiliser la structure de donnée 3D initiale ou une liste supplémentaire de sommets pour stocker les facettes partagées par un même sommet, mais l'important besoin en mémoire de ces solutions les rend impraticables avec nos images médicales volumineuses.

7.2.2 Algorithmes naïf

Le premier algorithme que nous présentons est une approche naïve. Sa complexité la rend pratiquement inutilisable pour des surfaces comportant un grand nombre de facettes. Nous expliquons ensuite notre algorithme de complexité optimale, dont la preuve est présentée au paragraphe 7.3.1.

Notations

- soit f une facette de la surface,
- soit $\mathcal{C}(f) = (i,j,k)$ les coordonnées entières de la cellule contenant f lors de sa génération par les Marching-Cubes,
- soit $\mathcal{I}(f)$ l'indice de f dans la liste,
- soit λ le nombre maximum de facettes que peut générer une cellule. Dans la version du Marching-Cubes que nous utilisons, $\lambda = 6$.
- soit n le côté moyen de la matrice 3D de données initiales. Nous supposons ainsi disposer d'un volume de $O(n^3)$ voxels.
- soit m la longueur de la liste de facettes.

Algorithme naïf Le principe de cet algorithme est de trouver les voisins de chaque facette f en cherchant à partir de la position courante dans le reste de la liste, les trois autres facettes ayant une arête en commun avec f. Le nombre d'opérations effectuées sur chaque triangle est alors lié à la distance maximale d dans la liste entre deux facettes adjacentes. Une borne supérieure évidente de d est $\mathcal{O}(m)$ conduisant à une complexité en $\mathcal{O}(m^2)$ pour l'algorithme. De la même manière, une borne inférieure de d est $\Omega(1)$, conduisant à une complexité en $\Omega(m)$. Bien que nous puissions exhiber des familles de surfaces pour lesquelles ces bornes supérieure et inférieure sont atteintes, ces bornes peuvent être affinées pour des données issues d'images médicales réelles. Nous nous proposons donc de donner des bornes plus réalistes sur d.

Soient les hypothèses suivantes sur les images 3D :

H1: le nombre d'éléments de surfaces est une fonction quadratique de n, c'est-à-dire $m = \mathcal{O}(n^2)$.

H2: le nombre m_k d'éléments de surfaces générés dans chaque couche k est borné par une fonction linéaire de n, *i.e.* $m_k = \mathcal{O}(n)$.

Ces hypothèses excluent les familles de surfaces générées mathématiquement, telles que les fractales qui peuvent produire $\mathcal{O}(n^3)$ facettes, ou les objets fil-de-fer conduisant à $\mathcal{O}(n)$ facettes seulement.

Théorème 1

L'algorithme naïf a une complexité $\mathcal{O}(m^{\frac{3}{2}})$ avec des images vérifiant les hypothèses H1 *et* H2.

Preuve :

- *de* H1, *nous pouvons déduire que :* $m = \mathcal{O}(n^2) \Rightarrow \exists \alpha \in \mathbb{R}^{+*} \mid \alpha n^2 \leq m$, *donc*

$$n \leq \sqrt{\frac{m}{\alpha}} \tag{7.1}$$

- *nous déduisons également de* H2 *que :*

$$m_k = \mathcal{O}(n) \Rightarrow \exists \beta > 0 \mid m_k \leq \beta n, \tag{7.2}$$

- *soit f une facette générée sur la couche k. Le nombre de facettes que nous devons parcourir pour trouver le voisin de f est borné par le nombre de facettes susceptible d'apparaître sur les deux couches k et $k+1$:*

$$d \leq m_k + m_{k+1}$$

nous avons ainsi à partir des équations (7.1) et (7.2) :

$$d \leq 2\beta\sqrt{\frac{m}{\alpha}},$$

donc $d = \mathcal{O}(m^{\frac{1}{2}})$.

La complexité de l'algorithme naïf est finalement de $\mathcal{O}(d \times m) = \mathcal{O}(m^{\frac{3}{2}})$. □

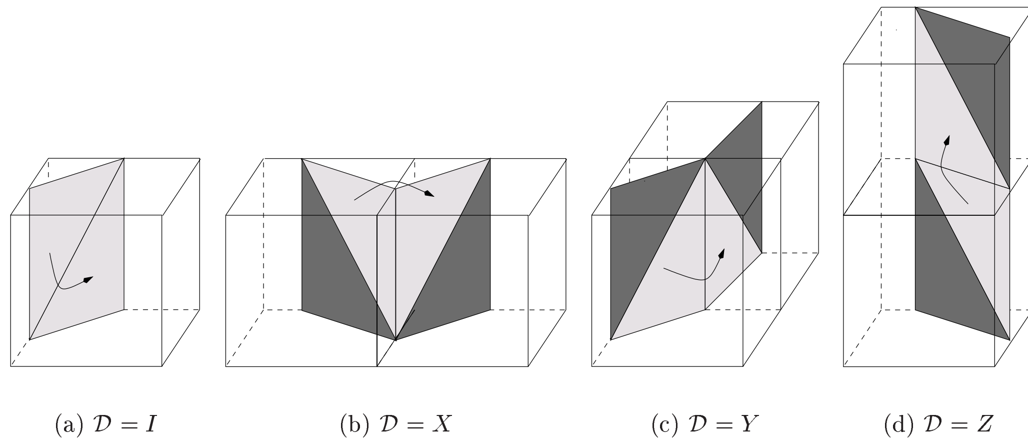
Néanmoins, cette complexité reste élevée et rend difficile une exploitation de cette méthode sur des surfaces provenant d'images médicales. Nous avons donc accéléré ce traitement en proposant une autre façon de construire la relation d'adjacence.

7.2.3 Un algorithme linéaire

L'idée fondamentale est d'éviter le parcours entier de la liste de facettes à partir de la position courante. Nous proposons de commencer le parcours à partir de différentes positions définies par les recherches précédentes.

Soit f une facette appartenant à la cellule $\mathcal{C}(f) = (i,j,k)$. Les voisins f' de f encore inconnus peuvent se trouver dans seulement quatre cellules différentes, qui sont $\mathcal{C}_I = \mathcal{C}(f)$ elle-même, ou les trois cellules 6–voisines de $\mathcal{C}(f)$ qui n'ont pas encore été parcourues, *i.e.* $\mathcal{C}_X = (i+1,j,k)$, $\mathcal{C}_Y = (i,j+1,k)$ et $\mathcal{C}_Z = (i,j,k+1)$. L'indice correspondant I, X, Y or Z est appelé la *direction* du voisin f' et est noté $\mathcal{D}$ dans la suite. Cette direction peut être déterminée à partir des coordonnées de l'arête commune à f et f'.

La figure 7.5 illustre ces quatre cas. Un cinquième cas noté $\mathcal{D} = \emptyset$ sera utilisé dans la section 7.3.1 pour traduire la situation où le voisin recherché a déjà été trouvé par l'algorithme, celui-ci étant placé avant f dans la liste.



(a) $\mathcal{D} = I$ (b) $\mathcal{D} = X$ (c) $\mathcal{D} = Y$ (d) $\mathcal{D} = Z$

FIG. 7.5 – *Les 4 configurations correspondantes aux 4 valeurs de $\mathcal{D}$ (les facettes adjacentes sont en gris léger et indiquées par une flèche)*

Lorsque $\mathcal{D} = I$ ou $\mathcal{D} = X$, nous savons que le voisin f' est proche de f dans la liste et peut être cherché de manière exhaustive, en temps constant. Nous utilisons une optimisation uniquement lorsque $\mathcal{D} = Y$ ou $\mathcal{D} = Z$.

Même si nous ne connaissons pas $\mathcal{I}(f')$, nous connaissons la position $\mathcal{I}(u')$ de la dernière facette qui a été trouvée (à partir d'une facette u) dans la direction $\mathcal{D}$. L'idée principale de notre algorithme consiste à commencer la recherche de f' dans un voisinage de u'. L'algorithme repose alors sur le résultat suivant :

Théorème 2

Soit f la facette courante dans la liste dont nous savons qu'elle possède un voisin f' dans la direction $\mathcal{D}$ ($\mathcal{D} = Y$ ou $\mathcal{D} = Z$). Soit u la dernière facette ayant eu un voisin dans la même direction $\mathcal{D}$. Nous avons la propriété suivante :

$$\mathcal{I}(f') > \mathcal{I}(u') - \lambda$$

Preuve :

Les cellules (i,j,k) sont implicitement ordonnées par le sens de parcours de

l'image par le Marching-Cubes. Suivant cet ordre, le nombre de cellules séparant $\mathcal{C}(u)$ et $\mathcal{C}(t)$ est égal au nombre de cellules entre $\mathcal{C}(u')$ et $\mathcal{C}(f')$.

- *si u et f ont été générés dans la même cellule (i.e. $\mathcal{C}(u) = \mathcal{C}(f)$), alors u' et f' ont également été générés dans le même cellule (i.e. $\mathcal{C}(u') = \mathcal{C}(f')$). Nous ne savons pas si u' a été générée avant, mais comme il y a au plus λ facettes par cellule, il suffit de commencer la recherche de f' à partir de l'index $\mathcal{I}(u') - \lambda + 1$ dans la liste.*
- *sinon (i.e. $\mathcal{C}(u) \neq \mathcal{C}(f)$), le fait que u est avant f dans la liste implique que $\mathcal{C}(u)$ a été parcouru avant $\mathcal{C}(f)$. Nous en déduisons alors que $\mathcal{C}(u')$ et $\mathcal{C}(f')$ ont également été parcourus dans cet ordre. Cela nous permet de conclure que $\mathcal{I}(f') > \mathcal{I}(u')$.*

De manière évidente, nous concluons de ces deux cas que le résultat annoncé est vérifié.

□

La seule structure de donnée nécessaire pour contrôler cet algorithme est réduite à trois adresses dans la liste des facettes : l'index courant $\mathcal{I}(f)$ et les dernières valeurs $\mathcal{I}_Y$ et $\mathcal{I}_Z$ de $\mathcal{I}(u')$, correspondant respectivement à $\mathcal{D} = Y$ et $\mathcal{D} = Z$.

La figure 7.7 représente une partie de la liste de facettes générées à partir de l'exemple de la figure 7.6. Dans cette situation, la facette f possède un de ses voisins, nommé f', dans la direction $\mathcal{D} = Y$. Avant de stocker la relation entre f et f', comme (u,u') était le dernier couple de facettes dans la direction $\mathcal{D} = Y$, $\mathcal{I}_Y$ était égal à $\mathcal{I}(u')$. Donc f' peut être recherchée à partir de la position de u' au lieu de celle de f comme dans l'algorithme naïf. Après avoir établi la relation entre f et f', la nouvelle valeur de $\mathcal{I}_Y$ est $\mathcal{I}(f')$.

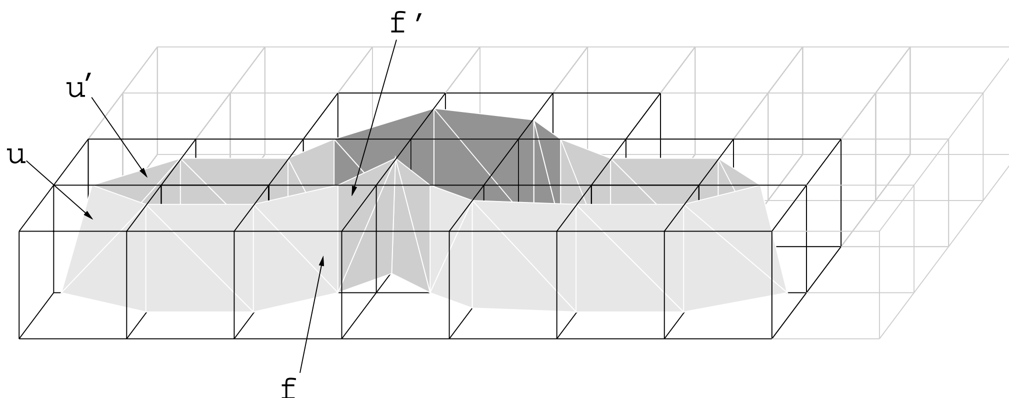


FIG. 7.6 – *Exemple d'une surface générée dans une couche (les couleurs des facettes correspondent à leur coordonnée sur l'axe Oy)*

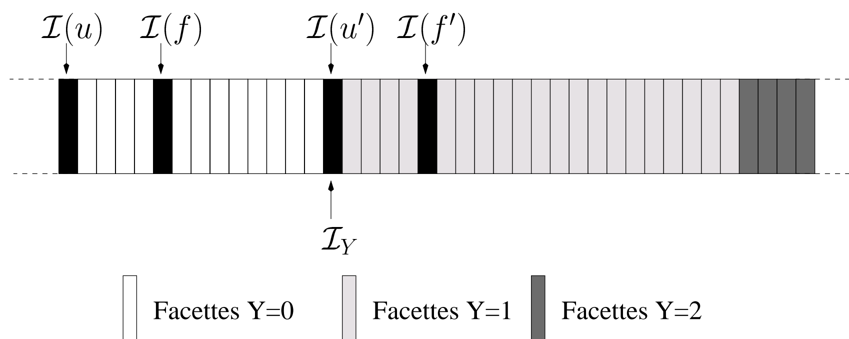


FIG. 7.7 – *Partie de la liste de triangles générée à partir de l'exemple précédent*

7.3 Complexité et tests expérimentaux

7.3.1 Analyse de la complexité

Dans cette section nous présentons une preuve de la linéarité de l'algorithme. Celui-ci parcourt entièrement la liste de facettes et cherche les voisins de la facette courante. Bien que certaines de ces recherches soient coûteuses et puissent être aussi longues que dans l'algorithme naïf, nous verrons qu'elles se produisent assez peu souvent pour être *amorties* par les nombreuses autres recherches en temps constant. Nous sommes dans un cas typique d'une preuve utilisant l'*analyse amortie*. Nous utilisons la méthode du potentiel de SLEATOR, avec les notations de [CLR90].

Soit D_i la structure de données à la $i^{\text{ème}}$ étape de l'algorithme. La méthode associe à la structure de données globale, une fonction potentiel Φ qui fait correspondre à chaque structure de données D_i un nombre réel $\Phi(D_i)$. Soit c_i le coût réel de la $i^{\text{ème}}$ étape. Par définition, la coût amorti $\hat{c}_i$ de la $i^{\text{ème}}$ opération est :

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{7.3}$$

La principale propriété utilisée par l'analyse amortie est que le total des coûts amortis après n étapes est égal au total des coûts réels de ces n étapes plus l'augmentation de potentiel :

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \tag{7.4}$$

Il suffit alors de vérifier que le potentiel final est supérieur ou égal au potentiel initial pour garantir que le total des coûts amortis est une borne supérieure du total des coûts réels.

La $i^{\text{ème}}$ étape de l'algorithme consiste en la recherche d'un voisin f' d'une facette f. Le coût réel de ces recherches dépend de la direction $\mathcal{D}$ de f' :

$\mathcal{D} = \emptyset$: il s'agit de la direction que nous utilisons lorsque f' a déjà été traitée parce que positionnée avant f dans la liste. Aucune recherche n'est nécessaire, nous avons donc $c_i = 1$.

$\mathcal{D} = I$: f et f' sont dans la même cellule, donc distantes dans la liste d'au plus $\lambda - 1$ facettes. Nous avons ainsi $c_i < \lambda$.

$\mathcal{D} = X$: f est dans la cellule suivante de f', donc f et f' sont distantes dans la liste d'au plus $2\lambda - 1$ facettes. Nous avons $c_i < 2\lambda$.

$\mathcal{D} = Y$: la recherche débute à l'index $\mathcal{I}_Y$. Soit d_Y le nombre de facettes parcourues avant que f' soit atteinte. Nous avons : $c_i = d_Y$.

$\mathcal{D} = Z$: la recherche débute à l'index $\mathcal{I}_Z$. Soit d_Z le nombre de facettes parcourues avant que f' soit atteinte. Nous avons $c_i = d_Z$.

Nous pouvons maintenant énoncer le résultat principal :

Théorème 3

L'algorithme précédemment décrit a une complexité linéaire *suivant le nombre de facettes de la liste*

Preuve :

La fonction de potentiel Φ que nous utilisons est :

$$\Phi(D_i) = i - \mathcal{I}_Y - \mathcal{I}_Z \tag{7.5}$$

Il est facile de vérifier que ce potentiel est nul au début de l'algorithme et positif à la fin. En effet, il y a m facettes ayant chacune 3 arêtes : la dernière valeur de i est donc $3m$, et $\mathcal{I}_Y$ et $\mathcal{I}_Z$ sont plus petits que m.
Calculons maintenant l'augmentation de potentiel associée aux 5 valeurs possibles de $\mathcal{D}$:

$\mathcal{D} = \emptyset$, $\mathcal{D} = I$ ou $\mathcal{D} = X$: *dans ces trois cas, $\mathcal{I}_Y$ et $\mathcal{I}_Z$ ne sont pas modifiés, donc $\Phi(D_i) - \Phi(D_{i-1}) = 1$,*

$\mathcal{D} = Y$: *la nouvelle valeur de $\mathcal{I}_Y$ correspond à la position de f'. Celle-ci a été augmentée de d_Y, donc $\Phi(D_i) - \Phi(D_{i-1}) = 1 - d_Y$,*

$\mathcal{D} = Z$: *la nouvelle valeur de $\mathcal{I}_Z$ correspond à la position de f'. Celle-ci a été augmentée de d_Z, donc $\Phi(D_i) - \Phi(D_{i-1}) = 1 - d_Z$,*

En ajoutant ces augmentations de potentiel aux coûts réels, nous pouvons vérifier que le coût amorti de chacune de ces 5 opérations est $\mathcal{O}(1)$. Nous pouvons donc en déduire, d'après l'équation 7.4, que les $3m$ opérations sont effectuées avec une complexité $\mathcal{O}(m)$.

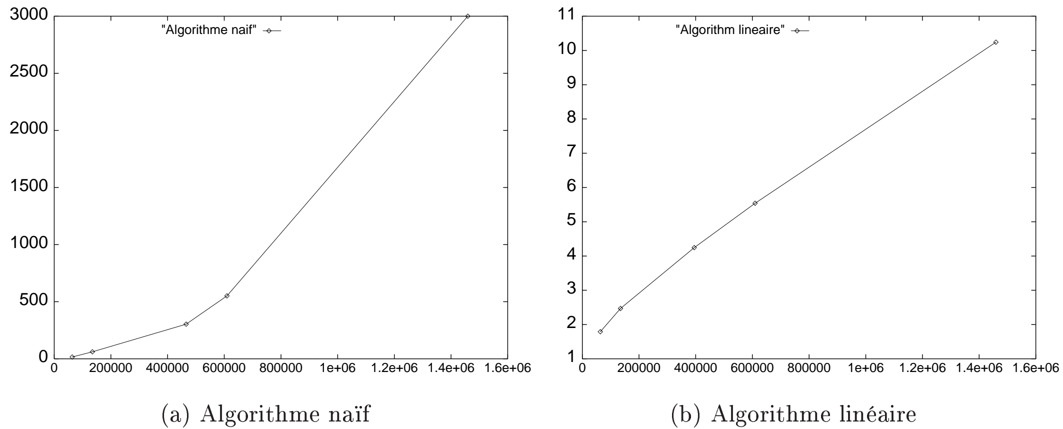
□

Dans la mesure où tout algorithme cherchant à résoudre ce problème doit parcourir au moins une fois toutes les facettes, notre approche est *optimale*.

7.3.2 Expérimentations et résultats

Dans ce paragraphe, nous montrons quelques expérimentations des mises en œuvre des deux approches pour construire la relation d'adjacence. Nous avons ainsi testé les deux algorithmes sur plusieurs surfaces extraites d'images médicale 3D par l'algorithme des Marching-Cubes. Les temps de calcul, pour les deux figures 7.8, sont obtenus avec les mêmes données. Nous pouvons ainsi observer expérimentalement la linéarité de notre algorithme. Pour une surface avec $1{,}5 \times 10^6$ facettes, le temps d'exécution décroît de 3000 à 10 secondes.

Les figures 7.9 et 7.10 montrent de façon graphique les bénéfices de nos améliorations : nous représentons les coûts cumulés des recherches effectuées pour trouver un voisin en fonction du déplacement d. Il s'agit de la probabilité (axe Oy) pour



(a) Algorithme naïf

(b) Algorithme linéaire

FIG. 7.8 – *Comparaison des temps de calcul des deux algorithmes*

une recherche de parcourir un certain nombre (axe Ox) de facettes avant de trouver le voisin. Nous avons volontairement tronqué les histogrammes à des recherches de moins de 250 facettes d'écarts, car même s'il existe des recherches plus longues, elles sont anecdotiques.

La surface de ces histogrammes est fonction du coût total de l'algorithme. Nous observons 3 pics dans le premier histogramme 7.9, correspondant aux trois directions de recherche $\mathcal{D} = I$ (et $\mathcal{D} = X$), $\mathcal{D} = Y$ ou $\mathcal{D} = Z$. Ces pics disparaissent complètement dans la figure 7.10, montrant que la plupart des recherches sont locales : 85% des recherches visitent moins de 10 facettes.

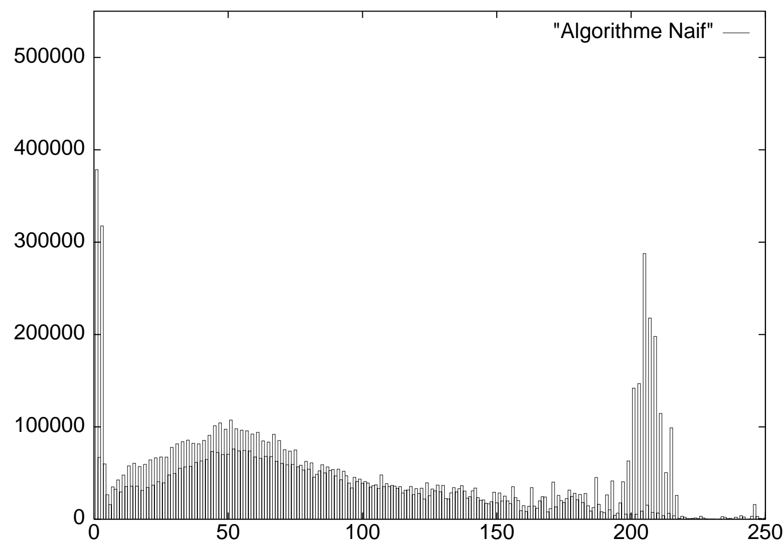


FIG. 7.9 – *Distribution du coût cumulé pour chaque déplacement dans la liste, algorithme naïf*

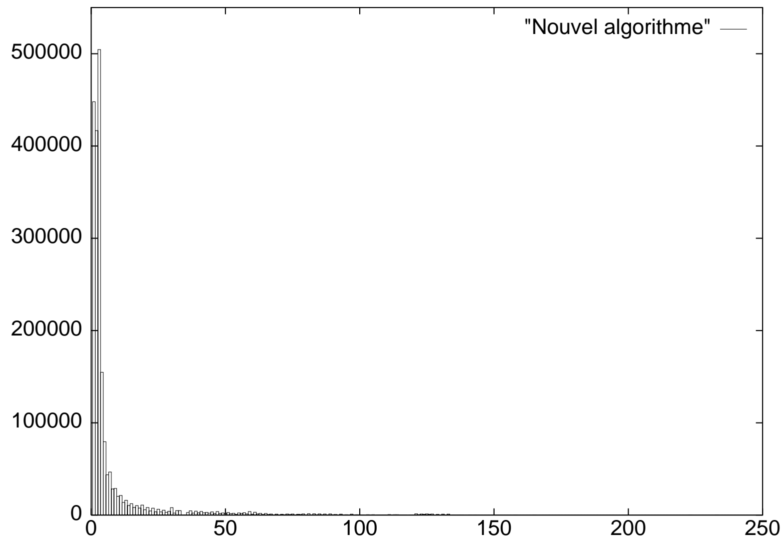


FIG. 7.10 – *Distribution du coût cumulé pour chaque déplacement dans la liste, nouvel algorithme*

7.3.3 Introduction à la décimation

Il s'agit d'un travail effectué lors du stage de D. COEURJOLLY co-encadré par L. TOUGNE et moi-même. Nous nous bornons à mentionner ce travail comme *application* à l'algorithme qui vient d'être décrit.

L'algorithme des Marching-Cubes est intrinsèquement très précis puisqu'il produit des surfaces dont les plus petits éléments sont de taille inférieure à celle d'un voxel. Les surfaces générées comportent alors généralement un très grand nombre de facettes : sur l'exemple présenté figure 7.11, la surface extraite comporte plus d'un million de triangles. Ainsi, lors d'un rendu surfacique avec des techniques classiques de lancé de rayons (*Z-buffer*), une telle quantité de détails ralentit considérablement l'affichage.

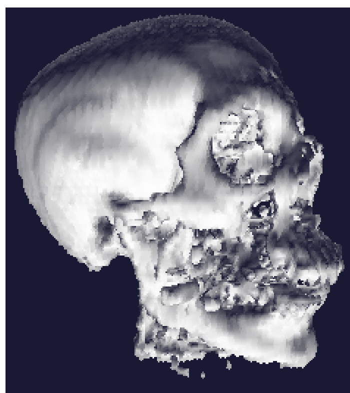


FIG. 7.11 – *Surface extraite par l'algorithme des Marching-Cubes sur une image de tomographie X*

Ainsi, afin de réduire le nombre de primitives géométriques définissant une surface, nous avons mis en œuvre un algorithme original de *décimation de surface*. Cette méthode se base sur l'approche développée par SCHROEDER *et al.* dans [SZL92] et lui apporte plusieurs améliorations. Le principe de base consiste à enlever successi-

vement un certain nombre de facettes. Pour savoir quelles facettes supprimer, nous calculons pour chacune un *coût* correspondant à une évaluation de la courbure locale (combinaison avec les angles formés par la facette courante et les différentes facettes voisines). L'objectif est de conserver les endroits de forte courbure et de remplacer les surfaces pratiquement planes composées de nombreux triangles par une seule facette. Une fois cette fonction de coût calculée, un histogramme cumulé de ces valeurs nous permet de fixer un seuil sur la fonction de coût correspondant au pourcentage voulu de décimation. Le reste de l'algorithme consiste à supprimer les facettes dont le coût est supérieur au seuil et à trianguler le trou ainsi formé. Une attention particulière est donnée à une utilisation économe des ressources mémoires, en accord avec l'algorithme précédent. La figure 7.12 montre ainsi la même surface que sur la figure 7.11, mais avec 40% de facettes en moins.

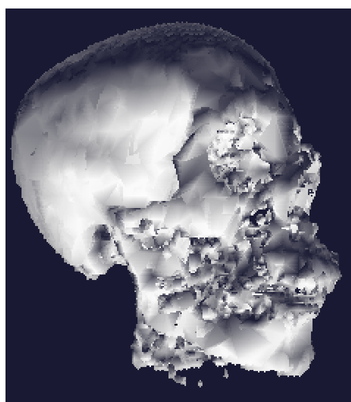


FIG. 7.12 – *Surface décimée,* 50% *des facettes ont été supprimées*

7.4 Conclusion

Ce chapitre a été consacré aux travaux réalisés dans le cadre d'une *approche surface* de l'imagerie médicale. À la suite de l'algorithme des Marching-Cubes, nous proposons un algorithme permettant de construire la relation d'adjacence de la surface triangulée. Une preuve par analyse amortie de la linéarité de l'algorithme est exhibée et nous présentons quelques résultats expérimentaux. Cette technique sert de base à un algorithme de décimation et se positionne en amont d'un bon nombre de traitements sur surface triangulée. Une partie de ce chapitre a été publiée dans [MNS97], et l'algorithme de décimation dans [CST99].