Thèse préparée à Inria Lille,

Pour obtenir le grade de Doctoreur de l'université des Sciences et Technologies de Lille

École Doctorale Sciences Pour l'Ingénieur
Spécialité : Informatique

Nouvelles architectures parallèles pour simulations interactives médicales

Préparée par **Hadrien Courtecuisse**

Soutenue le 09 Décembre 2011

Jury :

Directeur :
Stéphane Cotin - Directeur de Recherche à Inria Lille

Rapporteurs :
Yohan Payan - Directeur de Recherche CNRS, TIMC-IMAG Grenoble
Bruno Raffin - Chargé de Recherche à Inria Grenoble

Co-Encadrants :
Jérémie Allard - Chargé de Recherche à Inria Lille
Christian Duriez - Chargé de Recherche à Inria Lille

Examinateurs :
Hervé Delingette - Directeur de Recherche à Inria Sophia Antipolis
Philippe Meseure - Professeur au laboratoire XLIM-SIC Poitiers

Président :
Serge G. Petiton - Professeur à l'université de Lille 1

Résumé

Nouvelles architectures parallèles pour simulations interactives médicales

Résumé :

Cette thèse apporte des solutions pour exploiter efficacement les nouvelles architectures hautement parallèles, dans le contexte des simulations d'objets déformables en temps réel. Les premières contributions de ce document, se concentrent sur le calcul de la déformation des objets. Pour cela nous proposerons des solutions de parallélisations de solveurs linéaires, couplées à des techniques de preconditionnement asynchrone. Le second ensemble de contributions, repose sur le processeur graphique pour produire une nouvelle méthode de détection des collisions, basée sur le volume d'intersection entre les objets déformables. Enfin les derniers travaux apportent des solutions pour produire une réponse précise aux contacts, et compatible avec le temps réel. Nous aborderons notamment les problèmes liés à la découpe des organes, et à la prise en compte du couplage mécanique entre les contacts. Pour terminer, nous illustrerons nos contributions dans un ensemble d'applications médicales, qui tirent parti des contributions de ce document.

Mots clés : Simulation Médicale, Solveurs Linéaires, Détection des Collisions, Réponse aux Contacts, Préconditionneur, Rastérisation, Contraintes, GPU.

New parallel architecture for medical interactive simulations

Abstract :

This thesis provides solutions to effectively exploit the new highly parallel architectures, in the context of simulations of deformable objects in real time. The first contributions of this paper focus on calculating the deformation of objects. For that purpose, we will propose solutions of parallelization of linear solvers, coupled with asynchronous preconditioning techniques. The second set of contributions relies on the graphics processor to produce a new collision detection method, based on intersection volumes between deformable objects. Then the last works provide solutions to produce an accurate response to contacts and is compatible with real time. We will discuss issues related to the organs cutting, and the recognition of the mechanical coupling between the contacts. Finally, we will illustrate our contributions in a range of medical applications which make the most of the contributions of this paper.

Keywords : Medical Simulation, Linear Solver, Collisions Detection, Contacts Response, Preconditioner, Rasterization, Constraints, GPU.

Remerciements

Bien conscient que la réussite de cette thèse est en grande partie expliquée par l'encadrement exceptionnel dont j'ai bénéficié pendant ces trois années, c'est tout naturellement que je souhaite adresser mes premiers remerciements à mes encadrants.

Je remercie mon directeur de thèse, Stéphane Cotin pour avoir su orienter mes recherches vers des problèmes pertinents et m'avoir aidé à les surmonter. Merci de m'avoir intégré dans des projets d'envergure comme PASSPORT ou Help Me See, et de m'avoir donné la chance de travailler en étroite relation avec le CHU de Lille. J'ai beaucoup apprécié de travailler sur ce type de projets, à la fois concret et utile. Merci également pour ta disponibilité exceptionnelle, tes nombreux conseils, et tout le temps que tu as passé avec moi pour m'aider à avancer. Un grand merci également pour ta façon de diriger l'équipe, qui m'a permis de travailler dans un contexte de travail à la fois propice à une grande efficacité et en même temps plaisant et agréable.

Je remercie également Jérémie Allard qui a été co-encadrant de ma thèse. Merci pour tout ce que tu m'as enseigné aussi bien sur le fonctionnement de SOFA et la programmation GPU, que sur la parallélisation d'algorithmes. Merci pour la grande patience avec laquelle tu as répondu à mes nombreuses questions, et merci encore d'avoir su me motiver en avançant au rythme effréné des deadlines. Un énorme merci également pour les magnifiques voyages que nous avons fait ensemble.

Je remercie Christian Duriez qui m'a également co-encadré. Merci pour m'avoir fait découvrir le monde de la recherche, et l'univers passionnant des simulations médicales. Merci pour tout le temps que tu as passé à m'expliquer - avec la grande pédagogie qu'est la tienne - des notions aussi complexes que celles abordées dans cette thèse. Merci également pour m'avoir toujours soutenu et encouragé, et merci pour m'avoir très rapidement intégré dans des publications.

Je suis bien conscient de la chance considérable que j'ai eu de travailler avec vous, et je voulais encore une fois vous adresser, Stéphane, Jérémie et Christian un immense remerciement pour toute l'aide que vous m'avez apportée, qui a été bien au-delà de l'encadrement de ma thèse. J'espère sincèrement que nous serons amenés à retravailler ensemble.

Je tenais également à adresser mes remerciements à Yohan Payan et Bruno Raffin pour avoir accepté d'être rapporteur et de relire ce long document. Merci pour le temps qui vous y avez consacré, et pour les remarques et commentaires pertinents que vous m'avez donnés. Je remercie également Hervé Delingette et Philippe Meseure pour avoir accepté d'être examinateur et pour être venu à Lille, et merci à Serge Petiton pour avoir accepté d'être le président du jury.

Pendant cette thèse j'ai également eu la chance de travailler avec de nombreuses personnes que je souhaite également remercier. Merci à Jérémie Dequidt pour m'avoir beaucoup aidé dans mes travaux de recherche, et pour m'avoir confié la responsabilité d'enseigner à Polytech'lille. Merci Igor Peterlik, pour le travail exceptionnel que tu as fait à INRIA,

et pour les applications fascinantes que nous avons construites en combinant nos travaux. J'espère vraiment qu'on se reverra dans un futur proche, et qui sait peut-être qu'un jour j'arriverai à voir ton orgue. Merci à Jérémie Ringard (alias Manimoule), pour le rendu de la simulation de la cataracte, et merci de m'avoir accompagné pour les manipulations.

Je remercie Jean Philippe Deblonde, Yiyi Wei, Olivier Comas et Vincent Marjozik avec qui j'ai partagé mon bureau. Merci JP pour les blagues à répétition et pour la bonne humeur que tu dissipais dans le bureau. Un grand merci Olivier, pour m'avoir souvent servi d'exemple et m'avoir souvent aidé. Merci également pour ta bonne humeur, et tous ces bons moments passés à refaire le monde avec JP. Bon courage à Yiyi pour la fin de sa Thèse, ainsi qu'à tous les doctorants, Vincent, Hugo Talbot, Alex Bilger, Ahmed Yureidini, Julien Bosman et Mario Sanz. N'oubliez pas de continuer les réunions d'équipe le midi (surtout Hugo qui a encore besoin de beaucoup d'entraînement), et de prendre les pauses café même quand il y a du boulot ! Je remercie également Périne pour m'avoir aidé à faire le poster MICCAI.

Merci à Frédérick Roy, Juan Pablo de la Plata Alcalde, Pierre Jean Bensoussan, Frederic Chateau et Chi-Thanh Nguyen, les ingénieurs qui ont participé au développement de SOFA, et qui m'ont apporté leur aide à de nombreuses occasions. Merci de m'avoir aidé à prendre mes marques pour démarrer dans SOFA, et pour tous les développement de dernière minute pour finaliser un papier ou une démo.

Avant de terminer, je tiens à remercier tout particulièrement ma famille et mes proches pour m'avoir soutenu et encouragé pendant ma thèse. Un immense merci à Audrey Trullard pour avoir vécu cette thèse avec moi (au rythme des deadlines et des conférences) et pour m'avoir toujours supporté. Merci pour le temps considérable que tu as passé à m'aider tout au long de ces trois années, et pour toutes les relectures du manuscrit que tu as réalisé. Un grand merci également à mes parents pour l'aide qu'ils m'ont apporté, et pour leur soutien quotidien. Désolé pour mes nombreux changements d'humeur et les “désolé j'ai pas le temps, j'ai trop de travail”. Enfin, un grand merci à ma sœur Stéphanie, pour avoir de nombreuses fois corrigé l'anglais de mes papiers.

Pour finir, un grand merci à toutes les personnes que je n'ai pas mentionné, et qui ont contribué à la réussite de cette thèse.

Liste des publications

Chapitres de livres et Journaux

[1] H. Courtecuisse, H. Jung, J. Allard, C. Duriez, D. Y. Lee et S. Cotin. *GPU-based Real-Time Soft Tissue Deformation with Cutting and Haptic Feedback*. Progress in Biophysics and Molecular Biology, 2011c, special Issue on Soft Tissue Modelling.
URL http://codrt.fr/allardj/pub/2011/CJADLC11

[2] J. Allard, H. Courtecuisse et F. Faure. *Implicit FEM Solver on GPU for Interactive Deformation Simulation*. In *GPU Computing Gems Vol. 2*, NVIDIA/Elsevier, 2011b, to appear.
URL http://codrt.fr/allardj/pub/2011/ACF11

Articles référencés en conférences

[3] H. Courtecuisse, J. Allard, C. Duriez et S. Cotin. *Preconditioner-Based Contact Response and Application to Cataract Surgery*. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2011a.
URL http://www.lifl.fr/~courtecu/

[4] H. Courtecuisse, J. Allard, C. Duriez et S. Cotin. *Asynchronous Preconditioners for Efficient Solving of Non-linear Deformations*. In *Proceedings of Virtual Reality Interaction and Physical Simulation (VRIPHYS)*, 2010.
URL http://codrt.fr/allardj/pub/2010/CADC10

[5] J. Allard, F. Faure, H. Courtecuisse, F. Falipou, C. Duriez et P. G. Kry. *Volume Contact Constraints at Arbitrary Resolution*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010), vol. 29, no. 3, 2010.
URL http://www.sofa-framework.org/projects/ldi

[6] H. Courtecuisse et J. Allard. *Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors*. In *High Performance Computation Conference (HPCC)*, IEEE CS Press, 2009.
URL http:///allardj/pub/2009/CA09

[7] C. Duriez, H. Courtecuisse, J. Alcalde et P. Bensoussan. *Contact Skinning*. In *EUROGRAPHICS 2008 / K. Mania and E. Reinhard Short Papers Contact Skinning*, ACM Press, 2008

Autres contributions scientifiques

[8] J. Allard, H. Courtecuisse et F. Faure. *Implicit FEM and Fuild Coupling on GPU for Interactive Multi-Physics Simulation*. SIGGRAPH Talk, 2011a

[9] H. Courtecuisse, J. Allard, C. Duriez, I. Peterlik, S. Cotin et L. Soler. *GPU-based interactive simulation of liver resection Real-Time Live*. SIGGRAPH Real Time Live Demo, 2011b

Sommaire

1 Introduction générale **1**
- 1.1 Introduction . . . 3
- 1.2 Les simulations et leurs applications . . . 3
- 1.3 Les éléments constitutifs d'une simulation . . . 7
- 1.4 Les nouvelles architectures parallèles . . . 9
- 1.5 Contributions . . . 13
- 1.6 Plan . . . 13

2 Simulation d'objets déformables **15**
- 2.1 Introduction . . . 17
- 2.2 Mécanique des milieux continus . . . 17
- 2.3 La méthode des éléments finis . . . 27
- 2.4 Intégration et résolution du problème . . . 35
- 2.5 Les modèles déformables . . . 39
- 2.6 Programmation moderne des GPU . . . 45
- 2.7 Conclusion . . . 49

3 Résolution de systèmes linéaires **51**
- 3.1 Introduction . . . 53
- 3.2 Résolution de systèmes d'équations linéaires . . . 53
- 3.3 Intégrateur implicite optimisé sur GPU . . . 65
- 3.4 Techniques de pré-conditionnement asynchrone . . . 75
- 3.5 Conclusion . . . 86

4 Détection et formulation des contacts **89**
- 4.1 Introduction . . . 91
- 4.2 Les méthodes de détections et réponses aux contacts . . . 91
- 4.3 Détection de collision GPU . . . 103
- 4.4 Extension aux contacts volumiques . . . 114
- 4.5 Conclusion . . . 125

5 Résolution et calcul de la compliance **127**
- 5.1 Introduction . . . 129
- 5.2 Calcul de la réponse aux contacts . . . 129
- 5.3 Parallélisation du Gauss-Seidel . . . 134
- 5.4 Calcul de la compliance sur GPU pour modèles spécifiques . . . 145
- 5.5 Construction du LCP basée sur le préconditionneur . . . 154
- 5.6 Conclusion . . . 167

6 Applications et Conclusion **169**
- 6.1 Applications . . . 171

6.2 Bilan 181
6.3 Perspectives 183

Bibliographie **185**

Annexes **198**

A Définition du tenseur de contrainte **201**

B Algorithme du CG **203**

C Réduction parallèle **205**

D Implémentation du scan parallèle **207**

E Liste des symboles **209**

F Index **211**

Chapitre 1

Introduction générale

Table des matières

1.1 Introduction 3
1.2 Les simulations et leurs applications 3
1.2.1 Les simulations basées sur la physique 3
1.2.2 Les simulations en temps réel 4
Simulation pour les jeux 4
Simulation physiquement correcte 4
1.2.3 La simulation pour le médical 5
Identification des besoins 6
1.3 Les éléments constitutifs d'une simulation 7
1.3.1 Vue d'ensemble 7
Calcul de la déformation 7
Détection des collisions et réponse aux contacts 8
Le rendu des simulations 8
1.3.2 La plateforme opensource SOFA 9
1.4 Les nouvelles architectures parallèles 9
1.4.1 Développement des architectures parallèles 10
1.4.2 Calculer avec la carte graphique 11
1.4.3 Nouveaux langages de programmation 11
1.5 Contributions 13
1.6 Plan 13

1.1 Introduction

Le premier chapitre de ce document est consacré à la description générale du contexte dans lequel s'inscrit cette thèse. Nous présenterons les besoins grandissant pour les simulations virtuelles, en particulier dans le domaine de la chirurgie. Nous verrons que les simulations médicales par ordinateurs, présentent un grand intérêt pour la médecine d'aujourd'hui. Cependant, pour pouvoir exploiter un simulateur dans le domaine du médical, il est encore nécessaire de relever plusieurs défis. En effet, les besoins en terme de précision et de réalisme de ces applications sont tels que la puissance de calcul des ordinateurs actuels n'est pas suffisante.

D'un autre côté, l'informatique a récemment été bouleversée par l'apparition de nouveaux langages de programmation, qui permettent d'accéder facilement à la puissance des cartes graphiques. Ces nouveaux outils donnent accès à une puissance de calcul importante. Cependant, nous verrons que ces calculateurs restent difficiles à exploiter. De ces constatations nous en déduirons notre problématique, et nous présenterons nos solutions pour y répondre.

1.2 Les simulations et leurs applications

Depuis des centaines d'années, on cherche à formaliser les phénomènes physiques qui nous entourent. D'un ensemble d'expériences, on en a déduit des lois qui permettent de prédire avec précision la position et le comportement des objets. La résolution de ces lois par l'informatique permet de produire des simulations dans lesquelles les objets virtuels ont un comportement semblable aux objets réels. Une simulation consiste alors à calculer le mouvement des objets virtuels au cours du temps.

Par ailleurs, puisque tout est simulé on peut également connaître des informations comme la vitesse dans une région donnée, ou les forces s'exerçant sur une partie du matériau. Évidemment, ceci présente un intérêt majeur dans de nombreux domaines d'application. Par exemple, l'industrie de l'automobile utilise ces applications pour simuler des crash-tests, qui sont plus rapides, reproductibles, et surtout beaucoup moins chers à réaliser que d'en faire l'expérience. L'aéronautique, le bâtiment, le cinéma ou la robotique sont autant de domaines qui ont également un grand intérêt pour ces applications.

1.2.1 Les simulations basées sur la physique

Pour pouvoir utiliser une simulation dans le domaine de l'industrie, il est nécessaire qu'elle fournisse un minimum de garanties. Notamment, on attend d'une simulation que son résultat soit suffisamment représentatif du comportement des objets réels. Cela nécessite de résoudre l'ensemble des lois physiques qui décrivent le comportement des objets. Ces lois amènent bien souvent à des équations complexes dont la résolution par l'informatique n'est pas triviale, que ce soit pour obtenir une précision suffisante dans les équations, ou pour limiter les temps de calcul (qui peuvent facilement devenir très conséquent). La plupart du temps, il est alors nécessaire de simplifier les modèles, et on comprend facilement que cela affecte également les performances. On parle généralement d'un rapport entre le temps de calcul et la précision. En règle générale, cela signifie que plus on cherche à obtenir une grande précision, plus les temps de calcul seront importants, et inversement

on peut souvent diminuer le temps de calcul en simplifiant les équations de la physique.

Dans certains cas, on peut se contenter d'algorithmes qui nécessitent plusieurs heures, voir plusieurs jours pour fournir un résultat. C'est par exemple le cas des simulations utilisées dans le cadre des effets spéciaux. Même si le temps de calcul de chaque image est conséquent, il est toujours possible d'assembler les images une fois que la séquence a été calculée. On pourra ensuite rejouer les images plus rapidement pour obtenir le film. En revanche, dès que l'on souhaite interagir avec la simulation, il est impératif de garantir un rafraîchissement des images élevé. En effet, il est tout à fait inconcevable de demander à un utilisateur d'attendre plusieurs heures avant de voir le résultat de son interaction.

En réalité, on considère qu'il faut calculer au moins 25 images par seconde pour donner la sensation d'interactivité à l'utilisateur. Il faut alors réaliser l'intégralité des calculs dans un temps très court. De plus, une autre problématique touche au temps simulé. Pour que l'utilisateur puisse réaliser un geste facilement, il faut que le temps simulé s'écoule à la même vitesse que le temps réel. On parle alors de simulation en temps réel (nous détaillerons cette notion dans le chapitre suivant). La contrainte de temps réel complexifie énormément la construction d'une simulation, car d'une part on ne dispose que de quelques millisecondes pour effectuer le calcul de chaque image, et d'autre part les objets doivent réagir à la même vitesse que les objets réels. De grandes simplifications sont alors nécessaires, ce qui se fait bien souvent au détriment de la précision.

1.2.2 Les simulations en temps réel

Les simplifications qui permettent de diminuer les temps de calcul ne sont pas uniques, et dépendent de l'application qu'on cherche à réaliser.

Simulation pour les jeux

Les premières simulations en temps réels ont été réalisées dans le cadre des jeux vidéos. Dans ce type d'application, le temps de calcul est beaucoup plus important que la précision de la physique. En effet, dans un jeux vidéo on se contentera souvent d'un comportement plausible, et on choisira souvent les algorithmes les plus rapides. Par exemple, dans les jeux la majorité des objets sont considérés comme rigides, ce qui permet de limiter le calcul des objets à un seul point. En effet, si un objet ne subit aucune déformation alors tous ses points subissent la même transformation pendant le mouvement. Ceci permet bien évidemment d'économiser du temps de calcul, et de simuler de nombreux objets en temps réel. La détection des collisions est un autre exemple sur lequel on peut gagner du temps. En effet, l'hypothèse de mouvement rigide des objets, permet de réaliser d'autres optimisations, en supposant que leur forme reste convexe pendant toute l'application.

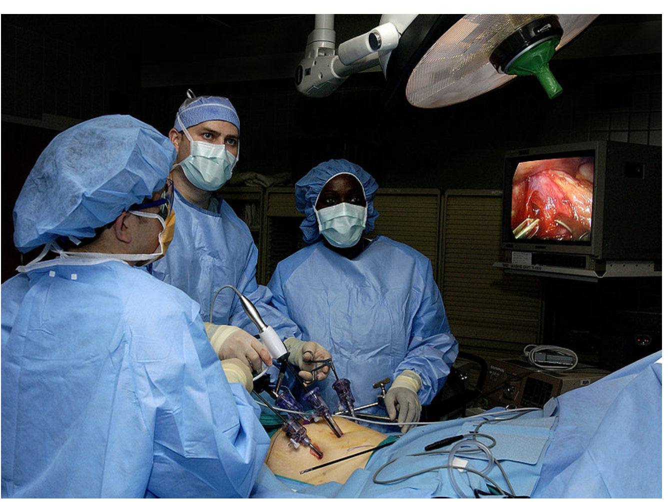
Simulation physiquement correcte

Plus récemment, on a cherché à utiliser l'informatique pour l'apprentissage et l'entraînement. Le meilleur exemple est sans doute les simulateurs de vols, qui peuvent reproduire le comportement d'un appareil qui réagit exactement comme s'il était dans les airs. Il est alors beaucoup moins dangereux de former les pilotes aux manœuvres d'urgence, tel que la panne d'un moteur, ou un atterrissage forcé. Ces simulateurs atteignent aujourd'hui un tel degré de réalisme qu'ils sont capable de reproduire les sensations que

peuvent ressentir les pilotes en vol. Ceci est possible car pendant la simulation toutes les forces sont connues, et on peut facilement les transmettre à un périphérique haptique (qui dans ce cas est la cabine du simulateur).

1.2.3 La simulation pour le médical

Plus récemment, le domaine de la médecine est un autre secteur qui a émis de nouveaux besoins envers les simulateurs. Durant la dernière décennie, un ensemble de nouvelles techniques chirurgicales moins invasives ont été introduites pour réduire la douleur du patient, le temps de récupération, et dans certains cas, le temps d'intervention du personnel médical. De nouvelles technologies sont sans cesse utilisées dans les salles d'opérations, et elles ajoutent de plus en plus une distance entre le chirurgien et le patient. La chirurgie abdominale par voie transluminale en est un exemple (voir figure 1.1a). Des micro-instruments sont insérés par voies naturelles sans aucune incision, ce qui permet une guérison rapide du patient. La vision du médecin se fait à l'aide d'une caméra fixée à l'extrémité d'un endoscope. Ces nouvelles techniques, nécessitent d'acquérir en permanence de nouvelles compétences pour s'adapter aux changements de la chirurgie ouverte conventionnelle (par exemple les tremblements amplifiés, la diminution des sensations tactiles, la perte de perception de la profondeur).



(a) Chirurgie laparoscopique[1]

(b) Chirurgie de la cataracte

Fig. 1.1 – Opération laparoscopique réelle, et opération virtuelle de la catarate.

Plus récemment, de nouvelles perspectives sont apparues avec le développement des techniques d'imagerie médicale, qui pourraient permettre de développer des simulations spécifiques à chaque patient (voir figure 1.1b). Ces simulations pourraient être bénéfiques dans certaines situations, lorsque le patient présente une pathologie rare, ou quand la meilleure stratégie chirurgicale n'est pas claire. Dans ce cas, la simulation peut être utilisée comme un outil de planification efficace. Ceci a été une motivation pour un certain nombre de travaux dans le domaine de la simulation de la chirurgie par ordinateur, et du rendu haptique (voir par exemple Marescaux et al. (1998), Picinbono et al. (2000), Brown

[1]Source : http://commons.wikimedia.org/wiki/File:Laparoscopic_stomach_surgery.jpg

et al. (2002), Forest et al. (2004), ou Harders (2008)). Il a été démontré que l'utilisation de simulateur informatique peut conduire à une formation plus efficace et systématique, fournissant ainsi une évaluation objective de la compétence technique Seymour et al. (2002). D'autres études montrent également que les compétences acquises grâce à une simulation peuvent être transférées dans la salle d'opération (voir Grantcharov et al. (2004) par exemple).

Assistance pre-opératoire Les simulateurs basés sur des données spécifiques aux patients, peuvent être un moyen d'entraînement pour les chirurgiens. Ils pourront alors tester plusieurs stratégies, ou choisir le matériel adapté, pour minimiser les risques. Aujourd'hui, ces choix dépendent intégralement de l'expérience du médecin, qui ne possède qu'une vision partielle des spécificités anatomiques du patient. Par ailleurs, de tels simulateurs permettraient également d'accélérer, et de compléter la formation des étudiants. En effet, la santé des patients étant une composante essentielle, la formation des étudiants se fait dans la mesure du possible dans des conditions idéales, où on évite les complications. Avec un simulateur, on peut facilement créer tout type de scénario et préparer les étudiants aux différentes éventualités. De même la reproductibilité et l'utilisation à la demande, permet de diminuer sensiblement le temps de formation des étudiants.

Assistance per-opératoire Un autre cas d'application concerne l'utilisation de système de réalité augmentée pour la chirurgie guidée par l'image. Le simulateur peut par exemple améliorer les informations transmises aux chirurgiens en dessinant en transparence la position d'une tumeur, ou des vaisseaux sanguins en temps réel. En effet, les chirurgiens ne disposent généralement que de quelques images prises avant l'opération, à l'aide d'une IRM par exemple. Ces moyens d'acquisition ont souvent des effets néfastes pour les patients, ce qui explique qu'ils sont utilisés de façon limitée. Cependant, l'effet de la respiration ou une position légèrement différente du patient, peut déformer les organes et invalider les clichés pré-opératoire. En recalant continuellement des données du patient, l'utilisation de simulateur pourrait ainsi améliorer sensiblement la qualité des soins.

Assistance post-opératoire L'utilisation des simulateurs après l'opération est également une motivation. La simulation permettrait de prédire l'évolution d'une pathologie à long terme, afin d'être en mesure d'apporter des soins au moment approprié. On pourrait par exemple prévenir plus facilement la rupture d'un anévrisme, ou corriger l'emplacement d'électrodes insérées dans le cerveau pour la stimulation cérébrale profonde. Cette dernière application, aussi connue sous le nom de *Deep brain stimulation* (DBS) consiste à implanter des électrodes délivrant un faible courant électrique dans le cerveau, pour traiter la maladie de Parkinson. Cependant, la zone à stimuler est généralement de très petite taille, et il est aujourd'hui très difficile de prévoir la déformation du cerveau post-opératoire quand les chirurgiens implantent les électrodes. Enfin, on pourra également se servir de la simulation pour mieux comprendre le fonctionnement des organes, et évaluer l'effet d'un traitement.

Identification des besoins

La simulation médicale nécessite de relever plusieurs défis, que nous énumérons (sans être exhaustif) :

1. Nous devons être en mesure de simuler de façon précise le comportement des organes, qui sont des structures généralement molles et déformables. La simulation doit alors reposer sur une formulation physique décrivant précisément le comportement de ces structures au cours du temps.
2. Les modèles anatomiques (la position ou la taille des organes), ainsi que les propriétés des tissus (la rigidité ou l'élasticité) doivent être spécifiques aux patients. Il doivent être faciles à obtenir pour pouvoir utiliser le simulateur à grande échelle.
3. Il faut également être en mesure de gérer les interactions entre les corps déformables. Pour cela, il faut détecter ces interactions de façon précise avec les surfaces complexes des organes. Il faut également produire une réponse aux collisions précises. Il est important de pouvoir simuler des phénomènes physiques comme le frottement qui permet de saisir des objets.
4. Pour augmenter le réalisme, on peut transmettre les forces de contacts à un périphérique haptique, de sorte que le chirurgien puisse avoir un retour tactile de l'opération virtuelle.
5. Nous devons également être en mesure de gérer les découpes et changements topologiques, car c'est une tâche courante dans les opérations chirurgicales. Cela implique généralement de nombreux calculs pour remettre à jour les informations mécaniques du matériau.
6. Pour finir, toutes ces contraintes génèrent une énorme quantité de calcul qu'il faut résoudre en temps réel, pour permettre à un utilisateur d'interagir avec le simulateur.

Dans ce document, nous contribuerons uniquement sur les aspects $1, 3, 5$ et 6. Cependant, nous montrerons des applications qui intègrent tous ces aspects, soit par des collaborations (modèles spécifiques aux patients), soit en utilisant les travaux disponibles dans l'équipe de recherche (retour haptique).

1.3 Les éléments constitutifs d'une simulation

Nous décrivons ici les principaux éléments constitutifs d'une simulation, afin d'avoir une vue d'ensemble du fonctionnement de ces applications.

1.3.1 Vue d'ensemble

Il existe de nombreuses façons de construire une simulation. Nous présentons ici une vue d'ensemble des principaux composants (voir figure 1.2) intervenant dans nos simulations.

Calcul de la déformation

Le calcul de la déformation est le point central dans une simulation d'objets déformables. Cela nécessite de définir des modèles de déformations, qui vont décrire le déplacement de chaque portion de matière en fonction des forces qui leur sont appliquées. Pour produire une simulation réaliste, la mise en équation de ces relations devra respecter un ensemble de lois physiques. Ces équations sont généralement difficiles à résoudre numériquement, et bien souvent nous devrons trouver des moyens pour les simplifier mais obtenir malgré tout une bonne approximation.

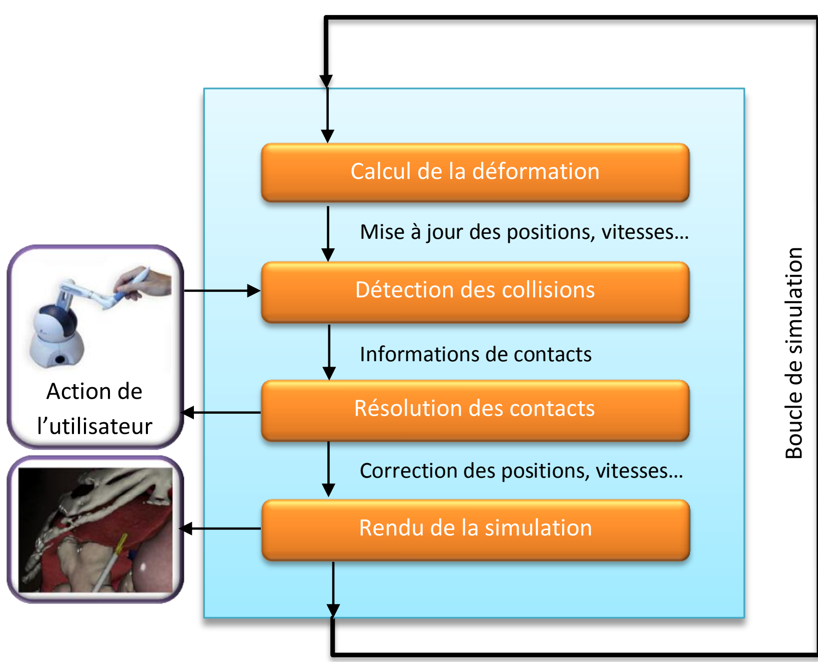


Fig. 1.2 – Schéma d'une simulation, plusieurs tâches sont répétées en boucle pour produire des images de l'application.

Détection des collisions et réponse aux contacts

Une fois que la déformation des objets est évaluée, on procède généralement à une détection des collisions. La détection des collisions est également une tâche difficile à réaliser dans les simulations en temps réel. Du point de vue informatique, les objets sont définis par un ensemble de positions dans l'espace qui délimitent la surface des objets. Ainsi, si l'on cherche à savoir si deux objets sont en collision, il est nécessaire de procéder à tout un ensemble de tests géométriques pour savoir si un point d'un objet est à l'intérieur ou à l'extérieur d'un autre objet.

La détection des collisions n'est cependant pas suffisante pour pouvoir interagir avec les objets déformables. En effet, une fois les collisions détectées nous devrons être capable de créer une réponse adaptée. Pour cela, nous devrons également nous baser sur un ensemble de lois physiques, pour calculer une force qui corrige les interpénétrations. On pourra également profiter du calcul de ces forces pour les transmettre à un périphérique haptique, et ainsi fournir un retour tactile à l'utilisateur.

Le rendu des simulations

La dernière étape consiste à afficher l'image des objets. Cette étape est primordiale puisqu'elle va produire l'image finale, qui sera la seule information visible par l'utilisateur. De nombreuses problématiques sont également à résoudre pour pouvoir réaliser cette tâche correctement. Par exemple le plaquage de textures sur les objets, ou le calcul de l'éclairage et des ombres. Ces algorithmes sont également consommateurs de temps de calcul, mais le rendu des simulations représente en général une discipline à part entière, que nous

n'aborderons pas dans ce document.

1.3.2 La plateforme opensource SOFA

La plateforme (SOFA) (Simulation Open Framework Architecture)[2] est un projet libre, spécialement orienté pour le médical, qui permet le développement rapide de simulations en temps réel. Pilotée par INRIA, ce projet intègre de nombreux résultats de recherche innovants, ce qui fait de ce logiciel un des acteurs principaux de la simulation physique. Il est aujourd'hui utilisé aussi bien dans des projets industriels que par des partenaires académiques.

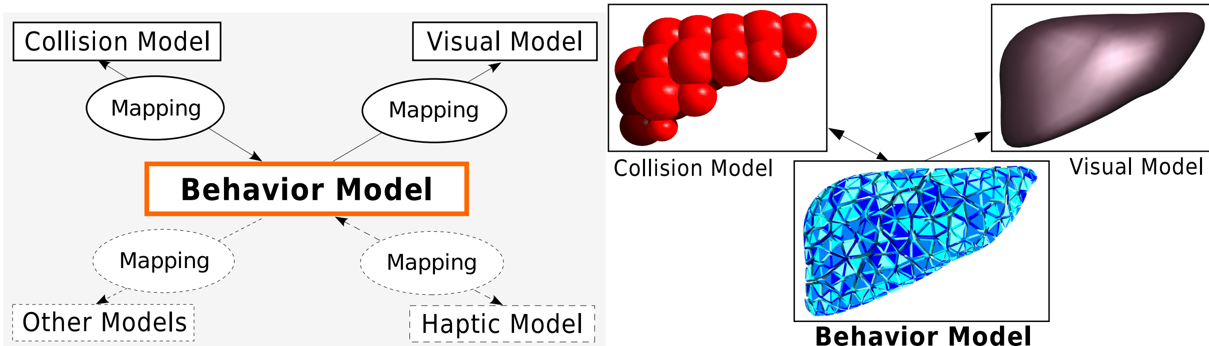


Fig. 1.3 – Approche multi-modèles pour représenter les modèles déformable.

La force de SOFA réside dans son approche multi-modèle (voir figure 1.3). Dans la simulation, un objet possède plusieurs représentations qui sont chacune spécialisées pour certaines actions. En règle générale, chaque objet possède un modèle mécanique, un modèle de collision et un modèle visuel. Leur résolution peut être adaptée indépendamment pour maximiser les performances. Ce concept repose sur la notion de mappings, qui permet de transférer les force mécaniques d'un modèle à l'autre. SOFA permet par exemple d'utiliser un modèle mécanique et un modèle de collision très simple afin de maximiser les performances, tout en gardant un modèle très détaillé, pour la partie visuelle.

SOFA offre, la possibilité de décrire simplement sa propre simulation virtuelle au travers de fichiers XML. La construction de simulations est alors accessible au plus grand nombre. Cette plate-forme permet ainsi de faire le lien entre le monde de la recherche et l'industrie, et facilite la collaboration entre les groupes de recherche.

Tous les travaux présentés dans cette thèse ont été réalisés sur cette plateforme. En effet, SOFA nous a offert un moyen de produire rapidement des algorithmes sans se soucier de toutes les tâches auxiliaires (pourtant nécessaires comme l'affichage, l'haptique...). De plus, son caractère libre permet de comparer, tester et valider son propre algorithme très simplement.

1.4 Les nouvelles architectures parallèles

Au cours des dernières décennies, la puissance des ordinateurs n'a pas cessé d'augmenter. À cette époque, si l'on disposait d'un algorithme trop coûteux il suffisait bien souvent

[2] http://www.sofa-framework.org/

d'attendre quelques années pour voir la puissance des ordinateurs augmenter, et permettre son exécution dans un temps raisonnable. Les simulations ont également pu tirer partie de cela, en développant des modèles toujours plus précis et proches de la réalité. Toutefois, ces dernières années, on constate que la fréquence des CPU n'augmente plus, principalement pour des raisons de consommation électrique et de dissipation thermique. Pour continuer à vendre des processeurs, la tendance actuelle est d'augmenter le nombre d'unités de calculs plutôt que leur vitesse individuelle. La plupart des CPU sur le marché sont dits multi-cœurs, c'est-à-dire qu'ils possèdent plusieurs cœurs de calcul sur la même puce. Aujourd'hui, il est fréquent de trouver 4 à 8 cœurs de calculs dans un ordinateur de bureau standard.

1.4.1 Développement des architectures parallèles

La multiplication des cœurs de calculs permet de maintenir une croissance constante, en terme de puissance brute de calcul. En effet, même si chaque cœur n'est pas plus rapide que la génération précédente, leur multiplication permet de réaliser un plus grand nombre de tâches simultanément. En théorie, on peut alors diminuer le temps d'exécution, si on arrive à trouver plusieurs tâches qui peuvent être réalisées simultanément. On parle alors de calcul parallèle. Cependant, le découpage en tâche parallèles est à la charge du programmeur, qui doit explicitement définir un ensemble d'opérations qui ne présentent pas de dépendances. En effet, il n'est pas toujours évident que deux instructions puissent être exécutées simultanément. Si on prend par exemple l'algorithme 1, les tâches a et b (ligne 1 et 2) peuvent être exécutées en même temps, mais il est impératif qu'elles soient terminées avant de faire l'instruction ligne 3. Si cette précaution n'est pas prise, on ne pourra pas garantir la consistance de C. Le processus exécutant la ligne 3 pourrait utiliser des valeurs partiellement écrites en mémoire pour A et B (ou pire, des valeurs non initialisées postérieures à l'affectation), ce qui produirait un résultat erroné.

1 $A = \text{tâche}(a)$;
2 $B = \text{tâche}(b)$;
3 $C = A + B$;

Algorithme 1 : Exemple de dépendance de données

Ainsi, pour pouvoir exécuter les deux premières tâches en parallèle nous avons besoin de synchroniser les processus. Une synchronisation consiste à attendre que plusieurs processus parallèles soient arrivés à un point précis du programme, pour pouvoir continuer. Bien que nécessaires, ces opérations vont introduire des retards dans les calculs. En effet, comme les deux tâches parallèles (lignes 1 et 2) ne vont certainement pas finir en même temps, l'une des deux va alors devoir attendre. De même, un seul processus pourra exécuter la ligne 3, pendant que le second n'aura plus rien à faire. Pendant ces temps d'attente, l'ensemble des processeurs fonctionnent à une puissance inférieure à la puissance théorique. On constate que, le calcul parallèle ne permet pas toujours de diviser le temps de calcul d'une tâche par le nombre de processeurs.

D'autres problèmes peuvent également diminuer sensiblement l'efficacité de la parallélisation. Par exemple, la bande passante mémoire est souvent un point bloquant quand on veut paralléliser un algorithme. En effet, même si les cœurs de calcul sont du-

pliqués, la vitesse d'accès à la mémoire ne l'est pas. La vitesse de transmission des données est alors partagée entre tous les processus exécutant le code parallèle, ce qui accentue les problèmes de latence de la mémoire.

Le constat est que la parallélisation des algorithmes n'est alors pas toujours évidente. Cette discipline nécessite une connaissance assez fine des dépendances de calcul des algorithmes, mais également des connaissances assez précises du matériel sous-jacent pour pouvoir proposer des optimisations efficaces. De plus, on comprend facilement que la complexité de cette discipline croît avec le nombre de tâches parallèles qu'il est nécessaire d'extraire.

1.4.2 Calculer avec la carte graphique

La multiplication des cœurs de calcul, a eu lieu aussi bien dans la conception des CPU que dans les processeurs graphiques (GPU : Graphics Processing Unit). En effet, les GPU étaient initialement destinés à réaliser les tâches graphiques, qui consistent bien souvent à effectuer un grand nombre de calculs simples et indépendants (comme calculer la couleur de chaque pixel de l'écran). Comme les tâches à réaliser restent spécifiques et limitées par leur nombre, les GPU possèdent des unités de calculs beaucoup plus simples que les cœurs d'un CPU. En effet, on demande à un CPU d'exécuter l'ensemble de ses instructions en un minimum de temps, alors qu'un GPU devra exécuter un nombre restreint d'instructions le plus vite possible. Ceci a permis une multiplication plus importante du nombre d'unités de calcul dans les cartes graphiques, à tel point qu'aujourd'hui ces architectures peuvent contenir des centaines de processeurs spécialisés. Ainsi, en comparant les puissances brutes de calcul (voir figure 1.4), on s'aperçoit qu'il peut être intéressant d'utiliser le processeur graphique pour réaliser des calculs "non-graphiques".

L'idée d'utiliser un GPU comme unité de calcul n'est pas nouvelle et a commencé à germer avec l'arrivée de Direct X. À la fin des années 90, les GPU ajoutent une partie programmable dans la pipeline graphique. En 2003, les GeForces FX font leur apparition, ce sont les premiers GPU à supporter le format de calcul flottant en simple précision. C'est alors que de nombreux langages se sont développés comme par exemple BrookGPU, un langage de programmation qui permet d'exploiter la puissance des cartes graphiques pour des applications de calculs scientifiques. Par la suite, beaucoup d'autres langages de programmation spécifiques sont apparus, comme Cg (C for Graphics) ou GLSL (OpenGL Shading Language). La syntaxe de ces langages est proche du C, mais ils sont clairement destinés à la programmation d'applications graphiques. De plus, ils demandent une très bonne connaissance du fonctionnement interne des GPU pour les utiliser.

1.4.3 Nouveaux langages de programmation

Récemment, les deux principaux fournisseurs de GPU, Nvidia et AMD, ont publié respectivement (CUDA) et (CTM), des langages qui fournissent un accès direct à la programmation de leurs processeurs graphiques. On parle alors de GPGPU (General-Purpose Computation on Graphics Hardware). Ces langages de programmation possèdent un jeu d'instruction complet, comme le calcul en double précision, et ils permettent d'exploiter des milliers de processus (aussi appelé *threads*) simultanément. Depuis l'apparition de ces langages, on assiste à une révolution dans l'informatique car les GPU donnent accès à des

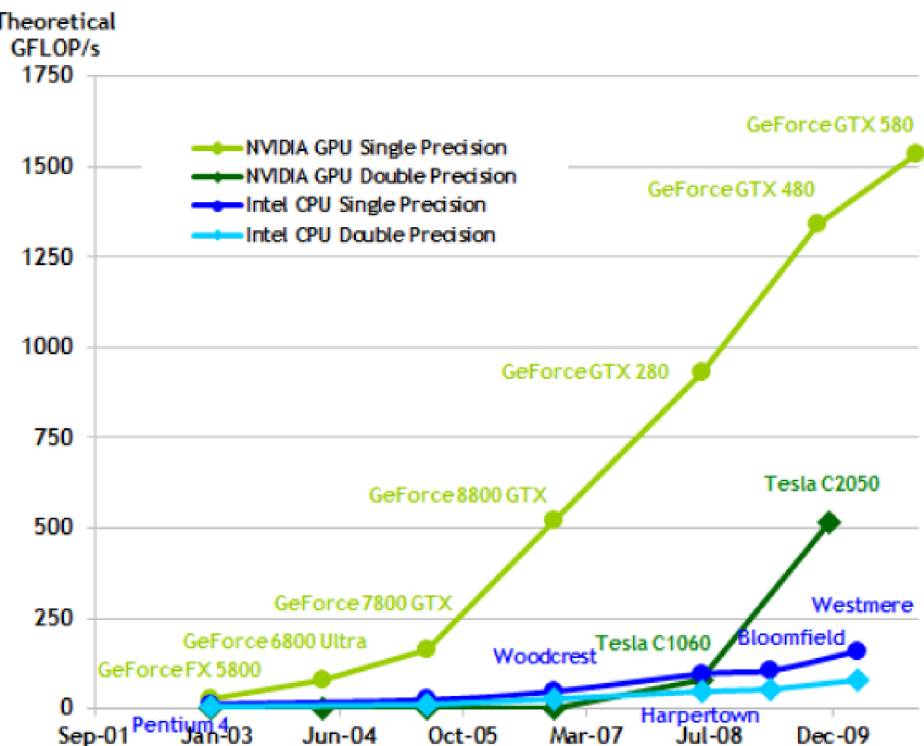


FIG. 1.4 – Évolution de la puissance maximale d'un CPU et d'un GPU[3]

calculateurs hautes performances, accessibles au plus grand nombre et à très faible coût.

En 2009, un fournisseur multi-standard, nommé (OpenCL), a été introduit, avec un modèle de programmation très similaire à CUDA. OpenCL est en réalité une liste de spécifications qui décrit précisément le comportement des instructions données par le programmeur. Chaque vendeur de matériel peut ainsi proposer leur implémentation, et d'une manière transparente un code OpenCL pourra être exécuté sur n'importe quel type de matériel[4]. Dans cette thèse, nous avons choisi de baser toutes nos expériences sur le langage de programmation CUDA. La première raison est qu'au début de ces travaux, OpenCL n'existait pas encore et le support pour CUDA était déjà existant dans SOFA. De plus, les parallélisations OpenCL et CUDA sont très similaires et la transposition devrait être relativement facile.

Le GPGPU a un impact très important sur la façon dont sont développées les applications. Pour exploiter efficacement le GPU, il ne suffit pas de diviser une tâche en une poignée de threads comme sur un CPU multi-cœurs, mais il faut extraire des milliers, de threads en parallèle. En effet, la puissance de la carte graphique tient dans la multiplication des cœurs de calculs, et il est important de tous les utiliser pour atteindre de bonnes performances. La parallélisation se fait donc à grain beaucoup plus fin, ce qui nécessite de revoir complètement les algorithmes existants. La plupart du temps, on devra ré-organiser l'ordre des calculs pour retirer des dépendances, et parfois on devra changer complètement

[3]Source : NVIDIA CUDA Programming Guide

[4]En revanche, on peut s'attendre à ce que les temps de calcul soient légèrement différents, notamment en ce qui concerne les optimisations dépendantes de l'architecture.

les algorithmes existants.

1.5 Contributions

La quantité d'opérations à effectuer pour produire une simulation d'objets déformables, nécessite de trouver un compromis entre la précision des calculs et leur rapidité d'exécution. En effet, toutes les méthodes permettant d'animer des objets déformables sont basées, soit sur des modèles très simples et rapides, soit sur des modèles précis mais lents. D'un autre côté, les nouvelles architectures hautement parallèles comme les GPU, permettent d'atteindre des performances bien plus importantes que les processeurs traditionnels, mais elles sont encore difficiles à exploiter efficacement.

Cette thèse a pour objectif d'apporter des solutions pour tirer parti des nouvelles architectures parallèles (aussi bien les GPU que les CPU multi-cœurs), afin d'augmenter le détail des objets simulés en temps réel. Pour répondre à ce problème, nous devrons dans certains cas ré-organiser les calculs, pour extraire du parallélisme, et dans les cas plus compliqués nous devrons complètement modifier l'algorithme.

Les contributions de ce document sont articulées autour de trois axes principaux. Le premier concerne la déformation des objets, qui repose sur la résolution de systèmes linéaires. Nous proposerons notamment une parallélisation de l'intégrateur implicite, qui inclue l'algorithme du gradient conjugué. Ceci nous permettra d'augmenter considérablement le détail des objets déformables. Nous proposerons également une méthode de préconditionnement asynchrone, qui permet de simuler des objets homogènes et non homogènes avec la même précision et les mêmes performances.

Le second axe concerne la génération, et la simplification des contacts. Ainsi, nous proposerons une méthode de détection des collisions (basée sur une parallélisation GPU) qui permet de déterminer en temps réel les volumes d'intersection entre les objets déformables. Nous proposerons ensuite d'utiliser ces informations pour produire de nouvelles équations (exprimées en volume), pour la réponse aux contacts et le frottement. Nous montrerons également que nos équations parviennent à simuler le même comportement que les méthodes traditionnelles, tout en produisant un système plus simple à résoudre.

Enfin, le dernier axe de recherche porte sur le calcul des forces de réponse aux contacts. Ainsi, nous commencerons par proposer une parallélisation de l'algorithme du Gauss-Seidel, qui nous servira à calculer les forces de réponses. Ensuite nous présenterons un ensemble de contributions qui permettent de poser le problème efficacement, tout en tenant compte des propriétés mécaniques des objets. Nous proposerons également des solutions pour pouvoir prendre en compte les changements topologiques (comme la découpe) au cours de la simulation.

1.6 Plan

Nous présentons maintenant le plan de ce document. Le chapitre 2 est consacré à la description des principes mathématiques et des lois physiques, qui constituent la base théorique des simulations d'objets déformables. Dans ce chapitre nous apporterons également un ensemble de justifications, ainsi qu'une présentation des choix des méthodes qui sont à l'origine de la problématique de cette thèse. De plus, le lecteur intéressé pourra

trouver un ensemble de références vers des travaux similaires, ainsi qu'un ensemble de définitions auxquelles nous ferons référence par la suite.

Les trois chapitres suivants présentent nos contributions, et sont tous construits suivant le même schéma. La première section présente un bref état de l'art spécifique aux problèmes abordés dans le chapitre, puis chaque section présentera une de nos contributions. Le chapitre 3, est consacré à la résolution de systèmes linéaires en temps réel. Le chapitre 4, traite de la détection des collisions et de la formulation des contacts. Le chapitre 5, aborde le problème de la résolution des équations de contacts.

Enfin, le chapitre 6, constitue la conclusion de ce document. Nous présenterons tout d'abord un ensemble de simulations médicales, qui démontrerons l'utilité de nos contributions dans de réelles applications. Enfin, nous présenterons un ensemble de pistes pour poursuivre ces travaux.

Chapitre 2

Simulation d'objets déformables

Table des matières

2.1 Introduction 17
2.2 Mécanique des milieux continus 17
2.2.1 Calcul des tenseurs de déformation 17
Description du mouvement 17
Mesure de la déformation 18
2.2.2 Calcul des tenseurs de contraintes 22
Le postulat de Cauchy 22
2.2.3 Loi de comportement 24
La loi de Hooke 25
Matériaux non linéaires 25
2.2.4 Mesure des forces 26
Formulation variationnelle 27
2.3 La méthode des éléments finis 27
2.3.1 Discrétisation du domaine 27
Fonction de forme et interpolation 29
2.3.2 Résolution numérique 30
Assemblage des matrices 32
Conditions limites 33
Éspace de déformation et espace visuelle 34
2.4 Intégration et résolution du problème 35
2.4.1 Système d'équations statique 36
2.4.2 Systèmes d'équations dynamique 36
Intégration explicite 37
Intégration Implicite 37
2.5 Les modèles déformables 39
2.5.1 Modèles basiques 39
2.5.2 Modèles éléments finis volumiques 40
2.5.3 Découpe et changements topologiques 44
2.6 Programmation moderne des GPU 45
2.6.1 L'architecture d'un GPU 45
2.6.2 Programmer en CUDA 46
Limitation du langage 48
2.7 Conclusion 49

2.1 Introduction

Ce chapitre est consacré à l'étude et à la description des bases théoriques, sur lesquelles sont fondés les algorithmes permettant d'animer les objets virtuels. Ainsi, nous commencerons par présenter brièvement les formulations physiques qui décrivent le comportement de la matière constituant les objets déformables, puis nous présenterons la méthode des éléments finis, qui nous permettra de résoudre ces équations. Nous poursuivrons en étudiant l'évolution des systèmes au cours du temps, et nous présenterons quelques modèles de déformations qui sont très largement utilisés dans les simulations en temps réel. Enfin, nous terminerons en présentant les contraintes des modèles de programmation des nouvelles architectures parallèles.

Aucune contribution ne sera présentée ici, cependant il nous a paru important de poser clairement les équations à l'origine des travaux de cette thèse. L'objectif est de fournir les explications les plus simples et compréhensibles possibles, tout en donnant au lecteur une notion intuitive des concepts physiques utilisés. Évidemment, nous ne fournirons pas une démonstration rigoureuse de chacune des équations, mais nous fournirons dans la mesure du possible des références vers des ouvrages appropriés. La lecture de ce chapitre permettra de mieux situer les contributions qui seront présentées par la suite, mais permettra également de présenter les choix et les méthodes qui sont à l'origine de la problématique de cette thèse.

2.2 Mécanique des milieux continus

Dans la réalité, il est bien connu que les tissus vivants sont composés de molécules et d'atomes qui sont eux-mêmes composés essentiellement de vide. En revanche, à une échelle macroscopique la matière nous apparaît comme lisse et continue, elle semble remplir intégralement l'espace qu'elle occupe. Malgré le fait que cette hypothèse soit erronée, elle a permis aux physiciens de produire des modèles mathématiques qui prédisent avec une grande précision le comportement des objets réels, à condition de regarder les déformations à une échelle macroscopique. Ces théories possèdent de nombreuses déclinaisons et s'appliquent aussi bien à la mécanique des fluides qu'à la mécanique des solides subissant des déformations plastiques ou élastiques... Dans cette thèse nous nous intéressons aux matériaux qui possèdent un comportement élastique, c'est-à-dire les matériaux qui se déforment sous l'application de contraintes puis retournent à leur état initial lorsque l'on supprime les sollicitations. Nous proposons au lecteur de se référer à Nealen et al. (2006) qui fournit un bon résumé des notions introduites dans ce chapitre.

2.2.1 Calcul des tenseurs de déformation

Description du mouvement

Dès lors que l'on souhaite décrire un mouvement, il est nécessaire de procéder à son repérage dans l'espace, et au cours du temps. Il existe principalement deux descriptions pour obtenir la valeur d'une grandeur physique à un instant donné.

Description lagrangienne La *description lagrangienne* est la plus intuitive, elle permet de déterminer pour chaque particule, sa position et ses propriétés physiques à partir de leur

configuration initiale. Ainsi, la position d'une particule $\boldsymbol{p}$ à un instant t, qui se trouvait à la position $\overline{\boldsymbol{p}}$ à l'instant 0 est obtenue par une relation de type :

$$\boldsymbol{p} = \Phi(\overline{\boldsymbol{p}}, t), \quad \overline{\boldsymbol{p}} = \Phi(\overline{\boldsymbol{p}}, 0) \tag{2.1}$$

Cette description permet en quelque sorte de "suivre" chaque particule pendant un déplacement ou une déformation. Elle est par conséquent particulièrement bien adaptée à l'étude des déformations solides élastiques.

Description eulérienne À l'opposé, la *description eulérienne* décrit pour chaque point de l'espace les variations des propriétés mécaniques étudiées. Les hypothèses de continuités assurent qu'en n'importe quel point de l'espace, on peut mesurer une grandeur physique d'un objet. On peut par exemple, obtenir la vitesse au point $\boldsymbol{m}$ de l'espace par :

$$\boldsymbol{v}(\boldsymbol{m}, t) \tag{2.2}$$

À des instants différents, la vitesse au point $\boldsymbol{m}$ sera donnée par la vitesse de différentes particules, mais qui se trouvaient en $\boldsymbol{m}$ au moment de la mesure. Cette description est en général mieux adaptée aux modèles de fluides, puisque dans ce cas on cherchera uniquement à connaître les modifications de vitesse ou densité à un endroit donné.

Mesure de la déformation

En description lagrangienne, une mesure de la déformation en deux point $\overline{\boldsymbol{p}}_1$ et $\overline{\boldsymbol{p}}_2$ infiniment proches, peut être vue comme la variation de leurs distances respectives. Dans la configuration initiale, on peut évaluer cette distance par :

$$d\overline{\boldsymbol{p}} = \| \overline{\boldsymbol{p}}_2 - \overline{\boldsymbol{p}}_1 \| \tag{2.3}$$

De même, on peut mesurer cette distance dans la configuration déformée :

$$d\boldsymbol{p} = \| \boldsymbol{p}_2 - \boldsymbol{p}_1 \| = \| \Phi(\overline{\boldsymbol{p}}_1 + d\overline{\boldsymbol{p}}, t) - \Phi(\overline{\boldsymbol{p}}_1, t) \| \tag{2.4}$$

Intuitivement, on peut dire que plus $d\boldsymbol{p}$ (la distance dans la configuration initiale) est différente de $d\overline{\boldsymbol{p}}$ (la distance dans la configuration déformée), plus l'objet s'est déformé pendant le mouvement. Voyons comment obtenir une description plus précise.

Champ de déplacement Il est possible de déterminer pour chaque particule dans la configuration initiale un vecteur de déplacement $\boldsymbol{u}$ qui décrit sa position dans la configuration déformée.

$$\boldsymbol{u}(\overline{\boldsymbol{p}}, t) = \Phi(\overline{\boldsymbol{p}}, t) - \overline{\boldsymbol{p}} \tag{2.5}$$

L'ensemble de ces vecteurs, décrit alors un *champ de déplacement* (voir figure 2.1). La connaissance de celui-ci permet de caractériser intégralement le mouvement du solide, mais il ne permet pas de définir la notion de déformation que subissent les particules. En effet, prenons par exemple un objet rigide, tout mouvement de celui-ci produit un champ de déplacement non nul, alors qu'il ne subit aucune déformation. Nous avons besoin d'un outil de mesure plus avancé.

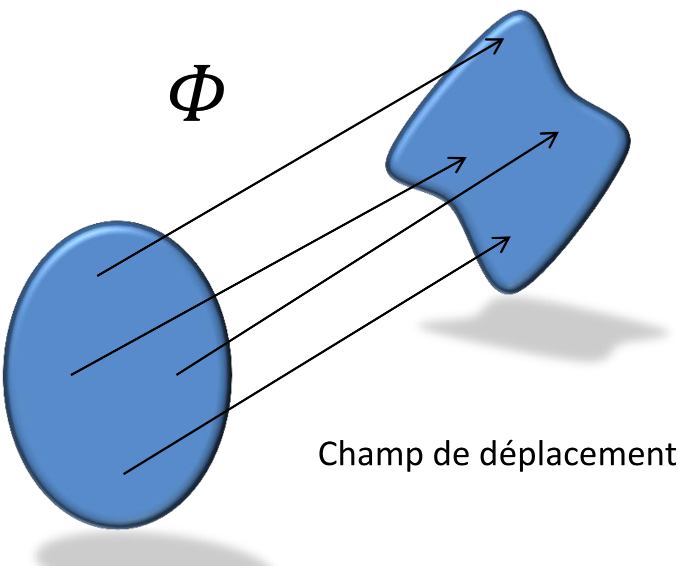


FIG. 2.1 – Champ de déformation décrit par une fonction bijective, qui associe à tout point de l'espace de départ, un point dans l'espace déformé.

Tenseur de déformation Nous introduisons la notion de *tenseur de déformation* notée $\boldsymbol{\varepsilon}$, qui traduit les déformations locales (élongation, compression, flexion, torsion ou cisaillement) du vecteur $d\overline{\boldsymbol{p}}$ en un point donné du domaine. L'opération de multiplication du tenseur de déformation permet d'associer à tout vecteur, un autre vecteur. On peut alors considérer cette opération comme une fonction de transformation du vecteur initial vers le vecteur déformé.

Le gradient de déformation Une première solution pour calculer la variation des distances entre les particules est de calculer le *gradient de déformation*, il se calcule de la façon suivante :

$$\mathbb{F} = \text{grad}(\Phi) = \frac{\partial \Phi}{\partial \overline{\boldsymbol{p}}} \tag{2.6}$$

Le gradient consiste alors à dériver dans toutes les directions la fonction Φ, il décrit ainsi la façon dont le vecteur $\boldsymbol{u}$ varie dans l'espace. En remplaçant (2.6) dans l'équation (2.4) on peut alors écrire :

$$d\boldsymbol{p} = \mathbb{F}\ d\overline{\boldsymbol{p}} \tag{2.7}$$

On constate que le gradient de déformation est un tenseur, puisqu'il permet d'associer à tout vecteur de la configuration initiale un autre vecteur dans la configuration déformée. Cependant, si aucune déformation n'est créée pendant le mouvement, ce tenseur n'est pas nul et vaut l'identité. Ainsi, on utilise souvent des outils plus avancés pour caractériser la déformation (le lecteur intéréssé peut trouver une justification dans Garrigues (2001)). Nous présentons maintenant les principaux tenseurs de déformation.

Le tenseur de Cauchy-Green Prenons les deux particules $\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ infiniment proches dans la configuration déformée, qui sont séparées par une distance $d\boldsymbol{p}$. En exprimant le carré de cette distance on obtient :

$$\begin{aligned}(d\boldsymbol{p})^2 &= d\boldsymbol{p} \cdot d\boldsymbol{p} = (\mathbb{F}\ d\overline{\boldsymbol{p}}) \cdot (\mathbb{F}\ d\overline{\boldsymbol{p}}) \\ &= (\mathbb{F}\ d\overline{\boldsymbol{p}})^T (\mathbb{F}\ d\overline{\boldsymbol{p}}) = (d\overline{\boldsymbol{p}}^T\ \mathbb{F}^T)(\mathbb{F}\ d\overline{\boldsymbol{p}}) \\ &= d\overline{\boldsymbol{p}} \cdot (\mathbb{F}^T\ \mathbb{F}\ d\overline{\boldsymbol{p}})\end{aligned} \tag{2.8}$$

On définit ainsi le *tenseur de Cauchy-Green droit*[1], en posant $\boldsymbol{\varepsilon_c} = \mathbb{F}^T\ \mathbb{F}$. La variation de distance des deux points considérés s'écrit alors :

$$(d\boldsymbol{p})^2 = d\overline{\boldsymbol{p}} \cdot (\boldsymbol{\varepsilon_c}\ d\overline{\boldsymbol{p}}) \tag{2.9}$$

$\boldsymbol{\varepsilon_c}$ nous fournit une relation entre les vecteurs de la configuration d'origine vers des vecteurs dans la configuration déformée. De plus, il ne peut être nul que si aucune variation de distance n'est apparue entre $d\boldsymbol{p}$ et $d\overline{\boldsymbol{p}}$. On vient alors de définir un premier tenseur de déformation.

Le tenseur de Green-Lagrange Si on cherche à calculer la différence des carrés des distances dans la configuration initiale et déformée, on obtient :

$$\begin{aligned}(d\boldsymbol{p})^2 - (d\overline{\boldsymbol{p}})^2 &= d\overline{\boldsymbol{p}} \cdot (\boldsymbol{\varepsilon_c}\ d\overline{\boldsymbol{p}}) - (d\overline{\boldsymbol{p}} \cdot d\overline{\boldsymbol{p}}) \\ &= d\overline{\boldsymbol{p}} \cdot (\boldsymbol{\varepsilon_c} - \mathbb{I})\ d\overline{\boldsymbol{p}} \\ &= 2\ d\overline{\boldsymbol{p}} \cdot (\frac{1}{2}(\boldsymbol{\varepsilon_c} - \mathbb{I})\ d\overline{\boldsymbol{p}})\end{aligned} \tag{2.10}$$

En posant $\boldsymbol{\varepsilon_g} = \frac{1}{2}(\boldsymbol{\varepsilon_c} - \mathbb{I})$, on obtient :

$$(d\boldsymbol{p})^2 - (d\overline{\boldsymbol{p}})^2 = 2\ d\overline{\boldsymbol{p}} \cdot (\boldsymbol{\varepsilon_g}\ d\overline{\boldsymbol{p}}) \tag{2.11}$$

Les distances étant positives, $(d\boldsymbol{p})^2 - (d\overline{\boldsymbol{p}})^2$ sont nulles si et seulement si, les deux distances sont égales (ce qui signifie que l'objet n'a subi aucune déformation). On constate alors que dans ce cas $\boldsymbol{\varepsilon_g} = 0$. On définit ainsi le *tenseur de Green-Lagrange*, qui est très largement utilisé dans la simulation d'objets déformables en temps réel. En effet, ce tenseur permet de faire des grandes simplifications dans le cas où on ne considère que de petites déformations.

[1] Par analogie on peut également définir le tenseur de Cauchy-Green gauche $\boldsymbol{\varepsilon_b} = \mathbb{F}\ \mathbb{F}^T$.

En utilisant les équations (2.5) et (2.6), on peut écrire[2] :

$$
\begin{aligned}
\varepsilon_g &= \frac{1}{2}(\mathbb{F}^T\ \mathbb{F} - \mathbb{I}) \\
&= \frac{1}{2}\left(\left(\frac{\partial\Phi(\overline{\boldsymbol{p}},t)}{\partial\overline{\boldsymbol{p}}}\right)^T \frac{\partial\Phi(\overline{\boldsymbol{p}},t)}{\partial\overline{\boldsymbol{p}}} - \mathbb{I}\right) \\
&= \frac{1}{2}\left(\left(\frac{\partial(\boldsymbol{u}(\overline{\boldsymbol{p}},t)+\overline{\boldsymbol{p}})}{\partial\overline{\boldsymbol{p}}}\right)^T \frac{\partial(\boldsymbol{u}(\overline{\boldsymbol{p}},t)+\overline{\boldsymbol{p}})}{\partial\overline{\boldsymbol{p}}} - \mathbb{I}\right) \\
&= \frac{1}{2}\left(\left(\frac{\partial\boldsymbol{u}(\overline{\boldsymbol{p}},t)}{\partial\overline{\boldsymbol{p}}}\right)^T \frac{\partial\boldsymbol{u}(\overline{\boldsymbol{p}},t)}{\partial\overline{\boldsymbol{p}}} + \left(\frac{\partial\boldsymbol{u}(\overline{\boldsymbol{p}},t)}{\partial\overline{\boldsymbol{p}}}\right)^T + \frac{\partial\boldsymbol{u}(\overline{\boldsymbol{p}},t)}{\partial\overline{\boldsymbol{p}}}\right) \\
&= \underbrace{\frac{1}{2}\left(\mathrm{grad}(\boldsymbol{u})^T \overline{\otimes}\, \mathrm{grad}(\boldsymbol{u})\right)}_{\text{Composante non linéaire}} + \underbrace{\frac{1}{2}\left(\mathrm{grad}(\boldsymbol{u})^T + \mathrm{grad}(\boldsymbol{u})\right)}_{\text{Composante linéaire}}
\end{aligned}
\tag{2.12}
$$

On vient de ré-écrire le tenseur de Green-Lagrange sous sa forme conventionnelle. On constate alors qu'il est possible de séparer une composante linéaire, d'une composante non-linéaire. Or, si les déformations restent petites par rapport à la taille des objets, $\mathrm{grad}(\boldsymbol{u})\overline{\otimes}\mathrm{grad}(\boldsymbol{u})^T$ est négligeable devant $\mathrm{grad}(\boldsymbol{u}) + \mathrm{grad}(\boldsymbol{u})^T$. Ainsi, on peut simplifier ce tenseur en le linéarisant :

$$\varepsilon_{g'} = \frac{1}{2}\left(\mathrm{grad}(\boldsymbol{u})^T + \mathrm{grad}(\boldsymbol{u})\right) \tag{2.13}$$

Le tenseur de Green-Lagrange linéarisé est alors beaucoup plus rapide à calculer. Il est très largement utilisé pour simplifier la résolution des équations. Cependant, dès lors que l'on s'éloigne de l'hypothèse de petits déplacements, la version linéarisée devient rapidement fausse.

Les autres tenseurs de déformation Il existe de nombreuses autres façons de calculer un tenseur de déformation. On peut par exemple citer le tenseur d'Euler-Almansi, le tenseur de Biot, le tenseur de Hill. Nous proposons au lecteur intéressé de se reporter aux nombreux livres traitant de la mécanique des milieux continus, comme par exemple Bonet et Wood (2005); Garrigues (2001), pour plus d'informations.

La diversité des tenseurs de déformations s'explique par le fait que ces outils doivent fournir une description la plus simple possible, des déformations en fonction des déplacements. Or, en raison de la présence de non linéarités (comme les rotations), cette évaluation est difficile et complexifie largement les calculs Choi et Ko (2005); Georgii et Westermann (2006). C'est d'ailleurs toujours un domaine de recherche très actif, sur lequel les informaticiens travaillant dans le domaine de la simulation, sont susceptibles de contribuer. On peut trouver dans Müller et al. (2002), une comparaison de différents tenseurs de déformations, ainsi que leur implication sur le comportement des objets.

[2] $\overline{\otimes}$ est le produit tensoriel contracté une fois.

2.2.2 Calcul des tenseurs de contraintes

À ce stade, nous venons d'expliquer comment à partir du déplacement d'un point on peut en déduire une mesure de la déformation. Ceci indépendamment des causes à l'origine de ce mouvement. Dans la suite, introduisons l'autre aspect de la mécanique, c'est-à-dire de posséder une mesure des forces internes appliquée ponctuellement dans le solide.

Dans le monde réel, la déformation des objets est soumise à des contraintes, par exemple l'impossibilité d'étirer indéfiniment un objet, ou encore de le comprimer à l'infini. Ces contraintes génèrent des forces à l'intérieur de la matière qui permettent de maintenir la cohérence du milieu. Un objet est soumis à deux types d'efforts : les efforts exercés par les systèmes extérieurs, ainsi que les efforts internes au système lui-même. Les forces extérieures (notées $\boldsymbol{f}_{\text{ext}}$) telle que la gravité où les forces d'inerties sont en général connues, et leur modélisation ne pose pas de problème. Les efforts intérieurs sont plus problématiques et nécessitent de poser des hypothèses supplémentaires.

Le postulat de Cauchy

Le postulat de Cauchy fait l'hypothèse que les efforts internes d'un objet peuvent être représentés par des forces surfaciques, qui ne dépendent du domaine que par leur normale extérieure. En d'autres termes, pour tout point $\boldsymbol{m}$ du domaine, on peut imaginer un plan de coupe qui sépare l'objet en deux et qui passe par $\boldsymbol{m}$ (voir figure 2.2). On peut alors avoir une représentation de la contrainte en ce point par un vecteur du type $\vec{\boldsymbol{c}}(\boldsymbol{m}, \vec{\boldsymbol{n}})$, appelé *vecteur contrainte* en $\boldsymbol{m}$ dans la direction $\vec{\boldsymbol{n}}$. Le postulat de Cauchy permet de considérer le rayon d'action des efforts internes suffisamment faible, pour qu'on puisse se limiter à l'étude de leurs actions dans un voisinage très proche.

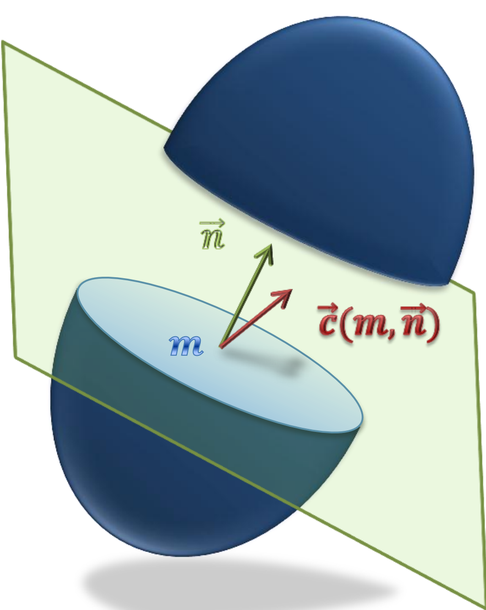


FIG. 2.2 – Évaluation du vecteur contrainte à l'aide d'un plan de coupe, pour définir les forces internes au point M.

Les composantes du vecteur contrainte sont homogènes à une pression, c'est-à-dire qu'elles ont la dimension d'une force par unité de surface. La direction de la contrainte n'est pas forcément la même que $\vec{n}$, et on définit la contrainte normale $\sigma_{\vec{n}}$ comme la projection de $\vec{c}(m, \vec{n})$ sur $\vec{n}$, et le cisaillement comme la projection de $\vec{c}(m, \vec{n})$ sur le plan de coupe. Si la contrainte normale est positive, cela se traduit localement par un état de traction, sinon on parle d'état de compression.

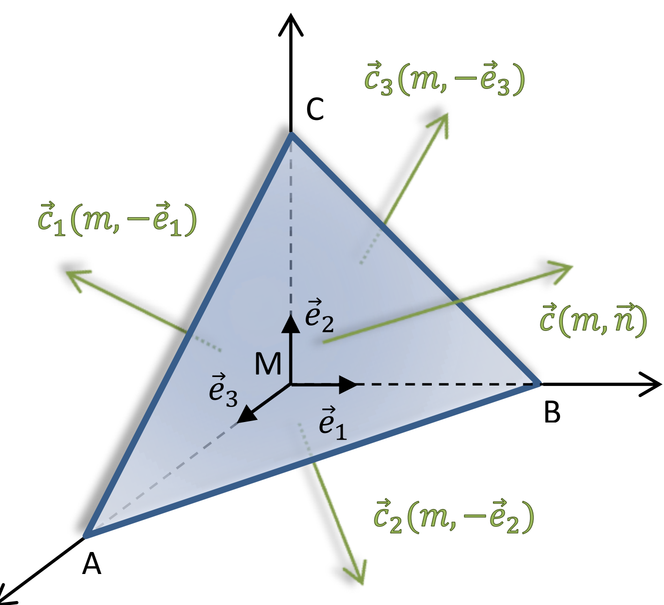


FIG. 2.3 – Équilibre des forces sur un tétraèdre aligné dans le repère cartésien.

Le vecteur contrainte ne suffit pas à caractériser la contrainte, car il dépend directement de la normale au plan de coupe (voir annexe A). Comme précédemment, on va alors transposer la notion de force ponctuelle appliquée à un point de l'objet, en un *tenseur de contraintes*. Considérons l'infime portion de matière contenue à l'intérieur d'un tétraèdre (voir figure 2.3), on peut exprimer l'équilibre des forces en son centre par :

$$\vec{c}(m, \vec{n}) = [\vec{c}_1(m, \vec{e}_1), \vec{c}_2(m, \vec{e}_2), \vec{c}_3(m, \vec{e}_3)] \overline{\otimes} \vec{n} \tag{2.14}$$

En posant $\sigma_c = [\vec{c}_1(m, \vec{e}_1), \vec{c}_2(m, \vec{e}_2), \vec{c}_3(m, \vec{e}_3)]$ on définit le *tenseur de contraintes de Cauchy*, et on obtient :

$$\vec{c}(m, \vec{n}) = \sigma_c \overline{\otimes} \vec{n} \tag{2.15}$$

On vient d'écrire que le vecteur de contraintes en un point quelconque est complètement déterminé par la connaissance des trois vecteurs de contraintes orthogonaux. σ_c fournit alors une mesure de la contrainte indépendamment du plan de coupe.

Autres tenseurs de contraintes Le tenseur de contraintes de Cauchy, s'exprime à partir de données de la configuration courante, qui n'est pas forcément connue à l'avance. Ainsi, le *premier tenseur de Piola–Kirchhoff* σ_p, qui permet d'exprimer les contraintes à partir des

positions de référence, mais toujours en utilisant les surfaces de la configuration déformée. Celui-ci s'exprime en fonction du gradient de déformation, et du tenseur de cauchy.

$$\boldsymbol{\sigma_p} = \mathbf{J} \cdot \boldsymbol{\sigma_c} \cdot \mathbb{F}^{-T} \tag{2.16}$$

où $\mathbf{J} = \det \mathbb{F}$, qui est une matrice qui permet de passer de la configuration initiale vers la configuration déformée. Cependant, le fait que ce tenseur ne soit pas symétrique et qu'il utilise toujours les surfaces de la configuration déformée, le rend difficile à utiliser. On définit alors le *deuxième tenseur de Piola–Kirchhoff* qui permet de définir les contraintes dans la configuration de référence, à partir des surfaces de référence. Il s'écrit :

$$\boldsymbol{\sigma_s} = \mathbf{J}\mathbb{F}^{-1} \cdot \boldsymbol{\sigma_c} \cdot \mathbb{F}^{-T} \tag{2.17}$$

Ce tenseur est alors symétrique par construction, et il est majoritairement utilisé dans les simulations. Ces tenseurs expriment en réalité la contrainte dans des configurations différentes (dans la configuration déformée pour cauchy et dans la configuration initiale pour Piola–Kirchhoff). Il sont donc différents des tenseurs de déformation, au sens où ces derniers faisaient des approximations sur les non-linéarités des déplacements.

2.2.3 Loi de comportement

On vient de décrire deux outils mathématiques qui permettent de mesurer en un point précis d'un volume continu, la déformation $\boldsymbol{\varepsilon}$ et la contrainte $\boldsymbol{\sigma}$, qui s'appliquent à un instant donné en ce point. En réalité, ces deux notions sont étroitement liées et il est impossible de les dissocier. En effet, tout déplacement d'une particule va générer des forces, qui elles-mêmes vont contraindre le déplacement (voir figure 2.4). Cette relation dépend bien évidemment des propriétés mécaniques du matériau considéré. Elle est souvent obtenue directement par des expérimentations.

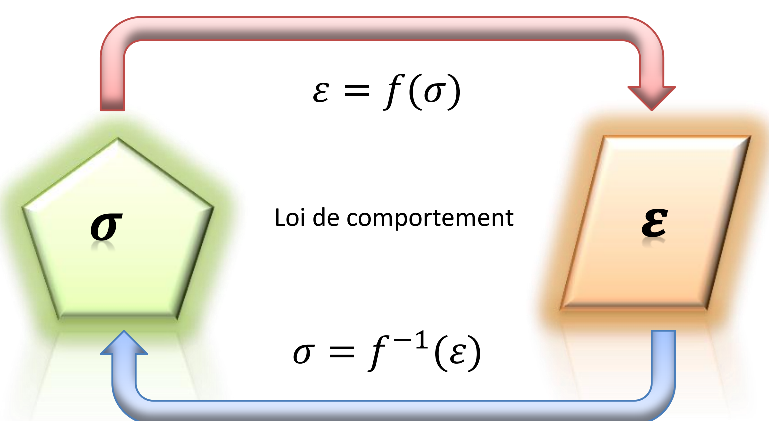


FIG. 2.4 – Les déplacements et les contraintes, sont liés par une loi de comportement.

La force agissant au sein de la matière ne peut être perçue qu'au travers de ses effets, autrement dit, elle ne peut pas être mesurée directement. Le principe est alors d'imposer

une force connue sur un échantillon de matière et de mesurer la déformation résultante. Évidemment, les contraintes résultantes ne peuvent pas être obtenues indépendamment pour tous les points de la matière, et on considère alors une valeur moyenne s'exerçant sur la totalité de l'échantillon. La seconde difficulté est qu'il faut se procurer un échantillon de matière, ce qui est souvent problématique lorsqu'il s'agit d'organes. Par ailleurs, l'échantillon peut subir des dégradations pendant les expériences. Il n'est alors possible de ne connaître que partiellement les informations dont nous avons besoin. Le lecteur intéressé pourra trouver des informations complémentaires dans Cotin (1997).

La loi de Hooke

La plus simple des relations liant la contrainte à la déformation, est donnée par la loi de Hooke. C'est une loi d'élasticité linéaire, qui propose de considérer que l'allongement est directement proportionnel à la force. On suppose alors qu'il existe une relation linéaire entre ces deux notions, ce qui est une bonne approximation pour de petites déformations (voir figure 2.5). On la note souvent :

$$\boldsymbol{\sigma} = 2\mu\boldsymbol{\varepsilon} + \lambda\,\mathrm{tr}(\boldsymbol{\varepsilon})\mathbf{I} \tag{2.18}$$

λ et μ sont les *coefficients de Lamé*, qui peuvent être déduits du *module de Young* E et du *coefficient de Poisson* ν, qui sont eux-mêmes directement obtenus par les expérimentations. Le coefficient de Poisson, compris entre 0 (parfaitement compressible) et $0,5$ (parfaitement incompressible) décrit la compressibilité du matériau alors que le module de Young E nous donne une mesure de la rigidité[3]. La linéarité de cette loi nous permet d'écrire un relation du type :

$$\boldsymbol{\sigma} = \mathbf{E} * \boldsymbol{\varepsilon} \tag{2.19}$$

où $\mathbf{E}$ est une matrice de dimension 6×6, qui contient 21 scalaires définis par les propriétés du matériau, Duriez (2004).

Matériaux non linéaires

Bien que l'hypothèse de linéarité reste une bonne approximation dans le cas de petites déformations, elle peut introduire un biais conséquent des grandes déformations. En effet, on comprend facilement que sous de fortes sollicitations, la relation liant les contraintes à la déformation, ne suit plus une loi purement linéaire comme stipulée par la loi de Hooke. Pour modéliser de tels phénomènes, on doit utiliser une relation où l'expression de la contrainte dépend alors de la déformation actuelle. Évidemment, ces relations sont beaucoup plus difficiles à formaliser et à évaluer. En revanche, il est établi que l'énergie W nécessaire à la déformation de tous matériaux hyperélastiques (élasticité non-linéaire) peut être caractérisée par la relation suivante :

$$\boldsymbol{\sigma} = \frac{\partial W(\boldsymbol{\varepsilon})}{\partial \boldsymbol{\varepsilon}} \tag{2.20}$$

Évidement, W peut être une fonction très complexe, mais on possède ici une loi de comportement, au sens où elle permet de lier la contrainte à la déformation. En règle générale

[3]Pour la relation de Hooke, E correspond à la pente de courbe qui lie la déformation à la contrainte.

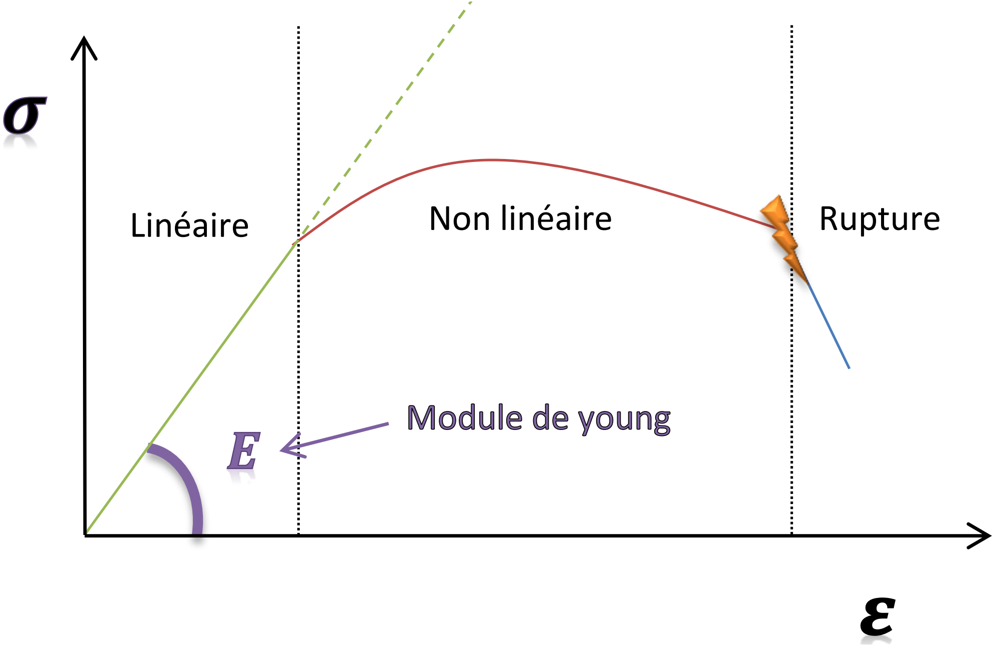


Fig. 2.5 – Linéarisation de la relation entre déformation et contrainte, suivant la loi de Hooke.

plus la loi de comportement est précise, et plus les calculs sont coûteux, et on a souvent tendance à utiliser des relations très simples dans les simulations en temps réel. La loi de comportement introduit alors une deuxième source de non-linéarité dans nos équations. Elle constitue par ailleurs, le second composant sur lequel les recherches sont portées, en vue de produire des simulations d'objets déformables en temps réel Hetnarski et Ignaczak (2006); Marchesseau et al. (2010).

2.2.4 Mesure des forces

Si on résume ce qu'on a présenté jusqu'ici, on a tout d'abord défini un tenseur de déformation qui permet de lier une déformation à un déplacement. Nous avons ensuite défini une mesure de la contrainte, et nous avons ajouté une loi de comportement, qui fournit une relation entre la contrainte et la déformation (et donc, également en fonction des déplacements). Pour pouvoir équilibrer les forces internes avec les forces externes, il suffit d'exprimer l'équilibre des forces sur le domaine :

$$\mathrm{div}(\sigma) + \boldsymbol{f}_{\mathrm{ext}} = \rho\, \ddot{\boldsymbol{u}} \tag{2.21}$$

où ρ est la masse volumique et $\boldsymbol{f}_{\mathrm{ext}}$ les forces extérieures connues (comme la gravité par exemple). Cette équation décrit le principe fondamental de la dynamique au niveau de chaque particule du solide, elle est aussi appelée formulation forte du problème. L'inconvénient avec cette formulation est que l'on peut difficilement distinguer les forces qui s'appliquent sur la frontière du domaine (contacts, pressions), des forces de volume (gravité, termes inertiels). On va alors introduire la formulation variationnelle qui est équivalente à la formulation forte, sous certaines conditions.

Formulation variationnelle

La première étape pour supprimer l'opérateur de divergence, consiste à ajouter de chaque côté des fonctions de test Ψ, puis d'intégrer le résultat sur le domaine occupé par l'objet (noté Ω). On écrit alors la forme variationnelle, où *formulation faible* du problème :

$$\int_{\Omega} \Psi(\boldsymbol{m}) \overline{\otimes} \operatorname{div}(\boldsymbol{\sigma}) dV + \int_{\Omega} \Psi(\boldsymbol{m}) \overline{\otimes} \boldsymbol{f}_{\text{ext}} dV = \int_{\Omega} \Psi(\boldsymbol{m}) \overline{\otimes} \rho \, \ddot{\boldsymbol{u}} \, dV \tag{2.22}$$

Cette équation fournit la même solution que l'équation (2.21), sous réserve qu'elle soit vérifiée quelque soit le champ vectoriel $\Psi(\boldsymbol{m})$, en tout point $\boldsymbol{m}$. Dans le terme de gauche, on retrouve encore l'opérateur de divergence. Cette fois-ci on va utiliser le théorème de *Green Ostrogradski*, qui nous permet d'exprimer ce terme sans faire apparaître l'opérateur de divergence :

$$\int_{\Omega} \Psi(\boldsymbol{m}) \overline{\otimes} \operatorname{div}(\boldsymbol{\sigma}) dV = \int_{\Omega} \boldsymbol{\sigma} \overline{\overline{\otimes}} \operatorname{grad}(\Psi(\boldsymbol{m})) dV + \int_{\partial\Omega} \Psi(\boldsymbol{m}) \overline{\otimes} \boldsymbol{\sigma} \overline{\otimes} \vec{\boldsymbol{n}} \, dS \tag{2.23}$$

En quelque sorte, on vient de "transférer" l'opérateur de divergence du tenseur de contraintes, vers la fonction de test $\Psi(\boldsymbol{m})$. L'avantage, est que si la fonction de test est suffisamment simple, on peut facilement calculer son gradient. De plus, on obtient un terme "surfacique" qui permet plus facilement de gérer les conditions aux limites.

Bien évidemment, ces équations doivent s'appliquer, et être résolues pour l'infinité de points qui constituent la matière du solide. Or, on ne sait pas résoudre une infinité d'équations, hormis sur des cas très simples. On va alors utiliser une méthode de résolution numérique qui va nous permettre d'obtenir malgré tout une bonne approximation de la solution de ce problème.

2.3 La méthode des éléments finis

Dans la section précédente, nous avons décrit un ensemble d'équations physiques qui régissent le comportement des objets. Le premier problème auquel nous sommes confrontés, est que ces équations doivent être intégrées sur l'ensemble du domaine, ce qui se traduit par une infinité d'équations. La *méthode des éléments finis* est une méthode d'approximation qui va nous permettre de contourner ce problème. Nous la présentons dans le cadre de la simulation d'objets déformables, mais c'est en réalité une méthode générale, qui permet de calculer une approximation numérique d'une équation aux dérivées partielles. Nous proposons aux lecteurs intéressés de se reporter à l'ouvrage de Reddy (1984), qui fournit une description en détail des concepts introduits ici.

2.3.1 Discrétisation du domaine

La première étape de la méthode des éléments finis consiste à générer un maillage composé de formes géométriques simples, afin d'obtenir une approximation du volume initial. L'infinité de points constituant la matière, est alors représenté par un nombre fini de points de contrôles qui sont reliés par des éléments. L'idée de base est alors de résoudre les équations de la mécanique au niveau de ces points, puis d'interpoler la solution sur le reste du domaine (à l'intérieur des éléments). Ceci permet de résoudre un nombre conséquent (mais

fini), d'équations (voir figure 2.6). Ainsi, on pourra obtenir une bonne approximation de la solution du problème de départ en résolvant de grands systèmes d'équations.

Il est facile de comprendre que plus le maillage est fin et précis, et meilleur sera la qualité de la solution. L'étape de maillage est alors primordiale dans la méthode des éléments finis, puisqu'elle va dicter d'une part la qualité de la solution obtenue, mais également les performances globales de la simulation. En pratique, nous devrons adapter la résolution du maillage au cas par cas pour maximiser le rapport entre performance et précision. Il est même fréquent d'adapter localement la précision des éléments en fonction des zones d'intérêts.

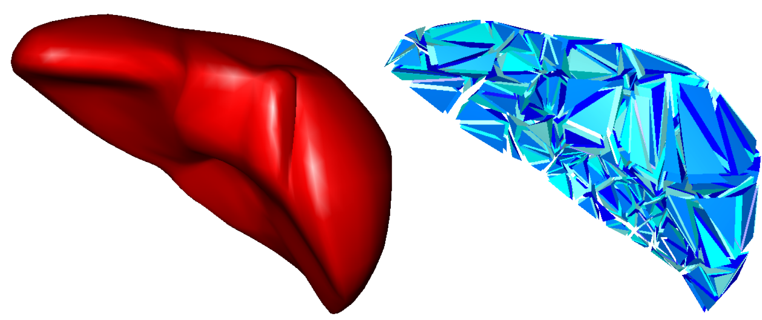


Fig. 2.6 – Surface continue modélisant un foie (à gauche), et un maillage approximant le volume à base de tétraèdre.

Pendant la génération du maillage, il est important de veiller à ce que les éléments ne se recouvrent pas mutuellement. De plus, les éléments doivent recouvrir intégralement le volume de l'objet déformable sans laisser de zones vides. En pratique, on utilise souvent des triangles ou quadrilatères en deux dimensions, et des tétraèdres ou hexaèdres en trois dimensions. En effet, ces formes peuvent se connecter facilement, et sans se recouvrir. Cette propriété est d'ailleurs la seule contrainte à respecter pour générer le maillage d'éléments finis. Il est alors relativement facile d'adapter sa résolution, ce qui se fait de façon quasi indépendante des calculs qui seront réalisés par la suite. Ceci est particulièrement important dès lors que l'on souhaite simuler des formes géométriques complexes comme des organes.

Même si on peut se contenter de maillages grossiers qui sont relativement faciles à générer, les déformations engendrées peuvent ne pas être cohérentes avec la représentation visuelle des objets. Il faut par exemple, veiller à ne pas connecter des éléments si la surface visuelle associée ne l'est pas, sous peine de ne plus être en mesure de les séparer pendant la simulation. L'obtention d'un maillage de bonne qualité, et qui limite le nombre de nœuds générés, est un problème complexe qui diffère largement le cadre de cette thèse. Pour nos expériences, nous avons utilisé des librairies dont la problématique est centrée sur ce sujet. Nous invitons le lecteur intéressé à se reporter à (CGAL; TetGen) pour plus d'informations.

Fonction de forme et interpolation

Maintenant qu'on possède un maillage, on va définir un ensemble de points de contrôle sur lesquels, on va intégrer les équations de la mécanique des milieux continus. La première constatation qu'on peut faire, est qu'il est impératif de connaître la valeur au niveau des nœuds du maillage car ils sont en général partagés par plusieurs éléments. À chacun des nœuds est associé un ensemble de *degrés de liberté*, qui sont les grandeurs physiques qui peuvent varier au cours du temps. Les degrés de liberté peuvent être de différentes natures et de dimension variable[4]. Ils peuvent par exemple représenter des positions (3 composantes en $3D$, qui décrivent les translations selon tous les axes), des rotations (3 composantes en $3D$, qui décrivent les rotations selon tous les axes), ou même une température (1 seule composante scalaire). Pour un objet déformable en $3D$, on va associer 3 degrés de liberté à chacun des points du maillage. Son nombre de degrés de liberté est donc fini, et c'est ainsi qu'on va pouvoir résoudre les équations de la section précédente.

On va alors définir un ensemble de fonctions d'interpolation qu'on va utiliser pour répartir les grandeurs physiques au travers d'un élément. Ces fonctions, sont appelées *fonctions de forme*, ou fonctions de base. On pourra alors évaluer les forces agissant en un point $\boldsymbol{m}$ quelconque d'un élément e avec :

$$\boldsymbol{u}_e(\boldsymbol{m}) = \underbrace{\sum_{i=1}^{n} \eta_e^i(\boldsymbol{m})}_{\mathbf{N}} \cdot {\boldsymbol{u}_e}^i \tag{2.24}$$

où ${\boldsymbol{u}_e}^i$ sont les valeurs mesurées aux nœuds i, et η_e^i sa fonction de forme associée. On peut alors exprimer une matrice $\mathbf{N}$, que nous utiliserons pour transformer une valeur quelconque en sa répartition nodale.

D'autre part, il faut choisir η de façon à ce qu'elle soit interpolante, c'est-à-dire qu'elle vaut 1 au nœud considéré, et 0 aux autres nœuds. De plus, on impose que les fonctions de forme soient nulles en dehors de l'élément, et il faut également s'assurer que les valeurs physiques mesurées aux nœuds soient intégralement réparties à l'intérieur de l'élément. Pour cela, on pose :

$$\sum_{i=0}^{n} \eta_e^i(\boldsymbol{m}) = 1 \tag{2.25}$$

Les fonctions de forme doivent alors être choisies de sorte qu'elles soient toutes linéairement indépendantes à l'intérieur d'un élément. Ceci afin de mesurer l'influence de chaque fonction, de façon indépendante sur les points de contrôle (voir figure 2.7). Bien qu'en théorie, il existe une infinité de fonctions de forme, on utilise très souvent des polynômes, car elles sont simples à dériver et fournissent dans la plupart des cas une précision suffisante.

Il est tout à fait possible d'utiliser des fonctions plus complexes, comme des polynômes quadratiques, ou cubiques pour compenser le manque de degrés de liberté du maillage

[4] Il est également possible de combiner les degrés de liberté, c'est typiquement le cas d'un objet rigide, qui est caractérisé par 6 valeurs (3 en translation et 3 en rotation).

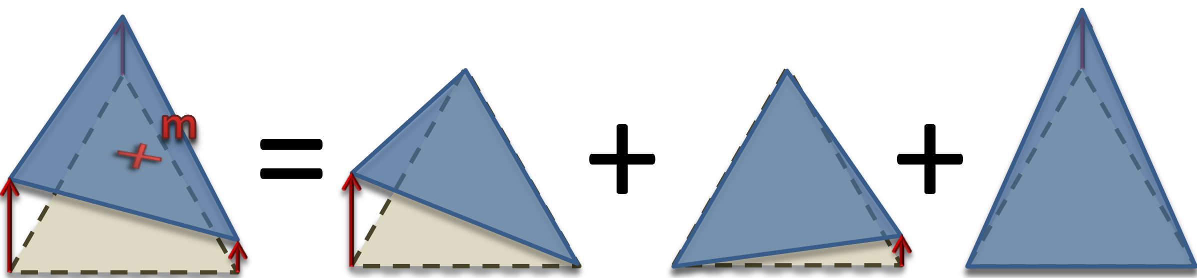


FIG. 2.7 – Décomposition des fonctions de formes définies par des polynômes linéaires.

Grinspun et al. (2002); Kaufmann et al. (2008). Mais, le fait d'utiliser des polynômes plus complexes implique de définir de nouveaux points de contrôle sur l'élément pour satisfaire la contrainte d'indépendance linéaire. Il est évident que le fait d'utiliser des fonctions d'interpolation de degrés supérieurs permet d'obtenir une meilleure solution, au détriment du temps de calcul.

En pratique, nous utiliserons toujours des polynômes linéaires, en supposant que les maillages sont suffisamment détaillés pour fournir une bonne précision. Ainsi, on supposera que les points de contrôles sont définis uniquement aux nœuds des éléments, et les fonctions de formes sont représentées par des polynômes dont le nombre de variables est égal au nombre de points de contrôle de l'élément.

2.3.2 Résolution numérique

Maintenant que nous disposons d'une méthode qui permet d'assurer la continuité d'une grandeur physique au sein du maillage, il nous reste à résoudre les équations de la mécanique des milieux continus, au niveau des nœuds. Pour cela on va utiliser le théorème de Galerkin, qui suggère d'utiliser les fonctions de formes construites précédemment comme fonctions de tests pour la formulation faible. On va donc pouvoir remplacer les fonctions de test $\Psi(\boldsymbol{m})$ définit dans l'équation (2.21), par nos fonctions de forme $\eta(\boldsymbol{m})$. Pour chaque élément e, on obtient l'équation suivante :

$$\underbrace{\int_{V_e} \boldsymbol{\sigma} \,\overline{\overline{\otimes}}\, \mathrm{grad}(\eta(\boldsymbol{m}))dV}_{\text{Rigidité du matériau}} + \underbrace{\int_{\partial V_e} \eta(\boldsymbol{m}) \,\overline{\otimes}\, \boldsymbol{\sigma} \,\overline{\otimes}\, \vec{\boldsymbol{n}}\, dS}_{\text{Conditions limites}} + \underbrace{\int_{V_e} \eta(\boldsymbol{m}) \,\overline{\otimes}\, \boldsymbol{f}_{\text{ext}} dV}_{\text{Forces de volume}} = \underbrace{\int_{V_e} \eta(\boldsymbol{m}) \,\overline{\otimes} \rho\, \ddot{\boldsymbol{u}}\, dV}_{\text{Forces d'inertie}} \tag{2.26}$$

On vient d'écrire l'équation principale de la méthode des éléments finis, locale pour chaque élément du maillage. La fin de cette section est alors consacrée à la description de chaque terme, ainsi qu'à la résolution de l'équation sur le volume total de l'objet. Pour cela, il faut ré-écrire la relation de façon à faire apparaître les déplacements en inconnus, ce qui se fait en extrayant un ensemble de matrices qui découlent de ces relations. Il faudra ensuite assembler toutes les matrices élémentaires pour construire le problème final à résoudre. On pourra alors connaître les forces s'exerçant au sein d'un élément, en réponse à tout déplacement de matière dans le solide.

Les composantes les plus simples à calculer, sont les forces de volumes. En effet, il suffit ici d'utiliser les matrices $\mathbf{N}$ des fonctions de formes définies dans l'équation (2.24), pour pouvoir transposer une force s'appliquant en n'importe quel point d'un élément, en une pondération de forces appliquées au niveau des nœuds. On obtient alors directement un vecteur $\boldsymbol{f}_e$ sur chaque point de l'élément.

Matrice de masse On s'intéresse ensuite aux forces d'inerties. En utilisant la matrice des fonctions de formes, on peut exprimer la masse en tout point de l'élément e comme une combinaison de la masse située sur les nœuds. On écrit alors :

$$\begin{aligned} \int_{V_e} \rho_e\, \eta(\overline{\boldsymbol{p}}) \overline{\otimes}\, \ddot{\boldsymbol{u}}_e \, dV &= \int_{V_e} \rho_e\, \eta(\overline{\boldsymbol{p}}) \overline{\otimes}\, \eta(\overline{\boldsymbol{p}}) dV \;\; \ddot{\boldsymbol{u}}_e{}^i \\ &= \underbrace{\int_{V_e} \rho_e \mathbf{N}^T \mathbf{N} \, dV}_{\mathbf{M}_e} \;\; \ddot{\boldsymbol{u}}_e{}^i \end{aligned} \tag{2.27}$$

où $\ddot{\boldsymbol{u}}_e{}^i$ est l'accélération définie au niveau des nœuds, et on en déduit une matrice de masse $\mathbf{M}_e$. Une technique appelée *mass lumping*, consiste à supposer que la masse est intégralement répartie sur les nœuds des éléments. Elle est très fréquemment utilisée, car elle a pour effet de diagonaliser la matrice de masse. Ceci permet d'une part de simplifier son stockage, et d'autre part de faciliter le calcul de son inverse. De plus, l'assemblage des matrices $\mathbf{M}_e$, est alors immédiat, puisqu'il suffit simplement de reporter la somme des masses calculées sur les nœuds des éléments.

Matrice de rigidité Dans le cas général, il n'est pas possible d'écrire explicitement la matrice de rigidité, quelle que soit la loi de comportement et le tenseur de déformation utilisé. En effet, dans le cas général le tenseur de contrainte est lié au tenseur de déformation par une fonction non linéaire (non linéarité du matériau). De même, le tenseur de déformation, est également lié au déplacement par une autre fonction non linéaire (non linéarité géométrique). En pratique, il n'est alors pas possible d'isoler la rigidité du matériau de la composante de déplacement $\boldsymbol{u}$. En effet, la rigidité du matériau est une fonction $\mathcal{K}(\boldsymbol{u})$ qui dépend de la position. À chaque modification du déplacement, la fonction $\mathcal{K}(\boldsymbol{u})$ va alors changer, ce qui va générer un nouveau déplacement.

Pour nous permettre d'écrire le problème sous forme matricielle, on peut utiliser le développement limité suivant :

$$\boldsymbol{f}(\boldsymbol{u} + \partial \boldsymbol{u}) \approx \boldsymbol{f}(\boldsymbol{u}) + \mathbf{K}(\boldsymbol{u})\; \partial \boldsymbol{u} \tag{2.28}$$

Ainsi, quelque soit le modèle de déformation utilisé, la fonction de rigidité peut être linéarisée pour des petits déplacements. Cette supposition mène à un calcul proche de la solution exacte tant que les déplacements restent petits. Dans la plupart des travaux, on considère la matrice $\mathbf{K}(\boldsymbol{u})$ constante pendant toute la durée du pas de temps. Dans cette thèse, on utilise également cette considération, ce qui permet de construire une matrice $\mathbf{K}$ à partir du déplacement actuel. Cependant, pour les matériaux les plus avancés, il sera nécessaire de reconstruire cette matrice à chaque nouveau pas de temps (ce qui implique que la matrice est susceptible de varier au cours du temps). D'un point de vue intuitif, la

matrice $\mathbf{K}$ décrit l'influence que peut avoir un point sur l'ensemble des autres points du maillage. Nous l'utiliserons pour connaître les forces internes crées sur les nœuds par leur déplacement.

Matrice d'amortissement Lorsqu'on dérive les équations, certaines composantes dépendent toujours de la vitesse Cook (1981). On peut alors exprimer une matrice d'amortissement qui va permettre de pénaliser la vitesse, afin de simuler la dissipation d'énergie du système (qui, s'exprime généralement sous forme de chaleur). Les paramètres physiques de l'amortissement sont très difficiles à obtenir par des expériences, et le plus souvent on modélise ces phénomènes de façon simplifiée. Une technique très classique consiste à utiliser l'amortissement de Rayleigh. Cette simplification propose d'exprimer la matrice d'amortissement comme une pondération de la matrice de masse et de la matrice de rigidité.

$$\mathbf{B}_e = \alpha\mathbf{M}_e + \beta\mathbf{K}_e \tag{2.29}$$

Où les coefficients α et β sont appelés respectivement les coefficients de *Rayleigh mass* et *Rayleigh damping*. On peut trouver dans la littérature des techniques pour déterminer ces coefficients de façon plus réaliste Chowdhury et Dasgupta (2003).

Assemblage des matrices

Des constatations précédentes, on peut en déduire que la relation (2.26) peut s'écrire pour chaque élément, sous la forme suivante :

$$\mathbf{M}_e\ddot{\boldsymbol{u}}_e + \mathbf{B}_e\dot{\boldsymbol{u}}_e + \mathbf{K}_e\boldsymbol{u}_e = \boldsymbol{f}_e \tag{2.30}$$

Or, dès qu'on possède plusieurs éléments, il est nécessaire de se rappeler que les nœuds sont en général connectés à plusieurs éléments. Par conséquent on ne peut pas résoudre chacun d'entre eux indépendamment, car la force appliquée sur un nœud doit être la même dans tous les éléments voisins. Ainsi, on va assembler[5] les matrices locales, pour former un grand système matriciel, dont la dimension est égale au nombre de degrés de liberté de l'objet[6]. Les règles d'assemblage pour une matrice élémentaire $\mathbf{S}_e$ quelconque, sont alors données par la relation suivante :

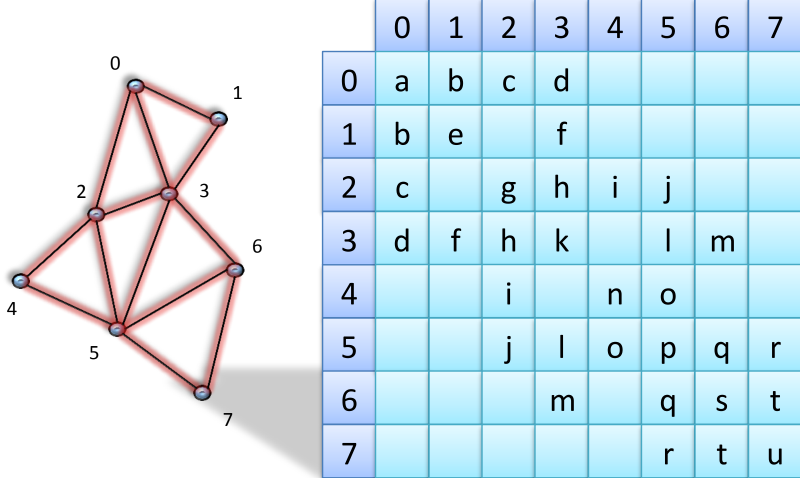
$$\mathbf{S} = \sum_{e=1}^{N_e} \mathbf{G}_e\mathbf{S}_e\mathbf{G}_e^T \tag{2.31}$$

où $\mathbf{G}$ est une matrice de globalisation, qui transforme les indices de l'élément e, en indices dans le système global. En d'autre termes, pour assembler les matrices élémentaires on associe à chaque point une numérotation globale unique, de façon à connaître les indices sur lesquels chaque matrice élémentaire va contribuer. Il suffit alors d'additionner les composantes des matrices locales pour former le système final. Ceci signifie simplement que l'énergie du domaine global est égale à la somme des énergies locales.

Il en résulte qu'une fois que les matrices sont assemblées, elles présentent une structure particulièrement creuse, puisque le taux de remplissage est exactement le même que celui

[5]Notons, que si on utilise la technique de *mass lumping*, ainsi que les coefficients, de Rayleigh pour l'amortissement, seul l'assemblage de la matrice de rigidité est problématique.

[6]c'est-à-dire le produit du nombre de nœuds par le nombre de degrés de liberté de chaque points



	0	1	2	3	4	5	6	7
0	a	b	c	d				
1	b	e		f				
2	c		g	h	i	j		
3	d	f	h	k		l	m	
4			i		n	o		
5			j	l	o	p	q	r
6				m		q	s	t
7						r	t	u

FIG. 2.8 – Assemblage des matrices sur un maillage $2D$. Chaque valeur représente un bloc de données de la taille du nombre de degrés de liberté.

du maillage d'éléments finis. En effet, une valeur non nulle sera présente sur la ligne i et la colonne j, si et seulement si, il y a une arête reliant les sommets i et j dans le maillage initial (voir figure 2.8). Or, même si on utilise des maillages très détaillés, le nombre maximum d'éléments connectés reste en général très petit. Il est alors primordial d'utiliser des structures de stockage informatique adaptées pour maximiser les performances de la simulation (ces aspects sont traités dans le chapitre suivant). De plus, on constate que toutes les matrices sont symétriques par construction, cela traduit par exemple que la rigidité entre deux points est la même dans les deux sens.

Par ailleurs, les matrices élémentaires sont elles-mêmes constituées de sous blocs dont la taille est égale au nombre de degrés de liberté (voir figure 2.9). En effet, chaque sous bloc traduit le déplacement d'un même point dans toutes les directions, et on impose que des indices successifs soient attribués aux degrés de liberté d'un même nœud. L'assemblage des matrices, peut alors se faire par blocs, pour maximiser les performances.

Conditions limites

Il reste à ajouter des conditions limites à notre système. Chaque type de conditions limites à des effets et implémentations différentes, mais en règle générale, elles se traduisent par des équations supplémentaires qu'il faut vérifier pendant la résolution du problème. Il existe deux types de conditions limites qui sont très utilisées dans la plupart des simulations.

La première permet d'attacher une partie des points des objets, de sorte qu'ils ne tombent pas sous la gravité et restent à leur position initiale. Une façon simple pour réaliser cela, consiste à modifier les équations qui imposent un déplacement nul sur un ensemble de points.

$$\boldsymbol{u}(\overline{\boldsymbol{p}}) = 0 \tag{2.32}$$

Cette équation se traduit par le fait de remplacer dans la matrice de rigidité, le couple

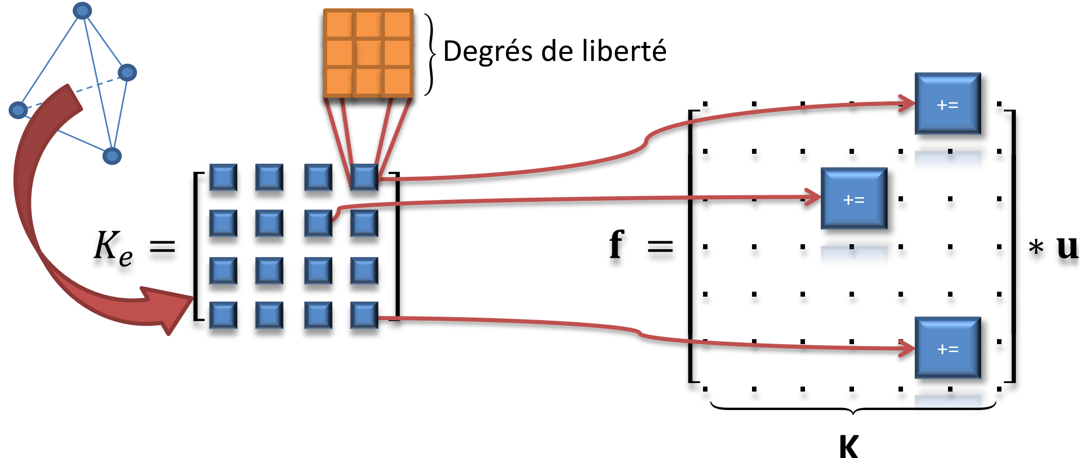


FIG. 2.9 – Assemblage de la matrice globale, en fonction des matrices $\mathbf{K}_e$ locales.

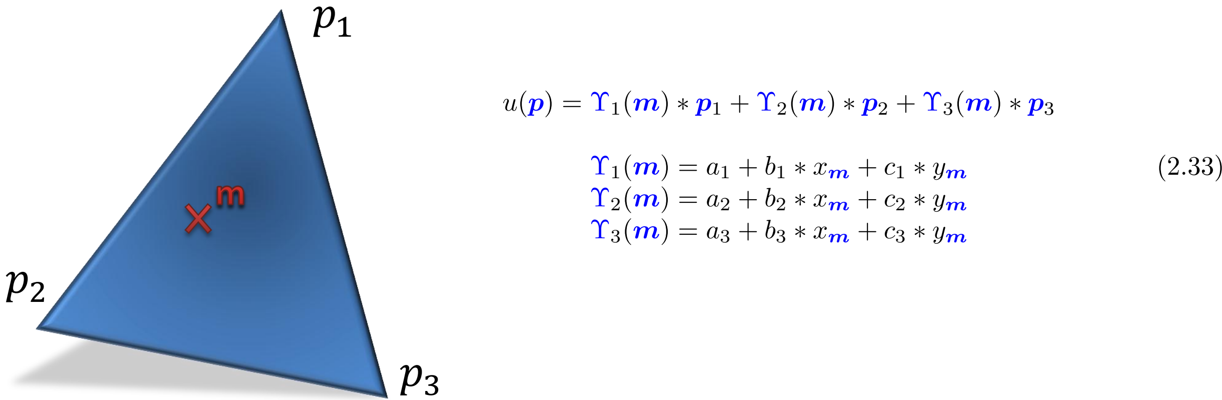
ligne/colonne de l'indice du point fixé par des 0. Du point de vue comportemental, cela signifie que le point ne peut plus influencer, ni être influencé, par les autres points du maillage. Il restera donc fixe, tant qu'aucune force externe ne lui sera appliquée. Une autre façon de procéder consiste à appliquer une force, qui va continuellement ramener le point vers sa position d'origine. L'avantage de cette méthode est qu'elle ne modifie pas le système matriciel, mais elle va autoriser un léger déplacement des points autour de la position initiale.

L'autre type de condition limite qu'il convient de présenter concerne la gestion des contacts, ou plus généralement les forces qui s'appliquent sur la surface des objets. Ces problèmes sont traités en détails dans le chapitre 5, mais on peut juste dire ici que ces forces rentrent dans le cadre des conditions limites qui sont appliquées à la surface des objets.

Éspace de déformation et espace visuelle

Pour terminer, nous voudrions rappeler que le nombre d'équations est directement dicté par le nombre de nœuds du maillage. Or, nous avons vu que pour obtenir de bonnes performances, il est important de maintenir ce nombre le plus bas possible, ce qui est cohérent pour caractériser les déformations mécaniques. En revanche, le fait de possèder un nombre relativement bas de points dans le maillage peut dégrader sensiblement la qualité visuelle des objets.

Pour remédier à ce problème, on utilise souvent plusieurs niveaux de représentations des objets Allard et al. (2007), et on utilise des *fonctions de mappings* qui permettent de passer de l'une à l'autre. Par exemple, sur un triangle en deux dimensions (voir figure 2.10), on définit les coefficients barycentriques du point $\boldsymbol{m}$ par l'équation (2.33). Si l'on cherche à connaître le déplacement du point $\boldsymbol{m}$, par rapport au déplacement du modèle mécanique, il suffit alors d'utiliser la dérivée de la fonction de mapping $\frac{d\Upsilon}{d\boldsymbol{p}}$. Quand le mapping est linéaire, on peut alors exprimer cette fonction sous forme de matrice $\mathbf{J}$, et son application revient à effectuer des opérations matricielles. L'avantage, est que l'on



$$u(\boldsymbol{p}) = \Upsilon_1(\boldsymbol{m}) * \boldsymbol{p}_1 + \Upsilon_2(\boldsymbol{m}) * \boldsymbol{p}_2 + \Upsilon_3(\boldsymbol{m}) * \boldsymbol{p}_3$$

$$\begin{aligned} \Upsilon_1(\boldsymbol{m}) &= a_1 + b_1 * x_{\boldsymbol{m}} + c_1 * y_{\boldsymbol{m}} \\ \Upsilon_2(\boldsymbol{m}) &= a_2 + b_2 * x_{\boldsymbol{m}} + c_2 * y_{\boldsymbol{m}} \\ \Upsilon_3(\boldsymbol{m}) &= a_3 + b_3 * x_{\boldsymbol{m}} + c_3 * y_{\boldsymbol{m}} \end{aligned} \tag{2.33}$$

FIG. 2.10 – Mapping du point $\boldsymbol{m}$ sur un triangle en utilisant ses coordonnées barycentriques.

peut alors utiliser une représentation visuelle qui soit beaucoup plus détaillée, sans perte de performance significative. Notons toutefois que ce principe est exactement le même que celui utilisé pour les fonctions de formes. Lorsqu'elles sont linéaires, tous points d'un élément est représenté comme des coefficients barycentriques de ses sommets, ce qui mène au final aux-mêmes équations.

2.4 Intégration et résolution du problème

À ce stade nous sommes capables de déterminer la force engendrée par un déplacement $\boldsymbol{u}$ au niveau de chaque nœud. Cette force dépend alors de l'état actuel des objets déformables, qui est donnée par un ensemble de positions $\boldsymbol{p}$ et vitesses $\boldsymbol{v}$. Une façon très générique de décrire leur comportement physique, est alors d'utiliser une fonction $\boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v})$ (ou simplement $\boldsymbol{f}(\boldsymbol{u})$), qui nous donne la force en fonction des déplacements. Or, la connaissance de cette force ne suffit pas à produire une simulation. En effet, on souhaite plutôt connaître le résultat inverse, c'est-à-dire déterminer les déplacements en fonction des forces qu'on applique. On pourra alors déplacer les objets, puis calculer de nouvelles forces (contacts, forces extérieures,...), afin d'obtenir un nouveau déplacement. En itérant ce processus, on pourra alors produire les images de la simulation. Dès lors il faut différencier deux cas, les systèmes statiques qui sont consacrés à l'étude de l'état d'équilibre des déformations (déformation à vitesse nulle) et les systèmes dynamiques dont la déformation dépend du temps (effets inertiels non négligeables) . On schématise avec l'équation suivante :

$$\boldsymbol{f}(\boldsymbol{u}) + \boldsymbol{f}_{\text{ext}} = \begin{cases} \nearrow \; 0 & \text{(Statique)} \\ \searrow \; \mathbf{M}\boldsymbol{a} & \text{(Dynamique)} \end{cases} \tag{2.34}$$

où $\boldsymbol{a}$ est un vecteur contenant l'accélération de tous les points du maillage. En utilisant un système statique, on cherche alors à connaître l'état d'équilibre des forces, tandis que

pour les systèmes dynamiques, on va chercher à connaître tous les états transitoires du mouvement. Même si, pour l'étude de notre problème, nous nous situerons toujours dans le cas dynamique, nous proposons de regarder rapidement le cas statique car il permet de ne pas se soucier de la composante temporelle. Nous discuterons du cas dynamique par la suite.

2.4.1 Système d'équations statique

Les systèmes statiques sont utilisés dans les simulations pour lesquelles on ne cherche à connaître que l'état dans la configuration d'équilibre. Dans ce cas, on veut juste savoir quelle sera la déformation des objets une fois qu'ils seront immobiles (quand la somme des forces sera nulle), sans se soucier des états transitoires. Ces systèmes sont par exemple utilisés dans l'étude de la déformation d'un pont, ou encore pour la déformation d'une étagère sous le poids des livres. Pour ce type de problème, on cherche en général une grande précision de la solution. On commence alors par écrire un développement de Taylor pour exprimer la force en fonction, en négligeant les termes d'ordre supérieurs à 1 :

$$\boldsymbol{f}_{\text{ext}} + \boldsymbol{f}(\boldsymbol{u} + \partial \boldsymbol{u}) = \boldsymbol{f}_{\text{ext}} + \boldsymbol{f}(\boldsymbol{u}) + \left[\frac{\boldsymbol{f}(\boldsymbol{u})}{\partial \boldsymbol{u}}\right] \partial \boldsymbol{u} = 0 \tag{2.35}$$

La composante des forces $\boldsymbol{f}_{\text{ext}}$ est connue, et on peut évaluer $\boldsymbol{f}(\boldsymbol{u})$, avec la méthode des éléments finis. On peut alors trouver la solution du problème en itérant avec la méthode de Newton-Raphson. L'idée de base est alors de construire une suite dans laquelle on fait varier le déplacement jusqu'à converger vers une configuration qui annule la somme de force. On prend alors un premier $\boldsymbol{u}_0$ puis on peut évaluer l'erreur par :

$$\left[\frac{\boldsymbol{f}(\boldsymbol{u}_0)}{\partial \boldsymbol{u}}\right] \partial \boldsymbol{u} = -\boldsymbol{f}_{\text{ext}} + \boldsymbol{f}(\boldsymbol{u}_0) \tag{2.36}$$

Si l'erreur est trop grande, on peut calculer un nouveau $\boldsymbol{u}_1 = \boldsymbol{u}_0 + \partial \boldsymbol{u}$, puis on itère le processus jusqu'à atteindre une convergence, Crisfield (1997). Pour chaque itération, il est alors nécessaire de résoudre les équations de déformation, en fonction des nouvelles positions.

2.4.2 Systèmes d'équations dynamique

Les systèmes d'équations dynamiques présentent les mêmes difficultés, avec la contrainte supplémentaire, que la répartition des forces évolue au cours du temps. Pour des raisons de temps de calcul, on applique une seule itération de l'algorithme Newton-Raphson, en pratique cela se trouve être suffisant. Cependant, comme on cherche à connaître tous les états transitoires par lesquels l'objet va passer, il faut définir un pas d'échantillonnage temporel h, qui donne la quantité de temps virtuel écoulée entre deux calculs. Il est important de choisir h de sorte que le temps simulé soit rigoureusement le même que le temps réel, pour qu'un utilisateur en dehors de la simulation perçoive le mouvement des objets à la même vitesse que dans le monde réel. Une fois l'intervalle de temps fixé, il reste à définir un schéma d'intégration pour définir les nouvelles positions en fonction des anciennes.

Intégration explicite

La méthode la plus simple, consiste à utiliser un schéma d'intégration explicite, comme par exemple le schéma d'*Euler explicite*. L'accélération $\boldsymbol{a}_t$ est directement calculée à partir des forces de l'état précédent, et on déduit directement les nouvelles positions et vitesses :

$$\left\{ \begin{array}{lcl} \mathbf{M}\,\boldsymbol{a}_t & = & \boldsymbol{f}(\boldsymbol{p}_t, \boldsymbol{v}_t) \\ \boldsymbol{v}_{t+h} & = & \boldsymbol{v}_t + h\,\boldsymbol{a}_t \\ \boldsymbol{p}_{t+h} & = & \boldsymbol{p}_t + h\,\boldsymbol{v}_t \end{array} \right. \tag{2.37}$$

Le principal avantage est que le processus de résolution ne concerne que la matrice de masse, qui est diagonale si on utilise la technique de mass lumping. Ainsi, les équations du mouvement peuvent être découplées et chaque degré de liberté peut être résolu de manière indépendante. De plus, le processus de résolution est alors très rapide, et il présente un fort potentiel de parallélisme Comas et al. (2008). Cependant, ces méthodes souffrent d'instabilité due au fait qu'on suppose la position et la vitesse connue au début du pas. De plus, une forte contrainte sur ces méthodes est qu'il faut être en mesure de réaliser les calculs au moins aussi rapidement que l'évolution du système mécanique. On constate alors une onde numérique de la propagation de la force qui s'effectue au travers des pas de temps de la simulation. Par exemple, si l'on applique une force sur un point à un instant t, on va alors mettre à jour les positions et vitesses des points de l'élément pendant le pas de temps. Cependant, c'est seulement au temps $t + 1$ que l'on va générer une force dans tous les éléments voisins de niveau 1 (en réponse aux modifications du premier élément) . Ces derniers vont alors mettre à jour les positions et les vitesses des points associés, et ce processus va ensuite se répéter jusqu'à propager la force à tous les éléments.

Dans la réalité, on peut aussi mesurer une onde mécanique qui se propage au sein de la matière. Or plus le matériau est rigide, et plus l'onde se propage rapidement. Il est impératif de propager l'onde numérique plus rapidement que l'onde mécanique sous peine de voir les objets virtuels exploser. Ceci entraîne des restrictions sur la taille des pas de temps, qui doivent être très petits. Cette contrainte est très difficile à garantir dans tous les cas, et les méthodes explicites sont en général mieux adaptées à la simulation d'objets mous, ou pour des algorithmes de recalage. Ainsi, elles ont été utilisées pour des applications en temps réel, visant à simuler l'évolution du cerveau pendant la chirurgie Joldes et al. (2009b). En effet, cet organe est particulièrement mou, et les contacts avec le squelette sont de très faibles intensités. Par ailleurs, les méthodes explicites restent un domaine de recherche actif, et certains travaux tendent à minimiser leurs contraintes. Par exemple Joldes et al. (2009a), qui proposent d'augmenter artificiellement la masse afin de prendre en compte des matériaux plus rigides.

Intégration Implicite

Pour contourner les problèmes des méthodes explicites, le schéma d'intégration implicite permet de mettre à jour les vitesses en fonction de l'accélération à la fin du pas de temps

courant, par exemple, le *schéma d'Euler implicite* :

$$\begin{cases} \mathbf{M}\, \boldsymbol{a}_{t+h} & = & \boldsymbol{f}(\boldsymbol{p}_{t+h}, \boldsymbol{v}_{t+h}) \\ \boldsymbol{v}_{t+h} & = & \boldsymbol{v}_t + h\, \boldsymbol{a}_{t+h} \\ \boldsymbol{p}_{t+h} & = & \boldsymbol{p}_t + h\, \boldsymbol{v}_{t+h} \end{cases} \tag{2.38}$$

On fait face à un problème non-linéaire. Pour le résoudre, on va utiliser une approximation du 1er ordre :

$$\boldsymbol{f}(\boldsymbol{p}_{t+h}, \boldsymbol{v}_{t+h}) \approx \boldsymbol{f}(\boldsymbol{p}_t, \boldsymbol{v}_t) + \mathbf{K} \cdot (\boldsymbol{p}_{t+h} - \boldsymbol{p}_t) + \mathbf{B} \cdot (\boldsymbol{v}_{t+h} - \boldsymbol{v}_t) \tag{2.39}$$

En combinant les équations (2.38) et (2.39), nous pouvons mettre à jour l'accélération[7] avec :

$$\begin{aligned} \mathbf{M}\, \boldsymbol{a}_{t+h} & \approx \boldsymbol{f}(\boldsymbol{p}_t, \boldsymbol{v}_t) + \mathbf{K} \cdot (\boldsymbol{p}_{t+h} - \boldsymbol{p}_t) + \mathbf{B} \cdot (\boldsymbol{v}_{t+h} - \boldsymbol{v}_t) \\ & \approx \boldsymbol{f}(\boldsymbol{p}_t, \boldsymbol{v}_t) + \mathbf{K} \cdot (\boldsymbol{p}_t + h\, \boldsymbol{v}_{t+h} - \boldsymbol{p}_t) + \mathbf{B} \cdot (\boldsymbol{v}_t + h\, \boldsymbol{a}_{t+h} - \boldsymbol{v}_t) \\ & \approx \boldsymbol{f}(\boldsymbol{p}_t, \boldsymbol{v}_t) + \mathbf{K} \cdot (h\, \boldsymbol{v}_t + h^2\, \boldsymbol{a}_{t+h}) + \mathbf{B} \cdot (h\, \boldsymbol{a}_{t+h}) \end{aligned} \tag{2.40}$$

Dans la suite, nous chercherons à égaliser les deux membres de ces équations à chaque pas de temps. Pour résoudre ce problème non-linéaire, on peut utiliser l'algorithme de Newton-Raphson, et effectuer plusieurs itérations pour évaluer le résidu. Cependant, en pratique nous n'effectuerons qu'une seule itération de l'algorithme (ou encore une seule linéarisation par pas de temps) ce qui fourni une solution acceptable dans un contexte de simulation en temps réel. Ainsi, dans la suite du document nous écrirons une égalité entre les deux membres de cette équation :

$$(\mathbf{M} - h\mathbf{B} - h^2\mathbf{K})\ \boldsymbol{a}_{t+h} = \boldsymbol{f}(\boldsymbol{p}_t, \boldsymbol{v}_t) + h\, \mathbf{K} \cdot \boldsymbol{v}_t \tag{2.41}$$

En règle générale, on préfère exprimer ce système en vitesses en posant $\partial \boldsymbol{v} = h\boldsymbol{a}$. Ainsi, on obtient le système linéaire final :

$$\underbrace{(\mathbf{M} - h\mathbf{B} - h^2\mathbf{K})}_{\mathbf{A}}\ \underbrace{\partial \boldsymbol{v}}_{\boldsymbol{x}} = \underbrace{h\boldsymbol{f}(\boldsymbol{p}_t, \boldsymbol{v}_t) + h^2\, \mathbf{K} \cdot \boldsymbol{v}_t}_{\boldsymbol{b}} \tag{2.42}$$

Un système linéaire doit alors être résolu à chaque pas de temps pour pouvoir mettre à jour les nouvelles positions et vitesses. Notons que dans ces équations, nous avons considéré $\mathbf{K}(\boldsymbol{u})$ constante pendant toute la durée du pas de temps, ce qui nous permet de l'écrire sous sa forme matricielle $\mathbf{K}$. En effet, ceci correspond à la linéarisation de la fonction introduite dans la section 2.3.2, ce qui nous permet d'écrire ce système quel que soit le modèle de déformation. On peut montrer que la matrice du système $\mathbf{A}$ est inversible, en raison de l'assemblage avec la matrice de masse. En effet, l'ajout de la masse rend le système défini positif, ce qui assure que le système assemblé, est inversible.

L'avantage d'une intégration implicite est qu'elle permet d'utiliser des pas de temps beaucoup plus grands puisque dès qu'on applique une force, on prend en compte toute la

[7]Nous avons choisi d'utiliser l'accélération comme inconnue, mais il est tout à fait possible de poser les équations au niveau des vitesses ou des positions en utilisant un changement de variable.

mécanique complète de l'objet. Ces systèmes sont dit inconditionnellement stables, mais ils nécessitaient la résolution du système linéaire à chaque pas de temps, ce qui fait qu'il est difficile de garantir la contrainte de temps réel. C'est notamment une grande partie de la problématique de cette thèse. En effet, nous avons fait le choix d'utiliser les méthodes implicites, et tout au long de ce document nous chercherons à apporter des solutions qui visent à réduire les temps de calcul conséquents à ce choix. La première raison est que nous souhaitions pouvoir simuler aussi bien des objets très mous comme le cerveau, que des objets plus rigides comme le foie, ou des instruments chirurgicaux. De plus, nous souhaitions laisser la possibilité à un utilisateur extérieur d'interagir avec la simulation, ce qui peut provoquer des contacts brusques et discontinus. Ainsi, bien que ce choix conduise à des difficultés supplémentaires par rapport à une approche explicite, nous montrons dans ce document qu'il peut être un choix judicieux pour répondre à toutes les exigences d'une simulation médicale interactive.

2.5 Les modèles déformables

Nous présentons maintenant quelques modèles de déformation qui seront exploités par la suite.

2.5.1 Modèles basiques

Le modèle masse-ressort La solution la plus simple pour créer un modèle déformable est de placer un ensemble de ressorts entre les particules du maillage Miller (1988); Montgomery et al. (2002). En associant une rigidité aux ressorts, on peut alors calculer des forces qui vont avoir tendance à ramener les particules vers leur position d'équilibre. Un vecteur est ainsi défini entre deux particules connectées, ce qui nous donne la direction des forces appliquées. Leur norme est souvent supposée directement proportionnelle au déplacement des particules (c'est-à-dire à la différence entre leur distance actuelle et leur distance dans la configuration initiale) :

$$\boldsymbol{f} = k * (\boldsymbol{l}_t - \boldsymbol{l}_0) \tag{2.43}$$

où $\boldsymbol{l}_0$ et $\boldsymbol{l}_t$ sont les positions respectivement initiales et actuelles, et k est un coefficient qui représente la rigidité du ressort. Évidemment les avantages des ces approches sont qu'elles sont très rapides à calculer, et elles sont bien adaptées pour bénéficier d'une parallélisation GPU, comme démontré par Sorensen et al. (2006). De plus, il est relativement facile de gérer les modifications topologiques pendant la simulation, pour simuler des découpes ou déchirures, puisqu'il suffit simplement de casser les ressorts découpés.

Le principal inconvénient c'est qu'ils ne permettent pas de modéliser adéquatement la déformation des tissus mous de façon réaliste. En effet, il est difficile de relier la raideur des ressorts, à un paramètre physique comme le module de Young. De plus, ils introduisent une anisotropie artificielle à travers le choix du maillage. La valeur de la rigidité doit alors être ajustée en fonction de la discrétisation, ce qui rend très difficile la recherche de valeurs cohérentes.

On trouve cependant un très grand nombre de travaux qui se reposent sur cette formulation, comme par exemple pour les simulations de vêtements Baraff et Witkin (1998). De récents travaux Delingette (2008), tendent à accroître la précision de ces modèles, ce qui

se fait bien évidemment au détriment de leur temps de calcul.

Le modèle de poutre On trouve également d'autres modèles de déformation, optimisés pour des objets présentant une structure particulière, comme ceux qui possèdent une structure linéique. On utilise ce type de modèle pour simuler le comportement d'un cathéter dans le réseau artériel par exemple, ou pour la simulation de suture Duriez et al. (2009).

	0	1	2	3	4
0	a	b			
1	b	c	d		
2		d	e	f	
3			f	g	h
4				i	j

FIG. 2.11 – Assemblage de la matrice du système décrit par un objet déformable composé de poutres connectées en série.

Les objets présentant une structure filaire sont modélisés par un ensemble de poutres connectées en série, chacune composées de deux points de contrôles. Les extrémités des segments sont considérées comme des rigides et possèdent alors un ensemble de 6 degrés de liberté (3 en translation et 3 en rotation) Cotin et al. (2005). Il est alors possible de manipuler le câble dans toutes les directions. Les éléments sont reliés en utilisant la méthode des éléments finis Dequidt et al. (2008), et on peut alors extraire une matrice de rigidité.

Les modèles déformables composés de poutres produisent un système numérique particulier. Si on organise les points dans l'ordre croissant (voir figure 2.11) le long de la poutre, la matrice générée par le système s'exprime dans une structure Block Tri-Diagonale (BTD). En effet, le déplacement d'un point de l'objet influence uniquement ses deux points voisins, on constate alors la présence de trois blocs de données non nuls sur chaque bloc de ligne de la matrice. Bien que ses valeurs changent à chaque pas de temps, la résolution de ce système est très rapide en utilisant l'algorithme de Thomas Kumar et al. (1993), qui permet de calculer l'inverse du système avec une complexité en linéaire en nombre de nœuds.

2.5.2 Modèles éléments finis volumiques

Modélisation linéaire en petit déplacements Le modèle linéaire est un modèle particulièrement simple basé sur les éléments finis, dans lequel on suppose que la matrice de rigidité est constante tout au long de la simulation. Il est généralement construit en utilisant une loi de Hooke avec le tenseur de contraintes de Green-Lagrange linéarisé

(hypothèse des petits déplacements). En exprimant un point comme des coefficients barycentriques des points de contrôle d'un élément (ce qui conduit à l'équation (2.10)), on peut alors exprimer[8] la relation entre contrainte et déplacement, en fonction des points de contrôle :

$$\begin{aligned} \begin{pmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{xy} \end{pmatrix} &= \begin{pmatrix} \frac{du_x}{d_x} \\ \frac{du_y}{d_y} \\ \frac{du_x}{d_y} + \frac{du_y}{d_x} \end{pmatrix} \begin{pmatrix} u_{1x} \\ u_{1y} \\ u_{2x} \\ u_{2y} \\ u_{3x} \\ u_{3y} \end{pmatrix} \\ &= \begin{pmatrix} b_1 & 0 & b_2 & 0 & b_3 & 0 \\ 0 & c_1 & 0 & c_2 & 0 & c_3 \\ c_1 & b_1 & c_2 & b_2 & c_3 & b_3 \end{pmatrix} \begin{pmatrix} u_{1x} \\ u_{1y} \\ u_{2x} \\ u_{2y} \\ u_{3x} \\ u_{3y} \end{pmatrix} \end{aligned} \tag{2.44}$$

Ce qu'on peut ré-écrire :

$$\boldsymbol{\varepsilon}_e = \mathbf{F}_e\, \boldsymbol{u}_e \tag{2.45}$$

À ce stade on va utiliser le principe des travaux virtuels qu'on peut montrer équivalent à l'équation (2.26) :

$$\boldsymbol{u}^T \cdot \boldsymbol{f} = \int_V \boldsymbol{\varepsilon}^T \overline{\otimes}\, \boldsymbol{\sigma}\, dV \tag{2.46}$$

Ainsi, en injectant les équations (2.45) et (2.19), on peut exprimer alors directement la matrice de rigidité locale à un élément par :

$$\begin{aligned} {\boldsymbol{u}_e}^T \cdot \boldsymbol{f}_e &= \int_{V_e} (\mathbf{F}_e \cdot \boldsymbol{u}_e)^T \cdot (\mathbf{E}_e\, \mathbf{F}_e \cdot \boldsymbol{u}_e)\, dV \\ {\boldsymbol{u}_e}^T \cdot \boldsymbol{f}_e &= \int_{V_e} {\boldsymbol{u}_e}^T \cdot {\mathbf{F}_e}^T\, \mathbf{E}_e\, \mathbf{F}_e \cdot \boldsymbol{u}_e\, dV \\ {\boldsymbol{u}_e}^T \cdot \boldsymbol{f}_e &= {\boldsymbol{u}_e}^T \cdot {\mathbf{F}_e}^T\, \mathbf{E}_e\, \mathbf{F}_e \cdot \boldsymbol{u}_e \int_v dV \\ \boldsymbol{f}_e &= {\mathbf{F}_e}^T\, \mathbf{E}_e\, \mathbf{F}_e \cdot \boldsymbol{u}_e * \text{Aire du triangle} \end{aligned} \tag{2.47}$$

On obtient alors directement la matrice de rigidité par :

$$\boldsymbol{f}_e = \mathbf{K}_e * \boldsymbol{u}_e \tag{2.48}$$

La matrice de rigidité ne dépend alors plus du tout du déplacement, et sera constante pendant toute la simulation. Le fait de posséder une matrice constante, va permettre de la pré-inverser et de l'utiliser pendant toute la simulation (ces aspects sont traités en détails dans le chapitre suivant). Cependant, le principal problème est que si les objets subissent de grandes déformations par rapport à leur position d'origine, on constate des

[8] Notons que nous utilisons ici la notation de Voight pour écrire le tenseur de déformation

déformations non réalistes. En effet, les contraintes sont dans ce cas sous-évaluées, par rapport à la déformation, ce qui entraîne généralement une trop grande déformation, et provoque le phénomène bien connu de gonflement des objets. Ainsi, malgré leur rapidité, ces modèles trop simples ne suffisent pas à produire des simulations médicales, où la précision joue un rôle très important. On peut malgré tout trouver dans la littérature de nombreux travaux basés sur l'élasticité linéaire Bro-Nielsen et Cotin (1996) ; Cotin et al. (1999) ou James et Pai (1999).

Le modèle co-rotationnel Les linéarisations du modèle précédent, comportaient deux aspects : l'approximation linéaire du matériau faite par la loi de Hooke, et l'approximation linéaire géométrique par le tenseur de Green Lagrange linéarisé. Or, on constate que dans la plupart des simulations médicales, la linéarisation de la partie géométrique de la déformation provoque des erreurs très importantes (bien plus que la linéarisation de la loi de comportement du matériau). Cela est en grande partie dû à une rotation non-négligeable des éléments. Si on comparait les matrices, on pourrait s'apercevoir qu'une simple rotation des éléments va avoir un impact important sur la matrice de rigidité.

Müller et al. (2002) proposent une façon élégante de remédier à ce problème, qu'ils appellent *stiffness warping*. Elle consiste à évaluer la rotation globale de l'objet déformé, puis à appliquer la rotation inverse autour de la matrice de rigidité (qui reste constante pendant toute la simulation). Ceci revient à calculer la déformation des éléments en les plaçant auparavant dans un repère qui annule la plus grande partie des rotations subies par l'objet. Pour évaluer les rotations, on peut utiliser la méthode de *shape matching* similaire à celle introduite dans Müller et al. (2005). La matrice de rotation globale est alors appliquée autour de la matrice de rigidité. Cette opération est rapide et permet d'obtenir des déformations plus réalistes.

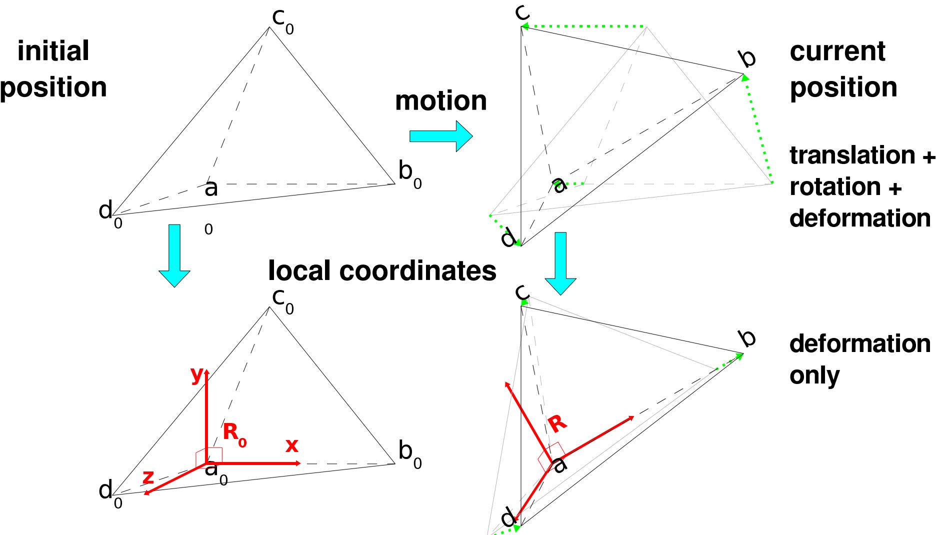


FIG. 2.12 – Co-rotational FEM : a local frame is computed on each element to handle large rotations.

La matrice de rotation globale ne tient pas compte des rotations locales, qui apparaissent indépendamment sur les différentes parties des objets quand ils se déforment. Ainsi, le

modèle co-rotationnel est basé sur une évaluation indépendante des rotations pour chaque élément. Les rotations locales de chaque élément sont alors beaucoup plus précises, et permettent d'accroître la gamme de déformation supportée. Dans chaque tétraèdre, le repère local est représenté par une matrice de rotation rigide $\mathbf{R}$, et sa transposée est utilisée pour replacer le tétraèdre dans sa configuration de référence. Le calcul de ces rotations n'est pas unique, et plusieurs méthodes ont été proposées Hauth et Straßer (2004) ;Müller et Gross (2004) ;Nesme et al. (2005b). Les déplacements et les forces sont calculés dans le système de coordonnée tourné, puis transformé à nouveau dans le système de coordonnée de l'objet (voir figure 2.12). En utilisant cette formulation, les forces de $\boldsymbol{f}_e$ appliquées aux sommets des éléments sont :

$$\boldsymbol{f}_e = \mathbf{R}_e \mathbf{F}_e{}^T \mathbf{E}_e \mathbf{F}_e \left(\mathbf{R}_e{}^T \boldsymbol{p}_e - \overline{\boldsymbol{p}}_e \right) \tag{2.49}$$

où les vecteurs $\boldsymbol{p}_e$ et $\bar{\boldsymbol{p}}_e$ de taille 12×1 sont les positions des sommets dans l'état déformé et non déformé. $\mathbf{R}_e$ est la matrice diagonale par blocs utilisés pour appliquer une rotation $\mathbf{R}$ pour les quatre vecteurs 3D. Il est ici important de remarquer que même si on peut évaluer facilement la rigidité, car $\mathbf{F}_e{}^T \mathbf{E}_e \mathbf{F}_e$ reste constant, les rotations ne le sont pas, et introduisent des modifications des matrices élémentaires. Ainsi, on ne peut plus supposer que la matrice assemblée du système $\mathbf{A}$ reste constante, et il sera donc nécessaire de résoudre un nouveau système linéaire à chaque pas de temps de la simulation. Le modèle co-rotationnel nous semble malgré tout un bon compromis pour les simulations médicales interactives, principalement pour sa rapidité, mais également parce qu'il est possible de corriger avec une opération très simple une grande partie des non-linéarités introduites pendant la déformation. Cependant, un grand nombre de contributions de cette thèse ne se limitent pas à ce modèle.

Modèles hyperélastiques Les modèles linéaires permettent d'aller encore plus loin dans la modélisation du comportement, et parviennent à simuler une déformation correcte même sous l'effet de fortes déformations. Le modèle non-linéaire le plus simple est le modèle de St venant kirchhoff. L'énergie de déformation est alors définie par :

$$W(\boldsymbol{\varepsilon_e}) = \frac{\lambda}{2}[\mathrm{tr}(\boldsymbol{\varepsilon_e})]^2 + \mu \mathrm{tr}(\boldsymbol{\varepsilon_e}^2) \tag{2.50}$$

Sa loi de comportement est une relation non linéaire, mais lorsqu'on calcule sa dérivée dans l'équation (2.20), on obtient au final une relation linéaire. Ce modèle servira de cas test, pour valider nos travaux sur des modèles plus complexes que le modèle co-rotationnel.

De nombreux autres modèles non-linéaires existants permettent d'approcher de mieux en mieux les relations déduites des expérimentations, mais c'est bien souvent au prix d'un temps de calcul plus long. Nous avons utilisé dans ce document le travail de Marchesseau et al. (2010) appelé Multiplicative Jacobian Energy Decomposition (MJED). Sa principale caractéristique est de proposer un assemblage rapide des matrices de rigidité pour une grande variété de matériaux isotropes et anisotropes. Ce modèle est une formulation générique de l'énergie à des matériaux hyperélastiques à l'intérieur de tétraèdres linéaires. Nous avions donc à disposition une implémentation des modèles ArrudaBoyce, St-Venant-Kirchhoff, NeoHookean, MooneyRivlin, VerondaWestman, Costa et Ogden qui sont des

modèles non-linéaires pouvant être simulés en temps réel. De notre point de vue, nous utiliserons ces modèles comme des boites noires, en sachant qu'à chaque pas de temps, ils nous fourniront une linéarisation de la matrice de rigidité du système. Ainsi, nous serons toujours capable d'assembler un système matriciel, mais évidemment celui-ci va changer de façon très importante au cours de la simulation.

2.5.3 Découpe et changements topologiques

Simuler des actes chirurgicaux tels que la découpe des organes, implique généralement plusieurs défis liés au remaillage du domaine, la mise à jour des informations mécaniques, et pour le calcul des contacts et le rendu haptique. Plusieurs approches ont été proposées pour remailler le domaine tout en maintenant une bonne qualité des maillages pendant la découpe Sifakis et al. (2007) Molino et al. (2007). Pour l'aspect déformation, les changements topologiques exigent essentiellement la mise à jour de la matrice de rigidité, et de la matrice de masse. Ceci est problématique si les objets sont simulés suivant un modèle linéaire, puisque dans ce cas la matrice du système doit être mise à jour et inversée à chaque modification topologique. Par souci de simplicité, nous considérons ici que les changements topologiques s'effectuent avec deux opérations très simples, qui sont la suppression et l'ajout d'éléments.

La façon la plus simple pour produire des changements topologiques est de supprimer des éléments, ce qui permet par exemple de simuler la brûlure des organes. L'avantage est que dans la plupart des cas le nombre de nœuds n'est pas modifié, et la dimension des matrices reste constante. En effet, comme les nœuds sont en général reliés à plusieurs éléments, le fait de supprimer un élément, consiste simplement à casser des connexions mécaniques. Dans les cas extrêmes, il est possible de séparer complètement un point de tous ses éléments, on peut alors choisir de le laisser "errer" dans l'espace, ou de le supprimer et changer la taille des matrices. La suppression de matière peut produire des effets indésirables, ainsi il est possible d'ajouter des éléments. Cette opération implique généralement l'ajout de nœuds, et il faut gérer avec beaucoup d'attention les dimensions des matrices. Pour obtenir une découpe plus fine, on pourra alors supprimer un élément découpé, le subdiviser, puis ajouter les éléments plus petits. La subdivision n'est pas une chose facile car elle nécessite de maintenir une bonne qualité de maillage Mor et Kanade (2000).

Du point de vue numérique, la suppression d'éléments revient à soustraire du système matriciel global, les matrices mécaniques des éléments supprimés . Inversement, l'ajout d'éléments revient à additionner les matrices de masse rigidité et amortissement au système. Il faut également calculer les informations mécaniques des éléments dans une configuration adaptée. Par exemple pour le modèle co-rotationnel il faut calculer la rigidité des nouveaux éléments dans la configuration initiale. Ainsi, dans la plupart des modèles non linéaires, la matrice globale du système n'est pas considérée comme constante, et elle est souvent reconstruite (ou évaluée) à partir de l'information des éléments. Dans ce cas, la découpe est automatiquement prise en compte du point de vue mécanique, et elle n'ajoute que de légers surcoûts.

2.6 Programmation moderne des GPU

Pour terminer ce chapitre, nous introduisons brièvement les concepts nécessaire à la parallélisation sur GPU. Cette section ne constitue pas un cours de parallélisme, et le lecteur intéressé pourra se référer aux quelques ouvrages parus depuis la création de ces langages Gaster et al. (2011); Sanders et Kandrot (2010).

2.6.1 L'architecture d'un GPU

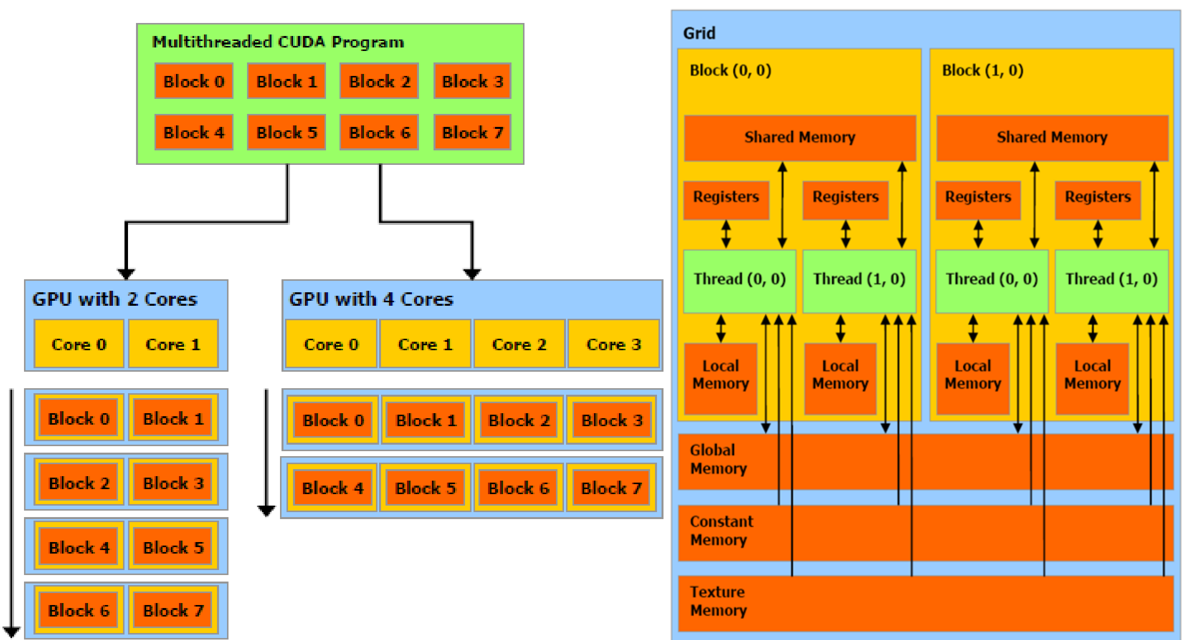
Pour comprendre la programmation des processeurs graphiques, regardons tout d'abord leur architecture. Le GPU est équipé de plusieurs processeurs (appelés *multiprocesseur*) qui fonctionnent indépendamment. Chaque multiprocesseur regroupe également plusieurs unités de calcul (8 sur les architectures NVIDIA) qui sont spécialisés et fonctionnent selon un schéma SIMD (Single Instruction Multiple Data). Au même moment, chaque multiprocesseur peut alors exécuter un ensemble de threads qui se trouvent au même endroit dans le programme (sur des données différentes). Cet ensemble de threads est appelé un *warp* (un groupe de 32 ou 64 threads sur l'architecture NVIDIA). Les warps sont affectés par tranche de 32 threads au lancement du programme, et restent fixes pendant toute l'exécution.

Malgré tout, chaque thread possède son propre environnement d'exécution (registres, compteur d'instructions). Chacun peut donc exécuter des parties différentes du programme de façon indépendante. En revanche, si plusieurs threads d'un même warp doivent exécuter un code différent (par exemple suite à un branchement), toutes les instructions divergentes sont sérialisées. Dans ce cas, le temps d'exécution sera égal à la somme des temps d'exécution de chaque partie divergente du programme.

Une autre différence fondamentale dans la programmation des GPU, concerne la gestion de la mémoire. En effet, sur un CPU traditionnel, il n'existe qu'une seule plage mémoire qui peut accueillir les données. En revanche, le GPU possède plusieurs types de mémoire, dont chacune présente des particularités et des temps d'accès spécifiques. Il existe 3 types de mémoire qui sont visibles par le CPU et par tous les multiprocesseurs (voir figure 2.13b). Ce sont la mémoire globale, la mémoire constante, et la mémoire de texture.

La mémoire constante et la mémoire de texture sont les plus rapides, mais elles ne peuvent pas être modifiées pendant l'exécution du programme (en pratique nous les utilisons pour des tâches spécifiques). La mémoire globale est plus générale mais elle présente une forte latence pour accéder aux données, en particulier si les lectures/écritures ne sont pas alignées. En effet, cette mémoire est optimisée pour fournir les données en un temps minimum, si plusieurs threads d'un même warp y accèdent de façon contiguës. Dans ce cas, le temps de transfert sera le même que le temps d'accès à une seule valeur. En revanche, si les threads d'un même warp y accèdent de façon non alignée, le temps d'accès sera multiplié par le nombre de plages non contiguës. Cette contrainte sera très importante pour maximiser les performances dans nos applications.

Enfin, il existe également deux types de mémoires locales à chaque multiprocesseur. Tout d'abord, chaque thread possède un ensemble de registres pour stocker ses propres données pendant l'exécution. De plus, chaque multiprocesseur possède une mémoire partagée qui



(a) Répartition de la charge de calcul

(b) Hierarchie de mémoire sur une Geforce

FIG. 2.13 – Programmation avec CUDA[9].

est très rapide. Cette mémoire est généralement utilisée pour communiquer (à faible coût) des données entre les threads exécutés sur le même multiprocesseur. En revanche, cette mémoire possède une capacité de stockage très limitée.

2.6.2 Programmer en CUDA

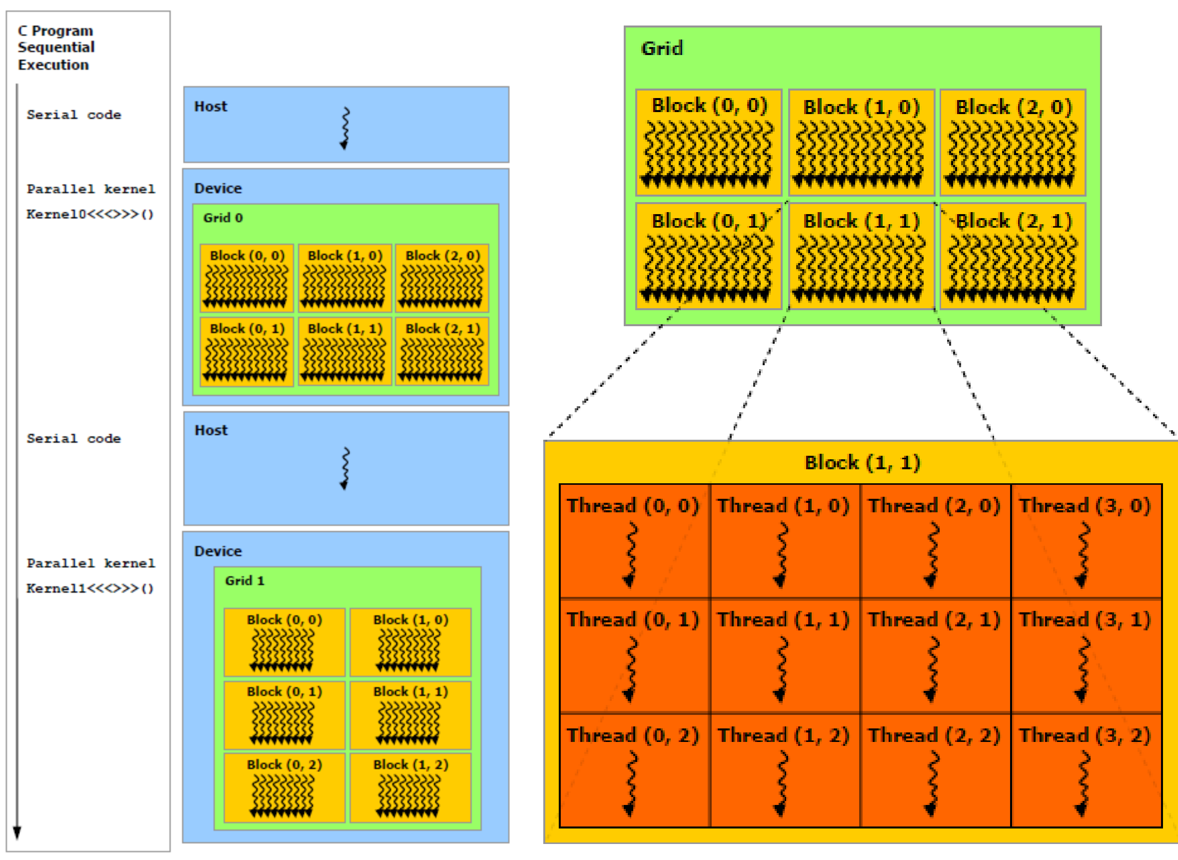
La programmation en CUDA suit un schéma maître/esclave. Le CPU exécute l'application de façon traditionnelle, puis il fait appel à des fonctions GPU appelés *kernels*, pour réaliser une tâche hautement parallèle (voir figure 2.14a). Du coté du CPU, l'exécution d'un kernel s'effectue donc généralement en trois temps :

1. La copie de données dans la mémoire du GPU.
2. L'appel au kernel.
3. L'attente du résultat[10].

Pour chacun de ces aspects, CUDA fournit un ensemble de primitives qui permet de communiquer avec le GPU. Ainsi, le processeur graphique est perçu comme un co-processeur spécialisé qui peut traiter très rapidement une tâche parallèle.

Du côté du GPU, la programmation du kernel démarre en spécifiant la répartition des tâches sur le processeur graphique. Pour cela, on utilise deux niveaux hiérarchiques, le premier correspond au nombre de **blocs** qui vont être exécutés sur le GPU, alors que le second est le nombre de **threads** contenus dans chaque bloc. Le nombre de blocs est décrit par une grille à deux dimensions, alors que le nombre de threads dans un tableau ayant au

[9]Source : NVIDIA CUDA Programming Guide
[10]Par défaut l'appel à un kernel n'est pas bloquant, jusqu'à ce que le CPU demande à lire les données résultantes



(a) Lancement de kernels par le CPU

(b) Répartition des threads sur GPU.

Fig. 2.14 – Exécution d'un kernel CUDA[9].

maximum trois dimensions (voir figure 2.13a). Chaque thread possède donc un identifiant de $(bloc, thread)$ unique, qu'on peut utiliser pour exécuter des parties spécifiques du code.

Concrètement, un bloc CUDA sera exécuté sans interruption par un seul multiprocesseur. Si le nombre de blocs est supérieur au nombre de multiprocesseurs, alors le GPU va répartir automatiquement la charge de travail sur chacun d'entre eux (voir figure 2.13a). De plus, chaque bloc contient plusieurs threads (une limitation de 512 est imposée sur les architectures NVIDIA), qui sont répartis dans différents warps. Les threads d'un même bloc peuvent communiquer entre eux via la mémoire partagée, et être synchronisés à l'aide de la primitive CUDA **__syncthread** (appelé *synchronisation locale*). Cependant, il n'est pas possible de synchroniser des threads qui appartiennent à des blocs différents.

En pratique, le code du kernel est une fonction écrite en C standard similaire au code sur CPU. Sur la figure 2.15, le kernel correspond uniquement aux lignes de 1 à 9 (sur la figure de droite). Les lignes 5 et 6 permettent à chaque thread de récupérer un identifiant $(bloc, thread)$, et la ligne 7 leur permet d'en déduire un identifiant unique. On écrit ensuite dans le tableau résultat C de la même manière que dans un code CPU. La principale différence, est que les boucles **for** sont généralement remplacées par un décalage dans un tableau. Pour cela, on utilise l'identifiant des threads pour connaître l'adresse des données à lire ou écrire. Comme plusieurs threads exécutent les instructions du kernel en parallèle,

cela revient à parcourir toutes les itérations de la boucle en parallèle.

```
1  void sumVec(int dim,
2          const float * A,
3          const float * B,
4          float * C) {
5
6     for (int id = 0; id < dim; id++) {
7        C[id] = A[id] + B[id];
8     }
9  }
10
11 int main (...) {
12    init (A); init (B);
13
14
15
16    sumVec(dim,A,B,C);
17 }
```

```
1  __global__ void sumVec(int dim,
2                 const float * A,
3                 const float * B,
4                 float * C) {
5     int tx = threadIdx.x;
6     int bx = blockIdx.x;
7     int id = bx*BSIZE + tx;
8     C[id] = A[id] + B[id];
9  }
10
11 int main (...) {
12    init (A); init (B);
13    dim3 threads(BSIZE,1);
14    dim3 grid((dim+BSIZE-1)/BSIZE,1);
15
16    sumVec<<< grid, threads >>>(dim,A,B,C);
17 }
```

FIG. 2.15 – Calcul CPU (gauche) et GPU (droite) de la somme de deux vecteurs

Les lignes de 11 à 17 sont exécutées par le CPU et permettent d'initialiser les données et de lancer le kernel. En particulier, les lignes 13 et 14 décrivent les dimensions des grilles de blocs et de threads qui vont exécuter le kernel, alors que la ligne 16 permet de lancer la fonction GPU. Si *dim* est la taille du vecteur, alors $(dim + BSIZE - 1)/BSIZE$ correspond au nombre de blocs qui doit exécuter le kernel ($BSIZE$ est une constante de l'application généralement fixée à 32 ou 64, qui permet de grouper les threads par blocs). En effet, si chaque bloc est composé de $BSIZE$ threads, alors ce calcul permet d'affecter au moins un thread par donnée. En pratique on va alors lancer un nombre plus important de threads que la dimension du vecteur. Dans ce cas, il faut exécuter la ligne 8 uniquement si l'indice du thread est inférieur à *dim*. Cependant, dans nos codes CUDA, nous avons toujours pris soin de redimensionner les vecteurs GPU avec une taille multiple de $BSIZE$. Ainsi, le fait de ne pas ajouter le test sur l'indice à la ligne 8 permet de maximiser les performances, en assurant que tous les threads d'un même warp seront toujours au même endroit dans le programme.

Limitation du langage

La plus grosse contrainte en CUDA est qu'il n'est pas possible de synchroniser des threads appartenant à des blocs différents. En effet, la seule façon de s'assurer qu'un calcul est terminé dans un autre bloc, est de rendre la main au CPU puis d'attendre la fin de l'exécution du kernel (et donc de tous les blocs). On pourra ensuite lancer un nouveau kernel dans lequel on aura la garantie que la tâche précédente est terminée (ce procédé sera nommé *synchronisation globale*). Évidemment le coût de lancement des kernels ainsi que l'attente des derniers blocs, rend cette synchronisation très coûteuse. Il faudra donc l'éviter au maximum.

Une autre difficulté réside dans les optimisations liées aux accès mémoire (en réalisant

des lectures alignées), qui sont à la charge du programmeur. Ainsi, il est nécessaire de bien penser à la représentation des données et au découpage des calculs. Une optimisation très courante consiste à charger les données dans la mémoire partagée, en effectuant des lectures alignées. Suite à une synchronisation locale, on pourra ensuite utiliser les données dans un ordre quelconque.

2.7 Conclusion

Nous avons présenté les fondements théoriques sur lesquels sont basés l'ensemble des travaux de cette thèse. Ces théories découlent de la mécanique des milieux continus, ce qui nous permet de mettre en équation le comportement de solides déformables. Nous avons également présenté la méthode des éléments finis, qui permet d'obtenir une approximation numérique des équations générées. Puis, nous avons présenté les solutions pour faire évoluer les systèmes mécaniques dans le temps. Nous avons terminé en présentant un ensemble de modèles de déformations.

L'ensemble des méthodes présentées ici nous permet d'animer virtuellement des objets, de telle sorte que leur comportement corresponde à celui obtenu lors des expérimentations. On peut alors s'attendre à ce que les objets virtuels reproduisent fidèlement les déformations réelles (sous réserve que les différentes approximations restent valides). Or, la simulation médicale, la précision des déformations joue un rôle essentiel, et nos choix seront orientés pour obtenir une grande précision des calculs. Cependant, pour garantir le côté interactif, il ne faut pas augmenter les temps de calcul, et pour cela nous exploiterons les architectures hautement parallèles.

Le choix le plus important a été de se baser sur un modèle d'intégration implicite. En effet, nous avons vu que cela nous permet de garantir une grande stabilité dans la simulation, mais nous amène également à résoudre un système d'équations linéaires à chaque pas de temps. Pour construire ce système, on a cependant fait la supposition que les relations entre les forces s'exerçant au sein de la matière, et les déplacements, peuvent être linéarisées sur de petites périodes de temps. Cette supposition est nécessaire, pour pouvoir utiliser nos travaux, mais elle conduit à une erreur très faible, puisque on suppose qu'un nouveau système linéaire est ré-évalué au début de chaque pas de temps, en fonction des positions courantes. Il faut donc pouvoir résoudre le système linéaire assez rapidement pour rester compatible avec le temps réel. D'autre part, nous avons vu que la précision de la méthode des éléments finis dépend du nombre d'éléments qui composent les objets virtuels. Or, dans l'objectif d'augmenter la précision des simulations, nous souhaitons pouvoir simuler un grand nombre d'éléments, le système linéaire étant d'autant plus problématique. Ainsi, le chapitre suivant est intégralement consacré à ces problèmes.

Chapitre

3

Résolution de systèmes linéaires

Table des matières

3.1 Introduction **53**
3.2 Résolution de systèmes d'équations linéaires **53**
3.2.1 Les méthodes de résolutions directes 53
Calcul explicite de l'inverse 54
Factorisation du système 54
Librairies et formats de stockage 56
3.2.2 Les méthodes itératives 58
Méthodes itératives stationnaires 58
La Méthode du gradient conjugué 59
Préconditionnement 61
Les méthodes multigrilles 63
3.2.3 Parallélisation de graphes 64
3.3 Intégrateur implicite optimisé sur GPU **65**
3.3.1 Parallélisation de l'intégrateur implicite sur GPU 65
3.3.2 Stockage mémoire pour maximiser les accès coalescent 67
3.3.3 Intégration implicite sans assemblage de matrice 68
Parallélisation du calcul FEM sur GPU 68
3.3.4 Optimisations pour le modèle co-rotationnel 70
3.3.5 Réduction des surcoûts en fusionnant les kernels 73
3.3.6 Résultats et Évaluations 73
3.4 Techniques de pré-conditionnement asynchrone **75**
3.4.1 Préconditionneur ré-utilisable 76
3.4.2 Préconditionneur asynchrone 78
3.4.3 Utilisation des rotations locales 79
3.4.4 Résultats et discussions 81
Influence des mises à jour du préconditionneur 81
Application à différents modèles de déformation 83
Évaluation des différents préconditionneurs 85
3.5 Conclusion **86**

3.1 Introduction

Nous avons vu dans le chapitre précédent que la méthode des éléments finis, couplée à un intégrateur implicite nous conduit à résoudre au cours de la simulation, un ou plusieurs systèmes d'équations linéaires définis par l'équation (2.42) où $\boldsymbol{x}$ est l'inconnue du problème. Cette équation peut être ré-écrite sous une forme générale par :

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b} \tag{3.1}$$

La résolution de ce type de systèmes a été très largement étudiée par différentes communautés (mathématique, physique, informatique, simulation, ...). Un grand nombre de méthodes ont été proposées pour exploiter les propriétés spécifiques à chaque matrice (taille, remplissage, format, ...) en vue de calculer la solution du problème le plus efficacement possible. Toutefois, il faut rappeler que nous sommes dans un contexte de simulation temps réel, ce qui implique que les algorithmes que nous sommes susceptibles d'utiliser doivent impérativement fournir une solution très rapidement.

Nous commencerons ce chapitre en présentant les différentes techniques de résolution de système linéaire, en orientant notre recherche vers les méthodes applicables à la simulation en temps réel. Ensuite, nous présenterons nos premières contributions permettant de paralléliser l'intégrateur implicite sur un processeur hautement parallèle tel qu'un GPU. Pour cela nous détaillerons en particulier les étapes nécessaires à la parallélisation de l'algorithme du gradient conjugué, et nous proposerons des optimisations pour le modèle co-rotationnel. Nous montrerons que cette méthode nous permet de simuler un nombre considérable d'éléments en temps réel. Ensuite, nous chercherons à simuler des objets hétérogènes, ou présentant un faible rapport masse-raideur. Ceci nous conduira à proposer une technique de préconditionnement exploitant la cohérence temporelle de la mécanique des objets.

3.2 Résolution de systèmes d'équations linéaires

Il n'existe pas de méthode universelle pour résoudre un système linéaire, le choix est guidé par la nature de la matrice, la mémoire et le temps dont on dispose. Nous commençons ce chapitre par un bref état de l'art des méthodes de résolution de système linéaire. On peut trouver dans la littérature deux grandes familles d'algorithmes pour résoudre ces problèmes. La première consiste à inverser le système (résolution directe), la seconde consiste à raffiner une solution initiale par une succession d'itérations (résolution itérative).

3.2.1 Les méthodes de résolutions directes

Les méthodes directes consistent à calculer de façon exacte l'inverse du système (ou une factorisation), permettant d'obtenir la solution du problème en un nombre fini d'opérations (voir Davis (2006) pour de plus amples informations). Ces méthodes sont considérées comme robustes puisqu'elles sont peu sensibles à la difficulté numérique des problèmes traités (imprécisions, mauvais conditionnement...). Nous détaillerons cette notion au cours de ce chapitre.

Calcul explicite de l'inverse

Une première technique consiste à calculer explicitement $\mathbf{A}^{-1}$ de telle sorte que la solution de l'équation (3.1) soit obtenue directement par $\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{b}$. La multiplication de $\mathbf{A}^{-1}$ par $\boldsymbol{b}$ est une opération facilement parallélisable, qui peut être implémentée efficacement sur un GPU. L'inversion explicite de la matrice est parfaitement bien adaptée si $\mathbf{A}$ ne change pas pendant la simulation. Il est également envisageable d'effecteur ce calcul en temps réel pour des matrices de petites tailles, ou si la matrice est facile à inverser. C'est le cas par exemple des matrices diagonales ou bloc-diagonales, dont l'inverse est trivial à obtenir. En revanche, dans le cas général, le calcul de $\mathbf{A}^{-1}$ est une opération complexe et très coûteuse. L'inversion explicite de $\mathbf{A}$ est limitée également par le coût de stockage de l'inverse $\mathbf{A}^{-1}$. En effet, bien que $\mathbf{A}$ soit une matrice creuse, son inverse est une matrice dense et son stockage devient très vite prohibitif.

Pour rendre ces méthodes applicables à la simulation interactive, une solution est de réduire le nombre de degrés de liberté. En statique ou quasi-statique, on peut utiliser la méthode de condensation Bro-Nielsen et Cotin (1996), qui permet de calculer la mécanique des objets uniquement à la surface du maillage. La sous matrice qui ne comprend que les nœuds de la surface, est inversée explicitement en dehors de la simulation (ce qui limite la méthode aux cas linéaires). Par la suite, on souhaite ne pas se limiter à des matrices constantes, ou de petites tailles, nous allons pour cela introduire les méthodes de factorisation.

Factorisation du système

Pour tirer parti du fait que les matrices mécaniques sont très creuses, une autre possibilité consiste à calculer une factorisation du système. Ceci permet d'exprimer $\mathbf{A}$ sous la forme d'un produit de matrice plus simple (souvent triangulaires ou diagonales) qui garde un faible taux de remplissage. Par exemple, toute matrice inversible peut être exprimée sous la forme d'une décomposition de type :

$$\mathbf{A} = \mathbf{LU} \tag{3.2}$$

où $\mathbf{L}$ est une matrice triangulaire inférieure et $\mathbf{U}$ une matrice triangulaire supérieure. Cette décomposition est unique, et la solution de l'équation (3.1) peut être obtenue par :

$$\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{b} \Leftrightarrow \begin{cases} \mathbf{L}\boldsymbol{y} = \boldsymbol{b} & \Leftrightarrow \quad \boldsymbol{y} = \mathbf{L}^{-1}\boldsymbol{b} \\ \mathbf{U}\boldsymbol{x} = \boldsymbol{y} & \Leftrightarrow \quad \boldsymbol{x} = \mathbf{U}^{-1}\mathbf{L}^{-1}\boldsymbol{b} \end{cases} \tag{3.3}$$

Ces opérations nécessitent de résoudre deux systèmes linéaires qui sont composés uniquement de matrices triangulaires. La résolution du système contenant la matrice $\mathbf{L}$ est appelée *descente* et peut être calculée par substitutions successives de la première à la dernière ligne. Alors que la résolution qui concerne la matrice $\mathbf{U}$ est appelée *remontée* et peut être calculée par substitutions successives de la dernière à la première ligne. Les substitutions réalisés pour la descente ou la remontée correspondent en réalité à l'algorithme du *pivot de Gauss*.

Les opérations de descente et de remontée sont souvent très rapides, mais l'algorithme de factorisation reste un point critique. Il consiste à calculer successivement chaque ligne

```
1 for k = 1 to n do
2   for i = k + 1 to n − 1 do
3     A_{i,k} = A_{i,k} / A_{k,k}
4     for j = k + 1 to n do
5       A_{i,j} = A_{i,j} − A_{i,k} A_{k,j}
```

Algorithme 2 : Algorithme de factorisation en place, de la matrice sous forme **LU**.

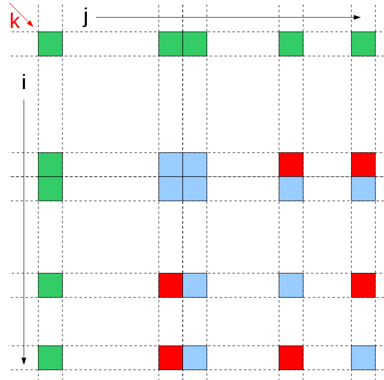


FIG. 3.1 – Phénomène de remplissage pendant une factorisation **LU**.

et chaque colonne des matrices **U** et **L** (voir l'algorithme 2). Pendant le déroulement de l'algorithme, on constate un phénomène de *remplissage* des triangulaires. En effet, dès lors que les coefficients $\mathbf{A}_{i,k}$ et $\mathbf{A}_{k,j}$ sont non nuls simultanément, l'algorithme génère de nouvelles valeurs dans la matrice **A**. Intuitivement, on comprend que ces nouvelles valeurs ont un effet cumulatif puisqu'elles se reportent sur les lignes suivantes. La factorisation obtenue est alors moins creuse que la matrice de départ (voir figure 3.1). Les éléments en verts sont les données initiales, les éléments en bleus sont uniquement modifiés, alors que les éléments en rouges sont ajoutés. À chaque itération ces derniers vont engendrer la création de nouvelles valeurs.

L'unicité de la décomposition **LU**, nous assure que si les matrices **U** et **L** sont denses, alors il n'est pas possible d'obtenir un résultat plus creux. Cependant, il est possible de minimiser le phénomène de remplissage en ré-organisant la matrice initiale (la factorisation ne s'applique alors mathématiquement plus sur **A**, mais sur une permutation de **A**). En effet, sur l'exemple suivant, la factorisation de la matrice de gauche produit des matrices triangulaires denses, puisque tous les $\mathbf{A}_{i,j}$ et $\mathbf{A}_{j,i}$ sont non nuls simultanément. À l'inverse, la matrice de droite ne génère aucun remplissage.

$$\begin{pmatrix} * & * & * & * & * \\ * & * & 0 & 0 & 0 \\ * & 0 & * & 0 & 0 \\ * & 0 & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{pmatrix} \qquad \begin{pmatrix} * & 0 & 0 & 0 & * \\ 0 & * & 0 & 0 & * \\ 0 & 0 & * & 0 & * \\ 0 & 0 & 0 & * & * \\ * & * & * & * & * \end{pmatrix}$$

Le passage de l'une de ces matrices à l'autre est possible en la multipliant par une *matrice de permutation* $\mathcal{P}$. Ainsi, en factorisant $\mathcal{P}\mathbf{A}\mathcal{P}^{-1}$ plutôt que **A** il est possible de diminuer le taux de remplissage de **U** et **L**. La solution du système initial est alors obtenue par :

$$\left\{ \begin{array}{lcl} \mathcal{P}\mathbf{A}\mathcal{P}^{-1} & = & \mathbf{LU} \\ \mathbf{A}\boldsymbol{x} = \boldsymbol{b} & \Leftrightarrow & \boldsymbol{x} = \mathcal{P}^{-1}\mathbf{U}^{-1}\mathbf{L}^{-1}\mathcal{P}\boldsymbol{b} \end{array} \right. \tag{3.4}$$

La multiplication $\mathcal{P}^{-1}\boldsymbol{b}$ est très rapide car $\mathcal{P}$ ne contient qu'une seule valeur égale à 1 dans chaque ligne et chaque colonne. Elle consiste donc à simplement échanger l'ordre

des valeurs du vecteur $\boldsymbol{b}$. Cependant, il faut déterminer la façon dont $\mathcal{P}$ est obtenu pour minimiser le nombre de création pendant la factorisation.

Ce problème est dit *NP-Complet*, ce qui signifie que sa résolution exacte est trop coûteuse à calculer, et qu'il faut se contenter d'heuristiques qui fournissent une solution acceptable en un temps limité. Ces heuristiques reposent sur la théorie des graphes, et constituent un domaine de recherche à part entière Schloegel et al. (2000). Plusieurs heuristiques ont été proposées (frontale, arbre d'élimination, bissections, ...), qui permettent de privilégier tantôt le stockage de données, tantôt le temps d'exécution, ou encore le degré de parallélisme de la future factorisation. Nous détaillerons ce dernier point dans la section 3.2.3.

En utilisant cette ré-organisation, les matrices factorisées conservent un faible taux de remplissage, ce qui permet d'appliquer la méthode sur de grands systèmes d'équations. La factorisation en elle-même s'effectue en deux temps : la *factorisation symbolique*, qui consiste à déterminer au préalable le nombre et la position des nouveaux coefficients (afin de réserver l'empreinte mémoire pouvant accueillir les triangulaires). La seconde est dite *factorisation numérique*, et consiste à calculer les coefficients réels. Cette étape est la plus coûteuse en temps de calcul, mais elle peut être optimisée en effectuant les calculs par bloc Toledo et al. (2003). Les opérations élémentaires sont alors transformées en des opérations matricielles sur des petites matrices denses. Cela permet d'utiliser des routines bas-niveau optimisées comme BLAS (Basic Linear Algebra Subroutines, Lawson et al. (1979)) sur ces blocs pour augmenter les performances.

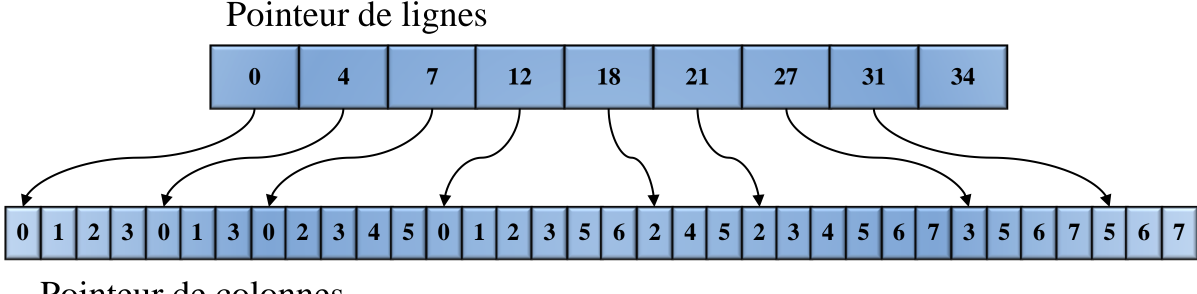
Par analogie, il est possible d'étendre ce procédé à d'autres types de factorisations. On peut citer notamment la factorisation de Cholesky (de forme $\mathbf{LL}^T$) qui s'applique uniquement aux matrices symétriques définies positives. Elle permet d'améliorer les performances, ainsi que le coût de stockage, puisqu'elle tient compte du fait que $\mathbf{U} = \mathbf{L}^T$. En contrepartie, la méthode est sensible aux instabilités numériques puisqu'elle repose sur le calcul de racines carrées (La matrice $\mathbf{L}$ est en quelque sorte la racine carrée de $\mathbf{A}$). La factorisation de Crout (de forme $\mathbf{LDL}^T$) permet de contourner ce problème en calculant séparément l'inverse de la diagonale dans la matrice $\mathbf{D}$, mais elle nécessite un coût légèrement plus élevé.

On peut trouver dans la littérature quelques travaux proposant des factorisations de matrices denses sur GPU Galoppo et al. (2005); Quintana-Ortí et al. (2009). Cependant, l'algèbre linéaire pour les matrices creuses reste difficile à traiter sur GPU, en raison des nombreuses dépendances dans les calculs. Les méthodes directes sont de bonnes candidates pour les simulations en temps réel, à condition d'avoir des matrices de taille raisonnable, car le coût de factorisation est de complexité polynomiale.

Librairies et formats de stockage

Les factorisations de matrices creuses sont constituées d'algorithmes complexes et surtout difficiles à optimiser. Il est alors intéressant d'utiliser une des nombreuses librairies qui facilitent leurs utilisations. Parmi ces librairies, on peut citer notamment CSPARSE Davis (2006), TAUCS Toledo et al. (2003), et PARDISO Schenk et Gärtner (2004), qui sont les librairies avec lesquelles nous avons réalisé nos expérimentations. Ces trois bibliothèques

proposent des factorisations **LU**, **LL**^{T} et **LDL**^{T} (avec approximativement les mêmes performances[1] pour des matrices de l'ordre de grandeur que nous avons utilisé). CSPARSE est la plus facile à utiliser dans la mesure où elle n'a aucune dépendance vers d'autres librairies. Les deux autres sont plus avancées et proposent également des factorisations par bloc ainsi qu'une parallélisation sur CPU. Ces bibliothèques peuvent également être couplées avec METIS, qui est spécialisée dans le découpage de graphes Schloegel et al. (2000). Elle peut être utilisée pour calculer la permutation des matrices qui minimisent le phénomène de remplissage des factorisations. Par ailleurs, les matrices produites par la simulation présentent en général une difficulté numérique constante, ce qui permet d'utiliser ces librairies comme de véritables boites noires.



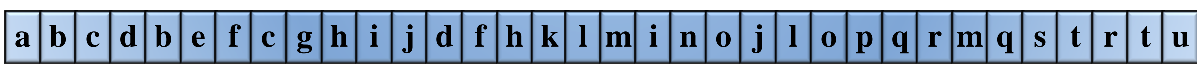


FIG. 3.2 – Exemple de stockage CRS, de la matrice donnée en figure 2.8.

Pour pouvoir utiliser facilement ces différentes librairies, il est nécessaire de stocker les matrices dans un format compatible. Elles reposent toutes sur le format de stockage CRS (*Compressed Row Sparse*) ou sa transposée le CCS (*Compressed Column Storage*), qui sont des formats standards. Ces formats permettent de stocker n'importe quelle matrice creuse, dans la mesure où ils ne font aucune supposition sur les propriétés de remplissage de la matrice. Il consiste à définir trois vecteurs (voir figure 3.2) :

1. **Pointeur de lignes** : Contient les indices du début de chaque nouvelle ligne dans les deux autres vecteurs. Sa taille est $n+1$ (où n est le nombre de lignes de la matrice).
2. **Pointeur de colonnes** : Contient les colonnes de chaque valeur non nulle, sa taille est égale au nombre de valeurs non nulles.
3. **Valeurs** : Contient les valeurs non nulles, sa taille est égale au nombre de valeurs non nulles.

Ainsi pour une matrice de taille n contentant k valeurs non nulles, ce format permet de stocker uniquement $2k+n+1$ valeurs, ce qui est en général plus petit que les n^2 valeurs théoriques. De manière plus importante, lorsqu'on veut accéder à l'ensemble des valeurs

[1]PARDISO et TAUCS sont plus efficaces pour des matrices de grandes tailles.

non nulles d'une même ligne, ce format de stockage offre la possibilité de lire des valeurs contiguës en mémoire. Cette propriété est très utile sur des architectures comme les GPU, pour effectuer des multiplications de matrices–vecteur par exemple Bell et Garland (2008).

3.2.2 Les méthodes itératives

L'utilisation de solveurs directs permet de construire des solveurs génériques et optimisés pour matrices creuses. Cependant, le coût de factorisation devient rapidement très élevé dès que la dimension de la matrice grandit. De plus, les méthodes directes nécessitent d'exécuter entièrement l'algorithme pour pouvoir poursuivre la simulation, ce qui peut être problématique si cette opération est trop longue à calculer. Pour palier cette contrainte, on peut utiliser les méthodes itératives qui constituent une autre famille de méthodes de résolution de système linéaire. Elles sont en général, plus faciles à mettre en œuvre, plus rapides, et présentent une consommation de mémoire moindre, par rapport aux méthodes directes Barrett et al. (1994).

Les méthodes itératives consistent à raffiner une solution initiale $\boldsymbol{x}^{(0)}$ par une succession d'itérations. Elles construisent une suite $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(n)}$ telle que $\|\boldsymbol{x} - \boldsymbol{x}^{(k)}\|$ tend vers 0, et donc se rapproche de la solution exacte. L'avantage par rapport aux méthodes directes, est qu'il est possible de stopper le calcul, dès lors que la solution nous paraît acceptable. En effet, il est tout à fait légitime d'arrêter les calculs, dès que l'approximation calculée engendre une erreur négligeable dans le déplacement des objets. On parle alors de convergence, quand l'algorithme parvient à fournir une solution dont l'erreur est plus petite qu'un certain seuil fixé par l'utilisateur.

En contrepartie, les algorithmes itératifs souffrent de généricité, et ne possèdent pas la robustesse des méthodes directes. Il est tout à fait possible qu'un algorithme itératif ne converge jamais, ou seulement après un nombre très grand d'itérations. Dans ce cas, on peut limiter le temps de calcul en bornant le nombre d'itérations. Le temps nécessaire à chaque itération étant approximativement constant, on peut ainsi garantir un rafraîchissement élevé des images. On dit alors que l'algorithme n'a pas convergé, et le comportement mécanique des objets simulés peut paraître incorrect[2]. Malgré tout, il est possible de poursuivre la simulation en espérant que les objets se stabiliseront rapidement pendant les pas de temps suivants.

D'autre part, la convergence de ces méthodes dépend de paramètres difficilement mesurables comme le spectre de la matrice. Les algorithmes les plus fiables assurent la convergence de la méthode, mais uniquement sous certaines conditions et après un nombre élevé d'itérations. Le coût total de calcul peut alors devenir supérieur aux méthodes directes et malgré tout fournir une solution de moins bonne qualité. Malgré ces limitations les algorithmes itératifs sont très largement utilisés dans les simulations en temps réel.

Méthodes itératives stationnaires

Les méthodes stationnaires consistent à modifier une à une chaque composante du vecteur solution jusqu'à obtenir une solution avec une précision satisfaisante. On peut citer la méthode de Jacobi qui consiste à mettre à jour toutes les composantes du vecteur solution

[2] En règle générale, ceci introduit artificiellement de l'amortissement, et les objets peuvent paraître plus mous.

de façon indépendante.

$$\boldsymbol{x}_i{}^{(k+1)} = \frac{1}{\mathbf{A}_{i,i}}(\boldsymbol{b}_i - \sum_{j=1}^{i-1} \mathbf{A}_{i,j}\boldsymbol{x}_j{}^{(k)} - \sum_{j=j+1}^{n} \mathbf{A}_{i,j}\boldsymbol{x}_j{}^{(k)}) \tag{3.5}$$

Chacune des inconnues est alors traitée comme si toutes les autres étaient fixées. Son implémentation est facilement parallélisable Østergaard Noe et al. (2008), mais la convergence n'est pas garantie, et dépend de la dominance de la diagonale.

La méthode de Gauss-Seidel est très similaire à la méthode de Jacobi, à la différence près qu'elle permet de prendre en compte les modifications des variables dès qu'elles sont calculées :

$$\boldsymbol{x}_i{}^{(k+1)} = \frac{1}{\mathbf{A}_{i,i}}(\boldsymbol{b}_i - \sum_{j=1}^{i-1} \mathbf{A}_{i,j}\boldsymbol{x}_j{}^{(k+1)} - \sum_{j=j+1}^{n} \mathbf{A}_{i,j}\boldsymbol{x}_j{}^{(k)}) \tag{3.6}$$

Chaque composante de la solution est alors évaluée séquentiellement, mais le calcul des inconnues suivantes utilisent le résultat de l'itération courante. Cet algorithme nécessite un nombre plus petit d'itérations pour converger, mais il est beaucoup plus difficile à paralléliser. Enfin, la méthode de surrelaxation successive (SOR) est une variante du Gauss-Seidel dans laquelle on pondère les itérations avec un paramètre ω afin d'accélérer la convergence de la méthode.

Les méthodes stationnaires sont peu utilisées pour résoudre les équations mécaniques dans les simulations en temps réel, car leur convergence n'est pas garantie et est difficile à prévoir.

La Méthode du gradient conjugué

La méthode du gradient conjugué (CG) a été inventée en 1952 simultanément par Lanczos (1952) et Hestenes et Stiefel (1952), et a été introduite dans le contexte de la simulation de vêtement par Baraff et Witkin (1998). Une condition nécessaire à son utilisation est que la matrice soit symétrique positive définie, ce qui est le cas des problèmes posés par la simulation.

Le CG repose sur la notion de vecteurs conjugués par rapport à $\mathbf{A}$, qui peut être décrit comme suit : Soit deux vecteurs $\boldsymbol{u}$ et $\boldsymbol{v}$, ils sont dits conjugués par rapport à $\mathbf{A}$ si $\boldsymbol{u}^T\mathbf{A}\boldsymbol{v} = 0$, ce qui signifie que $\boldsymbol{u}$ et $\boldsymbol{v}$ sont orthogonaux dans la base $\mathbf{A}$. L'algorithme consiste à utiliser successivement des projections dans des espaces particuliers pour exprimer la solution. Si l'on possède une suite de n vecteurs $\boldsymbol{p}_k$ orthogonaux, ces vecteurs forment une base dans laquelle la solution $\boldsymbol{x}$ peut être exprimée par :

$$\boldsymbol{x} = \sum_{i=1}^{n} \alpha_i \boldsymbol{p}_i \tag{3.7}$$

Le problème revient donc à chercher les coefficients α_i qui peuvent être obtenus par :

$$\begin{aligned}
\boldsymbol{b} &= \mathbf{A}\boldsymbol{x} = \sum_{i=1}^{n} \alpha_i \mathbf{A}\boldsymbol{p}_i \\
\boldsymbol{p}_k{}^T\boldsymbol{b} &= \sum_{i=1}^{n} \alpha_i \boldsymbol{p}_k{}^T \mathbf{A}\boldsymbol{p}_i \\
&= \alpha_k \boldsymbol{p}_k{}^T \mathbf{A}\boldsymbol{p}_k (\text{car } \forall i \neq j, \boldsymbol{p}_k \text{ et } \boldsymbol{p}_i \text{ sont conjugués}) \\
\alpha_k &= \frac{\boldsymbol{p}_k{}^T\boldsymbol{b}}{\boldsymbol{p}_k{}^T\mathbf{A}\boldsymbol{p}_k}
\end{aligned} \tag{3.8}$$

Il suffit ensuite d'effectuer le calcul pour chacune des composantes k, afin d'en déduire la solution exacte. Le CG a d'abord été présenté comme une méthode directe puisqu'il assure (sous réserve d'une précision supposée exacte des calculs) une convergence après, au plus n itérations (où n est la dimension du système).

D'un autre coté, la base décrite par les $\boldsymbol{p}_k$ n'est pas unique et en fonction de ce choix, il peut ne pas être nécessaire de calculer tous les α pour obtenir une bonne approximation. L'algorithme devient alors itératif au sens où il permet de calculer une approximation qui sera, au fil des itérations, de plus en plus proche de la solution exacte. Dans ce cas, le CG revient à minimiser la forme quadratique suivante :

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\mathbf{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} + \boldsymbol{c} \tag{3.9}$$

Trouver le minimum de cette fonction est équivalent à déterminer $\boldsymbol{x}$, pour lequel le gradient f' de la fonction f s'annule. Le gradient peut être vu comme la tangente de la courbe, et trouver le minimum de la fonction revient à chercher le point où la tangente est nulle. On constate que le minimum de l'équation (3.9) fournit la même solution que le système linéaire de l'équation (3.1) :

$$\begin{aligned}
f'(\boldsymbol{x}) = 0 \quad &\Leftrightarrow \quad \frac{1}{2}\mathbf{A}^T\boldsymbol{x} + \frac{1}{2}\mathbf{A}\boldsymbol{x} - \boldsymbol{b} = 0 \\
&\Leftrightarrow \quad \mathbf{A}\boldsymbol{x} - \boldsymbol{b} = 0, \qquad \text{si } \mathbf{A} \text{ est symétrique}
\end{aligned} \tag{3.10}$$

L'idée de base de la méthode du gradient conjugué, consiste donc à chercher une direction dans laquelle f diminue, puis de s'y déplacer. En répétant la procédure, on espère ainsi "descendre" le long de la courbe jusqu'à atteindre son minimum. Nous avons détaillé dans l'annexe B, la manière dont sont déterminées les directions et les coefficients associés, pour dérouler l'algorithme du CG. En pratique, on utilise une version légèrement modifiée (voir l'algorithme 3) qui ne nécessite le stockage que des deux derniers résidus et d'une seule multiplication matrice vecteur par itération. Nous proposons au lecteur de se reporter à l'article de Shewchuk (1994) pour plus d'informations.

Cet algorithme est très facile à mettre en œuvre puisqu'il ne repose que sur trois opérations de base qui sont des multiplications matrice—vecteur, vecteur—vecteur et la combinaison

1 $\boldsymbol{r} = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(0)}$
2 $\boldsymbol{p} = \boldsymbol{r}$
3 $\rho_{new} = \boldsymbol{r}^T\boldsymbol{r}$
4 $\rho_0 = \rho_{new}$
5 **for** $i = 1$ **to** n **do**
6   if $(\rho_{new} \leq \varepsilon^2 \rho_{new})$ **break**
7   $\boldsymbol{q} = \mathbf{A}\boldsymbol{p}$
8   $\alpha = \dfrac{\rho_{new}}{\boldsymbol{p}^T\boldsymbol{q}}$
9   $\boldsymbol{x}^{(i)} = \boldsymbol{x}^{(i-1)} + \alpha\boldsymbol{p}$
10   $\boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{q}$
11   $\rho_{old} = \rho_{new}$
12   $\rho_{new} = \boldsymbol{r}^T\boldsymbol{r}$
13   $\beta = \dfrac{\rho_{new}}{\rho_{old}}$
14   $\boldsymbol{p} = \boldsymbol{r} + \beta\boldsymbol{p}$

Algorithme 3 : Algorithme du gradient conjugué.

linéaire de vecteurs. Ces opérations étant parallélisables, le CG est potentiellement parallélisable également. Plusieurs travaux ont ainsi été publiés, Parker et O'Brien (2009); Hermann et al. (2009) proposent une parallélisation sur CPU, dont l'idée de base est de définir plusieurs régions indépendantes sur le maillage d'éléments finis. Ceci permet de les résoudre en parallèle, mais il faut gérer les frontières, de façon particulière. Plus récemment des travaux sur GPU ont également été menés Bolz et al. (2003); Buatois et al. (2009), qui reposent principalement sur l'utilisation du format CRS pour maximiser les lectures alignées sur le GPU.

Préconditionnement

Dans le cas où l'algorithme itératif ne parvient pas à converger en un nombre acceptable d'itérations, il est assez fréquent de le combiner avec un préconditionneur. En effet, lorsque le rapport masse/raideur est élevé, ou que les objets simulés sont constitués d'éléments non homogènes (en taille ou en rigidité), le système que l'on est amené à résoudre se trouve être mal conditionné. Le conditionnement de la matrice est un outil mathématique, qui donne une mesure de la difficulté numérique de la résolution du problème. Il est donné par la formule :

$$\kappa(\mathbf{A}) = \parallel \mathbf{A}^{-1} \parallel \cdot \parallel \mathbf{A} \parallel \tag{3.11}$$

Plus ce nombre est élevé, plus le système est difficile à résoudre. Dans le cas d'un solveur itératif, un mauvais conditionnement a pour conséquence d'augmenter de façon significative le nombre d'itérations nécessaire à la convergence de l'algorithme. Dans ce cas, l'efficacité des solveurs itératifs est plus contestable. Une solution pour continuer d'exploiter les bonnes propriétés des solveurs itératifs est d'utiliser un préconditionneur. Un préconditionneur $\mathbf{P}$ est une approximation de l'inverse de la matrice qui est idéalement plus facile à inverser. Le système posé par l'équation (3.1) se transforme alors en :

$$\begin{cases} \mathbf{P}^{-1}\mathbf{A}\boldsymbol{x} & = & \mathbf{P}^{-1}\boldsymbol{b} \\ \mathbf{P}^{-1}\mathbf{A} & \simeq & \mathbf{I} \end{cases} \tag{3.12}$$

Notons toutefois que cette équation décrit un préconditionnement gauche, mais il est

également possible d'appliquer un préconditionneur à droite, la résolution se fait alors en deux temps :

$$\begin{cases} \mathbf{A}\mathbf{P}^{-1}\boldsymbol{y} & = & \boldsymbol{b} \\ \boldsymbol{x} & = & \mathbf{P}^{-1}\boldsymbol{y} \end{cases} \tag{3.13}$$

Même si $\mathbf{A}$ et $\mathbf{P}$ sont deux matrices symétriques définies positives, le produit $\mathbf{PA}$ ne l'est pas forcément. Cependant, en factorisant le préconditionneur, puis en appliquant les membres de la factorisation à gauche puis à droite, on peut prouver que l'algorithme formé converge vers la même solution que le problème initial (voire Shewchuk (1994)). Cette version de l'algorithme est appelé *Transformed Preconditioned Conjugate Gradient Method*, mais elle est limitée par le calcul de la factorisation du préconditionneur. En substituant soigneusement l'ordre des calculs on peut aboutir sur l'algorithme *Untransformed Preconditioned Conjugate Gradient Method*, qui ne nécessite qu'une seule application de $\mathbf{P}^{-1}$ par itération.

Ainsi, l'application du préconditionneur laisse la solution du système initial inchangée quelle que soit la matrice $\mathbf{P}$ utilisée, pour peu qu'elle soit inversible. Et il est assez facile de montrer que le conditionnement $\kappa(\mathbf{P}^{-1}\mathbf{A})$ est plus faible que celui de $\mathbf{A}$. L'application du préconditionneur permet alors de diminuer le nombre d'itérations nécessaire à la convergence de l'algorithme itératif. D'une certaine façon, l'idée sous-jacente du préconditionnement est de combiner un solveur itératif, et un solveur direct appliqué à une approximation.

En revanche, cette méthode ajoute deux surcoûts dans la simulation, le premier pour calculer le préconditionneur lui même, et le second pour son application. En effet, pour chaque nouvelle matrice $\mathbf{A}$, il est nécessaire de recalculer explicitement l'inverse $\mathbf{P}^{-1}$. Or, le coût de cette opération croît en général avec la qualité de l'approximation. De plus, chaque application du préconditionneur nécessite la multiplication de $\mathbf{P}^{-1}$, par le vecteur solution (voire Shewchuk (1994) pour plus de détails). Cette dernière étape est en général relativement rapide, mais elle doit être exécutée à chaque itération de l'algorithme itératif. Le nombre d'itérations économisées avec le préconditionneur ne suffit pas à définir son efficacité, car il faut prendre en compte les surcoûts de la méthode. Dans le cas contraire, le meilleur préconditionneur serait l'inverse exacte du système $\mathbf{P}^{-1} = \mathbf{A}^{-1}$. En effet, avec ce préconditionneur, le CG converge en une seule itération puisque l'application de $\mathbf{P}^{-1}$ fournit directement la solution. Cependant, le calcul de ce dernier est aussi difficile que de résoudre le problème initial. Ainsi, le choix d'un préconditionneur exige de trouver le rapport optimal entre la qualité de l'approximation, et le coût de son utilisation.

Il existe une infinité de préconditionneurs, chacun présentant un rapport différent entre le coût de calcul et le nombre d'itérations économisées dans la boucle du CG. Certains d'entre eux peuvent être facilement obtenus à partir de la matrice du système et présentent un coût d'inversion négligeable. Comme par exemple le Jacobi qui consiste à prendre uniquement la diagonale de la matrice. Cependant, on s'attend à ce que des préconditionneurs aussi simples, requièrent toujours un grand nombre d'itérations. D'autres préconditionneurs sont plus précis, comme par exemple la factorisation incomplète de Cholesky (IC). Cette factorisation consiste à réaliser une factorisation de Cholesky dans laquelle on ignore les valeurs sous un certain seuil. Ceci permet d'accélérer la factorisation en limitant le

phénomène de remplissage, et diminue le coût d'application du préconditionneur. De plus, le résultat obtenu est une bonne approximation, puisque les valeurs ignorées influent peu sur le résultat. Cependant, le coût de la factorisation reste important.

Du point de vue de la simulation, les préconditionneurs ont déjà été très utilisés pour réduire les problèmes de convergence. Baraff et Witkin (1998) ont proposé d'utiliser le précondtionneur Jacobi pendant une simulation de vêtements. Choi et Ko (2002); Boxerman et Ascher (2004) ont étendu la méthode en utilisant les blocs 3×3 de la diagonale comme préconditionneur. Des préconditionneurs plus complexes tels que la factorisation incomplète de Cholesky ont également été utilisés Hauth et al. (2003). Cependant, les résultats montrent une amélioration des performances d'au maximum 20% dans une simulation générale, en raison des surcoûts d'application de la méthode. Enfin, plusieurs techniques ont été proposées en vue d'exploiter les spécificités des modèles de déformation. Boxerman et Ascher (2004) proposent un solveur très efficace, spécialement conçu pour les simulations de tissus, où seules les forces rigides sont intégrées de façon implicites. García et al. (2006) proposent d'exploiter une technique basée sur le calcul de la rotation globale des objets pour construire un très bon préconditionneur pour un modèle co-rotationnel. Ils calculent en dehors de la simulation l'inverse exacte du système, et ils actualisent leurs préconditionneurs utilisant la rotation globale de l'objet déformable.

Les méthodes multigrilles

La méthode multigrille consiste à utiliser récursivement des niveaux de plus en plus grossiers du problème. L'idée est de résoudre le système au niveau le plus grossier, puis récursivement d'utiliser la solution pour résoudre les niveaux plus fins. On décrit brièvement les concepts de cette méthode mais nous proposons aux lecteurs intéressés, de se reporter à Briggs et al. (2000) pour plus d'informations.

L'analyse modale du système fournit un ensemble de modes de vibration des objets pendant leur déformation. En possédant cette décomposition fréquentielle[3], on peut constater que les algorithmes itératifs sont très efficaces pour lisser les hautes fréquences, et beaucoup moins efficaces pour les basses fréquences. Cela se traduit par une décroissance rapide de l'erreur sur les premières itérations, et beaucoup plus lente par la suite. D'autre part, on peut montrer que les basses fréquences d'un niveau fin, se transforment en hautes fréquences à un niveau grossier. La méthode multigrille permet en quelque sorte d'utiliser les solveurs itératifs, uniquement dans des configurations où ils sont très efficaces.

Dans un schéma à deux niveaux, la méthode consiste tout d'abord à effectuer quelques itérations au niveau fin (appelé pré-lissage), afin de retirer les hautes fréquences. On procède ensuite à l'étape de restriction du résidu, qui permet de transformer la solution obtenue au niveau fin, en son équivalence niveau grossier. Cette étape consiste habituellement à retirer des degrés de liberté du problème. Ensuite, on calcule la correction au niveau grossier. Cette étape peut être faite soit à l'aide d'un autre solveur itératif, ou éventuellement avec un solveur direct si la taille du problème est petite. Il reste ensuite à transformer l'erreur du niveau grossier vers le niveau fin, ceci va permettre d'appliquer l'étape de correction afin de mettre à jour l'erreur calculée précédemment. Enfin, on procède au post-lissage, qui consiste à effectuer quelques itérations (souvent avec le même

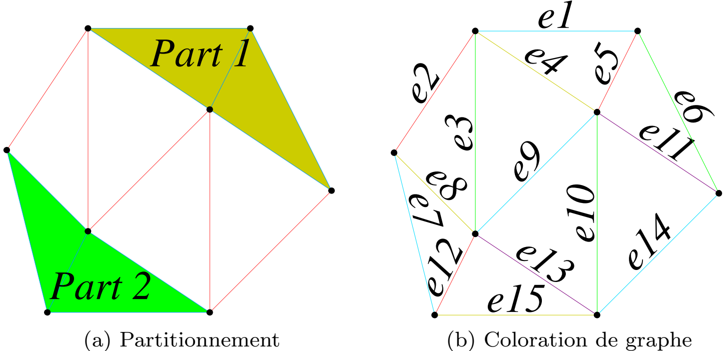
[3]Ce qui correspond aux modes de la décomposition de Fourrier.

solveur que dans la méthode de pré-lissage), avant de trouver la solution du problème initial. En pratique, il n'est pas obligatoire de se limiter à un schéma à deux niveaux, et il existe alors de nombreuses possibilités pour enchaîner les calculs des niveaux fins aux niveaux grossiers. On peut citer notamment le V-cycle, le N-cycle, et le W-cycle, ces noms symbolisent la façon dont sont enchaînées les étapes de calcul. D'autre part, cette méthode peut également servir de préconditionneur. En effet, même si on ne redescend pas jusqu'au niveau le plus fin, la solution obtenue reste une bonne approximation.

Cette méthode a été très largement utilisée dans le contexte de la simulation Zhu et al. (2010). C'est une des méthodes les plus efficaces pour résoudre des systèmes d'équations très grands. Mais l'accélération de ces techniques est moins flagrante pour des systèmes plus petits, de l'ordre de ceux utilisés dans une simulation interactive. De plus, la construction des niveaux hiérarchiques n'est pas une chose facile, surtout si le maillage n'est pas de structure régulière, ou possède des éléments de taille ou raideur fortement différentes.

3.2.3 Parallélisation de graphes

Pour terminer cet état de l'art, nous présentons des techniques de parallélisation de graphes, car les deux familles de solveurs que nous venons de présenter peuvent tirer parti de ces méthodes. En effet, nous avons vu que les solveurs directs ainsi que les solveurs itératifs nécessitent de réaliser des opérations matricielles, dont les dépendances sont directement définies par la structures des matrices. Or, toute matrice peut être vue comme un graphe où les nœuds représentent les indices de la matrice, et les arêtes correspondent aux coefficients non nuls. Une façon classique pour paralléliser les opérations matricielles est alors de considérer les nœuds du graphe comme des variables partagées, et les arêtes comme des calculs à effectuer.



(a) Partitionnement (b) Coloration de graphe

FIG. 3.3 – Les conflits d'écriture peut être éliminé par partitionnement des maillages, par la coloration des graphes ou en transformant la diffusion parallèles en une opération de réduction parallèle.

Une première technique de parallélisation Schloegel et al. (2000) est de partitionner le système en un ensemble de sous-graphes (voir figure 3.3a), chacun sera alors calculé par un processeur différent. Les arêtes qui relient plusieurs partitions seront calculées dans une étape séparée, exécutée en séquentielle. Le partitionnement tente ainsi de minimiser la taille de ces frontières, mais cette méthode n'est efficace que si le nombre de processeurs est faible par rapport au nombre de nœuds du graphe, sinon la plupart des arêtes devront être

traitées séquentiellement. Cette méthode est utilisée dans de nombreux solveurs (souvent exécutés sur des plates-formes distribuées) qui ont pour objectif de résoudre des systèmes d'équations bien plus grands que ceux produits par nos simulations en temps réel Camp et al. (1994).

Une autre méthode, nommée coloration de graphe (voir figure 3.3b), consiste à répartir les nœuds en sous-graphes, tels que l'arité[4] de tous les nœuds à l'intérieur d'un même sous-graphe soit d'au plus un. En d'autres mots, un nœud n'est jamais partagé par deux arêtes du même sous graphe. Cela permet à toutes les arêtes au sein de chaque sous-graphe d'être traitées en parallèle, mais $n-1$ synchronisations sont nécessaires, où n est le nombre de partitions (couleurs). Cette méthode est par exemple utilisée sur le GPU pour résoudre les contraintes de contact dans de grandes piles d'objets Tonge et al. (2010).

Dans la section suivante, nous proposerons une solution pour paralléliser ce type de graphe sur GPU.

3.3 Intégrateur implicite optimisé sur GPU

Cette section est consacrée à la description de notre proposition pour paralléliser un solveur éléments finis implicite sur GPU Allard et al. (2011b). Comme nous l'avons expliqué dans la section 2.4, le choix de l'intégrateur implicite nous permet d'utiliser de grands pas de temps et d'améliorer la stabilité des simulations. Ces aspects étaient un point limitant des précédentes implémentations FEM sur le GPU, qui ne portaient que sur des intégrateurs explicites. D'autre part, contrairement aux solveurs itératifs de matrices creuses existants sur GPU, notre solution ne nous oblige pas à construire explicitement la matrice du système. Cette propriété permet de réduire considérablement le nombre d'opérations effectuées. Et surtout, cela permet également de réduire la bande passante mémoire consommée, cet aspect était le principal facteur limitant des précédentes implémentations de solveur GPU pour matrice creuse. Le coeur de la méthode peut être utilisé dans de nombreuses applications scientifiques qui nécessitent de résoudre de grands systèmes d'équations irrégulières et creuses.

3.3.1 Parallélisation de l'intégrateur implicite sur GPU

Tout d'abord, nous présentons en figure 3.4 une vue schématique de l'ensemble des opérations nécessaires à la mise en œuvre des versions implicites et explicites des intégrateurs, basées sur les équations (2.37) et (2.38). Le graphe de la version implicite est mis en correspondance avec l'algorithme 4. Ce dernier est constitué de l'algorithme du gradient conjugué (lignes 3 à 15), et de quelques opérations réalisant l'intégration implicite (mise à jour de la vitesse $\boldsymbol{v}$ en fonction de l'accélération $\boldsymbol{x}$, et mise à jour de la position $\boldsymbol{p}$ en fonction de la vitesse $\boldsymbol{v}$). L'algorithme du gradient conjugué a été légèrement modifié par rapport à l'algorithme 3, en vue de minimiser les transferts mémoire, cependant ces algorithmes sont équivalents. La ligne 7 de l'algorithme 3, qui consistait à multiplier la matrice du système $\mathbf{A}$ par un vecteur, a été décomposée en l'application successive de la masse et de la rigidité. En effet, nous avons dans cette version supprimé la matrice d'amortissement, la matrice du système est alors considérée égale à $(\mathbf{M} - h^2\mathbf{K})$. L'amor-

[4] Le nombre d'arrètes connecté à un nœud.

tissement peut cependant être pris en compte en utilisant la méthode de rayleigh comme expliqué dans la section 2.3.2. De plus, les lignes 1 et 2 consistent à évaluer la force en fonction des positions et vitesses actuelles, on nomme cette opération **force**. Alors que les lignes 7 et 8 servent à évaluer la différence de force en fonction d'une différence de déplacement, on appelle cette tâche **dForce**.

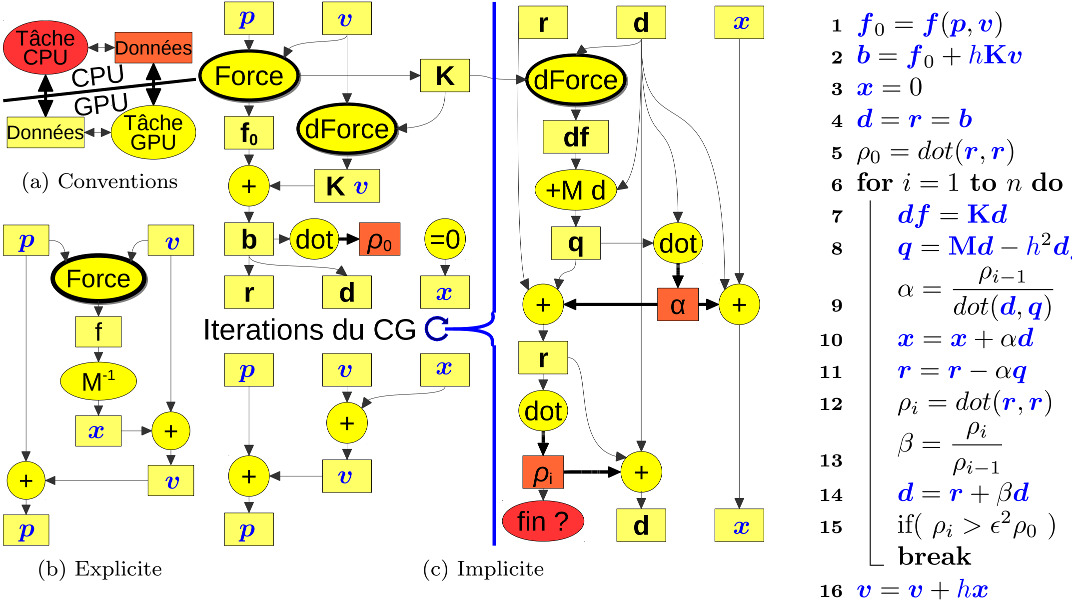


FIG. 3.4 – Graphe de tâches des versions explicites et implicites des intégrateurs. Les données sur CPU sont représentés en rouge, et celles sur GPU sont indiqués en jaune.

1 $\boldsymbol{f}_0 = \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v})$
2 $\boldsymbol{b} = \boldsymbol{f}_0 + h\mathbf{K}\boldsymbol{v}$
3 $\boldsymbol{x} = 0$
4 $\boldsymbol{d} = \boldsymbol{r} = \boldsymbol{b}$
5 $\rho_0 = dot(\boldsymbol{r}, \boldsymbol{r})$
6 **for** $i = 1$ **to** n **do**
7 $\quad \boldsymbol{df} = \mathbf{K}\boldsymbol{d}$
8 $\quad \boldsymbol{q} = \mathbf{M}\boldsymbol{d} - h^2\boldsymbol{df}$
9 $\quad \alpha = \dfrac{\rho_{i-1}}{dot(\boldsymbol{d}, \boldsymbol{q})}$
10 $\quad \boldsymbol{x} = \boldsymbol{x} + \alpha\boldsymbol{d}$
11 $\quad \boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{q}$
12 $\quad \rho_i = dot(\boldsymbol{r}, \boldsymbol{r})$
13 $\quad \beta = \dfrac{\rho_i}{\rho_{i-1}}$
14 $\quad \boldsymbol{d} = \boldsymbol{r} + \beta\boldsymbol{d}$
15 $\quad$ if($\rho_i > \epsilon^2\rho_0$)
$\quad$ **break**
16 $\boldsymbol{v} = \boldsymbol{v} + h\boldsymbol{x}$
17 $\boldsymbol{p} = \boldsymbol{p} + h\boldsymbol{v}$

Algorithme 4 : Euler implicite + CG

En comparant les figures 3.4b et 3.4c, on constate que la version explicite est beaucoup plus simple à mettre en œuvre. Cependant, on remarque également que toutes deux reposent essentiellement sur des opérations vectorielles standards (soit des combinaisons linéaires de vecteurs, soit des produits scalaires), à l'exception de **Force** et **dForce**. Pour les opérations vectorielles, il existe des solutions (CUBLAS) de parallélisation efficaces sur GPU, qui permettent de les réaliser sur le processeur graphique (représentés en jaune). On peut alors déjà remarquer que, malgré la complexité de la version implicite, si les tâches **Force** et **dForce** sont également exécutées sur GPU, alors la quasi-totalité de l'algorithme peut-être exécutée sur le GPU. En effet, seules deux scalaires (représentés en rouge) doivent transiter entre le GPU et le CPU, afin de vérifier la convergence du gradient conjugué. La convergence ne peut pas être directement calculée sur le GPU, car seul le CPU est en mesure de lancer de nouveaux kernels pour effectuer des itérations supplémentaires si nécessaire. Les deux étapes **force** et **dForce** sont alors les plus complexes et les plus coûteuses à mettre en œuvre, nous détaillerons leur implémentation GPU dans les sections suivantes.

3.3.2 Stockage mémoire pour maximiser les accès coalescent

Avant toute chose, il est important de s'intéresser au format de données que nous allons utiliser par la suite. Pour obtenir des performances maximales, il est essentiel de concevoir des structures de données en mémoire qui permettent d'accéder de la façon la plus régulière possible. Une approche commune pour améliorer l'efficacité des accès mémoire est de convertir de grands tableaux de structures (voir figure 3.5a) en une structure de tableaux (voir figure 3.5b). Pour accéder à la variable **m** d'un élément **i**, **e.m[i]** sera utilisé à la place de **e[i].m**. Cette approche permet de réaliser des lectures alignées si plusieurs threads doivent accéder au même champ d'une structure de données.

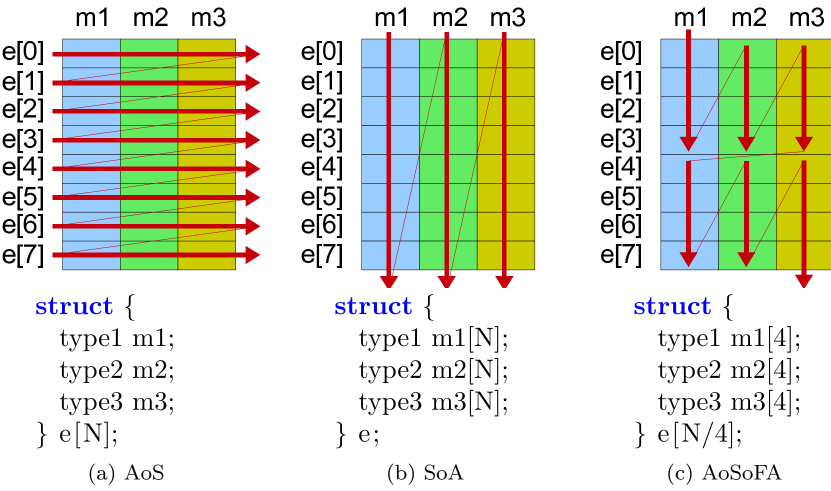


FIG. 3.5 – L'accès à la mémoire peut être optimisé en remplaçant le tableau de structure *Array-of-Structs* 3.5a en une structure de tableau *Struct-of-Arrays* 3.5b, où sur un GPU par un tableau de structure, dont les champs sont des tableaux de taille fixe *Array-of-Struct-of-Fixed-Arrays 3.5c*.

Une approche plus efficace, spécialement pour les processeurs SIMD ou pour le déroulage de boucles avec des blocs de b valeurs, est d'utiliser un tableau de structures où chaque membre est remplacé par un tableau de taille fixe b (voir figure 3.5c). L'accès à une variable **m** de l'élément **i** sera alors transformé de **e[i].m** à **e[i/b].m[i%b]**. Or, si b est une puissance de 2, les indices peuvent être obtenus par de simples opérations binaires. Sur le GPU si b correspond à la taille des blocs de threads, alors l'indice de bloc fournit directement l'index à utiliser dans **e**, et l'indice du thread fournit l'index à utiliser dans **m**.

Cette approche est utilisée pour stocker toutes les données des structures des éléments finis, à l'exception des vecteurs d'état des objets (comme les positions ou les vitesses) qui exigent parfois d'accéder de façon aléatoire aux données. Pour ces vecteurs, une optimisation consiste à utiliser la mémoire de texture, présente sur les GPU, pour effectuer des lectures non alignées. D'autre part, il est possible d'optimiser l'alignement des lectures mémoire en ré-organisant au préalable le maillage. Ainsi, si par exemple plusieurs threads d'un même groupe désirent connaître la vitesse des points d'un élément, on peut s'attendre à ce que les performances soient maximisées si ils doivent accéder au même

banc mémoire (voir section 1.4.3). Nous montrerons dans les résultats qu'un simple tri selon un axe permet d'améliorer les performances car les indices des éléments devraient se retrouver plus proche en mémoire. Cependant, il est possible d'utiliser des techniques plus avancées pour maximiser la cohérence des caches Tchiboukdjian et al. (2010).

3.3.3 Intégration implicite sans assemblage de matrice

Parallélisation du calcul FEM sur GPU

Bien que les calculs à effectuer pour chaque objet soient spécifiques à la formulation FEM, leurs parallélisations dépendent uniquement du maillage éléments finis. En effet, peu importe la façon dont sont calculées les nouvelles forces dans les opérations **Force** et **dForce**, il sera toujours nécessaire d'accumuler les contributions de chaque élément sur les nœuds du maillage.

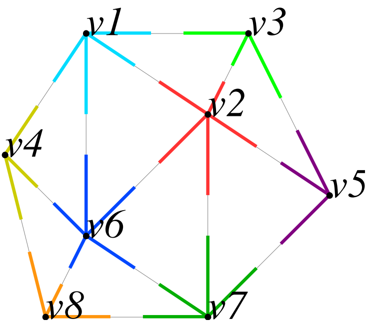


FIG. 3.6 – Parallélisation du calcul des éléments finis sur GPU.

En théorie, ces deux opérations nécessitent de calculer le produit de la matrice $\mathbf{K}$ par un vecteur. Mais, pour paralléliser ces opérations nous proposons de ne pas assembler la matrice $\mathbf{K}$, mais plutôt d'effectuer en parallèle, les calculs à partir des matrices élémentaires $\mathbf{K}_e$. Ce problème peut être vue comme un problème de parallélisation de graphe similaire à celui que nous avons évoqué dans la section 3.2.3. En effet, toutes les arrêtes connectées à un même nœud du maillage élément fini, doivent accumuler leur contribution de la force finale, ce qui pose des problème d'écriture. Pour paralléliser efficacement ce type de calcul sur GPU, nous proposons de transformer l'opération de *diffusion parallèles* en une opération de *réduction parallèle*. Au lieu de calculer les arêtes indépendamment, les nœuds sont traités en parallèle, chacun regroupant les résultats des arêtes connectées (voir figure 3.6). Pour cela, on va alors procéder en deux étapes (voir figure 3.7) :

1. En utilisant un thread par élément, tous les calculs par élément sont réalisées en parallèle, et le résultat est stocké de façon indépendante pour chaque élément dans un vecteur temporaire.
2. En utilisant un thread par nœud, nous accumulons les résultats temporaires sur chaque sommet, en lisant les adresses d'un tableau pré-calculé[5].

[5]Pour faciliter la lecture de la figure 3.7, nous avons représenté le vecteur pré-calculé sous sa forme standard. En réalité, nous stockons ce tableau dans le format AoSoFA (voir section 3.3.2), pour pouvoir accéder de façon alignée à des points différents simultanément.

La séparation en deux étapes correspond à une synchronisation globale[6], et s'explique par le fait que plusieurs éléments sont reliés à un même sommet, ce qui provoque des conflits d'écritures si les threads écrivent directement dans le vecteur solution.

Pour résoudre ce problème, on utilise un premier kernel où chaque thread est associé à un élément e. Chacun calcul alors la contribution partielle de la force, au sein de l'élément e en utilisant uniquement la matrice $\mathbf{K}_e$. Comme tous les éléments possèdent le même nombre de degré de liberté, on peut alors utiliser les identifiants des nœuds pour déterminer une adresse mémoire unique dans un tableau temporaire.

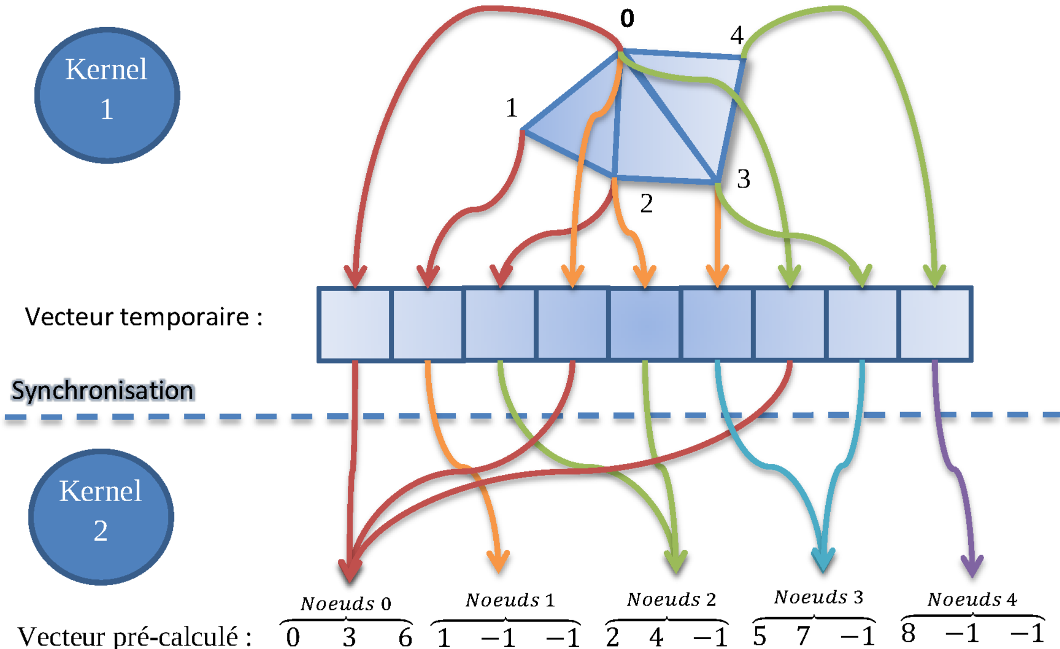


Fig. 3.7 – Parallélisation du calcul de la force pendant l'algorithme du CG.

Il reste ensuite à sommer les contributions partielles de tous les nœuds qui sont connectés à plusieurs éléments. Cela nécessite de connaître l'adresse de toutes les contributions d'un même nœud dans le vecteur temporaire. Cette information peut directement être obtenue à partir du maillage éléments finis, et tant qu'aucun changement topologique n'est simulé[7], on peut la pré-calculer dans un vecteur qui sera transféré une seule fois sur le GPU. Pour faciliter la lecture de ce vecteur, on associe **nbElemPerVertex** voisins à chaque nœud, où **nbElemPerVertex** est le nombre maximal d'éléments connectés à un même nœud. On va alors stocker **nbElemPerVertex** indices pour chacun des sommets (l'absence de données est codée par -1), ce qui nous permet de lancer un thread par vertex, car chaque thread peut ainsi connaître le début et le nombre de données qu'il doit accumuler.

[6]Sur les architectures GPU récentes, on peut noter qu'il est possible de retirer le deuxième kernel en utilisant les opérations atomiques, qui permettent d'accumuler les résultats directement de la première étape. Cependant, ces opérations ne sont pas encore supportées pour les données encodées en précision double.

[7]Si on désire simuler des changements topologiques, il est nécessaire de recalculer ce vecteur pour chaque modification, mais ce calcul est rapide et peut être fait en temps réel.

Listing 3.1 – Réduction parallèle avec 4 threads par nœuds

```
__global__ void addForce4(int nbVertex, int nb4ElemPerVertex,
    const float4* eforce, const int* velems, float* f) {
    int index1 = threadIdx.x;
    __shared__ float temp[BSIZE*3]; // Shared memory buffer
    float3 force = {0,0,0};
    // index into AoSoFA array velems with b = BSIZE/4
    int entry = blockIdx.x*nb4ElemPerVertex*BSIZE+index1;
    for (int s = 0; s < nb4ElemPerVertex; ++s) {
        int i = velems[entry]; // index into the temporary vector
        if (i == -1) break;
        float3 fe = getElemForce(i,eforce); // using memory access or texture
        force.x += fe.x; force.y += fe.y; force.z += fe.z;
        entry += BSIZE;
    }
    int iout = ((index1/4) + (index1&3)*(BSIZE/4))*3;
    temp[iout] = force.x;
    temp[iout+1] = force.y;
    temp[iout+2] = force.z;
    __syncthreads(); // BARRIER
    if (index1 < (BSIZE*3/4)) { // we need to merge 4 values together
        float res = temp[index1] +
        temp[index1 + (BSIZE*3/4)] +
        temp[index1 + 2*(BSIZE*3/4)] +
        temp[index1 + 3*(BSIZE*3/4)];
       f[blockIdx.x*(BSIZE*3/4) + index1] += res;
    }
}
```

Avec le vecteur d'association, on peut alors écrire une deuxième kernel pour rassembler les contributions partielles. Cependant, comme les maillages volumiques contiennent en général un plus petit nombre de nœuds que d'éléments, chaque thread aura plusieurs données à traiter (autant qu'il y a d'éléments connectés au nœud). Une optimisation possible consiste alors à utiliser plusieurs threads par sommet, pour réaliser l'accumulation d'un même nœud. Le kernel CUDA résultant en utilisant 4 threads par nœud, est représenté sur le listing 3.1. Les lignes 3 à 18 correspondent à l'accumulation des forces temporaires. Chaque thread lit les indices des forces à accumuler, dans le vecteur pré-calculé **velems**. Pendant cette boucle $BSIZE/4$ groupes de threads accumulent partiellement la force d'un même point, alors qu'a l'intérieur de chaque groupe, les threads calculent tous des forces différentes. Ceci est implicitement décrit par le format de stockage AoSoFA de **velems**, en utilisant comme taille de blocs $b = BSIZE/4$ pour 4 threads. Les résultats partiels sont stockés en mémoire partagée dans le tableau **temp**, en utilisant un indice unique par thread (ligne 15). Enfin, les lignes 19 à 26 permettent de rassembler le résultat partiel calculé dans les 4 groupes précédents qui travaillaient sur le même vertex.

3.3.4 Optimisations pour le modèle co-rotationnel

Dans la formulation FEM co-rotationnelle, chaque tétraèdre contribue sur la force, par une matrice dense de taille 12×12, qui est le résultat du produit de 5 matrices plus

petites, dont certaines changent à chaque étape (voir équation 2.49). Au lieu de calculer et de stocker la matrice finale, nous cherchons à stocker les quelques scalaires nécessaires à l'expression des sous matrices. Nous présentons tout d'abord, la structure de chacune des matrices nécessaires qui interviennent dans le calcul.

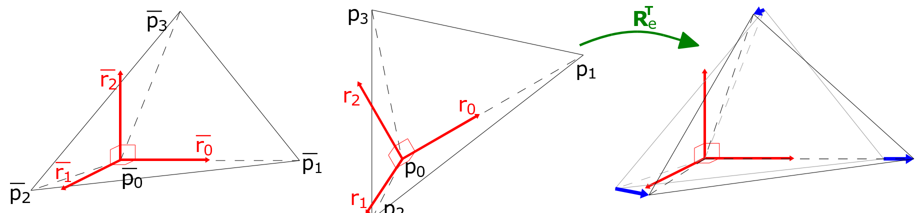


FIG. 3.8 – FEM co-rotationnel. Gauche : un élément tétraédrique dans la configuration de référence avec son repère local. Milieu : le tétraèdre déformé et son repère local tourné. À droite : la déformation (flèches bleues) telle que mesurée après l'alignement des repères.

La méthode la plus rapide (mais pas la plus précise) pour calculer le repère local d'un élément (voir figure 3.8) est de définir l'origine au sommet $\boldsymbol{p}_0$ et d'utiliser ses bords adjacents Nesme et al. (2005a). Le premier vecteur, $\boldsymbol{b}_0$, est initialisé sur la première arête $\boldsymbol{p}_0\boldsymbol{p}_1$. Le deuxième vecteur, $\boldsymbol{b}_1$, doit être perpendiculaire au premier, dans le plan de la première et de la deuxième arête $\boldsymbol{p}_0\boldsymbol{p}_2$, et le troisième vecteur, $\boldsymbol{b}_2$, doit être perpendiculaire à ce plan :

$$\boldsymbol{b}_0 = \frac{(\boldsymbol{p}_1-\boldsymbol{p}_0)}{\|(\boldsymbol{p}_1-\boldsymbol{p}_0)\|} \ , \quad \boldsymbol{b}_2 = \frac{(\boldsymbol{p}_1-\boldsymbol{p}_0)\wedge(\boldsymbol{p}_2-\boldsymbol{p}_0)}{\|(\boldsymbol{p}_1-\boldsymbol{p}_0)\wedge(\boldsymbol{p}_2-\boldsymbol{p}_0)\|} \ , \quad \boldsymbol{b}_1 = \boldsymbol{b}_2 \wedge \boldsymbol{b}_0 \tag{3.14}$$

Les éléments 3×3 de la matrice de rotation $\mathbf{R}_e = \begin{bmatrix} \boldsymbol{b}_0 & \boldsymbol{b}_1 & \boldsymbol{b}_2 \end{bmatrix}$ sont utilisés pour calculer la position non déformée $(a\ b\ c\ d)$ dans le repère tourné, avec l'origine $\boldsymbol{p}_0$, comme suit :

$$\begin{array}{l} a = \mathbf{R}_e^\mathsf{T}\,(\boldsymbol{p}_0 - \boldsymbol{p}_0) = (\ 0 \quad 0 \quad 0\ )^\mathsf{T} \\ b = \mathbf{R}_e^\mathsf{T}\,(\boldsymbol{p}_1 - \boldsymbol{p}_0) = (\ b_x \quad 0 \quad 0\ )^\mathsf{T} \\ c = \mathbf{R}_e^\mathsf{T}\,(\boldsymbol{p}_2 - \boldsymbol{p}_0) = (\ c_x \quad c_y \quad 0\ )^\mathsf{T} \\ d = \mathbf{R}_e^\mathsf{T}\,(\boldsymbol{p}_3 - \boldsymbol{p}_0) = (\ d_x \quad d_y \quad d_z\ )^\mathsf{T} \end{array} \tag{3.15}$$

En utilisant ces points, la matrice 6×12 de contrainte $\mathbf{F}_e$ est définie dans le repère local comme suit :

$$\mathbf{F}_e = \begin{pmatrix} -p_x(bcd) & 0 & 0 & -p_y(bcd) & 0 & -p_z(bcd) \\ 0 & -p_y(bcd) & 0 & -p_x(bcd) & -p_z(bcd) & 0 \\ 0 & 0 & -p_z(bcd) & 0 & -p_y(bcd) & -p_x(bcd) \\ p_x(cda) & 0 & 0 & p_y(cda) & 0 & p_z(cda) \\ 0 & p_y(cda) & 0 & p_x(cda) & p_z(cda) & 0 \\ 0 & 0 & p_z(cda) & 0 & p_y(cda) & p_x(cda) \\ -p_x(dab) & 0 & 0 & -p_y(dab) & 0 & -p_z(dab) \\ 0 & -p_y(dab) & 0 & -p_x(dab) & -p_z(dab) & 0 \\ 0 & 0 & -p_z(dab) & 0 & -p_y(dab) & -p_x(dab) \\ p_x(abc) & 0 & 0 & p_y(abc) & 0 & p_z(abc) \\ 0 & p_y(abc) & 0 & p_x(abc) & p_z(abc) & 0 \\ 0 & 0 & p_z(abc) & 0 & p_y(abc) & p_x(abc) \end{pmatrix}^\mathsf{T} \tag{3.16}$$

Avec $p(uvw) = u \wedge v + v \wedge w + w \wedge u$.

En combinant l'équation (3.16) et (3.15) nous obtenons :

$$S_b = p(cda) = \begin{pmatrix} c_y d_z \\ -c_x d_z \\ c_x d_y - c_y d_x \end{pmatrix}, S_c = -p(dab) = \begin{pmatrix} 0 \\ b_x d_z \\ -b_x d_y \end{pmatrix}, S_d = \begin{pmatrix} 0 \\ 0 \\ b_x c_y \end{pmatrix} \quad (3.17)$$

Et puisque la somme des forces appliquées doit être nulle, nous savons que :

$$S_a + S_b + S_c + S_d = 0 \quad \Leftrightarrow \quad S_a = -S_b - S_c - S_d$$

Enfin, la matrice de rigidité du matériau est représentée par une matrice 6×6 nommée $\mathbf{E}_e$ comme suit :

$$\mathbf{E}_e = \begin{pmatrix} \gamma+\mu & \gamma & \gamma & 0 & 0 & 0 \\ \gamma & \gamma+\mu & \gamma & 0 & 0 & 0 \\ \gamma & \gamma & \gamma+\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu/2 \end{pmatrix}, \quad \begin{array}{l} \gamma = \frac{1}{36V_e}\frac{E\nu}{(1+\nu)(1-2\nu)} \\ \mu = \frac{1}{36V_e}\frac{E}{1+\nu} \\ V_e = \frac{1}{6}b \cdot (c \times d) \end{array} \quad (3.18)$$

Où E est le module de Young, représentant la rigidité. $0 < \nu < 0,5$ est le coefficient de Poisson lié à la conservation du volume, et V_e le volume de l'élément e (voir section 2.2.3).

Étant donné les matrices ci-dessus, les forces et leurs dérivées sont calculées en utilisant l'équation (2.49). Nous voulons trouver le plus petit nombre de valeurs qui peuvent être utilisées pour calculer le produit matrice–vecteur $\mathbf{R}_e\ \mathbf{F}_e{}^T\ \mathbf{E}_e\ \mathbf{F}_e\ \mathbf{R}_e{}^T\ \boldsymbol{u}_e$. Pour chacune d'entre elles, il est possible soit de stocker directement les valeurs finales, soit d'utiliser les valeurs initiales i_a, i_b, i_c et i_d pour les recalculer :

- $\mathbf{R}_e$ peut être appliqué avec 9 valeurs (la matrice complète), ou 6 valeurs (deux vecteurs) et un produit vectoriel pour calculer les dernières valeurs (ce qui nécessite 9 opérations), ou 4 (un quaternion) et 30 opérations pour obtenir la matrice.
- $\mathbf{E}_e$ peut être appliqué en utilisant 2 valeurs (γ, μ).
- $\mathbf{F}_e$ peut être appliqué en utilisant 6 valeurs (Sb_x, Sb_y, Sb_z, Sc_y, Sc_z, Sd_z), ou 1 valeur (Sb_z) et 5 opérations pour recalculer les autres composantes. Une autre alternative est de diviser S_e par b_x, et de multiplier D_e par b_x^2 : $\mathbf{R}_e\ (\frac{1}{b_x}\mathbf{F}_e{}^\mathsf{T})\ (b_x^2\mathbf{E}_e)\ (\frac{1}{b_x})\mathbf{F}_e\ \mathbf{R}_e{}^\mathsf{T}\ \boldsymbol{u}_e$. Cela nous permet de stocker seulement 3 valeurs ($\frac{Sb_x}{b_x}$, $\frac{Sb_y}{b_x}$, $\frac{Sb_z}{b_x}$) sans avoir besoin d'opérations supplémentaires pour calculer les 3 autres.

Listing 3.2 – Structure de données pour calculer la matrice de rigidité sur GPU

```
struct GPUElement {
    /// index of the 4 connected vertices
    int ia[BSIZE], ib[BSIZE], ic[BSIZE], id[BSIZE];
    /// initial position of the vertices in the local frame
    float bx[BSIZE];
    float cx[BSIZE], cy[BSIZE];
    float dx[BSIZE], dy[BSIZE], dz[BSIZE];
    /// Values to compute the strain-displacement matrix S
    float Sbx_bx[BSIZE],Sby_bx[BSIZE],Sbz_bx[BSIZE];
    /// Values to compute the material stiffness matrix D
    float gamma_bx2[BSIZE], mu_bx2[BSIZE];
};
```

Dans notre implémentation actuelle, nous stockons la rotation à partir de 6 valeurs dans un vecteur séparé, car il doit être mis à jour à chaque pas de temps. Le reste des valeurs est stocké dans une structure mémoire représentée sur le listing 3.2. Cette structure est stockée dans un vecteur contenant un exemplaire pour chaque bloc de thread (voire section 3.3.2) afin de minimiser l'utilisation de la bande passante, qui est le principal facteur limitant de l'algorithme.

3.3.5 Réduction des surcoûts en fusionnant les kernels

Dans une application interactive, seules quelques millisecondes sont disponibles pour calculer un pas de temps complet, ce qui peut impliquer des centaines d'étapes de calcul différentes. Dans un tel contexte, la performance obtenue peut être grandement affectée par le surcoût de lancement de chaque kernel. Il est donc important de fusionner autant d'étapes que possible pour réduire le coût de lancement.

Fusionner des kernels ne supprime pas seulement le surcoût de leur lancement, cela peut aussi réduire la bande passante consommée, car les données calculées par une étape peuvent être directement utilisées par les autres kernels sans repasser par la mémoire globale du GPU. Le calcul FEM de la force $\boldsymbol{f}$ et sa dérivée $\boldsymbol{df}$ est traitée dans deux kernels CUDA qui ne peuvent pas être fusionnés, car elles reposent sur une synchronisation globale. Cependant quelques autres étapes pourraient tout à fait être combinées et faire gagner du temps. Dans la boucle principale de l'algorithme du gradient conjugué (voir figure 3.4c), nous pouvons voir 5 opérations d'algèbre linéaire sur des vecteurs (lignes 9 à 12 et 14) et deux produits matrice-vecteur (les lignes 7 et 8).

Les opérations successives d'algèbre linéaire sont les plus simples à fusionner. C'est le cas de la mise à jour des vecteurs $\boldsymbol{v}$ et $\boldsymbol{p}$ à la fin du pas de temps (lignes 16 et 17), où $\boldsymbol{x}$ et $\boldsymbol{r}$ dans chaque itération du gradient conjugué (lignes 10 et 11). Ces étapes peuvent être fusionnées très facilement, en écrivant un seul kernel qui effectue les deux mises à jour en même temps.

D'autre part, chacune des lignes 9 et 12 ont besoin d'une réduction parallèle pour calculer le produit scalaire. On pourrait les fusionner si nous pouvons évaluer $dot(\boldsymbol{r},\boldsymbol{r})$ avant de connaître la valeur de α qui sert à mettre à jour $\boldsymbol{r}$ (ligne 11), en soustrayant $\alpha\boldsymbol{q}$ de sa valeur précédente (que nous noterons que $\boldsymbol{r}'$ dans la suite). Pour cela, on peut exprimer le produit scalaire comme suit :

$$\boldsymbol{r}\cdot\boldsymbol{r} = (\boldsymbol{r}'-\alpha\boldsymbol{q})\cdot(\boldsymbol{r}'-\alpha\boldsymbol{q}) = (\boldsymbol{r}'\cdot\boldsymbol{r}') - 2\alpha(\boldsymbol{r}'\cdot\boldsymbol{q}) + \alpha^2(\boldsymbol{q}\cdot\boldsymbol{q}) \tag{3.19}$$

Par conséquent, nous pouvons calculer en une seule réduction parallèle les étapes de $\boldsymbol{d}\cdot\boldsymbol{q}$, $\boldsymbol{r}'\cdot\boldsymbol{q}$, et $\boldsymbol{q}\cdot\boldsymbol{q}$ ($\boldsymbol{r}'\cdot\boldsymbol{r}'$ étant connu de l'itération précédente). Après une synchronisation, nous pouvons calculer α et ρ_i ce qui nous permettra de mettre à jour tous les vecteurs (lignes 10, 11 et 14) en une seule fois.

3.3.6 Résultats et Évaluations

Nous cherchons tout d'abord à évaluer notre méthode en la comparant à une version CPU. Pour cela, nous présentons les temps de calcul du CG, en fonction du nombre de tétraèdres, des améliorations apportées par les optimisations successives, à la fois sur les versions CPU

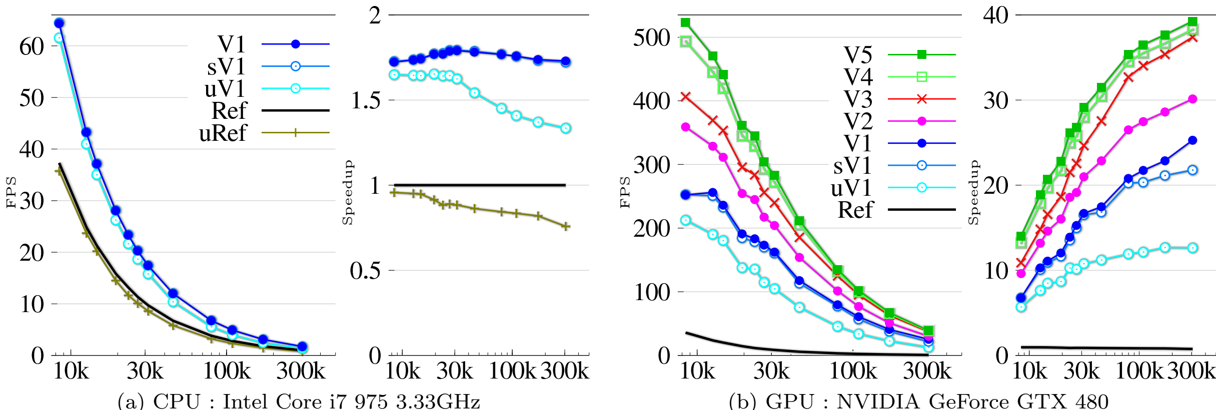
(voir figure 3.10a) et GPU (voir figure 3.10b). Ces mesures ont été faites sur l'exemple des dinosaures (voir figure 3.9), en limitant à 25 le nombre d'itérations du gradient conjugué. Nous avons utilisé comme référence le temps de calcul de l'implémentation initiale du FEM co-rotationnel de SOFA Allard et al. (2007).

Fig. 3.9 – Un objet déformable (en haut) et son maillage FEM (en bas), soumis à une force externe (ligne rose) et à des contraintes (points roses).

La figure 3.10 présente les améliorations en terme de performances pour les différentes optimisations. La représentation compacte des données (**V1**), présentée en section 3.3.2, permet d'obtenir une accélération significative, de l'ordre de $1,7\times$ plus rapide que la version initiale sur un CPU, et jusqu'à $14\times$ sur le GPU. Le fait de ré-organiser les points permet d'améliorer les lectures alignées sur le GPU, et de maximiser les lectures dans le cache sur le CPU. Les versions non ré-ordonnées, sont préfixées par la lettre **u**, et un simple ré-ordonnancement comme de trier les particules selon un axe donné (voir section 3.3.2), permet d'obtenir un gain significatif allant jusqu'à $1,3\times$ sur un CPU et jusqu'à $2\times$ sur un GPU. Le fait d'utiliser une méthode de ré-ordonnancement plus complexe comme présentée dans Tchiboukdjian et al. (2010), peut apporter un gain supplémentaire de l'ordre de 16%, sur des maillages de grandes tailles.

L'étape de calcul de la force (voir section 3.3.3) est une étape critique dans le déroulement de l'algorithme, car elle consomme environ la moitié du temps de calcul d'un pas de temps. L'utilisation de 4 threads par particule (**V2**) réduit de façon significative ce temps, offrant une amélioration globale de $1,3\times$. L'utilisation d'unités de textures (**V3**) pour réaliser les accès mémoire non alignés est bénéfique (jusqu'à +20%) sur les architectures Fermi.

Les dernières optimisations (**V4** et **V5**) consistent à fusionner les kernels (voir section 3.3.5). Cette optimisation est très utile pour les petits maillages (+37%), car elle permet de réduire le coût de lancement de la totalité des kernels, qui dans ce cas est important par rapport à leur temps d'exécution. Cependant, cette optimisation est également bénéfique



(a) CPU : Intel Core i7 975 3.33GHz
(b) GPU : NVIDIA GeForce GTX 480

FIG. 3.10 – Performances des différentes optimisations : **Ref** : implémentation de référence non optimisée sur CPU ; **V1** : optimisation du stockage des matrices (Sect. 3.3.2) ; **uref** et **uV1** : utilisation des maillages non ordonnés ; **V2** : en utilisant 4 threads par particule pour la réduction parallèle (Sect. 3.3.3) ; **V3** accès mémoire en utilisant les textures pour les positions et les forces ; **V4** et **V5** fusion des kernels en utilisant respectivement l'algèbre linéaire et la reformulation des équations (Sect. 3.3.5).

pour de grands maillages (+7%) car elle réduit également la bande passante mémoire.

Dans l'ensemble, en combinant toutes les optimisations on peut atteindre une accélération de 39× plus rapide que l'implémentation de référence. Ainsi, nous sommes capables de simuler un objet déformable avec 45k éléments tétraédriques (voir figure 3.9) à 212 FPS sur une NVIDIA GeForce GTX 480, 18× plus vite qu'avec l'implémentation CPU la plus optimisée sur un Intel Core i7 975 à 3,33 GHz. Nous avons également comparé les performances sur plusieurs architectures en utilisant des tailles de maillages diverses (voir figure 3.11). Avec le plus grand maillage, l'accélération peut atteindre 23× sur une GeForce GTX 480, et de $13,5\times$ sur une GTX 280.

Dans la mesure où notre objectif était de comparer l'accélération de la version parallèle, par rapport à la version séquentielle, nous avons basé nos expériences sur un exemple où le système était bien conditionné. Tous les éléments étaient globalement de la même taille, relativement mous, et leur rigidité variait très peu. De plus, nous avons limité à 25 le nombre d'itérations du CG, puisque notre objectif n'était pas d'obtenir une solution d'une grande qualité, quitte à ce que l'algorithme ne converge pas à chaque pas de temps. Nous allons par la suite présenter une technique de préconditionnement qui permet d'améliorer sensiblement la qualité de la solution tout en restant compatible avec notre intégrateur implicite sur GPU.

3.4 Techniques de pré-conditionnement asynchrone

Nous avons vu dans la section précédente comment paralléliser efficacement le gradient conjugué sur GPU, ce qui nous a permis de simuler plusieurs centaines de milliers d'éléments en temps réel sans se soucier de la précision des calculs. Dans cette section nous nous intéressons à l'autre aspect de la problématique, c'est-à-dire simuler des objets pour lesquels le système est très mal conditionné, tout en souhaitant une précision relativement grande. Dans ce cas, même si nous disposons d'une méthode très rapide pour calculer les itérations du CG, les performances globales de la simulation pourraient être largement

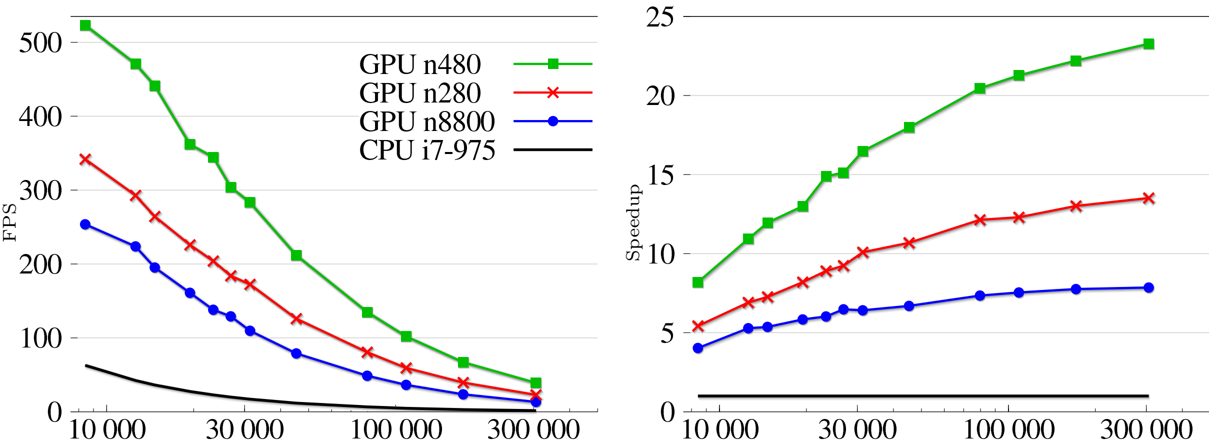


FIG. 3.11 – Comparaison des performances entre les versions CPU et GPU sur différentes architectures, en fonction du nombre de tétraèdres.

diminuées si des milliers d'itérations sont nécessaires pour obtenir une solution acceptable. Nous proposons dans cette section une méthode de préconditionnement qui permet d'augmenter sensiblement la qualité de la solution, tout en restant compatible avec la version parallèle de l'algorithme Courtecuisse et al. (2010). L'approche proposée est particulièrement bénéfique pour simuler des objets non-homogènes ou pour les problèmes mal conditionnés.

La première contribution est de désynchroniser le calcul d'un préconditionneur de la boucle de simulation. Pour cela, nous utilisons les architectures parallèles hétérogènes actuelles : le processeur graphique effectue les calculs mécaniques tandis que le CPU multi-cœurs calcule simultanément les prochains préconditionneurs. Nous exploitons la cohérence temporelle de la mécanique, pour mettre à jour de façon asynchrone un préconditionneur. Nous proposerons également une méthode qui permet de compenser les non-linéarités géométriques, afin de limiter la divergence du préconditionneur au fil du temps. Regardons dans un premier temps l'influence d'un préconditionneur mis à jour de façon synchrone avec la simulation.

3.4.1 Préconditionneur ré-utilisable

Comme nous l'avons vu en introduction, la simulation consiste en réalité à calculer successivement l'évolution de l'état des objets virtuels après de petits intervalles de temps. Ceci nous permet de supposer que la configuration mécanique des objets reste très similaire pendant plusieurs pas de temps successifs de la simulation. On va alors montrer comment exploiter la cohérence temporelle de la mécanique pour construire un préconditionneur efficace à très faible coût.

Cette idée n'est pas nouvelle, comme nous l'avons vu dans la section 3.2.2, les travaux de García et al. (2006) ont montré qu'en pré-calculant le préconditionneur, on pouvait s'attendre à ce qu'il reste une bonne approximation tout au long de la simulation. En revanche, dans la plupart des cas, la cohérence temporelle de la mécanique des objets n'est valable que sur de petites périodes de temps. Si le préconditionneur n'est jamais mis à jour, son utilisation pourrait devenir contestable si la configuration mécanique des objets devient

complètement différente de la configuration de référence. L'efficacité du préconditionneur est alors diminuée, et en en raison du coût d'application, si le préconditionneur n'est pas efficace, il a tendance à ralentir la simulation. Pour éviter ce problème, nous proposons de mettre à jour périodiquement le préconditionneur.

Nous pouvons exprimer le système actuel comme une perturbation des matrices précédentes. Si nous appelons $\Delta\mathbf{A}$ une perturbation du système actuel, on obtient :

$$\mathbf{A}_{t+\Delta t} = \mathbf{A}_t + \Delta\mathbf{A} \tag{3.20}$$

Quand Δt tend vers zéro, $\|\Delta\mathbf{A}\|$ est également proche de zéro. De plus, si à un moment t nous possédons un préconditionneur exact $\mathbf{P}_t^{-1} = \mathbf{A}_t^{-1}$, l'erreur générée par l'application du même préconditionneur après un délai Δt peut-être évaluée par :

$$\mathbf{P}_t^{-1} \cdot \mathbf{A}_{t+\Delta t} = \mathbf{I} + \mathbf{P}_t^{-1} \cdot \Delta\mathbf{A} \tag{3.21}$$

De ces deux constatations, on peut en déduire que si Δt est suffisamment petit, $\mathbf{P}^{-1}(x_t) \cdot \Delta\mathbf{A}$ reste également petit et donc le conditionnement $\kappa(\mathbf{P}_t^{-1} \cdot \mathbf{A}_{t+\Delta t})$ également. Ainsi, après avoir calculé une approximation du système, on peut la réutiliser pour les quelques pas de temps suivant. Pour décider si une mise à jour est utile, il suffit bien souvent de mettre à jour le préconditionneur après un nombre fixe de pas de temps (paramètre que l'utilisateur peut contrôler). Il est cependant possible de détecter les objets immobiles pour ne pas remettre à jour inutilement le preconditionneur en se basant sur le nombre d'itérations maximum du gradient conjugué au-delà duquel la prochaine mise à jour sera déclenchée.

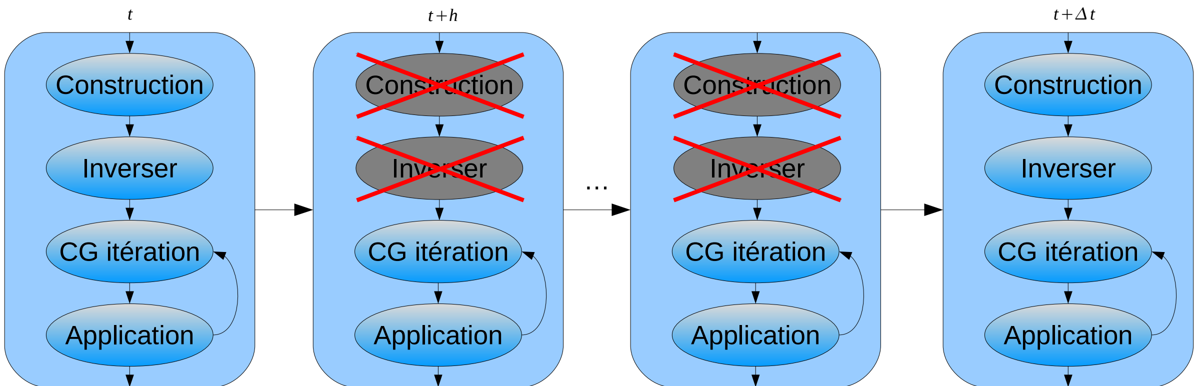


FIG. 3.12 – Composition de plusieurs pas de temps de simulation en mettant à jour périodiquement le préconditionneur. Les pas de temps ne nécessitent pas le même temps de calcul.

À chaque mise à jour il est nécessaire d'assembler l'intégralité ou une partie du système $\mathbf{A}_t$ (ce que nous appellerons l'étape de **Construction**). Ces valeurs sont nécessaires pour calculer explicitement le préconditionneur $\mathbf{P}_t$ (l'étape **Inverser**). Enfin, nous utilisons ce préconditionneur dans l'algorithme du gradient conjugué à chacune de ses itérations (l'étape **application**). Pour les pas de temps où le préconditionneur n'est pas mis à jour, seule l'étape d'**application** est nécessaire car nous gardons simplement le préconditionneur déjà construit précédemment (voir figure 3.12).

La ré-utilisation du préconditionneur permet d'amortir sur plusieurs pas de temps son coût de calcul, et ainsi de diminuer le temps total d'une simulation. Cette technique nous permet d'utiliser des préconditionneurs beaucoup plus complexes comme une factorisation exacte. En revanche, cette démarche produit l'effet indésirable de bloquer périodiquement la simulation. En effet, pour des préconditionneurs précis, l'étape **Inverser** est généralement coûteuse. À chaque exécution, elle peut produire des accoups qui sont très nuisibles pour toute interaction de l'utilisateur.

3.4.2 Préconditionneur asynchrone

Pour contourner cette limitation, nous proposons d'utiliser un deuxième thread CPU qui va réaliser la factorisation du préconditionneur en dehors de la boucle de simulation (voir figure 3.13). En effet, la tâche **Inverser** est bien séparée du reste de la simulation. Elle a besoin uniquement de la matrice du système $\mathbf{A}$ actuel en entrée, et produit en sortie une structure de données qui n'est utilisée que pour appliquer le préconditionneur. Le fait de maintenir l'étape de factorisation sur CPU, permet d'une part d'exploiter ces architectures qui sont en général très efficaces sur ce type de problème. D'autre part, cela permet d'utiliser les nombreuses librairies qui traitent de ce problème. Les librairies les plus optimisées fournissent une nouvelle factorisation en seulement quelques pas de temps de simulation. Certaines d'entre elles proposent même une version parallèle du calcul, ce qui reste compatible avec notre proposition si l'on dispose de plusieurs processeurs.

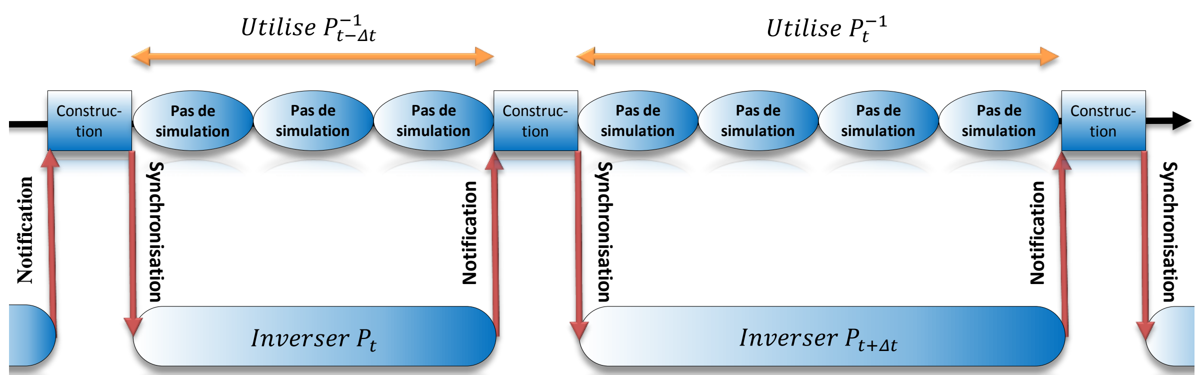


Fig. 3.13 – Préconditionneur calculé en parallèle. La simulation n'est jamais bloquée par le calcul du préconditionneur, pendant que le second thread calcul en permanance de nouvelles approximations.

De cette manière, pendant que le thread de factorisation met à jour le préconditionneur, la simulation continue d'avancer en exploitant potentiellement le GPU. Pour cela, elle utilise une ancienne factorisation comme préconditionneur. Ceci nécessite de dupliquer les structures de données servant à stocker le préconditionneur, mais en utilisant une représentation creuse du type CRS Barrett et al. (1994), la sur-consommation mémoire reste limitée.

Éventuellement, la tâche de **Construction**, peut elle aussi être déplacée dans le second thread, mais ceci n'est utile que si la simulation n'a pas besoin de construire explicitement la matrice. C'est par exemple le cas de notre algorithme du gradient conjugué sur

GPU, présenté dans la section précédente, qui permet de résoudre le système sans jamais assembler **A**. La migration de cette opération dans le second thread n'est cependant pas triviale. Ces changements, impliquent des modifications plus profondes dans le code, et les structures nécessaires à la construction doivent également être dupliquées car elles sont susceptibles d'être modifiées par la simulation.

Cependant, si les deux tâches **Construction** et **Inverser** sont exécutées dans le thread de factorisation, alors tous les pas de temps de la simulation présentent un temps d'exécution approximativement constant. En effet, si une mise à jour est effectuée, les étapes **Construction** et **Inverser** sont remplacées (du point de vue de la simulation) par quelques communications très rapides pour échanger les adresses mémoire de travail des deux threads. Seule l'étape d'**application** reste présente dans tous les pas de temps de la simulation, et c'est le seul surcoût engendré par l'utilisation d'un préconditionneur avec notre méthode. Or, si le préconditionneur est de bonne qualité, peu d'itérations seront nécessaires pour atteindre la convergence du CG, et donc le préconditionneur sera appliqué peu de fois. On constate alors que les préconditionneurs les plus efficaces sont ceux qui fournissent la meilleure approximation de la matrice, c'est-à-dire une factorisation exacte ou incomplète du système.

Dans la mesure où le coût de mise à jour d'un préconditionneur devient négligeable du point de vue de la simulation, on peut définir un nouveau critère de mise à jour, qui est de calculer une nouvelle approximation dès que possible, c'est-à-dire dès que l'inversion précédente est terminée. Cela permet, à la simulation, de toujours utiliser le préconditionneur le plus proche possible de l'état courant. Il est néanmoins importante de veiller à ce que la simulation n'attende jamais après le thread de factorisation.

Dans l'ensemble, notre solution permet de désynchroniser le calcul du préconditionneur de la simulation pour produire de bonnes approximations à un coût négligeable. En revanche, comparé à la version synchrone, le principal inconvénient est que chaque nouvelle mise à jour du préconditionneur fournit la factorisation d'une ancienne configuration des objets. En effet, le calcul du précondionneur n'est pas instantané et son résultat ne sera obtenu que quelques pas de temps plus tard, pendant lesquels la simulation évolue et modifie la matrice du système. Si Δt est la période moyenne de mise à jour en temps de simulation, chaque préconditionneur sera utilisé du temps $t + \Delta t$ jusqu'au temps $t + 2\Delta t$ (voir figure 3.13). Dans la version précédente, il aurait été utilisé du temps t au temps $t + \Delta t$. Ainsi, dans l'approche asynchrone les préconditionneurs peuvent être jusqu'à deux fois plus vieux que dans la version synchrone.

3.4.3 Utilisation des rotations locales

Pendant le temps de mise à jour du préconditionneur, le mouvement des objets virtuels peut inclure des déformations, des translations et des rotations. Dans la plupart des cas, les translations et les petites déformations ne modifient pas sensiblement les valeurs de la matrice **A**, mais en revanche les rotations peuvent détériorer l'efficacité de notre préconditionneur asynchrone. Pour atténuer ce problème, nous proposons d'utiliser une méthode inspirée de la méthode de warping proposée par Saupin et al. (2008b), dans laquelle une matrice de rotation est exprimée au niveau de chaque nœud, puis est utilisée

comme repère local pour calculer la matrice de rigidité (voir figure 2.12). Pour transposer cette idée à notre problème, on va "tourner" le préconditionneur avant chaque application, afin de prendre en compte les non-linéarités géométriques qui sont apparues depuis la dernière mise à jour. Avant chaque application, on estime alors une rotation $\mathbf{R}$, et l'équation (3.13) est remplacée par :

$$\mathbf{R}\mathbf{P}^{-1}\mathbf{R}^T\mathbf{A}\boldsymbol{x} = \mathbf{R}\mathbf{P}^{-1}\mathbf{R}^T\boldsymbol{b} \tag{3.22}$$

Contrairement à la méthode de warping, les rotations ne sont pas définies à partir de la position de repos, mais à partir de la position $\boldsymbol{p}_t$ au temps t qui a été choisie pour calculer le préconditionneur. Les rotations, sont alors beaucoup plus petites, et par conséquent la technique est plus précise. Pour chaque pas de temps, nous évaluons les rotations actuelles $\mathbf{R}_{t+\Delta t}$ à partir de la position au repos, auxquelles nous appliquons l'inverse des rotations $\mathbf{R}_t$ calculées au moment de la dernière mise à jour du préconditionneur (voir figure 3.14). On obtient alors la rotation courante par :

$$\mathbf{R} = \mathbf{R}_t^{-1}\mathbf{R}_{t+\Delta t} \tag{3.23}$$

où $\mathbf{R}$ est la matrice finale utilisée dans l'équation (3.22). Dans la pratique, le produit $\mathbf{R}\mathbf{P}^{-1}\mathbf{R}^T$ n'est jamais calculé explicitement. Au contraire, nous effectuons des produits matrice–vecteur successifs en commençant par le vecteur $\boldsymbol{b}$. Comme, les rotations de tous les nœuds sont stockées dans une matrice diagonale par bloc, ces opérations présentent une faible consommation mémoire, et sont assez rapides à calculer dans la boucle de simulation.

Le warping, inspiré de la formulation FEM co-rotationnelle, n'avait jamais été testé sur d'autres types de modèles. Nous avons testé cette approche pour d'autres modèles élastiques. En effet, les rotations apparaissent quel que soit le modèle de déformation utilisé, et la méthode peut donc être bénéfique pour chacun d'entre eux[8]. La seule difficulté est de parvenir à exprimer les rotations de façon générique.

Pour le modèle co-rotationnel, nous utilisons les rotations calculées pour chaque éléments pendant l'expression de la matrice de rigidité, que nous moyennons au niveau des nœuds. Pour appliquer le préconditionneur, nous devons également calculer $\mathbf{R}$ en fonction de $\mathbf{R}_t$ et $\mathbf{R}_{t+\Delta t}$. Pour les autres modèles, nous devons en supplément exprimer les rotations, car elles ne sont pas forcément nécessaires à la formulation du modèle déformable. Nous utilisons la méthode de shape matching introduite par Müller et al. (2005), où l'estimation des rotations de chaque nœud se fait en fonction du mouvement des nœuds se trouvant à un certain niveau (généralement 1 ou 2) de voisinage. Cette méthode est plus coûteuse que la formulation co-rotationnelle, mais peut-être appliquée à n'importe quel nuage de point, sans aucune restriction sur le maillage ni sur le modèle de déformation.

Cette méthode n'est utile que dans le cas où des rotations sont apparues depuis la dernière mise à jour, ce qui est un phénomène fréquent, mais pas obligatoire. Dans ce cas, l'utilisation des rotations ajoute un surcoût non nécessaire, mais celui-ci reste faible par rapport au calcul des itérations du CG, car le nombre d'opérations est linéaire en fonction

[8]Même si on s'attend à ce que l'efficacité de la méthode soit maximale pour le modèle co-rotationnel, dont les non-linéarités proviennent uniquement des rotations

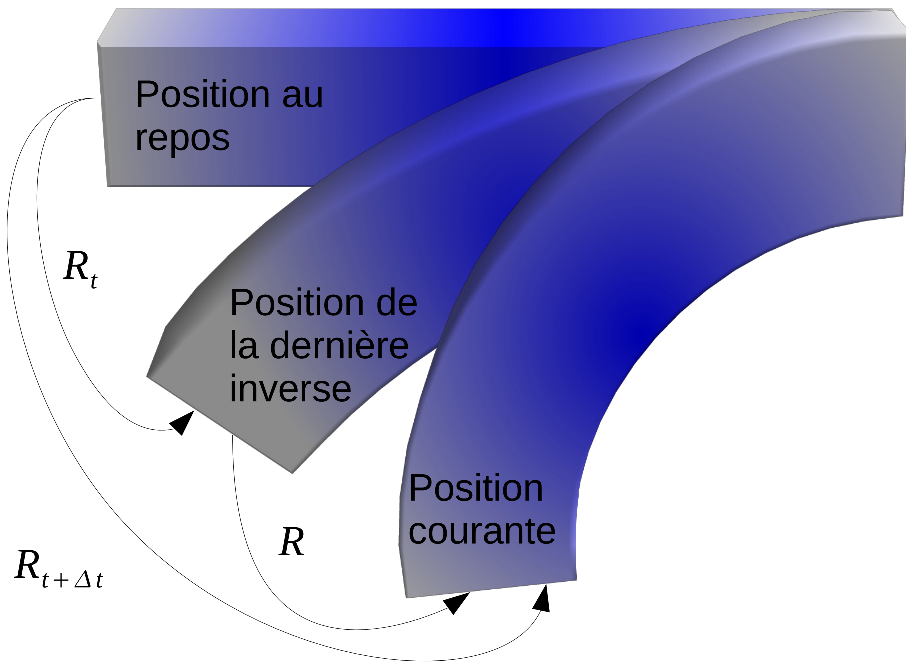


FIG. 3.14 – Nous estimons, pour chaque nœud d'un objet déformable, une rotation à des moments différents, en utilisant la position de repos comme référence. En combinant ces rotations, on obtient la matrice de rotation $\mathbf{R}$ au temps $t + \Delta t$ par rapport à la matrice $\mathbf{R}_t$, calculée au temps t, où le préconditionneur avait été calculé.

du nombre de nœuds. Dans le meilleur des cas, notre solution permet de diminuer de façon significative le nombre d'itérations nécessaire. Ainsi, cette technique peut-être vue comme une heuristique qui permet de limiter dans la majorité des cas, la divergence du préconditionneurs. Ce qui a pour conséquence de diminuer le nombre d'itérations du CG (et d'application du préconditionneur), améliorant les performances globales de la simulation.

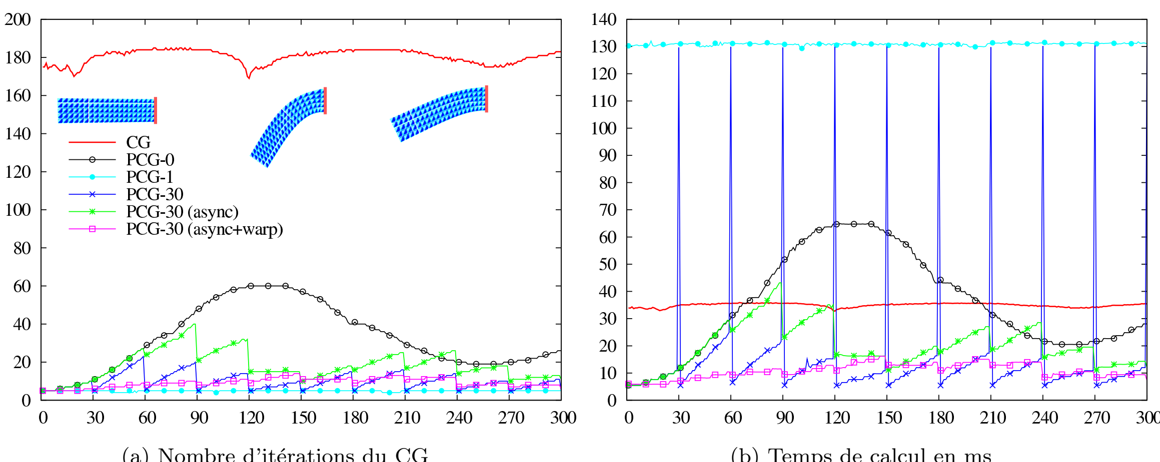
3.4.4 Résultats et discussions

Dans cette section, nous présentons les résultats de notre approche, en mesurant les taux de convergence et le temps de calcul dans différents scénarios. Pour les expériences, nous avons utilisé un processeur Intel quad-core® Core™ i7 à 3.07 GHz, et une Nvidia ® GeForce® GTX 480 programmée avec Cuda 3.0.

Influence des mises à jour du préconditionneur

Nous cherchons tout d'abord à évaluer l'influence des mises à jour du préconditionneur sur un cas simple. Nous avons choisi une poutre déformable homogène (voir figure 3.14) qui est fixée à une extrémité et se déforme sous l'effet de la gravité. Le préconditionneur utilisé est une factorisation de Cholesky incomplète (nous comparerons d'autres préconditionneurs par la suite) que nous mettons à jour à différentes fréquences. L'objet déformable est composé de 3000 éléments tétraédriques, et la fomulation est un modèle FEM co-rotationnel.

La figure 3.15a montre le nombre d'itérations nécessaire pour converger à une tolérence de



(a) Nombre d'itérations du CG

(b) Temps de calcul en ms

FIG. 3.15 – Performances des différents préconditionnements utilisés pour la déformation, sous un modèle co-rotationnel, d'une poutre fixée à une extrémité et déformée par la gravité. Tous les préconditionneurs utilisent une factorisation incomplète de Cholesky, mais les stratégies de mise à jour sont différentes.

l'ordre de 10^{-7}, alors que la figure 3.15b montre les performances associées. Les différentes courbes présentent les résultats de différents préconditionneurs utilisés dans l'algorithme du gradient conjugué préconditionné :

- *CG* : Sans préconditionneur.
- *PCG-0* : Preconditionneur pré-calculé (sans mise à jour).
- *PCG-1* : Préconditionneur mis à jour à chaque pas de temps.
- *PCG-30* : Préconditionneur mis à jour tous les 30 pas de temps (section 3.4.1).
- *PCG-30 (async)* : Préconditionneur mis à jour tous les 30 pas de temps en utilisant un thread parallèle (section 3.4.2).
- *PCG-30 (async+warp)* : Préconditionneur mis à jour dès que possible en utilisant un thread parallèle, et en appliquant les rotations locales (section 3.4.3).

Les courbes *CG* et *PCG-1* correspondent au gradient conjugué avec et sans préconditionneur, et fournissent une base pour comparer toutes les autres méthodes. On constate que tous les préconditionneurs améliorent le taux de convergence du CG, mais pas forcément le temps de calcul. *PCG-1*, permet d'atteindre dans tous les cas, le plus faible nombre d'itérations du CG, mais c'est également la méthode qui présente les plus mauvaises performances. Ceci s'explique par le fait qu'elle nécessite le calcul d'une factorisation incomplète de la matrice à chaque pas de temps, ce qui prend plus de temps que le temps nécessaire au CG non préconditionné pour converger. Il faut rappeler toutefois que le système est bien conditionné, ce qui fait que le gradient conjugué ne nécessite pas un trop grand nombre d'itérations. *PCG-0* offre globalement un taux de convergence raisonnable dans cet exemple, mais il se dégrade rapidement lorsque les déformations sont éloignées de la forme au repos.

PCG-30 correspond à notre première proposition, où nous nous contentons de mettre à jour le préconditionneur tous les 30 pas de temps. On peut voir que ce préconditionneur est bien plus rapide que le CG sur la plupart des pas de temps, sauf quand on met à jour le préconditionneur, où les performances sont égales au *PCG-1*, et on constate l'apparition

de pics dans les temps de calcul. Entre les pas de temps, la qualité du préconditionneur se dégrade légérement, mais reste bien meilleure que la version non préconditionnée. Bien que le coût total de ce préconditionneur soit meilleur que le *CG*, les blocages à répétition de la simulation ne permettent pas de l'utiliser dans les applications interactives.

Lorsque l'inversion est calculée dans un second thread (*PCG-30 async*), le calcul de la factorisation n'affecte pas le temps de calcul de la simulation, et supprime les blocages périodiques. En revanche, le préconditionneur est en général de moins bonne qualité, et un plus grand nombre d'itérations est nécessaire pour atteindre la convergence. Cet inconvénient est atténué dès lors que l'on applique les rotations (*PCG-30 async + warp*), qui limitent la dégradation du préconditionneur entre les mises à jour. Alors, on obtient un nombre d'itérations du même ordre que *PCG-1* à chaque pas de temps de la simulation, alors que le temps de calcul est très faible.

Fait intéressant, on peut voir que sur les 30 premiers pas de temps, *PCG-0*, *PCG-30* et *PCG-30 async* présentent exactement les mêmes résultats. Par la suite *PCG-0* se dégrade continuellement, alors que *PCG-30* peut prendre en compte instantanément la mise à jour du préconditionneur. *PCG-30 async* devra attendre la prochaine mise à jour car au trentième pas de temps, le second thread aura fini d'inverser le système initial. Ceci met en évidence l'importance des mises à jour du préconditionneur.

Éléments	Temps (ms) pour factoriser	CG itérations	Temps (ms) pour chaque pas de temps	Nombre de pas de temps pour factoriser
540	5.84	4.40	3.32	1.76
3000	38.47	5.01	12.78	3.01
7350	125.9	5.56	41.68	3.02
24000	730.5	6.09	208.72	3.50

Tab. 3.1 – Performances obtenues avec différentes tailles de maillage.

Nous avons également mesuré l'influence de la taille de la maille du système sur les performances (voir tableau 3.1). Pour cela, nous utilisons un préconditionneur basé sur une factorisation de Cholesky calculée à l'aide de la librairie Pardiso Schenk et al. (2006). Cette librairie permet d'effectuer la factorisation de la matrice en parallèle, nous avons donc utilisé 4 threads CPU pour effectuer cette tâche. Le temps pris pour factoriser la matrice augmente avec le nombre d'éléments, mais représente toujours un petit nombre d'étapes de la simulation (moins de 4 pas de temps). Ceci s'explique par le fait que chaque itération du gradient conjugué préconditionné est également de plus en plus coûteuse à mesure que la taille du système grandit. Ce résultat nous fait dire que cette méthode est bénéfique aussi bien pour de petits ou des grands objets, car la fréquence de mise à jour reste très basse. Bien sûr, à partir d'une certaine taille, la simulation n'est cependant plus en temps réel.

Application à différents modèles de déformation

Dans cette section nous cherchons à évaluer l'influence de notre solution appliquée à différents modèles de déformation. Nous avons modélisé une barre déformable que nous avons sollicité de différentes manières afin d'introduire soit des non-linéarités dues au matériau, soit des non-linéarités géométriques, soit les deux (voir tableau 3.2). Dans les

résultats, nous comparons quatre façons d'appliquer le préconditionneur : avec et sans les rotations autour d'un préconditionneur remit à jour ou non. Les modèles de déformations que nous avons utilisé sont le modèle masse-ressort, le modèle FEM co-rotationnel, et le modèle non linéaire St-Venant-Kirchhoff implémenté avec l'algorithme MJED Marchesseau et al. (2010).

		Sans Pre-conditionneur		Préconditionneur sans mise à jour				Préconditionneur avec mise à jour asynchrone			
Scénario	Modèle	Sans Rotations		Sans Rotations		Avec Rotations		Sans Rotations		Avec Rotations	
		Iter.	Temps	Iter.	Temps	Iter.	Temps	Iter.	Temps	Iter.	Temps
	Corot.	301.52	138.12	18.65	38.33	13.77	30.10	9.87	**20.38**	**9.19**	20.46
	MJED	198.11	29.34	17.40	31.63	18.56	34.41	5.50	**11.17**	**5.32**	12.34
	Spring	167.54	24.48	22.53	32.27	22.36	32.82	7.13	**13.29**	**6.88**	14.40
	Corot.	301.98	138.08	33.55	69.46	**9.14**	**20.08**	10.59	21.73	9.58	21.13
	MJED	217.50	31.59	19.25	34.72	14.42	27.00	8.27	**16.47**	**7.79**	17.06
	Spring	193.87	27.74	26.14	36.25	14.23	21.11	9.27	**17.31**	**8.68**	17.93
	Corot.	303.99	142.99	122.04	262.16	18.91	41.38	13.33	28.67	**12.05**	**26.93**
	MJED	220.26	35.38	48.42	86.38	36.79	69.24	10.77	**21.25**	**10.53**	22.58
	Spring	174.16	27.40	83.96	118.24	36.84	56.17	**7.32**	**13.79**	11.39	23.50

Tab. 3.2 – Influence des rotations sur différents modèles de déformation en fonction de différentes sollicitations. Le nombre d'itérations représente les valeurs moyennes par pas de temps. Les poutres déformables sont composées de 3000 éléments tétraédriques homogènes.

En utilisant notre préconditionneur asynchrone, nous augmentons significativement les performances de la simulation. En effet, la méthode permet de diviser le nombre d'itérations jusqu'à un facteur $32\times$ sur le modèle co-rotationnel, $37\times$ pour le modèle MJED et $24\times$ pour le masse-ressorts. Cependant, comme chaque itération de la version préconditionnée est plus coûteuse que la version non préconditionnée, on constate une accélération maximale de $6,7\times$ sur le modèle co-rotationnel, $2,3\times$ sur le modèle MJED et $1,7\times$ pour le modèle masse-ressorts.

Une mise à jour régulière du préconditionneur permet d'améliorer, pour tous les modèles, les performances par rapport à la version non préconditionnée. Sur cet exemple nous obtenions un nouveau préconditionneur toutes les $100ms$ en moyenne, ce qui correspond à 3 ou 4 pas de temps de la simulation. Ainsi, très peu de rotations étaient introduites entre les mises à jour, et leur utilisation n'apporte pas de gain significatif en terme d'itérations (mais le surcoût en terme de temps de calcul est également très faible). Or, si le préconditionneur n'est pas mis à jour les différentes sollicitations peuvent modifier complètement la configuration de l'objet par rapport à la version initiale. L'utilisation d'un tel préconditionneur sans rotation permet en général d'améliorer les performances pour le modèle co-rotationnel, mais diminue celles des autres modèles. L'activation du warping, permet alors d'obtenir malgré tout de bonnes performances même pour les modèles autres que le co-rotationnel. Ceci montre que si de grandes rotations sont introduites (par

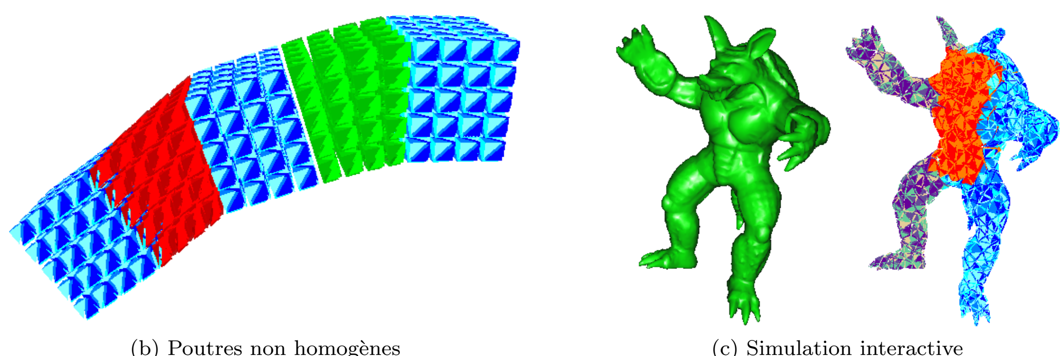
exemple si le préconditionneur nécessite un grand nombre de pas de temps pour être mis à jour), l'utilisation de la méthode de warping peut être d'une grande aide pour maintenir une bonne approximation. En définitive, le wraping seul n'est pas suffisant, et une mise à jour périodique du préconditionneur permet d'obtenir les meilleures performances.

Évaluation des différents préconditionneurs

Nous proposons d'évaluer la convergence en utilisant différents préconditionneurs sur des problèmes non homogènes. Ces problèmes sont connus pour être difficiles à résoudre par un CG traditionnel. Nous présentons l'évaluation des performances dans le tableau 3.16a. Les premiers (standard) préconditionneurs sont mis à jour à chaque pas de temps de la simulation alors que les versions (async + warp) sont factorisées dans un autre thread.

	Préconditionneur	CG itérations	Computation time (ms)		
			Inversion	Résolution	Total
Standard	– (CG)	654.03	–	126.44	126.44
	Jacobi	356.96	.15	86.42	86.58
	SSOR	116.72	30.60	55.61	86.21
	Cholesky	1.00	39.27	3.13	42.40
	IC	6.07	107.24	7.29	114.54
async + warp	Jacobi	357.99	.16	93.18	93.34
	SSOR	116.76	.26	62.70	62.97
	Cholesky	6.02	.06	11.92	11.98
	IC	8.71	.03	10.76	10.79

(a) Mesures de performances sur une poutre non homogène



(b) Poutres non homogènes

(c) Simulation interactive

FIG. 3.16 – (a) Évaluation des performances des différents préconditionneurs sur la poutre présentée en 3.16b. (b) Poutre non homogène composée de 3000 éléments, déformée sous l'effet de la gravité. Les différentes couleurs montrent les différents modules Young (1000, 50 et 10 pour respectivement les éléments bleus, verts et rouges). (c) Exemple Interactif composé de 4406 éléments non homogènes.

On peut tout d'abord remarquer que le CG a beaucoup de mal à converger seul sur ce type de problème, et même le plus simple des préconditionneurs comme le Jacobi permet d'améliorer sensiblement la convergence. Sur cet exemple, même la factorisation

incomplète de Cholesky IC mise à jour à chaque pas de temps fournit une accélération d'environ 10 % (alors qu'elle était de 4× plus lente que le CG sur la simulation homogène de la figure 3.15) . Pour la factorisation de Cholesky exacte, nous utilisons un algorithme parallèle optimisé basé sur un calcul de super-nœuds fourni par *Taucs* Toledo et al. (2003), tandis que cette optimisation n'est pas disponible pour la factorisation incomplète (ce qui explique pourquoi il est beaucoup plus long à factoriser). Le préconditionneur cholesky ne nécessite qu'une seule application puisqu'il fournit directement la solution au CG. Celui-ci permet ainsi d'obtenir une accélération d'environ 3× par rapport au CG[9].

Les préconditionneurs Jacobi et SSOR fournissent une accélération d'environ 40%. Toutefois, SSOR ne nécessite aucun calcul préalable pour inverser la matrice, et le temps d'inversion correspond uniquement aux temps de construction de la matrice. Cette opération représente 30% de l'application du préconditionneur SSOR. Notre méthode nous permet d'éviter le coût de cette construction dans la boucle de simulation, sans diminution significative de la qualité du préconditionneur. Le préconditionneur Jacobi est le seul qui ne bénéficie pas de notre méthode, car sa matrice est trop simple à construire et à inverser. Avec notre méthode, les factorisations complètes et incomplètes sont les préconditionneurs les plus efficaces. En effet, leur manque d'efficacité dans les approches traditionnelles était principalement dû au coût de factorisation, qui est complètement caché par notre méthode. Sur l'exemple non-homogène, le CG préconditionné converge en moins de 10 itérations, ce qui correspond approximativement aux mêmes résultats que les exemples présentés précédemment. Ainsi, notre méthode permet de simuler aussi bien des objets homogènes et hétérogènes avec le même niveau de précision.

Pour finir, nous avons également appliqué notre méthode à un cas interactif visible sur la figure. 3.16c. Nous avons modélisé un armadillo déformable avec des raideurs non-homogènes, que l'utilisateur peut déplacer et faire pivoter de façon interactive. Une version non-préconditionnée ne permet pas de simuler l'objet assez rapidement car le CG ne converge pas dans un délai raisonnable. Avec notre méthode nous avons été en mesure de maintenir un rafraîchissement des images entre 50 et 80 FPS, sans accoups dans la simulation.

3.5 Conclusion

On a présenté dans ce chapitre un ensemble de contributions qui permet d'obtenir efficacement, la solution d'un système d'équations linéaires. Pour cela, nous avons proposé une parallélisation GPU de l'intégrateur implicite, qui comprend notamment l'algorithme du gradient conjugué sur GPU. Avec notre méthode, nous avons montré une accélération significative par rapport à une méthode séquentielle, spécialement quand les maillages sont très détaillés. Or, on a vu dans le chapitre précédent, que l'utilisation de petits éléments conduit à une meilleure approximation de la méthode des éléments finis. Notre solution permet ainsi d'augmenter la qualité de la déformation, tout en restant compatible avec le temps réel.

D'autre part, nous avons proposé une technique de préconditionnement, qui peut être couplée avec l'algorithme du gradient conjugué sur GPU. Le calcul du préconditionneur

[9] Notons que ce temps de calcul correspond approximativement à ce qui serait obtenu si nous avions utilisé cette factorisation comme solveur direct.

repose sur la cohérence temporelle de la mécanique des objets, et sur une heuristique pour limiter l'influence des rotations. L'idée de base est de factoriser la matrice de façon exacte dans un second thread, de telle sorte que le coût de cette opération soit négligeable dans le calcul de la simulation. On récupère alors peu de temps après une factorisation plus ancienne de quelque pas de temps, mais qui reste une très bonne approximation. La représentation creuse de la matrice permet également de maintenir un nombre important d'éléments en temps réel (ce nombre est malgré tout plus petit que pour la version non préconditionnée sur GPU). Avec cette méthode, on arrive ainsi à garantir un nombre très faible d'itérations pour atteindre la convergence du gradient conjugué. Le préconditionneur est particulièrement utile dans le cas où le CG a des difficultés pour converger, par exemple sur des maillages non homogènes, ou si l'objet présente un mauvais rapport masse/raideur. Son utilisation permet alors de simuler avec une grande précision n'importe quel objet sans se soucier du conditionnement du système.

Toutes les méthodes présentées dans ce chapitre peuvent s'adapter aussi bien au modèle corotationnel, qu'aux autres modèles déformables (même si leur efficacité peut être variable). En définitive, nous sommes maintenant capables de simuler la déformation d'objets très détaillés en temps réel, avec une grande précision. Cependant, ce résultat n'est pas suffisant pour produire un simulateur médical, car on ne peut pas interagir avec les objets. C'est pourquoi, le reste de ce document est concentré sur cet aspect.

Chapitre 4

Détection et formulation des contacts

Table des matières

4.1 Introduction **91**
4.2 Les méthodes de détections et réponses aux contacts **91**
4.2.1 Détection des collisions et intégration 92
Détection continue ou discrète 92
Détection et intégration 92
4.2.2 Méthodes de détections des collisions 93
Hiérarchie de volume englobant 93
Autres méthodes 94
Méthodes basées sur le matériel graphique 95
4.2.3 Équations du contact 96
Lois de contacts 96
Méthodes particulaires 98
Réponse par contraintes 99
4.2.4 Réponses par pénalités basées sur le volume d'intersection 101
4.3 Détection de collision GPU **103**
4.3.1 Construction des LDI 103
Étapes préliminaires 104
Rastérisation des triangles 107
4.3.2 Utilisation des LDI 109
Détection des intersections 109
Transparence et rendu volumique 111
4.3.3 Résultats 113
4.4 Extension aux contacts volumiques **114**
4.4.1 Contraintes basées sur des volumes 115
Équivalence entre distance et volume 115
4.4.2 Extension aux contacts frottants 117
4.4.3 Extension au modèle Multi-Volumes 118
4.4.4 Résultats 120
Évaluation de la méthode 120
Mesure des performances 121
4.5 Conclusion **125**

4.1 Introduction

Dans les chapitres précédents, nous avons vu comment simuler la déformation d'un objet lorsqu'il est soumis à des forces externes ou internes. Bien que ceci nous permet d'animer de façon physiquement réaliste les objets virtuels, la gestion des interactions mécaniques (que ce soit entre les organes, et éventuellement avec les outils chirurgicaux) est aussi nécessaire. Cela ajoute plusieurs difficultés importantes, qui sont la détection des collisions, la mise en équation, et le calcul des forces de réponse.

La détection des collisions est un problème complexe en soi, qui implique de déterminer les primitives en intersection, ainsi que la profondeur et la direction de la réponse. La discrétisation du temps, et la nature imprévisible des contacts (en particulier si l'utilisateur interagit dans la boucle de simulation) empêchent bien souvent les pré-calculs. Ces contraintes font de l'étape de détection des collisions une tâche très difficile, surtout avec des objets déformables.

Une fois les intersections trouvées, la formulation du problème de contact n'est pas non plus une chose facile. Il est important de respecter un ensemble de lois physiques du contact (et du frottement) pour obtenir un comportement réaliste dans la simulation. Une formulation rigoureuse est alors nécessaire pour simuler des phénomènes complexes, comme par exemple la saisie d'un objet. La modélisation de la découpe (avec les changements topologiques associés) est également problématique, mais indispensable dans un grand nombre d'opérations chirurgicales.

Enfin le calcul de la force de réponse s'avère être un problème difficile également. En raison du couplage mécanique entre les contacts, cette tâche revient souvent à résoudre un problème combinatoire complexe, car en même temps que d'évaluer la force sur les points de contacts nous devons déterminer si les contraintes sont actives ou non. De plus, des contacts simultanés peuvent conduire à résoudre des contraintes incompatibles, ce qui entraîne des problèmes numériques.

La suite de ce document est consacrée à l'étude et à la description des composants permettant de simuler les interactions entre objets virtuels. Dans ce chapitre nous traiterons uniquement des aspects touchant directement à la détection des collisions, et à la mise en équation. Le chapitre suivant sera consacré à leur résolution.

4.2 Les méthodes de détections et réponses aux contacts

Nous avons la faculté de détecter visuellement et très rapidement une interpénétration entre les objets. En revanche, écrire un algorithme qui réalise cette tâche et qui soit précis, robuste et assez rapide pour être exécuté en temps réel n'est pas une chose facile. La géométrie des objets est souvent décrite par un ensemble de positions dans l'espace reliées par des polygones. Or, pour savoir si un objet est en collision avec un autre, il faut déterminer si au moins un des points de son volume est à l'intérieur des autres objets. Ce test nécessite tout d'abord de définir l'intérieur et l'extérieur des objets, mais surtout la recherche présente une complexité en $O(n^2)$ (où n est le nombre de primitives, ce qui est potentiellement très grand). Il est donc important d'utiliser des méthodes adaptées pour limiter le temps de calcul.

4.2.1 Détection des collisions et intégration

En règle générale, les interactions peuvent intervenir à tout moment, et entre n'importe quelle paire d'objet, ce qui fait que l'instant exact du contact est difficile à déterminer avec précision. De plus, les collisions vont provoquer des changements brutaux dans la vitesse des particules, ce qui implique qu'au moment du contact, l'accélération n'est pas définie. Ceci amène à traiter des problèmes de la mécanique non-régulière, qui reste aujourd'hui un sujet de recherche a part entière. Nous nous intéressons ici aux méthodes de détection des contacts et à la formulation de leurs équations dans le contexte des simulations en temps réel.

Détection continue ou discrète

Il existe deux approches pour détecter les collisions entre les objets, la *détection continue* et la *détection discrète*. La détection continue consiste à extrapoler le mouvement entre les pas de temps Larsson et Akenine-Moller (2001) ; Tang et al. (2008). Ces méthodes sont généralement précises, et permettent de trouver l'ensemble des contacts qui apparaissent entre les objets, ainsi que l'instant exact de l'impact. Même si on trouve certains travaux dans ce sens Tang et al. (2008); Jund et al. (2010), elles sont généralement coûteuses et sont difficilement adaptables aux objets déformable en temps réel.

À l'opposé, les méthodes de détection discrètes réalisent une détection des collisions par rapport à la configuration actuelle des objets. Ces méthodes sont donc beaucoup plus rapides, puisqu'il suffit de détecter les collisions une seule fois par pas de temps. Or, comme de nombreuses collisions peuvent intervenir pendant une petite période de temps, ces méthodes représentent un bon compromis entre la précision et la durée du temps de calcul. Cependant, ce type de détection peut "rater" des événements, car aucune détection n'est réalisée entre les pas de temps. Si par exemple de petits objets s'approchent très vite l'un de l'autre, il est tout à fait possible qu'ils ne soient pas en collision au moment des détections, malgré le fait qu'ils se soient croisés pendant leur mouvement. Ainsi, ces méthodes imposent généralement des restrictions sur la taille du pas de temps, ou sur la taille des objets.

Détection et intégration

Une fois les collisions détectées, il faut ajouter ces informations dans le système d'équations dynamique. Les méthodes *event-driven* consistent à arrêter l'intégrateur au moment précis de l'impact Baraff (1994). On pourra alors redéfinir l'ensemble des paramètres (comme la vitesse) pour relancer la simulation avec de nouvelles conditions post-contact. De cette façon on peut traiter l'intégralité des événements dans leur ordre d'apparition, et on obtient généralement une simulation de grande précision. Cependant, ces méthodes nécessitent de déterminer l'instant exact du contact avec une détection des collisions continues. Par ailleurs, le fait de stopper l'intégrateur à de multiples reprises, peut avoir une conséquence importante sur les performances. En effet, de nombreux contacts peuvent apparaître dans de très petits intervalles de temps, ce qui nécessite d'arrêter l'intégrateur plusieurs fois par pas de temps. Des solutions alternatives doivent également être mises en place, pour gérer le cas d'un contact continu entre des objets Mirtich et Canny (1995).

Les méthodes de *time-stepping*, vont quand à elles détecter l'ensemble des collisions à la fin de chaque pas de temps Anitescu et al. (1999). On peut alors choisir d'utiliser soit une détection continue en extrapolant le mouvement entre les détections, soit une détection discrète. Cette deuxième solution est généralement plus rapide, mais complexifie la tâche de la réponse. En effet, avec une détection discrète on se retrouve souvent à devoir traiter de multiples interpénétrations qui peuvent être plus ou moins profondes, et il est difficile de produire une réponse adaptée car les lois de la physique interdisent clairement toute interpénétration entre les objets. Dans la suite de cette section, nous présenterons un ensemble de méthodes qui permettent de produire une réponse qui respectent un ensemble de lois physiques.

4.2.2 Méthodes de détections des collisions

Nous allons maintenant présenter les principales familles de méthodes de détections de collisions. Comme la recherche des intersections est coûteuse, on effectue généralement une première étape appelée *broad-phase*. Elle consiste à éliminer avec des tests simples, les paires d'objets dont on est sûr qu'ils ne peuvent pas entrer en collision (par exemple en évaluant la distance entre les boîtes englobantes des objets). Pour toutes les paires d'objets potentiellement en collision, on réalise ensuite la détection des collisions à proprement parler. Cette étape, est elle même découpée en deux sous-étapes, la première appelée *narrow-phase*, qui détermine un ensemble de primitive proche de l'intersection. Et enfin l'*exact-phase* qui calcule les primitives effectivement en collision, et détermine la distance d'interpénétration.

De nombreuses difficultés sont ajoutées dans le cas d'une détection entre objet déformable, car la méthode de détection de collision utilisée doit pouvoir s'appliquer à n'importe quelle forme géométrique. En effet, comme on ne peut pas garantir la forme des objets, il faudra être en mesure de détecter les auto-collisions (collision d'un objet avec lui-même). Une quantité impressionnante de travaux a été menée sur ce sujet, et un excellent aperçu peut être trouvé dans Teschner et al. (2005)

Hiérarchie de volume englobant

L'une des méthodes les plus utilisées repose sur une hiérarchie de volumes englobants, mieux connue sous le nom de *bounding-volume hierarchies* (BVH). L'idée est de répartir les primitives constituants les objets (triangles, lignes, polygones, ...) à l'intérieur de formes géométriques simples, pour récursivement en construire une hiérarchie. En rassemblant de bas en haut ou en subdivisant de haut en bas les volumes, on obtient un arbre avec comme racine, un volume englobant toutes les primitives de l'objet. On peut alors exploiter cette structure pour déterminer efficacement les primitives en intersection.

Il existe deux approches pour choisir les formes géométriques servant à englober les primitives. La première est de minimiser le coût de calcul de chaque test d'intersection. Les sphères sont par exemple très rapides Hubbard (1995) puisqu'il suffit de tester la distance par rapport à leur centre. Cependant, le volume des primitives est en général mal approximé, et de nombreuses "zones vides" vont passer le test avec succès. L'autre approche consiste à minimiser le volume englobant. On cherche alors à utiliser des primitives très proches de la surface réelle, tout en gardant un test d'intersection rapide. De nombreuses

déclinaisons existent allant des AABB, OBB Gottschalk et al. (1996), k-DOPS... (voir Teschner et al. (2005))

Ces méthodes sont robustes et peuvent éventuellement gérer l'auto-collision Volino et Magnenat-Thalmann (1995); Provot (1997). Mais, dans un contexte déformable la principale difficulté consiste à mettre à jour la hiérarchie de volume. En effet, les objets pouvant se déformer, la structure de l'arbre doit également être mise à jour en conséquence. On trouve dans la littérature plusieurs solutions pour mettre à jour la hiérarchie van den Bergen (1997); James et Pai (2004), mais ces méthodes restent souvent limitées par le nombre de triangles consistants les objets. Cependant, ces méthodes sont très largement utilisées dans les simulations d'objets déformables ou rigides.

Autres méthodes

Subdivision spatiale Alors que les BVH construisent une hiérarchie axée sur les objets, d'autres méthodes cherchent à subdiviser l'espace pour y ranger leurs primitives. Par exemple, Teschner et al. (2003) proposent de subdiviser l'espace en petites cellules, pour y ranger les primitives à l'aide d'une fonction de hachage. Ces méthodes permettent de gérer facilement les objets déformables, ainsi que les auto-collisions, puisqu'elles sont indépendantes de la forme des objets. De plus, de nombreuses optimisations peuvent être apportées pour raffiner les zones de l'espace occupé. La principale difficulté réside dans le choix des structures de données, ainsi que dans le coût des fonctions de tests associées. En effet, seule une subdivision très fine permet d'obtenir une détection précise. Or on comprend facilement que cela introduit des limitations dans le stockage des informations, et multiplie les collisions dans les fonctions de hachage.

Méthodes stochastiques Les méthodes stochastiques reposent sur deux observations. Tout d'abord, les modèles polygonaux de la surface sont juste une approximation de géométrie. De plus, la perception de la qualité de détection dépend plutôt du temps de réponse que de l'exactitude de la détection. De ces deux postulats, certaines méthodes ont cherché à créer une détection de collisions qui soit très rapide, quitte à ce qu'elle ne soit pas exacte Lin et Canny (1992); Klein et Zachmann (2003); Joussemet et al. (2006). Deux approches différentes ont été proposées. La première utilise les probabilités pour estimer la possibilité d'une collision, en fonction du nombre de primitives trouvées dans la narrow-phase. L'idée est simplement de se dire que plus il y a de primitives dans une région donnée, plus il y a de chance d'avoir une collision. La seconde approche exploite la cohérence temporelle des simulations. Un nombre fixe de primitives est testé au hasard à chaque pas de temps dans des régions différentes. Quand une collision est détectée, elle est suivie et traitée pour la réponse. Évidemment ces méthodes présentent des temps de calcul très courts, mais elles ne sont applicables qu'aux cas où les contacts apparaissent lentement, et agissent pendant plusieurs pas de temps.

Champ de distance Les champs de distance peuvent être vus comme une fonction prenant une position de l'espace en paramètre, qui retourne la distance minimale par rapport à la surface Fuhrmann et al. (2003). Pour les objets rigides, ces méthodes sont très efficaces puisqu'on peut pré-calculer un échantillonnage de points, dans une grille autour de l'objet. Pendant la simulation, il suffit alors d'accéder à cette structure de données, afin de

déterminer (à très faible coût) la distance entre les objets. D'autres approches consistent à exprimer la surface des objets comme un ensemble de fonctions implicites, qui peuvent évoluer avec la forme des corps déformables. Évidemment, la difficulté consiste à trouver les fonctions qui fournissent une bonne approximation de la surface, et à les mettre à jour en fonction des déformations Frisken et al. (2000).

Méthodes basées sur le matériel graphique

Les méthodes basées sur le GPU permettent principalement d'améliorer le rendu de la simulation. Par exemple, la méthode de *depth peeling*, consiste à trier les triangles des objets dans l'ordre de profondeur, pour obtenir un effet de transparence. Cependant, plusieurs travaux ont cherché à déduire des informations utiles, à partir des calculs graphiques. Certaines méthodes exploitent le *Z-BUFFER* (qui sert à connaître la profondeur des objets affichés à l'écran), pour estimer la distance entre deux objets Shinya et Forgue (1991). D'autres ont exploité le GPU pour déterminer à très faible coût l'ensemble des objets en intersection Govindaraju et al. (2003).

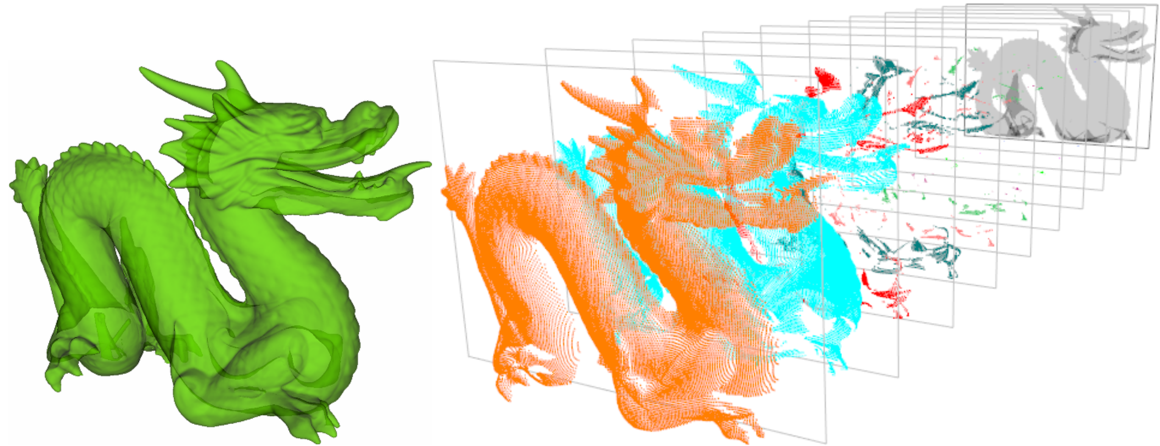


Fig. 4.1 – Un Layered Depth Image (LDI), représenté ici comme une série d'images 2D.

Plus récemment, d'autres méthodes ont exploité le GPU pour calculer le volume d'intersection entre les objets. Le volume d'intersection est calculé sur le GPU en utilisant une nouvelle structure de données appelée *Layer Depth Images* (LDI) Heidelberger et al. (2003, 2004). Les LDI sont généralement représentés par une pile d'images $2D$, dont chaque couche représente les différentes surfaces des objets. Le nombre d'images $2D$ dépend alors du nombre de couches sous un même pixel, dans le sens d'observation choisi (voir figure 4.1). La taille des pixels est alors un paramètre que l'utilisateur peut contrôler pour régler facilement la précision de la méthode. Durant ce travail de thèse, nous nous sommes particulièrement intéressé à ces méthodes, et nous proposerons des solutions pour construire et exploiter efficacement des LDI.

À chaque pixel d'un LDI est associé un ensemble de *layers*, qui sont en réalité la liste des intersections de la surface des objets sous ce pixel[1]. Ainsi, si on imagine un rayon (associé

[1]En règle générale, les layers stockent également des données supplémentaires telles que l'orientation du triangle ou la profondeur du triangle.

à un pixel) parallèle à la direction du LDI, un layer est défini à chaque fois que le rayon "entre" ou "sort" d'un objet. En triant les layers selon la profondeur, on obtient la liste ordonnée (dans la direction du LDI) des intersections entre les objets Heidelberger et al. (2003). En effet, le long de chaque rayon, le volume des objets est représenté par un ou plusieurs intervalles de profondeur, entre les points d'entrées et de sorties. La détection des collisions est simplement basée sur des intersections d'intervalles.

Pour déterminer l'ensemble des layers, Heidelberger et al. (2004) utilisent l'algorithme de *rastérisation* fourni sur toutes les cartes graphiques récentes. La rastérisation est le procédé qui consiste à transformer l'ensemble des positions définissant les sommets des triangles, en une image qui est projetée sur l'écran. La profondeur des triangles associés peut être obtenue à travers le Z-BUFFER, et il suffit de masquer successivement les premiers triangles affichés à l'écran pour construire les LDI. Le nombre de rendus nécessaires est cependant égal au nombre de couches des surfaces dans la direction choisie.

4.2.3 Équations du contact

Nous commençons par présenter un ensemble de lois physiques qu'il est important de vérifier pour produire une réponse aux collisions réalistes. Nous listerons ensuite les grandes familles de méthodes que l'on peut trouver dans la littérature, pour simuler ces phénomènes.

Contrainte unilatérale et bilatérale Les contraintes qui apparaissent pendant la simulation, ne sont pas toujours de la même nature. En fonction des simulations, on peut avoir besoin de simuler des contacts, ou éventuellement des attaches mécaniques (pour fixer ou attacher des objets). D'une manière générale, les lois de contacts peuvent être décrites avec la relation[2] suivante :

$$\psi(\boldsymbol{x}_1, \boldsymbol{x}_2) \geq 0 \tag{4.1}$$

Où ψ représente les *contraintes unilatérales* (égalité d'un seul côté de l'équation[3]) avec lesquelles on pourra modéliser des contacts, ou de la friction.

Lois de contacts

On présente maintenant un ensemble de lois qu'il est important de respecter pour produire une simulation réaliste.

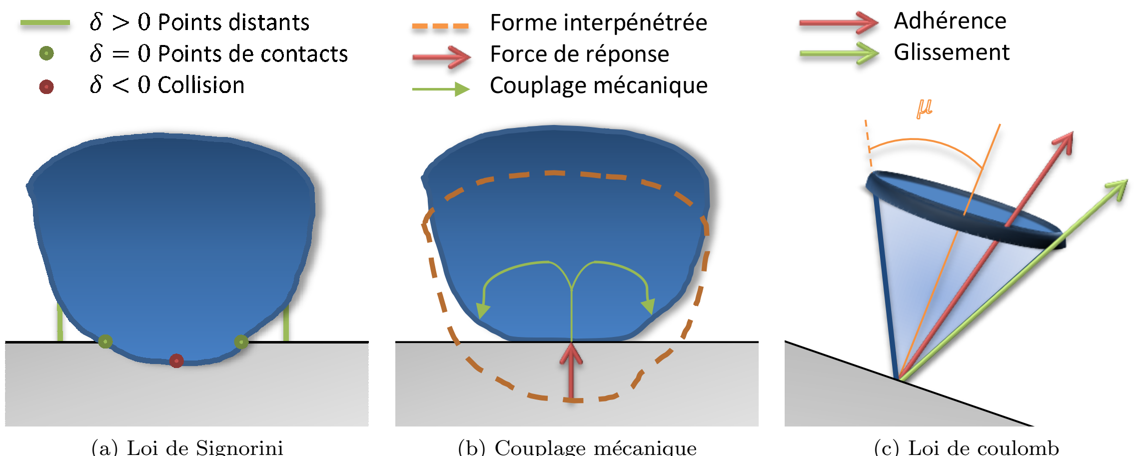
Loi de Signorini La loi de Signorini est la plus évidente puisqu'elle consiste à interdire que les objets s'intersectent mutuellement (voir figure 4.2a). On la formalise souvent par l'équation suivante :

$$0 \leq \boldsymbol{\delta} \perp \boldsymbol{f} \geq 0 \tag{4.2}$$

L'opérateur $\perp$ décrit l'orthogonalité entre la force de réponse $\boldsymbol{f}$, et la distance d'interpénétration $\boldsymbol{\delta}$. Cela signifie que si la distance entre $\boldsymbol{\delta}$ est positive, alors aucune force

[2]Pour plus de simplicité, nous présentons les équations pour deux objets interagissant 1 et 2, mais la méthode s'applique à n'importe quel nombre d'organismes qui interagissent.

[3]Par opposition il existe aussi des contraintes bilatérales (égalité des deux côtés de l'équation), avec lesquelles on va simuler des liaisons cinématiques complètes (pivot, glissière...).



(a) Loi de Signorini (b) Couplage mécanique (c) Loi de coulomb

FIG. 4.2 – Illustration des phénomènes physiques qu'on souhaite simuler pour la réponse aux contacts.

ne doit être appliquée en réponse (car les objets ne sont pas en contact). Si au contraire l'interpénétration est inférieure à zéro (interpénétration), alors il est impératif d'appliquer une force pour ramener l'interpénétration au point de contact ($\boldsymbol{\delta} = 0$).

Loi de Coulomb La loi de Coulomb permet de décrire le frottement qui agit entre deux objets au moment du contact. Cette loi est souvent schématisée par un cône, appelé le cône de frottement (voir figure 4.2c). Si le vecteur de force en réponse au contact reste à l'intérieur du cône de frottement, alors on dit qu'il y a adhérence. Si au contraire, la force est située sur la surface du cône, alors on est en présence d'un contact glissant et on aura une vitesse relative non-nulle selon la tangente au point de contact entre les deux objets. L'angle d'ouverture μ est caractérisé par un scalaire qui sera un paramètre de la simulation, il permet de régler les degrés d'adhérence des surfaces.

Le frottement de Coulomb nécessite de gérer des forces supplémentaires tangentielles au point de contact. Si $\boldsymbol{f}_{\vec{t_1}}$ est la projection de la force selon la tangente $\vec{t_1}$ (qui est orthogonale à la normale du contact $\vec{n}$), on peut détermine l'état de la contrainte en fonction des cas suivants :
- Si $\|\boldsymbol{f}_{\vec{t_1}}\| < \mu\|\boldsymbol{f}_{\vec{n}}\|$, alors il y a adhérence.
- Si $\|\boldsymbol{f}_{\vec{t_1}}\| = \mu\|\boldsymbol{f}_{\vec{n}}\|$, alors il y a glissement.

Couplage mécanique L'application d'une force sur un point quelconque, entraîne généralement le déplacement d'un ensemble de points situés dans le voisinage. Intuitivement, on peut remarquer que le nombre de points influencés va dépendre des propriétés mécaniques de l'objet (voir figure 4.2b). Si l'objet est très mou, seul un voisinage très local sera affecté, alors que si l'objet est rigide, l'intégralité des points de l'objet seront influencés. Pendant la résolution, il sera alors important de prendre en compte le couplage mécanique entre les contraintes.

Par ailleurs, le frottement est également influencé par le couplage mécanique des contacts. Par exemple, si on détecte un contact adhèrent pendant la résolution, il est tout à fait possible que ce même contact finisse par devenir glissant s'il se retrouve entraîné par des forces provenant d'autres points du maillage.

En couplant ce phénomène à la loi de Signorini, cela pose le problème du choix des contraintes sur lesquelles les forces doivent être appliquées (les contraintes actives). En effet, le déplacement d'un point va engendrer le déplacement de plusieurs points aux alentours, ce qui peut résoudre plusieurs contraintes sans qu'aucune force ne leur soit appliquée, et à l'inverse il peut générer ou amplifier de nouvelles collisions. Cela mène alors à un problème combinatoire complexe pour choisir les contraintes actives, et simultanément déterminer les forces à appliquer.

Méthodes particulaires

Maintenant qu'on connaît les phénomènes physiques qu'il est important de simuler, nous présentons les principales méthodes qui permettent de calculer des forces de réponses aux contacts dans une simulation interactive. Une première famille de méthodes consiste à appliquer des forces sur le système qui vont pénaliser les contraintes.

Réponse par pénalités Lorsque les contacts se produisent, ils doivent être pris en compte dans le système d'équations décrivant le mouvement des objets Moore et Wilhelms (1988). La solution la plus simple consiste à pénaliser les contacts, par l'ajout d'une force $\boldsymbol{f} = k\boldsymbol{\delta}\vec{\boldsymbol{n}}$ sur chaque point de contact. Cette solution est équivalente à placer des ressorts entre les primitives en contact, et où k correspond alors à la rigidité du ressort. La simplicité et la rapidité de cette solution fait qu'elle a été reprise dans de nombreux articles traitant des contacts d'objets déformables James et Pai (2004); Hirota et al. (2001) mais également pour des objets surfaciques comme les vêtements Bridson et al. (2002).

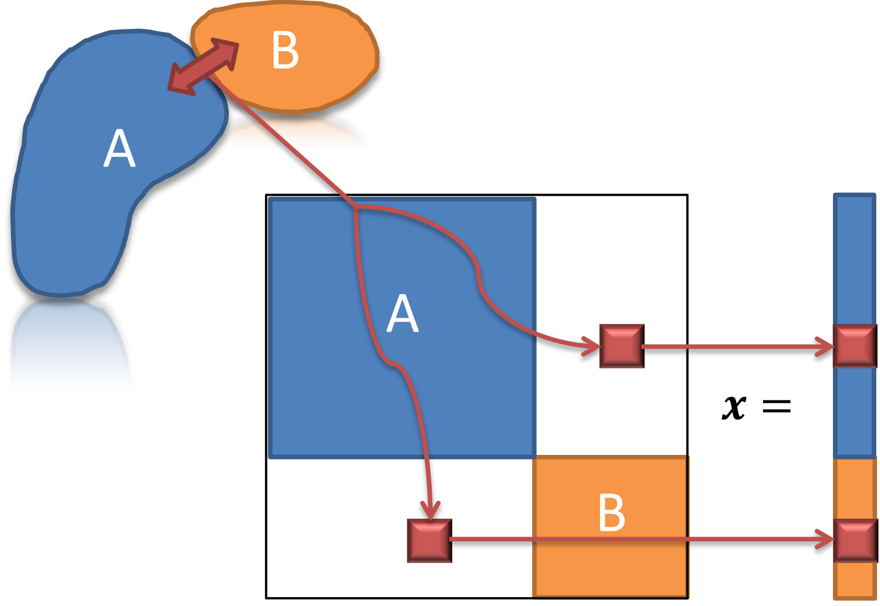


Fig. 4.3 – Ajout de ressorts entre les systèmes mécaniques des objets en interactions.

Les forces de contact doivent être suffisamment rigides pour annuler les collisions. Or on a vu que cela est problématique si on utilise un intégrateur explicite, puisque la rigidité implique d'utiliser de très petits pas de temps. À l'inverse, les forces de pénalités peuvent

être utilisées avec une intégration implicite, et les forces sont alors placées dans le même système que les forces internes des objets. Cependant, cela modifie les rigidités du système, et la matrice $\mathbf{A}$ doit alors être modifiée en correspondance. Une des conséquences est que les systèmes mécaniques des objets en contacts se trouvent couplés, on devra alors résoudre de très grands systèmes d'équations (voir figure 4.3), ce qui est pénalisant pour les temps de calcul. De plus, l'ajout de ressort tend à diminuer le conditionnement du système, et cette méthode n'est pas adaptée à l'utilisation du préconditionneur présenté dans le chapitre précédent, puisque les contacts apparaissent et disparaissent sans cohérence temporelle.

L'inconvénient majeur de cette technique, est qu'elle n'est pas physiquement correcte. Le paramètre de rigidité n'a pas de signification mécanique, et il dépend de nombreux autres paramètres (rigidité des objets, taille des pas de temps, ...). De plus, la seule garantie que l'on possède est que les ressorts vont avoir tendance à diminuer les interpénétrations, mais pas nécéssairement de les résoudre (ce qui ne respecte pas la loi de Signorini). Si on choisit une raideur trop forte les objets vont alors se séparer de façon trop importante, et si elle n'est pas assez forte de nombreuses interpénétrations vont apparaître.

Réponse par contraintes

Une autre famille de méthode consiste à imposer un mouvement de sorte que les contraintes soient totalement vérifiées après la résolution. Ces méthodes permettent d'assurer une configuration sans interpénétration après la résolution Popescu et Compton (2003) Galoppo et al. (2006). Pour cela il est nécessaire de déterminer les forces à appliquer sur chacune des contraintes pour qu'elles soient toutes vérifiées simultanément. Ces méthodes nécessitent de résoudre des équations complexes que nous présenterons dans le chapitre suivant. Nous introduisons ici uniquement les informations nécessaires à la compréhension des contributions de ce chapitre.

Multiplicateurs de Lagrange Les multiplicateurs de Lagrange sont un outil mathématiques, qui permet de trouver le minimum d'une fonction, sous contrainte. Ils peuvent être utilisés pour de nombreuses applications, mais nous les utilisons ici dans le cadre de la résolution des contacts. Pour deux objets en interaction, l'équation (2.42) est alors modifiée[4] par :

$$\begin{aligned} \mathbf{A}_1 \boldsymbol{x}_1 &= \boldsymbol{b}_1 + \mathbf{J}_1^T \boldsymbol{\lambda} \\ \mathbf{A}_2 \boldsymbol{x}_2 &= \boldsymbol{b}_2 + \mathbf{J}_2^T \boldsymbol{\lambda} \end{aligned} \tag{4.3}$$

où $\mathbf{J}_1 = [\frac{\delta\psi}{\delta \boldsymbol{x}_1}]$ et $\mathbf{J}_2 = [\frac{\delta\psi}{\delta \boldsymbol{x}_2}]$ avec ψ qui représente les contraintes unilatérales décrites en (4.1). Ces matrices (aussi appelées *Jacobiennes des contacts*) correspondent à une linéarisation des contraintes par rapport à la position. Géométriquement, la transposée de cette matrice nous donne la normale aux contacts. $\boldsymbol{\lambda}$ est un vecteur qui contient les multiplicateurs de Lagrange. C'est en réalité un ensemble de valeurs scalaires que l'on va appliquer le long des normales aux contacts, pour résoudre totalement les intersections.

[4]On rappelle ici que cette équation est la conséquence de notre choix basée sur un intégrateur implicite. En effet, l'utilisation d'intégrateur explicite ne permet pas de prendre en compte le couplage mécanique des objets à la fin du pas de temps.

On a vu dans l'équation (4.2), que la valeur de $\boldsymbol{\lambda}$ est soumise à des conditions de complémentarité. La détermination des multiplicateurs de Lagrange nécessite alors de résoudre un système d'équations soit par une méthode directe Lenoir et al. (2004), soit de façon indirecte en constituant un *problème de complémentarité linéaire* (LCP). Ces aspects seront traités dans le chapitre suivant, l'important pour l'instant est de comprendre qu'un LCP est un problème qui permet de trouver $\boldsymbol{\lambda}$ en tenant compte des conditions de complémentarité.

Contacts frottants La difficulté pour gérer le frottement dans les simulations interactives réside dans le fait que le cône de frottement est une structure purement non linéaire. Une solution classique consiste à discrétiser le cône de frottement en un ensemble de facettes Anitescu et Potra (1997). L'avantage est alors que l'on peut écrire une équation linéaire selon chaque facette :

$$\left\{ \begin{array}{ccc} 0 \leq \boldsymbol{\delta}_{\vec{n}} & \perp & \boldsymbol{f}_{\vec{n}} \geq 0 \\ 0 \leq \boldsymbol{\beta} + \boldsymbol{\delta}_{\vec{t_1}} & \perp & \boldsymbol{f}_{\vec{t_1}} \geq 0 \\ & \ldots & \\ 0 \leq \boldsymbol{\beta} + \boldsymbol{\delta}_{\vec{t_n}} & \perp & \boldsymbol{f}_{\vec{t_n}} \geq 0 \\ 0 \leq \mu \boldsymbol{f}_{\vec{n}} - \sum_{k=1}^{r} \boldsymbol{f}_{\vec{t_k}} & \perp & \boldsymbol{\beta} \geq 0 \end{array} \right. \tag{4.4}$$

où $\boldsymbol{\beta}$ est une valeur positive mesurant le déplacement tangent, La première ligne permet d'imposer la loi de Signorini selon la normale. Les r lignes suivantes assurent que les forces tangentielles vont nécessairement agir en opposition aux mouvements Duriez (2004). Dans ces équations, les vecteurs $(\vec{t}_1, ..., \vec{t}_n)$ forment une base qu'on notera $\mathbf{T}$. Sa dimension dépend de r qui est le nombre de facettes dans la discrétisation du cône de frottement. Enfin, la dernière ligne décrit la loi de Coulomb, c'est-à-dire qu'elle assure que le contact aura un mouvement tangentiel si et seulement si la force reste à l'intérieure du cône discrétisé.

L'inconvénient de cette formulation est qu'elle utilise une approximation du cône de frottement discrétisé en facettes, mais surtout que le nombre de contraintes générés pour chaque contact est égal à $r + 2$. Nous montrerons dans le chapitre suivant comment nous pouvons résoudre un contact frottant en ne générant que 3 contraintes avec un solveur itératif dédié. Cependant, de manière à garder une formulation de type LCP (Linear Complementarity Problem) nous utiliserons cette formulation dans ce chapitre. Cela permet de garder un grand choix dans la stratégie de résolution du problème de complémentarité. Ainsi, si on ajoute du frottement dans l'équation (4.3) on obtient :

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b} + \mathbf{J}^T\boldsymbol{\lambda} + \mathbf{T}^T\boldsymbol{\beta}, \tag{4.5}$$

Génération des contraintes Dans la loi de Signorini, les conditions sont posées au niveau des positions. Cependant, pour éviter des problèmes numériques certaines méthodes résolvent les contraintes au niveau des vitesses Stewart (2000); Stewart et Trinkle (1996); Anitescu et Potra (1997). Pour cela, ils proposent de résoudre un problème plus simple

dans lequel les conditions de Signorini sont affaiblies :

$$\text{Si } \boldsymbol{\delta} \leq 0, \qquad 0 \leq \mathbf{J}(\boldsymbol{v}_0 + \Delta \boldsymbol{v}) \perp \boldsymbol{\lambda} \geq 0 \tag{4.6}$$

Le traitement au niveau des vitesses est typique dans de nombreux ouvrages, car cela permet de produire des simulations stables avec de grand pas de temps Baraff et Witkin (1998). Otaduy et al. (2009) utilisent des contraintes au niveau de la vitesse pour simuler des objets déformables et rigides simultanément. D'autres méthodes à base de LCP formulés en vitesses ont également été utilisées pour le couplage fluide-solide Batty et al. (2007), ou pour la simulation de frottement anisotropique Pabst et al. (2009). Cependant, la résolution en vitesses ne permet pas de garantir que les intersections seront résolues entièrement à la fin du pas de temps, mais seulement qu'elles ne vont pas augmenter.

Pour interdire toute interpénétration, on peut stopper la simulation au moment du contact (quand delta=0) et résoudre en vitesse (mais cela implique d'utiliser une détection de collision continue). Une autre solution consiste à résoudre les intersections au niveau des positions. Cependant, le respect total des conditions de Signorini formulées au niveau de la position, peut introduire des instabilités. En effet, si on résout totalement les intersections, alors une détection de collision discrète ne pourra pas produire un contact continu. Le traitement pour "suivre" les contacts se rapprochent alors des problèmes de la détection de collision continue. D'autres solutions consistent à utiliser une *détection de proximité*, afin d'ajouter des contraintes supplémentaires pour les objets proches de l'intersection Johnson et Willemsen (2004). Ainsi, on peut anticiper les collisions, et appliquer une force continue.

Par ailleurs, la détection des collisions de façon discrète, nécessite d'être en mesure de traiter de grandes interpénétrations Bridson et al. (2002) ; Harmon et al. (2008) ; Guendelman et al. (2003). Ceci peut également provoquer des problèmes numériques. Une solution consiste à ajouter une étape de post stabilisation Ascher et Petzold (1998). Pour cela, on peut par exemple résoudre les contraintes en vitesse, puis corriger la position par un processus itératif Cline et Pai (2003) ; Anitescu et Hart (2004).

4.2.4 Réponses par pénalités basées sur le volume d'intersection

Nous terminons cette section en présentant les travaux de Faure et al. (2008), dans la mesure où la méthode proposée a servi de première base pour les contributions qui sont présentées dans ce chapitre. Les auteurs ont proposé une méthode élégante pour déduire une force de pénalité à partir des informations de volumes déduites obtenues à partir des LDI. L'idée de base est de minimiser le volume d'intersection entre deux polyèdres, pour résoudre les intersections.

La méthode est basée sur une technique de rendus successifs pour obtenir les LDI. La rastérisation peut être faite dans n'importe quelle direction, mais pour plus de simplicité, nous supposons qu'elle est réalisée dans une direction orthogonale sur l'un des axes

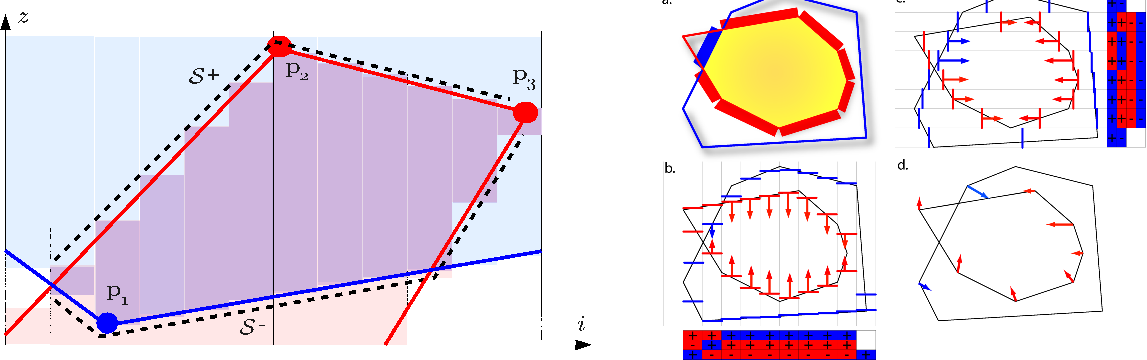


FIG. 4.4 – Une tranche 2D d'une LDI montrant les volumes objet d'intersection de deux objets. Ici, la direction du regard LDI est l'axe z. Les sommets sont étiquetés $\boldsymbol{p}$, tandis que les pixels sont illustrés par des lignes horizontales dans différentes colonnes. Le volume d'intersection apparaît en violet, et est délimité par des ensembles de pixels de surface, $\mathcal{S}^+$ sur le dessus, et $\mathcal{S}^-$ sur le fond (en pointillés).

FIG. 4.5 – Vue d'ensemble de l'algorithme pour deux objets en intersection (issu de Faure et al. (2008)). (a) Le volume d'intersection est dessiné en jaune, les rectangles rouge et bleu sont les pressions. Chaque rastérisation (b et c) donne un ensemble de forces unitaires appliquées aux pixels (dans la direction de la rastérisation). Leur somme sur les sommets, nous donne le gradient du volume.

principaux[5] ($\vec{x}, \vec{y}$ ou $\vec{z}$). Ainsi, le volume calculé selon la projection sur z est :

$$\mathcal{V} = a \sum_{(i,j)\in\mathcal{S}_z^+} \mathrm{z}_{ij}^+ \; - \; a \sum_{(i,j)\in\mathcal{S}_z^-} \mathrm{z}_{ij}^-, \tag{4.7}$$

où a est la surface d'un pixel, z_{ij}^+ et z_{ij}^- sont les profondeurs des pixels supérieures et inférieures, et les ensembles de $\mathcal{S}_z^+$ et $\mathcal{S}_z^-$ contiennent respectivement les emplacements de pixels (i, j) des surfaces de contact supérieures et inférieures (voir figure 4.4). Le volume nous indique la quantité d'interpénétration entre les deux polyèdres, mais cette information n'est pas suffisante pour résoudre les collisions. Il est nécessaire de savoir dans quelle direction il faut déplacer les points de contrôle pour diminuer l'intersection. Pour obtenir cette information, on cherche à calculer le gradient de volume au niveau des nœuds, qui représente la variation de volume (dans la direction du LDI) en réponse à un déplacement unitaire d'un sommet. Pour un LDI projeté selon la direction z, on peut exprimer le gradient sur le sommet k d'un triangle, comme suit :

$$\frac{\partial\mathcal{V}}{\partial\boldsymbol{p}_k^z} = a \sum_{(i,j)\in\mathcal{S}_z^+} \frac{\partial\mathrm{z}_{ij}^+}{\partial\boldsymbol{p}_k^z} \; - \; a \sum_{(i,j)\in\mathcal{S}_z^-} \frac{\partial\mathrm{z}_{ij}^-}{\partial\boldsymbol{p}_k^z}. \tag{4.8}$$

où $\boldsymbol{p}_k^z$ est la coordonnée z du vertex k. Le scalaire $\partial\mathrm{z}_{ij}(\boldsymbol{p})/\partial\boldsymbol{p}_k^z$ correspond simplement au coefficient barycentrique, utilisé pour interpoler la valeur de la profondeur en fonction du sommet k. Faure et al. (2008) attribuent une couleur différente à chaque sommet du

[5] Si les axes ne sont pas alignés, il est toujours possible de se rapporter à un problème équivalent en multipliant les coordonnées par une matrice de projection 4 × 4, de façon similaire à ce qui est fait en openGL pour afficher les triangles dans le repère de l'écran.

triangle avant de faire un rendu de la scène. De cette façon, la couleur résultante de chaque pixel traduit la valeur du gradient sur les sommets du triangle.

Le calcul du gradient du volume d'intersection nécessite trois LDI dans des directions orthogonales (voir figure 4.5). Pour chaque sommet, on accumule alors la somme du volume d'intersection, la somme des gradients dans les directions concernées. Autrement dit, le volume est accumulé à trois reprises (et par la suite corrigé par $\frac{1}{3}$), tandis que chaque composante du gradient est accumulée dans le vecteur gradient final.

$$\frac{\partial \mathcal{V}}{\partial \boldsymbol{p}_k} = \left(\frac{\partial \mathcal{V}}{\partial \boldsymbol{p}_k^x} \; \frac{\partial \mathcal{V}}{\partial \boldsymbol{p}_k^y} \; \frac{\partial \mathcal{V}}{\partial \boldsymbol{p}_k^z} \right) \tag{4.9}$$

où x, y, z sont les trois directions successives de visualisation des LDI. Faure et al. (2008) ont ensuite proposé d'exploiter ces informations pour en déduire une force de répulsion, basée sur une formulation en volume. L'énergie associée aux volumes d'interpénétration est donnée par $E = \frac{1}{2}k\mathcal{V}^2$, ainsi la force sur le vertex $\boldsymbol{p}_i$ est calculée comme suit :

$$-\frac{\partial E}{\partial \boldsymbol{p}_i} = -k\mathcal{V}\frac{\partial \mathcal{V}}{\partial \boldsymbol{p}_i} \tag{4.10}$$

Dans ce chapitre, nous exploitons le volume et son gradient différemment, mais ils restent les seules valeurs nécessaires à calculer.

4.3 Détection de collision GPU

Nous présentons maintenant notre méthode de détection de collisions, applicable à tous objets délimités par une surface fermée. La détection de collisions est basée sur une rastérisation des triangles, de laquelle nous pouvons déduire les volumes et les gradients d'interpénétration. Avec la facilité de programmation des processeurs graphiques, nous avons implémenté notre propre algorithme de rastérisation logicielle. Celui-ci permet d'obtenir un LDI, en une seule rastérisation, beaucoup plus rapidement que les méthodes basées sur des multiples rendus openGL, comme c'était le cas avec la méthode présentée précédemment Faure et al. (2008). La consommation mémoire est également diminuée, puisque nous ne stockons que la liste des layers sous chaque pixel, et non une image complète pour chaque nouvelle couche.

Les architectures visées sont les GPU programmables qui disposent des opérations atomiques. Toutefois, lorsque ces opérations ne sont pas disponibles, nous proposerons des solutions alternatives qui permettrons de contourner ce problème. Notre implémentation actuelle peut être utilisée sur toutes les cartes graphiques supportant CUDA, et il devrait également être possible de l'exporter n'importe quel matériel qui supporte OpenCL.

4.3.1 Construction des LDI

Notre premier objectif est de construire les LDI dans les 3 directions orthogonales, l'ensemble des calculs présentés ici sera donc exécuté trois fois par pas de temps. Par exemple, sur la figure 4.6 les pixels rouges, verts et bleus représentent des LDI calculés dans les directions respectivement $\vec{x}$, $\vec{y}$, et $\vec{z}$.

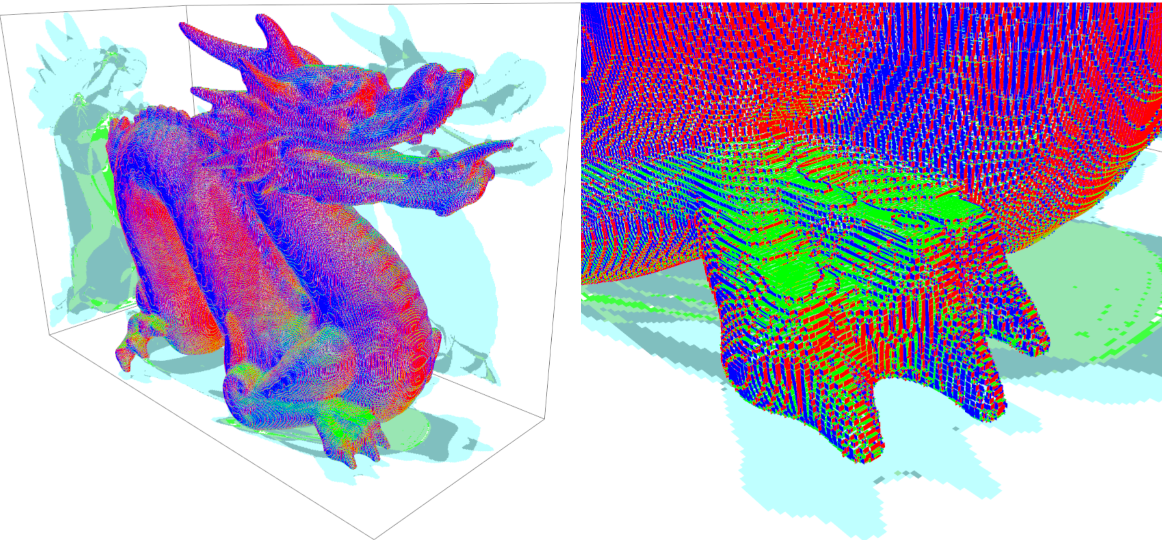


Fig. 4.6 – 3 LDI Combinés le long des trois axes mutuellement orthogonaux, fournissent une représentation exacte volumétrique.

Chaque LDI est stocké dans des structures indépendantes. Pour faciliter les regroupements des calculs sur le GPU, nous utiliserons la notion de *cellule*, qui est en réalité un groupement (de taille prédéfinie à la compilation) de pixels. Pour extraire du parallélisme, les triangles sont alors traités et triés dans des listes séparées, associées à chacune des cellules. Le résultat final sera une liste layer, ordonnée selon la profondeur, qui sera associée indépendamment à chaque pixel du LDI. Pour chacun, on stockera à la fois une valeur de profondeur ainsi que l'indice du triangle qui les a générés (voir figure 4.7).

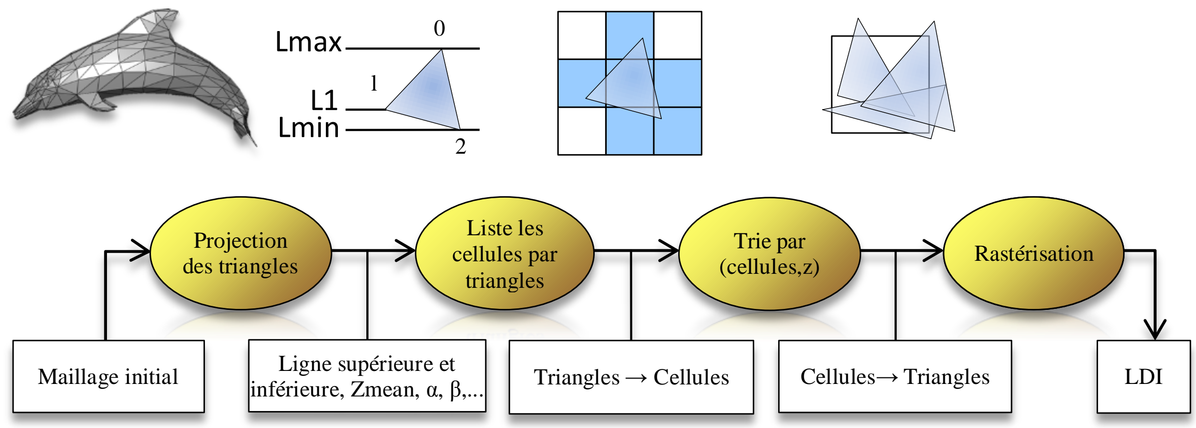


Fig. 4.7 – Vue d'ensemble des calculs éffectués pour obtenir un LDI.

Étapes préliminaires

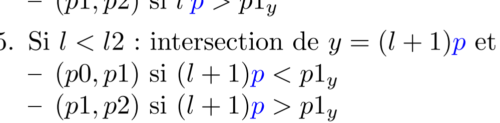
Nous décrivons tout d'abord un ensemble d'étapes préliminaires qui permettent d'obtenir pour chaque cellule, la liste des triangles qui l'intersecte.

Projection des triangles La première étape de l'algorithme de rastérisation consiste à projeter chaque sommet en coordonnées pixels, et à préparer les informations requises dans chaque triangle pour configurer sa rastérisation. La parallélisation utilisée est de considérer chaque triangle indépendamment, et on lance donc autant de threads GPU que de triangles. Lorsque le LDI est aligné sur les axes du repère de la scène (comme c'est le cas dans notre implémentation), l'étape de projection consiste simplement à permuter les coordonnées $3D$ de sorte que la coordonnée Z corresponde à la profondeur dans l'image du LDI. Les coordonnées X et Y sont également mises à l'échelle de sorte qu'une unité corresponde à un pixel.

Pour préparer la rastérisation, les 3 sommets de chaque triangle sont triés par Y, tels que $p0$ soit le plus bas sommet, $p1$ au milieu et $p2$ le sommet le plus haut. On détermine également si le triangle est *front-face* (entrée dans le volume) ou *back-face* (sortie du volume).

Lister les cellules La seconde étape consiste à calculer pour chaque triangle, la liste des cellules qu'il intersecte. Ainsi, en associant un thread GPU à chaque triangle, on cherche à connaître l'intervalle de lignes et de colonnes intersectées. Avec la permutation précédente, l'intervalle[6] des lignes de cellules couverts par le triangle est $[l0, l2] = \left[\left\lfloor \frac{p0_y}{p} \right\rfloor, \left\lfloor \frac{p2_y}{p} \right\rfloor\right]$, où p est la taille des pixels (qui est un paramètre contrôlé par l'utilisateur). Pour chaque ligne l, les colonnes de cellules intersectées correspondent au minimum et le maximum des valeurs calculées sur la figure 4.8.

1. $p0_x$ Si $l = l0$,
   Sinon intersection de $(p0, p2)$ et $y = l\,p$
2. $p2_x$ Si $l = l2$,
   Sinon intersection de $(p0, p2)$ et $y = (l+1)p$
3. $p1_x$ Si $l = \left\lfloor \frac{p1_y}{p} \right\rfloor$



4. Si $l > l0$: intersection de $y = l\,p$ et
   – $(p0, p1)$ si $l\,p < p1_y$
   – $(p1, p2)$ si $l\,p > p1_y$
5. Si $l < l2$: intersection de $y = (l+1)p$ et
   – $(p0, p1)$ si $(l+1)p < p1_y$
   – $(p1, p2)$ si $(l+1)p > p1_y$

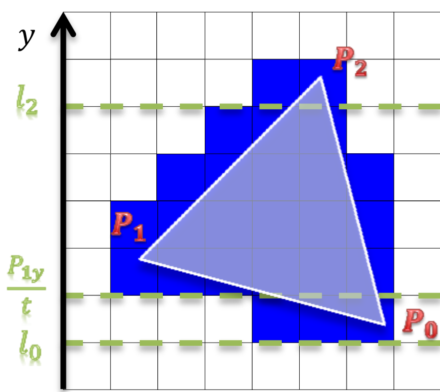


Fig. 4.8 – Test réalisé sur chaque ligne pour déterminer l'intervalle de colonne de cellules intersectées par un triangle. On prends le minimum et le maximum des test pour connaître l'intervalle désiré. t représente la taille des cellules.

En fonction que P_1 se trouve à gauche ou à droite du segment $[P_2, P_0]$, les deux premiers tests, nous donnent le minimum ou le maximum de l'intervalle cherché. Les autres tests nous assurent de calculer la seconde borne de l'intervalle recherché. On remarque que plusieurs tests sont redondants, mais comme nous calculons plusieurs triangles en parallèles, cela permet d'optimiser l'efficacité des processeurs SIMD en appliquant la même instruction aux threads d'un même warp. En énumérant chacune des cellules des lignes dans l'intervalle couvert par le triangle, chaque thread peut ainsi générer la liste des cellules

[6] Les triangles sont en général petits, par conséquence l'intervalle $[l0, l2]$ est petit également.

qu'il recouvre. On écrit ces informations dans deux vecteurs d'indices, l'un contient la liste des cellules, et l'autre les triangles associés.

Cependant, on se retrouve face à un problème d'écriture concurrente, car on ne sait pas à l'avance combien de cellules seront coupées par les triangles. Il faut donc utiliser un mécanisme permettant de résoudre les conflits d'écritures. Si une opération d'ajout parallèle dans une liste est disponible sur l'architecture (comme le CPU ou Larabee Seiler et al. (2008)), les entrées peuvent être directement ajoutées dans les listes de cellules. Dans ce cas on obtient directement la liste des triangles sous chaque cellule, et on peut directement passer à l'étape de rastérisation. Cependant, il n'existe pas de telle opération sur le GPU, mais plusieurs implémentations peuvent être envisagées :

1. Si les opérations atomiques sont disponibles, un compteur atomique peut être utilisé pour stocker la dernière entrée disponible dans le vecteur de sortie. Chaque thread incrémente alors le compteur une fois qu'il a calculé le nombre d'entrées nécessaires, puis il remplit le vecteur résultat à partir de la valeur pré-incrémentée. Cependant, il est nécessaire d'allouer un vecteur suffisamment grand pour accueillir toutes les données.
2. Sinon, on peut réaliser l'opération en deux temps. Un premier kernel compte le nombre de cellules sous chaque triangle, puis un second kernel écrit réellement les données. Comme le résultat du permier kernel est un seul scalaire par triangles, on peut stocker cette information sans conflit d'écriture. On utilise ensuite *scan* parallèle (voir annexe D) qui nous donne pour chaque triangle, l'indice de la première entrée dans le vecteur final. On peut enfin utiliser un dernier kernel pour remplir le vecteur de sortie en utilisant les indices calculés par scan pour connaître l'adresse d'écriture.

En pratique, nous avons implémenté la seconde solution car cette opération est relativement rapide, et nous souhaitions garder un maximum de compatibilité avec le matériel sans écriture atomique. De plus, l'utilisation d'un compteur atomique mène à des performances difficiles à évaluer puisque beaucoup de triangles sont susceptibles de vouloir incrémenter le compteur simultanément. En définitive, on obtient deux vecteurs associés l'un à l'autre, dont chaque valeur constitue une paire (indice du triangle, cellule).

Tri par cellules À la fin de l'étape précédente on connaît pour chaque triangle la liste des cellules qu'il intersecte, mais il est nécessaire de réordonner ces vecteurs de sorte que tous les triangles à l'intérieur de chaque cellule soient rassemblés (voir figure 4.9). Ceci peut être fait en utilisant un tri parallèle *radix sort* Satish et al. (2009), avec l'indice de cellule comme clé de tri.

À la suite de ce tri, tous les indices de triangles qui recouvrent une même cellule sont stockés de façon continue dans les vecteurs triés. Il reste cependant à déterminer le début et la fin des données de chaque cellule. Pour cela, on utilise un kernel, dans lequel chaque thread est associé à une valeur dans les vecteurs triés. Chacun va alors lire deux indices de cellules successifs [7]. Si ces indices sont différents, cela signifie que l'indice du thread correspond à la fin des données de la première cellule, et au début des données de la

[7] Notons que pour maximiser les performances, il est important de lire une seule valeur de façon alignée dans chaque thread, puis de copier les données en mémoire partagée. On pourra alors faire le test sur les données partagées après une synchronisation

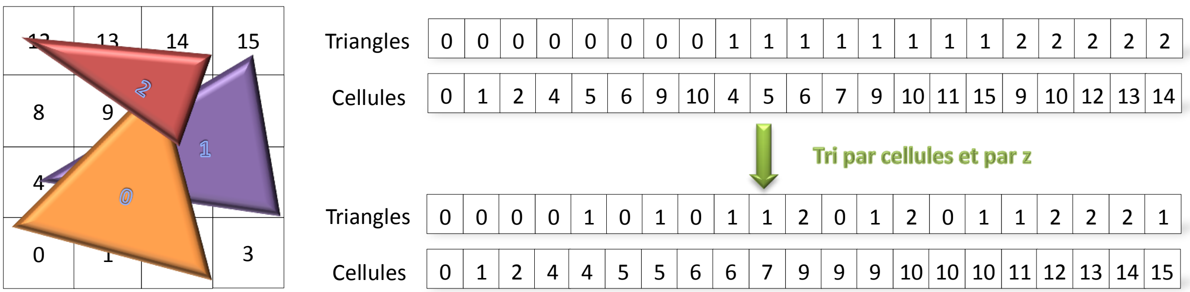


Fig. 4.9 – Tri par cellules. Avant le tri on connaît la liste des cellules sous chaque triangle, après ce tri on peut déterminer la liste des triangles dans chaque cellule.

seconde. Comme les cellules sont triées, un seul thread par cellule passera ce test, et il suffit de copier son indice dans le vecteur résultat (qui possède la même taille que le nombre de cellules). On évite ainsi les conflits d'écriture.

Par ailleurs, pour l'étape finale de rastérisation, il faudra générer les layers de chaque triangle, dans l'ordre de profondeur. Or, le nombre de cellules reste relativement petit, et par conséquent nous avons besoin d'un petit nombre de bits pour encoder leur indice. Une optimisation consiste donc à utiliser les bits de poids fort pour coder les indices de cellules, et les bits de poids faible pour coder la profondeur moyenne. Ceci peut être fait facilement dans l'étape précédente, où nous listions les cellules sous chaque triangle. Suite au tri, on obtiendra alors la liste des triangles sous chaque cellule, dans un ordre approximativement trié. Ceci permettra d'accélérer sensiblement la rastérisation en minimisant le nombre de ré-ordonnancements.

Rastérisation des triangles

On va maintenant procéder à la rastérisation de triangles, qui est le cœur de notre méthode, et c'est ici que la plupart du temps de calcul est passé. Cette étape doit donc être implémentée de façon optimisée. Les étapes précédentes ont permis de répartir les triangles dans chaque cellule, et on cherche maintenant à connaître l'intersection ainsi que la profondeur des triangles sous chaque pixel. Concrètement, on va associer un bloc CUDA au calcul de chaque cellule, dans chacun un ensemble de threads va être utilisé pour rastériser plusieurs triangles en même temps. Cette stratégie de parallélisation à deux niveaux est parfaitement adaptée aux architectures parallèles comme les GPU. L'algorithme de rastérisation est présenté dans l'algorithme 5.

Pour une cellule de $t \times t$ pixels, nous utilisons $t \times t$ threads, qui vont rastériser un groupe de T triangles (dans l'implémentation actuelle, nous utilisons $t = 8$ et $T = 24$). Chaque thread calcule d'abord un masque p-bits correspondant aux pixels qui sont coupés par le triangle sur une ligne (lignes 3 à 7). Le test effectué pour connaître les pixels intersectés par le triangle est très similaire à celui utilisé pour l'intersection des cellules.

Une fois que tous les threads ont terminé (ce qui nécessite une synchronisation), ces masques sont combinés pour calculer le nombre de layers générés sur chaque pixel (lignes 8 et 9). On peut alors allouer les layers (ligne 10), mais encore une fois nous ne pouvons pas connaître cette information à l'avance, et chaque thread ne peut pas savoir à quelle adresse

```
1  parfor all tiles tile do
2      parfor thread (x, y) = (0, 0) to (t − 1, t − 1) do
3          lcount(x, y) = 0
4          for triangle t0 = 0 to nb_tri_in(tile) − 1 step T do
5              for t1 = x to T − 1 step t do
6                  mask(t1, y) = bit i set if tri. t0 + t1 covers pixel (i, y)
7              barrier
8              for t1 = 0 to T − 1 do
9                  if bit x is set in mask(t1, y) then increment lcount(x, y)
10             alloc_layers(lcount(x, y))
11             for t1 = 0 to T − 1 do
12                 if bit x is set in mask(t1, y) then
13                     tid = index of triangle t0 + t1
14                     z = depth of triangle tid at pixel (x, y)
15                     add (tid, z) into layers(tile, x, y) with insertion sort
16             barrier
```

Algorithme 5 : Algorithme de rastérisation parallèle

écrire ses données. Comme précédemment, on peut utiliser un compteur atomique, ou alors utiliser une stratégie en deux temps (où on compte d'abord le nombre de layers générés, puis on les écrit dans un second kernel). Nous avons implémenté les deux stratégies, mais comme le kernel de rastérisation est le plus coûteux, nous utilisons la version atomique quand le matériel nous le permet. Cependant, même si cette solution autorise de connaître l'adresse de la première donnée pour un thread en particulier, il est important que le vecteur soit assez grand pour accueillir tous les layers. En pratique, on utilise un paramètre contrôlé par l'utilisateur qui estime le nombre de layers maximum sous chaque pixel dans la scène. En le multipliant par le nombre de pixels sur le LDI, on obtient généralement une taille plus grande que nécessaire, mais cela nous permet de diminuer le temps de calcul.

La deuxième étape de la rastérisation, consiste à calculer la profondeur de chaque triangle sous chaque pixel (la rastérisation en elle-même). Les threads sont maintenant utilisés pour traiter T triangles en parallèles, dont le masque est actif (lignes 11 à 15). On utilise ensuite les informations de profondeur des sommets du triangle, et les coordonnées du pixel, pour calculer la profondeur du layer. Chaque layer est ainsi ajouté dans le vecteur final, en appliquant un simple tri par insertion selon sa profondeur.

Grâce au tri approximatif pendant les étapes préliminaires, les layers sont souvent générés dans l'ordre correct, et le tri par insertion est en général très efficace. En revanche, si le tri approximatif ne fournit pas l'ordre exact sur le pixel considéré (si par exemple les triangles se croisent), il sera nécessaire de déplacer des layers précédemment écrits. Or, comme ces layers sont stockés en mémoire globale, cette opération est très coûteuse. Une optimisation qui permet d'améliorer grandement les performances, est de maintenir quelques derniers layers (2 dans notre implémentation) dans les registres. Ainsi, si seulement deux triangles sont inversés (qui se produit souvent dès que les triangles se coupent), leur ordre est corrigé en utilisant les registres qui sont très rapides. Les échanges dans la mémoire globale ne sont alors nécessaires que dans des cas exceptionnels.

4.3.2 Utilisation des LDI

Nous disposons maintenant des LDI dans les trois directions, et nous allons montrer comment nous pouvons les utiliser pour détecter les collisions. Remarquons cependant que contrairement aux travaux de Faure et al. (2008), nos LDI sont composés des layers des objets de toute la scène (au lieu d'un LDI par paire d'objets en contact). Ainsi, notre solution permet de diminuer largement les coûts de stockage des temps de calcul. En conséquence, on peut simuler des scènes avec de nombreux objets, de manière aussi efficace que pour quelques objets complexes. Le principal facteur qui influence le temps de calcul devient alors le nombre total de triangles, et la taille de la zone à rastériser, mais le nombre de layers n'est alors plus discriminant. On peut également améliorer les performances en ne rastérisant que l'intersection des boîtes englobantes de toutes les paires d'objets pouvant être en contact (ainsi que l'union des boîtes englobantes de tous les objets potentiellement en auto-collision).

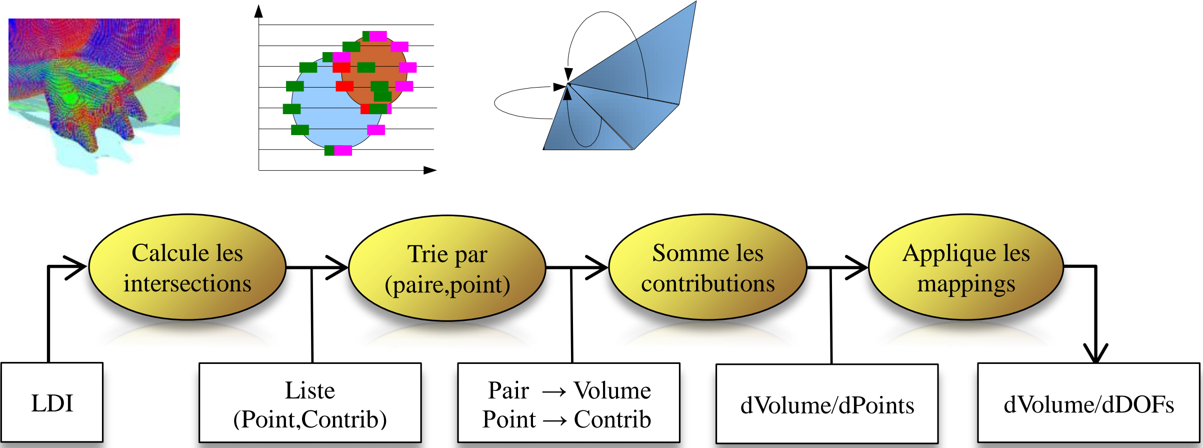


Fig. 4.10 – Vue d'ensemble des calculs effectués pour détecter les collisions et décrire les équations aux nœuds.

Détection des intersections

La motivation principale pour calculer les LDI, est de détecter les intersections qui se produisent entre les objets déformables. On va maintenant pouvoir trouver facilement les intersections entre les objets (voir figure 4.10).

Calcul des intersections Pour détecter les intersections, il faut maintenant parcourir tous les layers d'un même pixel. On va pour cela associer un thread à chaque pixel, qui va traiter la liste des layers associés. Comme les layers sont triés selon la profondeur, on peut facilement trouver une intersection à l'aide d'un compteur. Chaque fois qu'on passe par un layer front-face, on incrémente le compteur. Et inversement, à chaque fois qu'on passe par un layer back-face on le décrémente. Il y a collision dès que le compteur est strictement supérieur à 1 (voir figure 4.11).

Pour stocker les informations caractérisant une collision, on se retrouve face à un problème de conflit d'écriture parallèle. En effet, on ne sait pas combien d'intersections vont être

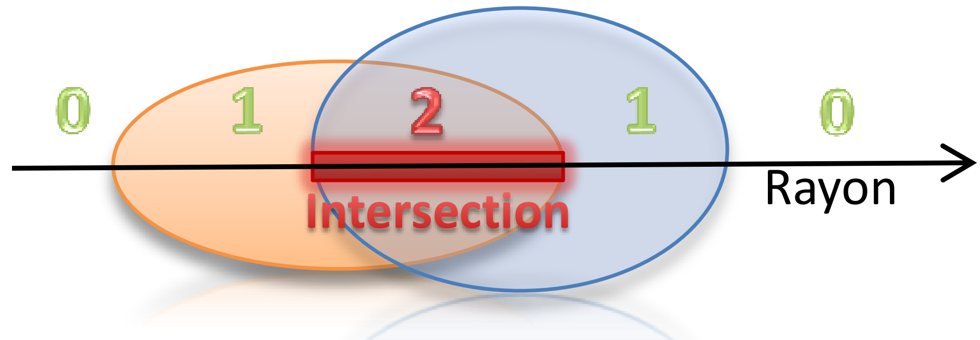


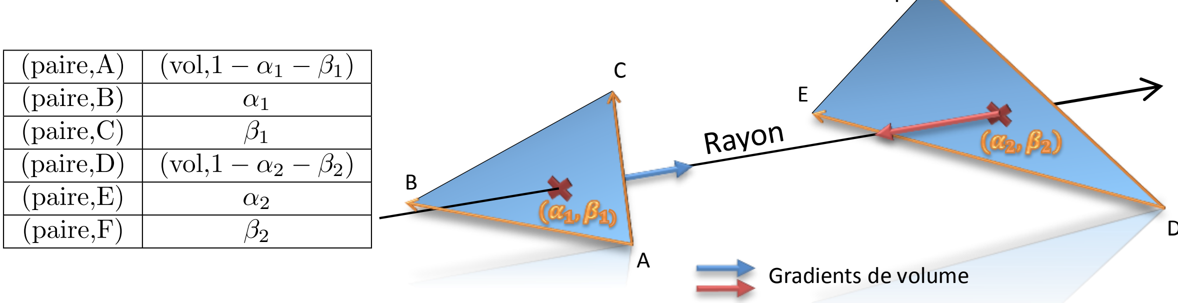
FIG. 4.11 – Détection des collisions sous un layer en utilisant un compteur.

générées sur chaque pixel, on ne peut donc pas savoir à l'avance l'emplacement mémoire où chaque thread peut écrire ses données. Comme précédemment, on peut utiliser (en fonction des capacités du matériel) les différentes stratégies que nous avons déjà présentées. Comme cette étape est relativement rapide, nous utilisons la version sans compteur atomique. Un premier kernel compte le nombre d'intersection, puis un scan parallèle nous donne l'indice de la première donnée de chaque thread, et enfin un dernier kernel calcule réellement les valeurs.

Quand une intersection est trouvée, on calcule la profondeur entre les deux layers, ainsi que le gradient du volume (qui est obtenu à partir des coefficients barycentriques des triangles en collision). Pour chacune, il faudra également être en mesure de retrouver les indices des triangles ayant généré l'intersection, mais également la paire d'objets en intersection. En effet, pour pouvoir accumuler le volume d'intersection indépendamment pour chaque paire d'objets en contact, il faut leur attribuer un identifiant unique. Les identifiants sont calculés sur le CPU, à partir des informations données par le premier kernel qui réalise le comptage des intersections.

Chaque intersection génère alors 6 valeurs pour la contribution du gradient sur chacun des points des triangles concernés, et 2 flottants pour le volume d'intersection. À chaque intersection est également associé l'indice de la paire d'objets en contact, et les indices des points des triangles concernés. Comme le nombre de paires d'objets reste petit, on stocke sur un même entier les indices des triangles sur les bits de poids faibles, et les indices des paires d'objets sur les bits de poids forts. Les contributions du gradient sont associées dans un second vecteur, qui contient également le volume d'intersection mais qui ne contribue qu'une seule fois par layer (voir figure 4.12). Ces informations sont stockées dans un vecteur d'entiers, pour cela la précision des floats est diminuée, afin de pouvoir encoder deux floats sur la même valeur. L'intérêt majeur réside dans le fait que pour l'étape suivante nous ne devrons trier qu'un seul tableau.

Chaque LDI est traité successivement, et toutes les intersections détectées selon les différents axes sont ajoutées à la suite dans le même vecteur final. Un autre avantage de la méthode, est qu'on peut également gérer les auto-collisions sans surcoûts importants. En effet, il suffit pour cela d'affecter un indice de paire d'objet en contact unique pour tous les objets pouvant être en auto-collisions. Ceci permettra de générer un volume indépendant pour un objet qui est en contact avec lui-même.



(paire,A)	(vol,$1-\alpha_1-\beta_1$)
(paire,B)	α_1
(paire,C)	β_1
(paire,D)	(vol,$1-\alpha_2-\beta_2$)
(paire,E)	α_2
(paire,F)	β_2

FIG. 4.12 – Structure de données calculée pour chaque intersection trouvée. α_1 et β_1, sont les coefficients barycentriques du triangle 1.

Tri par paire Pour pouvoir rassembler rapidement les volumes par points et non plus par pixels, on trie les tableaux générés à l'étape précédente. Comme précédemment, on utilise un radix sort, parallèle sur GPU, en utilisant les indices comme clé de tri. À la suite du tri, les intersections sont rassemblées par paires d'objets en intersection, et par indice de points.

Somme des contributions Notre objectif est maintenant de sommer les contributions du volume de toutes les intersections ayant le même indice de paire, pour obtenir le volume total d'intersections entre la paire d'objets concernée. On cherche par la même occasion à sommer les contributions du gradient de volume par points. Encore une fois, face au problème selon lequel on ne peut pas prédire à l'avance le nombre de points qui seront accumulés sur le même vertex, on ne sait pas où écrire les données finales. On procède avec la méthode classique de comptage, mais la différence est qu'il faudra appeler deux scans successifs pour déterminer les adresses des deux informations que l'on cherche (gradient de volume par points, et le volume par paires). L'accumulation est similaire à réaliser de nombreux scans en parallèles, à l'exception qu'il faut sommer les contributions entre les différentes bornes du tableau données par les indices. Au final, on obtient la somme du gradient de volume pour les 3 LDI, avec un vecteur d'indice associé qui donne le début et la fin des données de chaque point. On obtient également le volume d'intersection entre chaque paire d'objets en contact, dans la section suivante nous montrerons comment exploiter ces informations pour produire une réponse aux collisions basées sur des contraintes.

Transparence et rendu volumique

Notre rastériseur LDI peut également être utilisé pour rendre des objets transparents. En effet, comme tous les layers de chaque pixel sont disponibles dans l'ordre de profondeur correcte, on peut utiliser cette propriété pour simuler la transparence des objets, ainsi que d'autres effets comme le rendu volumétrique (translucidité, brouillard, ...). Pour chaque objet on peut définir un coefficient de transparence α compris entre O et 1 qui indique le degré de transparence de l'objet. Ainsi, en appliquant la formule de l'alpha blending, on est capable de calculer la quantité de lumière qui traverse un layer avec $\alpha_{pixel} = 1 - \alpha_{objet}$. La couleur sur le pixel est calculée par $color_{pixel} = color_{objet} * \alpha_{pixel} * \alpha_{objet}$.

Le rendu peut être mis en œuvre soit comme une étape de post-traitement, où le LDI

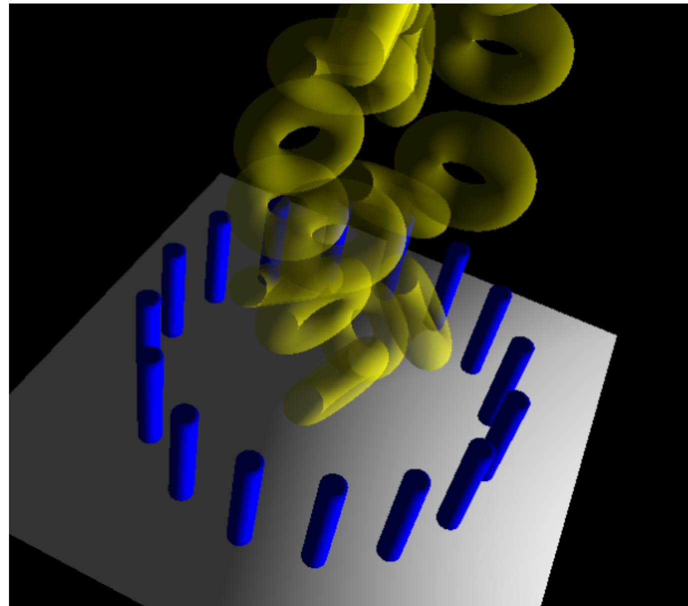
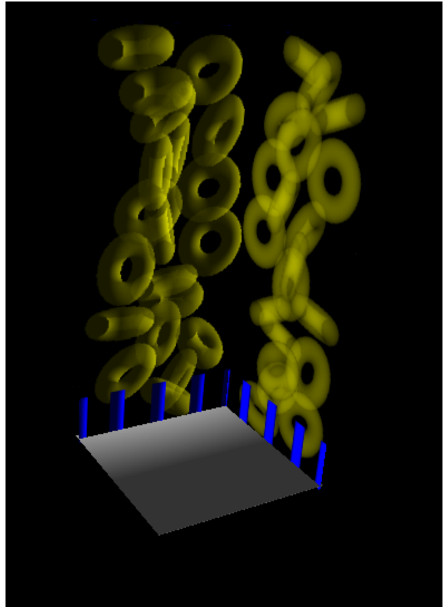



FIG. 4.13 – Rendu de transparence basé sur notre algorithme de rasterization.

est lu pour en déduire les informations de couleur. On peut également changer l'étape de rastérisation pour calculer directement la couleur sur chaque pixel. Or, comme les étapes de pré-traitement construisent la liste des triangles dans un ordre de profondeur approximativement trié, il est possible d'accumuler la couleur directement en mémoire partagée, sans jamais avoir besoin d'écrire dans la mémoire globale. Si en revanche les triangles ne sont pas correctement triés, nous utilisons les registres comme décrit précédement, pour ré-ordonner rapidement les layers. Si en revanche ces registres ne suffisent pas, à ré-ordonner correctement les layers, alors cet algorithme peut introduire des erreurs, mais ces cas sont rares et peu perceptibles. La qualité du rendu obtenue par cette méthode est comparable avec les méthodes basées sur des API graphiques (voir figure 4.13).

On peut également prendre en compte la lumière. Une première implémentation est d'utiliser un éclairage de type *flat shading*, qui consiste à calculer une normale par triangle. On peut alors obtenir le coefficient de lumière en calculant le produit vectoriel entre la normale du triangle $\vec{\boldsymbol{n}}$ et le vecteur lumière $\vec{l}$. La couleur d'un pixel est finalement déterminée par :

$$color_{pixel} = color_{objet} * \alpha_{pixel} * \alpha_{objet} * (\vec{\boldsymbol{n}}.\vec{l}) \tag{4.11}$$

Pour améliorer le rendu, on peut utiliser le *phong shading* qui interpole les normales en chacun des sommets. Ce calcul est en réalité directement récupéré d'un composant permettant d'afficher les objets directement sur GPU. On peut ensuite interpoler la normale au pixel en utilisant les coefficients barycentriques α et β calculés pendant la rastérisation.

Enfin pour pouvoir réaliser un rendu en $3D$, il est nécessaire de faire le rendu en fonction de la position de la caméra. La difficulté est alors que la rastérisation ne se fait plus selon les axes alignés du repère global. Pour se ramener à un problème équivalent, on utilise une matrice de projection, afin de transformer les coordonnées des points dans le repère de la caméra. Il reste enfin à dessiner à l'écran, la texture calculée.

4.3.3 Résultats

Nous avons implémenté une réponse par pénalité similaire à ce qui est fait dans Faure et al. (2008), ce qui nous permet d'utiliser cette méthode dans une simulation d'objets déformables. Nous avons reproduit une simulation similaire à celle qui a été présentée dans Irving et al. (2007). Dans le papier initial, la simulation nécessitait plusieurs minutes de calcul pour chaque image, et nous sommes capables en utilisant les divers optimisations sur GPU, de la simuler en temps réel (voir figure 4.14). Nous avons comparé les performances de deux simulations, l'une comprenant 10 corps, et l'autre 20. La collision est réalisée avec notre technique de rastérisation, en activant les auto-collisions. Pour la déformation nous avons utilisé les versions CPU et GPU des modèles co-rotationnels, et du modèle masse ressort.

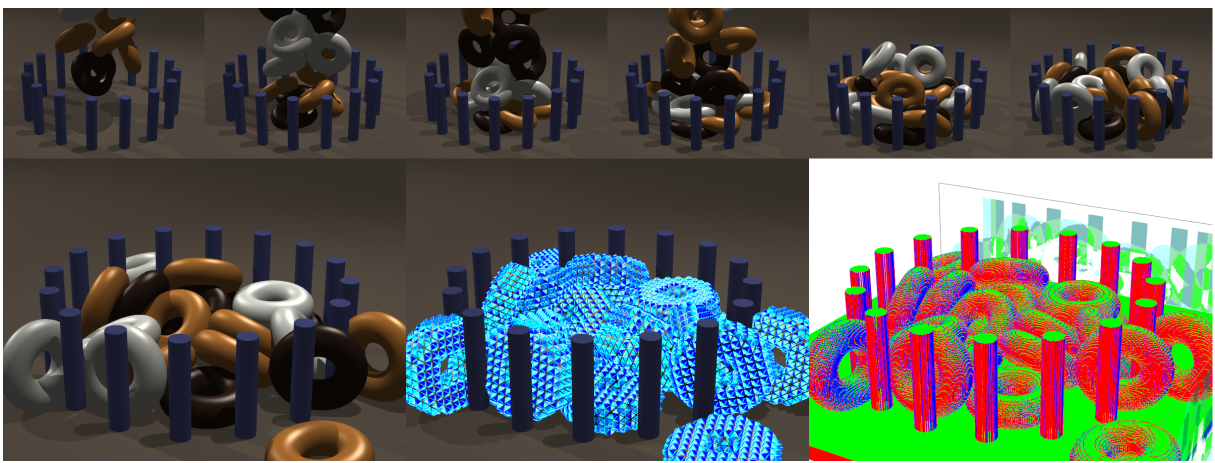


FIG. 4.14 – Simulation d'un empilement de tores déformables, soumis à de multiples collisions.

Les chiffres de performances sont répertoriés dans le tableau 4.1. On peut alors faire plusieurs constatations. La première est que le temps de la détection des collisions est très inférieur au temps de calcul de la mécanique. Même quand la mécanique est calculée sur GPU, la détection de collisions représente environ 7% du temps de calcul d'un pas de temps. Par ailleurs, son calcul est directement proportionnel au nombre de triangle, mais il n'est pas influencé par le modèle déformable.

L'étape de rastérisation est la plus coûteuse, puisqu'elle nécessite environ 0.7 millisecondes pour chaque LDI (dans la version a 20 tores). Le comptage des triangles par cellule ainsi que l'étape de tri, prennent chacune 0.3 millisecondes pour chacun des axes. Le calcul de la réponse nécessite 0.9 millisecondes par pas de temps.

Pour terminer, on peut souligner l'efficacité de la parallélisation GPU, en particulier pour le modèle co-rotationnel. En effet, la méthode de réponse étant basée sur des pénalités, cela implique de résoudre à chaque pas de temps, un grand système d'équations qui comprend l'intégralité des objets déformables. En utilisant la version GPU, on obtient une accélération d'environ $4,4\times$ sur l'application totale, ce qui lui permet de maintenir un rafraîchissement interactif des images, contrairement à la même simulation calculée sur le processeur.

		Collisions	Deformations	TOTAL	FPS
10 Tores 56332 Triangles	Torus Spring	4.80	89.10	94.78	10.55
	Torus Tetra	4.83	270.61	276.34	3.62
	Torus Spring GPU	4.48	57.56	62.86	15.91
	Torus Tetra GPU	4.50	57.97	63.29	15.80
20 Tores 97292 Triangles	Torus Spring	6.75	203.14	211.00	4.74
	Torus Tetra	6.81	568.48	576.43	1.73
	Torus Spring GPU	6.28	122.38	129.69	7.71
	Torus Tetra GPU	6.37	124.23	131.63	7.60

Tab. 4.1 – Temps en millisecondes des différentes parties de la simulation.

4.4 Extension aux contacts volumiques

Dans cette section, nous présentons une nouvelle façon de gérer les contacts à partir des volumes d'intersection et des dérivés de volume calculés dans la section précédente. Pour cela nous reformulons les conditions de Signorini, pour obtenir un problème équivalent dans lequel les forces s'exercent en fonction des pressions sur les contacts et du volume d'interpénétration. De plus, nous montrerons comment contraindre le mouvement tangentiel, afin de poser l'équation d'un contact frottant entre les objets. Notre méthode permet d'obtenir une formulation d'équations beaucoup plus simples que celles des méthodes traditionnelles et peut être utilisée sur des géométries complexes, avec la même efficacité. Le contact entre deux objets peut être simplifié à une équation de contrainte unilatérale unique, ou de façon exacte en définissant la résolution d'une grille, choisie arbitrairement et indépendamment de la géométrie des objets. On peut alors facilement choisir le nombre de contacts (ainsi que la précision) souhaitée (voir figure 4.15).

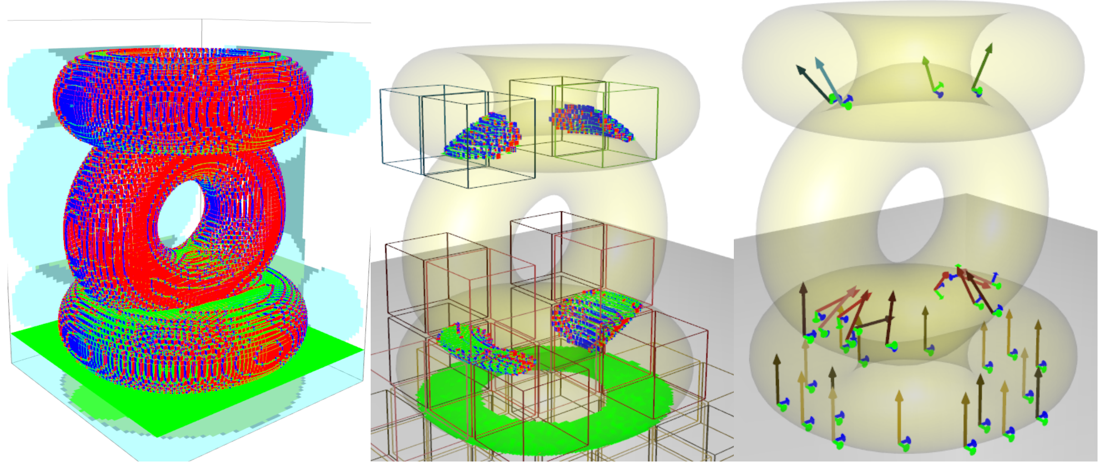


Fig. 4.15 – Une pile de tores, rastérisée (gauche), avec les volumes d'intersection (centre), desquelles on déduit les normales de contraintes (droite).

4.4.1 Contraintes basées sur des volumes

Nous commençons tout d'abord par étendre la notion de contrainte qui est habituellement formulée en distance, à une contrainte formulée en volume. Comme indiqué dans la première partie de ce chapitre, une manière classique pour traiter un contact entre deux objets est de construire un LCP entre les distances de séparation ψ et les forces de réaction $\boldsymbol{\lambda}$. Par souci de simplicité, nous considérons un système composé de deux objets en collision, où le volume d'intersection $\mathcal{V}(\boldsymbol{p})$ est calculé comme décrit dans la section 4.3. Nous montrons ici comment construire un problème équivalent basé sur le volume d'interpénétration.

Équivalence entre distance et volume

Pour rester le plus générique possible, nous présentons notre méthode avec une résolution au niveau des vitesses[8], et nous montrerons comment ajouter une étape de post-stabilisation pour corriger l'interpénétration. La solution du système de contrainte doit respecter le principe des travaux virtuels, c'est-à-dire que pour satisfaire la contrainte elle ne doit pas ajouter ni supprimer de l'énergie du système. Puisque nous résolvons le système au niveau des vitesses, cela peut être écrit $\boldsymbol{\lambda}_i \cdot \mathbf{J}_i(\boldsymbol{v}_0 + \Delta\boldsymbol{v}) = 0$, où $\mathbf{J}_i = \frac{\partial \psi_i}{\partial \boldsymbol{p}}$ est le gradient de la distance de séparation. Toutefois, on peut associer une petite surface $\mathcal{A}_i$ à chaque contact i, et observer que :

$$\boldsymbol{\lambda}_i \frac{1}{\mathcal{A}_i} \cdot \mathcal{A}_i \mathbf{J}_i(\boldsymbol{v}_0 + \Delta\boldsymbol{v}) = 0 \tag{4.12}$$

On pose $\boldsymbol{\rho}_i \equiv \boldsymbol{\lambda}_i \frac{1}{\mathcal{A}_i}$ qui est homogène à une pression, et $\frac{\partial \mathcal{V}_i}{\partial \boldsymbol{p}} \equiv \mathcal{A}_i \mathbf{J}_i$ qui est le gradient du volume. Ainsi, nous obtenons des conditions de complémentarité équivalentes. On peut alors reformuler la condition de Signorini en pression/volume

$$0 \leq \boldsymbol{\rho}_i \perp \frac{\partial \mathcal{V}_i}{\partial \boldsymbol{p}}(\boldsymbol{v}_0 + \Delta\boldsymbol{v}) \geq 0 \tag{4.13}$$

Cette formulation met en évidence que la pression appliquée sur les points en intersection va résoudre l'interpénétration des objets. Les forces s'exerçant sur les degrés de liberté en réponse à une intersection, agissent dans le sens du gradient, qui peut être considéré comme la somme de $-\mathcal{A}_i \mathbf{J}_i$ des contraintes actives. Pendant la résolution, on peut alors utiliser le volume d'intersection $\mathcal{V}(\boldsymbol{p})$ pour identifier les contraintes actives. S'il est négatif, alors il faut appliquer une force de réponse, et s'il est positif, les objets sont distants.

Avec cette équation, nous utilisons le volume total d'intersection pour contraindre le système. Un seul multiplicateur de Lagrange $\boldsymbol{\rho}$ est alors généré pour calculer une pression uniforme agissant à la surface de contact. Nous appelons cette approximation une contrainte de contact *mono-volume*. Lorsqu'il s'agit de résoudre nos contraintes en fonction du volume, des propriétés physiques importantes sont garanties :
– Le volume d'interpénétration ne peut pas augmenter.

[8] Cependant, notre méthode peut également être utilisée avec une formulation au niveau des positions, c'est ce que nous ferons dans le chapitre suivant.

– La pression ne peut agir que pour pousser des objets en dehors des interpénétrations.
– La pression agissant sur la surface de contact est non nulle, si et seulement si le volume d'interpénétration ne diminue pas.

Néanmoins, l'approche mono-volume calcule une pression uniforme sur toute la zone de contact dans son ensemble, ce qui n'est pas toujours réaliste. Cette limitation est traitée dans la section 4.4.3, qui propose une approche multi-volumes.

Stabilisation Une résolution en vitesse ne permet pas de garantir une configuration sans interpénétration à la fin du pas de temps, mais seulement que le volume ne va pas augmenter. Pour remédier à ce problème, on peut ajouter une étape de post stabilisation, afin de corriger également les positions. Cela consiste à réaliser de multiples itérations de l'algorithme de Newton-Raphson (voir l'algorithme 6), afin d'annuler complètement le volume d'intersection. La boucle est répétée jusqu'à ce que la précision souhaitée soit obtenue, ou lorsqu'un nombre maximal d'itérations est atteint.

1 Input : current state $\boldsymbol{p}_0, \boldsymbol{v}_0$
2 Output : next state $\boldsymbol{p}_{0+h}, \boldsymbol{v}_{0+h}$

1: $\Delta \boldsymbol{v} \leftarrow$ LCPsolve velocity update with friction
2: $\boldsymbol{v}_{0+h} \leftarrow \boldsymbol{v}_0 + \Delta \boldsymbol{v}$
3: $\boldsymbol{p}_{0+h} \leftarrow \boldsymbol{p}_0 + h\boldsymbol{v}_{0+h}$
4: Compute LDI : intersection $\mathcal{V}(\boldsymbol{p}_{0+h})$ and Jacobian ;
5: **tantque** $\mathcal{V}(\boldsymbol{p}_{0+h}) > \epsilon_{\mathcal{V}}$ **faire**
6: $\Delta \boldsymbol{p} \leftarrow$ LCPsolve position stabilization
7: $\boldsymbol{p}_{0+h} \leftarrow \boldsymbol{p}_{0+h} + \Delta \boldsymbol{p}$
8: Compute LDI : intersection$\mathcal{V}(\boldsymbol{p}_{0+h})$ and Jacobian ;
9: **fin tantque**

Algorithme 6 : Animation Loop

FIG. 4.16 – La réponse en position oblige à maintenir un volume non nul pour produire un contact continu.

Avec la post-stabilisation au niveau des positions, on peut garantir qu'aucune interpénétration subsistera après la résolution. Cependant, cela peut introduire des instabilités, dans les cas où de très petits volumes sont détectés. En effet, nous ne pouvons créer nos contraintes de volume que lorsqu'un volume d'intersection est non nul. C'est pourquoi, même si le volume d'intersection devrait en théorie rester à zéro, nous autorisons une petite interpénétration $\epsilon_{\mathcal{V}}$ pour permettre un contact continu (voir figure 4.16). Ainsi, nous modifions légèrement les conditions de complémentarité comme suit :

$$-\epsilon_{\mathcal{V}} \leq -\mathcal{V}(\boldsymbol{p}_0 + \Delta \boldsymbol{p}) \perp \boldsymbol{\rho} \geq 0. \quad (4.14)$$

En pratique, nous maintenons une couche d'interpénétration qui a une épaisseur d'environ un demi-pixel. Pour cela, nous estimons la surface du contact en additionnant les aires des contacts actifs. Nous pouvons alors en déduire la profondeur moyenne en intersection qu'il est nécessaire d'ignorer sur chaque contact. En maintenant un volume $\epsilon_{\mathcal{V}}$, on obtient un contact stable et continu.

4.4.2 Extension aux contacts frottants

Les forces de frottement agissent en opposition à la vitesse relative, dans le plan tangent à la normale. Dans le modèle standard de contact, une force de frottement est appliquée au point de contact. Nous avons besoin de modifier cette définition, parce que notre force est en fait une pression appliquée sur la surface d'un contact.

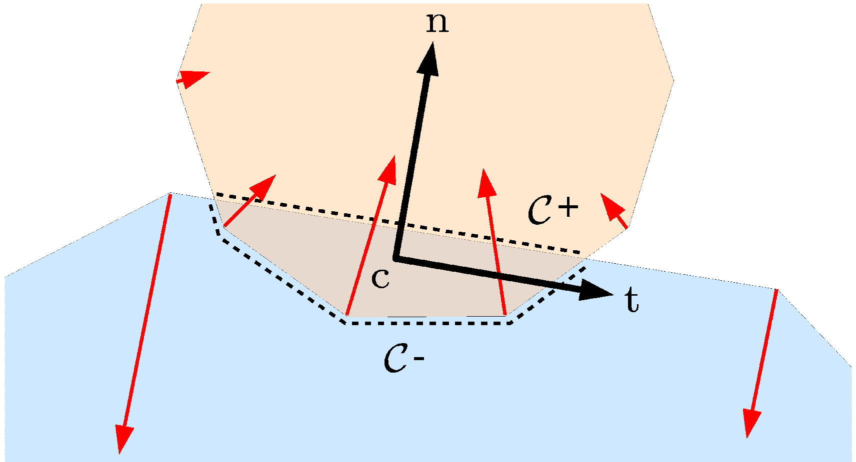


FIG. 4.17 – Deux objets en intersection et le modèle de contact associé. Le gradient d'intersection apparaît en lignes rouges sur chaque vertex. Le repère utilisé pour le frottement est construit par rapport à l'estimation de la direction de la normale $\vec{n}$.

La somme des forces de répulsion appliquées aux sommets d'un objet, nous donne la direction approximative de la normale au contact $\vec{\boldsymbol{n}}$. Nous calculons ensuite deux vecteurs unitaires orthogonaux $\vec{\boldsymbol{t}_1}$ et $\vec{\boldsymbol{t}_2}$ couvrant le plan tangent à l'aide d'une orthogonalisation de Gram-Schmidt. Cela nous donne une base dans laquelle on peut exprimer le frottement.

Pour définir l'équivalent des vitesses relatives appliquées au point de contact, nous avons besoin pour calculer une différence pondérée des vitesses sur les surfaces $\mathcal{C}^+$ et $\mathcal{C}^-$ (voir figure 4.17). En effet, le poids de chaque sommet doit tenir compte de la surface de contact associée. On peut évaluer cette surface en projetant le gradient de volume sur la normale du contact, et nous définissons :

$$\boldsymbol{v}_{\text{rel}} \equiv \sum_k \mathcal{A}_k \boldsymbol{v}_k, \quad \text{with } \mathcal{A}_k = \vec{\boldsymbol{n}} \cdot \frac{\partial \mathcal{V}}{\partial \boldsymbol{p}_k}, \tag{4.15}$$

où $\boldsymbol{v}_k$ est la vitesse sur le sommet de k, et $\mathcal{A}_k$ est l'aire associée au sommet k projeté sur le plan tangent. Le signe de $\mathcal{A}_k$ dépend de l'orientation de la surface associée au sommet par rapport à la normale $\vec{\boldsymbol{n}}$. Cela garantit que $\boldsymbol{v}_{\text{rel}}$ sera nulle lorsque les deux objets auront la même vitesse[9].

Les forces de contact visqueuses sont les plus simples à formuler, puisqu'elles peuvent être

[9]Notons que $\boldsymbol{v}_{\text{rel}}$ a des unités de vitesse par aire, donc le produit de $\boldsymbol{v}_{\text{rel}}$ avec notre pression de contact (force par unité de surface) donne une puissance (la vitesse par la force), comme dans le modèle de contact standard.

appliquées proportionnellement à la vitesse :

$$\vec{f_k} = -\nu \, \boldsymbol{v}_{\mathrm{rel}} \, \mathcal{A}_k \tag{4.16}$$

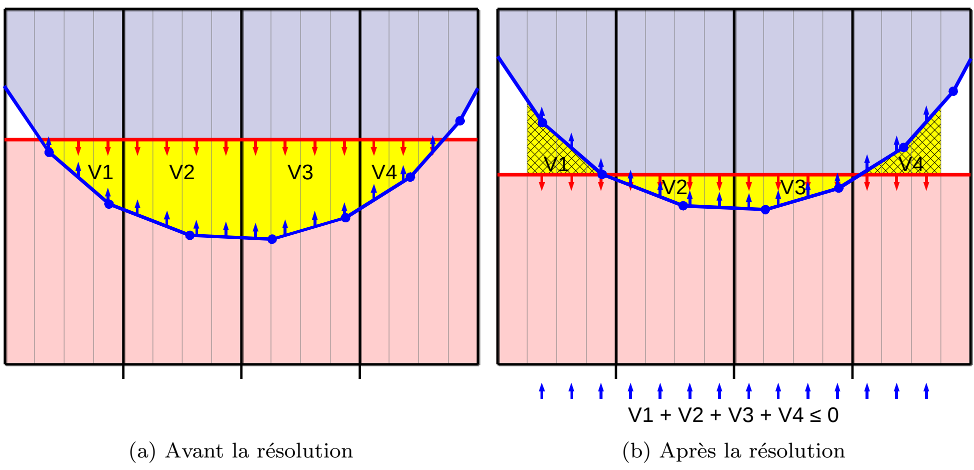
Pour appliquer le frottement de Coulomb, nous devons limiter les projections de la vitesse relative aux vecteurs de base du plan tangent $\vec{\boldsymbol{t}_1}$ et $\vec{\boldsymbol{t}_2}$. Il suffit pour cela de construire la base $\mathbf{T}$ de l'équation (4.5), dans le repère tangent du contact $\vec{\boldsymbol{t}_1}$ et $\vec{\boldsymbol{t}_2}$ que nous venons de calculer :

$$\mathbf{T} = \begin{pmatrix} \mathcal{A}_1 \vec{\boldsymbol{t}_1}^T & \cdots & \mathcal{A}_n \vec{\boldsymbol{t}_1}^T \\ \mathcal{A}_1 \vec{\boldsymbol{t}_2}^T & \cdots & \mathcal{A}_n \vec{\boldsymbol{t}_2}^T \\ -\mathcal{A}_1 \vec{\boldsymbol{t}_1}^T & \cdots & -\mathcal{A}_n \vec{\boldsymbol{t}_1}^T \\ -\mathcal{A}_1 \vec{\boldsymbol{t}_2}^T & \cdots & -\mathcal{A}_n \vec{\boldsymbol{t}_2}^T \end{pmatrix} \frac{\partial \boldsymbol{p}}{\partial \boldsymbol{v}} \tag{4.17}$$

On peut alors calculer la vitesse appropriée par rapport à notre modèle de contact volumique, et fournir une base pour appliquer des forces de friction sur les degrés de liberté.

4.4.3 Extension au modèle Multi-Volumes

Jusqu'à présent, nous avons montré que la minimisation du volume d'intersection en utilisant le modèle mono-volume génère une seule équation pour la répulsion des objets. Ceci est indépendant de leur résolution géométrique, ce qui conduit à des simplifications importantes par rapport aux méthodes traditionnelles basées sur la distance. Cependant, une limitation de cette approche est que l'équation de contact est valable pour toute la zone de contact, ce qui entraîne un comportement unique pour la réponse.



(a) Avant la résolution (b) Après la résolution

Fig. 4.18 – Erreur de l'approche mono-volumes

On peut voir avec l'exemple de la figure 4.18, que si on ne crée qu'un seul multiplicateur de Lagrange pour toute la zone en contact (en jaune sur la figure 4.18a), alors la résolution de cette équation unique peut conduire à une trop grande simplification (voir figure 4.18b). En effet, comme on ne peut pas redéfinir la zone de contact active, il est

possible qu'une partie du volume initialement en intersection devienne "négatif" pendant l'étape de résolution. Ce volume négatif peut alors compenser le volume d'intersection, ce qui peut amener à une configuration dans laquelle l'équation (4.13) sera vérifiée, alors que géométriquement, les objets présenteront encore des intersections. En revanche, on s'attend à ce que cette intersection diminue dans les pas de temps suivants, car une nouvelle détection sera réalisée avec le volume restant, ce qui devrait repousser d'avantage les objets.

Pour améliorer la précision de la méthode, nous proposons de diviser le volume d'intersection dans plusieurs régions avec des équations de contacts indépendants. Nous appelons cela le modèle *multi-volumes*. On divise alors le volume d'intersection à l'aide d'une grille régulière alignée avec les directions des LDI (voir figure 4.19). La résolution est définie par un nombre de pixels, qui seront regroupés dans chaque cellule[10]. En réalité, le nombre de pixels constituant une cellule est un paramètre que l'utilisateur peut contrôler pour affiner facilement le rapport entre la précision de la réponse, et le temps de calcul.

Pour définir des équations de contacts, nous avons besoin de calculer le gradient de volume de chaque cellule dans la direction du regard. La limite du volume d'intersection dans une cellule peut être composée de quatre types de surfaces qui sont la surface de l'objet supérieure $\mathcal{S}^+$, la paroi supérieure de la cellule $\mathcal{W}^+$, la surface de l'objet inférieur $\mathcal{S}^-$ et la paroi de la cellule inférieure $\mathcal{W}^-$. Les contributions des pixels situés sur la surface des objets ont déjà été décrites dans la section 4.3. La question qui reste est de savoir comment prendre en compte les pixels des parois cellulaires.

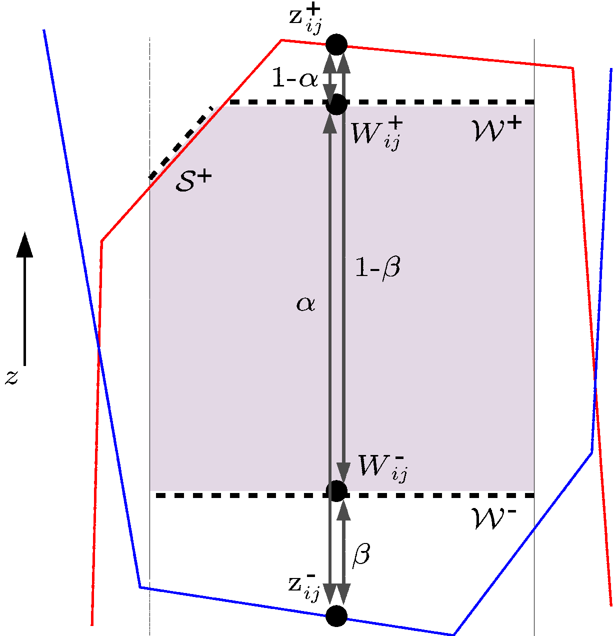


FIG. 4.19 – Approche multi-volumes. Exemple d'une cellule placée dans un volume d'intersection.

[10]On utilise ici le terme cellule pour définir un groupement de pixels qui va créer un volume indépendant. Cette notion est complètement indépendante des cellules que nous utilisions dans le rasterizer pour regrouper les calculs parallèles

Si les parois cellulaires sont considérées comme fixes, alors les pixels ne contribuent pas au gradient de volume, puisqu'ils sont indépendants des sommets des objets. Par exemple, la cellule de la figure 4.19 ne comprend pas la surface de l'objet bleu. Pourtant, la réponse à la collision pour cette cellule devra appliquer une force à la fois sur l'objet rouge et l'objet bleu. Pour chaque pixel de la paroi cellulaire, on calcule son coefficient barycentrique à l'intérieur du volume d'intersection dans le sens d'observation. Nous faisons cela de telle sorte que la profondeur d'un pixel de la paroi cellulaire haute $\mathcal{W}^+$ soit donnée par :

$$W_{ij}^{+} = \alpha_{ij} \mathrm{z}_{ij}^{+} + (1 - \alpha_{ij}) \mathrm{z}_{ij}^{-}, \tag{4.18}$$

alors que la profondeur d'un pixel de la paroi cellulaire basse $\mathcal{W}^-$ est donnée par :

$$W_{ij}^{-} = \beta_{ij} \mathrm{z}_{ij}^{+} + (1 - \beta_{ij}) \mathrm{z}_{ij}^{-}. \tag{4.19}$$

Un exemple de ces coefficients barycentriques est illustré à la figure 4.19. Dans chaque cellule, le gradient du volume en intersection est donc :

$$\begin{aligned}
\frac{\partial \mathcal{V}}{\partial \boldsymbol{p}_k^z} &= a \sum_{(i,j) \in \mathcal{S}_z^+} \frac{\partial \mathrm{z}_{ij}^+}{\partial \boldsymbol{p}_k^z} - a \sum_{(i,j) \in \mathcal{S}_z^-} \frac{\partial \mathrm{z}_{ij}^-}{\partial \boldsymbol{p}_k^z} \\
&\quad + a \sum_{(i,j) \in \mathcal{W}_z^+} \alpha_{ij} \frac{\partial \mathrm{z}_{ij}^+}{\partial \boldsymbol{p}_k^z} + (1 - \alpha_{ij}) \frac{\partial \mathrm{z}_{ij}^-}{\partial \boldsymbol{p}_k^z} \\
&\quad - a \sum_{(i,j) \in \mathcal{W}_z^-} \beta_{ij} \frac{\partial \mathrm{z}_{ij}^+}{\partial \boldsymbol{p}_k^z} + (1 - \beta_{ij}) \frac{\partial \mathrm{z}_{ij}^-}{\partial \boldsymbol{p}_k^z}
\end{aligned} \tag{4.20}$$

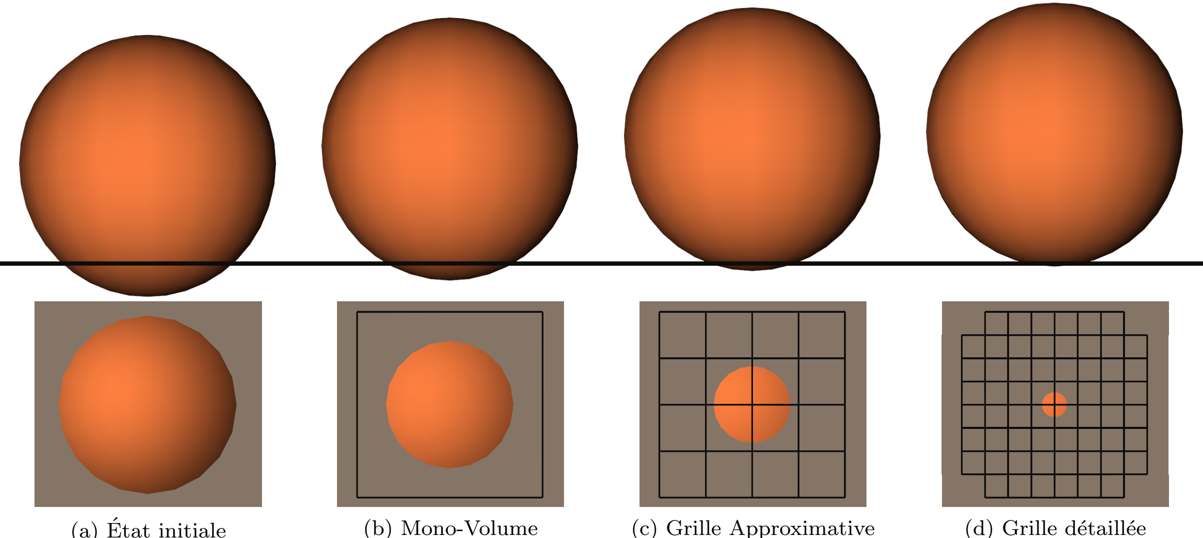
Une fois que le gradient est calculé, les équations de contact de la cellule sont simples et peuvent être mises en place de la même manière que présentées dans la section 4.4.1. Il suffit pour cela d'accumuler un nouveau volume indépendant pour chaque sous-volume identifié dans la grille.

4.4.4 Résultats

Évaluation de la méthode

Nous avons tout d'abord cherché à évaluer l'influence de l'approche multi-volumes sur la qualité de la réponse aux contacts, ainsi que sur le comportement du frottement.

Le découpage en multi-volumes, permet un meilleur respect des conditions de Signorini, comme on peut le voir sur la figure 4.20. La quantité d'interpénétration entre les objets est de plus en plus faible à mesure que l'on découpe le volume d'intersection en sous-volumes. En effet, en utilisant l'approche multi-volumes, les contacts les plus extérieurs sont progressivement éliminés au cours du calcul de la résolution, et n'influencent plus les contacts actifs. Ainsi, la résolution du LCP, fournit une force qui corrige presque intégralement le contact en un seul pas de temps.



(a) État initiale (b) Mono-Volume (c) Grille Approximative (d) Grille détaillée

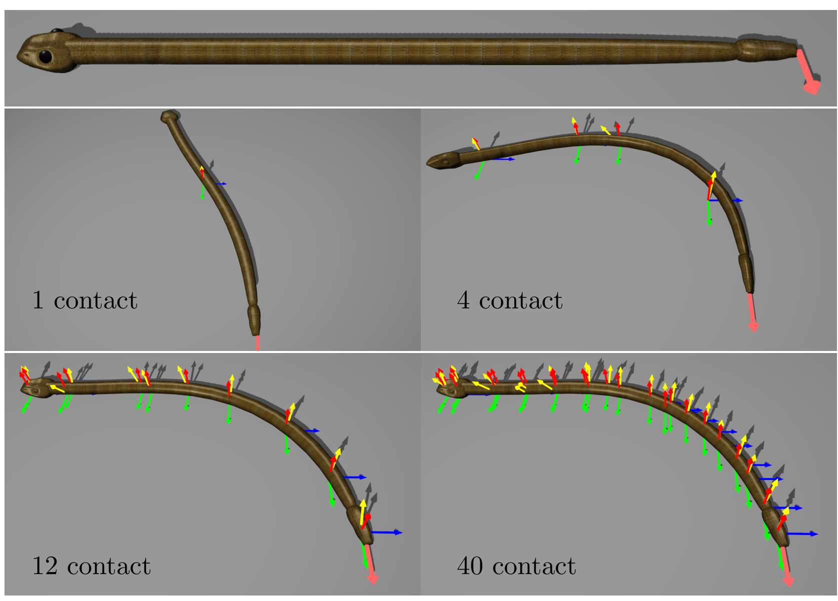
FIG. 4.20 – Intersection après la résolution des contraintes, pour différentes résolutions de la grille. Les figures du bas présentent un vue de dessous alors que les figures du haut présentent une vue de profile. En augmentant le nombre de multi-volumes, on obtient une solution plus précise.

Par ailleurs, le découpage en multi-volumes peut également améliorer le comportement du frottement dans la simulation. Nous avons validé ce modèle sur l'exemple du serpent illustré dans la figure 4.21. Avec le modèle mono-volume, le serpent tourne autour du centre de contact situé approximativement au milieu de son corps, alors qu'il évolue doucement vers une forme courbée et stable en utilisant le modèle multi-volumes. On peut voir dans le tableau 4.21 que plus on augmente le nombre de contraintes, et plus on se rapproche du comportement des méthodes traditionnelles formulées en distance. Cette propriété est importante, car elle met en évidence le fait que l'utilisateur peut facilement affiner le rapport entre la précision de la méthode et le temps de calcul, de façon indépendante de la géométrie des objets simulés. Enfin, quand un objet est coupé comme le montre la figure 4.23d, le modèle multi-volumes permet le traitement indépendant de chacune des parties.

Mesure des performances

Nous avons appliqué notre méthode à une variété d'exemples, comprenant aussi bien des objets rigides et déformables. Pour les objets déformables, nous utilisons le modèle co-rotationnel Nesme et al. (2005b). Nous avons effectué des simulations sur un CPU Intel Core i7 975 avec une Nvidia GeForce GTX 285.

Les deux armadillos de la figure 4.22 sont rigides, et constitués de 345 944 triangles. Avec notre méthode, ils sont simulés à 41 fps (sans le rendu), avec une moyenne de 7 contacts, (maximum de 16 contacts). Nous avons comparé ces performances à une version pré-calculée d'un champ de distance, qui sont très efficaces pour les objets rigides. Les résultats montrent que la simulation est plus rapide avec les champs de distance, puisqu'ils permettent d'atteindre un taux de rafraîchissement jusqu'à 57 images par seconde. Toutefois, ils génèrent des équations de contacts beaucoup plus complexes, (typiquement 200



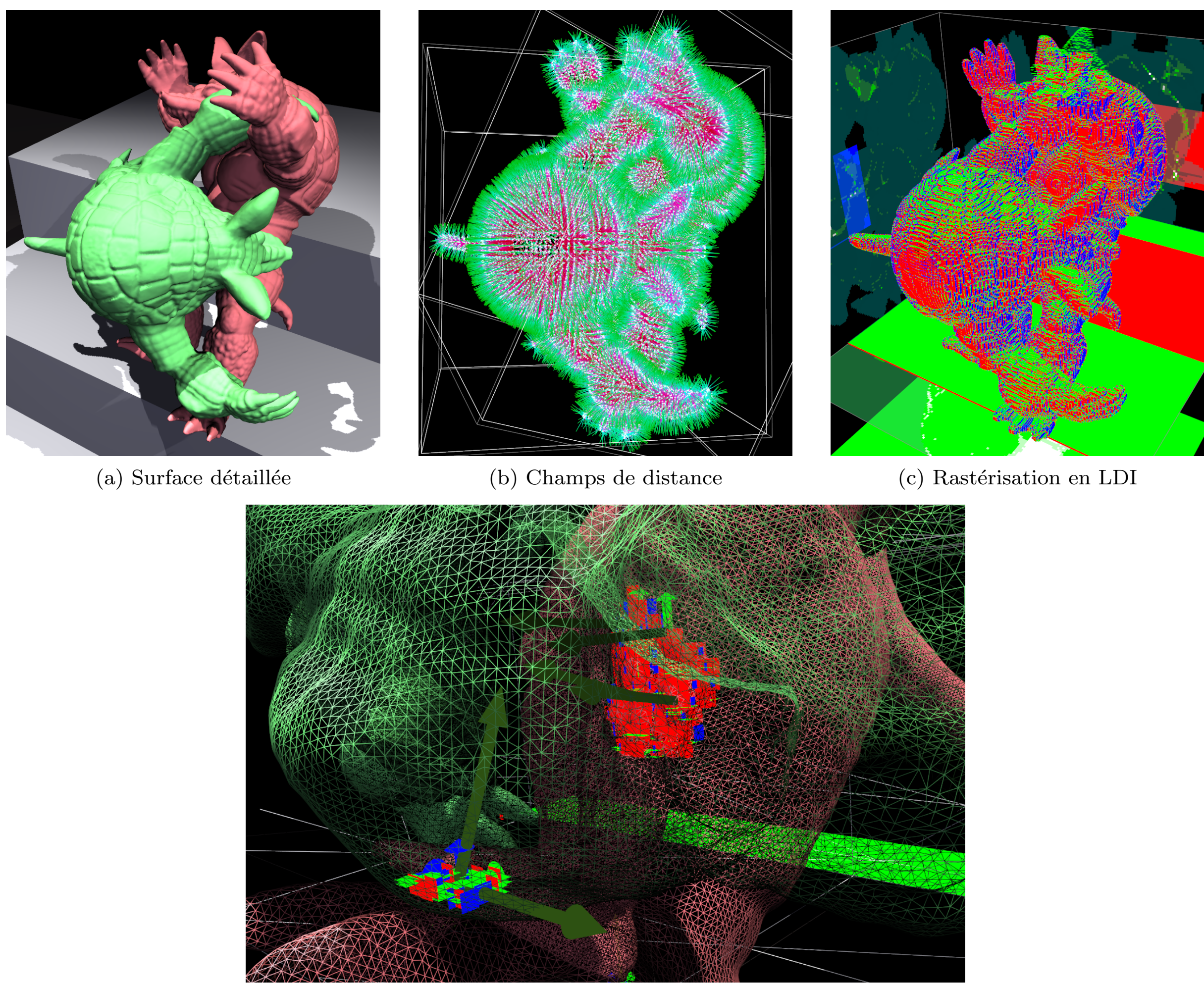
Taille de cellules (pixels)	Contacts	Erreur moyenne	Erreur maximale
–	1	26.21 %	61.30 %
250	4	3.47 %	9.94 %
100	7	3.85 %	6.58 %
90	12	0.40 %	0.83 %
50	20	0.25 %	0.46 %
22	40	0.18 %	0.35 %

FIG. 4.21 – Influence du modèle multi-volumes sur la friction. Les flèches rouges, vertes et bleues indiquent les repères des contacts, tandis que les flèches jaunes indiquent les forces. Le tableau montre l'erreur mesurée entre notre méthode avec des tailles de cellules différentes, par rapport à une formulation en distance sur tous les points de contacts (qui sert ici de référence)

contacts dans cet exemple, soit 600 équations en ajoutant du frottements). Ceci implique que beaucoup de temps est passé dans la résolution du LCP, et ce problème devrait s'accentuer à mesure que l'on augmentait la résolution de la surface des objets. Notre méthode ne souffre pas de ce problème, puisqu'elle est insensible à la résolution du maillage.

Les avantages de notre méthode sont évidents lorsque les champs de distances pré-calculés ne sont pas disponibles. Par exemple, pour simuler des objets déformables, ou encore pour les changements topologiques (voir figure 4.23d). De plus, une caractéristique intéressante de notre approche est que les auto-intersections peuvent être calculées sans surcoût important. Ceci est illustré dans la figure 4.23c, où 3 poulpes sont simulés de façon interactive, malgré les nombreux contacts qui apparaissent le long des tentacules. La main qui tient la balle déformable dans la figure 4.23b est simulée à 54 fps, avec 10 contacts, tandis qu'une méthode basée sur des proximités fonctionne à 23 fps en raison de la complexité de la détection de collision et d'un plus grand nombre de contacts (jusqu'à 70 dans cet exemple).

La scène médicale présentée dans la figure 4.23a est basée sur des données réelles. En raison du bruit inévitable sur les données (dû aux approximations dans le processus de reconstruction), la surface des organes présentent de multiples intersections dans leur configuration initiale. Certaines méthodes de détection de collision ou de proximité exigent



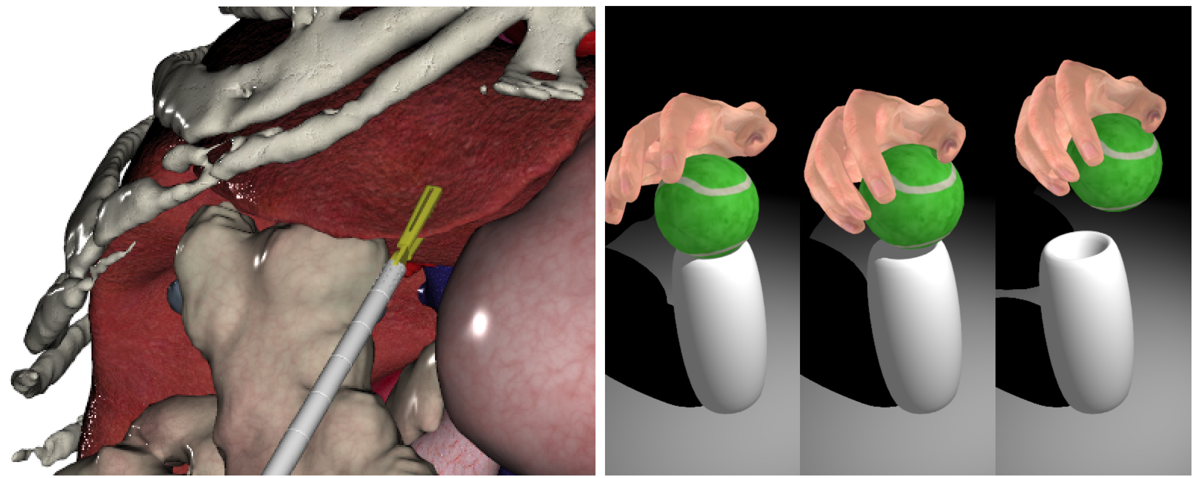
(a) Surface détaillée (b) Champs de distance (c) Rastérisation en LDI

(d) Volumes de contact sur le bras de l'Armadillo

FIG. 4.22 – Comparaison avec les champs de distance. Notre méthode est capable de gérer des objets très détaillés, sans aucun pré-calcul.

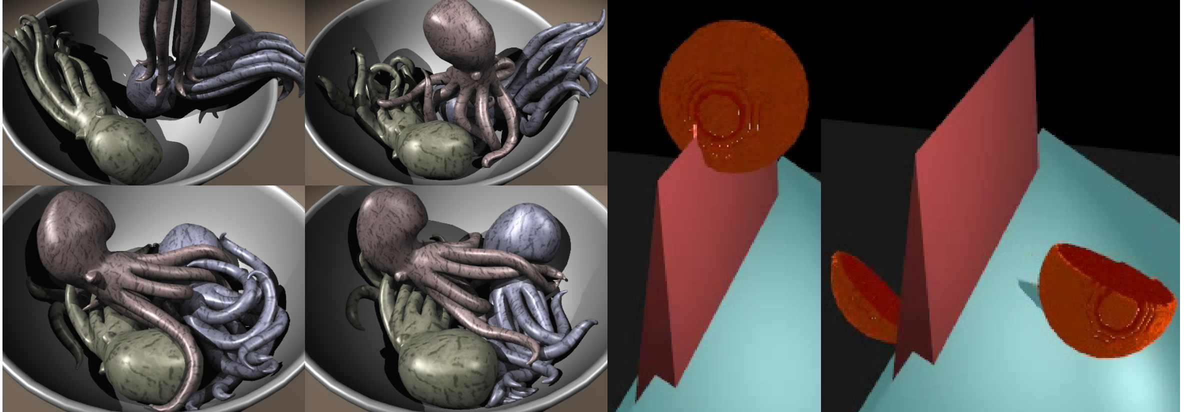
de ne pas avoir d'interpénétration au départ de la simulation. Au contraire, notre méthode est robuste aux intersections en profondeur, et ne présente aucun problème pour séparer les organes dès le premier pas de temps. De plus, la saisie de l'organe avec la pince est difficile, car cela repose sur une haute pression du contact et sur les forces de frottement. Avec nos contraintes volumiques, l'outil parvient à maintenir l'organe, puis glisse finalement lorsqu'on tire trop fort. Les forces de contact restent alors dans le cône de frottement au cours de la première partie du mouvement, puis se placent sur le bord du cône par la suite.

Notons que la géométrie de l'outil est mince avec des dents très fines. Il peut être manipulé par notre méthode, mais nécessite de très petits pixels, ce qui est très coûteux, car les LDI sont actuellement pixellisés avec une précision uniforme sur l'ensemble de la scène. Bien que la méthode puisse être étendue pour adapter localement la résolution, nous avons remplacé le mesh de collision pour cet outil avec une version un peu plus simple et plus



(a) Simulation Médicale avec du frottement

(b) Saisie avec un frottement stable



(c) Multiples collisions et auto-collisions

(d) Détection des collisions dynamiques pendant la découpe

FIG. 4.23 – Démonstration de plusieurs scénarios difficiles pour la modélisation de contact qui sont traités efficacement par la méthode proposée.

épaisse. La scène comprend 550 000 triangles, 9 500 degrés de liberté, et tourne à 7 images par seconde, y compris le rendu.

Le tableau 4.2 présente la complexité et les résultats mesurés pour les simulations présentées. Les colonnes *détection de collisions* et *formulation des Contacts* représentent le temps nécessaire pour calculer le LDI et évaluer les volumes et les gradients d'intersections (étapes 4 et 8 dans l'algorithme 6). La colonne *déformation* représente le mouvement libre (voir le chapitre précédent), tandis que la colonne *Résolution des contraintes* représente la correction des vitesses et des positions.

Une limitation évidente de notre approche est que l'interaction des objets doit avoir un volume, donc les objets très minces tels que les tissus ne sont pas pris en charge. De même, il est possible que des petits objets en mouvement rapide puissent passer l'un à travers l'autre pendant le mouvement. Pour ces deux cas, les méthodes basées sur la détection de collision continue sont plus appropriées.

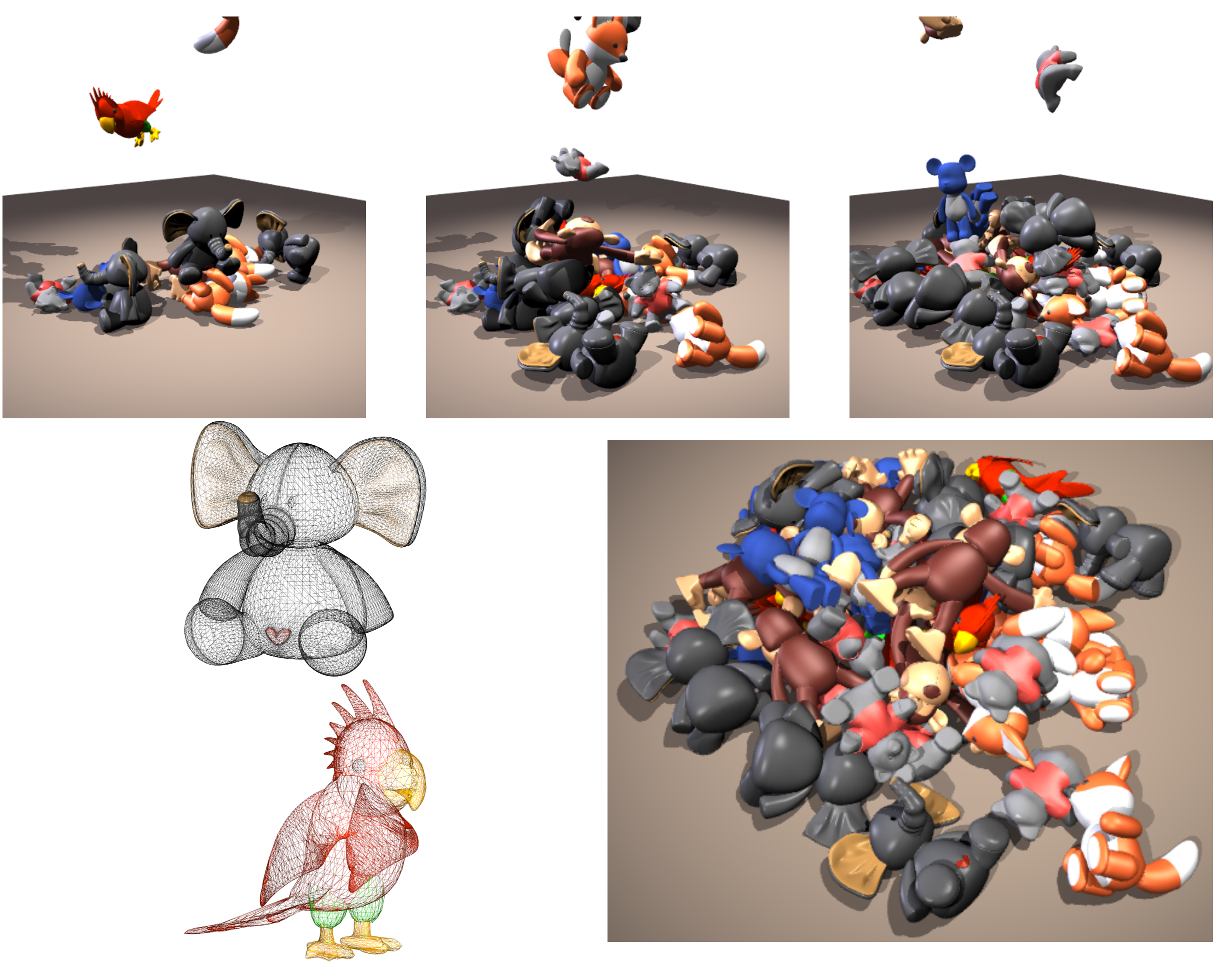


FIG. 4.24 – Pile de 40 jouets déformables avec des géométries complexes, comme l'oiseau qui contient des bords très francs, et un mélange de grands et petits triangles.

4.5 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle méthode de détection des collisions basée sur des volumes d'intersection. Nous avons ensuite utilisé les informations générées (volumes d'intersection et gradients de volume) pour formuler des équations de contraintes exprimées. L'originalité de notre approche est que les contraintes sont exprimées en volume, et nous avons ajouté le support du frottement de Coulomb. La méthode est basée sur le calcul de LDI, et peut être appliquée à n'importe quel objet déformable volumique, et elle peut gérer les changements topologiques, et les auto-intersections sans surcoût important.

La précision est totalement définie par deux paramètres qui sont la taille des pixels, et le nombre de pixels regroupés par volume. La taille des pixels est complètement indépendante de la géométrie initiale des objets, et le second paramètre est à la base du modèle de grille multi-volume. Celui-ci permet de produire un comportement physiquement réaliste (résolution des interpénétrations, respect de la loi de Signorini), tout en gardant un nombre faible de contraintes. Enfin, notre méthode est robuste aux interpénétrations profondes

Simulation	Objets	Triangles	LDI Pixels	LDI Layers	Nombre de Contacts	Détection des Collisions	Formulation des Contacts	Déformation	Résolution des Contraintes	Itérations par Secondes
Médical (Fig. 4.23a)	9	550K	224×168	32	16	22.69 ms	1.35 ms	17.85 ms	32.45 ms	13.30
Pieuvres (Fig. 4.23c)	3	46K	392×352	28	110	5.67 ms	1.20 ms	43.43 ms	12.34 ms	15.60
Saisie (Fig. 4.23b)	2	9K	112×112	14	10	2.73 ms	0.97 ms	7.36 ms	6.18 ms	54.0
Snake (Fig. 4.21)	1	7K	560×336	10	1	3.45 ms	1.01 ms	2.72 ms	1.53 ms	109.00
					6	3.41 ms	1.00 ms	2.87 ms	1.86 ms	104.00
					18	3.45 ms	0.95 ms	2.98 ms	3.09 ms	91.90
					41	3.46 ms	0.84 ms	2.99 ms	7.62 ms	65.30
Armadillo (Fig. 4.22) Rigids	2	*128×107×98 Champ de distance*			255	10.46 ms	0.16 ms	0.38 ms	6.68 ms	56.70
		692K	280×240	22	16	18.48 ms	1.15 ms	0.48 ms	4.08 ms	41.10
FEM	2	692K	208×192	26	28	18.47 ms	1.53 ms	13.54 ms	17.98 ms	19.30
Jouets (Fig. 4.24)	10	205K	456×456	24	99	16.61 ms	1.64 ms	71.24 ms	34.82 ms	7.29
	20	379K	1232×472	34	242	28.79 ms	2.37 ms	146.94 ms	207.02 ms	2.44
	40	725K	1232×528	46	544	44.45 ms	3.43 ms	309.48 ms	723.83 ms	0.88

TAB. 4.2 – Mesure de performances et détails des différents exemples de la section. Les colonnes objets, Triangles, LDI et Contacts, présentent respectivement le nombre d'objets simulés, le nombre de triangles, la taille des LDI, et le nombre de contacts générés. Le reste du tableau présente le temps moyen passé dans les quatre principales étapes de calcul, ainsi que la fréquence de rafraîchissement globale de la simulation (à l'exclusion du rendu).

et nous permet de simuler des configurations difficiles à traiter avec la plupart des autres méthodes de détection, comme le cas d'une intersection à l'état initial. Maintenant que nous savons détecter les collisions, et formuler les équations de contacts, il reste à les résoudre pour déterminer la force de réponse. Le chapitre suivant est consacré à ces aspects.

Chapitre 5

Résolution et calcul de la compliance

Table des matières

5.1 Introduction 129
5.2 Calcul de la réponse aux contacts 129
5.2.1 Résolution directe 129
5.2.2 Gestion des contacts par LCP 130
Méthode de Compliance Warping 131
Contacts frottants 132
Rendu haptique 133
5.2.3 Calcul de la réponse 133
5.3 Parallélisation du Gauss-Seidel 134
5.3.1 Stratégies de parallélisation 135
Parallélisations de base 135
Parallélisations avancées 136
5.3.2 Implémentation sur architectures parallèles 139
5.3.3 Mesures de performances 141
5.4 Calcul de la compliance sur GPU pour modèles spécifiques 145
5.4.1 Inverse d'une matrice bloc tri-diagonale sur GPU 145
Parallélisation sur GPU 146
5.4.2 Calcul de l'opérateur de Delasus sur GPU 147
5.4.3 Implémentation pour le modèle co-rotationnel 148
Mise à jour en cas de découpe 148
Implémentation GPU 151
5.4.4 Résultats 152
5.5 Construction du LCP basée sur le préconditionneur 154
5.5.1 Réponse avec le couplage mécanique correct 155
Simulation de plusieurs objets déformables 155
Construction du LCP 156
5.5.2 Parallélisation sur GPU 157
Résolution de systèmes triangulaires sur GPU 159
5.5.3 Avantages et extension de la méthode 163
Gestion des objets détaillés 163
Couplage mécanique et non homogénéité 163
Application aux modèles non linéaires 164
Changements topologiques 164
5.5.4 Résultats 165
5.6 Conclusion 167

5.1 Introduction

La détection des collisions ayant été détaillée, on sait maintenant comment déterminer l'ensemble des primitives qui sont en intersection, et on possède une mesure de la pénétration entre les objets, ainsi que la direction de la force de réponse. On a vu que la méthode des pénalités permet de produire une réponse rapide et robuste, mais elle ne permet pas de prendre en compte le couplage mécanique entre les contacts. Nous avons également évoqué des méthodes plus précises qui permettent de calculer une force qui va intégralement corriger les intersections à chaque pas de temps.

Dans ce chapitre, nous nous intéresserons à la résolution des contraintes et au calcul de la force de réponse. Nous présenterons tout d'abord les méthodes de résolution et nous verrons qu'une grande partie du temps de calcul est passé sur la construction du problème. Ensuite, nous proposerons un ensemble de contributions qui permettront d'améliorer les temps de calcul, et également la précision des calculs réalisés en temps réel.

5.2 Calcul de la réponse aux contacts

À partir de la détection de collision, nous supposons connu, un ensemble de paires de primitives en intersection (où éventuellement proches de l'intersection). Pour chaque paire, nous connaissons également la normale $\vec{\boldsymbol{n}}$, et une mesure de la pénétration $\boldsymbol{\delta}$. La normale, nous donne la direction de la force à appliquer, alors que $\boldsymbol{\delta}$ est une valeur scalaire signée (généralement une distance Duriez et al. (2006) ou un volume Faure et al. (2008)). Si $\boldsymbol{\delta}$ est négative, les primitives sont en intersections, et si au contraire elle est positive alors les primitives sont distantes.

On rappelle que l'équation dynamique du mouvement des objets, en incluant une force pour collision, peut s'écrire (voir section 4.2.3) :

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b} + \mathbf{J}_1^T \boldsymbol{\lambda} \tag{5.1}$$

Notre objectif dans ce chapitre, sera alors de déterminer les multiplicateurs de Lagrange $\boldsymbol{\lambda}$ à appliquer sur les points en intersection pour résoudre intégralement les interpénétrations.

5.2.1 Résolution directe

La première solution pour résoudre ce système consiste à former un grand système d'équations, en ajoutant les inconnues $\boldsymbol{\lambda}$, au même titre que les degrés de liberté du système :

$$\begin{pmatrix} \mathbf{A}_1 & 0 & {\mathbf{J}_1}^T \\ 0 & \mathbf{A}_2 & {\mathbf{J}_2}^T \\ \mathbf{J}_1 & \mathbf{J}_2 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ -\boldsymbol{\delta} \end{pmatrix} \tag{5.2}$$

On pourra alors utiliser un algorithme de résolution de système linéaire classique (voir chapitre 3) pour résoudre ce problème. Cependant, outre le fait que le système soit de très grande taille (puisqu'il contient les deux matrices des objets déformables), cette méthode ne permet pas de gérer les contraintes unilatérales. En effet, l'égalité entre les deux membres de l'équation (5.2) signifie que les points seront ramenés à la position de

contact. Ceci peut donner l'impression que les objets restent collés quand ils rentrent en collision, et il n'est également pas possible de gérer le frottement.

Il existe toutefois des solutions pour mieux prendre en compte ces aspects. On peut alors utiliser des algorithmes de type *Quadratic Programming* (QP) Baraff (1989); Lötstedt (1984). Le problème est alors formulé comme un problème de minimisation, dans lequel on va déterminer une solution par des résolutions multiples :

$$\begin{cases} min(\boldsymbol{x}^T \begin{pmatrix} \mathbf{A}_1 & 0 \\ 0 & \mathbf{A}_2 \end{pmatrix} \boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{b}) \\ \psi(\boldsymbol{x}_1, \boldsymbol{x}_2) \geq 0 \end{cases} \tag{5.3}$$

On pourra alors définir la notion de contrainte active pour gérer les inégalités des contacts, mais il n'est pas possible de gérer le frottement. Il existe néanmoins des solutions Kaufman et al. (2008), mais qui reposent sur la résolution de multiples petits systèmes QP. Dans tous les cas ces algorithmes sont très coûteux, et donc très peu utilisés dans les simulations en temps réel.

5.2.2 Gestion des contacts par LCP

Une solution alternative consiste à calculer $\boldsymbol{\lambda}$ en résolvant un LCP découlant de la loi de Signorini. Pour le cas du contact (LCP simple) on peut montrer l'équivalence entre les deux formulations Murty (1997). Pour résoudre ces problèmes, on découpe généralement le calcul en plusieurs sous étapes :

Mouvement libre Les objets en interaction sont tout d'abord résolus indépendamment en remplaçant $\boldsymbol{\lambda} = 0$ dans l'équation (5.1). On obtient un système linéaire à résoudre par objet, et ils peuvent être résolus indépendamment. On remarque que cela correspond exactement aux équations du calcul de la déformation des objets, et on peut utiliser les contributions du chapitre 2 pour résoudre les systèmes d'équations. On en déduit alors $\boldsymbol{x}_1^{\text{free}}$ et $\boldsymbol{x}_2^{\text{free}}$, les forces qu'il faudrait appliquer si aucune interaction n'était apparue. Cette étape est appelée *mouvement libre*. À l'issue du mouvement libre, les objets se trouvent en intersection, et c'est à ce moment qu'on peut effectuer la détection des collisions.

Linéarisation et résolution Comme on utilise un intégrateur implicite, on doit évaluer les contraintes à la fin du pas de temps. Pour cela on linéarise comme suit :

$$\underbrace{\left[\psi(\boldsymbol{x}_{1,t+h}, \boldsymbol{x}_{2,t+h})\right]}_{\boldsymbol{\delta}_{t+h}} = \underbrace{\left[\psi(\boldsymbol{x}_1^{\text{free}}, \boldsymbol{x}_2^{\text{free}})\right]}_{\boldsymbol{\delta}^{\text{free}}} + h\mathbf{J}_1\boldsymbol{x}_1^{\text{cor}} + h\mathbf{J}_2\boldsymbol{x}_2^{\text{cor}} \tag{5.4}$$

Avec $\boldsymbol{x}_1^{\text{cor}}$ et $\boldsymbol{x}_2^{\text{cor}}$ étant le mouvement correctif des objets 1 et 2. C'est la force que l'on cherche, et qui sera solution de l'équation (5.1). Pendant cette étape, on considère que $\boldsymbol{b}_1 = \boldsymbol{b}_2 = 0$, ce qui va nous permettre de ne considérer que les points en contact. Ainsi, lorsque l'on remplace (5.1) dans (5.4), on obtient :

$$\boldsymbol{\delta}_{t+h} = \boldsymbol{\delta}^{\text{free}} + h \underbrace{\left[\mathbf{J}_1\mathbf{A}_1^{-1}\mathbf{J}_1^T + \mathbf{J}_2\mathbf{A}_2^{-1}\mathbf{J}_2^T\right]}_{\mathbf{W}} \boldsymbol{\lambda} \tag{5.5}$$

où $\mathbf{W}$ est appelé l'*opérateur de Delasus*, qui décrit les connexions mécaniques dans l'espace

des contacts. Sa structure est dense, mais elle est généralement beaucoup plus petite que $\mathbf{A}$, puisque sa dimension est égale au nombre de contraintes.

L'équation (5.5) comporte deux inconnues, qui sont $\boldsymbol{\delta}_{t+h}$ (les distances de pénétration à la fin du pas de temps), et $\boldsymbol{\lambda}$ (les forces à appliquer aux nœuds pour résoudre les contraintes). Nous résolvons ce problème en utilisant l'algorithme du Gauss-Seidel, qui vérifie de façon itérative chacune des contraintes contenues ψ, et applique une correction en fonction de son état (active ou non). Nous obtenons à la fin $\boldsymbol{\lambda}$ qui satisfait la loi de Signorini Duriez et al. (2006), nous détaillerons cet algorithme dans la section 5.2.3.

Mouvement contraint Lorsque la valeur de $\boldsymbol{\lambda}$ est disponible, il ne contient des valeurs que sur les points en contact. Il est alors nécessaire de repasser dans l'espace de déformation, pour obtenir le déplacement de l'objet. Cette étape est appelée *mouvement contraint* :

$$\begin{aligned} \boldsymbol{x}_{1,t+h} &= \boldsymbol{x}_1^{\text{free}} + h\boldsymbol{x}_1^{\text{cor}} \quad \text{avec} \quad \boldsymbol{x}_1^{\text{cor}} = \mathbf{A}_1^{-1}\mathbf{J}_1^T\boldsymbol{\lambda} \\ \boldsymbol{x}_{2,t+h} &= \boldsymbol{x}_2^{\text{free}} + h\boldsymbol{x}_2^{\text{cor}} \quad \text{avec} \quad \boldsymbol{x}_2^{\text{cor}} = \mathbf{A}_2^{-1}\mathbf{J}_2^T\boldsymbol{\lambda} \end{aligned} \tag{5.6}$$

On obtient finalement la solution du problème initial, et on peut en déduire la position des objets qui respectent toutes les contraintes. On peut démarrer un nouveau pas de temps, sans aucune interpénétration.

En revanche, pour calculer les équations (5.6) et (5.5), nous avons fait appel à l'inverse du système $\mathbf{A}^{-1}$, et on sait que cette opération est consommatrice de temps de calcul. Cette méthode a alors été utilisée pour les modèles linéaires, où l'inverse de la matrice est facile à connaître puisqu'elle peut être pré-calculée. Cependant, on ne peut pas transposer cette méthode aux modèles non linéaires car la matrice change à chaque pas de temps. De même, les modifications topologiques sont difficiles à réaliser en temps réel. Certaines solutions ont été proposées pour mettre à jour une matrice de rigidité pré-inversée Lee et al. (2005), mais ils n'adressent que le modèle linéaire.

Malgré tout, Saupin et al. (2008b) ont montré qu'il est possible d'utiliser une approximation de l'inverse du système, pour résoudre les contacts. En effet, cela conduit à une approximation dans l'évaluation du couplage mécanique liant les contacts. Or, comme on ne modifie pas les lois de contacts, on garantit la non-interpénétration des objets à la fin du pas de temps. La plus extrême des approximations consiste alors à ne considérer que la diagonale de la matrice. Les contacts seront alors corrigés sans liaison mécanique, ce qui mène à des configurations fausses (on pourra par exemple déformer un partie très dure aussi facilement qu'une partie moins rigide). L'évaluation d'une matrice inverse ou d'une matrice inverse approchée est un problème difficile qui a déjà fait l'objet de nombreuses études pour le calcul de préconditionneurs. En général, les meilleurs pré-conditionneurs cherchent à tirer partie des spécificités des modèles.

Méthode de Compliance Warping

Pour le modèle co-rotationnel, nous pouvons obtenir une bonne approximation de la matrice $\mathbf{A}^{-1}$ en utilisant l'inverse de la matrice dans la configuration au repos $\mathbf{A}_0^{-1}$. Saupin et al. (2008b), ont proposé la méthode de *compliance warping*, qui est inspirée directement de la formulation du modèle co-rotationnel. Au début de la simulation, la matrice $\mathbf{A}_0^{-1}$ est pré-calculée, puis un ensemble de rotation (depuis la forme au repos jusqu'à la forme ac-

tuelle) sont évaluées à chaque pas de temps. Les rotations sont calculées indépendamment sur chaque nœud, en utilisant les méthodes que nous avons présentée dans la section 2.5.2. On obtient alors une bonne approximation de la matrice de compliance $\mathbf{C} \approx h\mathbf{A}^{-1}$ en utilisant l'équation suivante :

$$\mathbf{C} = h\mathbf{R}\mathbf{A}_0^{-1}\mathbf{R}^T \tag{5.7}$$

Où $\mathbf{R}$ est une matrice diagonale par bloc 3×3, qui rassemble les rotations associées à chaque nœud[1]. Cette simplification permet d'accélérer de manière significative le calcul puisqu'il est possible de tirer partie du faible taux de remplissage $\mathbf{R}$ (matrice diagonale), et surtout de $\mathbf{J}$ (matrice creuse). En effet, pour obtenir $\mathbf{W}$, on doit calculer le produit $\mathbf{J}\mathbf{C}\mathbf{J}^T$ pour chaque objet. On peut procéder en deux temps :

1. $\mathbf{S} = \mathbf{C}\mathbf{J}^T$
2. $\mathbf{W} = \mathbf{J}\mathbf{S}$

Même si $\mathbf{S}$ est en théorie une matrice dense, on se rend compte que pour calculer le second produit, nous n'utiliserons pas toutes les colonnes de $\mathbf{S}$. On peut donc se contenter de ne calculer que les colonnes de $\mathbf{S}$ qui nous intéressent, c'est-à-dire l'union de toutes les colonnes de $\mathbf{J}$ qui contiennent au moins une valeur non nulle (voir Saupin et al. (2008b)).

Évidemment cette solution est très intéressante, mais elle présente tout de même quelques limitations. La plus évidente est qu'elle empêche toute mise à jour de la topologie, puisqu'elle repose sur l'inversion de la matrice dans l'état initial. L'autre limitation importante concerne le stockage de la matrice inverse, qui est une matrice dense. Ainsi, cette méthode limite sensiblement le nombre de nœuds pouvant être simulés, non pas pour une question de performance, mais plutôt pour une question de stockage mémoire.

Contacts frottants

Le frottement de Coulomb implique de gérer des forces supplémentaires, tangentielles aux points de contact. On les définit généralement dans une base de 3 vecteurs orthogonaux $\{\vec{n}, \vec{t_1}, \vec{t_2}\}$). La composante selon la normale $\vec{n}$ permet de calculer la force pour sortir les objets de l'interpénétration, alors que les forces tangentes selon $\vec{t_1}$ et $\vec{t_2}$, définissent le frottement.

Plusieurs articles décrivent comment étendre la méthode à une formulation en LCP, afin d'inclure et de résoudre les frictions (voir, par exemple Anitescu et al. (1999) et Jourdan et al. (1998)). Le premier utilise une discrétisation de la pyramide du cône de frottement pour formuler le problème comme un LCP Baraff (1991); Milenkovic et Schmidl (2001).

Il est également possible d'utiliser le cône exact (un problème de complémentarité non linéaire) et de calculer la solution en utilisant une méthode itérative Duriez et al. (2006). Cette solution revient à calculer l'opérateur de Delasus dans un repère lié à chaque contact.

[1]On remarque que $\mathbf{C}$ n'est qu'une approximation de $\mathbf{A}^{-1}$. En effet, dans le modèle co-rotationnel on utilise les rotations pour "tourner" la rigidité locale de chaque élément (ce qui explique que $\mathbf{A}$ change à chaque pas de temps). Ici, on pré-inverse la matrice assemblée, et on évalue les rotations sur chaque nœud pour minimiser l'influence des non linéarités géométriques.

Ainsi, en 3D on forme le problème suivant :

$$\begin{bmatrix} \boldsymbol{\delta}_{\vec{n}} \\ \boldsymbol{\delta}_{\vec{t_1}} \\ \boldsymbol{\delta}_{\vec{t_2}} \end{bmatrix} = \begin{bmatrix} \mathbf{W}_{\vec{n}\vec{n}} & \mathbf{W}_{\vec{n}\vec{t_1}} & \mathbf{W}_{\vec{n}\vec{t_2}} \\ \mathbf{W}_{\vec{t_1}\vec{n}} & \mathbf{W}_{\vec{t_1}\vec{t_1}} & \mathbf{W}_{\vec{t_1}\vec{t_2}} \\ \mathbf{W}_{\vec{t_2}\vec{n}} & \mathbf{W}_{\vec{t_2}\vec{t_1}} & \mathbf{W}_{\vec{t_2}\vec{t_2}} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda}_{\vec{n}} \\ \boldsymbol{\lambda}_{\vec{t_1}} \\ \boldsymbol{\lambda}_{\vec{t_2}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{\delta}^{\text{libre}}_{\vec{n}} \\ \boldsymbol{\delta}^{\text{libre}}_{\vec{t_1}} \\ \boldsymbol{\delta}^{\text{libre}}_{\vec{t_2}} \end{bmatrix} \tag{5.8}$$

Ce qu'il est important de remarquer, c'est que la gestion de contacts avec frottement va générer 3 lignes d'inconnues dans le Gauss-Seidel, alors qu'un contact sans frottement sera traité par une seule équation.

Rendu haptique

Nous ne contribuerons pas sur la qualité du rendu haptique dans les simulations, et nous utiliserons l'implémentation disponible dans SOFA. Le plus gros problème à résoudre est que le rendu haptique nécessite un taux élevé (de $300Hz$ à $1000Hz$) pour le calcul de la force. Comme la simulation n'est généralement pas assez rapide, nous nous appuyons sur la technique asynchrone présentée dans Saupin et al. (2008a), qui sépare la boucle haptique de la boucle de simulation.

Toutefois, la résolution des contacts par LCP permet d'obtenir à la fois un comportement réaliste et un retour haptique convaincant. En effet, il est évident que le calcul d'une force correcte, et la gestion du frottement sont nécessaires pour pouvoir générer la force transmise au périphérique, mais le couplage mécanique joue également un rôle important dans le retour tactile. On pourra par exemple ressentir les différentes rigidités d'un organe non homogène, et éventuellement les objets qui se trouvent derrière l'organe que l'utilisateur manipule. En effet, le couplage mécanique entre les contacts nous permettra d'influencer la force de réponse même si on ne touche pas directement les objets. Ce rendu haptique n'est possible qu'avec des méthodes basées sur la compliance.

5.2.3 Calcul de la réponse

D'une manière générale, un LCP est défini par les équations suivantes Murty (1997) :

$$\boldsymbol{w} - \mathcal{M}\boldsymbol{z} = \boldsymbol{q} \tag{5.9}$$

$$\boldsymbol{w}_i \geq 0,\ \boldsymbol{z}_i \geq 0 \text{ and } \boldsymbol{w}_i\boldsymbol{z}_i = 0 \text{ for all } i \tag{5.10}$$

Où $\boldsymbol{z}$ et $\boldsymbol{w}$ sont les inconnues du problème, alors que $\boldsymbol{q}$ est un vecteur connu, et $\mathcal{M}$ une matrice dense. L'équation (5.9) ressemble à un système linéaire, mais elle doit être résolue en tenant compte de l'équation (5.10) qui définit une contrainte sur la solution. Cette contrainte signifie simplement que toutes les valeurs de $\boldsymbol{z}$ et $\boldsymbol{w}$ sont positives ou nulles, et pour chaque i soit $\boldsymbol{z}_i$, soit $\boldsymbol{w}_i$ est égal à 0. Ceci correspond exactement à la loi de Signorini, et il suffit de remplacer les termes de l'équation, par ceux du problème de contact pour obtenir le système suivant[2] :

$$\begin{cases} \boldsymbol{\delta}_{t+h} = \underbrace{\mathbf{J}_1\mathbf{A}_1^{-1}\mathbf{J}_1^T + \mathbf{J}_2\mathbf{A}_2^{-1}\mathbf{J}_2^T}_{\mathbf{W}}\boldsymbol{\lambda} + \boldsymbol{\delta}^{\text{free}} \\ 0 \leq \boldsymbol{\delta}_{t+h} \perp \boldsymbol{\lambda} \geq 0 \end{cases} \tag{5.11}$$

[2]Notons que nous présentons ici la version sans frottement

La méthode du Gauss-Seidel Barrett et al. (1994) a l'avantage de converger (relativement lentement) vers une solution. Par ailleurs, en tant que solveur itératif, on peut lui initialiser le calcul avec une solution approchée basée sur la cohérence temporelle de la simulation pour le faire converger plus vite. Ainsi, il présente un bon candidat pour résoudre les problèmes de type LCP. En effet, nous rappelons que cet algorithme consiste à parcourir itérativement une à une chaque inconnue du système. Pour chacune on met à jour la solution (changement d'état ou calcul d'une nouvelle force), puis on utilise le résultat calculé pour les inconnues suivantes dans la même itération.

L'algorithme sous-jacent est intrinsèquement séquentiel, et difficile à paralléliser. Plusieurs travaux ont cherché à exploiter la parcimonie du système pour en extraire du parallélisme. On peut citer par exemple Koester et al. (1994) et Adams (2001) qui sont basés sur la coloration de graphes, et Bond (2006) qui cherche à identifier des groupes de contraintes indépendantes. L'idée principale est alors de rassembler ou dissocier des groupes de contraintes indépendantes, qui peuvent être traités en parallèle. Bien que cette méthode peut produire une accélération significative, elle repose sur la supposition d'un faible taux de remplissage de $\mathbf{W}$. Dans le cas de plusieurs groupes d'objets distants en collisions, il est possible de construire de multiples petits LCP, et de les résoudre en parallèle. Cela supprime le besoin de synchronisation globale entre les résolutions, et Kipfer (2007) utilise cette technique pour exploiter efficacement tous les processeurs du GPU. Cependant, dans nos simulations médicales, on simule généralement peu d'objets, mais qui sont complexes et soumis à de nombreux contacts. Ceci produit un système étroitement couplé, et aucune de ces méthodes ne peut en extraire du parallélisme.

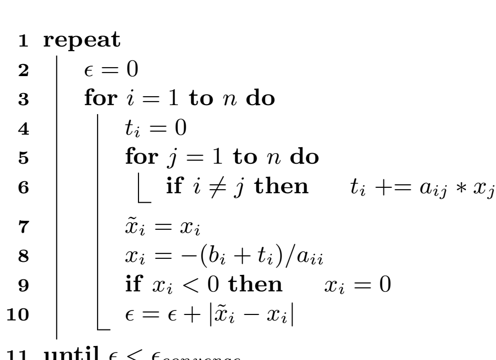
Il a également été proposé de retirer des dépendances entre les contraintes Renouf et al. (2004). On modifie alors l'algorithme, et en conséquence un plus grand nombre d'itérations est nécessaire pour obtenir une précision équivalente. En effet, si moins de dépendances sont respectées, cette solution ressemble alors à l'algorithme de Jacobi. Néanmoins, plus de parallélisme est exploitable même dans les cas fortement contraints. Ainsi, avec un grand nombre de processeurs, on peut espérer trouver une solution équivalente en un temps plus court.

5.3 Parallélisation du Gauss-Seidel

La résolution de nombreux contacts avec l'algorithme du Gauss-Seidel peut rapidement devenir problématique. C'est pourquoi, nous proposons ici une parallélisation de cet algorithme Courtecuisse et Allard (2009). Nous commençons par identifier ses dépendances, puis nous présenterons un ensemble de stratégies de parallélisation, et les mettrons en œuvre sur différentes architectures.

Nous nous plaçons dans un cas entièrement couplé, où il est impossible d'extraire du parallélisme en découpant la matrice. L'algorithme du Gauss-Seidel[3] (voir l'algorithme 7) est constitué d'une boucle principale qu'on effectue jusqu'à convergence (ligne 1). Dans chacune des itérations on parcourt chaque contrainte séquentiellement, et on met à jour la force correspondante en évaluant la contribution de toutes les autres contraintes (ligne

[3]Notons que l'algorithme présenté ne considère que la résolution de contraintes unilatérales. Cependant d'autres types de problèmes (contraintes bilatérales, la friction, ...) peuvent être manipulés en changeant les calculs dans les lignes 8 et 9, la structure générale et les dépendances de données restent les mêmes.



1 **repeat**
2 $\epsilon = 0$
3 **for** $i = 1$ **to** n **do**
4 $t_i = 0$
5 **for** $j = 1$ **to** n **do**
6 **if** $i \neq j$ **then** $t_i \mathrel{+}= a_{ij} * x_j$
7 $\tilde{x}_i = x_i$
8 $x_i = -(b_i + t_i)/a_{ii}$
9 **if** $x_i < 0$ **then** $x_i = 0$
10 $\epsilon = \epsilon + |\tilde{x}_i - x_i|$
11 **until** $\epsilon < \epsilon_{converge}$

Algorithme 7 : Algorithme du Gauss-Seidel algorithme appliqué à un LCP.

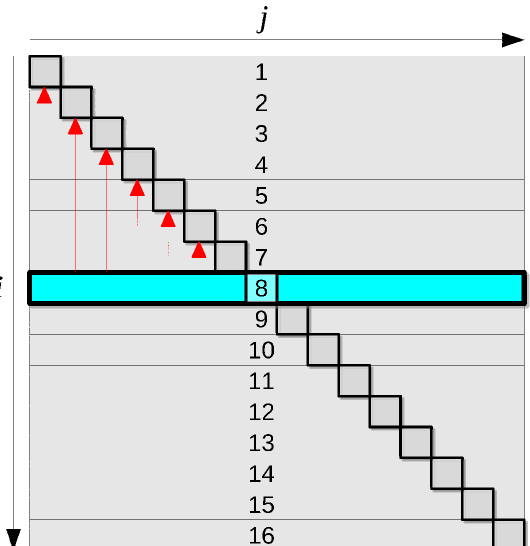


FIG. 5.1 – Dépendances de données, représentées ici par des flèches rouges.

5 et 6). Pour mettre à jour la contribution i on utilise la force de l'itération courante pour toutes les contraintes plus petites que i, et la force de l'itération précédente pour les autres contraintes. Ainsi, le calcul d'une ligne i exige le résultat de toutes les lignes précédentes.

5.3.1 Stratégies de parallélisation

Afin d'exploiter les architectures massivement parallèles, un grand nombre de tâches doivent être extraites. Ceci est difficile dans l'algorithme du Gauss-Seidel puisque chaque contrainte doit être traitée de manière séquentielle. Dans la suite, nous allons décrire comment paralléliser une itération de la boucle principale de l'algorithme du Gauss-Seidel appliqué à n contraintes.

Parallélisations de base

Nous commencerons par présenter deux premières stratégies de parallélisation qu'on peut considérer triviales. Elles nous serviront de base pour proposer deux autres stratégies plus complexes, qui seront capables d'extraire un plus haut degré de parallélisme.

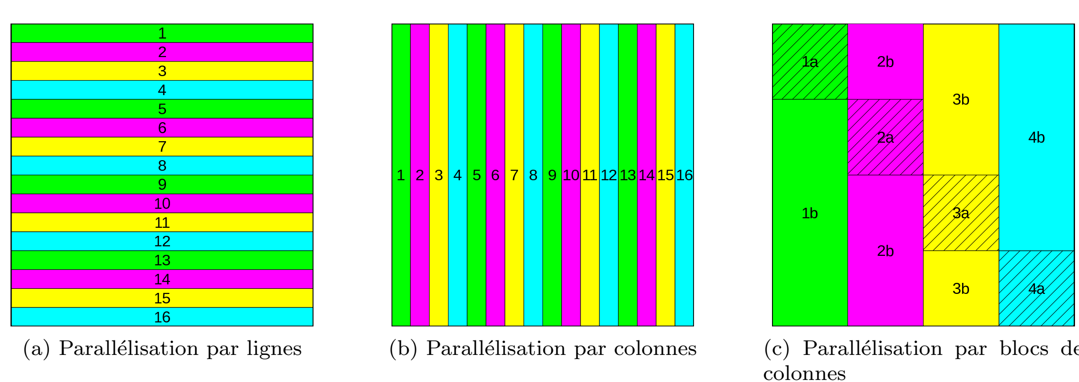
Parallélisation par lignes La boucle interne de l'algorithme 7 (ligne 5 et 6), calcule n produits (chacun correspondant à une colonne de la matrice du système), qui peuvent être évalués en parallèle (voir figure 5.2a). Cependant, leurs résultats doivent ensuite être assemblés. Ce calcul est similaire à un produit scalaire et peut être parallélisé en utilisant des approches classiques, comme une réduction parallèle (voir annexe C).

Pour mettre à jour la contrainte correspondante, il est nécessaire d'attendre que le calcul d'une ligne complète soit terminée. Cela nécessite donc n synchronisations globales pour chaque itération (et en plus des synchronisations supplémentaires pour les réductions parallèles). Entre les synchronisations globales, seules n tâches parallèles peuvent être

lancées, ce qui ne permet pas d'exploiter efficacement une architecture hautement parallèle comme un GPU.

Parallélisation par colonnes Il est possible d'éliminer l'étape de réduction parallèle de la stratégie précédente en créant des groupes de tâches correspondant aux colonnes (voir figure 5.2b). En effet, pour la première itération on constate que les données situées sur la triangulaire supérieure de la matrice ne présentent aucune dépendances. On peut donc pré-calculer leurs accumulations sur CPU, et stocker le résultat dans un vecteur temporaire de taille n. La force de la première ligne peut alors être mise à jour directement puisque le vecteur temporaire contient toutes les contributions de la première ligne. Une fois que la force est mise à jour, on peut alors ajouter dans le vecteur temporaire, toutes les contributions de la première colonne en parallèle. Ceci nous permettra de mettre à jour la force de la seconde ligne après une synchronisation.

Avec cette solution, on lance un groupe de calcul pour chaque colonne. Les threads travaillant sur les données en dessous de la diagonale accumuleront les contributions de l'itération courante, alors que les autres threads travailleront sur les données de la prochaine itération. Dans une même colonne, chaque thread peut alors accumuler sa contribution indépendamment des autres dans le vecteur temporaire, ce qui implique qu'il n'est plus nécessaire d'utiliser de réduction parallèle. Cependant, cette stratégie nécessite encore n synchronisations globales, avec seulement n tâches parallèles entre elles.



(a) Parallélisation par lignes

(b) Parallélisation par colonnes

(c) Parallélisation par blocs de colonnes

FIG. 5.2 – Schémas de parallélisation GPU : chaque rectangle représente un groupe de tâches de traitement d'un sous-ensemble de la matrice du système (16×16 dans cet exemple).

Parallélisations avancées

Les synchronisations globales sont très coûteuses sur les architectures hautement parallèles. Cependant, si l'on ne crée qu'un seul groupe de travail, nous serons en mesure d'utiliser uniquement des synchronisations locales, mais on exploitera alors qu'un faible pourcentage de notre architecture. Des synchronisations entre groupes sont alors nécessaires, nous allons voir comment les minimiser.

Parallélisation par groupes de colonnes Une solution consiste à remplacer un ensemble de synchronisations globales par des synchronisations locales (qui ne peuvent être utilisées qu'au sein d'un même bloc CUDA). En organisant nos n contraintes dans des groupes

de m colonnes, on produit $g = \lceil n/m \rceil$ groupes. L'idée de base de cette parallélisation est d'utiliser des synchronisations globales pour synchroniser les groupes de tâches entre eux, et des synchronisations locales pour synchroniser les threads à l'intérieur de chaque groupe.

En utilisant le schéma de parallélisation par colonnes au niveau des groupes, on identifie alors deux tâches bien distinctes, qui correspondent aux blocs[4] a et b dans la figure 5.2c.

La première tâche consiste à mettre à jour les forces des contraintes du groupe, ce qui correspond au calcul des données sur le bloc diagonal de la matrice. Comme ces calculs présentent de fortes dépendances, on n'utilise que m threads, qui seront synchronisés par des attentes locales. Ainsi, un premier thread met à jour la nouvelle force, puis alternativement libère les $m-1$ colonnes sur le bloc diagonal. En synchronisant les m threads colonnes par colonnes, on peut alors mettre à jour toutes les forces du groupe.

La seconde tâche est plus facile à paralléliser. Elle consiste à accumuler les contributions partielles de toutes les colonnes (hors diagonale) du groupe. On peut alors lancer un maximum de $m(n-m)$ threads, ce qui représente assez de travail pour exploiter efficacement le processeur parallèle. On remarque toutefois que tous les threads d'une même ligne doivent accumuler leur contribution au même endroit, ce qui peut se faire soit avec des réductions parallèles (voir annexe C), soit en utilisant plusieurs vecteurs temporaires.

Grâce à ce découpage, seules $2g$ synchronisations globales sont requises par cette stratégie. De plus, bien que la première étape ne crée que m des tâches parallèles, la deuxième étape permet de lancer $m(n-m)$ tâches en parallèles. Or, à mesure que la dimension du système grandit, le temps de calcul de la seconde tâche parallèle devient dominant.

Parallélisation suivant le graphe des dépendances En organisant les calculs en groupes on a pu réduire le nombre de synchronisations globales, mais elles restent nécessaires. Ainsi, si on souhaite supprimer complètement les synchronisations globales, un autre type de synchronisation doit être utilisé pour assurer un ordre de calcul correct.

En analysant le graphe de dépendances, on constate qu'à un moment donné, un seul groupe de forces peut être mis à jour. Ainsi, on peut utiliser un compteur en mémoire partagée *counter* qui stocke le nombre de blocs diagonaux qui ont déjà été traités. De ce compteur, nous pouvons en déduire implicitement si un bloc diagonal a_{jj} a déjà été mis à jour, afin de commencer le traitement d'un élément a_{ij} pour l'itération *iter* de l'algorithme :

$$counter > \lceil n/m \rceil (iter - 1) + \lfloor j/m \rfloor, \quad \text{for } j > i \tag{5.12}$$

$$counter > \lceil n/m \rceil iter + \lfloor j/m \rfloor, \quad \text{for } j < i \tag{5.13}$$

L'équation (5.12) exprime les dépendances vers des blocs de la partie triangulaire supérieure de la matrice (où les valeurs de l'itération précédente sont utilisées). Alors que (5.13) se rapporte à des blocs dans la partie triangulaire inférieure de la matrice (exigeant des valeurs mises à jour lors de l'itération courante). Si le matériel fournit suffi-

[4] Attention, il ne faut pas confondre la notion de bloc de données pour une matrice avec la notion de bloc CUDA qui permet de regrouper des calculs sur le GPU (voir section 2.6). En règle générale, nous parlerons de groupes pour désigner un ensemble de données traitées en parallèle, alors que nous parlerons de bloc CUDA pour désigner un ensemble de threads s'executant sur le GPU.

```
   Data : ε                              // shared current residual
 1 repeat
 2   ε = 0
 3   for j_g = 0 to g − 1 do
 4     i_g = j_g                         // (a) block on the diagonal
 5     parfor i_t = 1 to m do
 6       i = i_g m + i_t
 7       for j_t = 1 to m do
 8         j = j_g m + j_t
 9         if i = j then
10           x̃_i = x_i
11           x_i = −(b_i + t_i)/a_ii
12           if x_i < 0 then x_i = 0
13           t_i = 0
14           ε = ε + |x̃_i − x_i|
15         else
16           t_i = t_i + a_ij * x_j
17         barrier
18     global barrier
19     parfor (i_t, j_t) = (1, 1) to (n − m, m) do
20       i = i_t                         // (b) non-diagonal blocks
         // skip block on the diagonal
21       if i_t ≥ j_g m then i = i_t + m
22       j = j_g m + j_t
23       t_i = t_i + a_ij * x_j
24 until ε < ε_converge
```

Algorithme 8 : Parallélisation d'une itération de la boucle externe dans la méthode de Gauss-Seidel, avec un découpage en groupes de colonnes. Selon l'architecture matérielle, chaque parfor sera traduit en groupes de threads exécutés en parallèles, ou exécutés comme de petites boucles sur un processeur donné.

```
   Data : ε                              // shared current residual
   Data : counter = 0                    // shared atomic counter
 1 parfor (iter, i_g) = (0, 0) to (g − 1, iter_max − 1) do
 2   parfor i_t = 1 to m do
 3     i = i_g m + i_t
 4     t_i = 0
 5   for j_g = i_g + 1 to g − 1 do
 6     // (a) blocks after the diagonal
       wait(counter > (iter − 1) * g + j_g)
 7     parfor (i, j) = (1, 1) to (m, m) do
 8       (i, j) = (i_g m + i_t, j_g m + j_t)
 9       t_i = t_i + a_ij * x_j
10   for j_g = 0 to i_g do
11     // (b) blocks before the diagonal
       wait(counter > iter * g + j_g)
12     parfor (i_t, j_t) = (1, 1) to (m, m) do
13       (i, j) = (i_g m + i_t, j_g m + j_t)
14       t_i = t_i + a_ij * x_j
15   if counter = iter_max * g then return ε
16   j_g = i_g                           // (c) block on the diagonal
17   if i_g = 0 then ε = 0
18   for i_t = 1 to m do
19     i = i_g m + i_t
20     parfor j_t = 1 to m do
21       j = j_g m + j_t
22       if i ≠ j then t_i = t_i + a_ij * x_j
23     x̃_i = x_i
24     x_i = −(b_i + t_i)/a_ii
25     if x_i < 0 then x_i = 0
26     ε = ε + |x̃_i − x_i|
     // notify blocks waiting on this value
27   if (i_g = g − 1) and (ε < ε_converge) then
28     counter = iter_max * g
29   else
30     counter = iter * g + i_g
```

Algorithme 9 : Parallélisation de la méthode de Gauss-Seidel en utilisant un compteur atomique pour s'assurer que l'ordre des calculs respecte les dépendances.

samment de garantie quant à la lecture atomique de ce compteur, on peut l'utiliser pour synchroniser les groupes. C'est-à-dire que si un thread écrit une nouvelle valeur sur le compteur pendant que d'autres le lisent, ces derniers obtiendront soit l'ancienne valeur, soit la nouvelle, mais pas une composition binaire des deux. Or, puisqu'un seul bloc diagonal ne peut être calculé à la fois, alors un seul bloc CUDA peut écrire sur le compteur à un instant donné.

La stratégie de parallélisation consiste alors à lancer un nombre fixe de groupes, et à les répartir de façon *round-robin* sur le multiprocesseur. C'est-à-dire que chaque bloc CUDA calcule plusieurs lignes différentes qui sont déterminées par le numéro du groupe et de l'itération de l'algorithme. Avant de calculer un bloc, on ajoute une boucle d'attente basée sur le compteur partagé, qui va garantir le respect des dépendances. Évidemment, cette boucle d'attente va introduire des retards importants dans les calculs. Toutefois, si tous

les groupes sont exécutés à la même vitesse, ils garderont un décalage constant après la première diagonale calculée, et ils devraient rarement avoir à attendre les uns après les autres (voir figure 5.3c). L'algorithme résultant est détaillé dans l'algorithme 9, et la séquence de calcul est illustrée dans la figure 5.3b.

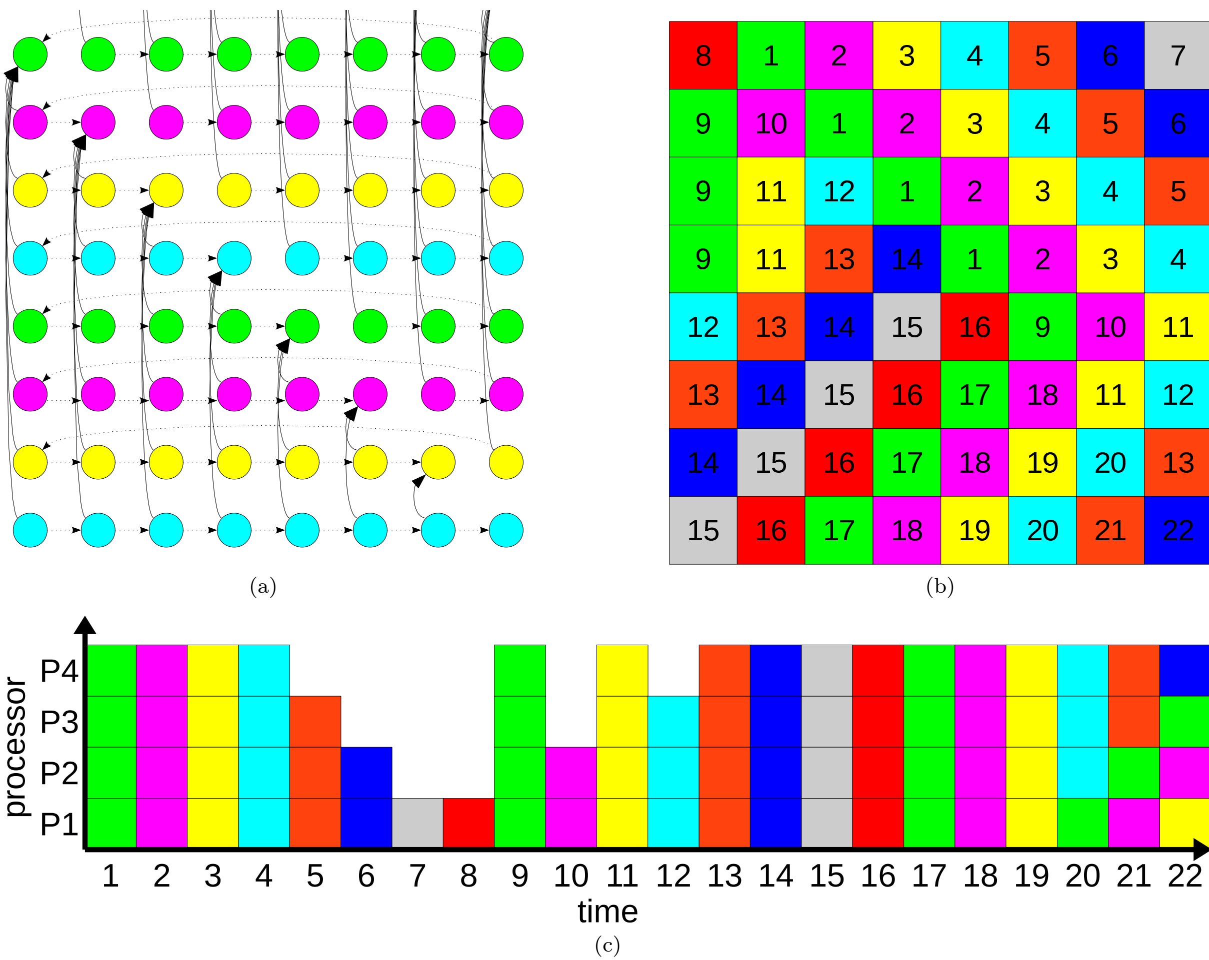


FIG. 5.3 – Parallélisation du Gauss-Seidel en utilisant un compteur atomique, appliqué à une matrice de 8×8 bloc. (a) Graphe d'exécution avec 4 processeurs (flèches horizontales) et dépendances de données (flèches verticales). (b) Ordre d'exécution sur 4 processeurs, en négligeant les problèmes latences. (c) Ordonnancement des calculs sur chaque processeur au cours du temps pour la première itération.

5.3.2 Implémentation sur architectures parallèles

Nous avons évalué les performances et l'évolutivité des algorithmes sur différentes architectures. Nous présentons maintenant deux implémentations différentes, l'une sur un processeur multi-cœurs et l'autre sur un GPU en utilisant CUDA.

Implémentation GPU en utilisant CUDA Pour chaque groupe de tâches représentés sur la figure 5.2, plusieurs blocs CUDA sont exécutés. Chaque bloc CUDA contient 64 threads pour les 3 premières stratégies, et 16×16 threads pour l'approche basée sur le graphe de dépendance. Chaque thread gère un élément de la matrice du système, et les synchroni-

sations globales sont assurées par différents appels de kernels.

Pour implémenter la stratégie basée sur les groupes de colonnes, nous avons utilisé des groupes de $m = 64$. Le premier kernel qui correspond à l'étape *(a)* de la figure 5.2c, est exécuté avec un seul bloc CUDA, contenant 64 threads. Les threads réalisent une boucle sur les 64 contraintes, et sont synchronisés à chaque itération en utilisant la primitive **__syncthread** (synchronisation locale). Le second kernel (l'étape *(b)*), est exécuté avec $n - 64$ blocs CUDA, chacun contenant 64 threads. Chaque bloc CUDA calcule une ligne de la matrice, et chaque thread écrit l'accumulation partielle qu'il a calculé dans des vecteurs temporaires différents. Bien que le premier kernel soit seulement en mesure d'exploiter un seul multi-processeur, les calculs exécutés par le second kernel dominent le temps de calcul si la taille de la matrice est grande.

La stratégie basée sur le graphe de dépendance, permet de supprimer toutes les synchronisations globales, elle est donc mise en œuvre par une seule invocation de kernel. Chaque itération de la boucle principale (ligne 1 de l'algorithme de 9) est exécutée par des blocs CUDA de 16×16 threads. Pour minimiser l'impact des boucles d'attente, un seul thread du bloc CUDA lit le compteur en mémoire partagée, pendant que les autres l'attendent dans un **__syncthread**. Quand la dépendance est satisfaite, le thread principal copie la valeur du compteur dans une variable en mémoire locale (qui est visible rapidement par tous les threads d'un même bloc CUDA), puis libère les autres threads en entrant dans le **__syncthread**. Ainsi, l'avantage est qu'un seul thread réalise des lectures en mémoire globale, ce qui limite l'impact sur la bande passante. De plus, comme plusieurs blocs CUDA sont exécutés en parallèle il est fréquent que la valeur obtenue dans une boucle satisfasse les dépendances de plusieurs groupes. Dans ce cas, seul une synchronisation locale sera nécessaire, et il ne sera pas utile de relire le compteur dans la boucle d'attente.

Toutefois, pour mesurer l'impact lié aux boucles d'attente, nous avons implémenté et testé trois variantes :

1. **graph** : Qui correspond à ce que nous venons de présenter, le premier thread de chaque bloc CUDA exécute les boucles d'attente pendant que les autres threads du bloc sont bloqués dans un **__syncthread**.
2. **graph-2** : Pendant la boucle d'attente, si après avoir lu le compteur partagé la dépendance n'est pas satisfaite, alors tous les threads commencent à accumuler les contributions des lignes suivantes (qui seront affectés au multiprocesseur). Ceci est équivalent à traiter deux groupes de forces en même temps avec le même multiprocesseur[5].
3. **relaxed** : Les boucles d'attente sont tous simplement ignorées. Cela change la sémantique de l'algorithme comme présenté dans Renouf et al. (2004), mais garantit que tous les processeurs calculent à pleine vitesse.

Contrairement aux parallélisations traditionnelles sur GPU (où l'objectif est de lancer un maximum de blocs pour extraire le maximum de parallélisme), notre approche ne nous permet pas de lancer un plus grand nombre de tâches que le nombre de blocs CUDA qui

[5] Cependant, il est important de donner une priorité plus grande pour la première ligne, car d'autres groupes peuvent être en attente du calcul de sa diagonale. Ainsi, après chaque accumulation dans la seconde ligne, la dépendance de la première contrainte doit être ré-évaluée.

peuvent être exécutés simultanément par le GPU. En effet, comme un bloc CUDA est exécuté sans interruption sur le GPU, nous ne pouvons pas nous permettre d'attendre le résultat des tâches qui ne sont pas encore lancées (sous peine de tomber dans une attente infinie). Il est donc impératif de limiter le nombre de bloc CUDA au nombre maximum pouvant être exécutés simultanément par les multiprocesseurs. Cependant, cette valeur est difficile à évaluer puisqu'elle dépend directement du matériel. Il faut l'ajuster en fonction du nombre de multiprocesseurs sur le GPU, mais également en fonction de la consommation mémoire du kernel. En effet, on sait que les GPU peuvent exécuter plusieurs blocs simultanément sur un même multiprocesseur, tant que la consommation mémoire de tous les blocs n'excède pas celle disponible.

Implémentation pour processeur Multi-threads Nous avons implémenté sur CPU la stratégie basée sur le graphe de dépendances de manière similaire à la version GPU. Cependant, tant que le nombre de cœurs de calcul reste relativement petit, cela ne nécessite l'extraction que de quelques tâches en parallèle. En conséquence, chaque processus effectue deux boucles imbriquées pour calculer une à une les valeurs du groupe (ce qui correspond approximativement à tous les threads exécutés dans un bloc CUDA). Ainsi, les synchronisations locales ne sont pas nécessaires, et les synchronisations globales sont traitées à l'aide de barrières disponibles via de nombreuses librairies parallèles (nous avons utilisé la librairie boost).

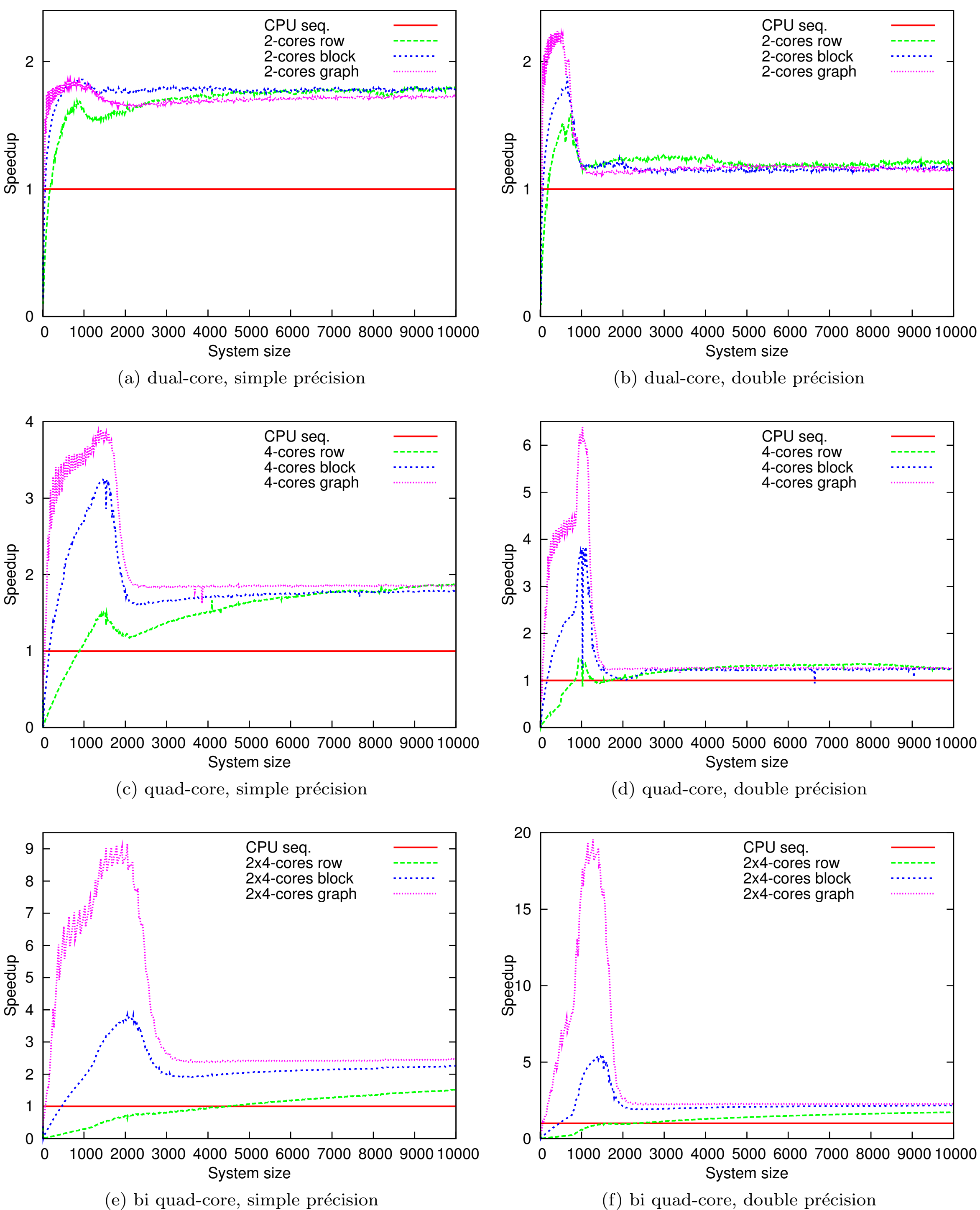
Nous avons également évalué notre algorithme sur une architecture présentant des temps d'accès mémoire non uniformes (NUMA). En effet, les plate-formes de grandes envergures tendent de plus en plus à être composées de plus d'un processeur, que ce soit au sein de la même machine, ou de façon distribuée. La différence pour cette architecture concerne l'étape d'initialisation, où chaque thread alloue et copie les parties de la matrice sur laquelle il va travailler, afin d'assurer une localité et des temps d'accès optimaux. Il faut pour cela s'assurer que l'association des tâches de façon round-robin affecte toujours le calcul d'une même ligne au même processeur, quitte à ce que certains cœurs de calcul n'aient pas de données à traiter à la fin de la matrice.

5.3.3 Mesures de performances

Nous avons mesuré le temps requis pour résoudre des LCP de différentes tailles, par rapport à une référence de base exécutée séquentiellement sur CPU. Les architectures suivantes ont été utilisées :

- Un *dual*-core Intel®Core™2 Duo CPU E6850 à 3.00 GHz (2 cœurs au total)
- Un *quad*-core Intel®Core™2 Extreme CPU X9650 à 3.00 GHz (4 cœurs au total)
- Un *bi quad*-core Intel®Core™2 Extreme CPU X9650 à 3.00 GHz (8 cœurs au total)
- Un *octo* dual-core AMD®Opteron™ Processor 875 à 2.2 GHz (16 cœurs au total)
- Une NVIDIA®GeForce™ GTX 280 GPU at 1.3 GHz (30, multiprocesseurs de 8-cœurs, soit 240 cœurs au total)

Les architectures Intel utilisent toutes une disposition de mémoire uniforme, tandis que le processeur *octo* dual-core fait partie des architectures NUMA, avec 4 plages mémoire de 8 Go de RAM, reliant chacune une paire de processeur.



(a) dual-core, simple précision

(b) dual-core, double précision

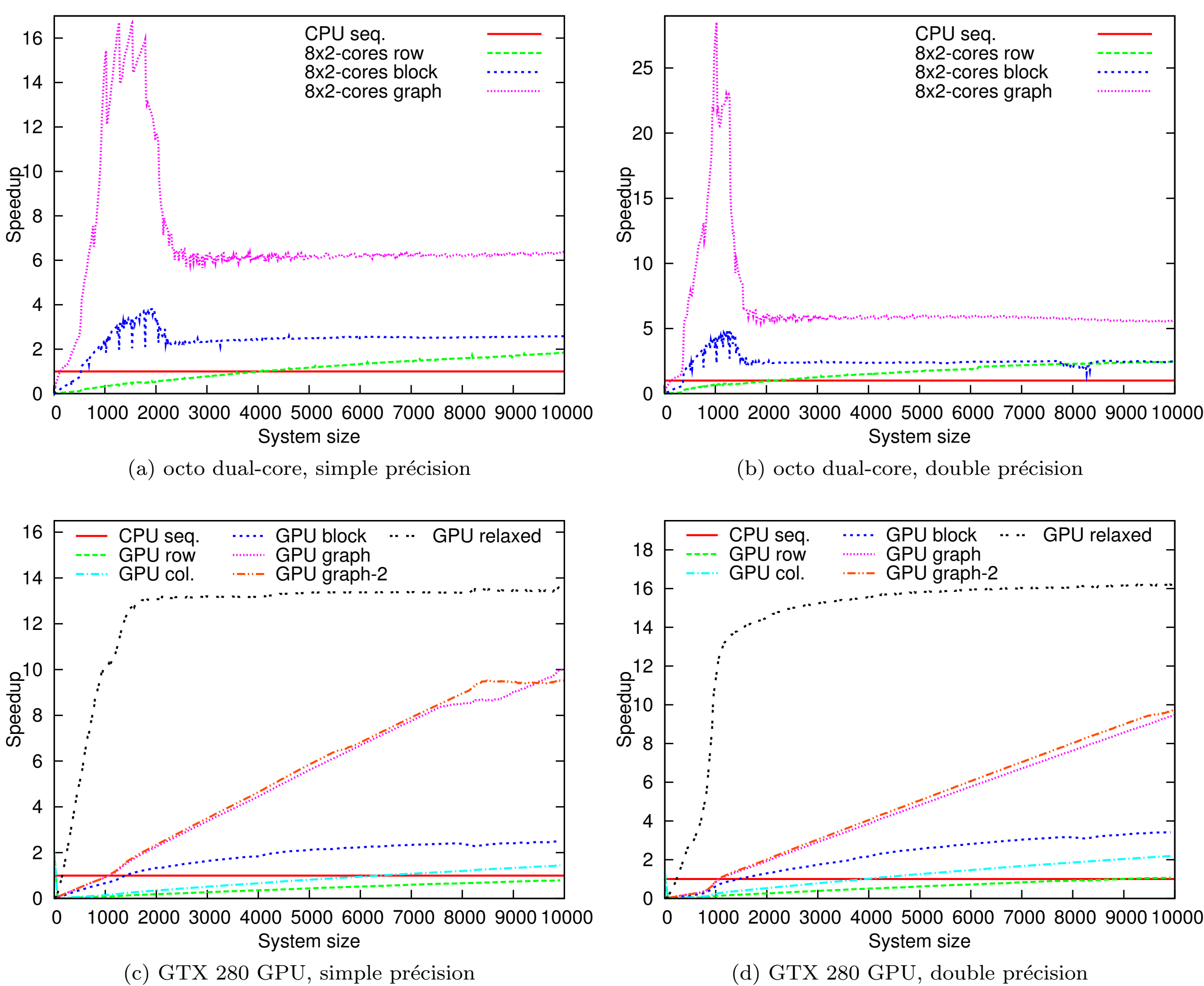
(c) quad-core, simple précision

(d) quad-core, double précision

(e) bi quad-core, simple précision

(f) bi quad-core, double précision

FIG. 5.4 – Mesures des temps de calcul sur les différentes architectures CPU parallèles, comparée à l'algorithme séquentiel sur un seul cœur.



(a) octo dual-core, simple précision

(b) octo dual-core, double précision

(c) GTX 280 GPU, simple précision

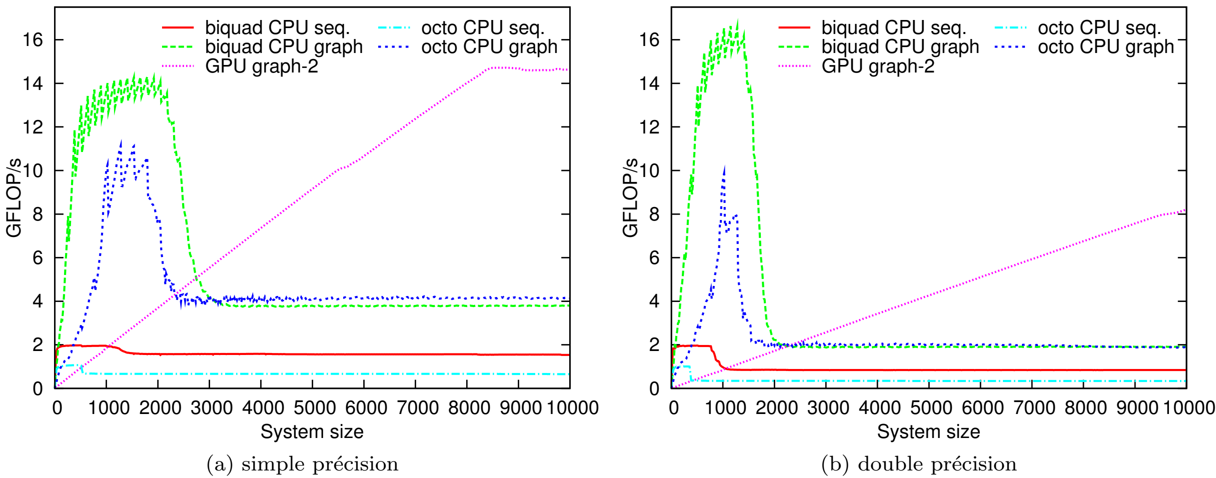
(d) GTX 280 GPU, double précision

FIG. 5.5 – Mesures du temps de calcul sur le GPU, comparé à l'algorithme séquentiel sur processeur.

Pour mesurer les performances, nous avons fixé à 100 le nombre d'itérations de l'algorithme du Gauss-Seidel, quelle que soit la qualité de la solution. Cependant nous maintenons les synchronisations et transferts vers le CPU car ils sont nécessaires pour déterminer (en fonction de l'erreur commise par la solution actuelle) si l'algorithme doit être poursuivi. Dans une application réelle, si l'algorithme converge rapidement les temps de calcul seront inférieurs à ceux présentés ici.

Les figues 5.4 et 5.5 présentent les facteurs d'accélération mesurées pour chaque architecture comparés à l'implémentation mono-thread séquentiel. Sur toutes les architectures à base de CPU, on observe un speedup "super-linéaire"[6] (jusqu'à 6× sur *quad*, 18× sur *biquad*, et 28 × sur *octo*) pour des matrices de taille 2000 × 2000. Il peut être expliqué par le fait que des matrices de cette taille ne peuvent plus contenir dans le cache d'un seul processeur, mais toujours dans le cache de plusieurs processeurs. Dans cette gamme de taille de matrice, la stratégie basée sur le graphe de dépendances est capable d'atteindre les meilleures performances. Quand la taille de la matrice augmente, le bus

[6] Une accélération plus grande que le nombre de processeurs parallèles.



(a) simple précision

(b) double précision

FIG. 5.6 – Nombre d'opérations par seconde réalisées sur chaque architecture.

mémoire devient le goulot d'étranglement, et la machine *bi-quad*, est à peine plus rapide que la parallélisation avec 2 threads. Seuls les systèmes NUMA comme *octo* sont capables d'atteindre de meilleures performances car ils possèdent plusieurs bancs mémoires qui ont chacun leur bande passante. L'accélération se stabilise autour de $6\times$ pour la stratégie axée sur les graphes, ce qui est proche du speedup optimal (le matériel possède 8 processeurs).

Sur GPU, les parallélisations *row* et *col* ne sont pas en mesure de réellement exploiter les unités de calculs disponibles. Le *bloc* algorithme surpasse l'implémentation CPU de référence, pour des matrices de plus de 1400×1400, mais elle ne peut pas réellement l'améliorer puisque le speedup est de $2,5\times$ pour des calculs en simple précision et de $3,4\times$ en double précision. La première implémentation de la parallélisation *graphe* permet d'atteindre des performances similaires pour les matrices de petites tailles. Il devient ensuite plus rapide pour des problèmes de grandes tailles, atteignant une accélération de $10\times$ en simple précision sur une matrice de taille 10000×10000, et $9,5\times$ en double précision. La version *graphe-2* algorithme, qui commence à calculer la ligne suivante pendant une synchronisation globale, est en mesure d'améliorer les performances de près de 10% en simple précision pour des matrices de taille 8500×8500, mais il devient alors moins efficace pour les grandes matrices. Par rapport à la version *relaxed* de l'algorithme, nous pouvons voir que les surcoûts généraux liés aux contrôles des dépendances restent très importants.

Finalement, le nombre d'opérations effectuées par seconde est résumé pour toutes les architectures dans la figure 5.6. Pour des matrices de petites tailles, les versions parallèles sur CPU sont extrêmement rapides, le maximum des performances est atteint pour une taille de 2800×2800 en simple précision et 2300×2300 en double précision. Le GPU est capable de surpasser toutes les implémentations parallèles sur CPU pour des matrices de grandes tailles, car les GPU possèdent une plus grande bande passante mémoire. Cependant, elles restent moins performantes que la version séquentielle pour les matrices très petites.

En conclusion, on peut dire qu'en fonction de la taille du problème, notre méthode de parallélisation permet d'obtenir une accélération significative sur les différentes architectures. En particulier, si le problème est de très grande taille, la parallélisation GPU permet de maintenir des performances proches du temps réel. Pour la suite du document, nous allons présenter l'autre aspect de la problématique en présentant des contributions pour calculer rapidement l'opérateur de Delasus que nous avons utilisé ici.

5.4 Calcul de la compliance sur GPU pour modèles spécifiques

L'algorithme du Gauss-Seidel, n'est pas le seul point bloquant dans l'étape de résolution des contacts. En effet, la construction d'un LCP dont la taille serait du même ordre que ce que nous venons de présenter, soulève d'autres problèmes plus importants. En effet, on a vu que pour poser le LCP, on avait besoin de calculer $\mathbf{W}$, qui nécessite de connaître l'inverse du système, ce qui demande un calcul encore plus complexe. On va ici présenter un ensemble de contributions qui ont été en partie publiées dans Courtecuisse et al. (2011c). On va d'abord proposer une optimisation pour le calcul de structure linéique (maillage de poutres en série) Ensuite, on présentera une parallélisation de la construction de $\mathbf{W}$. Enfin, on proposera une optimisation pour les modèles volumiques co-rotationnels, notamment pour gérer les changements topologiques.

5.4.1 Inverse d'une matrice bloc tri-diagonale sur GPU

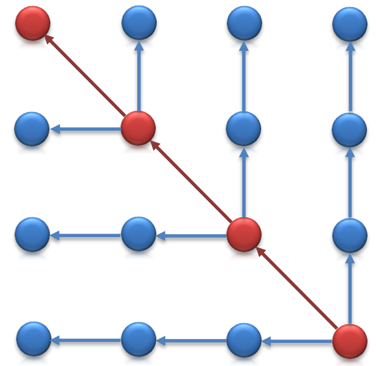
Dans la section 2.5.1, nous avons présenté le modèle de poutre, nous avons vu que la matrice du système produit une matrice Bande Tri-diagonale, dont l'inverse peut rapidement être calculé avec l'algorithme de Thomas. Cet algorithme est équivalent à trouver une décomposition $\mathbf{LU}$ de la matrice, ce qui revient dans ce cas à trouver les α et λ tels que :

$$\begin{pmatrix} a_0 & c_0 & 0 & 0 \\ b_1 & a_1 & c_1 & 0 \\ 0 & b_2 & a_2 & c_2 \\ 0 & 0 & b_3 & a_3 \end{pmatrix} = \begin{pmatrix} \alpha_0 & 0 & 0 & 0 \\ b_1 & \alpha_1 & 0 & 0 \\ 0 & b_2 & \alpha_2 & 0 \\ 0 & 0 & b_3 & \alpha_3 \end{pmatrix} \begin{pmatrix} 1 & \lambda_0 & 0 & 0 \\ 0 & 1 & \lambda_1 & 0 \\ 0 & 0 & 1 & \lambda_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.14}$$

Ce calcul est très efficace, puisqu'il présente une complexité linéaire en nombre de nœuds. Ainsi, même si la matrice change à chaque pas de temps on peut recalculer cette décomposition en temps réel[7]. Cependant, pour résoudre les contacts, il faut calculer explicitement les valeurs de $\mathbf{A}^{-1}$, qui est une matrice pleine. On peut faire cela assez facilement à partir de cette décomposition, en calculant successivement chaque ligne une à une Conte et Boor (1980). Ainsi, si n est le nombre de blocs de la matrice, l'inverse de $\mathbf{A}^{-1}$ s'obtient avec l'équation (5.15), où chaque coordonnée (i, j) correspond à un bloc de valeur (dont la taille dépend du nombre de degrés de liberté de chaque nœud, soit 6 valeurs pour le modèle de poutre).

Le calcul est alors fortement séquentiel, car les blocs sur la diagonale doivent être calculés les uns après les autres. De plus, pour calculer le bloc (i, j), il est nécessaire d'avoir calculé

[7]Évidemment on peut également utiliser cette factorisation pour calculer le mouvement libre des objets, et se passer de l'algorithme du Gradient Conjugué.



$$\left\{ \begin{array}{llrl} \mathbf{A}^{-1}_{i,j} & = & \lambda_i * \mathbf{A}^{-1}_{i+1,j} * \lambda^t_i, & \text{si } i < j \\ \mathbf{A}^{-1}_{i,j} & = & \alpha_i + \lambda_i * \mathbf{A}^{-1}_{i+1,j+1} * \lambda^t_i, & \text{si } i = j \\ & & & \text{et } j \neq n \\ \mathbf{A}^{-1}_{i,j} & = & \mathbf{A}^{-1}_{j,i}, & \text{si } i > j \\ \mathbf{A}^{-1}_{i,j} & = & \alpha_n, & \text{si } j = n \end{array} \right. \tag{5.15}$$

Fig. 5.7 – Graphe de dépendances pour le calcul de $\mathbf{A}^{-1}$. La diagonale est traitée sur CPU, et seul le calcul des blocs extra diagonaux est parallélisé sur GPU.

tous les blocs diagonaux (k, k) tel que $k >= i$, ainsi que tous les blocs de la ligne (i, k) tels que $k > j$. En d'autres termes, pour obtenir un élément sur une ligne il faut d'une part avoir calculé le bloc diagonal correspondant, et d'autre part avoir calculé tous les éléments dont la colonne est plus grande que celle de l'élément considéré.

Parallélisation sur GPU

La première remarque est que le calcul de la diagonale est une étape séquentielle où les blocs diagonaux sont calculés successivement, la mise à jour d'un groupe servant au calcul du suivant (voir figure 5.7). En revanche, une grande partie des dépendances proviennent du calcul de la diagonale, qui reste une opération rapide puisqu'elle comporte peu de valeur. Ainsi, nous avons choisi de calculer la diagonale sur CPU de façon séquentielle, le résultat est ensuite écrit dans un vecteur qui sera transféré sur le GPU. Ceci nous permet de retirer toutes les dépendances des blocs par rapport à la diagonale, et on peut maintenant calculer toutes les lignes en parallèles.

Pour effectuer le calcul sur GPU, on raisonne par groupe de dimension $t \times t$, où t est égal au nombre de degrés de liberté par nœud (6 pour le modèle de poutres déformables). Notre stratégie de parallélisation consiste à affecter le calcul d'un groupe de t lignes de la matrice, à un bloc CUDA. Pour cela, chaque bloc CUDA contient $t \times t$ threads qui vont chacun traiter une seule valeur par groupe de colonne. Chaque bloc CUDA calcule alors séquentiellement une ligne complète (mais aussi une colonne car la matrice est symétrique). Avec cette approche, on peut utiliser les synchronisations locales pour synchroniser les threads d'un même groupe de lignes. En effet, les synchronisations locales sont nécessaires car pour démarrer le calcul d'un nouveau groupe, il est important que tous les threads aient terminé de calculer le groupe précédent sur la même ligne.

Par ailleurs, on peut remarquer que le temps d'exécution des blocs n'est pas constant. En effet, le calcul de la première ligne comporte $n-t$ valeurs, alors que le calcul de la dernière n'en contient qu'une seule. Cependant, on se repose sur la façon dont sont ordonnées les tâches au sein du GPU, c'est-à-dire que dès qu'un multiprocesseur aura fini son travail, il exécutera une des lignes suivantes de façon transparente. Ainsi, le GPU va répartir automatiquement les calculs sur les multiprocesseurs, et on peut espérer occuper entièrement

le GPU si la matrice est grande. De plus, comme chaque bloc CUDA travaille sur des données différentes, cette parallélisation ne pose aucun problème d'écriture concurrente.

5.4.2 Calcul de l'opérateur de Delasus sur GPU

Une fois $\mathbf{A}^{-1}$ disponible sur le GPU, la construction du LCP implique encore de calculer le résultat de $\mathbf{J}_i\mathbf{A}_i^{-1}\mathbf{J}_i^T$ pour chaque objet i. Ces calculs pourraient être parallélisés par objet, mais leurs résultats doivent être additionnés pour obtenir $\mathbf{W}$ (voir équation 5.11). Ceci va provoquer des conflits d'écriture mémoire, et on ne peut donc pas calculer $\mathbf{W}_i$ en une seule étape. Dans notre implémentation GPU, nous traitons les objets séquentiellement. Pour chaque objet i, nous calculons la matrice locale $\mathbf{W}_i$ puis nous sommons le résultat dans la matrice finale $\mathbf{W}$.

Par ailleurs, on a vu que $\mathbf{J}$ est une matrice très creuse, et sa multiplication par $\mathbf{A}^{-1}$ peut être optimisée sur CPU, en ne calculant que les données nécessaires (voir section 4.2.3). Cependant, il est important d'effectuer ce calcul sur GPU, sous peine d'être obligé de transférer $\mathbf{A}^{-1}$ sur le CPU. Or, on sait que l'inverse du système est une matrice pleine et de grande taille, son transfert serait alors fortement pénalisant. L'utilisation de matrices creuses sur GPU n'est toutefois pas une chose facile. La première difficulté réside dans le fait de trouver une structure de données adaptées, qui permette de maximiser les lectures alignées, sans augmenter de façon considérable le stockage mémoire. Pour résoudre ce problème nous avons utilisé le format Jagged Diagonal Barrett et al. (1994). Celui-ci est constitué d'une matrice dense dont le nombre de lignes est égal[8] au nombre de lignes de $\mathbf{J}$, mais le nombre de colonnes est égal au maximum de valeurs non nulles dans une même ligne. On stocke ensuite les valeurs non nulles en continues dans la structure Jagged Diagonal, et pour chacune on aura alors besoin de retrouver l'indice de la colonne réelle. Pour cela, on utilise une deuxième matrice de même taille, qui contient les indices des colonnes (l'absence de donnée est codée par -1).

Pour calculer l'opérateur de Delasus sur GPU, nous utilisons deux kernels :

1. $\mathbf{H} = \mathbf{J}\mathbf{A}^{-1}$
2. $\mathbf{W}+ = \mathbf{H}\mathbf{J}^T$

où $\mathbf{H}$ est une matrice dense. Évidemment, pour minimiser les temps de calcul, il est important d'utiliser la même optimisation que sur CPU, en ne calculant que les colonnes de $\mathbf{H}$ qui seront réellement utilisées dans le second kernel. Pour cela on a besoin de connaître l'union de toutes les colonnes non vides de $\mathbf{J}$, et nous maintenons ce calcul sur CPU.[9] Pour le premier kernel, nous affectons un bloc CUDA pour le calcul de chaque valeur de $\mathbf{H}_{j,i}$, contenue dans l'union. Chacun des blocs comporte un ensemble de b threads qui effectue un produit scalaire, entre une ligne de $\mathbf{J}$ et une colonne matrice de $\mathbf{A}^{-1}$. De la même manière, nous associons un bloc CUDA au calcul d'un élément de $\mathbf{W}$. Ce kernel est très similaire au premier, à l'exception que le premier thread de chaque bloc CUDA accumule le résultat final dans la matrice $\mathbf{W}$.

[8] Le stockage initial ne stocke pas les lignes vides, et pour cela il utilise un vecteur supplémentaire pour retrouver les numéros de lignes réelles. Mais comme on sait que $\mathbf{J}$ ne contient pas de lignes non vides, les lignes de la structure Jagged Diagonal correspondent à celle de la matrice réelle.

[9] En effet, la matrice $\mathbf{J}$ provient des informations de détections de collisions, et de transferts à travers les mapping, qui sont des opérations rapides et difficiles à paralléliser. La matrice $\mathbf{J}$ est initialement construite sur CPU, et aucun transfert n'est nécessaire pour calculer l'union.

5.4.3 Implémentation pour le modèle co-rotationnel

L'avantage d'utiliser la méthode de *compliance warping* Saupin et al. (2008b) pour résoudre les contacts, est que la matrice $\mathbf{A}^{-1}$ est pré-calculée. Ainsi, même si $\mathbf{A}_0{}^{-1}$ est calculée sur CPU, on n'aura besoin de la transférer sur GPU qu'une seule fois. Seule l'application des rotations doit être effectuée à chaque pas de temps pour obtenir $\mathbf{W}$, mais on peut réorganiser les calculs de la manière suivante :

$$\mathbf{W} = \sum \underbrace{h(\mathbf{JR})\mathbf{A}_0{}^{-1}(\mathbf{JR})^T}_{\mathbf{W}_L} \tag{5.16}$$

En pratique, l'application des rotations sur la matrice $\mathbf{J}$ a l'avantage de ne pas modifier sa structure de données. En effet, un contact frottant influence tous les degrés de liberté d'un point, ce qui implique que $\mathbf{J}$ est constituée de sous blocs de données 3×3, pour un objet en $3D$. Or, comme $\mathbf{R}$ est une matrice bloc 3×3 diagonale regroupant les rotations de chaque nœud, le produit des deux matrices ne change pas la structure de $\mathbf{J}$. On va donc fonctionner en deux temps :

1. $\mathbf{H} = \mathbf{JR}$
2. $\mathbf{W} = \mathbf{W} + h\mathbf{H}\mathbf{A}_0{}^{-1}\mathbf{H}^T$

Pour réaliser le premier kernel, on transfert $\mathbf{J}$ sur GPU en utilisant la même structure de donnée basée sur la représentation Jagged Diagonal (voir section 5.4.2). Pour la matrice de rotation nous stockons ligne par ligne les valeurs non nulles dans un vecteur (on sait qu'il y a 3 valeurs non nulles sur chaque ligne). On obtient alors une nouvelle matrice $\mathbf{H}$, avec la même structure que $\mathbf{J}$. On peut ainsi ré-utiliser exactement les mêmes kernels que ce que nous venons de présenter dans la section précédente, et on obtient la matrice de compliance sur GPU.

Mise à jour en cas de découpe

L'inconvénient majeur de la méthode de *compliance warping*, est qu'elle ne permet pas de simuler des opérations de changements topologiques. Nous proposons maintenant, une solution pour étendre la méthode de *compliance warping* à la gestion des modifications topologiques en temps réel. Simuler une découpe nécessite de modifier le maillage de l'objet, et en réponse de mettre à jour toutes les matrices du système. La méthode proposée nous permet de calculer des forces de contacts cohérentes avec la découpe, qui prennent en compte les changements topologiques. Nous supposons que les mises à jour peuvent toutes être représentées avec trois actions élémentaires : la suppression d'éléments, la subdivision d'éléments, et l'ajout d'éléments. La figure 5.8 montre le processus que nous employons pour gérer les modifications topologiques.

Pour le mouvement libre, nous avons vu que le gradient conjugué peut ré-évaluer la raideur des éléments à chaque pas de temps, et ainsi de gérer automatiquement les modifications topologiques. Cependant, le modèle de contact repose sur le pré-calcul de la matrice de compliance dans la configuration initiale. Si cette matrice n'est pas recalculée à la suite d'une découpe, alors on va constater que les contacts sur une partie découpée influencent

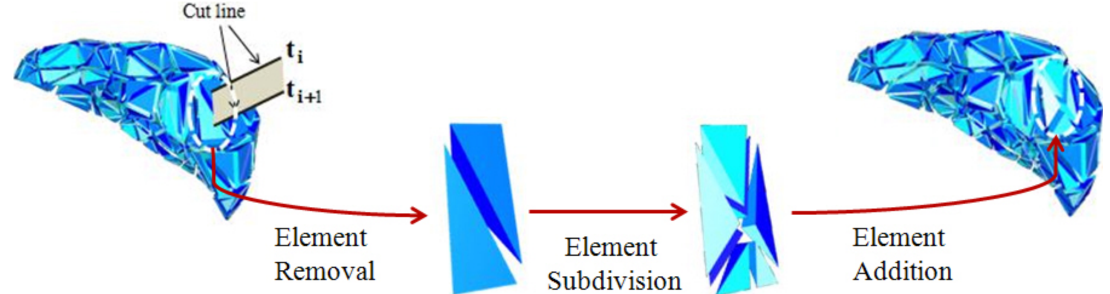


FIG. 5.8 – Le maillage est coupé, et la topologie est progressivement mise à jour.

toujours les parties connectées avant la découpe. Cette matrice n'est donc plus valide et elle doit être recalculée, mais un calcul direct est trop coûteux pour être effectué en temps réel.

Nous exploitons ici le fait que les changements topologiques ne modifient que quelques entrées des matrices $\mathbf{M}_0, \mathbf{B}_0$ et $\mathbf{K}_0$. Nous proposons alors de mettre à jour de façon incrémentale aussi bien le système initial que son inverse. Dans la suite, la matrice $\overline{\mathbf{A}}$ représente la matrice du système initial ($\overline{\mathbf{A}} = \mathbf{A}_0$), nous supprimons l'indice par souci de clarté.

Modification de la matrice du système Une fois que la topologie de l'objet déformable est modifié par la découpe, nous mettons à jour la matrice $\overline{\mathbf{A}}^{(i)} = (\mathbf{M}^{(i)} + h\mathbf{B}^{(i)} + h^2\mathbf{K}^{(i)})$ pour obtenir une nouvelle $\overline{\mathbf{A}}^{(i+1)}$, où l'exposant i représente la $i^{\text{ième}}$ modification. Nous définissons pour cela deux types de modification de la matrice $\overline{\mathbf{A}}^{(i)}$:

- Une correction, qui implique une modification au cours de laquelle la taille de la matrice reste constante.
- Une extension, qui implique une augmentation de la taille de la matrice.

Par conséquent, la mise à jour de la matrice $\overline{\mathbf{A}}^{(i+1)}$ peut être divisée en quatre blocs :

$$\overline{\mathbf{A}}^{(i+1)} = \begin{bmatrix} \overline{\mathbf{A}}_{11}^{(i+1)} & \overline{\mathbf{A}}_{12}^{(i+1)} \\ \overline{\mathbf{A}}_{21}^{(i+1)} & \overline{\mathbf{A}}_{22}^{(i+1)} \end{bmatrix} \tag{5.17}$$

Le bloc de la matrice $\overline{\mathbf{A}}_{11}^{(i+1)}$ a une dimension qui est exactement le même que la matrice $\overline{\mathbf{A}}^{(i)}$, ce qui correspond à la correction. Son calcul est réalisé en additionnant ou soustrayant les contributions des éléments ajoutés ou supprimés, de la matrice courante $\overline{\mathbf{A}}^{(i)}$:

$$\overline{\mathbf{A}}_{11}^{(i+1)} = \overline{\mathbf{A}}^{(i)} + \mathbf{G}\mathbf{N}\mathbf{G}^T \tag{5.18}$$

où $\mathbf{G}$, $\mathbf{G}^T$ sont les matrices creuses de globalisation permettant de répartir les nouvelles rigidités dans le système global. $\mathbf{N}$ est la matrice de correction qui est construite en additionnant la contribution des éléments supprimés et ajoutés. La dimension de la correction de la matrice $\mathbf{N}$ est proportionnelle au nombre de nœuds distincts touchés par la suppression et l'ajout d'éléments, mais elle est généralement très petite. En effet, comme le processus de découpe se déroule sur une longue période de temps, le nombre d'éléments modifiés par pas de temps restent limités (voir section 5.4.4).

Les matrices $\overline{\mathbf{A}}_{12}^{(i+1)}$, $\overline{\mathbf{A}}_{21}^{(i+1)}$, $\overline{\mathbf{A}}_{22}^{(i+1)}$ correspondent à l'extension de la matrice pour introduire dans le système mécanique les nœuds nouvellement créés. La dimension de la matrice carrée $\overline{\mathbf{A}}_{22}^{(i+1)}$, est proportionnelle au nombre de nœuds ajoutés pendant la découpe, ce qui est également très petit par rapport à la dimension de la matrice $\overline{\mathbf{A}}^{(i)}$. Les matrices $\overline{\mathbf{A}}_{12}^{(i+1)}$, $\overline{\mathbf{A}}_{21}^{(i+1)}$, sont quant à elles creuses et de tailles modérées. Leurs valeurs sont directement obtenues par la formulation d'éléments finis, que nous avons présentée dans le chapitre 2.

Mise à jour de l'inverse Pour calculer la mise à jour de la matrice inverse $\overline{\mathbf{C}}^{(i+1)}$, en fonction de la matrice modifiée $\overline{\mathbf{A}}^{(i+1)}$, nous employons également deux sortes d'algorithmes : Le premier est une décomposition modulaire de l'inverse de la matrice de compliance, et le second utilise la formule de Sherman-Morrison-Woodbury. On exprime la matrice de compliance avec 4 sous blocs, comme pour la matrice du système :

$$\overline{\mathbf{C}}^{(i+1)} = \begin{bmatrix} \overline{\mathbf{C}}_{11}^{(i+1)} & \overline{\mathbf{C}}_{12}^{(i+1)} \\ \overline{\mathbf{C}}_{21}^{(i+1)} & \overline{\mathbf{C}}_{22}^{(i+1)} \end{bmatrix} = \begin{bmatrix} \overline{\mathbf{A}}_{11}^{(i+1)} & \overline{\mathbf{A}}_{12}^{(i+1)} \\ \overline{\mathbf{A}}_{21}^{(i+1)} & \overline{\mathbf{A}}_{22}^{(i+1)} \end{bmatrix}^{-1} \tag{5.19}$$

Si aucun nœud n'est ajouté, alors il suffit de calculer $\overline{\mathbf{C}}^{(i+1)}$, en utilisant la formule de *Sherman-Morrison-Woodbury* :

$$\begin{aligned} \left(\overline{\mathbf{A}}_{11}^{(i+1)}\right)^{-1} &= \left(\overline{\mathbf{A}}^{(i)} + \mathbf{G}\mathbf{N}\mathbf{G}^T\right)^{-1} \\ &= \overline{\mathbf{C}}^{(i)} - \overline{\mathbf{C}}^{(i)}\mathbf{G}\big(\underbrace{\mathbf{N}^{-1} - \mathbf{G}\overline{\mathbf{C}}^{(i)}\mathbf{G}^T}_{\mathbf{Q}_A}\big)^{-1}\mathbf{G}^T\overline{\mathbf{C}}^{(i)} \end{aligned} \tag{5.20}$$

Où $\overline{\mathbf{C}}^{(i)}$ est la matrice de compliance avant la découpe. Calculer $\mathbf{Q}_A^{-1}$ est relativement rapide puisque sa dimension est égale au nombre d'éléments ajoutés, qui restent petits.

Si des nœuds sont ajoutés, on exprime chaque composante de la nouvelle matrice de compliance comme suit :

$$\begin{aligned} \overline{\mathbf{C}}_{11}^{(i+1)} &= \left(\overline{\mathbf{A}}_{11}^{(i+1)}\right)^{-1} + \left(\overline{\mathbf{A}}_{11}^{(i+1)}\right)^{-1} \overline{\mathbf{A}}_{12}^{(i+1)}\ \mathbf{Q}_C^{-1}\ \overline{\mathbf{A}}_{21}^{(i+1)}\ \left(\overline{\mathbf{A}}_{11}^{(i+1)}\right)^{-1} \\ \overline{\mathbf{C}}_{12}^{(i+1)} &= \left(\overline{\mathbf{C}}_{21}^{(i+1)}\right)^{T} = -\left(\overline{\mathbf{A}}_{11}^{(i+1)}\right)^{-1} \overline{\mathbf{A}}_{12}^{(i+1)}\ \mathbf{Q}_C^{-1} \\ \overline{\mathbf{C}}_{22}^{(i+1)} &= \mathbf{Q}_C^{-1} \end{aligned} \tag{5.21}$$

Avec $\mathbf{Q}_C = \left(\overline{\mathbf{A}}_{22}^{(i+1)} - \overline{\mathbf{A}}_{21}^{(i+1)}\ \left(\overline{\mathbf{A}}_{11}^{(i+1)}\right)^{-1}\ \overline{\mathbf{A}}_{12}^{(i+1)}\right)$, qui est appelé le complément de Schür de $\overline{\mathbf{A}}_{11}^{(i+1)}$. On constate alors que tous les blocs de la matrice de compliance peuvent être obtenus à partir de composants qui sont désormais connus. Suite à une série de multiplications matricielles, on peut en déduire la nouvelle matrice de compliance qui tient compte de modifications topologiques. Ceci est faisable en temps réel, tant que le nombre d'élément modifié reste petit.

Implémentation GPU

Le calcul de $(\overline{\mathbf{A}}^{(i+1)})^{-1}$ repose sur un ensemble d'opérations matricielles d'algèbre linéaire. Ces opérations présentent un fort potentiel de parallélisme, et peuvent être implémentées efficacement sur des architectures parallèles. Les matrices $\overline{\mathbf{A}}_{12}^{(i+1)}$, $\overline{\mathbf{A}}_{21}^{(i+1)}$, $\overline{\mathbf{A}}_{22}^{(i+1)}$ étant creuses et très rapides à calculer, nous maintenons leur calcul sur CPU.

Mise à jour avec Sherman-Morrison-Woodbury Cette étape est réalisée en évaluant d'abord la correction de matrice $\mathbf{Q}_A^{-1}$ sur le CPU. Cette matrice étant de petite taille, elle peut être transférée à un coût très limité sur le GPU. En remarquant que $\mathbf{G}\overline{\mathbf{C}}^{(i)} = \overline{\mathbf{C}}^{(i)}\mathbf{G}^T$ (en raison de la symétrie de la matrice de compliance), on peut mettre à jour $(\overline{\mathbf{A}}_{11}^{(i+1)})^{-1}$, en utilisant trois kernels comme suit (voir équation 5.20) :

1. $\mathbf{S} = \overline{\mathbf{C}}^{(i)}\mathbf{G}$
2. $\mathbf{T} = \mathbf{S}\mathbf{Q}_A^{-1}$
3. $(\overline{\mathbf{A}}_{11}^{(i+1)})^{-1} = \overline{\mathbf{C}}^{(i)} - \mathbf{T}\mathbf{S}^T$

Bien que les deux premières étapes pourraient être calculées sur le CPU en raison du petit nombre de données, une version GPU a l'avantage d'éviter des transferts entre CPU et GPU. Le premier kernel, repose sur une multiplication de matrice dense par une matrice creuse, et nous utilisons un format de données proche du CRS (voir section 3.2.1) pour stocker $\mathbf{G}$. La matrice $\mathbf{T}$ est une matrice dense, et le dernier kernel correspond au calcul le plus coûteux dans l'ensemble de la méthode présentée dans cette section. Il faut donc l'implémenter de façon optimisée, c'est pourquoi nous avons utilisé la routine **cublasSgemm** de (CUBLAS) qui correspond exactement à ce calcul.

Mise à jour de la compliance Lorsque des éléments sont ajoutés, nous commençons par calculer $\mathbf{Q}_C^{-1}$ sur CPU dans une petite matrice qui sera ensuite transférée sur GPU. On utilise ensuite un premier kernel qui calcule $\mathbf{Y} = \overline{\mathbf{A}}_{21}^{(i+1)} (\overline{\mathbf{A}}_{11}^{(i+1)})^{-1}$. Comme précédemment, ces opérations pourraient être réalisées sur CPU car les matrices ajoutées sont creuses, mais le fait d'utiliser un kernel permet de ne pas transférer $(\overline{\mathbf{A}}_{11}^{(i+1)})^{-1}$. Nous pouvons finalement mettre à jour la matrice de compliance au moyen de trois kernels GPU :

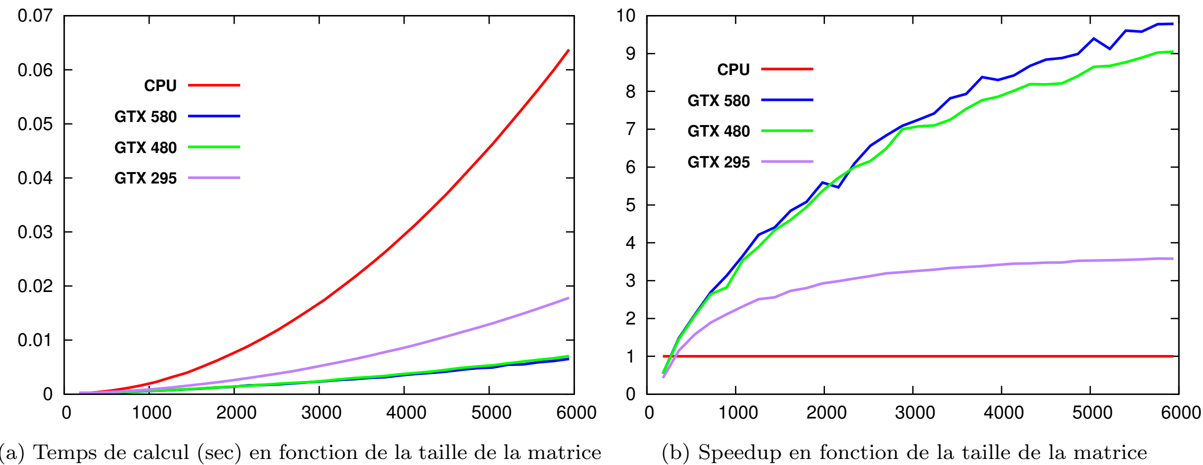
1. $\mathbf{X}_{12} = \mathbf{Y}\overline{\mathbf{A}}_{12}^{(i+1)}\mathbf{Q}_C^{-1}$
2. $\mathbf{X}_{11} = (\overline{\mathbf{A}}_{11}^{(i+1)})^{-1} + \mathbf{X}_{12}\ \overline{\mathbf{A}}_{21}^{(i+1)}\ \mathbf{Y}$
3. $\begin{cases} \overline{\mathbf{C}}_{11}^{(i+1)} = \mathbf{X}_{11} \\ \overline{\mathbf{C}}_{12}^{(i+1)} = (\overline{\mathbf{C}}_{21}^{(i+1)})^T = -\mathbf{X}_{12} \\ \overline{\mathbf{C}}_{22}^{(i+1)} = \mathbf{Q}_C^{-1} \end{cases}$

Notons que dans ce calcul, nous avons besoin d'une matrice temporaire $\mathbf{X}_{11}$, qui est de la même taille que $(\overline{\mathbf{A}}_{11}^{(i+1)})^{-1}$. Ainsi, même si on s'attend à obtenir des temps de calcul faibles, la consommation mémoire va limiter le nombre de nœuds que nous pouvons simuler (nous avons estimé la limite à environ 2000 nœuds).

5.4.4 Résultats

Nous évaluons maintenant les performances des méthodes présentées dans cette section.

Inverse d'une matrice BTD sur GPU Tout d'abord, nous présentons les temps de calcul pour obtenir l'inverse d'un matrice bande tri-diagonale sur GPU (voir section 5.4), afin de calculer les contacts sur le modèle de poutre. Nous avons comparé les temps de calcul sur différents GPU, à une implémentation séquentielle sur CPU. Les processeurs graphiques étant des Geforces GTX 295, 480 et 580, alors que le CPU était un Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz.



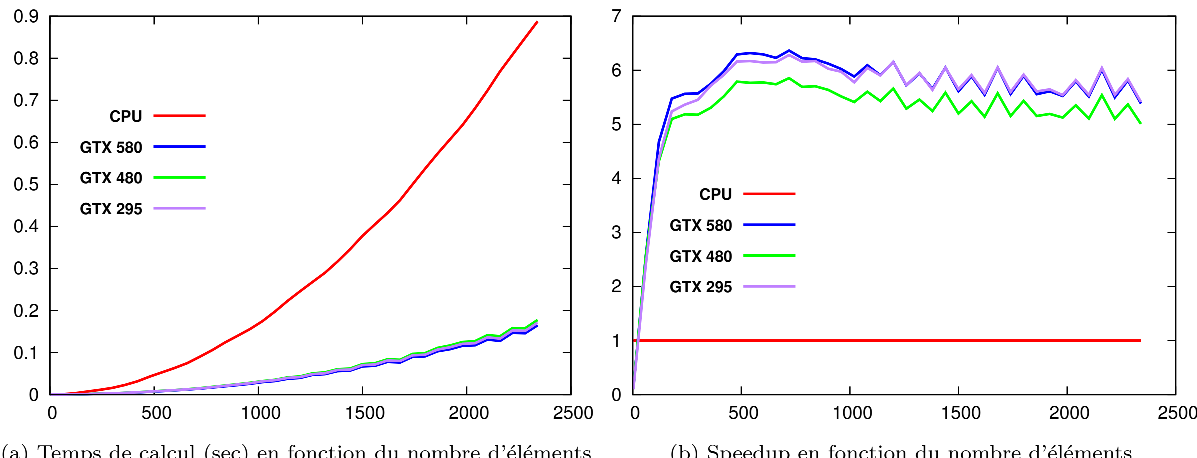
(a) Temps de calcul (sec) en fonction de la taille de la matrice (b) Speedup en fonction de la taille de la matrice

FIG. 5.9 – Comparaison des temps de calcul pour le calcul de l'inverse d'une matrice bande Tri-Diagonale sur GPU, en fonction de la dimension de la matrice.

Les temps de calcul présentés en figure 5.9a ont été réalisés en faisant varier le nombre de nœuds de la structure déformable. Les courbes présentent la dimension de la matrice, il faut donc diviser par 6 (qui est le nombre de degrés de liberté) pour obtenir le nombre de points de contrôle. Les temps mesurés incluent le temps d'inverser la diagonale de la matrice sur CPU, ainsi que son transfert sur le GPU avant d'exécuter le kernel.

On peut voir sur la figure 5.9b, que notre solution permet d'obtenir une accélération significative très rapidement (autour de 30 nœuds). Nous pouvons atteindre une accélération autour de 10× pour une matrice composée de 1000 points de contrôle. Et de façon plus intéressante, nous sommes toujours capables de simuler de tels objets en temps réel. Par ailleurs, on constate également que notre parallélisation tire parti des caches mémoire ajoutés sur les GPU GTX 480 et 580, puisque les résultats sont à peu près deux fois supérieurs à l'exécution sur la GTX 295.

Calcul de l'opérateur de Delasus sur GPU Nous présenterons maintenant une évaluation des performances pour le calcul de l'opérateur de Delasus sur GPU (voir section 5.4.2). Nous avons utilisé une matrice dense de taille 5376, et fait varier le nombre de contraintes. Nous avons réalisé nos expériences sur le même ensemble de machine que précédemment : les processeurs graphiques étaient des Geforces GTX 295, 480 et 580, et le CPU était un Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz.



(a) Temps de calcul (sec) en fonction du nombre d'éléments (b) Speedup en fonction du nombre d'éléments

FIG. 5.10 – Comparaison des temps[10] de calculs pour le calcul de l'opérateur de Delasus sur GPU, en fonction du nombre de contraintes.

On constate que l'on atteint très vite l'accélération maximale quelque soit le GPU (autour de 200 contraintes, ce qui représente 67 contacts frottants). L'accélération reste cependant modérée (autour de 6×), puisque la gestion (création et transfert sur GPU) de la matrice creuse sous le format Jagged Diagonal devient problématique si elle est de grande taille. Cependant, nous pouvons résoudre un nombre beaucoup plus important de contacts en temps réel que dans la version CPU.

Performances pour la découpe Pour évaluer la méthode de mise à jour de la matrice inverse (voir section 5.4.3), nous avons réalisé une simulation de découpe virtuelle d'un foie (voir figure 5.11). Le modèle du foie est maintenu par les conditions de Dirichlet (points fixes), une partie du modèle est découpée puis tombe dans le bol. Suite à la mise à jour de la matrice de compliance, les contacts sont correctement résolus puisque la partie découpée n'influence plus la partie attachée. Les expériences ont été executées sur un processeur Intel i7 950 3.07GHz Quad Core et une GPU GeForce GTX 295. Les calculs ont été réalisés en double précision pour minimiser les erreurs numériques, et la simulation se déroulait à une fréquence de 20 fps pendant la découpe, et à 30 FPS ensuite.

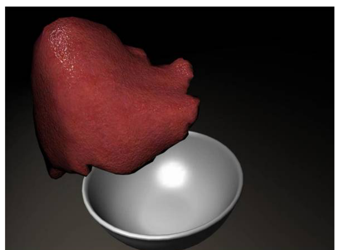
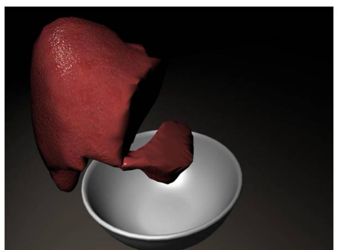
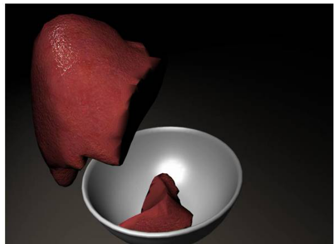
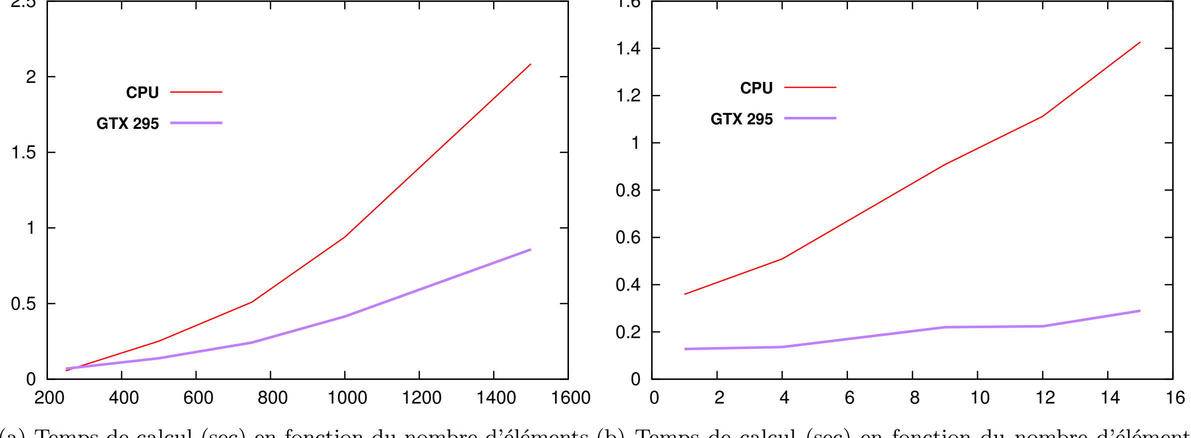
  

FIG. 5.11 – Simulation de découpe du modèle du foie, composée de 1000 tétraèdres.

La figure 5.12 montre les temps de calcul nécessaires pour mettre à jour la matrice de

[10]Ces temps ne comprennent pas le temps de transfert de $SystemMat^{-1}$ dans la mesure où cette matrice est transférée une seule fois pour la méthode de *compliance warping*, et est calculée directement sur GPU pour le BTD.

compliance pour un pas de temps sur le CPU et GPU. Il y a deux facteurs qui influencent les temps de calcul, le premier est le nombre de nœuds du modèle, et le second est le nombre d'éléments coupés en un seul pas de temps. Le graphique de gauche montre la variation du nombre de nœuds, allant de 250 à 1500, le nombre d'éléments coupés est alors fixé à 3. Le temps de calcul varie de 55.3 ms à 2085 ms sur le CPU et de 68.9 ms à 857.5 ms sur le GPU.



(a) Temps de calcul (sec) en fonction du nombre d'éléments (b) Temps de calcul (sec) en fonction du nombre d'éléments affectés

Fig. 5.12 – Gauche : Temps de calcul nécessaire pour mettre à jour la matrice de compliance en faisant varier le nombre de nœuds du maillage, mais en maintenant 3 éléments coupés par pas de temps. Droite : Temps de calcul nécessaire pour mettre à jour la matrice de compliance en faisant varier le nombre d'éléments coupés, sur un modèle du foie composé de 750 nœuds.

Le graphique de droite montre la variation du nombre d'éléments coupés en une seule étape de simulation, nous utilisons un modèle de foie composé de 750 nœuds. Les temps de calcul varient de 359.7 ms à 1426.1 ms sur le CPU, et de 127.6 ms à 289.5 ms sur le GPU, alors que le nombre d'éléments coupés varie de 1 à 15. Notre implémentation GPU permet alors d'obtenir une accélération d'environ 5×.

Une limitation importante de la méthode réside dans le fait que les erreurs numériques s'accumulent au fil des découpes. Ainsi, il est important de garantir une très bonne qualité de maillage pendant les modifications géométriques de la topologie. Une autre limitation réside dans le manque de généricité de la méthode de *compliance warping*. En effet, elle se base sur une approximation de l'inverse dans le cas où on utilise un modèle co-rotationnel. Ainsi, la méthode n'est pas généralisable à d'autres modèles non linéaires.

5.5 Construction du LCP basée sur le préconditionneur

Dans cette section, nous présentons une nouvelle méthode Courtecuisse et al. (2011a) pour calculer en temps réel, le comportement physique de plusieurs tissus mous en collision, qui tient compte du couplage mécanique entre les contacts. L'accélération est obtenue par l'utilisation du préconditionneur présenté en section 3.4, qui est mis à jour à basse fréquence. Nous verrons ici que le préconditionneur permet également de calculer en temps

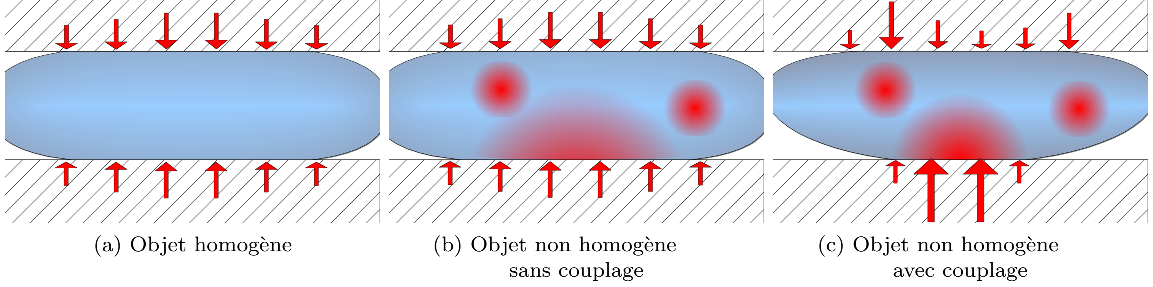
réel une réponse précise et optimisée des contraintes.

Par ailleurs, nous verrons aussi que cette approche présente également le grand intérêt d'être indépendante du modèle de déformation. Notamment, les tests que nous avons réalisés montrent que la méthode peut s'utiliser avec des modèles hyperélastiques. Enfin, nous montrerons que cette réponse à la collision peut également tenir compte de l'hétérogénéité des tissus et des changements topologiques.

5.5.1 Réponse avec le couplage mécanique correct

On rappelle que pour calculer le mouvement libre des objets (voir section 5.2.2), nous utilisons une version GPU d'un algorithme du gradient conjugué préconditionné. Le préconditionneur est obtenu en calculant une factorisation exacte de la matrice du système, de façon désynchronisée par rapport à la simulation (voir section 3.4). L'idée de base de cette contribution, repose sur la ré-utilisation du préconditionneur pour calculer la réponse aux contacts. En effet, comme le préconditionneur représentait une très bonne approximation pour le CG, il doit également fournir une très bonne approximation de la mécanique interne des objets.

On peut voir sur la figure 5.13 qu'il est important de prendre en compte le couplage mécanique correct entre les contacts. En effet, la répartition de la force de réponse est complètement dépendante des zones de rigidités des objets[11]. Or, puisque le préconditionneur est disponible, et est obtenu à faible coût du point de vue de la simulation, nous proposons de le réutiliser pour calculer la réponse aux contacts.



(a) Objet homogène (b) Objet non homogène sans couplage (c) Objet non homogène avec couplage

FIG. 5.13 – Répartition de la force pour des objets homogènes et hétérogènes sur lequel les collisions sont résolues avec et sans couplage mécanique. Les parties plus rigides sont indiquées en rouge.

Simulation de plusieurs objets déformables

Dans le chapitre 3, nous avions utilisé le préconditionneur pour simuler un seul objet. Cependant, comme nous nous intéressons maintenant aux problèmes de contacts, il est essentiel de simuler plusieurs objets en même temps. Or, comme chaque objet possède un état mécanique différent, nous devons posséder autant de préconditionneur que d'objets simulés, et cela implique d'exécuter plusieurs threads pour mettre à jour les différents préconditionneurs. Il est donc important de choisir une librairie *thread-safe*, c'est-à-dire

[11]Notons toutefois que la formulation en éléments finis tendra à ramener les objets dans une configuration correcte (qui tient compte des hétérogénéités). Cependant cela nécessite de réaliser de nombreuses intégrations pour converger vers une solution satisfaisante. Ainsi, il faudra utiliser des pas de temps très petits, ce qui rend la simulation impossible en temps réel.

que la librairie peut calculer plusieurs factorisations simultanées[12].

Par ailleurs, certaines simulations demandent d'effectuer le calcul des déformations d'un nombre important de structures anatomiques. Dès lors, on peut dépasser le nombre de processeurs disponibles sur la machine. Dans ce cas, le fait de lancer un grand nombre de processus en parallèle sur le CPU peut être pénalisant pour les performances. Une solution serait de lancer un nombre fixe de processus à l'initialisation de la simulation, chacun d'entre eux réalisant le calcul de plusieurs préconditionneurs tour à tour. On pourra alors mettre en place un système de priorité, pour recalculer le plus vite possible les préconditionneurs les moins performants (en se basant sur le nombre d'itérations du CG par exemple). D'autre part, on sait que même si le préconditionneur est plus long à calculer, on peut toujours limiter sa divergence en appliquant les rotations autour de l'approximation.

Bien évidemment, si le nombre de structures anatomiques simulées reste petit, et que l'on possède suffisamment de processeurs, la solution la plus avantageuse est d'affecter le calcul d'un préconditionneur à un seul processeur. Pour simplifier, c'est ce que nous supposons ici. Par ailleurs, il est intéressant de remarquer que lorsque nous allons calculer la réponse au contact, les préconditionneurs n'auront alors pas forcément été calculés à partir de la même étape de la simulation. Ceci nous permettra de répartir sur différents pas de temps, les transferts entre CPU et GPU nécessaires à la version GPU de la méthode.

Construction du LCP

Dans le chapitre 3, nous avions choisi de baser le calcul du préconditionneur sur une factorisation exacte de la matrice. En effet, on a vu qu'une factorisation exacte fournit la meilleure approximation de l'inverse du système. Nous avons cependant choisi d'utiliser une factorisation de type $\mathbf{LDL}^T$ pour résoudre les contacts. Ce choix est motivé par deux raisons, la première est que cette décomposition est numériquement plus stable. Ceci joue un rôle important pour la simulation des contacts qui peut dégrader le conditionnement du système en introduisant des déformations très locales. La seconde raison, est que la diagonale de $\mathbf{L}$, est uniquement composée de 1. Nous verrons que cette propriété est exploitable pour la parallélisation GPU, et va nous permettre de retirer un grand nombre de synchronisations. Par conséquent, l'expression complète du préconditionneur que nous utilisons est :

$$\mathbf{P} = \mathbf{R}^T \ \mathbf{LDL}^T \ \mathbf{R} \tag{5.22}$$

Où $\mathbf{R}$ est la matrice comprenant les rotations apparues depuis la dernière mise à jour, et $\mathbf{LDL}^T$ est la factorisation du système la plus récente. Ainsi, pour chaque objet on peut obtenir une bonne approximation de l'opérateur $\mathbf{W}$ de l'équation (5.5) par :

$$\mathbf{JA}^{-1}\mathbf{J}^T \approx \mathbf{J}(\mathbf{R}^T\mathbf{LDL}^T\mathbf{R})^{-1}\mathbf{J}^T \tag{5.23}$$

Or, si le preconditionneur reste une bonne approximation on peut espérer que la matrice obtenue soit très proche de la solution exacte. Cependant, on se rend compte que la connaissance de la factorisation n'est pas suffisante pour calculer directement l'opérateur

[12] Ce n'était pas le cas de la librairie TAUCS, la première difficulté a été de la modifier pour qu'elle n'utilise plus de variable **static**. Toutes les variables utilisées au sein de la librairie ont alors été dupliquées dans chaque processus.

de Delasus. En effet, on voit que la construction de $\mathbf{W}$ nécessite de connaître explicitement les valeurs de l'inverse du système, or nous ne possédons que sa factorisation. Pour remédier à ce problème, on peut remarquer que la résolution du système linéaire formé par le préconditionneur $\mathbf{P}$ et un vecteur $\boldsymbol{b}$ quelconque, revient à multiplier la solution par l'inverse de la factorisation.

$$\mathbf{P}\ \boldsymbol{x} = \boldsymbol{b} \quad \Leftrightarrow \quad \boldsymbol{x} = (\mathbf{R}^T\mathbf{LDL}^T\mathbf{R})^{-1}\boldsymbol{b} \tag{5.24}$$

Ainsi, en décomposant colonne par colonne la matrice $\mathbf{J}^T$, on peut alors calculer le produit de cette matrice par l'inverse du préconditionneur comme suit :

$$\mathbf{S} = \mathbf{P}^{-1}\ \mathbf{J}^T \quad \Leftrightarrow \quad \sum_{i=0}^{n_c} \mathbf{P}\ \mathbf{S}_{i,_} = \mathbf{J}^T_{i,_} \tag{5.25}$$

où $\mathbf{J}^T_{i,_}$ représente la $i^{\text{ème}}$ colonne de $\mathbf{J}^T$, n_c est le nombre de contraintes. On obtient alors le résultat de ce produit dans une matrice dense $\mathbf{S}$ qui a les mêmes dimensions que $\mathbf{J}$. Enfin, pour calculer $\mathbf{W}$, il reste à multiplier ce résultat par la jacobienne des contacts $\mathbf{J}$ (en exploitant son faible taux de remplissage).

$$\mathbf{W} \approx \mathbf{JS} \tag{5.26}$$

Il est important de remarquer que l'approche proposée calcule une approximation du couplage mécanique. Ceci est différent de l'approche utilisée dans le chapitre 3 où les itérations du CG assuraient la convergence de l'algorithme. Malgré tout, nous montrerons que cette approximation conduit à un résultat très proche du calcul exact, et présente un très bon compromis entre précision et temps de calcul.

La principale limitation réside dans le calcul de l'équation (5.25), qui implique d'effectuer autant de résolutions de systèmes linéaires que le nombre de contraintes. Or, si on effectue ces résolutions séquentiellement, on sera très vite limité par le nombre de contraintes (ou de contacts) pouvant être simulées en temps-réel. Ainsi, nous allons proposer une parallélisation GPU qui va permettre d'augmenter très largement le nombre de contacts pouvant être simulés en temps réel.

5.5.2 Parallélisation sur GPU

Le calcul d'une résolution de système linéaire basé sur une matrice creuse, est très difficile à paralléliser sur GPU. Le faible taux de remplissage et l'irrégularité des données empêche les lectures alignées. De plus, les nombreuses dépendances font qu'il est très difficile d'extraire du parallélisme. Cependant, pour calculer l'équation (5.25), nous devons effectuer de multiples résolutions qui ne présentent aucune dépendance entre elles. Ainsi, on peut facilement extraire du parallélisme en les exécutant en parallèle sur le processeur graphique. Le calcul peut alors être décomposé en trois étapes :

1. $\mathbf{H} = \mathbf{JR}$
2. $\mathbf{S} = (\mathbf{LDL}^T)^{-1}\ \mathbf{H}^T$
3. $\mathbf{W}+ = \mathbf{HS}$

On commence par appliquer les rotations (survenues depuis la dernière mise à jour) à la jacobienne des contacts. On a vu que cette opération est rapide, puisque les matrices sont creuses (bloc-diagonale pour $\mathbf{R}$, et creuses pour $\mathbf{J}$). Pour cette raison, nous maintenons cette étape sur le CPU, afin d'exploiter les structures de données creuses.

Avant d'exécuter la seconde étape, les valeurs non nulles de $\mathbf{H}$ sont copiées dans une matrice dense, de sorte que chaque ligne représente un vecteur solution $\mathbf{J}_{i,-}^T$ de l'équation (5.25). Cette matrice dense est ensuite copiée sur GPU. Par ailleurs, la factorisation de la matrice (qui est stockée dans le format CRS) doit également être transférée[13]. Cependant, ce transfert n'est nécessaire que lorsque le préconditionneur a été modifié, il n'influencera alors qu'un sous ensemble de pas de temps et devrait avoir une influence limitée.

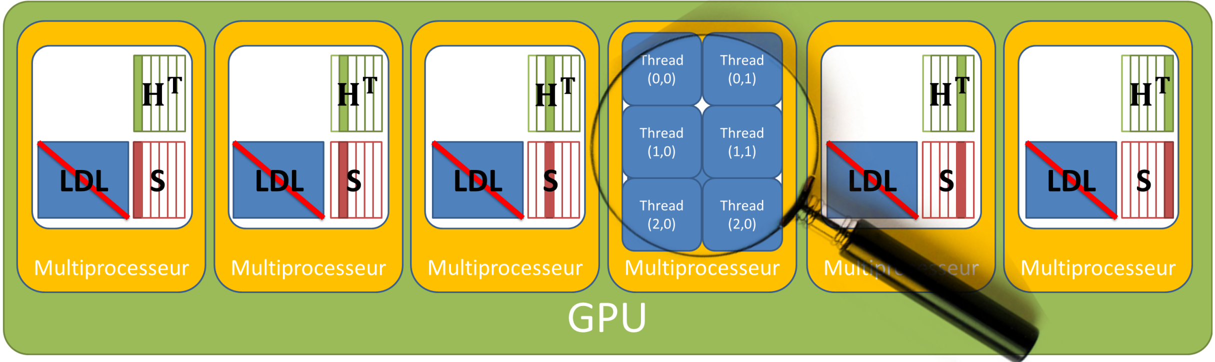


FIG. 5.14 – Parallélisation à deux niveaux des multiples résolutions de systèmes linéaires sur le GPU.

Une fois les données disponibles sur le processeur graphique, on cherche alors à calculer $\mathbf{S}$ sur GPU. Ceci correspond à appliquer le préconditionneur en résolvant successivement deux systèmes triangulaires par contrainte. Pour résoudre ces deux systèmes d'équations, on utilise deux kernels, l'un pour l'application de $\mathbf{L}$ et l'autre pour l'application de $\mathbf{L}^T$. Ainsi, chaque kernel va résoudre plusieurs fois le même système triangulaire, sur des vecteurs solutions différents (voir équation 5.25). Nous proposons d'effectuer la résolution d'un système triangulaire avec un seul bloc CUDA (voir figure 5.14). On utilisera alors un seul multiprocesseur par résolution, mais comme nous avons potentiellement beaucoup de contraintes on peut ainsi espérer occuper complètement le GPU. De plus, ce découpage simplifie le problème, puisqu'il permet d'utiliser les synchronisations locales au sein des blocs CUDA. En effet, dans chaque résolution un grand nombre de données peuvent être traitées en parallèle. Ainsi, nous utilisons un second niveau de parallélisme qui est bien adapté aux architectures parallèles telles que les GPU. Nous détaillerons cette deuxième parallélisation dans la section suivante.

Une fois ce calcul terminé, le résultat est ensuite stocké dans une matrice dense $\mathbf{S}$ de même dimension que $\mathbf{J}$. Il reste enfin à multiplier la matrice des contraintes par $\mathbf{S}$, puis d'ajouter le résultat final à $\mathbf{W}$. Pour cela, on peut utiliser le CPU en tirant parti du faible taux de remplissage de $\mathbf{H}$, ou éventuellement le GPU en utilisant les représentations denses (qui sont déjà disponibles sur GPU) et la primitive **cublasSgemm** de (CUBLAS), qui fournit

[13]Notons qu'il est impératif de réaliser ce transfert dans la boucle de simulation, car les transferts mémoire en CUDA ne peuvent être initiés que par un seul thread dans l'application.

une implémentation optimisée. L'avantage de la deuxième approche est que l'opérateur de Delasus sera finalement disponible directement sur le GPU.

Résolution de systèmes triangulaires sur GPU

Nous présentons maintenant le cœur de notre méthode, qui consiste à paralléliser la résolution de système triangulaire sur GPU. Tout d'abord, il faut remarquer que les calculs à effectuer dépendent de la factorisation choisie. Nous avons implémenté notre méthode pour deux factorisations l'une sous forme $\mathbf{LDL}^T$ et l'autre sous forme $\mathbf{LL}^T$. La seule différence réside dans l'application de l'inverse de la diagonale entre les calculs, ce qui est rapide et trivial à paralléliser. La méthode que nous présentons reste donc valable pour ces deux factorisations, mais nous verrons qu'il est possible d'effectuer des optimisations supplémentaires pour la factorisation $\mathbf{LDL}^T$.

Nous nous intéressons donc ici à la résolution successive de deux systèmes triangulaires. Les résolutions avec la triangulaire inférieure seront appelées *lower solve*, alors que *upper solve* désignera les résolutions avec la triangulaire supérieure. Pour chacune, le calcul à effectuer est similaire à un pivot de Gauss, qui consiste à éliminer une à une les contributions d'une même ligne pour se ramener à un système diagonal. Pour le lower solve par exemple, on peut calculer les composantes $\boldsymbol{y}_i$ de la solution avec la formule suivante :

$$\boldsymbol{y}_j = \frac{\boldsymbol{b}_j - \sum_{i=0}^{i<n} (\boldsymbol{y}_i * \mathbf{L}_{i,j})}{\mathbf{L}_{j,j}} \tag{5.27}$$

On constate alors que le calcul de la composante j nécessite d'avoir calculé au préalable toutes les valeurs $\boldsymbol{y}_i$, tel que $i < j$. En d'autres termes, le calcul d'une nouvelle ligne nécessite d'avoir calculé toutes les lignes précédentes, et on ne peut mettre à jour qu'une seule valeur à la fois. Ceci rend cet algorithme difficile à paralléliser, puisque de nombreuses synchronisations seront nécessaires. En effet, le calcul de chaque ligne, implique :

1. Accumuler toutes les contributions de la ligne.
2. Soustraire ce résultat à la solution actuelle.
3. Diviser la valeur par la diagonale de la triangulaire, pour obtenir le résultat final.

Pour la factorisation $\mathbf{LDL}^T$, on a toujours $\mathbf{L}_{j,j} = 1$, ce qui permet de retirer la troisième étape (et les synchronisations associées). Dans la suite, nous appliquerons notre solution à cette factorisation, mais il faut garder en mémoire que des synchronisations supplémentaires seront nécessaires si l'on veut l'utiliser pour la factorisation $\mathbf{LL}^T$.

Par symétrie, le upper solve conduit à une équation très similaire qui présente les mêmes dépendances, à la différence près que les inconnues sont accédées de bas en haut :

$$\boldsymbol{x}_j = \frac{\boldsymbol{y}_j - \sum_{i=n}^{i\geq 0} (\boldsymbol{x}_i * \mathbf{L}_{j,i})}{\mathbf{L}_{j,j}} \tag{5.28}$$

Nous cherchons maintenant à résoudre ces équation sur GPU.

Parallélisation sur GPU On rappelle que nous utilisons un seul bloc CUDA par résolution de système triangulaire, et que chaque bloc CUDA effectue la résolution du même système triangulaire sur des vecteurs. Avec ce schéma, on peut donc utiliser les synchronisations locales pour respecter les dépendances de calculs au sein des résolutions.

Pour chaque résolution le calcul de toutes les contributions d'une même ligne (ou d'une même colonne pour le lower solve) peut être fait en parallèle (voir figure 5.15). Ainsi, pour maximiser les performances on utilise plusieurs threads par bloc, chacun calculant la contribution d'une ligne (ou d'une colonne pour le lower solve). Entre chaque ligne, nous utiliserons une synchronisation locale pour respecter les dépendances. La première chose que l'on peut remarquer est qu'on ne peut pas exploiter un grand nombre de threads avec cette parallélisation. En effet, comme une seule ligne (ou colonne) de la matrice peut être traitée à un instant donné, et que la matrice est creuse, il est inutile de lancer un plus grand nombre de threads par bloc que le nombre de valeurs non nulles par ligne.

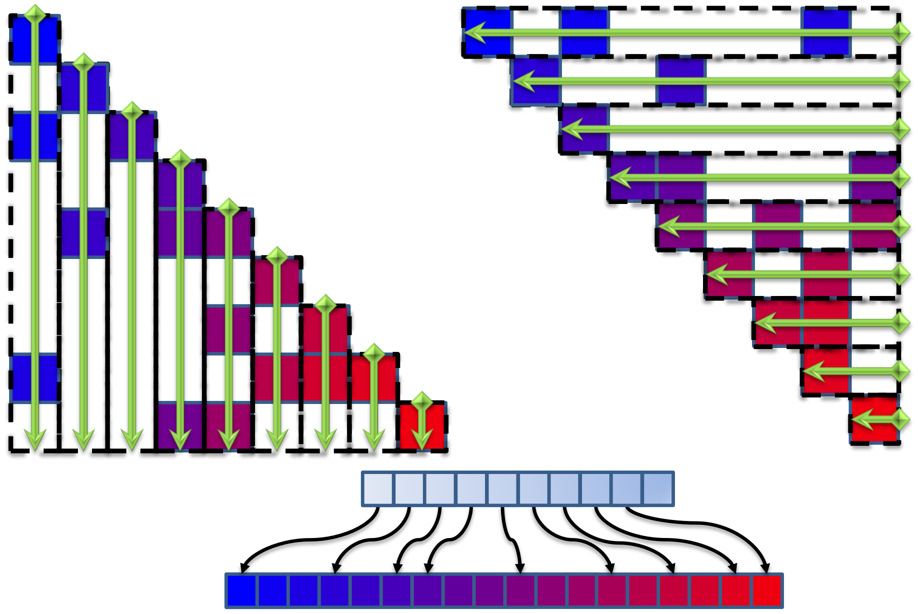


Fig. 5.15 – Résolution du lower solve (gauche) et upper solve (droite) sur GPU, à partir d'une matrice stockée sous le format CRS (bas). Les flèches symbolisent les données traitées en parallèle. Les losanges aux extrémités symbolisent une synchronisation.

Par ailleurs, que ce soit pour la factorisation $\mathbf{LDL}^T$ ou la $\mathbf{LL}^T$, on possède en réalité une seule représentation de la matrice sous le format CRS. En effet, l'une peut être facilement déduite de l'autre, puisque c'est sa transposée (dans notre implémentation, il s'agit de la représentation de la triangulaire supérieure qui correspond à $\mathbf{L}^T$). Cependant, le format CRS oblige à lire la matrice en ligne, ce qui génère des problèmes pour accéder aux données dans notre version parallèle.

Pour le lower solve, le format de stockage nous oblige à lire les données en colonnes. En effet, la matrice inférieure $\mathbf{L}$ est en réalité la transposée de la matrice qui est disponible sur le GPU $\mathbf{L}^T$. Or, une lecture en ligne dans $\mathbf{L}^T$, est équivalent à une lecture en colonne dans

L. On propose donc d'utiliser une parallélisation par colonne similaire à ce que nous avions proposé dans la section 5.3.1 (à la différence que notre matrice est creuse). Ainsi, chaque thread calcule l'accumulation de la nouvelle force sur des inconnues différentes, et on évite ainsi les problèmes d'écritures concurrentes. De plus, comme les données sont accédées de façon continue, chaque thread peut réaliser des lectures alignées dans la matrice CRS (mais doit écrire de façon non alignée[14] dans le vecteur solution).

Pour le upper solve, le format de stockage CRS nous impose une lecture des données en ligne. On peut alors utiliser une parallélisation par ligne similaire à la section 5.3.1, mais cette parallèlisation requiert une réduction parallèle pour chaque ligne. En effet, comme chaque thread calcule une partie de la même solution, il est nécessaire de rassembler les données pour mettre à jour la solution. Évidemment, ceci va largement influencer les performances, et on devra veiller à optimiser les réductions parallèles au sein d'un même bloc (voir annexe C). On pourra également organiser les threads pour qu'ils accèdent aux données de façon alignée sur la matrice, mais également sur le vecteur solution.

Un autre aspect essentiel pour les performances (que ce soit pour le lower solve ou le upper solve), consiste à minimiser la quantité de mémoire consommée par chaque thread. En effet, comme la stratégie de parallélisation repose en grande partie sur les multiples résolutions réalisées par différents blocs CUDA, il est important que le GPU puisse exécuter le maximum de blocs simultanément. Or, on sait qu'un seul multiprocesseur peut exécuter plusieurs blocs simultanément, si la consommation mémoire de tous les threads est plus petite que la mémoire totale disponible sur le matériel.

[14] Une optimisation que nous avons implémentée consiste à pré-charger en mémoire partagée une partie de la solution. En effet, les valeurs du vecteur solution proche de la diagonale en cours de traitement sont susceptibles d'être souvent utilisées.

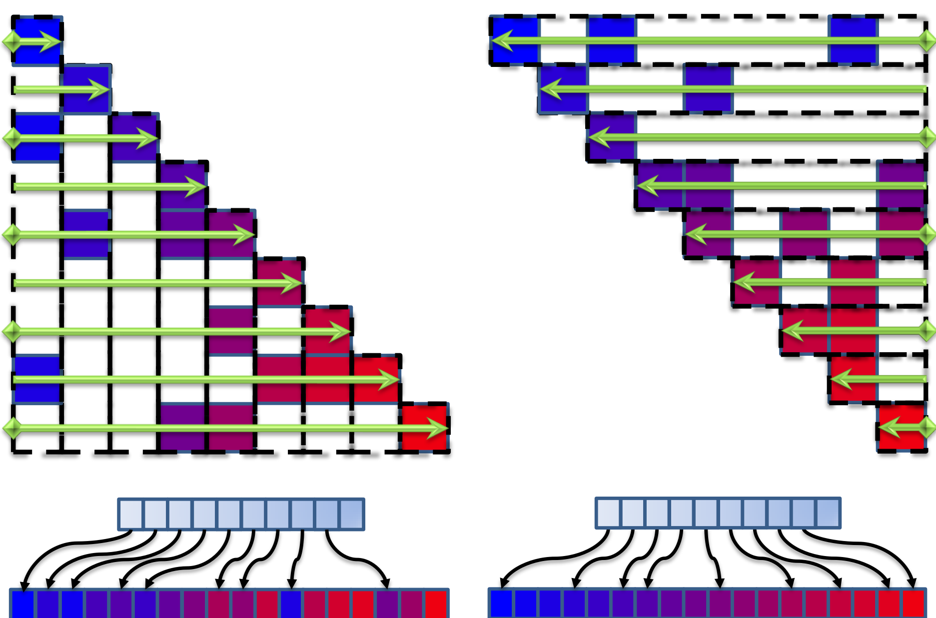


Fig. 5.16 – Optimisations du lower solve (gauche) et du upper solve (droite) sur GPU, à partir de la matrice initiale et de sa transposée stockées dans le format CRS (bas). Les flèches symbolisent les données traitées en parallèle. Les losanges aux extrémités symbolisent une synchronisation.

Optimisations et regroupements Nous proposons maintenant d'augmenter le nombre de threads par bloc. De façon similaire à la parallélisation par bloc de la section 5.3.1, on peut calculer les accumulations de plusieurs lignes simultanément, et diminuer le nombre de synchronisations. Cependant, nous devons ici utiliser une parallélisation par groupes de lignes. En effet, si on accède aux données en colonnes, on ne peut pas prédire l'ordre des indices de lignes de la représentation creuse. Ainsi, si plusieurs colonnes sont calculées en parallèle, on peut éventuellement avoir plusieurs threads qui vont écrire simultanément leurs contributions sur la même valeur. Ainsi, en raison du format CRS, cette optimisation n'est pour l'instant envisageable que pour le upper solve.

Le kernel produit est alors composé de deux parties, la première a pour objectif de calculer les contributions se trouvant en dehors du bloc diagonal, alors que la seconde met à jour les composantes de la solution. La difficulté réside dans le fait que l'on ne peut pas prévoir avant de lire la matrice, si les données appartiennent ou non au bloc diagonal.

La stratégie que nous avons adoptée consiste à lire l'intégralité du groupe de lignes, si les données appartiennent au bloc diagonal alors elles sont simplement copiées dans un premier tableau en mémoire partagée. En revanche, si elles n'appartiennent pas au bloc diagonal, alors leurs contributions sont accumulées dans un second tableau en mémoire partagée. À la fin de cette étape, on possède donc deux tableaux en mémoire partagée :
– L'un contenant les **valeurs** de la matrice sur le bloc diagonal (dans un format dense).
– L'autre contenant les **contributions** des valeurs en dehors du bloc diagonal.
Une fois que ces tableaux sont disponibles (synchronisation locale), on applique une réduction parallèle des contributions hors bloc diagonal, auxquelles on soustrait la valeur du vecteur solution. En d'autres termes, si t est le nombre de lignes traitées simultanément, on vient de calculer les contributions partielles $\boldsymbol{r}_j$ des lignes $[j, j+t]$ tel que :

$$\boldsymbol{r}_j = \boldsymbol{y}_j - \sum_{i=n}^{i \geq j+t} (\boldsymbol{x}_i * \mathbf{L}_{j,i}) \tag{5.29}$$

Pour compléter l'équation (5.28), il reste à accumuler les contributions du bloc diagonal. Pour cela, on utilise le tableau en mémoire partagée qui contient les valeurs de la diagonale dans une représentation dense. On peut alors y accéder dans n'importe quel ordre de lecture, avec n'importe quel thread. Ainsi, pour accumuler les dernières contributions, on choisit d'utiliser une parallélisation en colonne (ceci permet d'éviter les réductions parallèles). On utilise alors une boucle qui traite chaque colonne séquentiellement, et dans laquelle plusieurs threads mettent à jour le vecteur solution. Une optimisation consiste à utiliser les threads d'un même warp afin de ne plus avoir recours aux synchronisations locales. Pour cela, on utilise les t premiers threads, du bloc CUDA (ce qui est possible car le bloc diagonal est en mémoire partagée, et peut être accédé par tous les threads).

Évidemment, l'approche proposée présente une consommation mémoire plus grande que la solution précédente, ce qui implique qu'un nombre plus petit de blocs CUDA pourront être exécutés simultanément par les multiprocesseurs. Cependant, on s'attend à ce que chaque résolution soit plus rapide, puisqu'elle nécessite beaucoup moins de synchronisations. En effet, en comparant les performances on constate que cette version devient beaucoup plus rapide que le lower solve. Une solution pour adapter cette parallélisation à l'application

du lower solve consiste à calculer la transposée de la matrice triangulaire. On pourra ainsi accéder aux données en ligne et utiliser le même schéma de parallélisation (voir figure 5.16). De plus, ce calcul peut être complètement caché, en le déportant dans le thread de factorisation. En conséquence, nous obtiendrons le préconditionneur légèrement plus tard, mais nos expériences ont montré que la différence de temps est négligeable. Cette solution ajoute malgré tout un surcoût dans la boucle de simulation, pour transmettre les deux versions de la factorisation au GPU. Encore une fois, ce coût est modéré, car ce transfert n'est réalisé qu'une seule fois à chaque mise à jour du préconditionneur.

5.5.3 Avantages et extension de la méthode

Nous discutons maintenant des avantages, et des applications de cette contribution.

Gestion des objets détaillés

La première chose importante à remarquer, c'est que nous ne calculons jamais la matrice inverse explicitement, et nous gardons toujours une représentation creuse des factorisations. Ceci était un point limitant de la méthode de *compliance warping*, qui stockait l'inverse de la matrice dans une représentation dense. La consommation mémoire était alors problématique sur CPU, et le problème était amplifié sur GPU. Avec la représentation creuse, notre méthode permet de simuler, des contacts ou des contraintes sur des objets détaillés comprenant plusieurs milliers de degrés de liberté.

Couplage mécanique et non homogénéité

La prise en compte du couplage mécanique de façon quasi exacte entre les contacts est également une contribution importante. Cela nous permet d'améliorer la qualité de la réponse aux contacts, dans les simulations en temps réel. En effet, comme le préconditionneur représente (tout au long de la simulation) une bonne approximation de la mécanique des objets, nous montrerons qu'il permet d'obtenir une réponse aux contacts qui soit très similaire à un calcul basé sur le couplage exact. Or, nous avons vu que le calcul exact nécessite d'inverser une grande matrice à chaque pas de temps, ce qui est impossible à faire en temps réel. Ainsi, notre méthode permet d'améliorer la qualité de la réponse dans le contexte des simulations en temps réel.

D'autre part, on a vu dans le chapitre 3 que le préconditionneur fournit une bonne approximation de la mécanique des objets qu'ils soient homogènes ou hétérogènes. Ceci était une autre limitation de la méthode de *compliance warping* qui devait se contenter d'une approximation calculée dans l'état initial. Or, la déformation des objets modifie les propriétés mécaniques des objets, ce qui invalide le résultat pré-calculé. En particulier, les zones plus rigides d'un objet non homogène vont se déformer de façon moins importante que les zones plus molles ce qui va provoquer des variations plus ou moins importantes de rigidité sur différents éléments (ce qui ne peut pas être compensé, en appliquant les rotations). Notre méthode permet de prendre en compte ces aspects dans le calcul de la réponse aux contacts en temps réel (voir figure 5.18).

Application aux modèles non linéaires

L'application du préconditionneur représente alors une bonne approximation de l'inverse quelle que soit la façon dont le système matriciel a été calculé. Ainsi, la seule contrainte pour utiliser notre méthode est d'être capable de construire la matrice du système pour être en mesure de la factoriser. Or, nous avons supposé dans le chapitre 2 que quel que soit le modèle de déformation utilisé, on peut le linéariser entre deux pas de temps. En effet, de très faibles variations dans la rigidité devraient être introduite entre deux pas de temps, et on considère que la relation suit une loi linéaire.

Cependant, on s'attend à ce que les résultats soient optimaux pour le modèle co-rotationnel. En effet, nous avons montré dans la section 3.4 que le préconditionneur fournit la meilleure approximation pour ce modèle. Pour les modèles non linéaires en matériaux la variation de rigidité est généralement plus importante, ce qui implique de mettre à jour le préconditionneur plus rapidement pour garder une erreur faible. Cependant, cette méthode est la première à proposer une solution pour gérer une réponse aux contacts pour les modèles non linéaires, qui soit basée sur une formulation en LCP. En particulier, nous avons réussi à appliquer notre approche aux modèles de St-Venant-Kirchhoffs et Mooney Rivlin, (tous deux implémentés dans le modèle MJED Marchesseau et al. (2010) disponible dans SOFA).

Une limitation actuelle de l'expérimentation vient du fait que la résolution des contacts peut engendrer l'inversion de certains éléments. C'est le cas surtout quand les maillages sont détaillés et les éléments relativement petits. Dans ce cas, le matrice obtenue n'est plus garantie d'être symétrique définie positive. Et, même si l'utilisation du solveur **LDL**^{T}, nous permet d'obtenir malgré tout une factorisation (contrairement au solveur **LL**^{T}), l'algorithme du Gauss-Seidel ne parvient pas à converger, sur ce type de problème. Ceci ne remet pas en cause la méthode, mais nécessite de lui ajouter des solutions pour empêcher que les éléments s'inversent.

Changements topologiques

Pour terminer, nous souhaitons évoquer le fait que cette technique permet également de gérer les changements topologiques pour la réponse aux contacts. Ceci de façon automatique et sans sur-coût important, à condition d'être capable de mettre à jour la rigidité du matériau. Pour de nombreux modèles de déformations, il est assez facile de mettre à jour la rigidité du matériau puisqu'il suffit alors de recalculer les rigidités locales des éléments affectés. Or, comme le nombre d'éléments impactés par une découpe est généralement faible, la mise à jour de la rigidité du matériau en temps réel est possible dans la plupart des modèles déformables. Dans ce cas, l'utilisation du préconditionneur asynchrone pour résoudre les contacts va alors se baser sur un système matriciel qui tient compte des changements topologiques. Par ailleurs, comme on sait que le préconditionneur ne nécessite que quelques pas de temps (de l'ordre de 3 ou 4 dans les expérimentations que nous présenterons en section 6.1) pour calculer une nouvelle factorisation, les changements dans la compliance seront effectifs très rapidement. En effet, avec une simulation en temps réel (au moins 25 FPS), le délai de mise à jour sera très peu perceptible pour l'utilisateur (voir chapitre 6).

Par ailleurs, contrairement à la solution que nous avions proposée en section 5.4.3 pour mettre à jour l'inverse pré-calculée de façon incrémentale, cette méthode n'accumule pas d'erreurs numériques au cours des changements topologiques. En effet, le calcul du préconditionneur se base toujours sur un re-calcul complet à partir de l'état courant du système, il est donc complètement insensible à l'historique de la découpe. Ainsi, dans la section 6.1 nous montrons des exemples dans lesquels nous avons réussi à simuler un modèle co-rotationnel et à le découper avec une méthode de découpe de type carving (qui consiste simplement à supprimer des éléments, sans jamais ajouter ou supprimer de nœuds). Ceci, en résolvant les contacts avec des contraintes, qui tiennent compte du couplage mécanique et des modifications topologiques.

Cependant, la principale limitation de l'implémentation actuelle est que le nombre de nœuds doit rester constant après la modification topologique. l'ajout ou la suppression de nœuds aura pour effet de modifier la dimension du système mécanique. Or, comme le préconditionneur ne tient pas compte des modifications instantanément, les factorisations posséderont une dimension différente de celle du système entre les mises à jour. Il n'est alors plus possible d'utiliser la méthode (que ce soit comme préconditionneur, ou pour la résolution des contacts). Cependant, plusieurs pistes peuvent être envisagées pour poursuivre ces travaux. Tout d'abord, les temps de mise à jour étant relativement courts, on pourra alors se contenter d'une rigidité très approchée sur les points nouvellement créés. On peut par exemple considérer de calculer la compliance de façon découplée jusqu'à la prochaine mise à jour (ce qui se traduit par l'inverse d'une diagonale). Pour les modèles de grandes tailles où les mises à jour sont plus longues, une autre piste peut être d'étudier une façon de mettre à jour les factorisations avec une formule du type Sherman-Morrisson modifiée (comme nous l'avons présenté dans la section 5.4.3).

5.5.4 Résultats

Tous les résultats suivants, sont obtenus sur un Intel Quad-Core® Core™ i7 3.07 GHz et une Nvidia® GeForce® GTX 580, GTX 480 et GTX 295.

Évaluation Nous avons mesuré l'erreur introduite en utilisant différentes approximations de la matrice de compliance, par rapport à une factorisation exacte calculée à chaque pas de temps que nous utiliserons comme référence. Nous avons produit une simulation impliquant une lentille plus rigide au centre que sur la périphérie, qui est poussée par une sphère à travers une cavité au milieu d'un tore. Ainsi, nous avons mesuré la différence de position des sommets avec les différentes approximations (voir fig 5.17).

Lorsque nous utilisons uniquement la matrice diagonale (voir figure 5.18a), la sphère ne parvient pas à enfoncer la lentille dans la cavité. En effet, les forces qui sont appliquées par la sphère pour faire descendre la lentille sont situées au centre (sur la zone rigide), alors que les forces qu'appliquent le cylindre sont situées sur la périphérie et maintiennent l'objet vers le haut. La lentille reste alors en équilibre sans se déformer mécaniquement, puisque les contacts sont résolus de façon découplés sans tenir compte de la cohérence globale du système. En revanche, en diminuant le pas de temps on constate que la position de l'objet déformable tend vers la position de référence. En effet, comme chaque intégration tend à minimiser l'énergie de déformation, le fait de réaliser de nombreuses intégrations et de

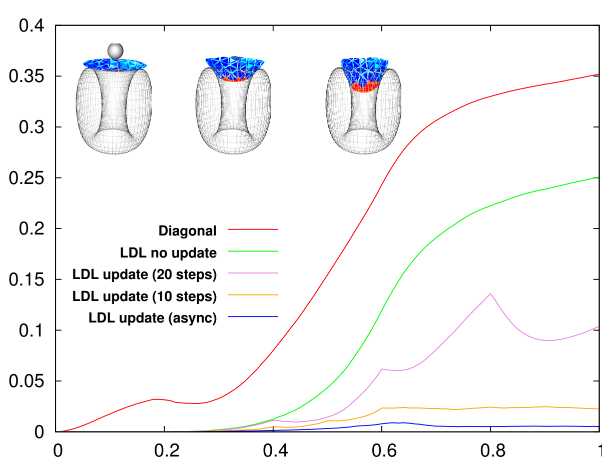


FIG. 5.17 – Différence de positions au fil du temps par rapport à une factorisation exacte, introduite par l'utilisation de différentes approximations.

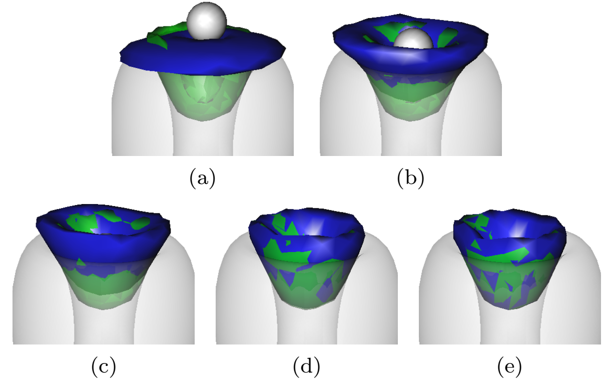


FIG. 5.18 – Position de la lentille (bleue) après 0.8 secondes de simulation, par rapport à la position exacte (vert). (a) version découplée. (b) compliance warping. (c) et (d) mise à jour tous les 20 et 10 pas de temps. (e) mise à jour asynchrone (calculée en temps réel).

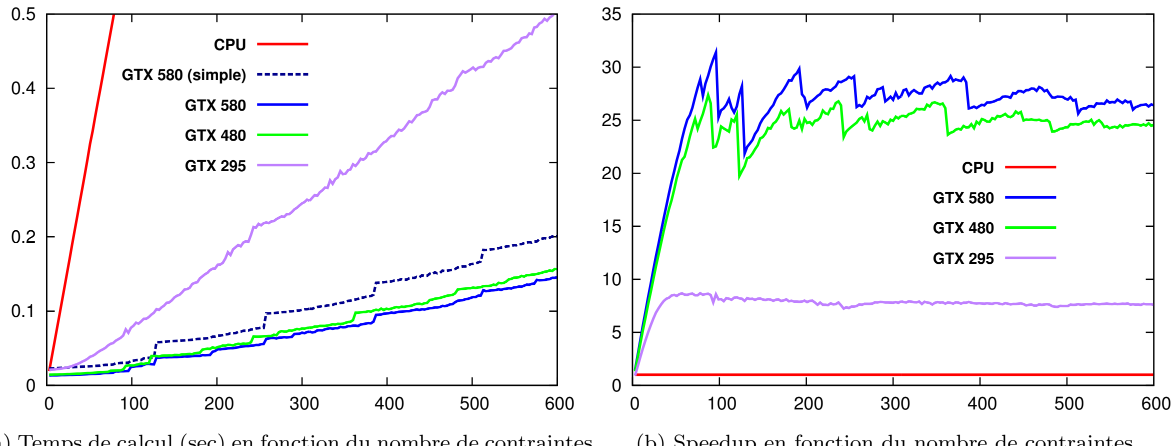
déplacer continuellement les points en contact va avoir tendance à déformer correctement le disque. Cependant, même avec un pas de temps de l'ordre de 10^{-5}, la position de la lentille reste relativement éloignée, et la simulation n'est plus du tout en temps réel.

En utilisant la méthode du compliance warping (voir figure 5.18b), on constate que le résultat est bien meilleur que la version non couplée, tout en restant en temps réel. Cependant, la rigidité est mal évaluée dans la configuration déformée, et très vite la sphère ne parvient plus à enfoncer la lentille.

Lorsque nous mettons à jour le préconditionneur tous les 20 et 10 pas de temps, (voir figures 5.18b et 5.18a), on constate que l'erreur est de plus en plus faible à mesure que la période est petite. Sur la courbe où le préconditionneur est mis à jour tous les 20 pas de temps, on peut facilement l'effet de chaque mise à jour puisqu'à chacune d'entre elles, la sphère parvient à enfoncer la lentille plus profondément et l'erreur diminue en conséquence. Pour la version où le préconditionneur est mis à jour tous les 10 pas de temps, on ne constate quasiment plus d'erreur. Enfin, en utilisant la version asynchrone, où la mise à jour est faite dès que possible en utilisant un thread séparé (voir figure 5.18b), on peut simuler un comportement quasiment exact. De plus la méthode reste en temps réel et compatible avec de grands pas de temps.

Implémentation GPU Nous avons mesuré les temps de calcul (voir figure 5.19) nécessaires pour construire la matrice de compliance avec notre version GPU, comparée à une version séquentielle.

Comme prévu, le temps de calcul pour obtenir une seule résolution avec notre algorithme GPU est à peu près deux fois supérieur à la version CPU. Ainsi, nous maintenons l'application du préconditionneur sur CPU pendant le mouvement libre (car chaque itération du CG ne nécessite qu'une seule résolution). Cependant, le temps de calcul pour résoudre les multiples contacts reste presque constant avec l'algorithme de GPU, ce qui permet de



(a) Temps de calcul (sec) en fonction du nombre de contraintes (b) Speedup en fonction du nombre de contraintes

FIG. 5.19 – Comparaison des temps de calcul pour la construction de l'opérateur de Delasus basée sur le préconditionneur **LDL**^{T} sur GPU. Les temps de calcul concernent la version optimisée ou plusieurs lignes sont traitées en parallèle pendant une résolution triangulaire, à l'exception de la courbe GTX 580 (simple) qui montre les temps de calcul de la solution où chaque ligne est traitée séquentiellement par plusieurs threads sur une architecture GTX 580.

profiter rapidement du processeur graphique. En effet, jusqu'à 100 contraintes, les unités de calcul du GPU ne sont pas pleinement exploitées, et chaque résolution est calculée en parallèle. Au-delà de ce nombre, certaines unités de calcul vont alors calculer plusieurs résolutions successivement, et la courbe des temps de calcul ressemble aux marches d'un escalier.

Toutefois, le GPU est capable de superposer les temps d'attente comme les synchronisations ou les lectures/écritures en mémoire, avec des calculs pour une autre solution. Ainsi, la résolution de 260 contraintes est de seulement 1.7 fois plus lente que pour 130 contraintes. Par ailleurs si on compare la version simple (où chaque ligne est traitée successivement) et optimisée sur la figure 5.19a, on voit que la version bloc offre de meilleurs résultats. Surtout, elle permet de diminuer le coût du kernel quand la simulation présente peu de contacts, mais en contrepartie, on constate des sauts dans les temps de calcul plus fréquents. En effet, comme la version bloc consomme plus de mémoire, chaque multiprocesseur peut en exécuter un plus petit nombre simultanément, en contrepartie chacun est plus rapide. Enfin, on constate que l'apparition des caches sur les architectures GTX 480 et GTX 580 fournissent une accélération significative par rapport à la version GTX 295.

5.6 Conclusion

Dans cette section, nous nous sommes intéressés à la création et à la résolution (basée sur des contraintes), des contacts dans une application temps réel. Nous avons une fois de plus fait des choix pour modéliser le plus précisément ces phénomènes, ce qui nous a amené à proposer des parallélisations de différents algorithmes basées sur des contraintes et une formulation en LCP. Nous avons tout d'abord proposé une parallélisation de l'algorithme du Gauss-Seidel pour GPU et processeur parallèle, qui permet de trouver la force à appliquer pour résoudre les contacts. Nous avons ensuite constaté que la résolution du

Gauss-Seidel n'était pas la seule étape consommatrice des temps pour la réponse aux contacts, mais que la construction du LCP était également problématique.

Ainsi, nous avons proposé une méthode pour calculer efficacement l'inverse d'une matrice bande tri-diagonale sur GPU. Nous avons ensuite montré comment paralléliser la construction de l'opérateur de Delasus directement sur GPU quand on possède l'inverse (ou une approximation) du système. Cette contribution est alors applicable aussi bien aux modèles de poutre déformable, qu'au modèle co-rotationnel, et nous avons montré que nous pouvions gérer un nombre important (de l'ordre de 2500) de contacts en temps réel. Ensuite, nous avons étendu la méthode de *compliance warping* à la gestion de la découpe et des changements topologiques.

Enfin, nous avons proposé un nouvel algorithme pour construire la matrice de compliance en ré-utilisant le calcul du préconditionneur présenté au chapitre précédent. Cette méthode n'est envisageable qu'avec l'utilisation d'un processeur parallèle comme un GPU, car une version séquentielle ne peut garantir des temps de calcul en temps réel que pour un nombre très limité de contraintes. En revanche, même si la version GPU peut gérer un nombre plus important de contacts en temps réel, il est tout de même important de limiter leur nombre. Ainsi, cette méthode est parfaitement bien adaptée à la méthode de détection de collisions basée sur les volumes d'interpénétration que nous avons décrit précédemment (voir chapitre 4). De plus, cette dernière contribution peut gérer un grand nombre de nœuds, et s'appliquer aux modèles non linéaires. Elle offre également une meilleure précision, et gère la découpe de façon automatique et sans dérive numérique. Cependant, elle est plus coûteuse et permet de gérer un nombre moins grand de contacts en temps réel. Ainsi, le dernier chapitre est consacré à la détection des contacts, et nous allons présenter un ensemble de contributions pour limiter leur nombre.

Chapitre

6

Applications et Conclusion

Table des matières

6.1 Applications **171**

6.1.1 Simulation du cathéter 171

6.1.2 Simulation de la chirurgie de la cataracte 172

Simulation de la procédure MSICS 174

Simulation de la Phacoemulsification 175

Rendu de la simulation 176

6.1.3 Simulateur de résection hépatique en laparoscopie 176

Construction de la simulation 177

Mise à jour incrémentale de la compliance 179

Calcul de la compliance basé sur le préconditionneur 180

6.2 Bilan **181**

6.3 Perspectives **183**

6.1 Applications

Pour conclure ce document, nous commençons par présenter un ensemble d'applications qui tirent parti des travaux présentés dans cette thèse. Nous montrerons tout d'abord une application de radiologie interventionnelle pour l'embolisation d'un anévrisme par un coil. Ensuite nous présenterons une simulation de la chirurgie de la cataracte, et nous terminerons en présentant un simulateur de résection hépatique en laparoscopie. L'ensemble de ces simulations sont calculées en temps réel. À chaque fois, nous montrerons comment nos contributions ont permis d'améliorer les performances (et la précision) des simulations.

6.1.1 Simulation du cathéter

Un anévrisme cérébral se produit lorsqu'une zone fragilisée d'une artère se déforme de façon anormale (suite à un traumatisme par exemple). Il se manifeste sous la forme de petites poches dans laquelle le sang s'accumule. Sous l'effet de la pression exercée par le flux sanguin, le risque est que l'anévrisme se dilate et finisse par se rompre. Dans ce cas, les dégâts causés par l'hémorragie interne peuvent être dramatique pour le patient. Pour prévenir la rupture d'un anévrisme, il est possible de pratiquer une chirurgie ouverte (avec tous les risques qu'elle comporte). Cette chirurgie consiste à poser un clip métallique autour du collet (structure reliant le vaisseau à l'anévrisme) pour interrompre la vascularisation. Le traitement endovasculaire, représente aujourd'hui une alternative à la voie chirurgicale. Cette procédure est dite non invasive, elle consiste à déployer un fil de platine (appelé coil) dans l'anévrisme afin de l'oblitérer complètement.

Un simulateur endovasculaire a été développé dans SOFA Dequidt et al. (2009), pour la formation médicale et la planification des interventions. Un cathéter peut être manipulé pour naviguer à l'intérieur des artères, puis être déployé à l'intérieur d'un anévrisme (voir figure 6.1). La version la plus récente de la simulation permet d'utiliser un périphérique haptique pour contrôler le cathéter virtuel. Un espace de démonstration à Euratechnologie[1] est même dédié à ce simulateur, qui met en scène la simulation avec un mannequin virtuel, et plusieurs écrans de contrôle.

Le coil, est modélisé par des centaines de poutres en série. Étant constitué uniquement de segments, la détection de collisions est assez rapide à réaliser, avec une détection par proximité. Cependant, elle peut produire de multiples contacts tout le long de la structure. De plus, comme l'objet déformable glisse le long des parois de l'anévrisme, nous avons besoin d'utiliser la formulation étendue du LCP (voir chapitre 5) pour inclure le frottement de Coulomb. Ceci crée trois contraintes par contact, ce qui demande de résoudre un nombre important de contraintes en temps réel.

Le nombre de contraintes générées pendant la simulation présentée en figure 6.1 est compris entre 600 et 3000 contraintes. Pour les résoudre, nous utilisons l'algorithme de Gauss-Seidel parallèle que nous avons présenté en section 5.3. Dans cette gamme, la parallélisation sur processeur multi-cœur est actuellement la plus efficace, et nous obtenons une accélération entre 5× et 20× pour cette étape, conduisant à une diminution significative des temps de calcul. Le GPU est capable d'atteindre des accélérations jusqu'à 3×, mais surtout il devrait nous permettre d'augmenter la complexité de nos simulations.

[1] http://www.euratechnologies.com/

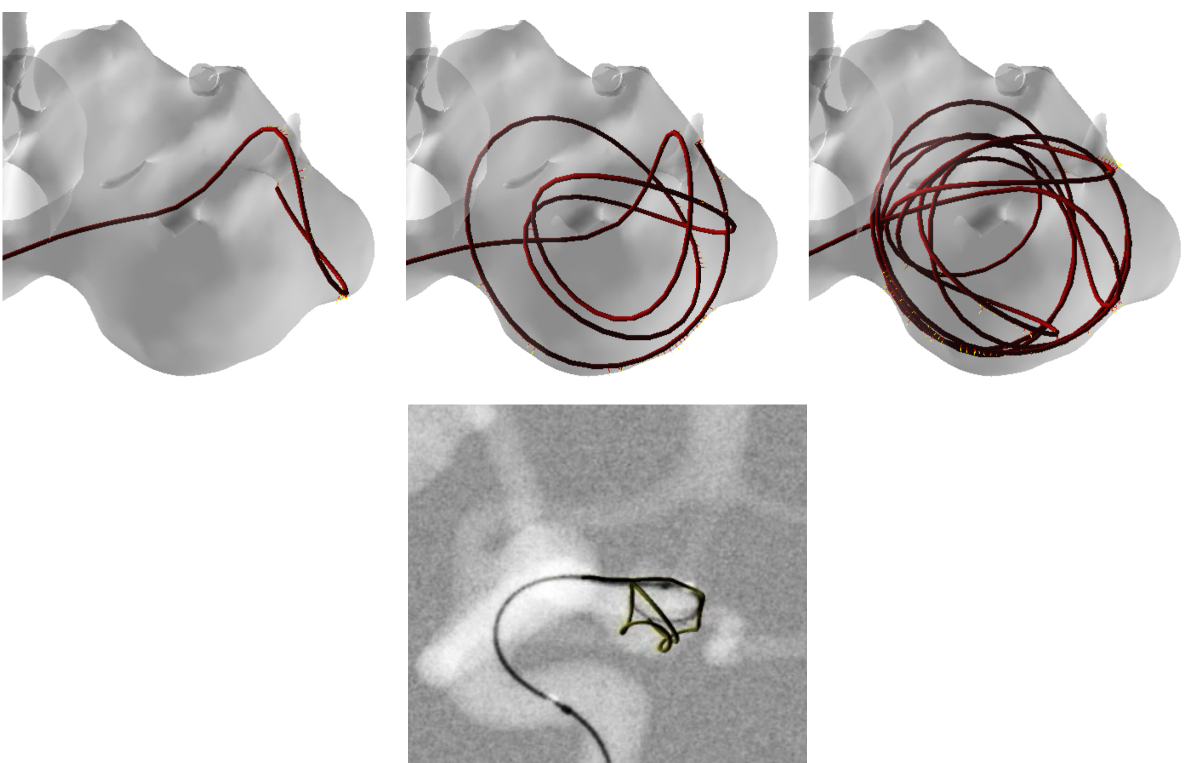


Fig. 6.1 – Séquence de 3 étapes de la simulation du déploiement du coil à l'intérieur d'un anévrisme (haut), et vue de contrôle en réalité augmentée.

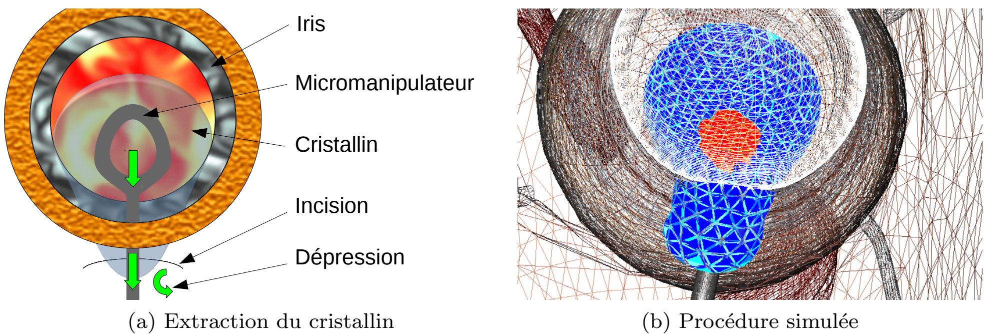
D'autre part, avec un tel nombre de contacts la construction de la matrice de compliance peut rapidement devenir l'étape la plus consommatrice de temps de calcul. Ainsi, nous utilisons la version parallèle de la construction de la matrice de compliance que nous avons détaillé dans la section 5.4. Cette contribution nous permet de maintenir des performances en temps réel. De plus, et en inversant la matrice BTD directement sur GPU nous pouvons encore améliorer les temps de calcul. Cela permet également de traiter l'intégralité de la réponse aux contacts sur GPU, et donc d'économiser plusieurs transferts entre les architectures. Á l'avenir, ces aspects seront de plus en plus importants à mesure que nous allons augmenter le détail des simulations.

6.1.2 Simulation de la chirurgie de la cataracte

La cataracte touche aujourd'hui près de 20 millions de personnes dans le monde, dont la majorité est située dans les pays en voie de développement. La pathologie se traduit par une opacification du cristallin, qui empêche le passage de la lumière. Les symptômes sont une perte partielle ou totale de la vue, et c'est la première cause de cécité dans le monde. Il existe un traitement d'ordre chirurgical, qui consiste à extraire le cristallin opacifié pour le remplacer par un implant intraoculaire. L'intervention est alors réalisée en milieu chirurgical sous microscope.

La phacoemulsification, est la procédure standard pour traiter cette pathologie. Cette chirurgie repose sur l'utilisation d'une sonde (appelée du phacoemulsificateur) introduite dans l'œil par une micro incision (2 à 3 millimètres). La sonde émet des ultrasons qui

émulsifient le cristallin en petits morceaux qui sont aspirés en continu par le phacoemulsificateur. L'avantage de cette technique est que l'incision reste de très petite taille, ce qui permet une guérison rapide des patients. En revanche, elle impose d'utiliser un matériel technologiquement avancé et onéreux. Selon une étude[2] récente, le coût moyen pour traiter un seul œil avec cette technique est estimé à environ $2000. Ce coût peut représenter plusieurs fois le salaire annuel des patients dans les pays sous développés. La conséquence est que ces patients ne sont tout simplement pas traités.



(a) Extraction du cristallin (b) Procédure simulée

Fig. 6.2 – Extraction du cristallin, à l'aide d'un micromanipulateur. 6.2a mouvement de l'outil, 6.2b cristallin simulé. Les éléments en rouges sont plus rigides que les éléments en bleus.

Une autre intervention chirurgicale connue sous le nom *Manual Small Incision Cataract Surgery* (MSICS) requiert une technologie mois avancée, et offre des résultats presque équivalents lorsqu'elle est pratiquée par un spécialiste expérimenté. Cette technique nécessite une incision légèrement plus large (environs 5 mm), et elle consiste à extraire le cristallin en une seul pièce. Pour cela, le chirurgien utilise un micromanipulateur (appelé *Irrigating Vectis cannula*), qu'il place sous le cristallin opacifié. En s'aidant de la pression naturelle à l'intérieur de l'œil et en déformant les structures anatomiques, le chirurgien peut alors extraire le cristallin sans le découper (voir figure 6.2). Le coût cette procédure est nettement inférieur à celui de la phacoemulsification, puisqu'il est estimé entre $35 à $50 par œil. Malheureusement, il y a très peu de spécialistes de cette technique dans le monde. Un rapport récent de HelpMeSee (2009) montre qu'il y a un grand besoin de formation pour cette chirurgie, afin de faire face au nombre grandissant de personnes touchées par la cataracte.

Les simulateurs peuvent représenter une réponse à ce problème, en proposant un environnement sûr et reproductible, pour entraîner et former les chirurgiens. La simulation de la cataracte a déjà été présentée dans des applications en temps réel. Une solution commerciale est proposée par VRmagic Wagner et al. (2002) pour la phacoemulsification, et certaines étapes de la chirurgie ont également été étudiées en détail comme l'insertion de l'implant Comas et al. (2010). Cependant, ce travail est motivé par un simulateur d'entraînement pour la technique MSICS, dont les besoins ne peuvent être abordés avec les méthodes existantes.

Cette simulation comporte plusieurs enjeux. La première est une contrainte de temps réel, puisqu'on souhaite pouvoir interagir avec le simulateur. D'autre part, le globe oculaire et

[2] www.allaboutvision.com

le cristallin (voir figure 6.2) doivent être tous les deux simulés, et subissent de grandes déformations ainsi que de multiples contacts. Il faut également prendre en compte les hétérogénéité des structures anatomiques car elles influent sur la réussite ou non de la chirurgie. En effet, le noyau du cristallin est généralement plus rigide que la périphérie, et ce phénomène peut nécessiter d'adapter la taille de l'incision. De même, la taille des éléments est potentiellement très différente (en particulier près de l'incision située sur le globe oculaire). Toutes ces caractéristiques conduisent à un conditionnement très faible, de la matrice du système.

Dans ce projet, nous nous sommes uniquement intéressés à l'extraction du cristallin. Ainsi, nous avons réalisé l'incision de l'œil en dehors de la simulation. Les maillages de surface ont été réalisés par un graphiste de l'équipe qui a travaillé en collaboration avec des médecins, pour respecter les spécificités anatomiques de l'œil. À partir de la surface, nous en avons extrait un maillage volumique de chacune des structures anatomiques, en utilisant la librairie (CGAL). La difficulté de cette étape était de trouver un jeu de paramètres qui permettait de respecter l'incision très fine, sans pour autant obtenir un nombre trop grand d'éléments. Enfin, l'instrument chirurgical est contrôlé par l'utilisateur avec un dispositif haptique. Comme nous calculons la matrice de compliance des objets en contact avec l'instrument, nous sommes capables de calculer et transmettre des forces à l'appareil.

Simulation de la procédure MSICS

Nous montrons dans la figure 6.3 notre simulation de la cataracte. Dans cette simulation, le cristallin est modélisé avec 1113 nœuds et 4862 tétraèdres, tandis que l'œil contient 1249 nœuds et 3734 tétraèdres. Le centre du cristallin est cinq fois plus rigide que la périphérie, et nous avons utilisé notre préconditionneur asynchrone (aussi bien pour le globe oculaire que pour le cristallin) afin d'assurer la convergence du CG. En effet, le gradient conjugué préconditionné ne nécessitait en moyenne que de $11,6$ itérations pour converger, malgré les fortes déformations et les hétérogénéité.

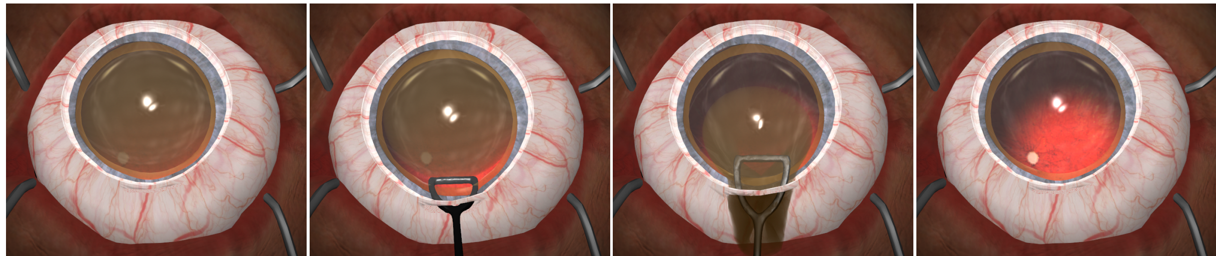


Fig. 6.3 – Simulation de l'extraction du cristallin avec la technique MSICS.

Le cristallin est retiré avec l'aide des déformations du globe oculaire, et des frottements avec l'instrument. Grâce à la formulation en volume et les LDI nous avons pu maintenir un nombre très petit de contraintes pour une simulation aussi complexe En effet, sur l'intégralité de la simulation $47,34$ contraintes frottantes étaient générées pas pas de temps (soit un peu plus de 15 contacts). En fonction de ce nombre, les performances de la simulation étaient comprises entre 18 et 25 FPS. Au sein d'un seul pas de temps,

la répartition du temps de calcul était la suivante : 40, 56% pour le gradient conjugué préconditionné, 44, 69% pour l'assemblage de la matrice de compliance sur GPU, et 6, 58% pour la détection de collisions et la formulation de la réponse.

Dans cette simulation, l'ensemble de nos contributions porte sur plus de 90% d'un pas de temps. Or, sans utiliser l'ensemble de ces contributions cette simulation n'aurait pas pu être exécutée interactivement. Par exemple, sans utiliser le préconditionneur asynchrone et à précision égale, le CG nécessite plus de 2000 itérations pour simuler le globe oculaire et 900 itérations pour l'oeil. D'autre part, la détection des collision basée sur la rastérisation est réalisée en un temps très court, et surtout permet de maintenir un nombre de contraintes très bas. Ceci nous permet d'utiliser le préconditionneuer asynchrone pour résoudre les contacts en tenant compte du couplage mécanique et des hétérogénéité.

Simulation de la Phacoemulsification

Pour valider l'utilisation du préconditionneur pour la réponse aux contacts avec changements topologiques, nous avons également créé une scène similaire à la technique de phacoemulsification (voir figure 6.4). Dans cette scène, nous découpons le cristalin avec la méthode de carving (qui consiste à supprimer des éléments sans ajouter ni supprimer de nœuds). Pour obtenir un résultat visuel plus convaincant, nous subdivisons chaque tétraèdre en 8 éléments plus petits. Cependant, cette subdivision est purement visuelle, et n'est utilisée que pour la visualisation et la découpe (les points sont reliés au modèle mécanique à l'aide d'un mapping fourni dans SOFA).

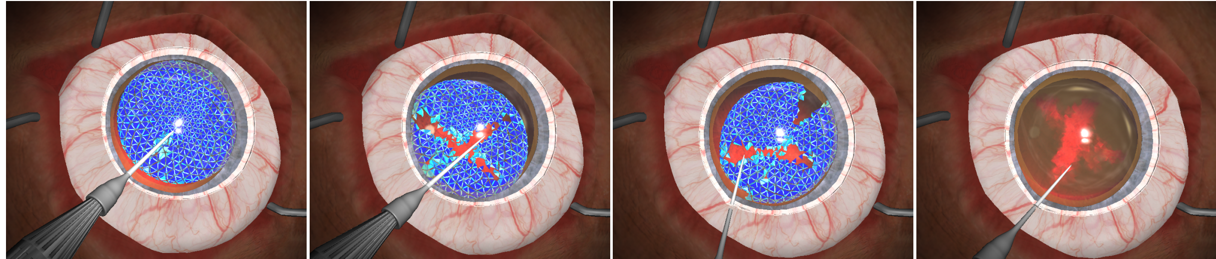


Fig. 6.4 – Simulation of the extraction of the eye lens during MSICS

Cette application montre que le préconditionneur peut prendre en compte les changements topologiques appliqués pendant la découpe. En effet, les modifications de rigidité sont directement prises en compte avec les mises à jour de la factorisation. Or, la fréquence de mise à jour du préconditionneur était de 4 pas de temps en moyenne pour l'œil, et de 3 pas de temps pour le cristallin, ce qui restait beaucoup trop faible pour être perceptible par l'utilisateur. Enfin, dans la mesure ou nous utilisons exactement le même jeu de données que pour la version MSICS, les performances sont similaires à ce que nous avons présenté auparavant. Cependant, il est intéressant de remarquer que les changements topologiques influencent très peu les temps de calcul, puisque la découpe utilisée ne nécessite aucune étape de remaillage. Notre solution est la seule méthode qui permette de simuler en temps réel des objets très détaillés avec une réponse par contrainte (qui tient compte du couplage mécanique entre les contacts) et des changements topologiques.

Rendu de la simulation

Pour terminer, nous avons ajouté un rendu en dehors de la simulation pour produire les images visibles en figure 6.5. Ce rendu n'est pour l'instant pas simulé en temps réel, mais permet d'avoir un aperçu de l'application finale. Il inclut notamment une gestion précise de la lumière et de la transparence.

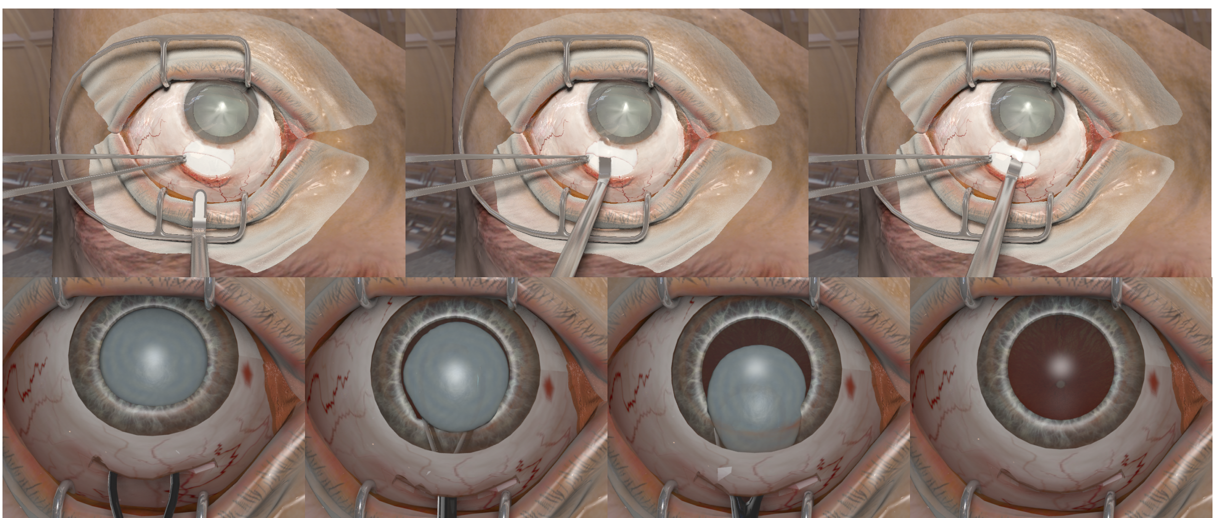


Fig. 6.5 – Simulation of the extraction of the eye lens during MSICS

Pour générer ces images, nous avons exporté la position des maillages à chaque pas de temps, afin de produire une séquence d'animation $3D$. Le rendu final à été réalisé dans modo 3d[3], qui est un logiciel spécialisé. Réaliser ce rendu en temps réel fait partie des travaux futurs pour cette application.

6.1.3 Simulateur de résection hépatique en laparoscopie

Une hépatectomie (ou résection hépatique) peut être envisagée pour le traitement de tumeurs bénignes ou malignes très localisées. Ce geste chirurgical consiste à retirer la tumeur par l'ablation de la partie contaminée. Un ensemble de bilans pré-opératoires très détaillés est nécessaire pour valider ou contre-indiquer le geste chirurgical. En effet, une hépatectomie reste une opération difficile et délicate, qui comporte de nombreux risques. Par exemple, le foie est un organe très vascularisé et le risque de couper un vaisseau et de provoquer une hémorragie est très important. Ces contraintes font qu'aujourd'hui, l'expertise du médecin est déterminante pour valider le recours à une hépatectomie. La simulation permettrait d'avoir une représentation plus objective, en fournissant une meilleure représentation aux chirurgiens (que ce soit avant l'opération en proposant un simulateur d'entraînement ou pendant l'opération avec un système de réalité augmentée).

Un nombre important d'hépatectomies est aujourd'hui réalisé en laparoscopie, ce qui signifie que les instruments médicaux sont insérés grâce à un petite incision au niveau de l'abdomen. Ces nouvelles techniques permettent de minimiser sensiblement les risques liés

[3] http://www.luxology.com/

à la chirurgie ouverte traditionnelle, et le temps de récupération des patients est également diminué. En contrepartie, elles nécessitent auprès du personnel médical d'acquérir de nouvelles compétences, comme par exemple le guidage par caméra ou comme le guidage tactile. En effet, la sensation tactile ressentie par le chirurgien est alors réduite aux sensations transmises via les instruments chirurgicaux. Cette interface qui s'est glissée entre le médecin et le patient, nous permet de reproduire un champ opératoire similaire dans nos simulations. On peut par exemple reproduire des forces similaires avec un périphérique haptique.

Toutefois, au même titre que les applications précédentes ces simulations présentent toujours de nombreux défis à surmonter. De multiples structures anatomiques interagissent ensemble, et peuvent potentiellement subir de grandes déformations. Une autre difficulté importante porte sur la découpe progressive du foie tout au long de la simulation. On peut trouver d'autres simulateurs semblables qui ont été développés dans le passé Bourquain et al. (2002); Lamadé et al. (2002). Cependant, l'application présentée ici s'appuie sur les méthodes avancées que nous avons présentées dans cette thèse.

Construction de la simulation

Cette démonstration est le résultat d'une collaboration entre l'INRIA et l'IRCAD[4] dans le cadre du projet PASSPORT[5]. En effet, la première étape pour construire cette simulation a été de générer la surface des structures anatomiques. Cette partie du travail a été réalisée par l'IRCAD, en utilisant des données réelles d'un patient. L'équipe a développé un moyen automatique pour segmenter les images issues d'un CT-scan. Ce procédé consiste à reconstruire le maillage $3D$ de la surface des structures anatomiques, à partir d'un ensemble d'images médicales (voir figure 6.6). L'automatisation de ce processus permet notamment de produire facilement des simulations spécifiques aux patients, ce qui est ce qui est particulièrement utile pour l'entraînement pré-opératoire des cas pathologiques rares ou complexes.

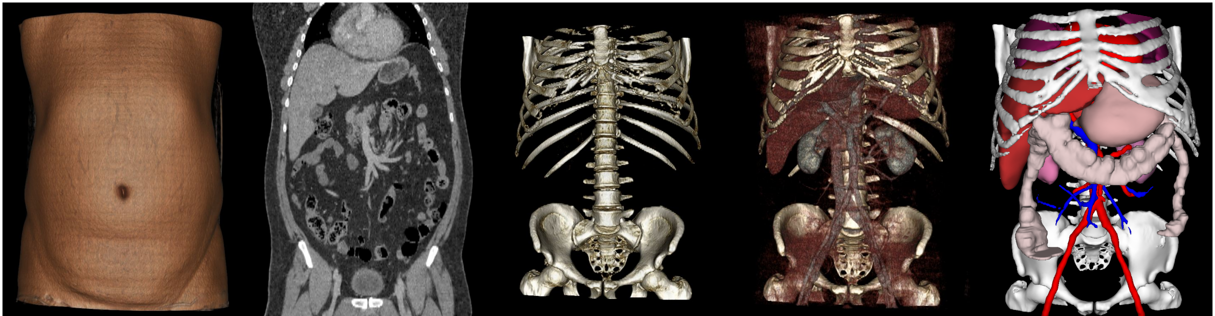


Fig. 6.6 – Segmentation d'un CT scan. Source IRCAD

À partir de la surface des organes, nous avons ensuite généré les différentes représentations de la simulation (voir figure 6.7). Nous utilisons trois représentations : une représentation visuelle, une mécanique et un modèle de collisions. Pour la partie visuelle, nous utilisons directement les maillages segmentés, ainsi que des composants fournis par SOFA pour

[4] www.ircad.fr/softwares/3Dircadb/3Dircadb1
[5] http://passport-liver.eu/

appliquer des textures sur les organes. Pour la représentation mécanique, nous avons utilisé le logiciel (CGAL) pour mailler le foie, l'estomac et le colon, car ils sont susceptibles de subir les plus grandes déformations. Pour les intestins et le diaphragme, nous avons utilisé une grille régulière de tétraèdres (affichée en fil de fer), car leur déformations sont moins importantes. Enfin, nous utilisons un modèle de collisions simplifié car les maillages obtenus par la segmentation ne constituent pas un volume fermé (ce qui est une contrainte de notre méthode à base de LDI). Ces maillages ont été générés manuellement, en décimant puis en corrigeant les erreurs issues de la segmentation.

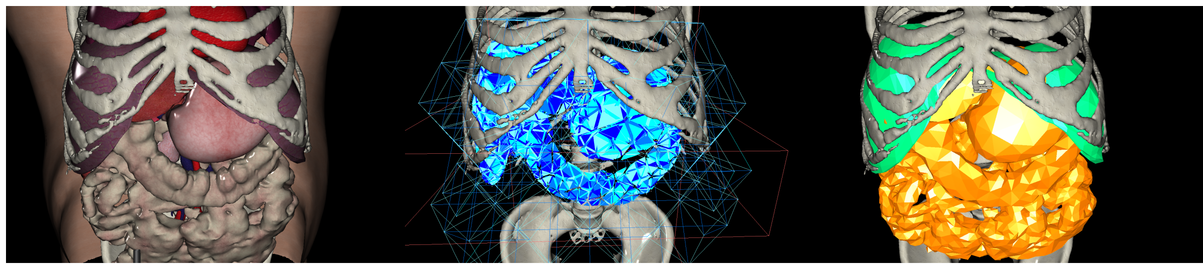


Fig. 6.7 – Différentes représentations des structures anatomiques utilisées dans la simulation (De gauche à droite : modèles visuels, modèles mécaniques, modèles de collisions).

Pour interagir avec la simulation, nous utilisons un ensemble de dispositifs haptiques (voir figure 6.8). La pince laparoscopique est contrôlée par un Xitact IHP[6], alors que la caméra laparoscopique est contrôlée par un PHANTOM OMNI[7]. Ces deux périphériques permettent de retranscrire mécaniquement une force calculée dans la simulation.

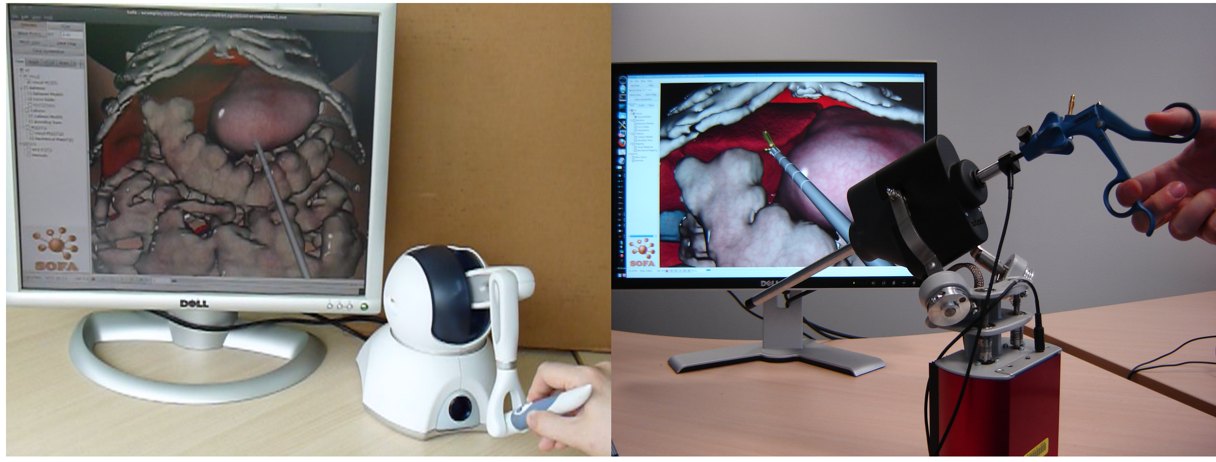


Fig. 6.8 – Différents périphériques haptiques pour la simulation médicale.

En tenant compte du couplage mécanique entre les contacts, on peut ainsi produire un retour haptique convaincant. Par exemple, l'utilisateur peut ressentir la rigidité des côtes en appliquant un contact uniquement sur le foie. En effet, la matrice de compliance permet de coupler les contacts indirects entre le foie, les côtes et l'instrument. On peut alors transmettre la force résultante au périphérique haptique.

[6] www.mentice.com
[7] www.sensable.com

Mise à jour incrémentale de la compliance

Nous présentons tout d'abord une première version de la simulation (voir figure 6.9) dans laquelle nous ne simulons que 3 organes déformables (le foie, l'estomac et le colon). Le foie étant l'un des organes les plus grands du corps humain, et il difficile d'utiliser un maillage uniforme composé de petits éléments sur l'intégralité du volume. En revanche, cette structure anatomique est décomposée en 8 régions théoriques (appelés segments anatomiques) qui permettent d'en avoir une représentation schématique. Ainsi, grâce aux bilans pré-opératoire, et en fonction de la localisation de la tumeur on peut déterminer à l'avance les segments anatomiques qui vont être manipulés pendant l'opération.

Dans cette simulation nous avons décomposé le foie en deux parties, dont une seule peut être découpée. La première est limitée à la zone proche de la tumeur et contient 4664 tétraèdres (1143 points), alors que la seconde est composée de 11251 tétraèdres (2382 points) répartis dans un volume beaucoup plus grand. Ainsi, seule la zone proche de la tumeur peut être découpée, et pour cela nous utilisons la méthode de carving. Pour relier les deux parties du foie, nous avons placé 22 contraintes afin d'imposer une position commune aux points se trouvant à l'interface entre les deux sections.

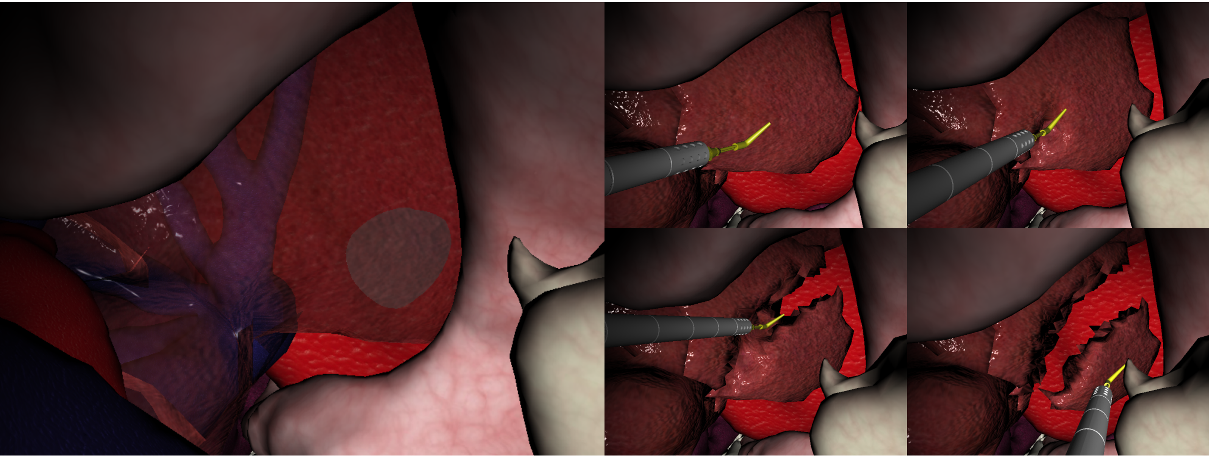


Fig. 6.9 – Simulation d'une résection hépatique avec la mise à jour incrémentale de la matrice de compliance. À gauche, on peut voir la tumeur en transparence.

Les organes sont simulés en utilisant le modèle co-rotationnel implicite, avec le CG non préconditionné calculé sur GPU (voir chapitre 3). Les collisions sont détectées avec la méthode des LDI présentée dans le chapitre 4, et les contraintes sont formulées en volume avec frottement. Les contraintes et les contacts frottants sont ajoutés dans le LCP, qui est obtenu avec la version GPU de la méthode de *compliance warping* (voir section 5.4.2). Quand une découpe est réalisée, nous utilisons la méthode de mise à jour incrémentale de la compliance calculée sur GPU.

Cette solution permet de découper le modèle mécanique tout en gardant la matrice de compliance cohérente par rapport aux modifications. En pratique, l'instrument ne supprime qu'un nombre limité d'éléments par pas de temps, ce qui permet de garantir des performances proches du temps réel (autour de 15 FPS), même pendant la découpe. Par

ailleurs, la méthode permet de gérer un nombre important de contacts simultanés grâce à la parallélisation du calcul de la compliance sur GPU. En revanche, une limitation de cette simulation est que chaque découpe accumule des erreurs numériques dans la matrice de compliance. Ainsi, la simulation d'une découpe plus longue s'effectuant sur plusieurs segments anatomique rendrait la simulation instable.

Calcul de la compliance basé sur le préconditionneur

Nous présentons maintenant une dernière application, dans laquelle nous construisons la compliance à partir du préconditionneur (voir section 5.5). Cette démonstration a été présentée à Siggraph Courtecuisse et al. (2011b).

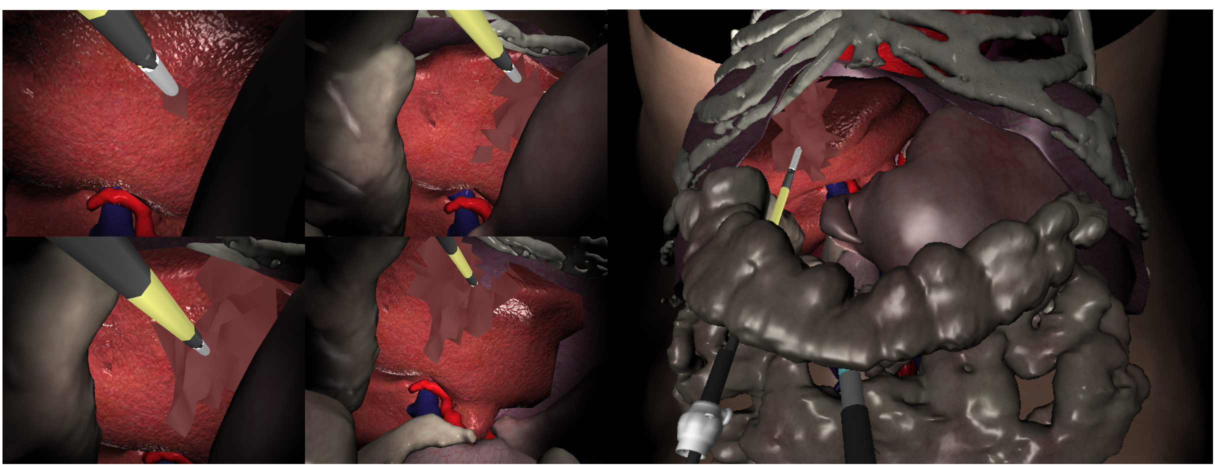


Fig. 6.10 – Simulation d'une résection hépatique, en construisant la matrice de compliance à partir du préconditionneur asynchrone.

Dans cette simulation, 5 objets déformables (le foie, l'estomac, le colon, les intestins et le diaphragme) sont simulés (Le détail sur structures anatomiques est donné dans le tableau 6.1). Pour ajouter du réalisme à la simulation, nous avons modélisé le diaphragme, auquel nous appliquons des forces périodiquement pour simuler la respiration. L'utilisateur peut contrôler alternativement une caméra, et un bistouri électrique capable de découper le foie.

	Nombre de points	Nombre d'éléments	Nombre de triangles
Foie	498	1999	2568
Estomac	306	756	594
Colon	347	819	702
Diaphragme	131	328	8794
intestins	64	142	12460

Tab. 6.1 – Informations sur les objets simulés.

Les structures anatomiques sont simulées avec le modèle co-rotationnel et une intégration implicite calculée sur GPU. Les contacts sont détectés avec la méthode des LDI, et des contraintes en volume avec frottement sont formulées. Cette solution est très pratique

pour animer cette simulation, puisque les maillages issus de la segmentations automatique présentent de multiples interpénétrations à l'état initial. De plus, elle permet de garder un nombre de contacts extrêmement bas (en moyenne 49.05 contraintes) malgré la complexité géométrique de la scène. Ainsi, nous utilisations le préconditionneur asynchrone (voir section 5.5) pour calculer la réponse aux contacts sur le foie. Pour les autres organes nous utilisons la méthode de *compliance warping* sur GPU (voir section 5.5.2), car ils ne présentent pas de changement topologiques.

Du point de vue des performances, la simulation déroule à une fréquence d'environ 25 FPS, y compris pendant la découpe. La répartition des temps de calcul dans une pas de temps est la suivante : 27.77% pour le mouvement libre. 11.01% pour la détection des collisions. 19.07% calculer la matrice de compliance, et 3.41% pour la résolution des contraintes. Dans cette application, le gradient conjugué préconditionné avait en moyenne besoin de $5,82$ itérations pour converger sur le modèle du foie. Pendant la découpe, le nombre d'itérations était légèrement plus élevé, mais se stabilise très rapidement dès qu'on arrête de découper.

Pour terminer, nous avons également réussi à simuler le foie avec le modèle non linéaire de St-Venant-Kirchhoff. La simulation était stable dans la mesure ou le foie était composé de grands éléments difficiles à inverser. De plus, nous étions capable de transmettre un retour haptique sur ce modèle qui est cohérent avec le modèle hyperélastique. Dans les perspectives à court terme, nous souhaitons d'ailleurs proposer un framework permettant de faire du rendu haptique temps-réel sur des modèles hyperélastiques, ce qui n'a encore jamais été réalisé. Encore une fois, cette contribution représente une avancée majeure, car il n'existe pas d'autre méthode pour simuler un rendu haptique convaincant sur des modèles non linéaires.

6.2 Bilan

Pour faire le bilan de cette thèse, on peut commencer par rappeler que le domaine de la médecine a récemment émis de nouveaux besoins envers les simulations physiquement réalistes d'objets déformables. Ces applications pourront contribuer à améliorer la qualité des soins dans un futur proche, puisqu'elles permettent de reproduire facilement des pathologies rares et complexes. En interagissant avec le simulateur, les médecins ont désormais accès à de nouveaux outils pour l'apprentissage ou la planification des actes chirurgicaux, le tout dans un environnement sûr et reproductible. Pour parvenir à construire de telles applications et obtenir un degré de réalisme suffisant, nous avons eu recours aux nouvelles architectures parallèles qui permettent d'atteindre des performances de calculs nettement supérieures à celles des processeurs traditionnels. En revanche, elles nécessitent de repenser totalement les algorithmes pour en produire une version hautement parallèle.

Concrètement, nos contributions ont d'abord porté sur la résolution de systèmes linéaires, afin de calculer la déformation des organes. Pour améliorer les temps de calcul, nous avons proposé une parallélisation sur GPU de l'intégrateur implicite, qui comprend notamment l'algorithme du Gradient Conjugué. Nous avons également proposé une technique de préconditionnement asynchrone qui permet d'améliorer la convergence du CG, tout en maintenant de bonnes performances.

Ensuite, nous avons proposé un algorithme de détection des collisions basé sur une rastérisation des objets. Nous en déduisons les volumes d'intersections ainsi que les gradients de volumes à partir des LDI. Nous avons ensuite montré comment exploiter ces informations pour construire une réponse en contraintes volumiques avec frottement, et nous avons proposé un modèle multi-volumes qui permet d'améliorer le comportement des objets en réponse aux collisions.

Enfin, le dernier groupe de contributions traite du problème de la réponse aux contacts. Nous avons tout d'abord proposé une méthode de parallélisation de l'algorithme du Gauss-Seidel, qui permet de calculer la force de réponse aux contacts. Ensuite, nous avons proposé plusieurs contributions pour calculer la matrice de compliance efficacement en tirant partie des spécificités du modèle de déformation. Pour terminer, nous avons proposé une méthode générique qui repose sur l'utilisation du préconditionneur asynchrone pour construire une approximation de la matrice de compliance quel que soit le modèle de déformation.

Nous avons également présenté un ensemble d'applications qui montre que nos contributions sont capables de répondre aux besoins des applications médicales. Nous garantissons une grande stabilité et des temps de calcul très bas, ce qui permet aux utilisateurs d'interagir avec les simulations. Les contributions présentées dans ce document ont des implications à différents niveaux :

Diminuer les temps de calcul En identifiant et en ré-organisant les dépendances des algorithmes utilisés dans les simulations physiques, nous avons proposé des solutions pour extraire un haut degré de parallélisme. L'accélération obtenue nous a permis de diminuer sensiblement les temps de calcul et d'atteindre des performances en temps réel. En règle générale, nous avons utilisé cette accélération non pas pour simuler les objets plus vite, mais plutôt pour augmenter le détail des modèles. Ainsi, nos solutions ont permis de changer l'ordre de grandeur du nombre d'éléments simulés en temps réel.

Meilleure précision en temps réel Les nouvelles architectures parallèles, nous ont permis de proposer des nouvelles solutions dans le contexte des applications en temps réel. Ces algorithmes n'étaient pourtant pas nouveaux, mais ils étaient auparavant limités par leurs temps de calculs. En proposant une version parallèle et optimisée, nous avons rendu possible l'utilisation des ces méthodes dans un contexte interactif (comme le calcul du contact avec couplage mécanique). En utilisant ces méthodes dans nos applications, nous avons pu augmenter la précision et la généricité dans les simulations s'exécutant en temps réel. Nous avons également rendu possible la simulation de phénomènes complexes, comme par exemple les changements topologiques nécessaires dans de nombreuses simulations médicales.

Nouveaux algorithmes Enfin, les propriétés radicalement différentes des architectures parallèles nous ont conduites à proposer de nouveaux algorithmes (LDI et contacts volumiques). Ces nouvelles solutions ont permis de repousser les limites de la simulation en temps réel, en diminuant sensiblement les temps de calcul des approches traditionnelles. Ainsi, ces nouveaux algorithmes permettent à la fois de retirer les points qui étaient bloquants des approches classiques, tout en maintenant un comportement similaire. De plus, ils sont à la fois plus robustes, plus génériques et présentent moins de limitations que les approches classiques. Ceci facilite la création des simulations et s'adaptent à de nombreux

cas d'utilisation.

En conclusion, nous pouvons dire que les méthodes présentées dans ce document ont permis de rendre utilisable les simulations interactives basées sur un comportement physique. Jusqu'à présent, les simulations basées sur des éléments finis restaient trop limitées (soit par le faible nombre d'éléments, soit par la simplification des équations) pour être utilisées dans de réelles applications. Nos méthodes offrent maintenant la possibilité de développer des simulations complexes, pouvant être exploitées dans de réelles applications.

6.3 Perspectives

De nombreuses pistes sont encore à explorer pour continuer ces travaux. Pour la partie des solveurs linéaires, il serait intéressant d'explorer l'utilisation de nouveaux préconditionneurs. En effet, nous avons par exemple commencé à travailler sur l'utilisation d'une factorisation $\mathbf{X}\mathbf{X}^T$, où $\mathbf{X}$ est l'inverse de $\mathbf{L}$, dans une factorisation $\mathbf{L}\mathbf{L}^T$. Le calcul de $\mathbf{X}$ est une opération coûteuse, mais qui peut être calculée dans le thread de factorisation. Le taux de remplissage de $\mathbf{X}$ reste relativement faible, ce qui permet de l'utiliser sur des maillages très détaillés. Le grand intérêt de ce préconditionneur réside dans le fait que son application est fortement parallélisable, puisqu'il s'agit simplement de multiplications matrice–vecteur. On espère alors diminuer le coût d'application du préconditionneur dans le thread de simulation.

Pour la partie de la détection des collisions basée sur la rastérisation, nous souhaiterions étendre la méthode à une grille multi-résolutions, en fonction des zones d'intérêt et de la taille des objets. En effet, notre méthode souffre actuellement du fait que la résolution des pixels est un paramètre global de la scène, et il est difficile de trouver une valeur optimale quand on veut simuler des grands objets (comme les organes) avec des objets plus petits (comme la pointe d'un instrument). Pour cela, on pourrait spécifier une taille de pixel propre à chaque objet et calculer les volumes d'intersection en fonction des différentes résolutions de grilles. De plus, nous souhaitons étendre la méthode à une détection de collision continue, ce qui permettrait d'améliorer la qualité de la réponse. En particulier, cette piste semble prometteuse pour réduire totalement les problèmes de stabilisations post-résolution.

Enfin, pour la résolution des contacts basée sur l'utilisation du préconditionneur, il reste encore à étendre la méthode à une réelle découpe des éléments. En effet, nous avons vu que nous sommes capables de supprimer des éléments, sans changer le nombre de degrés de liberté du système. Cependant, une découpe plus fine est souvent nécessaire, ce qui impose de changer les dimensions du système. Pour y parvenir, on peut utiliser le fait que la mise à jour du préconditionneur est très rapide. On peut se contenter d'une approximation grossière pendant ce temps, et espérer obtenir une nouvelle mise à jour rapidement qui prendra en compte le changement de taille de façon correcte.

D'un point de vue plus général, maintenant que nous sommes capables de construire des applications réalistes, avec un nombre importants de modèles et de paramètres associés ; une autre piste concerne la génération automatique de simulation spécifique à chaque patient. Nous avons vu que nous utilisons déjà des techniques de segmentations automatiques, pour obtenir la surface des organes à partir d'une segmentation d'un scanner.

Cependant, il reste à automatiser la génération des mailles éléments finis, notamment pour respecter les spécificités anatomiques de la surface des objets (comme l'incision de l'œil par exemple). Un autre problème qui reste encore ouvert, c'est de déterminer les paramètres des modèles mécaniques à partir des données de patients et, idéalement, de manière automatique. En effet, pour l'instant ces paramètres ont été choisis de façon arbitraire, et il serait intéressant de pouvoir les spécifier automatiquement en fonction des propriétés des organes propres à chaque patient.

Par ailleurs, ceci nous amène directement au problème de validation des applications. Nous avons déjà validé nos méthodes en comparant l'exactitude des résultats par rapport aux méthodes traditionnelles. Nous avons par exemple comparé nos résultats par rapport aux méthodes exactes, et nous avons vérifié l'exactitude du résultat par rapport aux approches séquentielles. Cependant, il serait intéressant d'évaluer l'utilité du simulateur du point de vue applicatif. Une piste serait de valider le fait que l'utilisation du simulateur a une réelle utilité pour l'apprentissage des chirurgiens. Par exemple, on peut imaginer que le simulateur permette aux étudiants d'avoir une représentation pratique de la chirurgie. Ils pourront par exemple acquérir des connaissances sur la localisation des incisions ou le placement des outils, ou encore apprendre à coordonner leurs gestes pour réaliser une tâche. Il serait intéressant de valider ces aspects pour intégrer l'utilisation d'un simulateur dans la formation des chirurgiens, en vue de diminuer le temps d'apprentissage.

Une validation validation mécanique, qui aurait pour objectif de prouver que le comportement des objets simulés, est similaire à celui des organes du patient, aurait également un grand intérêt. Le simulateur pourrait alors être utilisé avant l'opération pour évaluer différentes techniques, et certifier le bon déroulement. L'enjeu est énorme, puisqu'une telle application pourrait réduire de façon significative le risque opératoire. De plus, en combinant ce type de validation avec une méthode automatique pour générer une simulation spécifique à chaque patient, on s'attend à ce que le simulateur devienne un nouvel outil d'aide à la chirurgie.

Bibliographie

[Adams, 2001] M. F. Adams. *A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers*. In *Proc. of the 2001 ACM/IEEE conference on Supercomputing*, 2001, ISBN 1-58113-293-X. 134, 185

[Allard et al., 2007] J. Allard, S. Cotin, F. Faure, P.-J. Bensoussan, F. Poyer, C. Duriez, H. Delingette et L. Grisoni. *SOFA - an Open Source Framework for Medical Simulation*. In *Medicine Meets Virtual Reality (MMVR'15)*, pages 1–6, 2007.
URL `http://sofa-framework.org` 34, 74, 185

[Allard et al., 2010] J. Allard, F. Faure, H. Courtecuisse, F. Falipou, C. Duriez et P. G. Kry. *Volume Contact Constraints at Arbitrary Resolution*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010), vol. 29, no. 3, 2010.
URL `http://www.sofa-framework.org/projects/ldi` 185

[Allard et al., 2011a] J. Allard, H. Courtecuisse et F. Faure. *Implicit FEM and Fuild Coupling on GPU for Interactive Multi-Physics Simulation*. SIGGRAPH Talk, 2011a. 185

[Allard et al., 2011b] J. Allard, H. Courtecuisse et F. Faure. *Implicit FEM Solver on GPU for Interactive Deformation Simulation*. In *GPU Computing Gems Vol. 2*, NVIDIA/Elsevier, 2011b, to appear.
URL `http://codrt.fr/allardj/pub/2011/ACF11` 65, 185

[Anitescu et Hart, 2004] M. Anitescu et G. D. Hart. *Constraint-stabilized time-stepping approach for rigid multibody dynamics with joints, contact and friction*. International Journal for Numerical Methods in Engineering, vol. 60, no. 14, pages 2335–2371, 2004. 101, 185

[Anitescu et Potra, 1997] M. Anitescu et F. Potra. *Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems*. Nonlinear Dynamics, vol. 14, no. 3, pages 231–247, 1997. 100, 185

[Anitescu et al., 1999] M. Anitescu, F. Potra et D. Stewart. *Time-stepping for three-dimensional rigid body dynamics*. Computer Methods in Applied Mechanics and Engineering, pages 183–197, 1999. 93, 132, 185

[Ascher et Petzold, 1998] U. M. Ascher et L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998, ISBN 0898714125. 101, 185

[Baraff, 1989] D. Baraff. *Analytical methods for dynamic simulation of non-penetrating rigid bodies*. SIGGRAPH '89, vol. 23, no. 3, pages 223–232, 1989, ISSN 0097-8930. 130, 185

[Baraff, 1991] D. Baraff. *Coping with friction for non-penetrating rigid body simulation*. Computer Graphics (Proceedings of SIGGRAPH 91), vol. 25, no. 4, pages 31–41, 1991, ISSN 0097-8930. 132, 185

[Baraff, 1994] D. Baraff. *Fast Contact Force Computation for Nonpenetrating Rigid Bodies.* In *Proceedings of SIGGRAPH 94*, pages 23–34, ACM, 1994, ISBN 0-89791-667-0. 92, 186

[Baraff et Witkin, 1998] D. Baraff et A. Witkin. *Large steps in cloth simulation.* In *SIGGRAPH '98 : Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 43–54, ACM, 1998, ISBN 0-89791-999-8. 39, 59, 63, 101, 186

[Barrett et al., 1994] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine et H. V. der Vorst. *Templates for the Solution of Linear Systems : Building Blocks for Iterative Methods, 2nd Edition.* SIAM, 1994. 58, 78, 134, 147, 186

[Batty et al., 2007] C. Batty, F. Bertails et R. Bridson. *A fast variational framework for accurate solid-fluid coupling.* ACM Transactions on Graphics, vol. 26, no. 3, page 100, 2007, ISSN 0730-0301. 101, 186

[Bell et Garland, 2008] N. Bell et M. Garland. *Efficient sparse matrix-vector multiplication on CUDA.* Technical Report NVR-2008-004, NVIDIA, 2008. 58, 186

[Bolz et al., 2003] J. Bolz, I. Farmer, E. Grinspun et P. Schröoder. *Sparse matrix solvers on the GPU : conjugate gradients and multigrid.* ACM Trans. Graph., vol. 22, no. 3, pages 917–924, 2003, ISSN 0730-0301. 61, 186

[Bond, 2006] A. Bond. *Havok FX : GPU-accelerated Physics for PC Games.* In *Proc. of Game Developers Conference 2006*, 2006. 134, 186

[Bonet et Wood, 2005] J. Bonet et D. Wood. *Nonlinear continuum mechanics for finite element analysis.* 2005. 21, 186

[Bourquain et al., 2002] H. Bourquain, A. Schenk, F. Link, B. Preim, G. Prause et H. Peitgen. *HepaVision2 – A software assistant for preoperative planning in living-related liver transplantation and oncologic liver surgery.* Cancer Research, pages 1–6, 2002. 177, 186

[Boxerman et Ascher, 2004] E. Boxerman et U. Ascher. *Decomposing cloth.* In *Proceedings of SCA '04*, pages 153–161, ACM SIGGRAPH/Eurographics, 2004, ISBN 3-905673-14-2. 63, 186

[Bridson et al., 2002] R. Bridson, R. Fedkiw et J. Anderson. *Robust treatment of collisions, contact and friction for cloth animation.* In *Proceedings of SIGGRAPH 2002*, pages 594–603, ACM, 2002. 98, 101, 186

[Briggs et al., 2000] W. L. Briggs, V. E. Henson et S. F. McCormick. *A multigrid tutorial : second edition.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000, ISBN 0-89871-462-1. 63, 186

[Bro-Nielsen et Cotin, 1996] M. Bro-Nielsen et S. Cotin. *Real-time volumetric deformable models for surgery simulation using finite elements and condensation.* Comput. Graph. Forum, vol. 15, no. 3, pages 57–66, 1996. 42, 54, 186

[Brown et al., 2002] J. Brown, S. Sorkin, J.-C. Latombe, K. Montgomery et M. Stephanides. *Algorithmic tools for real-time microsurgery simulation.* Medical image analysis, vol. 6, no. 3, pages 289–300, 2002. 5, 187

[Buatois et al., 2009] L. Buatois, G. Caumon et B. Lévy. *Concurrent number cruncher - A GPU implementation of a general sparse linear solver.* Int. J. Parallel Emerg. Distrib. Syst., vol. 24, no. 3, pages 205–223, 2009, ISSN 1744-5760. 61, 187

[Camp et al., 1994] W. J. Camp, S. J. Plimpton, B. A. Hendrickson et R. W. Leland. *Massively parallel methods for engineering and science problems.* Communications of the ACM, vol. 37, no. 4, pages 30–41, 1994, ISSN 0001-0782. 65, 187

[Choi et Ko, 2002] K.-J. Choi et H.-S. Ko. *Stable but responsive cloth.* ACM Trans. Graph., vol. 21, no. 3, pages 604–611, 2002, ISSN 0730-0301. 63, 187

[Choi et Ko, 2005] M. G. Choi et H.-S. Ko. *Modal Warping : Real-Time Simulation of Large Rotational Deformation and Manipulation.* IEEE Transactions on Visualization and Computer Graphics, vol. 11, pages 91–101, 2005, ISSN 1077-2626.
URL `http://dx.doi.org/10.1109/TVCG.2005.13` 21, 187

[Chowdhury et Dasgupta, 2003] I. Chowdhury et S. Dasgupta. *Computation of Rayleigh Damping Coefficients for Large Systems.* The Electronic Journal of Geotechnical Engineering, vol. 8, 2003. 32, 187

[Cline et Pai, 2003] M. B. Cline et D. K. Pai. *Post-Stabilization for Rigid Body Simulation with Contact and Constraints.* In *IEEE International Conference on Robotics and Automation*, 2003. 101, 187

[Comas et al., 2008] O. Comas, Z. Taylor, J. Allard, S. Ourselin, S. Cotin et J. Passenger. *Efficient nonlinear FEM for soft tissue modelling and its GPU implementation within the open source framework SOFA.* In *ISBMS*, pages 28–39, 2008.
URL `http:///allardj/pub/2008/CTAOCP08` 37, 187

[Comas et al., 2010] O. Comas, S. Cotin et C. Duriez. *A Shell Model for Real-Time Simulation of Intra-ocular Implant Deployment.* In *Int. Symp. on Biomedical Simulation (ISBMS)*, pages 160–170, 2010. 173, 187

[Conte et Boor, 1980] S. D. Conte et C. W. D. Boor. *Elementary Numerical Analysis : An Algorithmic Approach.* McGraw-Hill Higher Education, 1980, ISBN 0070124477. 145, 187

[Cook, 1981] R. D. Cook. *Concepts and Applications of Finite Element Analysis.* 1981. 32, 187

[Cotin, 1997] S. Cotin. *Modèles Anatomiques Déformables en temps-Réel - Application à la simulation de la chirurgie avec retour d'effort.* 1997. 25, 187

[Cotin et al., 1999] S. Cotin, H. Delingette et N. Ayache. *Real-Time Elastic Deformations of Soft Tissues for Surgery Simulation.* IEEE Transactions on Visualization and Computer Graphics, vol. 5, no. 1, pages 62–73, 1999. 42, 187

[Cotin et al., 2005] S. Cotin, C. Duriez, J. Lenoir, P. F. Neumann et S. Dawson. *New Approaches to Catheter Navigation for Interventional Radiology Simulation.* In *MICCAI (2)'05*, pages 534–542, 2005. 40, 188

[Courtecuisse et Allard, 2009] H. Courtecuisse et J. Allard. *Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors.* In *High Performance Computation Conference (HPCC)*, IEEE CS Press, 2009.
URL `http:///allardj/pub/2009/CA09` 134, 188

[Courtecuisse et al., 2010] H. Courtecuisse, J. Allard, C. Duriez et S. Cotin. *Asynchronous Preconditioners for Efficient Solving of Non-linear Deformations.* In *Proceedings of Virtual Reality Interaction and Physical Simulation (VRIPHYS)*, 2010.
URL `http://codrt.fr/allardj/pub/2010/CADC10` 76, 188

[Courtecuisse et al., 2011a] H. Courtecuisse, J. Allard, C. Duriez et S. Cotin. *Preconditioner-Based Contact Response and Application to Cataract Surgery.* In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2011a.
URL `http://www.lifl.fr/~courtecu/` 154, 188

[Courtecuisse et al., 2011b] H. Courtecuisse, J. Allard, C. Duriez, I. Peterlik, S. Cotin et L. Soler. *GPU-based interactive simulation of liver resection Real-Time Live.* SIGGRAPH Real Time Live Demo, 2011b. 180, 188

[Courtecuisse et al., 2011c] H. Courtecuisse, H. Jung, J. Allard, C. Duriez, D. Y. Lee et S. Cotin. *GPU-based Real-Time Soft Tissue Deformation with Cutting and Haptic Feedback.* Progress in Biophysics and Molecular Biology, 2011c, special Issue on Soft Tissue Modelling.
URL `http://codrt.fr/allardj/pub/2011/CJADLC11` 145, 188

[Crisfield, 1997] M. A. Crisfield. *Non-Linear Finite Element Analysis of Solids and Structures : Advanced Topics.* John Wiley & Sons, Inc., New York, NY, USA, 1st ed., 1997, ISBN 047195649X. 36, 188

[Davis, 2006] T. A. Davis. *CSparse.* Society for Industrial and Applied Mathematics, Philadephia, PA, 2006, ISBN 9780898718881.
URL `http://link.aip.org/link/doi/10.1137/1.9780898718881.ch9` 53, 56, 188

[Delingette, 2008] H. Delingette. *Biquadratic and Quadratic Springs for Modeling St Venant Kirchhoff Materials.* In *Proceedings of ISBMS 2008*, pages 40–48, 2008. 39, 188

[Dequidt et al., 2008] J. Dequidt, M. Marchal, C. Duriez, E. Kerrien et S. Cotin. *Interactive Simulation of Embolization Coils : Modeling and Experimental Validation.* In *MICCAI (1)'08*, pages 695–702, 2008. 40, 188

[Dequidt et al., 2009] J. Dequidt, C. Duriez, S. Cotin et E. Kerrien. *Towards Interactive Planning of Coil Embolization in Brain Aneurysms.* In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2009*, vol. 5761 of *Lecture Notes in Computer Science*, edited by G.-Z. Yang, D. Hawkes, D. Rueckert, A. Noble et C. Taylor, pages 377–385, Springer Berlin / Heidelberg, 2009, ISBN 978-3-642-04267-6. 171, 188

[Duriez, 2004] C. Duriez. *Contact frottant entre objets déformables dans des simulations temps-réel avec retour haptique*. 2004. 25, 100, 189

[Duriez et al., 2006] C. Duriez, F. Dubois, A. Kheddar et C. Andriot. *Realistic Haptic Rendering of Interacting Deformable Objects in Virtual Environments*. IEEE Transactions on Visualization and Computer Graphics, vol. 12, no. 1, pages 36–47, 2006, ISSN 1077-2626. 129, 131, 132, 189

[Duriez et al., 2008] C. Duriez, H. Courtecuisse, J. Alcalde et P. Bensoussan. *Contact Skinning*. In *EUROGRAPHICS 2008 / K. Mania and E. Reinhard Short Papers Contact Skinning*, ACM Press, 2008. 189

[Duriez et al., 2009] C. Duriez, C. Guébert, M. Marchal, S. Cotin et L. Grisoni. *Interactive Simulation of Flexible Needle Insertions Based on Constraint Models*. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2009*, vol. 5762 of *Lecture Notes in Computer Science*, edited by G.-Z. Yang, D. Hawkes, D. Rueckert, A. Noble et C. Taylor, pages 291–299, Springer Berlin / Heidelberg, 2009.
URL http://dx.doi.org/10.1007/978-3-642-04271-3_36 40, 189

[Faure et al., 2008] F. Faure, S. Barbier, J. Allard et F. Falipou. *Image-based Collision Detection and Response between Arbitrary Volumetric Objects*. In *ACM Siggraph/Eurographics Symposium on Computer Animation (SCA 2008)*, 2008. 101, 102, 103, 109, 113, 129, 189

[Forest et al., 2004] C. Forest, H. Delingette et N. Ayache. *Surface Contact and Reaction Force Models for Laparoscopic Simulation*. In *International Symposium on Medical Simulation*, 2004. 6, 189

[Frisken et al., 2000] S. F. Frisken, R. N. Perry, A. P. Rockwood et T. R. Jones. *Adaptively Sampled Distance Fields : A General Representation of Shape for Computer Graphics*. In *Proceedings of SIGGRAPH 2000*, edited by K. Akeley, pages 249–254, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.
URL citeseer.ist.psu.edu/frisken00adaptively.html 95, 189

[Fuhrmann et al., 2003] A. Fuhrmann, G. Sobottka et C. Gross. *Distance fields for rapid collision detection in physically based modeling*. In *International Conference on Computer Graphics and Vision (Graphicon)*, 2003. 94, 189

[Galoppo et al., 2005] N. Galoppo, N. K. Govindaraju, M. Henson et D. Manocha. *LU-GPU : Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*. In *Proc. of the 2005 ACM/IEEE conference on Supercomputing*, 2005. 56, 189

[Galoppo et al., 2006] N. Galoppo, M. A. Otaduy, P. Mecklenburg, M. Gross et M. C. Lin. *Fast simulation of deformable models in contact using dynamic deformation textures*. In *Symposium on Computer animation (SCA)*, pages 73–82, 2006, ISBN 3-905673-34-7.
URL http://portal.acm.org/citation.cfm?id=1218064.1218074 99, 189

[García et al., 2006] M. García, C. Mendoza, L. Pastor et A. Rodríguez. *Optimized linear FEM for modeling deformable objects*. Comput. Animat. Virtual Worlds, vol. 17, pages 393–402, 2006, ISSN 1546-4261. 63, 76, 189

[Garrigues, 2001] J. Garrigues. *Eléments d'algèbre et d'analyse tensorielle à l'usage des mécaniciens*. 2001.
URL http://dx.doi.org/10.1007/978-3-642-04271-3_36 19, 21, 190

[Gaster et al., 2011] B. Gaster, D. Kaeli, L. Howes, P. Mistry et D. Schaa. *Heterogeneous Computing With Opencl*. Elsevier Science & Technology, 2011, ISBN 9780123877666.
URL http://books.google.com/books?id=dK70tgAACAAJ 45, 190

[Georgii et Westermann, 2006] J. Georgii et R. Westermann. *A multigrid framework for real-time simulation of deformable bodies*. Comput. Graph., vol. 30, pages 408–415, 2006, ISSN 0097-8493.
URL http://portal.acm.org/citation.cfm?id=1652316.1652571 21, 190

[Gottschalk et al., 1996] S. Gottschalk, M. C. Lin et D. Manocha. *OBBTree : a hierarchical structure for rapid interference detection*. In *Proceedings of SIGGRAPH 96*, pages 171–180, ACM, 1996. 94, 190

[Govindaraju et al., 2003] N. K. Govindaraju, S. Redon, M. C. Lin et D. Manocha. *CULLIDE : interactive collision detection between complex models in large environments using graphics hardware*. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 25–32, Eurographics Association, 2003, ISBN 1-58113-739-7.
URL http://dl.acm.org/citation.cfm?id=844174.844178 95, 190

[Grantcharov et al., 2004] T. Grantcharov, V. Kristianson, J. Bendix, L. Bardram, J. Rosenerg et P. Funch-Jensen. *Randomized clinical trial of virtual reality simulation for laparoscopic skills training*. Br J Surg., vol. 91, pages 146–150, 2004. 6, 190

[Grinspun et al., 2002] E. Grinspun, P. Krysl et P. Schröder. *CHARMS : A Simple Framework for Adaptive Simulation*. In *ACM Transactions on Graphics*, pages 281–290, ACM Press, 2002. 30, 190

[Guendelman et al., 2003] E. Guendelman, R. Bridson et R. Fedkiw. *Nonconvex rigid bodies with stacking*. ACM Trans. Graph., vol. 22, pages 871–878, 2003, ISSN 0730-0301.
URL http://doi.acm.org/10.1145/882262.882358 101, 190

[Hang] S. Hang. *TetGen : A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*. http ://tetgen.berlios.de/.
URL http://tetgen.berlios.de/ 28, 190

[Harders, 2008] M. Harders. *Haptic Rendering : Algorithms and Applications*, chap. Haptics in Medical Applications, pages 589–612. A K Peters, Ltd. M.Lin et M.Otaduy, 2008. 6, 190

[Harmon et al., 2008] D. Harmon, E. Vouga, R. Tamstorf et E. Grinspun. *Robust treatment of simultaneous collisions*. ACM Transactions on Graphics, vol. 27, no. 3, pages 1–4, 2008. 101, 190

[Hauth et Straßer, 2004] M. Hauth et W. Straßer. *Corotational Simulation of Deformable Solids*. In *Proceedings of WSCG*, pages 137–145, 2004. 43, 190

[Hauth et al., 2003] M. Hauth, O. Etzmuß et W. Straßer. *Analysis of numerical methods for the simulation of deformable models*. The Visual Computer, vol. 19, no. 7-8, pages 581–600, 2003. 63, 191

[Heidelberger et al., 2003] B. Heidelberger, M. Teschner et M. Gross. *Real-time volumetric intersections of deforming objects*. In *Proc. of Vision, Modeling, Visualization (VMV)*, 2003. 95, 96, 191

[Heidelberger et al., 2004] B. Heidelberger, M. Teschner et M. Gross. *Detection of Collisions and Self-collisions Using Image-space Techniques*. In *Proceedings of WSCG'04*, pages 145–152, 2004. 95, 96, 191

[HelpMeSee, 2009] HelpMeSee. 2009.
URL `http://helpmesee.org/about/statement-of-purpose/` 173, 191

[Hermann et al., 2009] E. Hermann, B. Raffin et F. Faure. *Interative Physics Simulation on Multicore Architectures*. In *Proceedings of the 9th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV'09)*, 2009. 61, 191

[Hestenes et Stiefel, 1952] M. R. Hestenes et E. L. Stiefel. *Methods of Conjugate Gradients for Solving Linear Systems*. Journal of Research of the National Bureau of Standards, vol. 409-432, no. 6, pages 33–53, 1952. 59, 191

[Hetnarski et Ignaczak, 2006] R. B. Hetnarski et J. Ignaczak. *Mathematical Theory of Elasticity*. Taylor and Francis, 2004, 2006. 26, 191

[Hirota et al., 2001] G. Hirota, S. Fisher, A. State, H. Fuchs et C. Lee. *An Implicit Finite Element Method for Elastic Solids in Contact*. In *Proceedings Computer Animation*, 2001. 98, 191

[Hubbard, 1995] P. M. Hubbard. *Collision Detection for Interactive Graphics Applications*. Ph.D. thesis, Brown University, 1995. 93, 191

[INRIA] INRIA. *SOFA Simulation Open Framework Architecture*. 9, 191

[Irving et al., 2007] G. Irving, C. Schroeder et R. Fedkiw. *Volume conserving finite element simulations of deformable models*. ACM Trans. Graph., vol. 26, 2007, ISSN 0730-0301.
URL `http://doi.acm.org/10.1145/1276377.1276394` 113, 191

[James et Pai, 1999] D. James et D. Pai. *ARTDEFO : Accurate real time deformable objects*. In *26th International Conference on Computer Graphics and Interactive Techniques*, Proceedings of SIGGRAPH, ACM, pages 65–72, 1999. 42, 191

[James et Pai, 2004] D. L. James et D. K. Pai. *BD-tree : output-sensitive collision detection for reduced deformable models*. ACM Transactions on Graphics, vol. 23, no. 3, pages 393–398, 2004. 94, 98, 191

[Johnson et Willemsen, 2004] D. Johnson et P. Willemsen. *Accelerated haptic rendering of polygonal models through local descent*. In *Proceedings of the Symposium on Haptic Interfaces for Virtual Environments and Teleoperator Systems (HAPTICS 2004).*, 2004. 101, 191

[Joldes et al., 2009a] G. R. Joldes, A. Wittek, M. Couton, S. K. Warfield et K. Miller. *Real-Time Prediction of Brain Shift Using Nonlinear Finite Element Algorithms*. In *Proceedings of MICCAI 2009*, pages 300–307, Springer-Verlag, Berlin, Heidelberg, 2009a, ISBN 978-3-642-04270-6. 37, 192

[Joldes et al., 2009b] G. R. Joldes, A. Wittek et K. Miller. *Suite of finite element algorithms for accurate computation of soft tissue deformation for surgical simulation*. Medical Image Analysis, vol. 13, no. 6, pages 912 – 919, 2009b, ISSN 1361-8415, includes Special Section on Computational Biomechanics for Medicine.
URL http://www.sciencedirect.com/science/article/B6W6Y-4V70NKS-1/2/fb564b94a71de20ceb469f6ea39f361e 37, 192

[Jourdan et al., 1998] F. Jourdan, P. Alart et M. Jean. *A Gauss-Seidel like algorithm to solve frictional contact problems*. Comp. Meth. in Appl. Mech. and Engin., pages 33–47, 1998. 132, 192

[Joussemet et al., 2006] L. Joussemet, A. Crosnier et C. Andriot. *ESPIONS : A Novel Algorithm Based on Evolution Strategy for Fast Collision Detection*. In *EuroHaptics 2006*, pages 159–167, 2006. 94, 192

[Jund et al., 2010] T. Jund, D. Cazier et J.-F. Dufourd. *Edge collision detection in complex deformable environments*. In *Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)*, pages 69–78, Eurographics, Eurographics Association, 2010.
URL http://lsiit-cnrs.unistra.fr/Publications/2010/4-JCD10a 92, 192

[Kaufman et al., 2008] D. M. Kaufman, S. Sueda, D. L. James et D. K. Pai. *Staggered projections for frictional contact in multibody systems*. ACM Transactions on Graphics, vol. 27, no. 5, pages 1–11, 2008. 130, 192

[Kaufmann et al., 2008] P. Kaufmann, S. Martin, M. Botsch et M. Gross. *Flexible simulation of deformable models using discontinuous Galerkin FEM*. Graph. Models, vol. 71, pages 153–167, 2008, ISSN 1524-0703.
URL http://portal.acm.org/citation.cfm?id=1550957.1551026 30, 192

[Kipfer, 2007] P. Kipfer. *LCP Algorithms for Collision Detection Using CUDA*. In *GPU Gems 3*, chap. 33, pages 723–740, Addison-Wesley, 2007. 134, 192

[Klein et Zachmann, 2003] J. Klein et G. Zachmann. *ADB-Trees : Controlling the Error of Time-Critical Collision Detection*. In *8th International Fall Workshop Vision, Modeling, and Visualization (VMV)*, University München, Germany, 2003, ISBN 1-58603-393-X.
URL http://www.gabrielzachmann.org/ 94, 192

[Koester et al., 1994] D. P. Koester, S. Ranka et G. C. Fox. *A parallel Gauss-Seidel algorithm for sparse power system matrices*. In *Proc. of the 1994 ACM/IEEE conference on Supercomputing*, pages 184–193, 1994, ISBN 0-8186-6605-6. 134, 192

[Kumar et al., 1993] S. Kumar, Shashi et A. Pethö. *An algorithm for the numerical inversion of a tridiagonal matrix*. Communications in Numerical Methods in Engineering, vol. 9, no. 4, pages 353–359, 1993, ISSN 1099-0887.
URL http://dx.doi.org/10.1002/cnm.1640090409 40, 192

[Lamadé et al., 2002] W. Lamadé, M. Vetter, P. Hassenpflug, M. Thorn, H.-P. Meinzer et C. Herfarth. *Navigation and image-guided HBP surgery : a review and preview.* Journal of Hepato-Biliary-Pancreatic Surgery, vol. 9, pages 592–599, 2002, ISSN 0944-1166, 10.1007/s005340200079.
URL http://dx.doi.org/10.1007/s005340200079 177, 193

[Lanczos, 1952] C. Lanczos. *Solution of systems of linear equations by minimized iterations.* J. Res. Natl. Bur. Stand, vol. 49, pages 33–53, 1952. 59, 193

[Larsson et Akenine-Moller, 2001] T. Larsson et T. Akenine-Moller. *Collision detection for continuously deforming bodies.* In *Eurographics 2001*, pages 325–333, 2001. 92, 193

[Lawson et al., 1979] C. L. Lawson, R. J. Hanson, D. R. Kincaid et F. T. Krogh. *Basic Linear Algebra Subprograms for Fortran Usage.* ACM Trans. Math. Softw., vol. 5, pages 308–323, 1979, ISSN 0098-3500.
URL http://doi.acm.org/10.1145/355841.355847 56, 193

[Lee et al., 2005] B. Lee, D. C. Popescu, B. Joshi et S. Ourselin. *Efficient Topology Modification and Deformation for Finite Element Models Using Condensation.* In *Medicine Meets Virtual Reality*, pages 299–304, 2005. 131, 193

[Lenoir et al., 2004] J. Lenoir, L. Grisoni, P. Meseure, Y. Rémion et C. Chaillou. *Smooth constraints for spline variational modeling.* In *Proc. of GRAPHITE*, pages 58–64, ACM, 2004, ISBN 1-58113-883-0.
URL http://doi.acm.org/10.1145/988834.988844 100, 193

[Lin et Canny, 1992] M. C. Lin et J. Canny. *Efficient Collision Detection for Animation and Robotics.* Tech. rep., 1992. 94, 193

[Lötstedt, 1984] P. Lötstedt. *Numerical Simulation of Time-Dependent Contact and Friction Problems in Rigid Body Mechanics.* vol. 5, no. 2, pages 370–393, 1984, ISSN 10648275.
URL http://dx.doi.org/doi/10.1137/0905028 130, 193

[Marchesseau et al., 2010] S. Marchesseau, T. Heimann, S. Chatelin, R. Willinger et H. Delingette. *Multiplicative Jacobian Energy Decomposition Method for Fast Porous Visco-Hyperelastic Soft Tissue Model.* MICCAI'10, 2010. 26, 43, 84, 164, 193

[Marescaux et al., 1998] J. Marescaux, J.-M. Clement, V. Tassetti, C. Koehl, S. Cotin, Y. Russier, D. Mutter, H. Delingette et N. Ayache. *Virtual Reality Applied to Hepatic Surgery Simulation : The Next Revolution.* Annals of Surgery, vol. 228, no. 5, pages 627–634, 1998. 5, 193

[Milenkovic et Schmidl, 2001] V. J. Milenkovic et H. Schmidl. *Optimization-based animation.* In *Proceedings of SIGGRAPH 2001*, pages 37–46, ACM, 2001, ISBN 1-58113-374-X. 132, 193

[Miller, 1988] G. S. P. Miller. *The motion dynamics of snakes and worms.* In *Proceedings of SIGGRAPH '88*, pages 169–173, ACM, 1988, ISBN 0-89791-275-6. 39, 193

[Mirtich et Canny, 1995] B. Mirtich et J. Canny. *Impulse-based simulation of rigid bodies.* In *Interactive 3D graphics*, 1995. 92, 193

[Molino et al., 2007] N. Molino, Z. Bao et R. Fedkiw. *A Virtual Node Algorithm for Changing Mesh Topology During Simulation*. Proceeding of Eurographics, pages 73–80, 2007. 44, 194

[Montgomery et al., 2002] K. Montgomery, C. Bruyns, J. Brown, G. Thonier, A. Tellier et J.-C. Latombe. *Spring : A General Framework for Collaborative, Real-Time Surgical Simulation*. In *Medicine Meets Virtual Reality (MMVR02)*, 2002. 39, 194

[Moore et Wilhelms, 1988] M. Moore et J. Wilhelms. *Collision Detection and Response for Computer Animation*. In *Computer Graphics (Proceedings of SIGGRAPH 88*, pages 289–298, ACM, 1988. 98, 194

[Mor et Kanade, 2000] A. B. Mor et T. Kanade. *Modifying Soft Tissue Models : Progressive Cutting with Minimal New Element Creation*. In *Proceedings of MICCAI 2000*, pages 598–607, 2000. 44, 194

[Müller et Gross, 2004] M. Müller et M. Gross. *Interactive virtual materials*. In *GI '04 : Proc. of Graphics Interface 2004*, pages 239–246, Canadian Human-Computer Communications Society, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004, ISBN 1-56881-227-2. 43, 194

[Müller et al., 2002] M. Müller, J. Dorsey, L. McMillan, R. Jagnow et B. Cutler. *Stable real-time deformations*. In *Proceedings of SCA '02*, pages 49–54, ACM SIGGRAPH/Eurographics, 2002, ISBN 1-58113-573-4. 21, 42, 194

[Müller et al., 2005] M. Müller, B. Heidelberger, M. Teschner et M. Gross. *Meshless deformations based on shape matching*. ACM Trans. Graph., vol. 24, no. 3, pages 471–478, 2005, ISSN 0730-0301. 42, 80, 194

[Munshi, 2008] A. Munshi (editor). *The OpenCL Specification Version : 1.0*. The Khronos Group, 2008. 12, 194

[Murty, 1997] K. Murty. *Linear Complementarity, Linear and Nonlinear Programming*. Internet Edition, 1997. 130, 133, 194

[Nealen et al., 2006] A. Nealen, M. Mueller, R. Keiser, E. Boxerman et M. Carlson. *Physically Based Deformable Models in Computer Graphics*. Comput. Graph. Forum, vol. 25, 2006. 17, 194

[Nesme et al., 2005a] M. Nesme, M. Marchal, E. Promayon, M. Chabanas, Y. Payan et F. Faure. *Physically realistic interactive simulation for biological soft tissues*. Recent Research Developments in Biomechanics, vol. 2, pages 1–22, 2005a.
URL `http://hal.archives-ouvertes.fr/hal-00080378/en/` 71, 194

[Nesme et al., 2005b] M. Nesme, Y. Payan et F. Faure. *Efficient, Physically Plausible Finite Elements*. In *Eurographics 2005 - Short Papers*, pages 77–80, 2005b. 43, 121, 194

[NVIDIA Corporation, 2007a] NVIDIA Corporation. *NVIDIA CUBLAS Library*. 2007a. 66, 151, 158, 194

[NVIDIA Corporation, 2007b] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. 2007b. 11, 195

[Otaduy et al., 2009] M. A. Otaduy, R. Tamstorf, D. Steinemann et M. Gross. *Implicit Contact Handling for Deformable Objects*. Computer Graphics Forum (Proceedings of Eurographics), vol. 28, no. 2, pages 559–568, 2009.
URL `http://www.gmrv.es/Publications/2009/OTSG09` 101, 195

[Pabst et al., 2009] S. Pabst, B. Thomaszewski et W. Straßer. *Anisotropic friction for deformable surfaces and solids*. In *SCA '09 : Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 149–154, 2009, ISBN 978-1-60558-610-6. 101, 195

[Parker et O'Brien, 2009] E. G. Parker et J. F. O'Brien. *Real-time deformation and fracture in a game environment*. In *SCA '09 : Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 165–175, 2009, ISBN 978-1-60558-610-6.
URL `http://graphics.cs.berkeley.edu/papers/Parker-RTD-2009-08/` 61, 195

[Peercy et al., 2006] M. Peercy, M. Segal et D. Gerstmann. *A performance-oriented data parallel virtual machine for GPUs*. In *SIGGRAPH '06 : ACM SIGGRAPH 2006 Sketches*, page 184, ACM, 2006, ISBN 1-59593-364-6. 11, 195

[Picinbono et al., 2000] G. Picinbono, H. Delingette, H. Delingette, N. Ayache et N. Ayache. *Non-Linear Anisotropic Elasticity for Real-Time Surgery Simulation*. Graphical Models, vol. 65, pages 305–321, 2000. 5, 195

[Popescu et Compton, 2003] D. C. Popescu et M. Compton. *A model for efficient and accurate interaction with elastic objects in haptic virtual environments*. In *Proc. of GRAPHITE*, pages 245–250, ACM, 2003, ISBN 1-58113-578-5.
URL `http://doi.acm.org/10.1145/604471.604518` 99, 195

[Provot, 1997] X. Provot. *Collision and Self-Collision Handling in Cloth Model Dedicated to Design Garments*. In *Proceedings of 8th Eurographics Workshop on Animation and Simulation*, pages 177–189, 1997. 94, 195

[Quintana-Ortí et al., 2009] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí et R. van de Geijn. *Solving Dense Linear Algebra Problems on Platforms with Multiple Hardware Accelerators*. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009. 56, 195

[Reddy, 1984] J. N. Reddy. *An introduction to the finite element method*. 1984. 27, 195

[Renouf et al., 2004] M. Renouf, F. Dubois et P. Alart. *A parallel version of the non smooth contact dynamics algorithm applied to the simulation of granular media*. J. Comput. Appl. Math., vol. 168, no. 1-2, pages 375–382, 2004, ISSN 0377-0427. 134, 140, 195

[Sanders et Kandrot, 2010] J. Sanders et E. Kandrot. *CUDA by Example : An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1 ed., 2010, ISBN 0131387685.

URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0131387685 45, 195

[Satish et al., 2009] N. Satish, M. Harris et M. Garland. *Designing Efficient Sorting Algorithms for Manycore GPUs*. In *Proc. of IEEE Int. Parallel & Distributed Processing Symposium 2009 (IPDPS 2009)*, 2009. 106, 196

[Saupin et al., 2008a] G. Saupin, C. Duriez et S. Cotin. *Contact Model for Haptic Medical Simulations*. In *ISBMS '08 : Proceedings of the 4th international symposium on Biomedical Simulation*, pages 157–165, 2008a, ISBN 978-3-540-70520-8. 133, 196

[Saupin et al., 2008b] G. Saupin, C. Duriez, S. Cotin et L. Grisoni. *Efficient Contact Modeling using Compliance Warping*. In *Computer Graphics International Conference*, 2008b. 79, 131, 132, 148, 196

[Schenk et Gärtner, 2004] O. Schenk et K. Gärtner. *Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO*. Journal of Future Generation Computer Systems, vol. 20, pages 475–487, 2004. 56, 196

[Schenk et al., 2006] O. Schenk, M. Bollhöfer et R. A. Römer. *On Large-Scale Diagonalization Techniques for the Anderson Model of Localization*. SIAM J. Sci. Comput., vol. 28, no. 3, pages 963–983, 2006, ISSN 1064-8275. 83, 196

[Schloegel et al., 2000] K. Schloegel, G. Karypis et V. Kumar. *A Unified Algorithm for Load-balancing Adaptive Scientific Simulations*. In *Supercomputing*, 2000, ISSN 1063-9535. 56, 57, 64, 196

[Seiler et al., 2008] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan et P. Hanrahan. *Larrabee : a many-core x86 architecture for visual computing*. ACM Trans. Graph., vol. 27, no. 3, pages 1–15, 2008, ISSN 0730-0301. 106, 196

[Seymour et al., 2002] N. Seymour, A. Gallagher, M. O'Brien, S. Roman, D. Andersen et R. Satava. *Virtual reality training improves operating room performance : results of a randomized, double-blinded study*. Ann Surg., vol. 236, no. 3, pages 458–464, 2002. 6, 196

[Shewchuk, 1994] J. R. Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain*. Tech. Rep. CMU-CS-TR-94-125, Carnegie Mellon University, 1994. 60, 62, 196, 204

[Shinya et Forgue, 1991] M. Shinya et M.-C. Forgue. *Interference detection through rasterization*. The Journal of Visualization and Computer Animation, vol. 2, no. 4, pages 132–134, 1991, ISSN 1099-1778.
URL http://dx.doi.org/10.1002/vis.4340020408 95, 196

[Sifakis et al., 2007] E. Sifakis, K. G. Der et R. Fedkiw. *Arbitrary Cutting of Deformable Tetrahedralized Objects*. Proceeding of Eurographics, pages 73–80, 2007. 44, 196

[Sorensen et al., 2006] T. Sorensen, G. Greil, O. Hansen et J. Mosegaard. *Surgical simulation–a new tool to evaluate surgical incisions in congenital heart disease ?* Interactive cardiovascular and thoracic surgery, vol. 5, no. 5, pages 536–539, 2006. 39, 196

[Stewart, 2000] D. E. Stewart. *Rigid-Body Dynamics with Friction and Impact.* SIAM Review, vol. 42, no. 1, pages 3–39, 2000. 100, 197

[Stewart et Trinkle, 1996] D. E. Stewart et J. C. Trinkle. *An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction.* International Journal of Numerical Methods Engineering, vol. 39, no. 15, pages 2673–2691, 1996. 100, 197

[Tang et al., 2008] M. Tang, S. Curtis, S.-E. Yoon et D. Manocha. *Interactive continuous collision detection between deformable models using connectivity-based culling.* In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, SPM '08, pages 25–36, ACM, New York, NY, USA, 2008, ISBN 978-1-60558-106-4.
URL `http://doi.acm.org/10.1145/1364901.1364908` 92, 197

[Tchiboukdjian et al., 2010] M. Tchiboukdjian, V. Danjean et B. Raffin. *Binary Mesh Partitioning for Cache-Efficient Visualization.* IEEE Trans. on Vis. and Comp. Graph., vol. 16, no. 5, pages 815–828, 2010. 68, 74, 197

[Teschner et al., 2003] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets et M. Gross. *Optimized spatial hashing for collision detection of deformable objects.* In *Proceedings of Vision, Modeling, Visualization (VMV)*, 2003. 94, 197

[Teschner et al., 2005] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, W. Strasser et P. Volino. *Collision Detection for Deformable Objects.* Computer Graphics Forum, vol. 24, no. 1, pages 61–81, 2005.
URL `http://www-evasion.imag.fr/Publications/2005/TKHZRFCFMSV05` 93, 94, 197

[The CGAL Project, 2011] The CGAL Project. *CGAL User and Reference Manual.* CGAL Editorial Board, 3.8 ed., 2011, http //www.cgal.org/Manual/3.8/doc_html/cgal_manual/packages.html. 28, 174, 178, 197

[Toledo et al., 2003] S. Toledo, D. Chen et V. Rotkin. *TAUCS : A Library of Sparse Linear Solvers Version 2.2.* 2003. 56, 86, 197

[Tonge et al., 2010] R. Tonge, B. Wyatt et N. Nicholson. *PhysX GPU Rigid Bodies in Batman : Arkham Asylum.* In *Game Programming Gems 8*, chap. 7, pages 590–601, Cengage, 2010. 65, 197

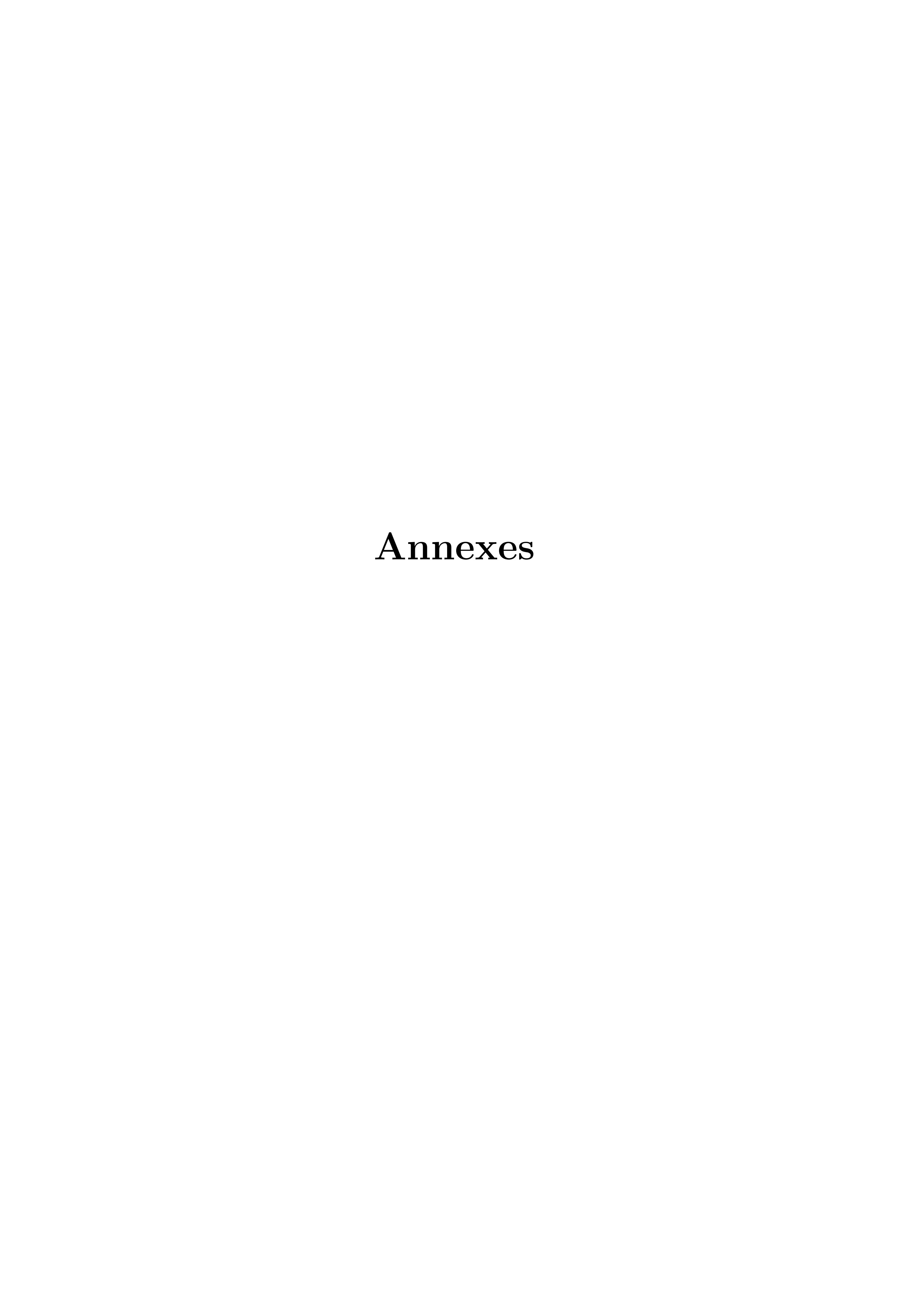
[van den Bergen, 1997] G. van den Bergen. *Efficient collision detection of complex deformable models using AABB trees.* Journal of Graphics Tools, vol. 2, no. 4, pages 1–13, 1997. 94, 197

[Volino et Magnenat-Thalmann, 1995] P. Volino et N. Magnenat-Thalmann. *Collision and Self-Collision Detection : Efficient and Robust Solutions for Higly Deformable Surfaces.* In *Computer Animation and Simulation '95*, pages 55–65, 1995. 94, 197

[Wagner et al., 2002] C. Wagner, M. Schill et R. Männer. *Intraocular surgery on a virtual eye.* Commun. ACM, vol. 45, pages 45–49, 2002, ISSN 0001-0782.
URL `http://doi.acm.org/10.1145/514236.514262` 173, 197

[Zhu et al., 2010] Y. Zhu, E. Sifakis, J. Teran et A. Brandt. *An efficient multigrid method for the simulation of high-resolution elastic solids*. ACM Trans. Graph., vol. 29, no. 2, pages 1–18, 2010, ISSN 0730-0301. 64, 198

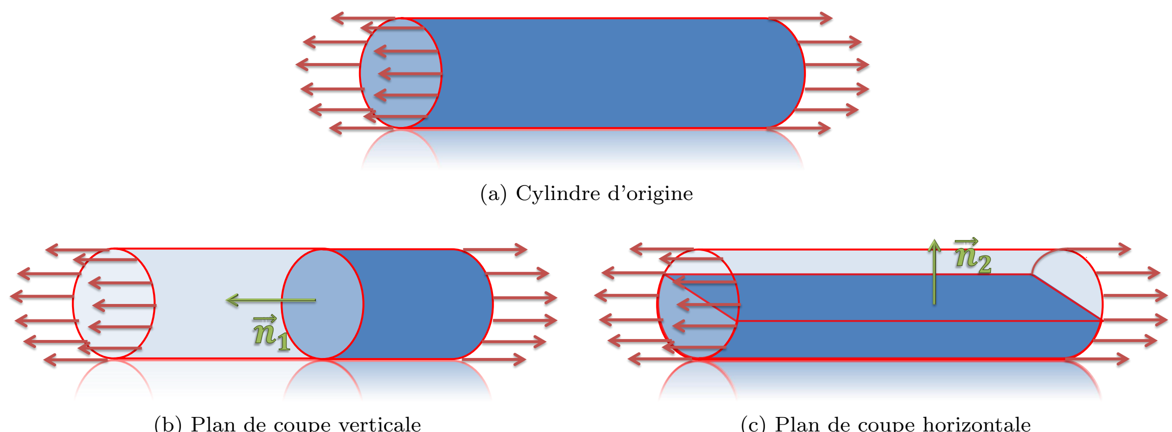
[Østergaard Noe et al., 2008] K. Østergaard Noe, K. Tanderup, J. C. Lindegaard, C. Grau et T. S. Sørensen. *GPU Accelerated Viscous-fluid Deformable Registration for Radiotherapy*. Studies in health technology and informatics, vol. 132, pages 327–332, 2008. 59, 198



Annexes

Définition du tenseur de contrainte

Le vecteur contrainte ne suffit pas à caractériser la contrainte en un point matériel en raison de sa dépendance vis-à-vis de la normale. En effet, considérons une poutre déformable sollicitée en traction le long de son axe (pas de cisaillement), sur laquelle on néglige les efforts extérieurs tels que la force de gravité (voir figure A.1). Si on place la normale du plan de coupe dans l'axe de la poutre (voir figure A.1b), on peut exprimer le vecteur contrainte par $\vec{\boldsymbol{c}}(M, \vec{n_1}) = \frac{\vec{F}}{S}\vec{n_1}$. Alors que dans le cas où le plan de coupe est longitudinale (voir figure A.1c), $\vec{\boldsymbol{c}}(M, \vec{n_2}) = 0$. Ainsi, en un même point, on peut avoir deux vecteurs contraintes différents.



(a) Cylindre d'origine

(b) Plan de coupe verticale

(c) Plan de coupe horizontale

Fig. A.1 – Transposition des efforts internes en efforts de surface, en un point donné, à l'aide de différents plans de coupe.

Mesure de la contrainte

Comme précédemment, on va transposer la notion de force ponctuelle appliquée à un point de l'objet, en un tenseur de contraintes. Considérons l'infime portion de matières contenue à l'intérieur d'un tétraèdre dont les faces sont alignées avec le repère de base orthonormée $(\vec{e}_1, \vec{e}_2, \vec{e}_3)$ (voir figure 2.3). Le tétraèdre étant considéré infiniment petit,

on peut se limiter à l'étude de la force appliquée en un point M au centre du repère. On note alors dS_i et $\vec{c}_i(M, -\vec{e}_i)$, respectivement l'aire et le vecteur contrainte de la face orthogonale à la normale $\vec{e}_i$. Il reste à définir la dernière face, de normale $\vec{n}(n_1, n_2, n_3)$, dont l'aire est notée dS, et le vecteur contrainte $\vec{c}(M, \vec{n})$.

Un peu de géométrie nous montre que :

$$\begin{aligned} 2dS\vec{n} &= \vec{AB} \wedge \vec{AC} = (\vec{MB} - \vec{MA}) \wedge (\vec{MC} - \vec{MA}) \\ &= \vec{MB} \wedge \vec{MC} + \vec{MA} \wedge \vec{MB} + \vec{MC} \wedge \vec{MA} \\ &= 2dS_1\vec{e}_1 + 2dS_2\vec{e}_2 + 2dS_3\vec{e}_3 \end{aligned} \tag{A.1}$$

En projetant successivement sur $(\vec{e}_1, \vec{e}_2, \vec{e}_3)$ on obtient :

$$dS_1 = dSn_1, \quad dS_2 = dSn_2, \quad dS_3 = dSn_3 \tag{A.2}$$

Le principe d'action, réaction, dit que $\vec{c}(M, -\vec{n}) = -\vec{c}(M, \vec{n})$. En utilisant ce principe, on peut exprimer l'équilibre des efforts extérieurs qui s'appliquent sur le tétraèdre par :

$$\begin{aligned} &dS\vec{c}(M, \vec{n}) + dS_1\vec{c}_1(M, -\vec{e}_1) + dS_2\vec{c}_2(M, -\vec{e}_2) + dS_3\vec{c}_3(M, -\vec{e}_3) = 0 \\ &dS\vec{c}(M, \vec{n}) + n_1dS\vec{c}_1(M, -\vec{e}_1) + n_2dS\vec{c}_2(M, -\vec{e}_2) + n_3dS\vec{c}_3(M, -\vec{e}_3) = 0 \\ &\vec{c}(M, \vec{n}) = n_1\vec{c}_1(M, \vec{e}_1) + n_2\vec{c}_2(M, \vec{e}_2) + n_3\vec{c}_3(M, \vec{e}_3) \end{aligned} \tag{A.3}$$

En écrivant ce résultat sous forme matricielle, on obtient finalement l'équation (2.15).

Annexe B

Algorithme du CG

Cette annexe présente la façon dont sont construites les directions (et les coefficients associés) dans l'algorithme du gradient conjugué. On rappelle que l'on cherche à minimiser la forme quadratique suivante :

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \mathbf{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} + \boldsymbol{c} \tag{B.1}$$

On va alors chercher un ensemble de directions mutuellement conjuguées pour “descendre” le long de la forme quadratique. Pour déterminer les vecteurs $\boldsymbol{p}^{(k)}$ (qui correspondent aux directions), on procède par récurrence. Le premier vecteur est obtenu à partir d'une solution initiale $\boldsymbol{x}^{(0)}$ arbitraire, les autres vecteurs sont déterminés au cours du processus itératif.

On pose $\boldsymbol{r}^{(k)}$ le résidu d'une itération k, défini comme suit :

$$\boldsymbol{r}^{(k)} = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k)} \tag{B.2}$$

On remarque que $\boldsymbol{r}^{(k)} = -f'(\boldsymbol{x}^{(k)})$ qui est l'opposé du gradient évalué en $\boldsymbol{x}^{(k)}$. Cela correspond à la direction où $f(\boldsymbol{x})$ diminue le plus rapidement. Ainsi, on pose $\boldsymbol{p}_1 = -f'(\boldsymbol{x}^{(0)})$, et si $\boldsymbol{x}^{(0)} = 0$, on obtient :

$$\boldsymbol{p}_0 = -f'(\boldsymbol{x}^{(0)}) = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(0)} = \boldsymbol{b} \tag{B.3}$$

Nous allons ensuite calculer la prochaine direction, de telle sorte qu'elle soit conjuguée aux directions précédentes. Pour cela, on utilise l'algorithme de *Gram-Schmidt*, qui permet de déterminer un vecteur orthogonal à un ensemble d'autres vecteurs, en le projetant successivement dans chacune des directions. La direction suivante s'écrit alors :

$$\boldsymbol{p}^{(k+1)} = \boldsymbol{r}^{(k)} - \sum_{i\leq k} \frac{\boldsymbol{p}^{(i)T}\mathbf{A}\boldsymbol{r}^{(k)}}{\boldsymbol{p}^{(i)T}\mathbf{A}\boldsymbol{p}^{(i)}}\boldsymbol{p}^{(i)} \tag{B.4}$$

La position suivante est donc obtenue en calculant $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_{k+1}\boldsymbol{p}_{k+1}$, avec α_{k+1} qui

peut être exprimé directement en fonction du résidu et de la direction suivante :

$$\alpha^{(k+1)} = \frac{\boldsymbol{p}^{(k+1)^T} \boldsymbol{r}^{(k)}}{\boldsymbol{p}^{(k+1)^T} \mathbf{A} \boldsymbol{p}^{(k+1)}} \tag{B.5}$$

Bien que très simple, cette version du gradient conjugué nécessite le stockage de toutes les directions précédentes ainsi qu'un grand nombre de multiplications de matrice–vecteur pour résoudre l'équation (B.4). En pratique, on utilise une version légèrement modifiée qui ne nécessite le stockage que des deux derniers résidus et d'une seule multiplication matrice–vecteur par itération Shewchuk (1994).

Annexe

Réduction parallèle

Une réduction parallèle consiste à sommer toutes les valeurs d'un vecteur. Dès lors il faut différencier deux cas : si les threads sont répartis dans plusieurs blocs CUDA, ou si les threads doivent être synchronisés au sein du même bloc.

Évidemment, la première solution est plus coûteuse, pour la paralléliser on utilise deux kernels. Pour le premier kernel, on affecte plusieurs blocs CUDA à différentes portions du vecteur. Pour cela, si dim est la dimension du vecteur et $BSIZE$ une constante définie pour répartir les threads dans les différents blocs, on lancera $BSIZE$ blocs contenant chacun un nombre arbitraire de thread (généralement 256 ou 512). Avec ce découpage, chaque thread va traiter séquentiellement $(dim/(BSIZE * 512))$ valeurs (si on a choisi d'utiliser 512 threads). Après une synchronisation locale, on pourra réaliser une réduction parallèle à l'intérieur de chaque bloc afin d'obtenir une somme partielle dans chacun des blocs. Pour cela on utilise la seconde variante de la réduction parallèle, où tous les threads sont contenus dans un seul bloc. Enfin, le premier thread de chaque bloc écrit le résultat de l'accumulation de son bloc dans un vecteur de taille $BSIZE$, en utilisant l'indice du bloc. Pour terminer on utilise une synchronisation globale, pour s'assurer que tous les blocs ont écrit leur résultat. Cela se traduit par le lancement du second kernel qui ne contient que $BSIZE$ threads, et un seul bloc. ce kernel correspond alors à la réduction parallèle des $BSIZE$ accumulations partielles des blocs précédents, dans laquelle tous les threads appartiennent au même bloc.

Nous présentons maintenant la seconde variante de la réduction parallèle, dans laquelle tous les threads appartiennent au même bloc. Cette variante de l'algorithme est beaucoup plus souvent utilisé au sein des kernels, et il est important de l'optimiser. De plus, cela représente un excellent exemple d'optimisations spécifiques au GPU (voir le listing C.1). Le principe consiste à diviser successivement la taille du tableau en deux parties égales, les threads situés à gauche de la division accumulent les valeurs situées à droite. Entre chaque division, on utilise des synchronisations pour garantir l'exactitude du résultat, et on a alors besoin de $log(n)$ synchronisations. Cependant, il est nécessaire d'utiliser des synchronisations si et seulement si les threads réalisant les accumulations sont répartis dans plusieurs warps. Ainsi, si le tableau restant à une taille inférieure à 32, les synchronisations

sont inutiles car les threads qui vont réaliser les calculs restant seront automatiquement synchronisés. Enfin, on utilise des macros qui dépendent de $BSIZE$, pour réaliser le minimum d'opérations et de synchronisations. Enfin, une fois ce code exécuté, le premier thread possède la valeur de la somme de tous les éléments du tableau.

Listing C.1 – Réduction parallèle

```
int tx = Threads.x;
__shared__ float stemp[BSIZE];
volatile float * temp = stemp;

#if BSIZE > 256
  __syncthreads();
  if (tx<256) temp[tx] += temp[tx+256];
#endif
#if BSIZE > 128
  __syncthreads();
  if (tx<128) temp[tx] += temp[tx+128];
#endif
#if BSIZE > 64
  __syncthreads();
  if (tx<64) temp[tx] += temp[tx+64];
#endif
#if BSIZE > 32
  __syncthreads();
  if (tx<32) temp[tx] += temp[tx+32];
#endif
#if BSIZE > 16
  if (tx<16) temp[tx] += temp[tx+16];
#endif
#if BSIZE > 8
  if (tx<8) temp[tx] += temp[tx+8];
#endif
#if BSIZE > 4
  if (tx<4) temp[tx] += temp[tx+4];
#endif
#if BSIZE > 2
  if (tx<2) temp[tx] += temp[tx+2];
#endif
#if BSIZE > 1
  if (tx<1) temp[tx] += temp[tx+1];
#endif
```

Implémentation du scan parallèle

Un scan parallèle consiste à additionner successivement chaque valeur d'un tableau, avec la valeur de la case précédente. Par exemple sur le tableau suivant :

1	0	3	1	2	0	0	3	2	5	2	0	1

On obtiendra le tableau suivant après le scan :

0	1	1	4	5	7	7	7	10	12	17	19	19	20

On utilise plusieurs fois ce calcul dans le rasterizer pour déterminer l'adresse d'écriture de chaque thread dans les différents kernels. Par exemple, si on considère que le premier

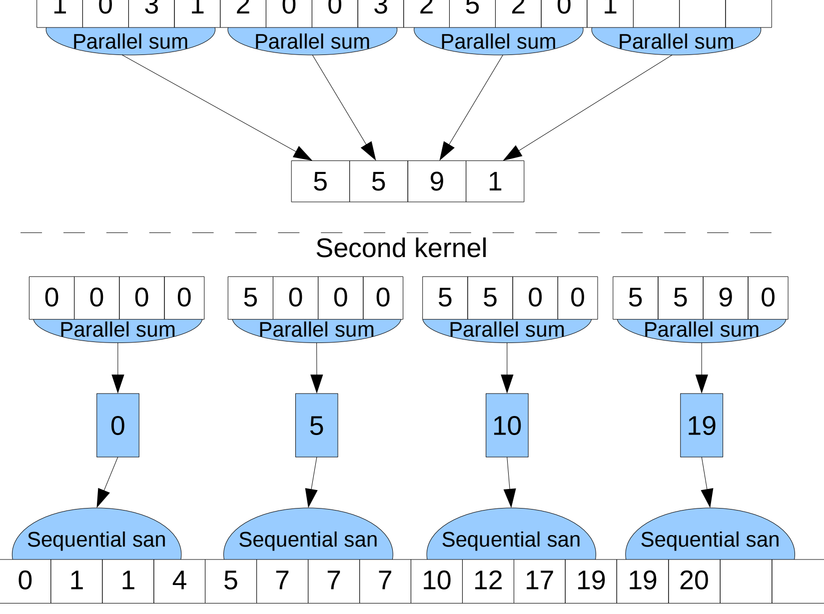


Fig. D.1 – Scan parallèle avec une synchronisation globale

tableau contient le nombre de collisions sous chaque pixel, le second tableau donne pour chaque thread le décalage à effectuer pour écrire les données de chaque thread. Sur cet exemple, si *ptr* est l'adresse du tableau de sortie, le thread n° 1 devra écrire à l'adresse $ptr + 1$ alors que le thread n° 5 devra écrire à l'adresse $ptr + 7$. Ainsi, après le calcul du scan, on pourra traiter toutes les collisions en parallèle, en possédant une adresse unique pour chaque thread.

Évidemment ce calcul est difficile à paralléliser puisqu'il présente de nombreuses dépendances. Pour cela, nous procédons en deux temps (voir figure D.1). Un premier kernel (exécuté par un nombre fixe de blocs CUDA) calcule la somme des contributions d'un sous ensemble du tableau. Ce calcul est exactement le même que celui réalisé par le premier kernel d'une réduction parallèle (voir annexe C). Ensuite, un second kernel réalise le scan en relisant le tableau précédent, en en utilisant les synchronisations locales pour garantir le résultat.

Annexe E

Liste des symboles

Symboles	Signification
Φ	Champ de déplacement
Ψ	Fonction de test
η	Fonction de forme
ψ	Contraintes unilatérales
Υ	Relation barycentrique
$\mathbb{F}$	Gradient de déformation
$\boldsymbol{\varepsilon_c}, \boldsymbol{\varepsilon_b}$	Tenseur de déformation de Cauchy-Green (gauche et droit)
$\boldsymbol{\varepsilon_g}, \boldsymbol{\varepsilon_{g'}}$	Tenseur de déformation de Green-Lagrange, et sa linéarisation
$\boldsymbol{\sigma_c}$	Tenseur de contraintes de Cauchy
$\boldsymbol{\sigma_p}, \boldsymbol{\sigma_s}$	Premier et deuxième tenseur de contraintes de Piola-Kirchhoff
$\otimes, \overline{\otimes}, \overline{\overline{\otimes}}$	Produit tensoriel réduit une, deux et trois fois
$\mathbf{M}, \mathbf{B}, \mathbf{K}$	Matrices de masse, amortissement et rigidité
$\mathbf{A}$	Matrice du système assemblé
$\mathbf{C}$	Matrice de compliance
$\mathbf{P}$	Matrice de Préconditionnement
$\mathcal{P}$	Matrice de permutation
$\mathbf{L}, \mathbf{U}$	Matrices triangulaires inférieures et supérieures
$\mathbf{D}$	Matrice diagonale
$\mathbf{W}$	Opérateur de Delasus
$\mathbf{R}$	Matrice de rotations
$\mathbf{G}$	Matrice de globalisation
$\mathbf{J}$	Matrice de mapping
$\mathbf{E}$	Linéarisation de la relation contrainte - déformation
$\mathbf{F}$	Linéarisation de la relation déformation - déplacement

Symboles	Signification
$\boldsymbol{a}, \boldsymbol{v}, \boldsymbol{p}$	Vecteurs d'accélération, vitesse et position
$\boldsymbol{u}$	Vecteur de déplacement
$\boldsymbol{f}$	Vecteur de force
$\boldsymbol{b}$	Vecteur solution du système linéaire
$\boldsymbol{x}$	Vecteur inconnu du système linéaire
$\boldsymbol{\delta}$	Vecteur des interpénétrations
$\boldsymbol{\lambda}$	Vecteur des multiplicateurs de Lagrange
$\boldsymbol{\rho}$	Vecteur des multiplicateurs de Lagrange exprimés en volume
$\vec{\boldsymbol{c}}$	Vecteur de contrainte
$\vec{\boldsymbol{n}}$	Vecteur normal
$\vec{\boldsymbol{t}}_1, \vec{\boldsymbol{t}}_2, \vec{\boldsymbol{t}}_n$	Vecteurs tangents
ρ	Masse volumique
h	Pas de temps
p	Taille des pixels
$\mathcal{A}$	Aire associé aux pixels
E, ν	Module de Young et coefficient de Poisson
λ, μ	Coefficients de Lamé
μ	Coefficient de frottement

Index

B

Bounding-volume Hierarchies . . . 93

C

Cellule . . . 104
Champ De Déplacement . . . 18
Coefficient De Poisson . . . 25
Coefficients De Lamé . . . 25
Compliance Warping . . . 131
Compressed Column Storage . . . 57
Compressed Row Sparse . . . 57
Contraintes Unilatérales . . . 96

D

Détection Continue . . . 92
Détection De Proximité . . . 101
Détection Discrète . . . 92
Degrés De Liberté . . . 29
Descente . . . 54
Description Eulérienne . . . 18
Description Lagrangienne . . . 17
Diffusion Parallèles . . . 68

E

Euler Explicite . . . 37
Event-driven . . . 92

F

Factorisation Numérique . . . 56
Factorisation Symbolique . . . 56
Fonctions De Forme . . . 29
Fonctions De Mappings . . . 34
Formulation Faible . . . 27

G

Gradient De Déformation . . . 19
Gram-Schmidt . . . 203
Green Ostrogradski . . . 27

J

Jacobiennes Des Contacts . . . 99

K

Kernels . . . 46

L

Layer Depth Images 95
Layers 95
Lower Solve 159

M

Mass Lumping 31
Matrice De Permutation 55
Méthode Des Éléments Finis 27
Modèle Co-rotationnel 43
Module De Young 25
Mono-volume 115
Mouvement Contraint 131
Mouvement Libre 130
Multi-volumes 119
Multiprocesseur 45

O

Opérateur De Delasus 130

P

Pivot De Gauss 54
Problème De Complémentarité Linéaire 100

R

Réduction Parallèle 68
Radix Sort 106
Rastérisation 96
Rayleigh Damping 32
Rayleigh Mass 32
Remontée 54
Remplissage 55
Round-robin 138

S

Scan 106
Schéma D'Euler Implicite 38
Shape Matching 42
Sherman-Morrison-Woodbury 150
Stiffness Warping 42
Synchronisation Globale 48
Synchronisation Locale 47

T

Tenseur De Cauchy-Green Droit 20
Tenseur De Contraintes 23
Tenseur De Contraintes De Cauchy 23
Tenseur De Déformation 19
Tenseur De Green-Lagrange 20
Tenseur de Piola–Kirchhoff 23, 24
Thread-safe 155
Threads 11
Time-stepping 93

U

Upper Solve 159

V

Vecteur Contrainte 22

W

Warp 45