Détection de Collision pour Environnements Large Échelle : Modèle Unifié et Adaptatif pour Architectures Multi-coeur et Multi-GPU

Quentin Avril

Thèse

THÈSE INSA Rennes
sous le sceau de l'Université Européenne de Bretagne
pour obtenir le grade de
DOCTEUR DE L'INSA DE RENNES
Spécialité : Informatique

présentée par
Quentin Avril
ÉCOLE DOCTORALE : MATISSE
LABORATOIRE : INSA-IRISA – UMR6074

Détection de Collision pour Environnements Large Échelle : Modèle Unifié et Adaptatif pour Architectures Multi-cœur et Multi-GPU.

Thèse soutenue le 16 Septembre 2011
devant le jury composé de :

François Bodin
Professeur à l'Université de Rennes 1 / *Président*

Laurent Grisoni
Professeur à l'Université de Lille 1 (PolyTech'Lille) / *Rapporteur*

Bruno Raffin
Chargé de Recherches HDR à l'INRIA Grenoble / *Rapporteur*

Guillaume Moreau
Professeur à l'École Centrale de Nantes / *Examinateur*

Valérie Gouranton
Maître de Conférences à l'INSA de Rennes / *Encadrante*

Bruno Arnaldi
Professeur à l'INSA de Rennes / *Directeur de thèse*

Remerciements

Il me serait impossible de remercier tous ceux qui, de près ou de loin, ont contribué à l'élaboration de ce manuscrit. Si je m'y risquais, j'omettrais à coup sûr des personnes indispensables, des discussions constructives ainsi que des rencontres déterminantes ayant occupées une place essentielle tout au long de ces trois années de recherche. Vous tous qui m'avez apporté votre aide et votre soutien, je sais qui vous êtes et ce que vous avez fait pour moi. Et pour tout ça je tiens à vous remercier, car sans vous ce manuscrit de thèse n'aurait, sans nul doute, jamais pu voir le jour.

Je dois, tout de même, des remerciements particuliers à : Valérie, mon encadrante, pour son aide, ses conseils, son enseignement, nos innombrables discussions et son précieux soutien durant ces trois années ; Bruno, mon directeur, pour son inspiration sur notre sujet, son soutien et son enseignement. A eux deux, j'adresse mes plus sincères et chaleureux remerciements pour m'avoir fait confiance, pour m'avoir donné la chance d'effectuer ce doctorat et pour m'avoir tant appris. Merci également à la région Bretagne d'avoir financé en intégralité ce projet (Financement ARED - Projet *GriRV* N^o4295).

Merci à tous mes "co-buraux" et aux membres de mon équipe (*Bunraku* puis *VR4i*) pour l'aide, l'attention accordée ainsi que pour l'excellente ambiance. Merci également aux membres de mon jury pour leurs remarques et critiques constructives sur mes travaux, mon manuscrit et ma soutenance. Je remercie aussi ma famille, belle-famille et amis pour leur soutien et leur présence à ma soutenance.

Et enfin, je remercie Claire, ma moitié, pour sa présence, sa grande compréhension et son amour.

Table des matières

Introduction

Il est aujourd'hui possible, grâce à la simulation 3D, de reproduire la plupart des phénomènes et situations qui nous entourent : la chute et la fracture de plusieurs milliers d'objets, les déformations d'un tissu, l'onde de choc d'une explosion, l'écoulement de l'eau dans une canalisation ou bien encore le souffle du vent sur les feuilles d'un arbre. Ces simulations incluant de nombreux éléments sont de plus en plus complexes et engendrent, par conséquent, une multitude de calculs à effectuer. La tendance des diverses communautés manipulant la simulation 3D allant clairement vers un accroissement continu et permanent de la taille et de la complexité des environnements 3D, maintenir un niveau de temps interactif devient, dans ce cas, impossible à garantir. La réalisation de crash tests virtuels pour l'automobile en temps interactif, l'évaluation de la résistance d'un bâtiment soumis à de très fortes contraintes ou pire encore, la simulation, toujours en temps interactif, de tremblements de terre sur des villes entières sont des exemples de ce qu'il est impossible de réaliser à l'heure actuelle. Une solution simple et naïve serait de dire que ces simulations deviendront réalisables lorsque les architectures machines le permettront. Car si l'on en croit la loi de Moore, un processeur sera deux fois plus puissant d'ici deux ans, il suffit donc d'attendre. Or, depuis quelques années, les architectures machines subissent un profond bouleversement dû au fait que des limites physiques (perte d'énergie, chaleur et intégration sur puce) deviennent de plus en plus difficiles à repousser. C'est le cas des processeurs CPU où les spécialistes ne peuvent en accroitre la puissance que très difficilement. La tendance est donc désormais à la duplication du nombre de cœurs plutôt qu'à l'accroissement de la fréquence des processeurs. Les cartes graphiques sont également sujettes à une impressionnante évolution. Passées d'un statut de simple périphérique d'affichage graphique à celui de supercalculateur générique et dépassant très largement la puissance de calcul d'un processeur CPU, elles jouissent désormais d'une attention toute particulière de la communauté de la simulation physique. Il est désormais primordial de tenir compte de ces nouvelles architectures afin de continuer à améliorer les performances des simulations physiques, et plus précisément des algorithmes de détection de collision, afin de tendre vers des simulations de réalité virtuelle large échelle en temps interactif.

Nous présentons, dans la suite de cette introduction, une définition rapide de la détection de collision ainsi que les différents domaines d'utilisation. Nous nous attachons à développer un ou plusieurs exemples permettant d'illustrer les limites de performance de la détection de collision au sein de ces domaines. Nous décrivons ensuite la problématique principale de ce manuscrit suivie de la description de son organisation.

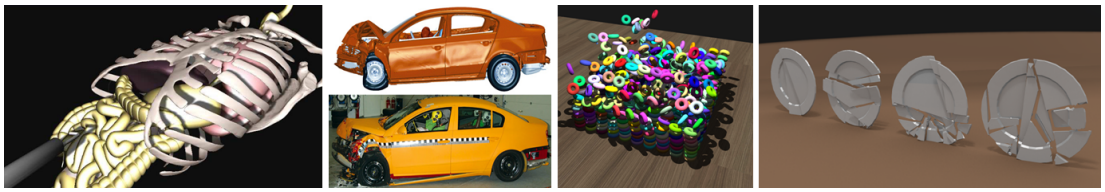


FIGURE 1 – Exemples de simulations physiques impliquant la détection de collision - De gauche à droite : SOFA : un Framework open source pour la simulation médicale [ACF+07, FBAF08] - Performance virtuelle pour les professionnels des crashs tests et Sécurité (PAM-CRASH 2G - www.esi-group.com) - Simulation n-body large échelle [AGA10b] - Simulation temps réel de fracture en utilisant l'analyse modale [GMD11].

Domaines d'utilisation

La détection de collision est un domaine très large traitant un problème, en apparence, simple : détecter si au moins deux éléments géométriques sont, ou vont être, en collision au sein d'un environnement virtuel. Elle est un aspect fondamental de beaucoup d'applications, incluant les jeux-vidéo, les simulations basées physique, l'animation, la robotique, les applications haptiques, le prototypage virtuel industriel et toutes autres simulations virtuelles mécaniques. Nous présentons, par la suite, ses différentes utilisations et problématiques.

Jeux-vidéo - Dans les jeux-vidéo, la recherche à tout prix de performance, conduit à des algorithmes de détection de collision relativement primitifs et inexacts mais tout à fait crédibles. Elle empêche les joueurs de pouvoir passer à travers le sol et les murs et est en charge des requêtes de visibilité déterminant si une interaction est possible entre deux protagonistes [Ber01]. Étant perpétuellement en quête de réalisme, les jeux-vidéo offrent désormais la possibilité à l'utilisateur d'interagir avec la quasi-totalité de l'environnement 3D. Cette interaction massive est la principale cause du goulet d'étranglement calculatoire de la physique au sein des jeux-vidéo [RCE05].

Simulations physiques - La détection de collision est également utilisée dans les simulations physiques 3D où l'on peut chercher à reproduire tout phénomène physique coûteux voire impossible à reproduire dans la réalité. La réalisation, par exemple, de crashs tests virtuels pour l'automobile ou de simulateur de tremblements de terre sont des exemples où la reproduction réelle de ces phénomènes est soit coûteuse soit impossible. La quantité de données à traiter et de calculs à effectuer pour détecter les collisions au sein de ce type de simulation est colossale et pose clairement un problème quant à une potentielle interaction temps-réel. La Figure 1 présente différents exemples applicatifs de la simulation physique 3D.

Animation - Au sein du vaste domaine de l'animation, la détection de collision contribue à la production de mouvements réalistes d'objets plus ou moins complexes comme la simulation d'objets déformables (ex : vêtements [SSIF09]). La simulation

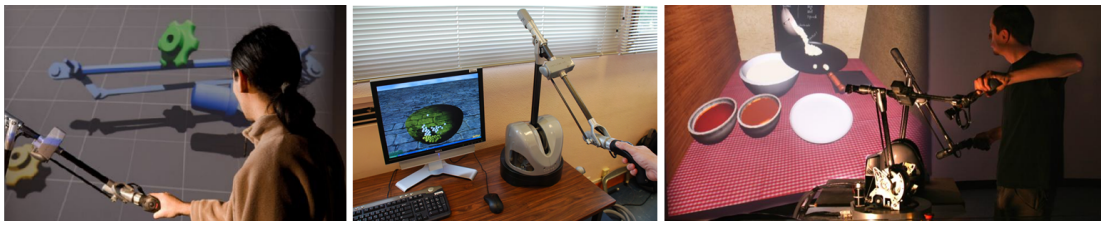


FIGURE 2 – Différents exemples d'interfaces haptiques - De gauche à droite : Rendu haptique spatialisé : Fournir des renseignements sur la position de l'impact en utilisant des vibrations [SLAA09] - Traitement générique des interactions haptiques pour l'assemblage d'objets issus de CAO [Tch10] - Six degrés de liberté d'interaction haptique avec des fluides [CMHL11]

d'humains au sein de foule virtuelle fait également appel à la détection de collision afin que les agents s'évitent entre eux et planifient leur chemin au sein d'environnements contraints. Simuler plusieurs millions d'humains au comportement réaliste au sein d'environnements virtuels complexes est toujours un véritable challenge pour cette communauté.

Robotique - Dans les applications dédiées à la robotique, la détection de collision est utile pour la planification de chemin, elle fournit une aide aux robots leur permettant de se tenir à l'écart des obstacles [Lin93]. Chaque robot doit percevoir son environnement et agir de façon indépendante, sans communication avec d'autres robots ou de coordination centralisée. Faire naviguer de nombreux robots ensemble au sein d'environnements complexes et très contraints est très coûteux en termes de temps de calcul et est une des raisons poussant à améliorer les performances de la détection de collision.

Haptique - Les applications haptiques permettent à l'utilisateur de manipuler des objets virtuels via le sens du toucher grâce à un ou plusieurs dispositifs mécaniques [LBCC01] (cf. Figure 2)). On peut ainsi faire ressentir à l'utilisateur les forces provenant du monde virtuel. Les applications de chirurgie virtuelle (cf. Figure 1), les applications d'aide à l'assemblage d'objets usinés ou le ressenti des différents états de la matière sont des exemples applicatifs de l'utilisation des technologies haptiques. La détection de collision tient une place cruciale au sein de ces applications et il n'est généralement pas admis que les objets (bras mécanique virtuel + objets) s'interpénètrent. Cette contrainte très forte implique l'utilisation d'algorithmes de détection de collision possédant une fréquence de calcul très élevée ($\geq 1000Hz$). Actuellement, cette haute fréquence nécessaire aux applications haptiques, rend la manipulation de plusieurs milliers d'objets complexes impossible à garantir avec un calcul en moins d'une milliseconde.

Problématique

La majorité des applications citées précédemment, telles que la planification de chemin en robotique, l'animation, les jeux-vidéo et les simulations physiques 3D nécessitent d'avoir des réponses en temps interactif. En effet, ces applications nécessitent qu'un très grand nombre de requêtes soient lancées et résolues à une fréquence bien précise. Le travail présenté dans ce manuscrit de thèse s'inscrit dans les domaines de la réalité virtuelle et de l'architecture machine. Plus précisément, il se focalise sur les domaines de la détection de collision en simulation physique et le calcul haute performance. En effet, malgré les nombreux travaux de la littérature ayant contribué à l'amélioration de la détection de collision, la combinatoire de ces algorithmes demeure toujours très problématique pour les environnements large échelle. La détection de collision fait partie des goulets d'étranglements majeurs de l'interaction au sein d'environnements virtuels [Pen90]. Ce qui s'explique par la complexité croissante des scènes virtuelles due à un accroissement du niveau de détail des objets et au nombre très important et croissant d'éléments simulés. Actuellement, les algorithmes de détection de collision les plus utilisés possèdent des complexités quasi-linéaires voire linéaires ce qui, parallèlement à l'accroissement de la complexité des objets simulés, montre que l'apport possible au sein de ce domaine ne peut plus être uniquement algorithmique mais aussi architectural (matériel et logiciel). Il est donc désormais indispensable de reprendre le problème plus en profondeur en s'attachant à proposer des modèles de correspondance génériques adaptés au traitement de larges scènes virtuelles. En fonction de la nature de la simulation, il faut également tenir compte de différents paramètres tels que le type d'objets (rigides, déformables, fluides, etc.), le nombre d'interactions existantes ainsi que la qualité et la vitesse de la réponse que l'on souhaite obtenir de l'application.

Objectifs de la thèse

L'objectif de ce travail a donc été de proposer des solutions performantes pour réduire le goulet d'étranglement calculatoire critique engendré par les algorithmes de détection de collision en établissant un lien générique et adaptatif entre ces algorithmes et l'architecture machine d'exécution. Nous présentons, par la suite, les quatre objectifs poursuivis dans ce mémoire.

1 - Modifier la structure du pipeline pour les architectures parallèles

Le pipeline de détection de collision étant organisé de manière séquentielle depuis sa création, il est primordial de revoir sa structure afin de bénéficier au maximum du parallélisme des nouvelles architectures. Notre objectif a donc été de proposer une nouvelle représentation parallèle de ce pipeline et de mettre en place une version fonctionnelle basée sur cette représentation.

2 - Proposer un modèle de pipeline hybride (multi-CPU/multi-GPU)

La démocratisation des nouvelles architectures multi-cœur et multi-GPU interpelle également la communauté sur la nécessité de mettre en place des algorithmes tenant compte de l'architecture d'exécution pour accroitre leur performance. Nous avons donc travaillé sur la mise en place d'un premier modèle hybride pour la détection de collision.

3 - Concevoir des algorithmes génériques pour multi-cœur et GPU

Le nombre de cœur croissant sur CPU et l'impressionnante puissance de calcul des nouvelles architectures GPU contraignent la communauté de la simulation physique à étudier et proposer des modèles génériques pour faire face à cette évolution. Notre objectif a donc été de mettre en place des modèles algorithmiques permettant aux différentes étapes du processus de détection de collision de s'adapter et de s'exécuter en parallèle sur ces architectures.

4 - Adapter les algorithmes pendant les simulations

Les environnements large échelle de simulation physique peuvent avoir des besoins évolutifs en termes de puissance de calcul. Nous avons donc travaillé sur l'adaptation algorithmique dynamique permettant aux simulations de bénéficier, au moment voulu, de l'algorithme le plus rapide en fonction des différentes conditions de simulation.

Organisation du manuscrit

Ce manuscrit de thèse est composé de sept chapitres et est organisé de la façon suivante :

Les Chapitres 1 et 2 sont les deux chapitres consacrés à notre état de l'art de la littérature. Le Chapitre 1 présente et détaille le domaine de la détection de collision au sein des applications de réalité virtuelle. Nous présentons les différentes approches existantes et familles d'algorithmes ayant été proposées pour les différentes étapes du pipeline de détection de collision. Le Chapitre 2 est un état de l'art sur les solutions parallèles existantes qui, grâce à l'évolution des architectures, permettent d'accroitre les performances des algorithmes. Nous présentons les trois familles existantes : multi-cœur, GPU et hybride.

Le Chapitre 3 présente nos contributions basées sur les architectures multi-cœur. Nous présentons, dans un premier temps, un nouvel algorithme pour la première étape (Broad phase) du pipeline de détection de collision. Un répartiteur parallèle est ensuite présenté pour la seconde étape du pipeline (Narrow phase). Chacune de ces deux contributions est évaluée afin de mettre en avant le gain apporté par le parallélisme.

Nous abordons, dans le Chapitre 4, les solutions basées GPU. Nous présentons deux modèles de Broad phase. Le premier se fonde sur une nouvelle approche de l'algorithme du *Sweep and Prune* [CLMP95] adaptée pour les architectures hautement parallèles du GPU. Le second algorithme est une approche de force brute optimisée, couplée à une technique de subdivision spatiale afin d'être massivement parallélisée et permettant de réduire le temps de calcul au sein de larges environnements virtuels.

Nous avons également travaillé sur la mise en place d'un pipeline hybride que nous détaillons au sein du Chapitre 5. Ce pipeline prend en compte le nombre de cœurs et de GPUs afin d'exécuter en parallèle un pipeline hybride de détection de collision. Il s'exécute sur des architectures multi-cœur et multi-GPU. L'évaluation de ce nouveau pipeline illustre le gain significatif que l'on peut tirer de ces architectures hybrides.

Nous présentons ensuite, dans le Chapitre 6, nos travaux sur la dynamicité, en présentant, tout d'abord la structuration parallèle du pipeline. Nous avons ainsi proposé une modification du pipeline permettant de prendre en compte l'architecture d'exécution. Nous présentons également une nouvelle approche de pipeline parallèle adaptatif permettant l'exécution simultanée des différentes phases en suivant un modèle producteur/consommateur. Nous présentons ensuite nos travaux sur l'adaptation algorithmique dynamique pour la détection de collision. Nous présentons notre approche utilisant des scénarios de précalcul hors-lignes afin d'établir les domaines de performance optimale des algorithmes disponibles. Ces différents domaines permettent de classer les algorithmes selon leur performance et de pouvoir dynamiquement adapter les simulations par l'algorithme le plus approprié au nombre d'objets présents dans l'environnement.

Nous concluons ensuite nos travaux sur les solutions parallèles proposées pour la détection de collision et présentons les différentes perspectives possibles de ces travaux.

Chapitre 1

État de l'art : Détection de Collision

En reprenant la définition donnée par Ericson [Eri05], *"la détection de collision a pour but de déterminer Si, Où et Quand deux objets sont en collision"*. Le *Si* impliquant l'établissement d'une réponse binaire (oui ou non) quant au contact ou non entre deux objets, le *Où* établissant comment les objets entrent en contact et le *Quand* ajoutant une dimension temporelle à la réponse sur le moment où la collision se produit. Son rôle est donc de déterminer des temps d'impact et/ou des points d'impact dans une simulation physique. La réunion des informations sur le *Si*, *Où* et *Quand* est souvent labellisée par la *Détermination du contact*. Les termes *Détection d'intersection* et *Détection d'interférence* sont également utilisés comme synonyme de la détection de collision. Nous présentons, par la suite, une approche préliminaire décrivant les modèles de représentation ainsi que les types d'environnements et d'approches existants (cf. Section 1.1). Nous détaillons ensuite les algorithmes séquentiels permettant une détection exacte de collision entre deux objets virtuels (cf. Section 1.2) suivis par les techniques d'accélération existantes (cf. Section 1.3).

1.1 Approche préliminaire

Afin d'avoir une meilleure représentation du contexte de nos travaux, nous proposons dans un premier temps de présenter les différents modèles de représentations 3D (cf. Section 1.1.1). Nous présentons également les différents types d'environnements utilisés (cf. Section 1.1.2) ainsi que les différentes approches adoptées pour répondre au problème de la détection de collision (cf. Section 1.1.3). Un lecteur déjà familier avec ces notions peut directement se rendre à la Section 1.2.

1.1.1 Modèles de représentation

La Figure 1.1 présente une taxonomie des différentes représentations des modèles 3D. Nous décrivons, par la suite, les différents modèles existants : les modèles polygonaux (cf. Section 1.1.1.1) et les modèles non-polygonaux (cf. Section 1.1.1.2) comme les CSG,

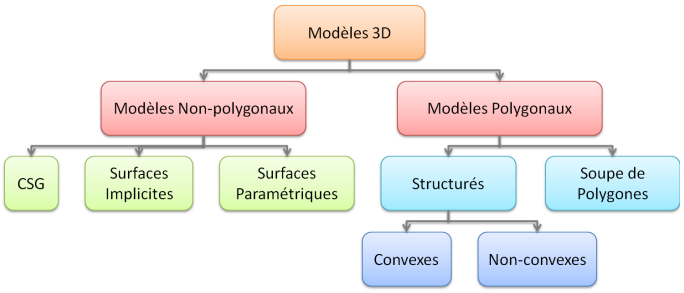


FIGURE 1.1 – Taxonomie des représentations de modèles 3D [LG98].

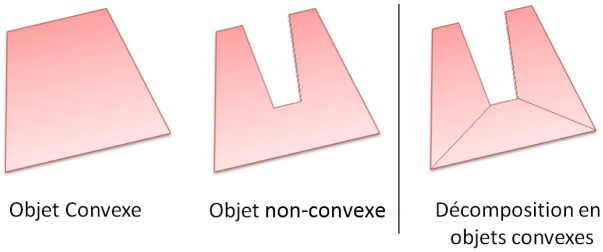


FIGURE 1.2 – Exemple de convexité et de non-convexité d'un objet ainsi qu'une possible décomposition en sous-éléments convexes.

les surfaces implicites et les surfaces paramétriques.

1.1.1.1 Modèles polygonaux

Les modèles polygonaux sont actuellement prédominants dans le domaine de la 3D en temps interactif (jeux-vidéo, simulations...) et ce pour plusieurs raisons. Tout d'abord, leur simplicité mathématique facilite les différents traitements, le matériel permettant un rendu accéléré est également largement disponible, et enfin, la plupart des modèles utilisés en CAD (Computer-Aided Design) (splines ou surfaces implicites (expliquées ci-dessous)) sont convertibles en modèles polygonaux. Nous pouvons noter sur la Figure 1.1 que cette représentation à base de polygones se divise en deux sous-familles : les soupes de polygones, correspondant à des ensembles non structurés de facettes ne possédant aucune information topologique, et son opposé, les ensembles structurés. Il existe deux ensembles de représentation structurés : les objets convexes et les objets non-convexes (cf. Figure 1.2).

1.1.1.2 Modèles non-polygonaux

Il existe trois principaux types de modèles non-polygonaux. Les CSG (Constructive Solid Geometry), plus communément appelés méta-objets, sont des formules mathématiques effectuant des opérations logiques sur des primitives géométriques (cf. Figure 1.3

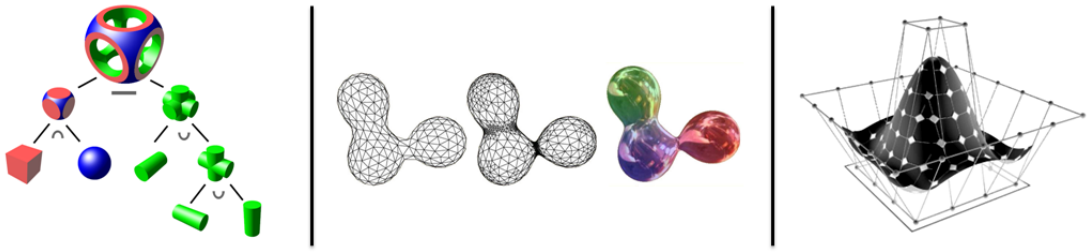


FIGURE 1.3 – **Gauche** : Objet représenté à l'aide de CSG (Constructive Solid Geometry) - **Milieu** : Objet représenté à l'aide de fonctions implicites. - **Droite** : Surface de type NURBS.

gauche). Les primitives utilisées peuvent être des sphères, des cubes, des cylindres ou toutes autres formes simples. Ces primitives sont ensuite combinées à l'aide de fonctions logiques pour former l'objet souhaité. Les surfaces implicites sont définies à l'aide de fonctions implicites (cf. Figure 1.3 milieu) définies de $\mathbb{R}^3 \rightarrow \mathbb{R}$ de telle sorte que l'ensemble des points de la surface répond à $f(x, y, z) = 0$. Les inéquations $f(x, y, z) < 0$ et $f(x, y, z) > 0$ permettent de savoir si un point est à l'intérieur ou à l'extérieur de l'objet. Le lecteur peut se référer aux travaux de Grisoni [Gri05] pour une description approfondie de ce type de surface. Les surfaces paramétriques sont définies à l'aide de fonctions de $\mathbb{R}^2 \rightarrow \mathbb{R}^3$. Contrairement aux surfaces implicites, elles ne permettent pas de représenter un modèle solide complet mais plutôt une description de la frontière d'une surface (cf. Figure 1.3 droite).

1.1.2 Types d'environnement

La détection de collision est utilisée dans beaucoup d'applications diverses, chacune se focalisant sur des aspects de la détection bien précis et différents d'une application à l'autre. Certaines cherchent à faire interagir l'utilisateur à l'aide d'un bras à retour d'effort (cf. Introduction) pour simuler par exemple l'activité d'un technicien sur son poste de travail tandis que d'autres vont chercher à recréer des crashs tests automobiles virtuels (cf. Figure 1) afin d'analyser les déformations de la carrosserie. En fonction du type de simulation que l'on souhaite réaliser, il faut tenir compte de différents paramètres tels que le type d'objets simulés (cf. Section 1.1.2.1), le nombre d'interactions existantes (cf. Section 1.1.2.2), la qualité de la réponse que l'on souhaite obtenir de l'application (cf. Section 1.1.2.3) ainsi que la méthode utilisée (cf. Section 1.1.2.4).

1.1.2.1 Objets rigides et objets déformables

Il est important de savoir si les objets que l'on souhaite simuler sont de nature rigides, déformables (matériaux élastiques, vêtements...) (cf. Figure 1.4) ou fluides. En effet les objets déformables sont plus coûteux en temps calcul que les objets rigides. En effet, la mise à jour permanente de leur structure, pour détecter les auto-collisions, ralentit fortement la fréquence des calculs. Les objets fluides, quant à eux, sont généralement

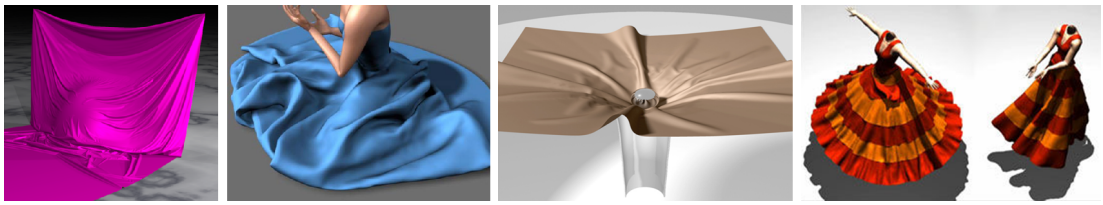


FIGURE 1.4 – Exemple d'objets déformables. De gauche à droite : Selle et al. [SSIF09] - Govindaraju et al. [GKJ+05] - Pabst et al. [PKS10] - Lauterbach et al. [LMM10].

gérés par des modèles basés particules, où chaque particule peut être représentée par un point ou une sphère.

1.1.2.2 2-Body et N-Body

Il est également important de se poser la question du nombre d'objets que l'on souhaite simuler. Si l'on désire simuler un seul et unique objet en mouvement nous sommes dans le cas des méthodes *2-body*, très présentes dans le domaine de l'haptique où l'unique objet en mouvement est l'objet/outil manipulé ou déplacé par l'opérateur. Ces méthodes sont très précises car les calculs ne sont focalisés que sur la détection de contact entre l'objet en mouvement et l'environnement. À l'inverse, les méthodes *n-body* permettent de simuler plusieurs objets en mouvement, ce sont les méthodes les plus répandues car elles offrent une plus grande interactivité. Elles restent, en règle générale, moins précises que les méthodes 2-body. Elles sont également nettement plus coûteuses en temps de calcul car l'approche naïve de force brute possède une complexité quadratique ($O(n^2)$).

1.1.2.3 Types de requêtes

Les informations rendues par une application peuvent être de nature très diverses. Où les objets se touchent-ils ? Quelle est leur intersection ? Quand sont-ils ou vont-ils entrer en collision ? Quelle est la distance minimale entre deux objets ? Toutes ces questions nécessitent une réponse spécifique. Les informations sur la distance inter-objet sont très utiles dans les applications de calcul d'interaction et dans les simulations dynamiques [Lin93]. Les informations sur les intersections d'objets sont utilisées dans les simulations et modélisations basées physique ainsi que pour les calculs de réponse de collision. L'estimation du temps de l'impact permet également de contrôler les pas de temps au sein de la simulation. On peut donc différencier deux grandes familles d'approches : celles dites discrètes détectant les collisions au moment où les objets sont en contact et celles dites continues anticipant sur la collision avant qu'elle ne se produise.

1.1.2.4 Méthodes statiques et dynamiques

Certains objets étant en constante évolution au sein des simulations, il est parfois intéressant d'utiliser la propriété dite de cohérence temporelle. En effet, cette propriété

exploite le fait qu'un objet en mouvement ne subit que très peu de variations entre deux pas de temps. Ceci permet donc d'extraire le fait que si deux objets sont au plus proche en deux points respectifs à un instant t, il existe de grande chance qu'à l'instant $t+1$ ces deux points soient également les deux parties les plus proches de ces deux objets. Afin d'exploiter au mieux cette cohérence temporelle, certains algorithmes nécessitent d'avoir des mouvements d'objets bornés (vitesses ou accélérations). D'autres nécessitent que le mouvement des objets soit exprimé à l'aide de fonction temporelle.

1.1.3 Types d'approches

La détection de collision admet plusieurs formulations de problème dépendant du type de données souhaité en retour ainsi que sur les contraintes imposées sur les données d'entrée. Comme l'indiquent Jiménez et al. [JTT01] le plus simple problème étant de chercher une réponse "oui ou non" est souvent décrit de la sorte : étant donné un ensemble d'objets et une description de leur mouvement selon un certain pas de temps, le but est de déterminer si certaines paires vont entrer en contact. Des versions plus complexes nécessitent de trouver également le temps de l'impact ainsi que les caractéristiques impliquées dans la collision. Quatre principales approches existent pour traiter les différents problèmes de la détection de collision, nous présentons ci-dessous leur principe. Nous les détaillons de manière plus approfondie dans la Section 1.2.

1.1.3.1 Détection d'interférence multiple

La voie la plus simple pour résoudre le problème de la détection de collision est d'échantillonner les trajectoires des objets et d'appliquer de façon répétitive un test d'interférence statique. La qualité du résultat obtenu est fortement liée à la manière d'échantillonner une trajectoire. En d'autres termes un échantillonnage trop large pourrait conduire à manquer une collision tandis qu'un échantillonnage trop fin serait très coûteux en temps de calcul. La manière la plus raisonnable utilisée actuellement est l'application d'un échantillonnage adaptatif, idéalement le prochain pas de temps devrait être celui le plus tôt où une collision serait en mesure de se produire. La quête de cet échantillonnage parfait crée différentes stratégies. La plus rudimentaire consiste à faire le rapprochement entre une borne inférieure de la distance entre deux objets et une borne supérieure de leurs vitesses relatives [Cam85, CK86]. Des stratégies plus élaborées proposent de ne pas seulement prendre en compte la distance inter-objets et les vitesses mais d'ajouter l'utilisation d'informations directionnelles [GH89, GF90]. De telles techniques requièrent le calcul des points les plus proches ainsi que la ligne liant deux objets convexes au pas de temps courant (cf. Figure 1.5).

1.1.3.2 Intersection spatio-temporelle

La représentation la plus générale de la détection de collision est basée sur l'opération d'extrusion, le volume extrudé d'un objet correspondant au volume occupé par l'objet tout au long de sa trajectoire. Une collision entre deux objets ne peut se produire que si, et seulement si, leurs volumes extrudés sont en intersection. L'accroissement de l'utilisation de cette méthode d'extrusion dans le contexte des CSG [Cam90] est dû au

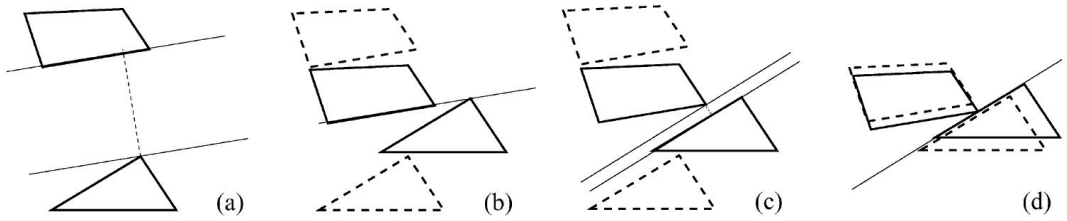


FIGURE 1.5 – Échantillonnage adaptatif temporel. Le point de départ est montré en (a) où les points les plus proches et la ligne joignant les deux objets sont calculés. La projection des objets sur la ligne se fait en (b), qui va être considérée comme le prochain échantillon de temps. Calcul des point les plus proches en (c), le prochain pas de temps, où les polygones sont effectivement en collision, est calculé en (d)

fait que ce type d'opération est distributive en ce qui concerne l'union, l'intersection et toutes autres opérations différentielles. Cette propriété assure le fait qu'un objet et son volume extrudé peuvent être tous deux représentés par les mêmes combinaisons booléennes de primitives géométriques et d'extrusions de ces primitives. Mais malgré le côté assez intéressant de ce type d'approche, le coût de son implémentation demeure assez élevé.

1.1.3.3 Interférence par volume balayé

Le volume contenant tous les points occupés par un objet en mouvement durant un certain laps de temps est appelé "volume balayé". Si tous les volumes balayés par les objets de la scène ne sont pas en intersection, aucune collision ne peut se produire durant le pas de temps défini. C'est une condition suffisante pour la non-collision mais deux volumes peuvent être en intersection sans que les objets ne soient en collision. Afin d'éviter ce type de problème, le balayage des volumes doit être effectué selon le mouvement relatif d'un objet par rapport à un autre. Dans ce cas, un des deux objets est considéré comme fixe et seul le volume balayé par le mouvement relatif de l'autre objet est calculé. Si les volumes sont en intersection, il y a effectivement collision. La génération du volume balayé est aussi une opération assez coûteuse, c'est pourquoi plusieurs travaux portant sur ce type d'approche utilisent l'approximation convexe du volume balayé. Dans ce cas, lorsque les volumes principaux sont en intersection, un échantillonnage de la trajectoire est effectué ainsi qu'une nouvelle approximation du volume balayé de chaque morceau [KVLM04, HFL07, Cam85].

Différentes alternatives de simplification [Her86] ont été proposées, restreignant les formes et les trajectoires à des formes très simples. Ce qui, implicitement, crée un volume balayé grâce aux volumes balayés des primitives de la représentation bornée [Boy79].

1.1.3.4 Paramétrisation de la trajectoire

L'instant même d'une collision peut être déterminé grâce à l'expression des trajectoires des objets sous forme de fonctions temporelles. Prenons un exemple simple,

considérons un point subissant un mouvement rectiligne et nous désirons détecter s'il est en intersection ou non avec un triangle fixe dans l'espace. Nous posons l'équation vectorielle paramétrique suivante :

$$p + (p' - p)t = p_0 + (p_1 - p_0)u + (p_2 - p_0)v$$

Ici p et p' sont les positions initiales et finales du point et les p_i définissent les points du triangle. L'équation est ensuite résolue pour les variables u, v et t où u et v représentent les variables paramétriques du plan défini par le triangle tandis que t est la variable temporelle valant 0 au début et 1 à la fin. Si $0 <= t <= 1$ et $u >= 0$ et $v >= 0$ et $u + v <= 1$ alors le point est en intersection avec le triangle durant le pas de temps [MW88]. Cette équation vectorielle représente trois équations scalaires à trois inconnues pouvant être réduites à une seule équation polynomiale en t. Les conditions d'intersection pour des polyèdres, suivant des trajectoires plus complexes, peuvent être mises en place de la même manière, à la seule différence que le degré du polynôme s'accroit. Lorsque des rotations sont présentes dans les trajectoires des objets, des fonctions trigonométriques sont forcément incluses dans l'expression finale du mouvement. Ces fonctions trigonométriques ne peuvent être réduites en fonctions polynomiales à variable simple qu'avec l'aide du changement de variable. Comme décrit précédemment selon la complexité de la trajectoire le degré du polynôme résultant peut être très élevé. Les polynômes de degré supérieur ou égal à 5 ne peuvent pas être réduits analytiquement, le fait de déterminer le moment exact de la collision peut donc devenir très coûteux en calcul.

Pour Snyder et al. [SWF+93] le problème de détecter des collisions entre des modèles déformables est considéré comme un problème de minimisation contraint résolu en utilisant les méthodes des intervalles de Newton.

Canny [Can86] utilise une représentation de la rotation à base de quaternions fournissant ainsi des contraintes purement algébriques dans un espace à plus haute dimension. Par simple manipulation les contraintes peuvent être projetées dans un espace à six dimensions sans accroissement de la complexité.

Dans le domaine du graphisme par ordinateur, une condition de collision paramétrée peut aisément être dérivée des représentations surfaciques triangulaires [MW88]. Cette méthode peut également être étendue aux surfaces paramétriques non-rigides [HBZ90].

1.1.4 Synthèse

Cette approche préliminaire montre la portée de la détection de collision. Étant utilisée par de nombreux domaines (jeux-vidéo, applications haptiques, ...) ayant des contraintes diamétralement opposées (temps réel, hors-ligne, précision, ...), elle a généré un vaste ensemble de problématiques et d'approches (2 ou n body, convexes ou non, polygonales ou CSG, intersection spatiale ou temporelle, ...). Il existe un nombre important de familles de méthodes permettant de détecter des collisions entre des entités au sein d'un environnement virtuel. Nous présentons, par la suite, les méthodes de détection exacte de collision.

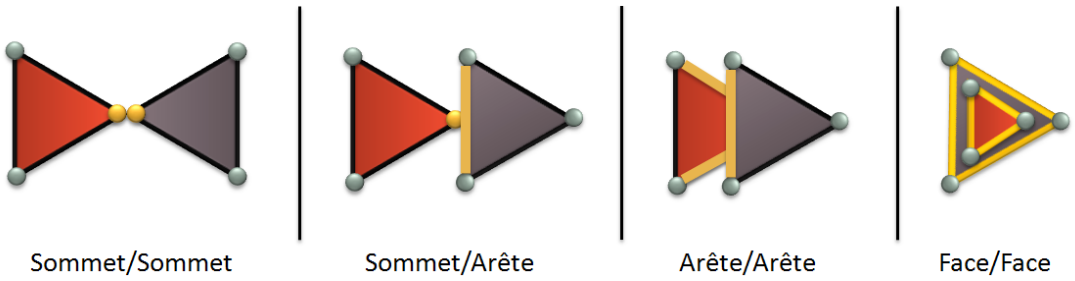


FIGURE 1.6 – Différents tests de primitives (point, arête et polygone). Les primitives d'un objet sont testées avec celles d'un autre objet.

1.2 Détection exacte de collision

L'algorithme en charge de la détection exacte d'une collision dépend du modèle géométrique utilisé pour la représentation des objets virtuels. Dans le cas d'une représentation à l'aide de modèles polygonaux, la détection de collision s'effectue entre les primitives des différents objets (sommets, segments et faces). Le problème majeur est que la précision des modèles polygonaux dépend de la discrétisation, notamment lorsque les objets sont courbés. Lors d'une représentation par CSG, la détection de collision est effectuée avec les formes géométriques élémentaires utilisées pour représenter les objets virtuels (cônes, cylindres, cubes ...) (cf. Figure 1.6). Dans le cas d'une représentation à l'aide de surfaces implicites ou paramétriques, la détection s'effectue à l'aide de fonction d'appartenance, les inéquations $f(x, y, z) < 0$ et $f(x, y, z) > 0$ permettent de savoir si un point est à l'intérieur ou à l'extérieur de l'objet. Nous présentons plusieurs algorithmes permettant de détecter si une collision entre deux objets a lieu. Nous suivons la classification présentée dans l'Action Spécifique du CNRS N^o90 [MKF03] organisant la présentation des algorithmes par types de primitives en critère principal (convexes (cf. Section 1.2.1), non-convexes (cf. Section 1.2.2) puis non-polygonaux (cf. Section 1.2.3)) et par type de détection (détection d'intersection vide, calcul d'interpénétration ou détection de contact).

1.2.1 Détection entre polyèdres convexes

Pour déterminer si deux objets convexes sont en interaction, il existe deux principales approches. La première cherche à déterminer s'il existe un espace entre les deux objets tandis que la seconde cherche à déterminer la profondeur d'interpénétration entre les deux objets.

1.2.1.1 Détection d'intersection vide

Il existe différentes méthodes permettant de savoir si deux polyèdres convexes sont en intersection ou pas, parmi elles, la détection d'intersection vide a pour but de déterminer si l'intersection entre les deux objets est vide ou non. Afin d'y parvenir plusieurs voies

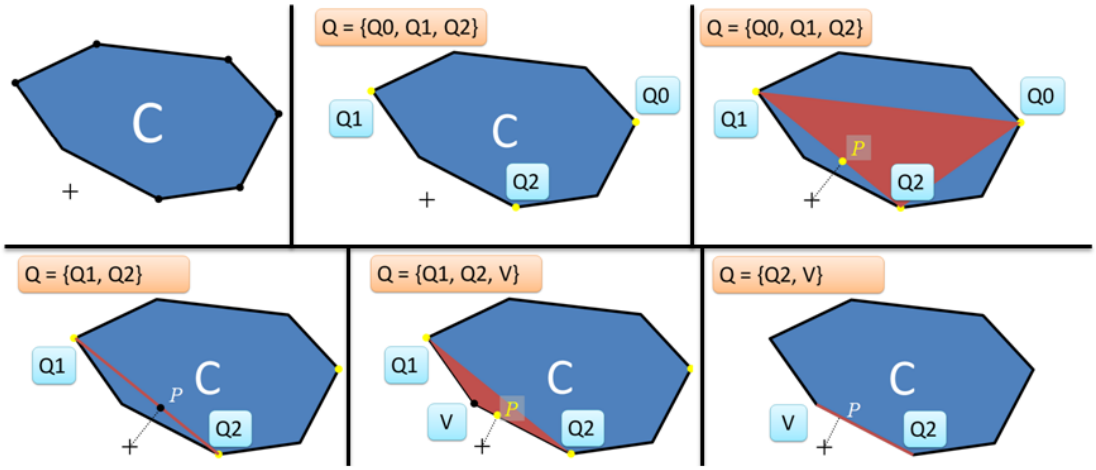


FIGURE 1.7 – L'algorithme du GJK.

sont explorées tels que la recherche d'un plan séparant les objets ou le calcul d'une borne minimale positive de la distance les séparant.

Construction d'un plan séparateur

Il est assez simple d'imaginer que l'on puisse prendre pour acquis le fait que deux objets, étant séparés par un plan, ne sont pas en intersection [Roc70]. Partant de cette conclusion, différents scénarios ont été envisagés quant à la construction de ce plan, certains y ont vu une construction par produit vectoriel [CW96] exploitant en parallèle la cohérence temporelle et géométrique. Cette dite-cohérence assure le fait qu'un plan, séparant deux objets à un pas de temps donné, puisse être réutilisé au pas de temps suivant aux vues des variations infimes de mouvement subis par les objets. D'autres ont imaginé la construction de ce plan par calcul d'une borne minimale de la distance séparant les objets [Mey86, GO94, Ber99]

Calcul de la distance inter-objets

Afin d'évaluer la distance séparant deux objets, trois principales approches ont été proposées. Elles évaluent de façon précise ou approximative cette distance. L'une des plus connues est la méthode dite GJK (Gilbert-Johnson-Keerthi) (cf. Figure 1.7) [GJK88] utilisant la différence de *Minkowski* (cf. Figure 1.8) des polyèdres approchés par des simplexes. Il permet de déterminer la distance Euclidienne minimale et les points les plus proches de deux ensembles convexes de points. Son fonctionnement est le suivant :

Soient A et B deux ensembles convexes de points et x et y deux vecteurs positions correspondant à des paires de points dans A et B. La différence de Minkowski est définie par :

$$A \Theta B = \{x - y : x \in A, y \in B\}.$$

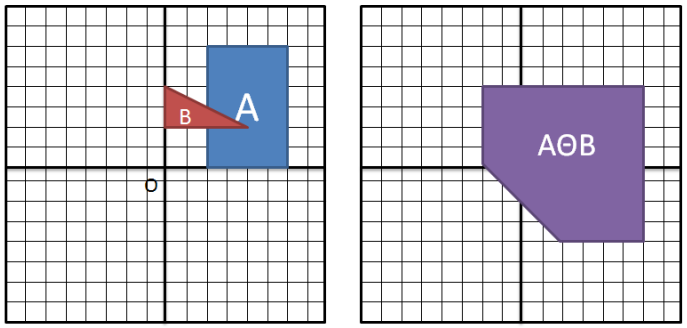


FIGURE 1.8 – La différence de Minkowski entre deux objets convexes $A\Theta B = x - y : x \in A, y \in B$. Les deux objets sont en collision car leur différence de Minkowski contient l'origine.

L'algorithme est basé sur le fait que la distance de séparation entre deux polyèdres convexes est égale à la distance entre la somme de Minkowski et l'origine.

La seconde approche, dite *LC* ou *Voronoï Marching* [LC91] est le premier algorithme de la littérature à se baser sur les primitives des objets. L'espace aux alentours des polyèdres est découpé en région de Voronoï permettant ainsi de détecter les paires de caractéristiques les plus proches entre deux polyèdres (cf. Figure 1.9).

Les polyèdres peuvent également être précalculés [DK90] afin de réduire la complexité de la détection de l'intersection à $O(log(n)log(m))$. Le précalcul met $O(n+m)$ temps à construire une représentation hiérarchique des deux polyèdres possédant respectivement n et m sommets. Le niveau le plus bas au sein de cette hiérarchie (appelé P_1) est le polyèdre original tandis que le niveau le plus haut (appelé P_r) est un tétraèdre. À chaque niveau de la hiérarchie, les sommets du polyèdre original sont enlevés de façon à former un ensemble indépendant (non adjacent) correspondant au précédent niveau hiérarchique. Les segments et faces adjacents correspondants sont mis à jour. Ce type de représentation ne doit être calculé qu'une seule fois et peut être utilisé pour tout type de requête impliquant le polyèdre. L'algorithme décrit dans [DK90] calcule la séparation entre deux polyèdres.

1.2.1.2 Détection par calcul d'interpénétration

Tant que les objets sont disjoints, la distance Euclidienne minimale est la distance la plus utilisée, mais lorsque les objets se chevauchent la distance Euclidienne ne fournit plus d'information utile sur l'intersection ou la pénétration. La distance d'interpénétration entre deux objets est définie comme l'amplitude de la plus petite translation permettant de séparer deux objets [AGHP+00], elle peut être formalisée comme suit :

Soit A et B, deux polytopes (polyèdres bornés) convexes dans $\mathbb{R}^3$ avec respectivement m et n facettes, la profondeur d'interpénétration de A et B, notée $\pi(A,B)$ est définie par :

$$\pi(A,B) = min\{\|t\| \, | int(A+t) \cap B = \oslash, t \in \mathbb{R}^3\}$$

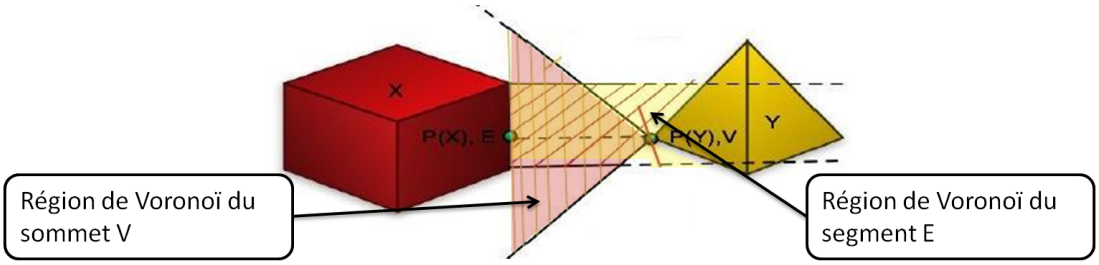


FIGURE 1.9 – Algorithme de Lin-Canny [LC91] : la paire de caractéristique et la paire de points la plus proche (P(X), P(Y)) pour une paire Sommet-Segment.

Le concept de distance négative ou plutôt de profondeur de pénétration entre deux objets se chevauchant a été introduite par Buckley et Leifer [BL85] et Cameron et Culley [CC86]. Pour plus d'informations sur les méthodes de calcul d'interpénétration entre deux objets, le lecteur peut se référer à [Gan07]. Plusieurs algorithmes utilisent la méthode du GJK pour la calculer [Cam97, JSL99] et l'EPA (*Expanded Polytope Algorithm* [Ber01]. Gregory et al. [GME+00] proposent d'étendre la méthode du Voronoï Marching tout en conservant l'utilisation de la cohérence temporelle. Ils l'appliquent à une application de rendu haptique d'interaction mécanique, permettant ainsi d'obtenir une détermination plus juste du contact et une réponse haptique plus homogène qu'avec les méthodes précédentes. Pour une direction donnée, la distance d'interpénétration entre deux polyèdres convexes avec n et m sommets, peut se calculer à l'aide de la hiérarchie de Dobkin et al. [DK90] avec une faible complexité ($O(log(n) * log(m))$) [DHKS93].

Une autre méthode propose d'utiliser des cartes de normales (espace de Gauss) pour trouver la direction de la translation afin de détecter les facettes intervenant dans le calcul [KLM02]. L'algorithme utilise la somme de Minkowski calculée implicitement grâce à la construction d'une carte de Gauss locale (comparable à une carte des normales). Une technique utilise la technique du lancer de rayon pour détecter l'interpénétration [HFR08], un rayon est envoyé à partir de chacune des faces d'un objet dans la direction opposée à la normale (vers l'intérieur de l'objet). Lorsqu'une intersection entre ce rayon et une face externe d'un autre objet est détectée, il y a interpénétration et une force de contact est alors appliquée à cet endroit.

1.2.2 Détection entre polyèdres quelconques

La plupart des algorithmes décrits précédemment ne s'appliquent que sur des polyèdres convexes. Afin de pouvoir établir réellement la présence ou non d'une collision, des algorithmes spécifiques aux objets non-convexes sont nécessaires.

1.2.2.1 Détection d'intersection vide

Afin de déterminer la distance entre deux objets non-convexes, une approche triviale serait de partitionner les objets en un ensemble d'objets convexes et d'utiliser les

algorithmes présentés précédemment. La distance entre les objets serait alors la plus petite distance entre toutes les paires de composantes convexes. Cette implémentation naïve qui examine les paires les unes après les autres possède une complexité de $O(nm)$, où n et m sont le nombre de composantes convexes de chaque objet. Il est cependant intéressant d'étudier la littérature afin d'approfondir la notion de décomposition d'un objet non-convexe en un ensemble de composantes convexes. Ces travaux peuvent porter sur une décomposition en 2-D [For89, LA04] ou en 3D [BD92, CP92, LA07]. Chazelle [Cha81, Cha84] a démontré que le problème du calcul d'une décomposition convexe minimale est discutable et il pense que seules des heuristiques efficaces pour les approximations de calcul doivent être envisagées.

Afin d'éviter ce problème, la plupart des méthodes proposées se basent sur une hiérarchie de volumes englobants permettant ainsi de créer une suite croissante convergeant vers cette distance minimale [EL01, JC98]. Quilan [Qui94] utilise une représentation hiérarchique à base de sphères pour élaguer des portions de son modèle. Sato et al. [SHMA96] proposent une méthode de détection de collision en combinant les arbres à sphères de Quilan et la méthode GJK.

1.2.2.2 Détection de surface en intersection

Les conditions géométriques permettant d'affirmer que deux objets sont en intersection sont que :

- soit un sommet d'un des objets est dans l'autre objet
- soit une arête d'un des objets coupe une face de l'autre objet

Les deux conditions doivent être vérifiées. En fonction du type de réponse souhaitée, l'ordre dans lequel les tests d'intersection sont effectués diffère [Boy79, MW88]. Une autre méthode propose de comparer les triangles des deux objets entre eux [Möl97] pour détecter une éventuelle intersection. Cette dernière a connu des améliorations pouvant être appliquées à des tests d'intersection rectangle-rectangle [TTS06] et permettant, si besoin, de déterminer le segment exact de l'intersection. Cependant, avec ce type d'algorithmes, c'est seulement après avoir comparé tous les triangles des objets que l'on pourra statuer sur la présence ou non d'une collision. La complexité de ces algorithmes est donc quadratique.

1.2.2.3 Calcul d'interpénétration

L'application des algorithmes de calcul d'interpénétration fonctionnant sur des objets convexes n'est pas applicable aux objets non-convexes. Il est certes possible d'utiliser la propriété liant l'éventualité d'une intersection entre deux objets et leur somme de Minkowski. Mais comme l'indiquent Fisher et Lin [FL01], la construction de cette somme en 3D pour des objets non-convexes peut aboutir à une complexité en $O(n^6)$. Dobkin et al. [DHKS93] fournissent une méthode permettant, avec une complexité linéaire, de calculer la profondeur d'interpénétration entre un objet convexe et un non-convexe. Appliquée à deux objets non-convexes, la complexité de cette approche passe en $O(n^2)$. Kim et al. [KOLM02] proposent un algorithme décomposant chaque polyèdre non-convexe en un ensemble de pièces convexes et calculant la profondeur d'interpénétration entre

les pièces convexes se chevauchant. La méthode de calcul est une amélioration de leur méthode précédente [KLM02].

D'autres approches se servent de champ de distance pour calculer la profondeur de l'interpénétration [FL01, EHK+00]. Il y a cependant certaines limitations à ces approches car elles ne calculent que la carte de profondeur interne de chaque objet. Cela n'est donc pas bien adapté pour la manipulation d'objets très fins.

1.2.3 Détection pour modèles non-polygonaux

Comme l'explique Ming C. Lin [LG98], dans le domaine de la modélisation géométrique, le problème de la détermination de l'intersection de surfaces représentées à l'aide de CSG (Constructive Solid Geometry), de surfaces implicites ou paramétriques, a reçu beaucoup d'attention [PG86, Hof89]. Le problème étant de déterminer clairement et précisément tous les éléments de l'intersection entre deux objets. Cela inclut tous les travaux effectués sur l'intersection de courbes et de surfaces [Hof89]. Comme nous les avons décrits précédemment (cf. Section 1.1.1.2), les CSG sont construits à l'aide d'opérateurs ensemblistes opérant sur des formes 3D simples, le problème de l'intersection apparait donc comme relativement plus simple que pour les modèles polygonaux [Til84, SLY99]. De nombreuses approches géométriques offrent la possibilité de détecter si deux courbes se croisent en deux dimensions. Les approches sont classifiées en méthodes par découpage, par réduction d'intervalles ou par subdivision. L'une des méthodes les plus répandues pour détecter si deux ou plusieurs surfaces complexes sont en intersection est celle de l'approximation. On cherche à simplifier les surfaces en approximant des portions de celles-ci par des volumes englobants [HDLM96, HBZ90]. Afin de calculer la distance entre des surfaces, Lin et Manocha [LM95] ont proposé un système permettant de déterminer quels sont les points impliqués dans cette distance minimale. Snyder et al. [SWF+93] utilisent une approche par contraintes pour détecter les collisions et une condition de tangence afin de réduire la dimension de l'espace de recherche. Cette approche permet une détection de collision multipoint entre les surfaces complexes. Van den Bergen [Ber99] a proposé une adaptation de l'algorithme du GJK pour les surfaces paramétriques. Il existe différentes techniques permettant de déterminer si des objets définis par fonctions implicites sont en collision. Lin et Manocha [LM95] proposent de chercher deux points permettant d'estimer la distance minimale entre les deux objets. Savchenko et al [SP95] ont montré que pour deux objets définis par des fonctions implicites F et G, si la borne supérieure de la fonction $min(F, G)$ est supérieure à zéro alors les deux objets sont en interaction.

1.2.4 Synthèse

À travers l'étude des nombreux travaux concernant les méthodes de détection exacte de collision, nous avons pu constater le nombre important d'approches existantes expliquant la difficulté d'effectuer une telle tâche. Les algorithmes présentés permettent de détecter les interactions entre deux objets mais, appliqués sur plusieurs centaines de milliers d'objets, cela forme un goulet d'étranglement calculatoire critique. Il est donc nécessaire de se munir de méthodes d'accélération afin de réduire cette combinatoire.

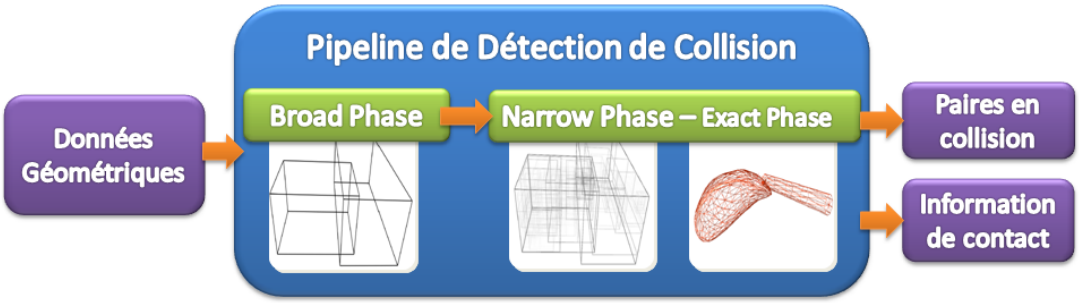


FIGURE 1.10 – Pipeline de détection de collision [Hub93].

1.3 Étapes accélératrices

Les algorithmes présentés précédemment nous ont permis d'appréhender les différentes techniques de détection de collision entre deux objets. Nous avons également montré que certaines de ces méthodes possédaient une complexité quadratique en $O(n^2)$ voire supérieure. Afin de réduire cette complexité, il est indispensable de se munir de techniques d'accélération permettant d'éliminer au plus tôt les paires d'objets n'étant pas en collision.

Tout comme le processus de rendu graphique ou celui de visualisation, la détection de collision est également organisée sous forme d'un pipeline. Malgré le fait que de nombreux algorithmes ont été proposés afin de réduire le nombre de paires à tester [BF79, PS85], Hubbard [Hub93] fut le premier à formaliser cette nouvelle structuration en pipeline, permettant ainsi de visualiser le problème comme une succession séquentielle de filtres de plus en plus précis (cf. Figure 1.10). Le pipeline prend, en entrée, toutes les données géométriques de l'environnement et des objets, et transmet en sortie les informations spatiales et/ou temporelles de contact entre objets étant en collision. La première étape de ce pipeline est appelée *Broad phase* ou phase "large". Elle est en charge de déterminer rapidement et efficacement si deux objets sont ou non en collision. Son objectif est de diminuer le plus largement possible le nombre d'objets à tester lors des étapes suivantes. La seconde étape, appelée *Narrow phase* ou phase "étroite", récupère les objets en forte probabilité d'intersection (fournit par la Broad phase) et exécute un test plus précis afin de déterminer les zones des objets en interaction. Une phase de détection exacte est ensuite appliquée pour connaitre le lieu précis de la collision.

Il existe différentes manières d'agencer le pipeline de détection de collision. La grande majorité d'entre eux est organisée à l'aide de trois phases où deux sont plus lourdes en termes de temps de calcul. Cependant, cela reste tout à fait possible de concevoir un pipeline avec un plus grand nombre de phases. Tout dépend des informations spatio-temporelles dont on a besoin.

1.3.1 *Broad Phase*

Selon Kockara [KHI+07] il existe trois principales familles d'algorithmes offrant la possibilité de connaitre rapidement si deux objets sont ou non en collision : la méthode de force brute (recherche exhaustive), la méthode dite du "*Sweep and Prune*" (SaP) [CLMP95] et celle utilisant le découpage spatial. On peut ajouter les méthodes basées topologie et celles utilisant la cinématique.

1.3.1.1 Méthode de force brute

L'approche de force brute est basée sur la comparaison des volumes englobants des paires d'objets entre eux afin de déterminer s'ils sont ou non en collision. Ce test est très exhaustif en raison de sa complexité quadratique en $O(n^2)$. Il correspond à l'approche la plus simple et la plus coûteuse mais reste cependant très utilisé pour la validation d'algorithmes plus complexes. Dans le cas d'une simulation avec n objets rigides, le nombre de tests à effectuer est de $\frac{(n^2-n)}{2}$ car on ne teste ni un objet avec lui-même (absence d'auto-collision pour les objets rigides) ni une paire dans les deux sens. Les paires du type (x, x) sont donc écartées, ce qui supprime n tests et enfin les paires du type (x, y) et (y, x) ne sont testées qu'une fois, ce qui divise par deux le nombre de tests.

1.3.1.2 Découpage spatial

Les méthodes dites de découpage spatial utilisent une logique simple qui est de dire que deux objets étant éloignés l'un de l'autre ne peuvent vraisemblablement pas être en collision [GASF94]. Le découpage de l'espace peut s'opérer de deux façons. La première est considérée comme absolue car les subdivisions appliquées peuvent être indépendantes de l'environnement [BF79] tandis que la seconde tient compte de la configuration des obstacles immobiles présents au sein de l'environnement. L'espace peut être divisé en grille uniforme [Tur89, Ove92] ou hétérogène [EG07] (cf. Figure 1.11), en structure hiérarchique de type quadtree [FB74] ou octree [BT95, VCC98, SKTK95] ou bien en structure de type *k-d tree* [KMSZ98, FF03]. Samet [Sam90] présente dans son livre une description approfondie des structures de données spatiales et leur utilisation dans les bases de données spatiales. L'environnement peut également fournir les subdivisions comme dans le cas des BSP [Nay92, NAT90, ACT00]. Luque et al. proposent une nouvelle structure baptisée *"Semi-Adjusting BSP-tree"* pour représenter des scènes composées de milliers d'objets [LCF05]. Ils montrent que l'arbre n'a pas besoin d'une restructuration complète même pour les scènes très dynamiques. On peut également utiliser d'autres méthodes telle que la projection des contraintes (translation et rotation) dans un sous-espace [Can86] ou des maillages tétraédriques conformes [HKM95] (topologie de maillage engendrant un volume minimal).

1.3.1.3 Méthode cinématique

L'approche cinématique prend en compte le mouvement des objets. Ainsi si les objets s'éloignent ils ne peuvent pas entrer en collision [LTT91]. Vaněček [Van94] présente une technique de *back-face culling* afin de réduire le nombre de tests à effectuer. Les normales

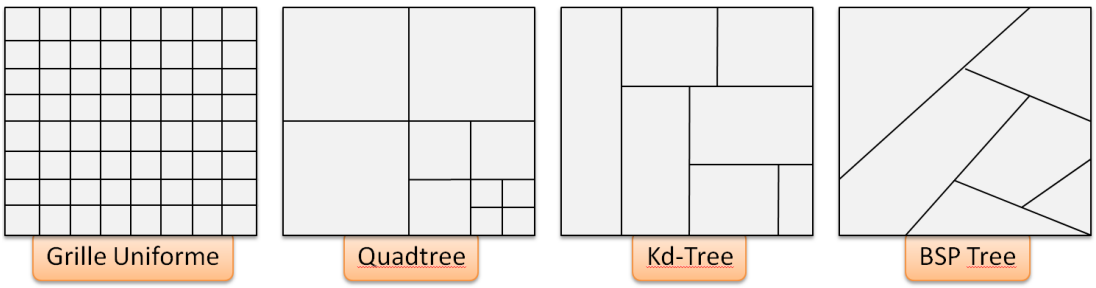


FIGURE 1.11 – Différents types de structures de subdivision spatiale.

des faces d'un objet sont comparées au vecteur vitesse, les faces en sens opposé sont évincées du calcul. Redon et al. [RKC02b] proposent d'améliorer la technique de Vaněček en ajoutant des informations géométriques sur les hiérarchies de volumes englobants. Leur méthode s'applique à n'importe quel type de hiérarchie de volumes englobants et offre un bon compromis entre vitesse et mémoire.

On peut également chercher à prédire l'instant des collisions [CC86, CK86]. Lin et Canny [LC91] proposent une méthode de prédiction des points les plus proches exploitant la cohérence temporelle. Leur méthode est appliquée à la recherche de chemin en robotique. Dworkin et Zelter [DZ93] proposent d'utiliser la mécanique déterministe Newtonienne pour prédire les futures collisions. Kim et al. [KGS97] identifient trois types d'événement afin de détecter la séquence des collisions : collision, entrant et sortant. En traçant tous ces événements dans leur ordre d'apparition, ils sont en mesure de simuler n sphères en mouvement. Basch et al. [BEG+99] proposent de subdiviser l'espace entre les polygones, certifiant de leur non-interaction, et de faire évoluer cette subdivision en fonction du mouvement des polygones.

1.3.1.4 Méthode topologique

Littéralement, la topologie est l'étude du lieu. Les algorithmes utilisant des techniques topologiques s'intéressent donc à la position des objets les uns par rapport aux autres. L'objectif est d'évincer les paires composées d'objets trop éloignés l'un de l'autre. Afin d'éviter un calcul trop coûteux, les algorithmes de Broad phase basés topologie effectuent une approximation des objets par des volumes englobants.

Structures topologiques

Différentes structures ont été proposées afin d'organiser ces volumes englobants, tels que :

- Le R-Tree [Gut85] qui consiste en une structure d'indexation dynamique des localités spatiales.
- Le $R+$-Tree [SRF87] qui est une amélioration de celui de Guttman évitant le chevauchement de volumes entre des nœuds intermédiaires.

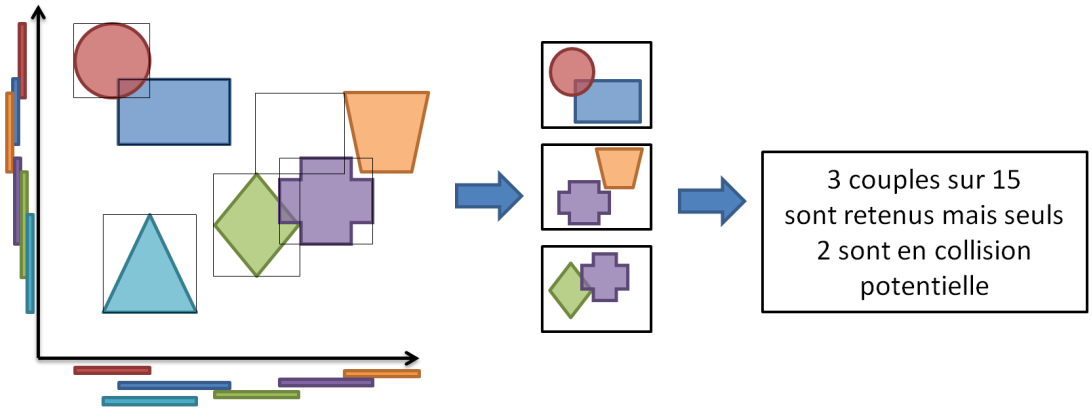


FIGURE 1.12 – Algorithme topologique du Sweep and Prune [CLMP95].

– Le R^*-Tree [BKSS90] qui intègre une optimisation combinée de la surface, de la marge et du chevauchement de chaque rectangle.

Overmars et al. [OvdS96] ont proposé une structure de données différentes des précédentes qui était en mesure de ranger un ensemble de points d'objets complexes disjoints.

***Sweep and Prune* (SaP)**

A l'inverse des algorithmes de subdivision spatiale qui vérifient les collisions parmi les objets d'une même région, l'algorithme du Sweep and Prune (SaP) est une technique alternative permettant de réduire significativement les tests à effectuer. Il est également connu sous le nom de *Sort and Sweep* [Eri05] ayant été appelé de cette façon dans la thèse de Baraff [Bar92] et ses travaux [Bar89, Bar90, Bar91]. D'autres travaux, comme celui de Cohen et al. sur *I-Collide* [CLMP95], se réfèrent à cet algorithme. Il est l'un des algorithmes les plus utilisés dans les algorithmes de Broad phase car il fournit une élimination efficace et rapide des paires d'objets et ne dépend pas de la complexité de ces derniers. Il prend en entrée tous les objets de l'environnement et donne en sortie une liste des paires d'objets en collision potentielle. L'algorithme est divisé en deux parties principales : la première est en charge de la mise à jour de chaque volume englobant des objets actifs (mobiles au sein de l'environnement). La grande majorité du temps, les volumes englobants utilisés sont des volumes alignés sur les axes (AABB - cf. Section 1.3.2.4). La deuxième étape est en charge de détecter les chevauchements entre les volumes englobants. Pour ce faire, une projection des limites supérieures et inférieures des coordonnées de chaque AABB est faite sur les trois axes de l'environnement (x, y et z). Trois listes sont ainsi obtenues comportant les bornes inf et sup de chacun des objets. Les trois listes sont ensuite triées. Il existe deux concepts liés mais différents sur la façon dont le SaP fonctionne en interne :

– **Force brute** : On repart de zéro à chaque fois.
– **Modèle incrémental** : (*persistent* en anglais) On met à jour des structures internes.

La mise en œuvre d'*I-Collide*[CLMP95] utilise une matrice triangulaire pour garder une trace des chevauchements pour chaque dimension plutôt que de réaliser des comparaisons AABB à chaque balayage. Cela nécessite beaucoup de mémoire, mais réduit le travail au cours des balayages. Plus important encore, *I-Collide* utilise un mécanisme de tri local, qui trie chaque borne d'objets dès sa mise à jour. Ceci élimine la nécessité de trier les listes du début à la fin (un tri global), et améliore les performances lorsque seulement quelques objets sont en mouvements.

De nombreuses applications utilisent l'algorithme du *SaP* : SOLID [Ber99, Ber97], Swift++ [EL01] et V-Collide [HLC+97]. Larsson and Akenine-Möller [LAM03, LAM01] l'utilisent pour des objets déformables en élargissant les AABBs pour englober les déformations potentielles. Mais leur méthode réduit la qualité de l'élagage due à la taille plus importante des AABBs. Van den Bergen [Ber97] a également présenté une méthode pour traduire linéairement les objets qui améliore le SaP en calculant les temps d'échange et les reproduit par temps croissant en utilisant une file de priorité. Coming et Staadt [CS06] l'ont adaptée à une approche basée événement pour des environnements dont les objets suivent des chemins simples en précalculant et organisant les événements d'échange. L'algorithme incrémental a été amélioré en utilisant une liste segmentée d'intervalles associée à une technique de subdivision spatiale fournissant une exécution plus rapide pour des environnements virtuels large échelle [TBW09].

Une paire d'objets ayant franchie positivement l'étape de *Broad phase* signifie que les objets la composant sont considérés comme potentiellement en collision. Cette paire est ensuite transmise à la *Narrow phase* afin de subir un test plus précis.

1.3.2 *Narrow-phase*

Le rôle de la Narrow phase est désormais d'appliquer un test plus précis sur une paire d'objets précédemment filtrée par la Broad phase. Son objectif est d'éliminer plus finement deux objets n'étant pas en collision. Pour les paires restantes la probabilité de collision est plus élevée que pour la Broad phase. Les algorithmes présents au sein de la Narrow phase sont plus précis et donc plus coûteux en temps de calcul que ceux de la Broad phase. L'accroissement de la précision engendre un traitement plus important du volume de données géométriques. Selon Kockara et al. [KHI+07] il existe cinq familles d'algorithmes de Narrow phase : Ceux basés caractéristiques (primitives de l'objet), basés simplex, basés espace image, basés volumes et basés hiérarchies de volumes. Or nous avons présenté, dans les parties précédentes, différentes approches permettant une détection exacte de collision utilisant des méthodes présentes ci-dessous. De plus en plus l'étape de Narrow phase est associée à la phase de détection exacte de collision. Cela est dû au fait que beaucoup de domaines utilisant des moteurs physiques se satisfont pleinement de la réponse fournie par cette étape pour calculer les réponses physiques à associer. Il est vrai que la frontière entre Narrow phase et phase exacte est parfois très mince voire inexistante. Les approches, présentées dans la Section 1.2 sur la détection exacte de collision entre deux objets, possèdent donc plusieurs liens avec les approches que nous présentons par la suite.

1.3.2.1 Algorithmes basés caractéristiques

Ce type d'algorithmes travaille directement sur les primitives (segments, faces et sommets) des objets. Moore et Wilhelms [MW88] ont proposé un algorithme testant si des points représentatifs d'un polyèdre sont à l'intérieur d'un autre polyèdre. Les points d'un polygone B sont testés avec ceux du polyèdre A, puis le processus est inversé et les points de A sont testés pour l'inclusion dans B. Ces deux étapes se combinent pour couvrir tous les cas particuliers et donner une réponse fiable. L'algorithme se termine quand un seul point d'interpénétration est trouvé, ce qui est suffisant pour la détection de collision. L'algorithme de Lin-Canny [LC91] peut également être utilisé comme algorithme de Narrow phase (nous l'avons décrit dans la section 1.2.1.1 traitant de la distance inter-objets entre polyèdres convexes). V-Clip [Mir98] est un algorithme qui définit les points les plus proches entre deux polyèdres en termes de caractéristiques les plus proches de la paire de polyèdres. Ehmann et Lin [EL01] ont proposé une amélioration de ce modèle pour les objets non-convexes. Hutter et Fuhrmann [HF07] ont ensuite proposé d'utiliser des volumes englobants sur les caractéristiques pour les modèles déformables. Curtis et al. [CTM08] ont amélioré ce modèle en introduisant la notion de *Triangle représentatif* qui consiste en un triangle géométrique standard auquel on ajoute des informations sur les caractéristiques du maillage.

1.3.2.2 Algorithmes basés simplexe

La plupart des algorithmes basés simplexe peuvent être utilisés en tant que filtre pour la Narrow phase. Un simplexe est une enveloppe convexe d'un ensemble indépendant de points. Le plus célèbre des algorithmes est sans aucun doute celui de Gilbert, Johnson et Keerthi (GJK) [GJK88]. Deux objets convexes sont en collision si et seulement si leur différence de Minkowski contient l'origine. Différentes améliorations ont été proposées sur la base de cet algorithme :

- *Enhancing GJK* [Cam97] utilisant la technique d'optimisation mathématique dite du *Hill Climbing* pour gagner en temps de calcul.
- *ISA-GJK* [Ber99] utilisant le *Data caching* pour accroitre les performances et proposant également une méthode plus rapide pour la construction de l'axe séparateur exploitant la cohérence d'image. Cette cohérence existe entre les images successives d'une animation par rapport à un point de vue précis.
- *MS* [KSM00] couplant l'algorithme GJK à celui de Lin-Canny (LC [LC91]).

1.3.2.3 Algorithmes basés image

Les approches permettant d'utiliser l'image pour détecter des collisions sont basées sur des requêtes d'occlusion. Il est néanmoins possible de séparer les moyens mis en œuvre pour y parvenir. Nous présentons deux méthodes proches mais légèrement différentes : celle basée lancer de rayon et celle basée *Depth Peeling*.

Lancer de rayon

Les algorithmes de détection de collision basés image "rastérisent" les objets et effectuent des tests de chevauchement en 2D ou 2.5D [BW04]. La rastérisation consiste à transformer une image vectorielle en image matricielle représentant la grille de pixels. Des calculs de visibilité peuvent être effectués en utilisant des requêtes d'occlusion et utilisés pour calculer les collisions intra et inter-objets entre plusieurs objets [GRLM03, GLM05a, GLM05b, GKLM07].

Sud et al. [SOM04] utilisent un ensemble de primitives et une distance afin de calculer le champ de distance pour chaque tranche d'une grille spatiale uniforme en "rastérisant" les fonctions de distance des primitives. Ils calculent ainsi les régions de Voronoï de chaque primitive. Ces bornes sont utilisées pour supprimer et fixer les fonctions de distance rendues pour chaque tranche. GI-Collide [BV05] est une approche basée image qui stocke la hiérarchie de sphères englobantes des objets sous forme d'image. Les images les plus hautes illustrent les moyennes des images inférieures tandis que celles du bas montrent comment la taille des sphères est codée. Plus les sphères sont petites, plus les intensités de gris sont renforcées.

Cinder [KP03] est un algorithme exploitant le GPU afin de mettre en œuvre une méthode de lancer de rayon pour détecter les interférences statiques entre des objets solides. Il est basé sur une version 3D du théorème de Jordan [KMW91]. L'algorithme est linéaire par rapport au nombre d'objets et de polygones dans l'environnement. Il ne nécessite aucun prétraitement ou structure de données particulière. Il utilise le *buffer* image pour implémenter un algorithme de lancer de rayon pour chaque pixel. Dans un premier temps, les segments des objets sont écrits dans le buffer. Puis on applique l'algorithme de lancer de rayon qui va, pour chaque rayon, comptabiliser le nombre de polygones traversés. On incrémente lors du franchissement d'une face extérieure et on décrémente pour les faces intérieures. Si le résultat final est différent de zéro, cela signifie que le segment du pixel spécifique est à l'intérieur d'un objet. Si le résultat est pair, cela signifie que le point se situe à l'extérieur de l'objet. Si, à l'inverse, le résultat s'avère impair, le point est à l'intérieur d'un autre objet et il y a collision. L'inconvénient majeur de ce type d'approche est que des collisions proches de se produire ou qui se sont déjà produites ne seront pas détectées. Ce type de scénario peut se produire avec des objets très fins se passant à travers entre deux pas de temps. Ces cas pathologiques sont liés à la précision utilisée.

Hermann et al. [HFR08] présentent une nouvelle approche de détection de collision et de modélisation pour les objets déformables. Leur technique permet des intersections profondes entre les objets tout en allégeant les difficultés de mise à jour du champ de distance. Un rayon est tiré pour chaque sommet de surface dans la direction intérieure de la normale. Dès que le rayon entre en contact avec une face intérieure d'un autre objet, il y a collision. Dans ce cas, une force de contact entre le sommet et le point correspondant est alors créée.

Depth Peeling

Un *Layered Depth Images* (LDI)[Eve01] est une méthode de représentation et de rendu des objets. Similaires à une image en deux dimensions, le LDI est constitué d'une

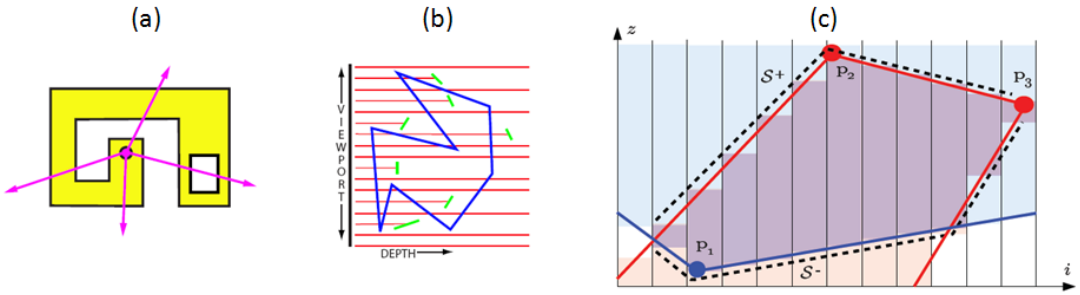


FIGURE 1.13 – Exemples d'algorithmes basés image - de gauche à droite : **(a)** Lancer de rayon à partir d'un point situé dans un polygone [KP03] - **(b)** - Rayons lancés sur les bords d'un objet d'intérêt [KP03] - **(c)** Une coupe 2D selon l'axe z d'un LDI qui illustre le volume d'intersection entre deux objets [AFC+10].

matrice de pixels. Contrairement à une image 2D, un pixel LDI possède des informations de profondeur et il y a plusieurs couches à l'emplacement d'un pixel. L'algorithmique basé LDI a été introduit par Shade et al. [SGwHS98] pour représenter de multiples couches géométriques à partir d'un point de vue. Heidelberger et al. [HTG03, HTG04] ont étendu le modèle de LDI pour construire des modèles géométriques des volumes d'intersection. Morvan et al. [MRS08] ont proposé une amélioration de l'algorithme de détection de collision basé Depth Peeling permettant de réduire au minimum les relectures et passes de rendu, ce qui a conduit à une amélioration des performances. Faure et al. [FBAF08, AFC+10] utilisent une technique basée LDI pour détecter les collisions et créer la réaction physique appropriée. Eiseman et Décoret [ED08] ont proposé une nouvelle méthode de voxelisation de scènes dynamiques grâce à une grille de haute résolution. Leur technique apparait comme plus efficace et plus simple à manipuler que celle du Depth Peeling. L'avantage pour la détection de collision n'est pas réellement prouvé mais semble donner des résultats cohérents quant à la simulation de particules. Govindaraju et al. [GKJ+05] proposent de précalculer une décomposition chromatique d'un maillage en primitives non-adjacentes en utilisant des algorithmes de coloration de graphes. La décomposition chromatique permet de vérifier les collisions entre primitives non-adjacentes en utilisant un algorithme linéaire de suppression. Leur méthode permet d'atteindre un rendement plus élevé de suppression et de réduire considérablement le nombre de faux positifs. Le principal inconvénient des méthodes basées image est la dépendance à la relation taille/précision. Elles sont limitées par rapport à la résolution de l'image car une résolution de 640x480 pixels apporte moins d'informations sur des potentielles collisions qu'une résolution de 1024x768 pixels.

Les propriétés graphiques du GPU peuvent également être utilisées pour détecter les collisions entre surfaces paramétriques déformables [GGK06].

1.3.2.4 Hiérarchies de volumes englobants

Les hiérarchies de volumes englobants (BVH pour *Bounding Volume Hierarchies*) ont prouvé depuis longtemps être les structures de données les plus efficaces pour la dé-

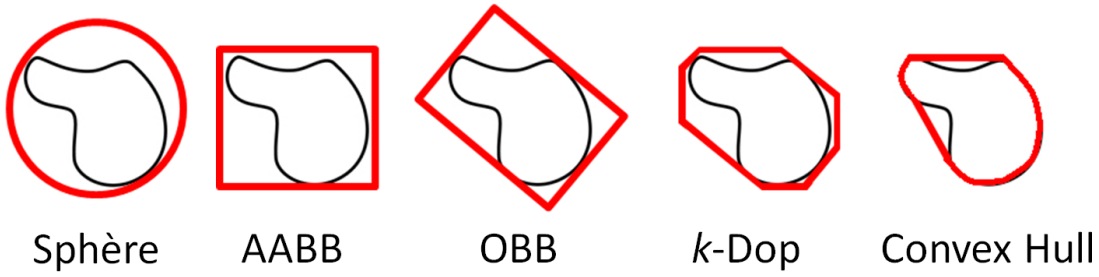


FIGURE 1.14 – Exemples de volumes proposés pour les algorithmes de détection de collision utilisant des hiérarchies de volumes englobants.

tection de collision [TKH+05]. Elles sont généralement précalculées pour chaque objet. La stratégie principale choisie lors de l'utilisation des BVHs est de partitionner récursivement l'ensemble des primitives d'un objet jusqu'à atteindre ces "feuilles". Ces feuilles contiennent la plupart du temps une unique primitive, mais il est possible de trouver des feuilles avec un nombre fixe de primitives. Les primitives utilisées dans cette section sont les éléments formant la géométrie d'un objet. Pour une discussion plus approfondie sur les BVHs, le lecteur pourra se référer aux travaux de Zachmann et al. [ZL03] ou de Teschner et al. [TKH+05].

Types de volumes englobants

L'un des principaux choix à effectuer lors de l'utilisation de BVHs est celui du choix du volume englobant l'objet. On peut voir dans la littérature qu'une multitude de volumes (cf. Figure 1.14) ont été proposés tels que :

- **Sphères** [Hub93, Hub95, Hub96] : il s'agit de la plus petite sphère contenant l'objet définie par un centre et un rayon. Les tests d'intersection entre deux sphères, étant très rapides à effectuer, font des sphères d'excellentes candidates pour détecter les collisions entre objets dynamiques. elles sont fréquemment utilisées [PG94, Din99, BdCRV05].

- **AABBs** (*Axis Aligned Bounding Boxes*) [Zac95, Ber97, LAM01] : ce type de volume englobant est fréquemment utilisé grâce à ces propriétés d'alignement avec les trois axes de l'environnement. Ils sont représentés à l'aide de six valeurs représentant le minimum et le maximum sur les axes x, y et z. Les tests d'intersection entre deux AABBs sont très rapides à calculer car les faces sont parallèles selon l'axe choisi. L'alignement du volume sur les axes entraine parfois une trop grossière approximation comme dans le cas d'un objet à trois branches alignés sur les axes. L'AABB résultat sera un rectangle 3D couvrant un volume beaucoup trop important en comparaison de la forme de l'objet englobé.

- **OBBs** (*Oriented Bounding Boxes*) [GLM96, ES99, RKC02a, RKLM04] : la différence principale avec les AABBs est que les OBB peuvent être arbitrairement

orientés, ce qui peut amener à approximer plus finement un objet. Pour calculer l'intersection entre deux OBBs, Gottshalk et al. [GLM96] l'expliquent comme suit : Deux polytopes convexes disjoints dans l'espace peuvent toujours être séparés par un plan qui est parallèle à une face du polytope soit parallèle à une arête de chaque polytope. Une des conséquences est que deux polytopes convexes sont disjoints si et seulement s'il existe un axe de séparation orthogonal à une face d'un polytope ou orthogonal à une arête de chaque polytope. Une preuve de ce théorème est donnée dans [Got96]. Chaque OBB possède trois faces d'orientation uniques, et 3 segments de directions uniques. Ceci conduit à 15 axes potentiels de séparation à tester (3 faces d'un OBB, 3 faces de l'autre OBB et 9 combinaisons deux à deux des arêtes). Ainsi, le test des 15 axes est un critère suffisant pour déterminer l'état des chevauchements entre deux OBBs. Le test d'intersection entre OBBs est plus lent que pour des AABBs mais il offre une meilleure approximation d'un objet. Cela montre donc qu'il existe un compromis à trouver entre la forme des objets et leurs mouvements et le choix du volume englobant.

- ***k*-DOPs** (*Discrete Oriented Polytope*) [KMSZ98, Zac98, FF03] : il s'agit d'un polytope convexe contenant l'objet (un polygone en 2D ou un polyèdre en 3D). Sa construction consiste à prendre un nombre fixe de plans orientés et à les déplacer jusqu'à ce qu'ils entrent en collision avec l'objet. Le DOP est alors le polytope convexe résultant de l'intersection de l'espace défini par les plans. Il existe différents choix quant à la construction des DOPs qui incluent l'AABB. On peut soit le fabriquer à partir de six plans alignés sur les axes (boîte de sélection en biseau), soit à partir de 10 plans (si biseauté uniquement sur les bords verticaux), soit 18 plans (si biseauté sur tous les bords) ou encore 26 plans (si biseauté sur tous les bords et les coins). Un DOP construit à partir de k plans est appelé un k-DOP.

- ***Convex Hull*** (Enveloppes convexes) [BD92, EL01] : l'enveloppe convexe d'un objet est le plus petit volume convexe qui l'englobe. Si l'objet est l'union d'un ensemble fini de points, son enveloppe convexe est un polytope.

- ***Spherical Shells*** (Coques) [KGL+98] : ces volumes, organisés en *Shell tree*, correspondent à une portion du volume entre deux sphères concentriques. Il faut décomposer les surfaces des objets en courbes de Bézier afin de calculer ces "coques".

- **Rectangular Swept Sphere** (RSS) [LGLM00] : ce volume correspond à l'ensemble des points obtenus en balayant le centre d'une sphère sur un rectangle 3D. On effectue la somme de Minkowski d'une sphère centrée à l'origine avec un rectangle orienté arbitrairement.

D'autres volumes, que nous ne détaillons pas, ont également été proposés tels que les *QuOSPO* [He99], les prismes [BCG+96] ou encore les cylindres [WHG84, HLC+97]. Il existe donc une large palette de volumes englobants utilisés pour approximer plus ou moins finement des objets. Les tests d'intersection entre ces différents volumes sont également plus ou moins rapides à exécuter. On constate donc qu'il existe un réel curseur

Volume Englobant	**Test**
Sphere	$dist(Centre_1, Centre_2)^2 > (R1 + R2)^2$
AABB	$\exists axe \in \{x, y, z\} \mid Max_{AABB}[axe] < Min_{AABB}[axe]$
k-DOP	$\exists k \in \{1, ..., n\} \mid Max_{k-DOP}[k] < Min_{k-DOP}[k]$
OBB	$\exists \vec{V} \in \{15axes\} \mid\mid \vec{T}.\vec{V}\mid > \sum_{i=1}^{3} \mid a_i \vec{A_i}.\vec{V}\mid + \sum_{i=1}^{3} \mid b_i \vec{B_i}.\vec{V}\mid$
Enveloppe Convexe	Algorithme du *GJK*, *LC* ou *DK*

TABLE 1.1 – Tests à effectuer pour détecter une non-collision entre deux objets. Les volumes sont classés de haut en bas selon leur rapidité de calcul.

entre approximation et vitesse quant au choix des volumes. Les propriétés géométriques et physiques des objets permettent généralement d'élaguer ce choix.

Parcours d'arbres

Lors d'un test de collision entre deux objets A et B, la plupart des algorithmes de détection de collision basés volumes englobants utilise la stratégie suivante :

Algorithme 1 Test de collision entre deux objets A et B avec parcours d'arbre

```
Parcours(A,B)
Si A et B ne sont pas en intersection Alors
  Fin
Sinon
  Si A et B sont des feuilles Alors
    Rendre intersection entre les primitives encapsulées par A et B
  Sinon
    Pour Tout Fils de A et Fils de B Faire
      Parcours(A[i],B[i])
    Fin Pour
  Fin Si
Fin Si
```

Pour tester les collisions entre deux objets ou les auto-collisions d'un objet, les BVHs sont traversés de haut en bas et des paires de nœuds de l'arbre sont testées de manière récursive. Si les nœuds qui se chevauchent sont des feuilles de l'arbre alors les primitives encapsulées sont testées pour l'intersection. Si un des nœuds est une feuille tandis que l'autre est un nœud interne, le nœud feuille est testé contre chacun des fils du nœud interne. Si toutefois, les deux nœuds sont des nœuds internes, on cherche à minimiser la probabilité d'intersection le plus rapidement possible. Van den Bergen [Ber97] teste le nœud avec le plus petit volume contre les fils du nœud avec le plus grand volume.

Construction d'arbres

Il existe principalement trois différentes stratégies pour construire les arbres de volumes englobants : *bottom-up*, *top-down* et *insertion*. Lors de la construction d'un BVH,

il est commode d'oublier les primitives d'un objet et de considérer plutôt les BVs comme les atomes. Une autre simplification peut être d'approximer chaque objet par son centre (barycentre[GLM96] ou un centre de boîte englobante) puis ne traiter qu'avec les ensembles de points lors de la construction. Bien évidemment, lorsque les BVs sont finalement calculés pour les nœuds, les primitives réelles doivent être prises en compte. Nous décrivons, par la suite, les trois stratégies de construction existantes :

- ***Bottom-up*** [RL85, VT95] Il existe deux différentes sous-stratégies pour construire un arbre en suivant cette technique.

  La première consiste à prendre un ensemble E de volumes englobants au plus haut niveau, puis pour chaque $e_i \in E$ trouver le plus proche voisin $e'_i \in E$, trier l'ensemble E selon les distances d_i entre chaque couple (e_i, e'_i), combiner ensuite les n premiers nœuds de E sous un père commun. On fait la même chose pour les n suivants éléments et ainsi de suite. On répète ensuite le processus à tous les fils jusqu'à atteindre les primitives de l'objet.

  La deuxième stratégie est un peu moins coûteuse en temps de calcul. L'algorithme commence par calculer les centres c^i de chaque e_i de E, puis trie E selon l'axe x en fonction de c^i_x (la coordonnées x de c^i). L'ensemble E est ensuite découpé en $\sqrt{\frac{n}{k}}$ tranches verticales, toujours en fonction de c^i_x. Chaque carreau est ensuite trié selon c^i_y puis également divisé en $\sqrt{\frac{n}{k}}$ carreaux. Ce qui permet d'obtenir k carreaux. Au final, tous les nœuds dans un carreau sont regroupés sous un père commun et le processus se répète pour un nouvel ensemble E'.

- ***Top-down*** [MKE03] L'idée générale est de commencer avec l'ensemble des BVs élémentaires séparé en k parties et de créer un BVH pour chaque partie de manière récursive. On cherche à diviser récursivement un ensemble de primitives d'un objet jusqu'à ce qu'un critère soit atteint. Le fractionnement est guidé par un critère spécifié par l'utilisateur ou des heuristiques qui donneront des BVHs répondant au critère choisi.

- ***Insertion***[GS87] La construction commence par un arbre vide. Les objets sont ensuite insérés successivement et on cherche le meilleur point d'insertion pour chaque nouveau nœud. Comme tous les nœuds de l'arbre ne peuvent pas être considérés comme des éventuels points d'insertion, la recherche ne doit suivre que quelques chemins. Le choix des sous-arbres d'un nœud est déterminé par la plus petite incrémentation subit par le volume englobant du nœud si le nouveau nœud devait être inséré comme un fils de ce dernier. En règle générale, les algorithmes utilisant cette stratégie ont une complexité en $O(nlog(n))$.

Mise à jour de l'arbre

A la différence des hiérarchies des objets rigides, les hiérarchies d'objets déformables doivent être mises à jour à chaque pas de temps. Il existe deux possibilités : la mise à jour ou la reconstruction. Les techniques de mise à jour sont beaucoup plus rapides

que la reconstruction, mais pour de grandes déformations les BVs sont généralement moins serrés et ont un plus grand volume de chevauchement. Van den Bergen [Ber97] a constaté que mettre à jour la hiérarchie à chaque pas de temps est environ dix fois plus rapide par rapport à une reconstruction complète d'une hiérarchie AABB. En outre, aussi longtemps que la topologie de l'objet est conservée, il n'y a aucune perte de performance significative dans la requête de collision par rapport à la reconstruction.

Le BD-Tree [JP04] est une hiérarchie de sphères pouvant être mise à jour après la déformation d'un objet dans n'importe quel ordre, et à un coût indépendant de la complexité géométrique de l'objet. Coming and Staadt [CS08] ont présenté les *VADOP* (*Velocity-Aligned Discrete Oriented Polytopes*) comme un nouveau modèle accélérateur offrant des mises à jour rapides et particulièrement adaptées pour des applications avec de nombreux objets en mouvement rapide. La sélection des axes qui détermine la forme de leurs volumes de délimitation est basée sur des revêtements sphériques. Kim et al. [TJBDS09] ont présenté les *RACBVHs* (Random-Accessible Compressed Bounding Volume Hierarchies). Ce type de hiérarchie est basé sur une méthode de compression qui préserve la forme originale d'une BVH et comprime séquentiellement les volumes englobants.

1.3.3 Synthèse

Ces deux étapes accélératrices, que sont la Broad et la Narrow phase, permettent donc, grâce à des algorithmiques plus simples et plus rapides, de significativement réduire le temps de calcul. Quatre approches (force brute, spatiale, cinématique et topologique) se partagent donc l'étape de Broad phase. Les approches basées subdivision spatiale et topologie sont, actuellement, celles ayant le plus d'intérêt par la communauté de simulation physique. Leurs rapidités de mise en place et d'exécution ainsi que leur capacité d'élagage en font d'excellentes candidates pour accélérer de manière significative les performances. Malgré cette rapidité de calcul, cette étape reste une étape essentielle et primordiale pour les simulations large échelle. Elle est le fondement même de la qualité et de la rapidité du traitement de la physique au sein de ces environnements. Sur de très larges et complexes environnements, cette étape, exécutée de façon séquentielle, occupe toujours une place importante dans le goulet d'étranglement calculatoire des simulations.

A l'inverse, les approches choisies pour effectuer l'étape de Narrow phase restent encore très nombreuses au sein de la communauté. L'algorithme utilisé est très généralement en lien avec le type de simulation effectué et la qualité de réponse souhaitée (haptique, jeux-vidéo, etc.). Les nombreuses approches existantes sont difficiles à comparer au sein d'environnements large échelle de par leur utilisation particulière et spécifique. Quel que soit l'environnement large et complexe utilisé, l'exigence du temps interactif est toujours compromise par le temps de calcul nécessaire au déroulement de cette étape. Il reste cependant indispensable de les étudier pour déterminer quelles phases sont de bonnes candidates pour une exploitation sur des architectures parallèles. Les deux approches CPU et GPU permettent donc d'envisager l'exécution de cette étape sur des architectures parallèles de deux types : multi-CPU et multi-GPU.

1.4 Conclusion

A travers ce survol de la littérature, nous avons montré que la détection de collision tient une place prépondérante au sein de nombreux domaines applicatifs et que de nombreux travaux ont contribué, pendant plus de trente ans, à en améliorer la qualité et la performance. La structuration de la détection de collision sous forme de pipeline proposée par Hubbard [Hub95] a permis de significativement réduire le goulet d'étranglement calculatoire. La stratégie consistant à faire passer les objets par une succession de filtres (*Broad phase*, *Narrow phase* et *Exact phase*) est désormais l'unique stratégie adoptée par tous les moteurs physiques.

Cependant, quelle que soit la stratégie adoptée (espace image, tests sur primitives, hiérarchies de volumes, GJK, etc.) l'algorithme retenu reste très coûteux en termes de temps de calcul au sein de larges environnements virtuels. Les algorithmes utilisés demeurent toujours incapables de gérer un très large volume de données et de tâches en temps interactif. Ce n'est pas uniquement dû à l'algorithmique mais également au plafond calculatoire des architectures d'exécution. Les approches présentées dans ce chapitre possèdent toutes une exécution séquentielle et ne tirent aucun bénéfice à s'exécuter sur des architectures parallèles. Afin d'accroitre la rapidité et la précision des algorithmes, il est donc primordial de prendre en compte l'architecture d'exécution en proposant des solutions adaptatives et génériques. Nous décrivons et analysons dans le chapitre suivant les solutions parallèles basées CPU et/ou GPU proposées pour améliorer les performances de la détection de collision.

Chapitre 2

État de l'art : Détection de Collision & Solutions Parallèles

Nous présentons dans ce chapitre les solutions parallèles existantes permettant de réduire le goulet d'étranglement des simulations physiques et le passage aux environnements large échelle. Le fait de prendre en compte l'architecture d'exécution pour les algorithmes de détection de collision est une thématique relativement récente. L'augmentation de la vitesse de calcul des précédents algorithmes étant essentiellement basée sur l'évolution des processeurs, la fin de la loi de Moore a fait passer le parallélisme à un statut de nécessité pour l'avenir des simulations physiques. Au sein du domaine de la détection de collision en parallèle sur un seul ordinateur, trois familles principales se distinguent : celle des algorithmes basés CPU (multi-cœur), celle basée GPU et multi-GPU et une nouvelle famille, plus récente, proposant de lier les deux précédentes en offrant des solutions hybrides CPU et GPU. Nous présentons, dans la suite, l'évolution des architectures CPU et GPU (cf. Section 2.1) suivie des méthodes parallèles existantes pour la détection de collision basées CPU (cf. Section 2.2) et basées GPU (cf. Section 2.3). Nous présentons ensuite la famille la plus récente traitant des solutions parallèles hybrides multi-cœur et multi-GPU (cf. Section 2.4).

2.1 Évolution des architectures

Nous présentons dans cette section, l'évolution des processeurs CPU et des cartes graphiques durant ces dernières années. Nous décrivons d'abord l'émergence et la propagation des processeurs multi-cœur (cf. Section 2.1.1), nous retraçons ensuite l'impressionnante évolution des GPU en termes de puissance de calcul et de facilité d'utilisation (cf. Section 2.1.2). Et enfin, nous présentons brièvement l'utilisation du parallélisme en réalité virtuelle (cf. Section 2.1.3).

2.1.1 D'un processeur séquentiel à une architecture multi-cœur

En 1965, Gordon Moore [Moo65] prédisait un dédoublement du nombre de transistors sur un microprocesseur tous les deux ans. Pendant plus de quarante ans, cette

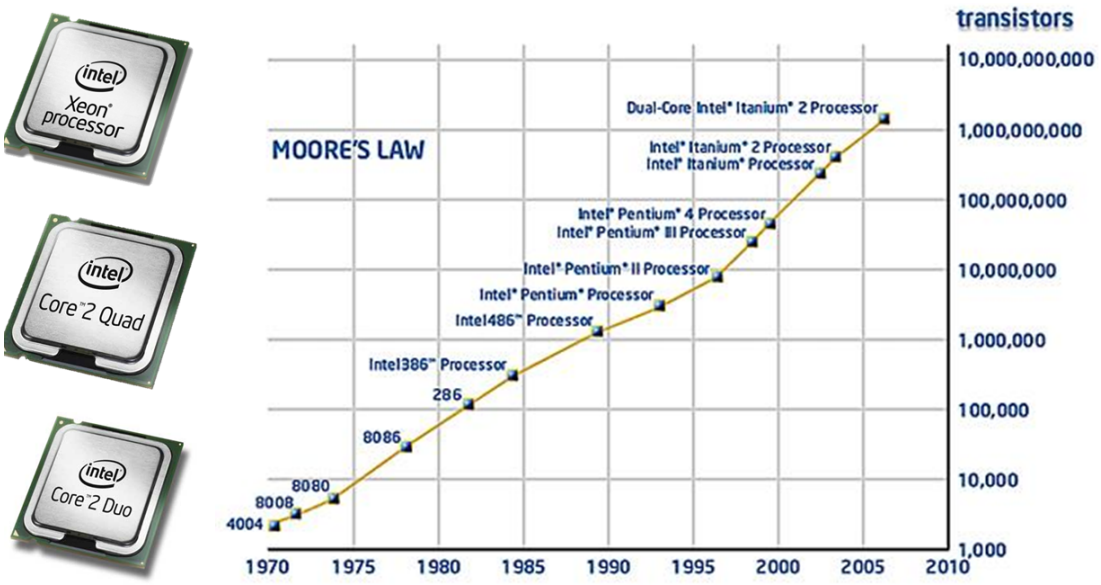


FIGURE 2.1 – 40 ans d'évolution du nombre de transistors sur un microprocesseur de 1970 à 2010 (source : www.intel.com).

conjecture s'est avérée exacte (cf. Figure 2.1) avec un nombre de transistors multiplié par 2 tous les 1,96 ans.

Des limites physiques, telles que la miniaturisation ou le dégagement de chaleur, rendent désormais cette augmentation de la fréquence des processeurs très difficile à effectuer.

2.1.1.1 Évolution Matérielle

La solution, proposée par les spécialistes *hardware*, pour préserver la loi de Moore concernant la puissance de calcul, est de faire davantage de cœurs. Actuellement, la tendance de ces spécialistes est d'accroitre en permanence le nombre de cœurs sur les processeurs. Le premier ordinateur personnel muni d'un processeur multi-cœur est arrivé en 2005 avec AMD [1] atteignant une fréquence de 3,2 GHz pour un processeur à deux cœurs. En 2006, Sun a présenté son nouveau Niagara2 octo-core, où chaque cœur est capable de prendre en charge 8 processus, soit une gestion totale de 64 processus (threads) en parallèle. Intel [2] a présenté en 2008 un 32 cœurs x86 baptisé "Larrabee" [SCS+08] spécialement conçu pour mettre en avant la programmabilité des cœurs. Ce projet a depuis été abandonné suite à différents problèmes de consommation (performance par watt) et également au regard de la complexité logicielle. Il semble désormais que les nouvelles tendances ne soient pas seulement au multi-cœur mais aussi aux many-cœur. La différence principale entre ces types de noyaux tient au nombre de cœurs, le

1. www.amd.com
2. www.intel.com

many-cœur correspondant à un processeur avec significativement plus de cœurs. La notion de démarrage et d'arrêt à la volée dans le cadre des architectures many-cœur est également une différence avec le multi-cœur. S'il est nécessaire d'utiliser n cœurs pour obtenir de meilleures performances, le processeur n'en activera que n. Les architectures many-cœur sont très utiles car il est très rare d'avoir besoin de toute la puissance des cœurs.

Depuis leur apparition les processeurs sont donc en perpétuelle évolution. Mais le virage forcé, pris suite à la grande difficulté d'accroitre la fréquence, conduit désormais vers de nouvelles architectures parallèles. Il force également toute la communauté de l'informatique, tout domaine confondu, à se pencher sur de nouvelles problématiques portant sur l'amélioration des performances sur architectures parallèles. Le but recherché n'étant pas uniquement d'accroitre les performances des applications mais d'y parvenir tout en améliorant leur comportement et leur qualité de réponse.

2.1.1.2 Évolution de la programmabilité

Comme le décrit E. Hermann dans sa thèse [Her10], nous pouvons classer les modèles de programmation en trois différentes sections : le passage de message (*message passing*), le parallélisme de tâches et le parallélisme de données.

Message Passing

Le modèle de passage de messages (ou modèle de Message Passing) est un modèle de communication entre ordinateurs permettant l'envoi de messages simples. Il constitue la couche de base des *middlewares* orientés messages. Il contient un ensemble de processus de communication indépendants, nommés de manière unique, et interagissant à l'aide d'envoi et de réception de messages. Chaque processus possède son propre espace d'adressage, et l'accès aux données des autres processeurs ne peut se faire que par échanges de messages explicites. Cette communication impose le développement d'applications parallèles bien structurées. Les deux systèmes les plus populaires sont *PVM*[Sun90] et *MPI*[For94, GLS99]. Ces deux systèmes portables permettent à des machines d'architectures différentes ou des systèmes d'exploitation différents de communiquer et de collaborer dans une même application.

Programmation parallèle avec mémoire partagée

L'un des avantages majeurs de la programmation parallèle avec mémoire partagée est l'accès aux données. Les données sont toutes disponibles localement et il n'y a nul besoin de transfert réseau pour y accéder. L'accès aux données en mémoire partagée peut se réaliser en transmettant uniquement un pointeur mémoire. L'apparition massive des architectures multi-cœur a accru la popularité de la programmation parallèle avec mémoire partagée. Ce type de programmation apparait comme plus intuitif et plus simple aux personnes non expertes en parallélisme. Parmi la large palette d'environnements de programmation parallèle existant, nous présentons rapidement *OpenMP*[3], considéré

3. http ://www.openmp.org/

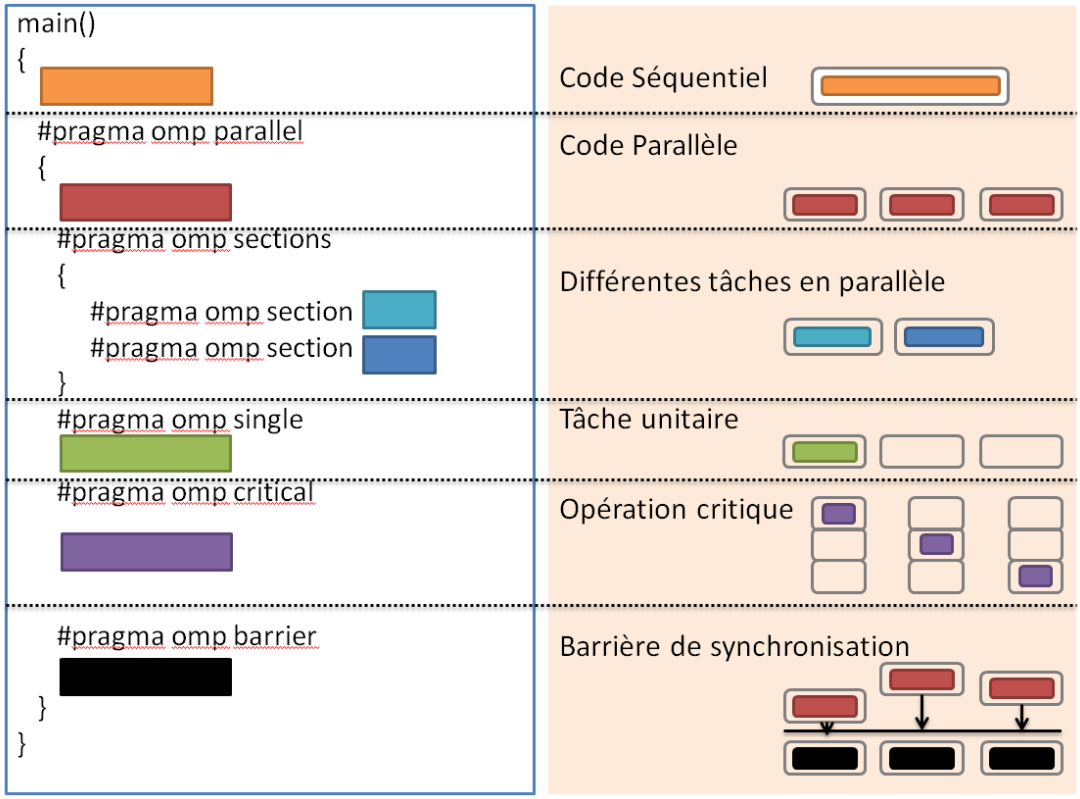


FIGURE 2.2 – Syntaxe et sémantique de la programmation avec OpenMP.

comme le plus populaire et *IntelTBB*[4], standard plus récent et très prometteur pour le parallélisme de tâches.

OpenMP [DM97] - Considéré à ses débuts comme l'alternative portable au parallélisme par passage de messages, OpenMP est un modèle de programmation parallèle multi-tâches dont le mode de communication entre les tâches est implicite. Ce qui signifie que la gestion des communications est à la charge du compilateur. Il se présente sous la forme de directives, de variables d'environnements et d'une bibliothèque logicielle. Il est utilisé sur des machines multiprocesseurs à mémoire partagée. La mise en œuvre d'une parallélisation à deux niveaux (MPI et OpenMP) est tout à fait possible et conseillée sur une grappe de machines indépendantes (nœuds) multiprocesseurs à mémoire partagée. La Figure 2.2 présente la syntaxe et la sémantique de différentes directives utilisées en programmant à l'aide d'OpenMP.

IntelTBB [Rei07] (*Intel Threading Building Blocks*) - Il s'agit d'une bibliothèque logicielle, développée par la société Intel, pour tirer profit des architectures multi-cœur. La bibliothèque d'IntelTBB se compose de structures de données et d'algorithmes per-

4. http ://threadingbuildingblocks.org/

mettant aux développeurs d'éviter certaines complications découlant de l'utilisation de threads natifs. À la différence d'OpenMP, IntelTBB ne nécessite aucun langage ou compilateur particulier. Une technique d'équilibrage de charge basée sur le vol de tâches est utilisée pour redistribuer le travail entre les cœurs.

Il est difficile de comparer ces deux environnements de programmation. Cela dépend essentiellement de la situation du développeur. Ce qui est sûr c'est qu'un développeur n'ayant pas ou peu d'expérience en programmation parallèle, se tournera très probablement vers l'utilisation d'OpenMP ou d'IntelTBB comparativement, par exemple, aux threads natifs de Windows. L'avantage du code OpenMP est qu'il est facile à entretenir. Avec IntelTBB le développeur n'a pas besoin de comprendre le fonctionnement des threads, il suffit de présenter des tâches à IntelTBB et de lui faire confiance pour exécuter l'application avec une meilleure performance.

2.1.2 D'un processeur graphique au GPGPU

Ces dernières années, les cartes graphiques sont passées d'un statut de simple unité de fonctions fixe d'affichage graphique à celui de supercalculateur générique de plus en plus programmable.

2.1.2.1 Évolution matérielle

Contrairement aux processeurs CPU, les GPU (*Graphics Processing Unit*) ont subi une évolution de la puissance de calcul très importante. Cette évolution peut être expliquée par le fait que, d'un côté, le CPU est un processeur généraliste traitant des données ordinaires à forte interdépendance où plusieurs de ses composants sont en charge du contrôle de flux de données. Sa période de latence mémoire est couverte par la mise en cache des données. De l'autre côté, le GPU fournit des microprocesseurs bien adaptés à des calculs hautement parallélisables, il traite avec des données indépendantes de telle sorte qu'il n'a pas besoin de fournir un contrôle accru sur le flux de données, sa période de latence de la mémoire est couverte par le calcul.

Le *General-Purpose Processing on Graphics Processing Unit* (GPGPU) est un domaine de l'informatique visant à effectuer des calculs génériques (sans obligatoirement de lien avec les calculs graphiques) sur les GPUs. L'objectif du GPGPU est de tirer le maximum de profit de l'architecture hautement parallèle des GPUs fournissant une forte puissance de calcul. Il permet de réaliser certains calculs lourds qui sont néanmoins facilement parallélisables et peuvent donc bénéficier d'une architecture pensée pour le calcul parallèle. Owens et al. [OLG+07] présentent, dans un état de l'art sur le GPGPU, les développements logiciels et matériels ayant conduit à l'intérêt de ce domaine. Ils décrivent les modèles mis en œuvre pour mettre en lien le calcul générique aux architectures graphiques.

2.1.2.2 Évolution de la programmabilité

L'utilisation des cartes graphiques, afin de réduire le temps de calcul, n'est pas récente. Au milieu des années 90, certains chercheurs utilisent la capacité de "rasté-

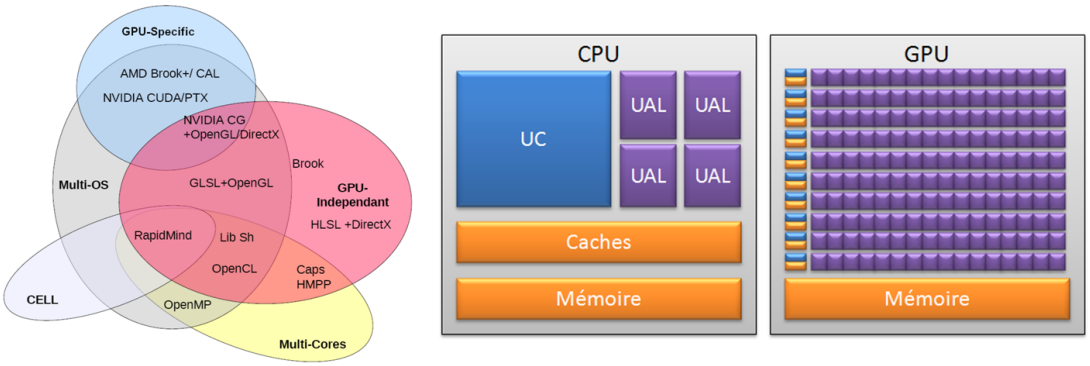


FIGURE 2.3 – **Gauche** : Présentation de différents niveaux de langage utilisés dans le GPGPU. **Droite** : Comparaison très simplifiée de la structure interne d'un CPU et d'un GPU.

risation" du Z-Buffer des cartes graphiques pour accélérer la recherche de chemin ou la création de texture. La fin des années 90 marque également un tournant avec l'apparition du *"Multitexturing"*. Le début des années 2000 est, quant à lui, marqué par l'apparition des techniques d'éclairage de scènes. La révolution apparaît en 2001 avec les *"vertex shaders"* programmable permettant les calculs de matrices sur les cartes graphiques. Les *"pixel shaders"* apparaissent, quant à eux, au cours des années 2002-2003. Depuis 2003, une section Siggraph [5] est exclusivement dédiée aux techniques de calculs génériques sur simple ou multi GPU.

En 2003, *OpenGL* [6] ou *Direct3D* [7] étaient essentiels pour gérer les GPU. *Brook* d'ATI [BFH+04] fut la première extension du langage C à permettre d'utiliser le GPU en tant que co-processeur pour les calculs parallèles. En 2007, *Nvidia* [8] a développé un langage et un logiciel appelé *CUDA* [CUD07] (Compute Unified Device Architecture) permettant d'exploiter la puissance du GPU en utilisant les principes de la programmation parallèle avec les threads. Cette API peut être comparée à une extension du langage C. Son langage d'assemblage est PTX. *ATI/AMD* [9] a également développé son propre langage pour les cartes graphiques *ATI*, appelé *Brook+*. Même si la technologie AMD est aussi efficace que celle développée par Nvidia, *Brook+* est nettement moins utilisé que CUDA. Ceci en raison d'un manque de documentation et à une plus grande difficulté de programmation. La Figure 2.3 présente différents niveaux de langages utilisés dans le GPGPU. Il existe désormais un nouveau standard ouvert et libre permettant de programmer des systèmes parallèles hétérogènes (CPU et GPU) nommé *OpenCL* [SGS10] (Open Computing Language). Il est décrit comme un langage accessible pour accéder à des ressources informatiques de type CPU multi-cœur et GPU. Il est, entre autre, supporté sur les récents produits Intel, AMD/ATI et Nvidia. Il fut initialement pensé

5. www.siggraph.org
6. www.opengl.org
7. www.microsoft.com
8. www.nvidia.com
9. www.amd.com

FIGURE 2.4 – Exemples de middlewares parallèles développés et/ou utilisés pour des applications de réalité virtuelle.

par *Apple* puis conçu et développé par un très large consortium d'industriels (*Khronos Group*[10]). Malgré les grandes similitudes avec CUDA, OpenCL a des objectifs plus larges n'étant pas uniquement destinés aux GPUs.

D'autres approches ont ensuite été proposées afin d'améliorer CUDA tel que *BSGP* (Bulk-Synchronous GPU Programming) [HZG08] permettant de réduire la complexité du code et le nombre de fonctions GPU réduisant ainsi le nombre d'octets alloués pour l'exécution. La taille mémoire utilisée et la fréquence des calculs restent toutefois similaires à celles fournies par CUDA.

Il est à noter qu'une autre approche, appelée *ASTEX*[11], a été développée par l'équipe IRISA-INRIA CAPS et permet le portage des applications existantes vers le GPGPU sans la nécessité de modifier le code source. Les données de l'exécution sont utilisées pour extraire automatiquement des bouts de code existants afin de fournir des informations spéculatives sur les accès aux structures de données. Le partitionnement du code source de l'application est alors exprimé au moyen de directives appelées HMPP [DBB07].

2.1.3 Parallélisme en Réalité Virtuelle

Le principal défi des applications de réalité virtuelle étant l'exigence d'atteindre le temps réel, le besoin en termes de puissance de calcul est constant et permanent. Depuis la fin des années 90, marquée par l'émergence des cartes graphiques haute performance, une voie a été ouverte quant à l'utilisation du parallélisme (multi-GPU, *clusters* de PC, grilles, etc.) pour le calcul haute performance en réalité virtuelle.

Le choix de la stratégie de parallélisation peut être influencé par différents facteurs tels que l'architecture cible, l'application d'origine et même l'effort ou le budget alloué à la parallélisation [Her10]. À un niveau macroscopique, il est très fréquent de trouver différents moteurs (graphiques, physiques ...) exécutés en parallèle à l'aide de méthodes simples basées sur le parallélisme fonctionnel. Les jeux-vidéo sont un exemple de ce type de séparation en différents modules [RCE05]. Ces derniers, en charge du rendu de la physique ou bien de l'IA, communiquent entre eux afin de rendre cohérent l'ensemble du jeu. À un niveau plus fin, comme au sein de la détection de collision, il est possible de paralléliser les opérations sur les objets n'étant pas en interaction en exécutant, en

10. http ://www.khronos.org/
11. http ://www.irisa.fr/caps/projects/Astex/

parallèle, des calculs de collision sur des groupes distincts. En descendant à un niveau microscopique, il est possible, à l'aide d'études approfondies, d'exploiter le parallélisme au niveau de l'objet par l'exécution d'opérations en parallèle. Il est également possible de paralléliser en interne un objet en utilisant le parallélisme de données.

Le parallélisme fonctionnel est l'approche qui consiste en la séparation de la simulation en différents modules. Différentes plateformes pour le développement et l'exécution des programmes modulaires sont présents dans les domaines de l'animation, la simulation et la réalité virtuelle. *OpenMask* [MAC+02, LCAA08], développé par l'IRISA-INRIA est une plateforme offrant un compromis entre le calcul haute performance, l'abstraction et la modularité des programmes de réalité virtuelle de tous types. *FlowVR* [AGL+04, AR06] est un middleware spécialement conçu pour les applications de réalité virtuelle distribuées sur architectures de types grilles ou clusters. Le projet Orocos [12], initialement conçu pour le contrôle robotique [Bru01], permet de créer une composante par moteur/capteur et de lier le tout avec un Framework. L'intérêt étant, par exemple, d'exécuter quatre fenêtres de rendu différentes gérées par quatre threads indépendants communiquant avec un seul thread en charge de la physique. Raffin et al. [RS06] présentent les différentes approches, conçues pour les applications de réalité virtuelle, permettant de bénéficier de la puissance des clusters.

2.1.4 Synthèse

La difficulté croissante d'accroitre la fréquence des processeurs a forcé les spécialistes de l'architecture a proposé de nouvelles configurations afin de continuer à accroitre la puissance de calcul des processeurs. Ces nouvelles architectures multi-cœur offrent une possibilité quant à la réduction du goulet d'étranglement des algorithmes de détection de collision. L'attention toute particulière donnée aux GPUs en a également fait des supercalculateurs apportant une solution supplémentaire pour les simulations physiques. La mise en place de standards et d'environnements de programmation orientés multi-cœur et GPGPU permet de rendre plus accessibles ces nouvelles architectures aux communautés non expertes en parallélisme. Le parallélisme fonctionnel est un domaine connu et étudié au sein de la communauté de la réalité virtuelle mais l'utilisation d'architectures multi-cœur ou multi-GPU est encore très récente. Nous présentons, dans la suite, les méthodes parallèles existantes basées CPU et GPU pour la détection de collision.

2.2 Détection de collision basée CPU

Afin de bénéficier des nouvelles architectures multi-cœur sur les processeurs CPU, plusieurs travaux ont été réalisés proposant des solutions parallèles innovantes et performantes pour la détection de collision. Le parallélisme offert par ces architectures a ouvert la voie à différentes approches tels que la subdivision spatiale parallèle (cf. Section 2.2.1), la résolution parallèle de systèmes linéaires (cf. Section 2.2.2), la construction et le parcours d'arbres en parallèles (cf. Section 2.2.3) ou bien encore les simulations

12. http ://www.orocos.org

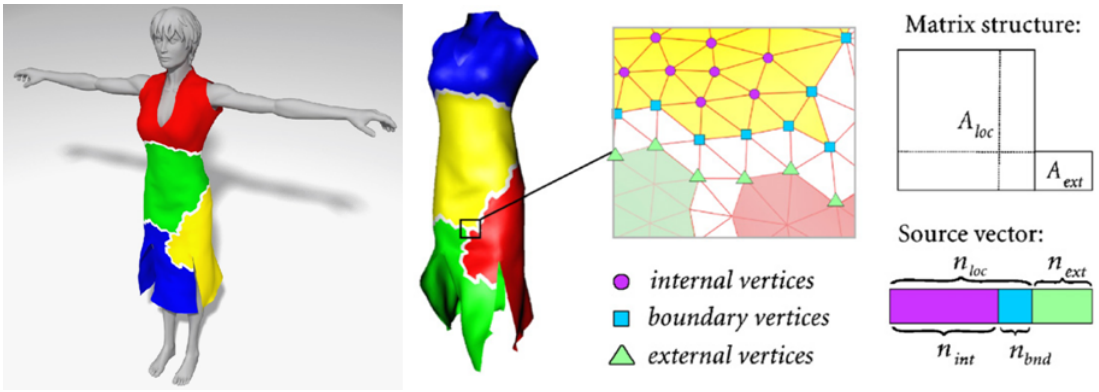


FIGURE 2.5 – Modèle multi-cœur de Thomaszewski [TPB08] - La décomposition d'un maillage en quatre partitions disjointes, indiquées par des couleurs différentes afin d'indiquer la décomposition du problème de la phase d'intégration temporelle.

de particules en parallèle (cf. Section 2.2.4). Les approches parallèles basées CPU sont également utilisées par les simulations physique hors-lignes. Chen et al. (Intel)[CCH+07] utilisent des processeurs *Tera-Scale* pour accélérer les applications hors-lignes de simulation physique.

2.2.1 Subdivision spatiale parallèle

Brown et al. [BAPH00] utilisent une méthode de dichotomie (ou bissection) parallèle récursive de subdivision spatiale pour effectuer les tests de détection de collision dans une application dynamique. Lawlor et Laxmikant [LK02] partent du principe que deux objets ne peuvent pas occuper le même espace à un instant donné. Ils proposent une méthode de détection de collision basée voxel pour les modèles statiques et atteignent 60% de gain en appliquant une méthode générique d'équilibrage de charge avec 1500 processeurs. La limitation majeure de leur approche est que tous les voxels sont de la même taille et de la même orientation, ce qui limite les performances quant à la simulation d'objets très petits ou très grands. Yeh et al. [YFR06] proposent une version parallèle de l'*Open Dynamics Engine* (ODE). L'idée principale est de regrouper des objets en collision en "îles" et de distribuer les îles parmi les threads disponibles. Cette approche fonctionne correctement quand le nombre d'îles indépendantes est suffisant et surtout avec des objets rigides plus simples à résoudre. Appliquée à des objets déformables, cette approche se traduirait par un seul thread et n'apporterait pas de gain de performance.

2.2.2 Résolution parallèle de systèmes linéaires

Jerabkova et al. [JTS+07] analysent l'exécution et l'évolutivité d'une simulation dynamique à l'aide de la méthode des éléments finis sur des architectures *dualcore* et *quadcore*. Des méthodes pour la simulation de petites et grandes déformations sont testées en parallèle avec OpenMP. Thomaszewski et al. [TPB08] proposent un modèle

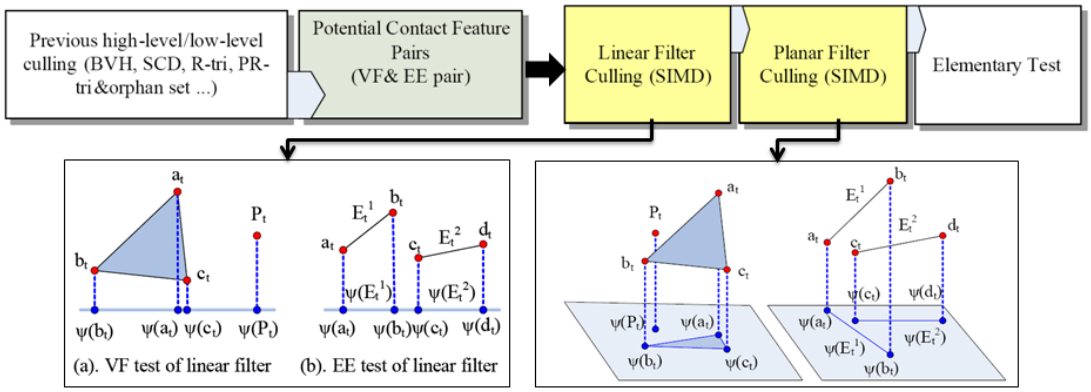


FIGURE 2.6 – Modèle multi-cœur de Tang [TLW11] - Application de filtres linéaires et planaires en parallèles.

parallèle de résolution de systèmes linéaires basé sur une répartition dynamique des tâches pour la détection de collision. L'intégration temporelle est traitée en utilisant une décomposition en problème statique. Le système linéaire résultant est résolu en utilisant un algorithme parallèle basé sur la méthode du gradient conjugué (cf. Figure 2.5). Hermann et al. [HRF09] ont proposé un nouveau modèle de détection de collision permettant d'éviter les synchronisations inutiles dans le graphe des tâches de l'intégration temporelle. Leur approche permet de gérer le contrôle au sein des boucles de détection de collision et permet de réduire significativement les points de synchronisations qu'effectuent OpenMP ou IntelTBB après chaque instruction. Très récemment, Tang et al. [TLW11] ont proposé une nouvelle approche parallèle basée sur la notion de "filtres dans des sous-espaces" (cf. Figure 2.6). Ils proposent une nouvelle étape se situant entre la Broad phase et la Narrow phase où avant chaque test élémentaire, ils appliquent deux filtres successifs. Le premier est un filtre linéaire réduisant les coordonnées des éléments dans $\mathbb{R}$ tandis que le second est un filtre planaire dans $\mathbb{R}^2$. Cette technique permet d'élaguer très largement les tests élémentaires à effectuer et ainsi de réduire significativement le temps de calcul.

2.2.3 Construction et parcours d'arbres parallèles

Nous avons présenté dans la Section 1.3.2.4 les algorithmes séquentiels de construction, de mise à jour et de parcours d'arbres. Nous détaillons dans cette section les approches basées sur ces algorithmes mais modifiées et améliorées afin d'être exécutées en parallèle. En 2001, Wan Huagen et al. [WHQ01] présentent une approche hiérarchique qui adopte un schéma adaptatif de subdivision spatiale pour construire une représentation hybride de délimitation des polyèdres non-convexes. Ils l'utilisent pour parcourir la hiérarchie de volumes englobants en parallèle sur une architecture multi-processeur. Figueiredo et Fernando [FF04] ont conçu un algorithme parallèle pour un environnement de prototypage virtuel. Cette méthode atteint 100% d'amélioration en comparant l'utilisation d'un quadcore à un simple cœur. Ils montrent également que l'ajout de

plus de cœurs réduit les performances sur leur modèle. Tang et al. [TMT08] proposent d'utiliser une représentation hiérarchique afin d'accélérer les requêtes de détection de collision couplée à un algorithme incrémental permettant d'exploiter la cohérence temporelle. L'ensemble est réparti entre plusieurs cœurs. Ils obtiennent un gain de 4X-6X sur un processeur double quadcore entre des modèles déformables. Leur méthode assure une évolutivité élevée, même en utilisant 16 cœurs. Une mise en œuvre parallèle de la reconstruction de hiérarchies de volumes est présentée dans [Wal07]. Kim et al. [KHeY08] proposent d'utiliser une hiérarchie de volumes englobants basée sur les primitives pour améliorer les performances de la détection de collision en continue. Ils proposent également de nouvelles méthodes de décomposition et d'attribution des tâches. Leur modèle atteint un gain de 7X-8X en comparant l'utilisation de 8 cœurs à un simple cœur. Des méthodes statiques et dynamiques d'équilibrage de charge sont également utilisées par Kitamura et al. [KS95] afin d'accélérer la détection de collision lors d'un parcours d'arbres en parallèle. Ils démontrent d'ailleurs que la communication entre les processeurs est la principale cause de la limitation des performances. Assarsson et Stenstrom [AS01] réduisent par trois la détection de collision sur huit processeurs en proposant une solution intéressante permettant de se débarrasser de la plupart des verrouillages nécessaires. Une méthode pour extraire le parallélisme en utilisant une partition précalculées des tâches a été proposée par Grinberg et Wiseman [GW07]. Ils proposent un algorithme parallèle, évolutif et portable pour la simulation de la détection de collision qui s'exécute sur clusters et machines MPI.

2.2.4 Simulations parallèles basées particules

La technique de simulation n-body large échelle, dans laquelle les équations de mouvement de n particules sont intégrées numériquement, a été et est l'un des outils les plus puissants au sein de beaucoup de domaines d'étude tels que l'astronomie (système solaire [MTES97], etc.) ou les systèmes moléculaires. La parallélisation de ce type de système est également un vaste domaine d'étude. La non dépendance des particules entre elles, facilite le parallélisme et permet d'utiliser facilement des techniques de subdivision spatiale. Lewis et al. [LM06] présentent une méthode de multi-threading Java sur multi-cœur pour simuler des anneaux planétaires. Leur méthode tire parti de la localité spatiale de la dynamique de collision pour traiter efficacement l'ordre temporel des collisions.

Selle et al. [SSIF09] proposent un modèle pour la simulation de vêtements en parallèle. Chaque particule des mailles du tissu est attribuée à un processeur à l'aide d'une division médiane récursive. Leurs tests sont effectués sur une machine bi-processeur dualcore. Thomaszewski et Blochinger [TB07] présentent également un modèle parallèle de simulation de vêtement mais orienté processeur sur clusters.

2.2.5 Synthèse

À travers ces quatre familles d'approches basées CPU, nous avons pu constater que les approches multi-cœur sont encore peu nombreuses et il reste encore beaucoup d'études à effectuer et de comparaisons à mettre en place. En recoupant avec le Chapitre

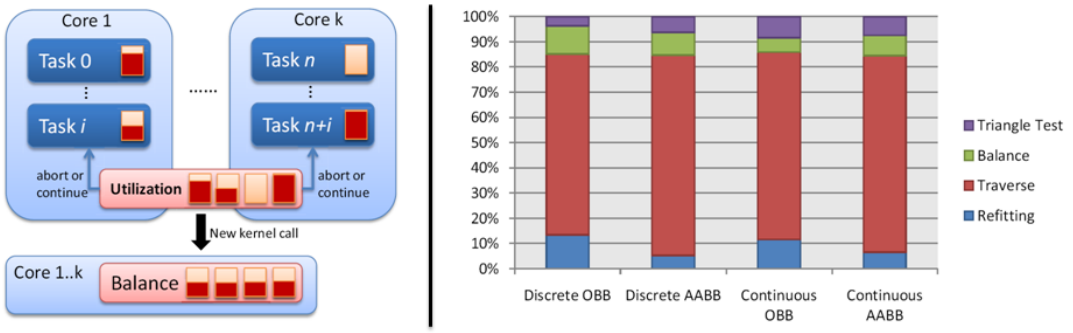


FIGURE 2.7 – Modèle de Lauterbach [LMM10] - **Gauche** : équilibrage de charge pour le calcul de la hiérarchie et son parcours - **Droite** : temps passé dans les parties de l'algorithme selon la méthode continue ou discrète et le volume utilisé.

1, nous avons pu constater que certains algorithmes très utilisés en simulation physique n'ont jamais été adaptés sur multi-cœur tel que l'algorithme de Broad phase du Sweep and Prune (SaP). L'utilisation de la subdivision spatiale afin de répartir les calculs entre les cœurs est également très peu étudiée. On constate également qu'aucune approche existante n'a étudié les solutions basées force brute, pourtant considérées comme les meilleures candidates pour le parallélisme sur architectures massivement parallèles. Enfin, les techniques d'équilibrage de charge sont également très peu utilisées sur les architectures multi-cœur pour la détection de collision. Cependant, nous pouvons ressortir des caractéristiques communes aux approches basées multi-cœur telles que la généricité, par le fait de s'exécuter sur des nombres quelconques de cœurs, et la rapidité car les approches se doivent d'être plus performantes que celles existantes. Ces caractéristiques sont donc des éléments essentiels dont il faut tenir compte pour proposer de nouveaux algorithmes parallèles basés multi-cœur.

2.3 Détection de collision basée GPU

Nous avons présenté dans la Section 1.3.2.3 les algorithmes utilisant le GPU pour déterminer l'absence ou la présence de collision. Ces derniers étaient basés sur des approches incluant les propriétés graphiques du GPU tels que le *Depth Peeling* ou le lancer de rayon. Dans cette section, nous développons les différentes contributions existantes basées GPGPU où les calculs effectués ne sont pas d'ordre graphique. Nous pouvons différencier différentes approches : les algorithmes basés structures hiérarchiques (cf. Section 2.3.1) et ceux utilisant la subdivision spatiale (cf. Section 2.3.2).

2.3.1 Structures hiérarchiques

La détection des collisions utilisant les structures hiérarchiques consiste à traverser en profondeur un arbre de volumes englobants. De nombreux travaux ont consisté en la réalisation d'algorithmes parallèles pour accélérer le parcours en profondeur de ces

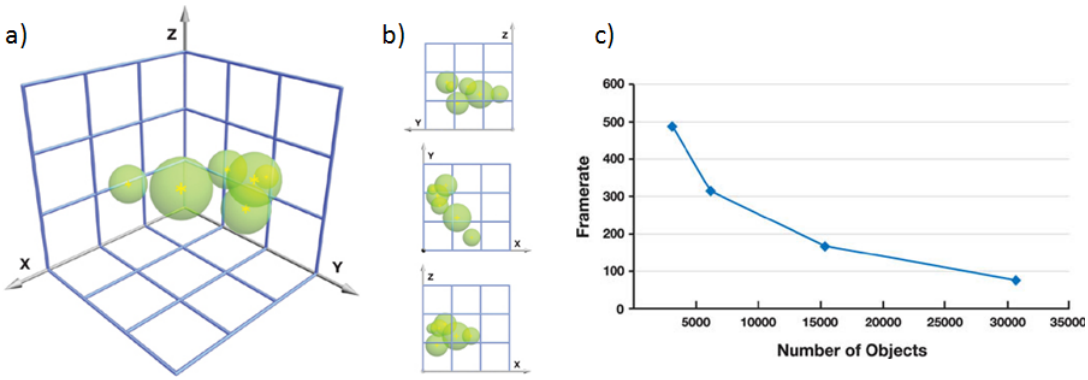


FIGURE 2.8 – Méthode de Le Grand [LG07] - **(a)** : un ensemble d'objets superposés à la grille uniforme qui partitionne l'espace 3D - **(b)** : les mêmes objets projetés sur les trois plans - **(c)** : évolution de la fréquence en fonction du nombre d'objets.

arbres. Rao et Kumar [KRR88] soulignent que l'efficacité de la recherche en profondeur parallèle est fortement influencée par le régime de répartition du travail et des caractéristiques architecturales. L'évolutivité de plusieurs techniques d'équilibrage de charge est analysée pour différentes architectures [KGR94]. Reinefeld et Schnecke [RS94] comparent la stratégie d'équilibrage de charge de deux méthodes de recherche en profondeur et proposent un système qui utilise des paquets de travail de taille fixe. Ces paquets de travail sont les unités élémentaires que se transmettent les unités de calcul lors de l'utilisation de méthodes d'équilibrage de charge.

Deux nouveaux algorithmes pour la construction rapide de BVH sur GPU ont été proposés par Lauterbach et al. [LGS+09]. Le premier utilise un classement spatial linéaire de type Z-curve des primitives pour construire des hiérarchies très rapidement. Il possède une forte évolutivité parallèle. Le second est une approche *top-down* qui utilise des heuristiques de surface (plus adaptées au lancer de rayon que les BVHs) pour construire des hiérarchies optimisées pour le lancer de rayon. Les hiérarchies qui en résultent sont très proches des hiérarchies SAH optimisées (*Surface Area Heuristics*), mais le processus de construction est sensiblement plus rapide. Ce qui conduit à un gain significatif car les deux coûts de construction et de parcours sont pris en compte et réduits. Ils réduisent ainsi le temps de construction par trois comparativement aux autres méthodes. Les structures de type KD-tree sont également optimisées pour être construites et mises à jour plus rapidement en utilisant le GPU [FS05, ZHWG08]. Très récemment, Tang et al.[TMLT11] ont proposé une nouvelle méthode de compactage de flux GPU permettant de réduire les temps d'accès et de calcul pour les calculs d'intersection entre primitives.

gProximity [LMM10] est une approche utilisant le parallélisme de tâches et de données sur GPU pour la construction, la mise à jour et le parcours d'arbres de volumes de type OBB et RSS (cf. Section 1.3.2.4). Ils montrent également que des hiérarchies de volumes englobants serrées (plus fines) offrent de meilleures performances sur les architectures GPU. Dans leur approche, chaque tâche conserve sa propre file d'attente de

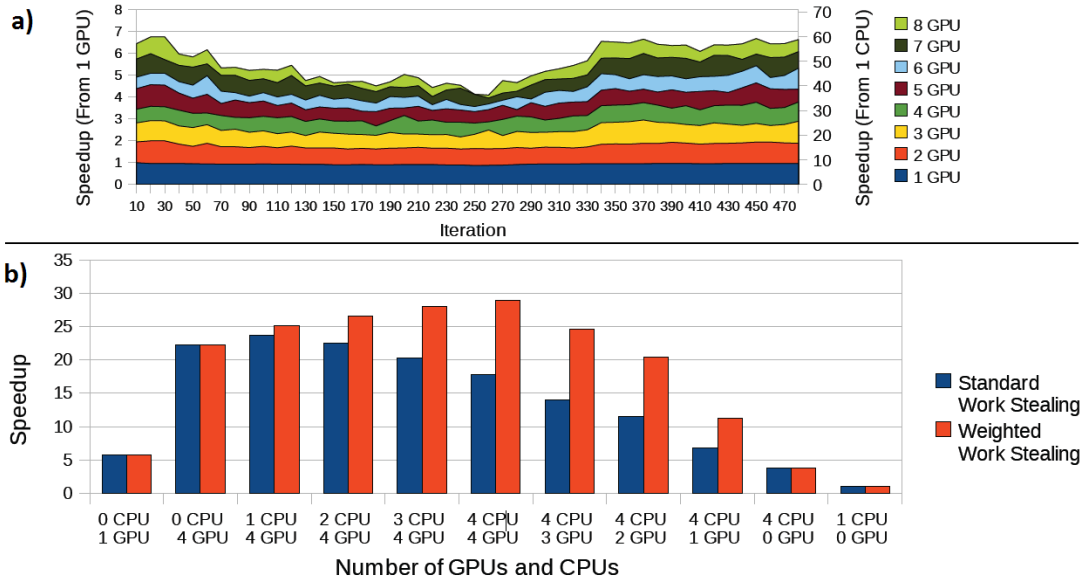


FIGURE 2.9 – Modèle hybride d'Hermann [HRF+10] - **(a)** : gain à chaque itération en simulant 64 objets déformables subissant la gravité - **(b)** : performances de simulation avec différentes combinaisons CPU/GPU.

travail dans la mémoire locale et peut générer des nouvelles unités de travail (telle que l'opération intersection) sans se coordonner avec les autres. Après le traitement d'une unité de travail, chaque tâche est soit en mesure d'en exécuter d'autres soit, si elle a une file d'attente vide ou complètement pleine, de s'annuler (cf. Figure 2.7).

2.3.2 Découpage spatial

Le Grand [LG07] fut le premier a proposé un algorithme de Broad phase sur GPU basé sur le SaP. Sa technique permet de coupler la subdivision en grille uniforme à la projection de la topologie des objets permettant ainsi un élagage rapide des paires n'étant pas en collision (cf. Figure 2.8). Liu et al. [LHLK10] proposent une nouvelle version de l'algorithme du SaP sur GPU avec subdivision spatiale, sans aucune restriction sur la taille des objets ou la nature de leur mouvement. L'idée principale est de profiter de la nature parallèle du SaP en l'effectuant simultanément pour de nombreux objets en utilisant le nombre de blocs de threads disponibles sur les GPUs. Ils ajoutent une analyse en composantes principales pour choisir la meilleure direction de balayage de l'algorithme ainsi qu'une nouvelle technique à deux niveaux de subdivision spatiale pour réduire le problème des intervalles densément projetés. Ils ont testé leur nouvel algorithme sur un NVIDIA Tesla C1060 avec au maximum un million d'objets rigides en mouvement.

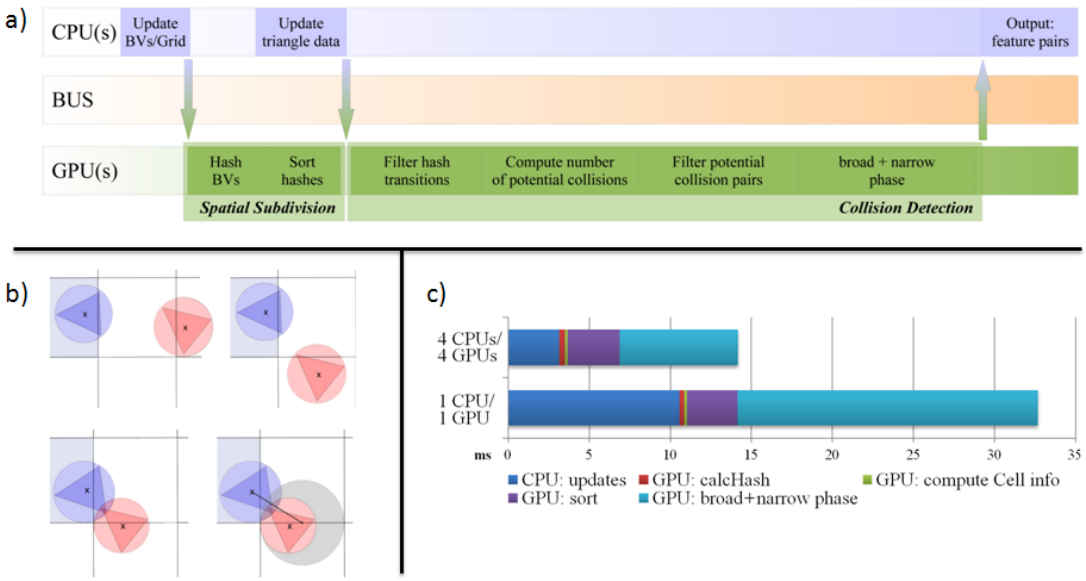


FIGURE 2.10 – Modèle hybride de Pabst [PKS10] - **(a)** : aperçu haut niveau du pipeline CPU/GPU - **(b)** : exemple 2D d'une paire *fantôme-fantôme* en collision (cf. Section 2.4) - **(c)** : répartition montrant le coût relatif de calcul des étapes clés de l'algorithme.

2.3.3 Synthèse

Les deux familles d'algorithmes basés GPU que nous avons présenté montrent clairement que l'objectif visé est de proposer une méthode performante pour massivement paralléliser les calculs. L'utilisation de structures hiérarchiques pour représenter les objets et de subdivision spatiale pour l'environnement permettent donc d'obtenir une indépendance au niveau des données et facilitent donc le parallélisme sur GPU. Tout comme les approches basées CPU multi-cœur présentées dans la Section précédente, les approches basées GPU possèdent des caractéristiques communes telles que la généricité et la rapidité. Cet aspect de parallélisme massif sur GPU diffère de ceux basés CPU multi-cœur. Le principal inconvénient de l'utilisation du GPU réside dans le transfert de données CPU-GPU. Cet inconvénient peut, dans certains cas, lorsque les dépendances de données le permettent être masqué lors de calculs effectués de façon asynchrone avec le transfert de données. Proposer des méthodes qui utilisent le GPU comme accélérateur de calculs semble donc une solution plus que performante mais nécessitant d'être davantage étudier. L'optimisation des transferts de données en entrée et en sortie du GPU doivent également être optimisés. Les approches de force brute sont également peu étudiées sur GPU, il serait intéressant de connaitre les limites précises de ce type d'approche et leur efficacité selon le type de simulation effectuée.

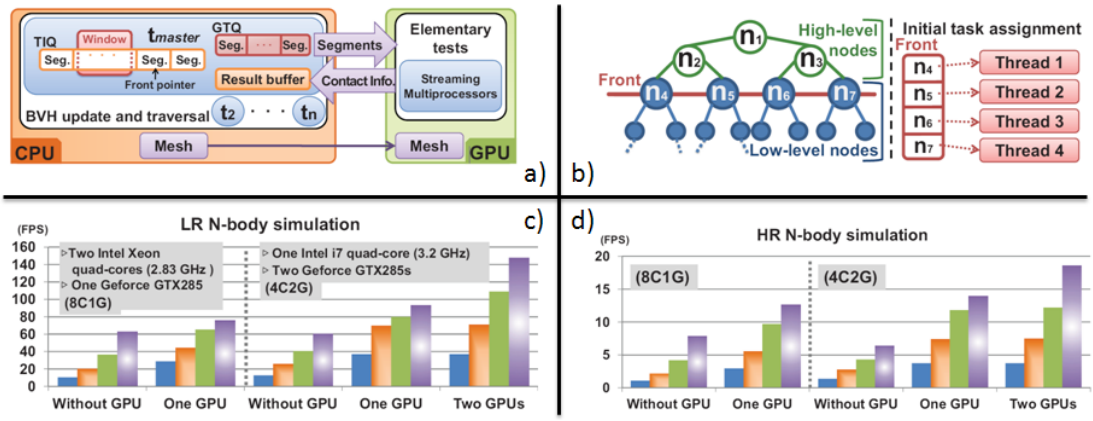


FIGURE 2.11 – Modèle hybride de Kim [KHH+09] - **(a)** : structure globale de la méthode HPCCD - **(b)** : nœuds haut et bas niveau pour quatre threads disponibles - **(c)** : résultats sur 8 cœurs/1 GPU et sur 4 cœurs/2 GPUs de simulation *n*-body de 34K triangles - **(d)** : résultats pour simulation *n*-body de 146K triangles.

2.4 Solutions Hybrides

Les travaux s'inscrivant dans cette nouvelle thématique des solutions hybrides proposent des solutions permettant de combiner le (ou les) CPU(s) au (ou aux) GPU(s) afin de répartir les calculs. De nombreuses combinaisons hybrides d'algorithmes GPU et CPU ont été proposées pour effectuer des tests de collisions et des requêtes de distance basées sur l'espace objet. Juarez-Comboni et al. [JMJC05] ont dès 2005, décrit l'utilisation de plusieurs GPUs pendant le processus de détection de collision. Un GPU est en charge du processus de détection de collision en utilisant des requêtes simples basées volume tandis que l'autre GPU est en charge des opérations de rendu. Les résultats du second étant dans certains cas utilisés par le premier.

2.4.1 Parcours d'arbres

Kim et al. [KHH+09] ont présenté une méthode hybride parallèle continue de détection de collision, appelée *HPCCD*, basée sur une hiérarchie de volumes englobants (BVH) (cf. Figure 2.11). *HPCCD* tire profit du parallélisme des architectures multi-cœur pour effectuer le parcours des BVH afin d'éliminer les paires n'étant pas en collision tandis que les GPUs sont utilisés pour effectuer des tests élémentaires qui permettent de réduire la résolution des équations cubiques. Leur méthode est basée sur une décomposition des tâches qui conduit à un algorithme parallèle sans verrou au sein de la boucle principale de parcours des BVH, permettant ainsi d'être évolutif et porté sur n GPUs. Cependant, ils n'ont testé leur approche que sur une architecture munie seulement de deux GPUs.

2.4.2 Subdivision spatiale et topologie

Pabst et al. [PKS10] ont présenté un nouvel algorithme hybride pour les objets rigides et déformables basé sur la subdivision spatiale. La Broad phase et la Narrow phase sont toutes deux exécutées l'une après l'autre sur une architecture multi-GPU. La Broad phase utilise une méthode adaptée de subdivision spatiale hautement parallèle permettant d'élaguer rapidement les paires d'objet de l'environnement n'étant pas en collision. Pour cela, ils étendent la méthode de Le Grand [LG07] (appliquée aux particules) et l'appliquent à des maillages triangulaires de taille variable. Avant la création de la grille, un test de chevauchement basé k-DOP est effectué sur les objets actifs, des sphères englobantes sont ensuite calculées pour chaque triangle en parallèle sur le GPU. La taille de cellule de la grille est ainsi définie par le rayon de sphère le plus grand (moyennant un epsilon de l'utilisateur). La Narrow phase est, quant à elle, basée sur des tests élémentaires parallèles sommet-triangle et arête-arête. Leur comparaison avec les travaux existants révèle que leur approche hautement parallèle est mieux adaptée aux architectures multi-GPU/multi-CPU.

2.4.3 Ordonnancement et équilibrage

Herman et al. [HRF+10] ont également proposé un modèle hybride de détection de collision. Leur méthode s'appuie sur un ordonnancement à deux niveaux associant un partitionnement du graphe de tâches et une technique de vol de tâches. Ils profitent de la localité spatiale et temporelle pour la planification des tâches. La localité temporelle repose principalement sur la réutilisation de la distribution de partition entre les itérations consécutives.

2.4.4 Synthèse

Cette famille de solutions hybrides est encore très récente au sein du domaine des solutions parallèles pour la détection de collision. Elle est, pour le moment, assez peu explorée et seules quelques approches ont été proposées. Il est tout de même possible d'établir des similitudes entre les différentes approches existantes, telles que la rapidité, la généricité et la structuration qui apparaissent comme les trois caractéristiques majeures les plus mises en avant. Le critère de structuration correspond à l'organisation du pipeline que les algorithmes hybrides choisissent afin d'effectuer des étapes sur CPU et/ou sur GPU. Il reste encore de nombreuses voies à explorer au sein des solutions hybrides afin de continuer à réduire le temps de calcul.

2.5 Conclusion

De l'analyse de ces solutions parallèles de détection de collision, nous déduisons un certain nombre de fonctionnalités essentielles à la réduction du goulet d'étranglement calculatoire au sein d'environnements large échelle :

– **La Généricité** : les algorithmes et modèles se doivent d'être génériques pour pouvoir, dans un premier temps, s'adapter aux nombreuses architectures actuelles,

mais surtout pour pouvoir anticiper les prochaines. Le nombre perpétuellement croissant de cœurs sur un microprocesseur et de GPU dont dispose une machine, force les algorithmes à prendre en compte un nombre quelconque d'unités de calcul.

- **L'Adaptativité** : les scénarios de simulation physique n'étant généralement pas prévisibles à l'avance, le nombre d'objets et de polygones dans un environnement peut fortement évoluer au cours d'une simulation (Ajout, Suppression, Fracture etc.). L'algorithmique en charge de détecter les collisions doit être en mesure de s'adapter dynamiquement à ces variations. Or le survol de la littérature nous montre qu'aucune approche ne va, pour le moment, dans ce sens.

- **La Rapidité** : hormis le fait d'être générique et adaptatif, les algorithmes se doivent également d'être de plus en plus rapides. Au travers de la littérature nous avons pu constater que cette propriété est le principal gage de qualité d'une nouvelle approche. Le but n'est pas uniquement d'accroitre la rapidité des modèles en profitant du parallélisme mais de coupler ce parallélisme à une nouvelle algorithmique de détection de collision plus performante.

- **La Structuration** : on constate, dans les travaux en cours, que la structure en pipeline est l'unique stratégie adoptée. La tendance va d'ailleurs vers l'ajout de nouveaux filtres au sein du pipeline pour réduire encore plus le nombre d'éléments à tester. Les approches détaillées précédemment ne remettent pas en cause la structure même du pipeline qui pourrait, elle-même, être parallélisée en suivant le principe du parallélisme fonctionnel présenté dans la Section 2.1.3. Utiliser des algorithmes différents en fonction des propriétés géométriques des objets fait également partie de ce critère structurel auquel peu de travaux font, pour le moment, allusion.

Notre étude approfondie de la littérature nous a permis de mieux cerner l'objectif de nos travaux en s'attachant à proposer de nouveaux modèles répondant à toutes les fonctionnalités présentées ci-dessus [AGA09b]. Nous avons ainsi pu concevoir des modèles répondant aux différents critères nécessaires à la réduction du goulet d'étranglement engendré par les algorithmes de détection de collision. Nous présentons dans le prochain chapitre, des modèles basés multi-cœur permettant de réduire ce goulet d'étranglement et prenant en compte les quatre critères évoqués.

Chapitre 3

Solutions Parallèles basées CPU

Ce troisième chapitre présente nos contributions basées CPU. Nous avons étudié l'évolution des processeurs simple cœur à multi-cœur afin de proposer des algorithmes adaptés à ce type d'architecture répondant aux critères de généricité, de structure et de rapidité établis précédemment. Nous présentons, dans un premier temps, un algorithme basé multi-cœur pour l'étape de Broad phase (cf. Section 3.1). Cet algorithme respecte les critères de généricité et de rapidité en étant en mesure de s'exécuter sur un nombre quelconque de cœurs tout en réduisant le temps de calcul. Nous présentons ensuite un répartiteur parallèle pour l'étape de Narrow phase (Section 3.2) basé sur nouveau concept de matrice algorithmique. Cette solution permet de paralléliser les calculs selon des classes d'objets préalablement définies. Ce nouvel algorithme de Narrow phase répond aux critères de structuration (le fonctionnement algorithmique interne est remanié), de rapidité (nos tests montrent qu'il est plus rapide que ceux existants) et de généricité (prise en compte d'un nombre quelconque d'unités de calcul).

3.1 *Sweep and Prune* multi-cœur

Nous décrivons dans cette partie une solution développée pour les architectures multi-cœur. Nous nous sommes intéressés à l'optimisation et la création d'un algorithme conçu pour être paralléliser sur un nombre quelconque de cœurs. Nous présentons un algorithmes de Broad phase basé sur l'algorithme du Sweep and Prune (SaP) [CLMP95] au sein duquel nous avons mis en place une nouvelle structure de données ainsi qu'une amélioration de l'algorithme permettant ainsi de supprimer les dépendance existantes et donc de le paralléliser sur un nombre quelconque de cœurs.

3.1.1 Positionnement de l'approche

Notre premier modèle d'algorithme de Broad phase basé multi-cœur est basé sur l'algorithme du SaP. Après le survol de la littérature, nous avons pu constater que cet algorithme est très largement utilisé car il offre des capacités d'élagage et de rapidité faisant de lui un excellent candidat pour la Broad phase. Les nombreux travaux ayant porté sur l'amélioration de cet algorithme se sont principalement attachés à réduire le temps de calcul et/ou à améliorer l'élagage mais généralement basé sur une exécution

séquentielle. Coming et Staadt [CS06, CS08] proposent d'améliorer l'algorithme pour les objets qui suivent des trajectoires en fonction du temps. La complexité de calcul du SaP est connu pour être en $O(n+s)$, où s est le nombre de *swaps* nécessaires pour maintenir l'ordre de tri des objets. Tracy et al. [TBW09] ont ainsi montré que s peut augmenter de manière "super-linéaire" par rapport au nombre d'objets. Ils motivent ainsi l'utilisation d'une approche hybride du SaP couplée à une subdivision spatiale permettant de réduire le nombre de swaps. Il n'existe aucune approche de SaP permettant de bénéficier du parallélisme offert par les architectures multi-cœur.

Notre nouvel algorithme est conçu pour s'exécuter sur un processeur multi-cœur quel que soit le nombre de cœurs [AGA09a, AGA10a]. Nous avons travaillé sur deux modèles différents. Le premier est un algorithme de force brute (complexité en $O(n^2)$) de comparaison de volumes englobants. Le second utilise une subdivision spatiale pour exécuter en parallèle l'algorithme sur des sous-parties de l'environnement. Ce second algorithme, différent de l'approche de force brute, utilise une structure de données incrémentale mise à jour à chaque pas de temps.

3.1.2 Approche basée force brute

Pour n objets, l'approche basée force brute consiste à effectuer n^2 tests. L'intérêt majeur de cette approche est qu'elle offre une possibilité de paralléliser massivement les calculs. Un autre avantage de ce type d'approche est l'absence de calcul en amont pour réduire les tests à effectuer. Les données sont prises brutes et transmises à l'algorithme qui calcule, pour toutes les combinaisons possibles, la potentielle présence ou l'absence de collision. L'inconvénient principal réside dans le nombre de tests à effectuer. En effet, le nombre de tests croit de manière quadratique en fonction du nombre d'objets. Si on multiplie par 10 le nombre d'objets on multiplie par 100 le nombre de paires à tester. Or le calcul à effectuer est relativement simple. Étant donné deux objets A et B, on cherche à déterminer si les volumes englobants sont ou non en collision. Nous utilisons dans notre approche des AABBs pour englober nos objets. Nous avons précédemment montré que le test de collision entre deux AABBs est simple et consomme très peu de temps de calcul (cf. Section 1.3.2.4). Nous avons donc décidé d'étudier le comportement de l'algorithme de force brute en le parallélisant sur une architecture multi-cœur dans le but de déterminer si, parallélisé sur un certain nombre de cœurs, il s'avérerait plus performant que l'algorithme incrémental séquentiel. Et si oui, à partir de quel nombre de cœurs la méthode de force brute devient plus performante ?

3.1.2.1 Calcul de chevauchement

Dans le cas d'une simulation physique n-body, nous partons du principe qu'un algorithme dit de force brute effectue à chaque pas de temps n^2 calculs. Plus précisément nous n'effectuons que $\frac{n^2-n}{2}$ tests (cf. Figure 3.1). Les paires du type (n, n) ne sont pas testées (suppression de la diagonale de la matrice) ainsi que toutes les paires correspondant aux paires symétriques d'autres paires ((x, y) et (y, x)). La matrice est donc une matrice triangulaire supérieure privée de sa diagonale.

Le calcul effectué sur chacune de ces paires d'objets consiste simplement à déter-

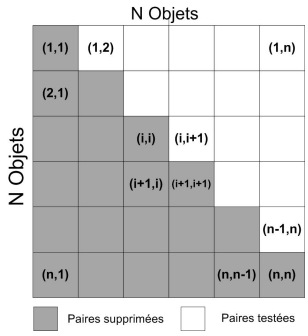


FIGURE 3.1 – Matrice des paires à tester dans un environnement composé de n objets.

miner si les volumes englobants des deux objets composant une paire sont ou non en chevauchement. Nous utilisons des volumes englobants alignés sur les trois axes de l'environnement x, y et z (AABB). Nous avons donc besoin, pour chaque objet, de six valeurs :

- Bornes maximales sur les trois axes.
- Bornes minimales sur les trois axes.

Le volume englobant d'un objet est donc représenté comme suit :

$$\begin{pmatrix} Xmin & Xmax \\ Ymin & Ymax \\ Zmin & Zmax \end{pmatrix}$$

L'algorithme utilisé pour déterminer s'il y a ou non chevauchement entre deux volumes AABBs est décrit dans l'Algorithme 2.

Algorithme 2 Test de chevauchement entre deux AABBs

```
Paire p = (A, B)
Pour axe x, y et z Faire
  Si A.max[axe] < B.min[axe] OU B.max[axe] < A.min[axe] Alors
    Pas de collision entre A et B ;
  Fin Si
Fin Pour
Retour Collision
```

Cet algorithme permet de déterminer si l'une des bornes max d'un objet selon un axe est inférieure à la borne min de l'autre objet sur le même axe. Si c'est le cas, cela signifie qu'il existe un espace (de taille $max - min$) entre les deux objets. Il n'y a donc pas de chevauchement entre les deux AABBs.

3.1.2.2 Parallélisme

Les calculs effectués par les deux étapes de l'algorithme initial étant totalement indépendants les uns des autres, la parallélisation peut être effectuée sur multi-cœur.

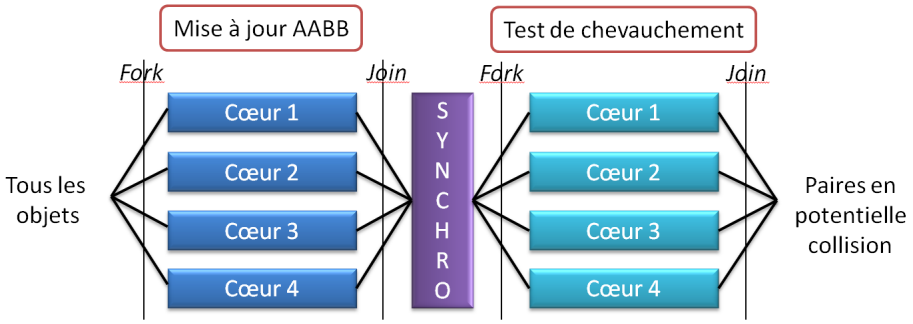


FIGURE 3.2 – Schéma global simplifié de l'algorithme du Sweep and Prune parallèle basé force brute.

	Intel Core(x2)		Intel Xeon(x4)		Intel Xeon(2x4)	
	CPUX7900 @2.8GHz		CPUX5472 @3.0GHz		CPUX5482 @3.2GHz	
Nb Objets	1 threads/C	2 threads/C	1 threads/C	2 threads/C	1 threads/C	2 threads/C
100	**0,43 ms**	0,471 ms	**0,312 ms**	0,37 ms	**0,11 ms**	0,137 ms
1000	**7,135 ms**	7,32 ms	**5,121 ms**	5,67 ms	**2,075 ms**	2,89 ms
10000	**41,115 ms**	42,615 ms	**29,175 ms**	31,41 ms	**18,75 ms**	20,03 ms
100000	**309,12 ms**	316,7 ms	**221,02 ms**	229,56 ms	**137,085 ms**	149,213 ms

FIGURE 3.3 – Vérification du nombre optimal de threads sur différents processeurs multi-cœur.

La première étape de l'algorithme consiste à mettre à jour les volumes englobants des objets. En effet, ces derniers évoluant au sein de l'environnement, les AABBs englobant les objets dynamiques doivent être recalculés à chaque pas de temps de la simulation. Cette première partie possède une complexité linéaire en fonction du nombre d'objet. Les calculs effectués sont également indépendants les uns des autres, il est donc tout à fait possible de massivement les paralléliser.

La seconde partie est en charge d'effectuer le test de chevauchement entre les différentes paires d'objets. La figure 3.2 présente les différentes étapes de notre algorithme. Tous les objets sont pris en entrée pour effectuer la mise à jour de leur AABB, on synchronise ensuite les résultats obtenus par les différents threads s'exécutant sur les différents cœurs. Les objets sont ensuite transmis à la deuxième étape afin de calculer les paires en chevauchement. Nous avons utilisé OpenMP pour paralléliser nos calculs sur les différents cœurs. L'emploi de directives OpenMP permet d'éviter la modification du code séquentiel contrairement à la librairie IntelTBB.

Chaque thread étant en charge d'opérations géométriques indépendantes, n'a nul besoin d'attendre et/ou de se synchroniser durant ces calculs avec un autre thread. La stratégie que nous adoptons est, par conséquent, la création d'un unique thread par cœur. La stratégie consistant à créer plus d'un thread par cœur ne réduirait pas, dans notre cas, le temps de calcul. Afin de s'en assurer, nous avons évalué les performances

des deux stratégies. La Figure 3.3 présente les résultats obtenus en exécutant le même scénario sur une architecture cible en utilisant, tout d'abord, un thread par cœur puis en doublant ce nombre. On constate que, quelle que soit l'architecture d'exécution et le scénario de test utilisé, la stratégie optimale est l'utilisation d'un thread par cœur. La création de deux threads par cœur augmente dans tous les cas le temps de calcul. Cette réduction n'est pas très importante mais montre que l'utilisation de deux threads par cœur n'est pas optimale.

L'algorithme 3 est la version séquentielle de l'algorithme tandis que l'algorithme 4 est la version parallèle adaptée pour une exécution sur multi-cœur.

Algorithme 3 Algorithme du Sweep and Prune séquentiel

```
Pour Tout (Objets) Faire
  Si (Objet courant est actif) Alors
    Mettre à jour AABB
  Fin Si
Fin Pour
Pour Tout (Paires d'objets) Faire
  Pour Tout (Axes(x,y,z)) Faire
    Si (A.max[axe] < B.min[axe] OU B.max[axe] < A.min[axe]) Alors
      Collision = faux
    Fin Si
  Fin Pour
  Si (Collision = Vrai) Alors
    Conserver paire
  Sinon
    Supprimer paire
  Fin Si
Fin Pour
```

Dans la première étape de l'algorithme parallèle, chaque thread travaille sur $\frac{n}{c}$ objets où n est le nombre d'objets dans l'environnement et c le nombre de cœurs. Il est possible de diviser les calculs des objets par le nombre de threads car la mise à jour des AABBs ne dépend pas de la complexité des objet, le temps passé sur un objet par un thread est, à peu de chose près, homogène quel que soit l'objet manipulé. Contrairement à l'algorithme séquentiel, où les nouveaux volumes calculés sont écrits sur la même structure de données, nous ne pouvons pas utiliser le même système sans éviter les sections d'écriture critique entre les threads. C'est pourquoi nous introduisons une nouvelle structure de données plus petite, utilisée par chaque thread afin de stocker les nouveaux AABBs calculés. Cette nouvelle structure est un tableau alloué dynamiquement en fonction du nombre de cœurs disponibles et du nombre d'objets. La synchronisation entre ces deux étapes est obligatoire afin de regrouper tous les nouveaux AABBs dans la même structure de données. Seuls les pointeurs de ces structures sont regroupés lors de l'étape de synchronisation pour éviter d'avoir à recopier les structures complètes.

Dans la deuxième partie de l'algorithme, chaque thread travaille sur $\frac{(n^2-n)}{2}/c$ des paires d'objets où c représente le nombre de cœurs. Comme dans la première partie,

Algorithme 4 Algorithme du Sweep and Prune parallèle

C : Nombre de cœurs
N : Nombre d'objets
t_{id} : Identifiant du thread
$Tab_1[]$: Tableau de C listes d'AABBs
$Tab_2[]$: Tableau de C listes de paires
Pour Tout (C threads) **Faire**
  Pour Tout (Objets entre $t_{id} * \frac{N}{C}$ et $((t_{id}+1) * \frac{N}{C} - 1)$) **Faire**
    Si Objet courant est actif **Alors**
      Mettre à jour AABB
      Stocker résultat dans $Tab_1[t_{id}]$
    Fin Si
  Fin Pour
Fin Pour
Synchronisation de tous les $Tab_1[t_{id}]$
Pour Tout (C threads) **Faire**
  Pour Tout (Paires entre $t_{id} * \frac{(N^2-N)}{2}$ et $((t_{id}+1) * \frac{(N^2-N)}{2} - 1)$) **Faire**
    Test de Chevauchement
    Stocker résultat dans $Tab_2[t_{id}]$
  Fin Pour
Fin Pour
Synchronisation de tous les $Tab_2[t_{id}]$

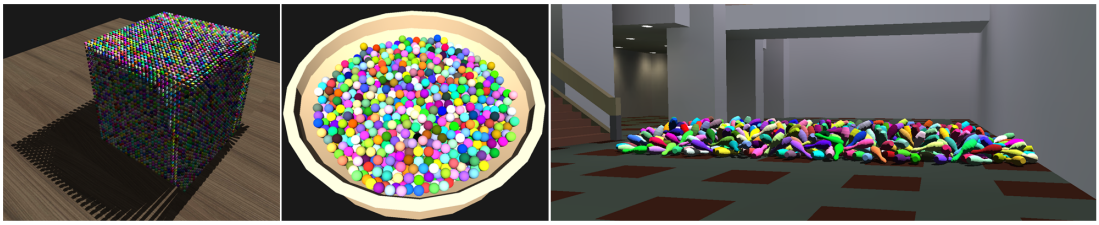


FIGURE 3.4 – Environnements de tests pour le modèle du Sweep and Prune basé multi-cœur.

	Cubes	Sphères	Quilles
1 cœur	8,89ms	4,45ms	1,6ms
2 cœurs	4,96ms	2,48ms	0,9ms
4 cœurs	2,76ms	1,4ms	0,5ms
8 cœurs	1,52ms	0,74ms	0,27ms

FIGURE 3.5 – Temps de calcul pour la mise à jour des AABBs pour chaque environnement de test de 1 à 8 cœurs.

chaque calcul, effectué par un thread, est un test de chevauchement entre les coordonnées des volumes englobants des objets d'une paire. Il n'existe donc pas de dépendance avec la complexité des objets manipulés. Pour éviter les sections d'écriture critique entre les threads, nous utilisons une technique similaire, où chaque thread est équipé de sa propre strucuture de stockage de données pour y stocker les paires dont les coordonnées des objets se chevauchent. Toutes les structures des threads sont ensuite fusionnées afin de créer la liste globale des paires d'objets en potentielle collision. Cette liste est ensuite transmise à la Narrow phase afin d'y subir un test de détection de collision plus précis.

3.1.2.3 Évaluation des performances

Nous présentons dans cette section les résultats de l'algorithme de Broad phase basé multi-cœur. Les tests ont été réalisés sur trois environnements large échelle (cf. Figure 3.4). Afin d'obtenir des résultats homogènes, nous avons mesuré ces temps de calcul sur le même processeur Intel Xeon (Double-Quad) CPU X5482 de 3,2 GHz. Les environnements utilisés sont les suivants :

- 10K balles de 2K polygones tombant dans un bol de 600 polygones (= 1.1M polygones).
- 20K cubes de 12 polygones tombant sur un plan (= 240K polygones).
- 3.5K formes non-convexes (quilles de 20K polygones) dans un environnement complexe de 10K polygones.

Nous présentons les résultats des temps de calcul pour tous les scénarios de tests utilisés (cubes, sphères et quilles). Les tests sont effectués sur des simulations d'objets rigides englobés par des AABBs. Les résultats numériques concernant la première étape de l'algorithme sont présentés dans le tableau 3.5. La réduction du temps de calcul total est représentée sur le graphique de la Figure 3.7. La figure présente le pourcen-

	Cubes	Sphères	Quilles
1 cœur	53,339ms	26,7ms	10,71ms
2 cœurs	31,65ms	15,748ms	6,35ms
4 cœurs	18,76ms	9,51ms	3,742ms
8 cœurs	11,43ms	5,82ms	2,314ms

FIGURE 3.6 – Temps de calcul pour le calcul de paires en chevauchement pour chaque scénario de test en utilisant de un à huit cœurs.

tage de réduction du temps de calcul comparativement à l'exécution séquentielle. Pour un scénario, les quatre colonnes montrent le temps de un à huit cœurs. Nous pouvons remarquer que le temps diminue lorsque le nombre de cœurs augmente. Le temps de calcul total est réduit à 56,04% sur deux cœurs, à 31,49% sur quatre cœurs et à 17,03% sur huit cœurs.

Les résultats numériques pour la deuxième partie de l'algorithme sont présentés dans le tableau 3.6. Cette deuxième partie de l'algorithme est présentée graphiquement sur la Figure 3.8. On remarque le même gain de temps que la première partie. Le temps de calcul total est réduit à 59,2% sur deux cœurs, à 35,34% sur quatre cœurs et à 21,56% sur 8 cœurs.

Le gain général de notre algorithme parallèle est illustré sur la Figure 3.9. Sur ce graphique, notre algorithme est représenté par la courbe rose bornée par la courbe bleue correspondant à la vitesse théorique optimale. La Figure 3.10 présente, quant à elle, la comparaison de notre algorithme parallèle du SaP avec le modèle séquentiel de l'algorithme basé sur une structure incrémentale. Celui-ci est l'approche la plus fréquemment utilisée au sein des simulateurs physiques. La simulation de test a été effectuée avec 10.000 Objets en mouvement. Nous constatons que, parallélisé avec huit threads, notre algorithme parallèle devient plus performant que l'algorithme incrémental séquentiel. Plus précisément, notre approche devient aussi performante que le modèle incrémental à partir de six threads. Le recours a une algorithmique plus simple mais plus facilement parallélisable permet donc d'obtenir des résultats plus performants que les modèles séquentiels incrémentaux. Et ceux à partir d'un certain nombre de cœurs. Ce nombre de cœurs étant fortement dépendant du nombre d'objets dans la simulation.

3.1.2.4 Synthèse et perspectives

Nous avons présenté un nouvel algorithme parallèle pour la Broad phase basé sur la technique du SaP. La parallélisation est réalisable grâce au fait qu'il n'y ait plus de dépendance entre les calculs. Ces derniers peuvent être répartis sur un nombre quelconque de cœurs, sans perturber les résultats. Les mesures de performance montre que cette approche de force brute est une bonne candidate aux architectures parallèles de type multi-cœur. Les critères de généricité et de rapidité sont donc respectés. Nous n'avons pas tester notre algorithme sur un nombre supérieur de cœurs mais les résultats permettent déjà de montrer que, parallélisée sur huit cœurs, l'approche de force brute se révèle plus performante que le modèle incrémental séquentiel du SaP. La tendance

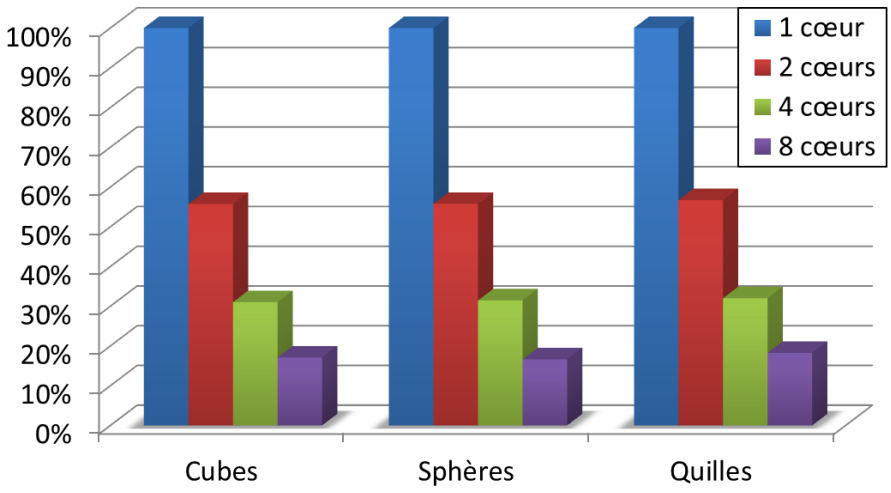


FIGURE 3.7 – Le temps de calcul de la mise à jour des AABBs en fonction du nombre de cœurs. Le temps de calcul est réduit à 17.03% en utilisant huit cœurs.

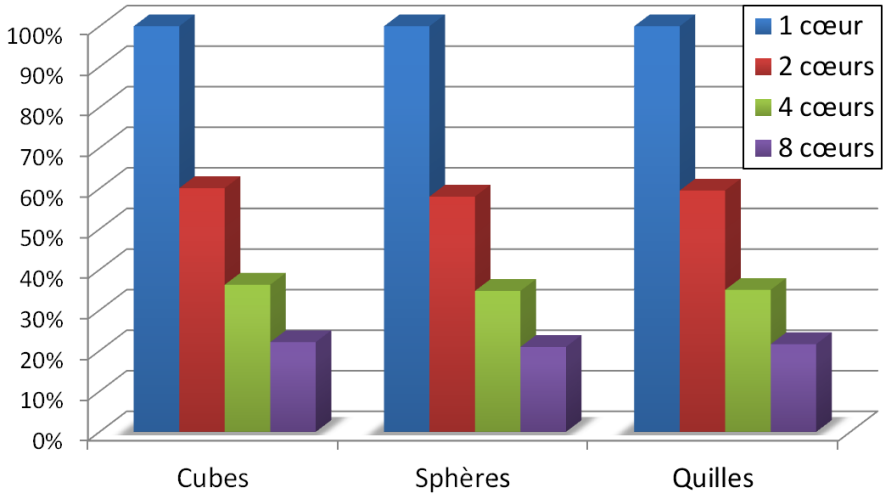


FIGURE 3.8 – Le temps de calcul du test des paires en chevauchement en fonction du nombre de cœurs. Le temps de calcul est réduit à 21.56% en utilisant huit cœurs.

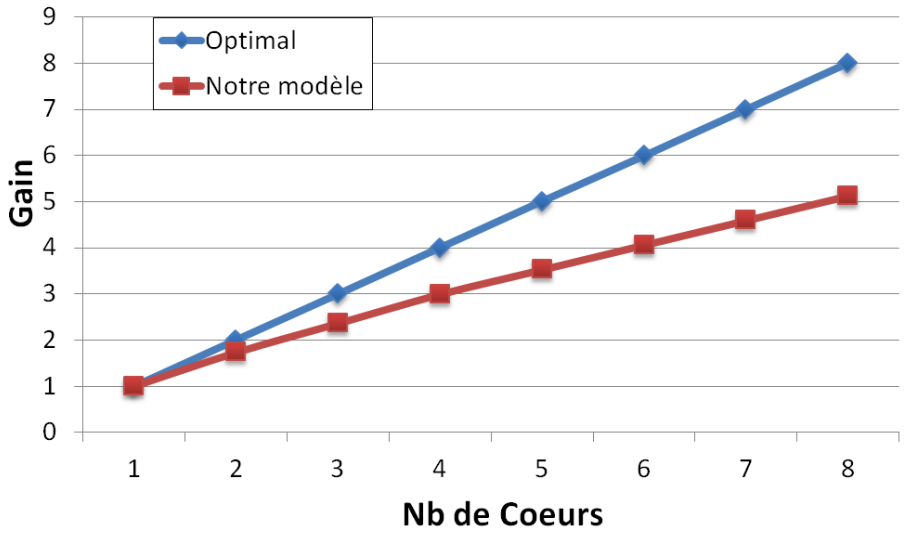
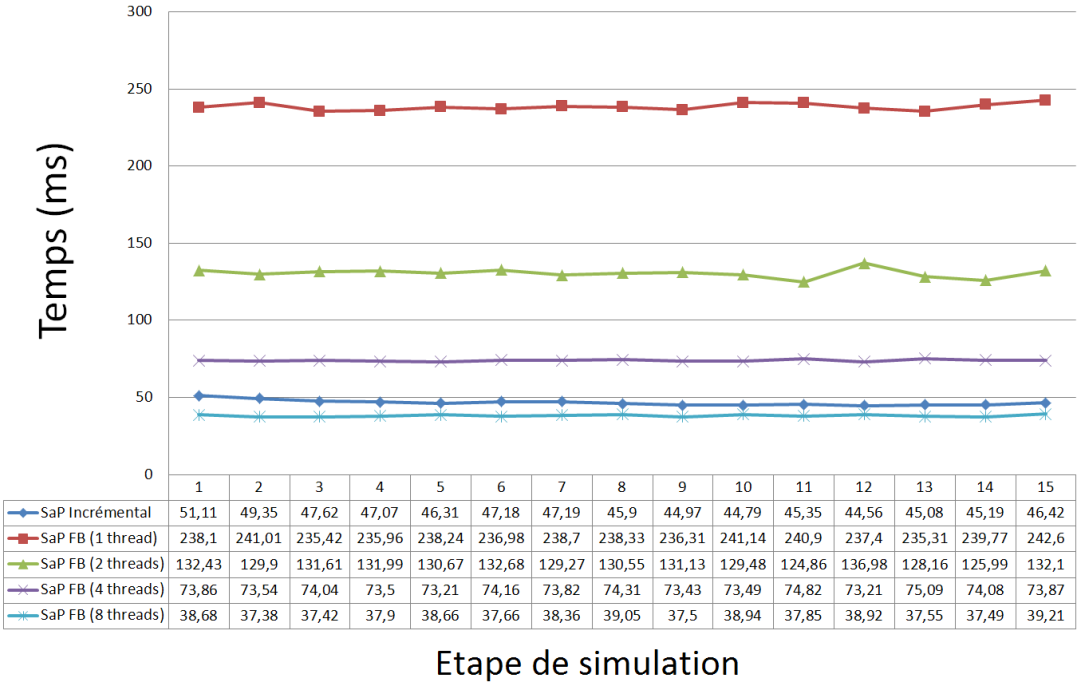


FIGURE 3.9 – Le gain d'accélération.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SaP Incrémental	51,11	49,35	47,62	47,07	46,31	47,18	47,19	45,9	44,97	44,79	45,35	44,56	45,08	45,19	46,42
SaP FB (1 thread)	238,1	241,01	235,42	235,96	238,24	236,98	238,7	238,33	236,31	241,14	240,9	237,4	235,31	239,77	242,6
SaP FB (2 threads)	132,43	129,9	131,61	131,99	130,67	132,68	129,27	130,55	131,13	129,48	124,86	136,98	128,16	125,99	132,1
SaP FB (4 threads)	73,86	73,54	74,04	73,5	73,21	74,16	73,82	74,31	73,43	73,49	74,82	73,21	75,09	74,08	73,87
SaP FB (8 threads)	38,68	37,38	37,42	37,9	38,66	37,66	38,36	39,05	37,5	38,94	37,85	38,92	37,55	37,49	39,21

FIGURE 3.10 – Comparaison de notre modèle parallèle de l'algorithme du SaP avec le modèle séquentiel de l'algorithme basé sur une structure incrémentale.

allant clairement vers un accroissement du nombre de cœurs sur les processeurs CPU, la généricité de l'algorithme à s'exécuter sur un nombre quelconque de cœurs permettra de bénéficier pleinement de ces nouvelles unités de calcul afin de continuer à réduire le temps de calcul.

Il serait intéressant de le tester sur des processeurs 16, 32 voire 64 cœurs afin d'analyser son comportement sur un nombre nettement supérieur d'unités de calcul. Il serait également intéressant d'enlever le point de synchronisation entre les deux étapes de mise à jour des AABBs et du calcul de chevauchement. Une possible voie d'optimisation pourrait être de coupler ce modèle à des solutions comme celles proposées par Hermann et al. [HRF09] afin d'éviter des points de synchronisation inutiles sur multi-cœur.

3.2 Narrow Phase multi-cœur

Nous nous sommes également intéressés à porter la seconde étape du pipeline sur architecture multi-cœur. Nous présentons, dans un premier temps, notre nouveau concept de matrice algorithmique dynamique permettant de rompre avec l'utilisation d'un seul et unique algorithme durant une simulation (cf. Section 3.2.1). Ce concept offre la possibilité de déterminer l'algorithme le plus approprié en fonction des classes d'objets considérées. Nous présentons ensuite notre approche permettant de porter ce concept sur architecture multi-cœur (cf. Section 3.2.2). Nous parallélisons les calculs des paires d'objets effectués par l'intermédiaire de la matrice. Nous évaluons ensuite les performances de notre approche en la comparant à l'approche existante (cf. Section 3.2.4). Cette "surcouche" multi-cœur offre donc la capacité à l'algorithme de Narrow phase de répartir les calculs de collision sur un nombre quelconque de cœurs (généricité) tout en proposant un nouveau mode de fonctionnement (structuration) plus rapide que le modèle existant (rapidité). Cette surcouche est également munie d'une technique d'équilibrage de charges basée sur le vol de tâches permettant d'adapter la charge de chacun des cœurs (adaptativité).

3.2.1 Matrice algorithmique

Dans le cas de notre étude de l'amélioration des modèles de détection de collision, nous avons également proposé un nouveau concept appelé "*matrice algorithmique*". Cette matrice permet de déterminer l'algorithme de Narrow phase le plus approprié à une paire d'objets en fonction des classes d'objets considérées. Les propriétés géométriques des objets 3D étant très larges, il est impossible de trouver un seul et unique algorithme capable de détecter des collisions entre deux objets quelconques et plus rapide que n'importe quel autre algorithme. Au vu de ce constat, il parait clair qu'une simulation physique munie d'objets très différents ne peut s'exécuter avec un seul algorithme. Nous proposons donc une répartition des objets selon différentes catégories telles que les objets simples convexes (cubes, sphères, cônes, cylindres etc.), les objets complexes convexes et les objets non-convexes. On peut visualiser cette matrice comme une table 2-D contenant des fonctions de détection de collisions spécifiques à des paires de formes géométriques précises. Les informations de forme peuvent alors être utilisées comme des clefs d'accès pour retrouver la fonction appropriée. Chaque objet se voit

donc attribuer des "*flags*" le décrivant tels que :

```
//Flags décrivant la géométrie des objets
CUBE, SPHERE, CYLINDER, CONE, CONVEX_MESH, NON_CONVEX_MESH;
```

On peut ainsi retrouver la fonction appropriée grâce à des appels du type :

```
// Méthode pour détecter les collision entre 2 objets
DetectCollision[Objet1.flag][Objet2.flag](Objet1, Objet2);
```

Nous présentons ensuite la manière dont l'évaluation des algorithmes est effectuée en distinguant deux types d'algorithmes : les fixes et ceux nécessitant des pré-calculs.

3.2.1.1 Évaluation algorithmique

Les algorithmes dont nous disposons sont testés sur toutes les différentes paires d'objets aux propriétés différentes afin d'évaluer leurs performances. Nous remplissons ensuite la matrice algorithmique avec les meilleurs candidats évalués. Cette matrice permet donc de classifier les algorithmes selon leur capacité à détecter rapidement une collision entre deux objets. Nous avons également constaté que certains algorithmes apparaissent très rapides mais fournissent moins d'informations sur le contact que d'autres algorithmes plus lents. Nous avons donc ajouté une nouvelle dimension à chaque case de la matrice permettant de classifier le degré de précision des informations de contact obtenues par l'algorithme. Il est donc possible de qualifier cette précision et de choisir à l'aide d'un curseur la qualité désirée. Ce curseur permet donc pour une paire de classe d'objets donnée d'étalonner le choix entre les deux extrêmes que sont la vitesse optimale et la précision optimale. Cette qualité peut s'exprimer, par exemple, comme le choix de la résolution de l'image lors de l'utilisation d'un algorithme de depth peeling. Une résolution 128*128 fournissant moins d'informations sur la localité de la collision qu'une résolution 800*600. La première étant, par contre, plus rapide que la seconde. Le remplissage de cette matrice algorithmique s'opère de deux façons différentes, certaines cases (algorithmes) sont considérées comme fixes et ne sont pas réévaluées. En revanche, certaines cases nécessitent des pré-calculs afin de déterminer le meilleur candidat.

Algorithmes constants

Certains algorithmes de la matrice algorithmique ne sont jamais remis en question et nous les considérons comme les meilleurs candidats pour un certain type de propriétés géométriques. Ces algorithmes, dits constants, sont ceux en charge de détecter les collisions entre formes géométriques simples (cubes, sphères, cylindres, plans, cônes etc.). L'algorithme permettant de détecter les collisions entre deux sphères est l'un des algorithmes les plus simples présents dans la matrice et ne nécessite pas de remise en cause. Il fonctionne simplement en comparant les centres des sphères et leur rayon respectifs (direction et longueur) pour déterminer s'il y a ou non collision et, le cas échéant, l'endroit exacte de la collision. Toutes ces algorithmiques entre formes géométriques simples

	Cube	Sphère	Convexes	Concaves
Cube	Cube/ Cube	Sphère/Cube	GJK	Parcours AABB
Sphère	Sphère/ Cube	Sphère/Sphère	GJK	Arbres de Sphères
Convexes	GJK	GJK	GJK	Parcours AABB
Concaves	Parcours AABB	Arbres de Sphères	Parcours AABB	Parcours AABB

FIGURE 3.11 – Exemple de matrice algorithmique obtenue en évaluant les performances des différents algorithmes.

ont déjà été étudiées longuement et améliorées à travers la littérature, il apparait donc qu'il n'est pas nécessaire de les remettre en cause. Cependant, ceci est un choix que nous avons effectué, il reste tout à fait envisageable de comparer un nouvel algorithme avec ceux existants pour déterminer le meilleur candidat.

Algorithmes pré-calculés

L'évaluation des algorithmes de détection de collision est une chose assez difficile à mettre en place. A l'inverse des algorithmes fixes, présentés précédemment, ceux, en charge de détecter les collisions entre objets convexes ou non-convexes ne correspondant pas à des formes géométriques simples, sont nombreux. Comme nous l'avons présentée dans notre descriptif de la littérature, il existe une large palette de familles d'algorithme (cf. Section 1.3.2). L'objectif est donc de comparer ces différents algorithmes sur les mêmes scénarios afin d'évaluer leur performance. Cette évaluation permettant ensuite de les classer par ordre de rapidité de calcul. Voici les algorithmes que nous avons étudiés pour établir un classement :

- Convexe/Convexe : Parcours AABB, GJK[GJK88], LC[LC91]
- Convexe/Non-convexe : Parcours AABB, Parcours Sphère, Depth Peeling (GPU)
- Non-convexe/Non-convexe : Parcours AABB, Parcours Sphère, Depth Peeling (GPU)

Notre objectif à travers ces scénarios hors-lignes est d'établir un classement des algorithmes. Nous ne disposons évidemment pas de tous les algorithmes récents existants, ce listing n'est donc pas un classement officiel des meilleurs algorithmes pour détecter les collisions entre deux objets aux propriétés spécifiques. Notre but est de proposer un nouveau concept offrant la possibilité d'utiliser plusieurs algorithmes durant une simulation. La Figure 3.11 présente un exemple de matrice qu'il est possible d'obtenir en évaluant les performances des différents algorithmes entre eux sur des scénarios similaires.

3.2.1.2 Comparaison et perspectives

Malgré le fait que la plupart des moteurs physiques procèdent désormais de la même manière, en définissant un tableau statique regroupant des classes d'objets, notre modèle élargit le concept en introduisant un caractère dynamique à ces tableaux. Ces tableaux indiquaient quels étaient les algorithmes utilisés pour traiter une paire d'objet appartenant à telle ou telle classe d'objet mais ils ne tenaient pas compte de l'architecture d'exécution. En introduisant une dynamique dans ces tableaux, il est désormais possible de comparer les performances des algorithmes entre eux afin d'établir un classement des meilleurs candidats. Ces évaluations de performance doivent être effectuées pour chaque architecture d'exécution différente. L'approche des moteurs physiques est similaire à la nôtre lors du calcul de collision entre formes géométriques simples mais l'inconvénient majeur de leur structure se situe à nouveau dans le traitement des objets complexes. En effet, lors du test d'intersection entre deux maillages complexes, ces moteurs apparaissent comme très peu optimisés car non conçus pour traiter ce type d'objets. Leur stratégie est généralement basée sur un parcours d'arbre en profondeur. Ils ne permettent pas de remettre en cause la rapidité des algorithmes utilisés afin de modifier la matrice algorithmique utilisée. Nos travaux s'étant focalisés sur les objets rigides, nous n'avons pas étendu cette matrice aux objets déformables et fluides mais cela reste tout à fait réalisable.

3.2.2 Répartiteur parallèle

La répartition des calculs se fait en divisant le nombre de paires dans la liste transmise par la Broad phase par le nombre de cœurs disponibles. Chaque thread se voit ensuite attribué un tantième de cette liste et effectue les calculs sur les paires de cette sous-liste. Afin de réduire au minimum l'attente des threads, chacun d'entre eux manipule différentes structures locales de données. Nous présentons également comment réduire les temps de répartition et de synchronisation engendré par le parallélisme.

3.2.2.1 Données locales

Afin de déterminer l'algorithme, parmi ceux déterminés comme les meilleurs candidats pour des propriétés géométriques définies, à appliquer sur la paire courante, les threads doivent avoir accès à la matrice algorithmique. Malgré le fait que les accès ne seront que des lectures de la matrice, ils se feront de manière séquentielle. Afin d'éviter toute séquentialité durant l'exécution, chaque thread possède sa propre copie de la matrice algorithmique. Cette copie locale pour chaque thread leur permet d'avoir autant d'accès que nécessaire sans aucune attente. Les résultats des calculs des paires sont également stockés localement dans une structure de données propre à chaque thread. Cela évite également les accès mutuels en écriture des threads sur une même structure de données globale. À la fin de tous les calculs, les structures locales des threads sont synchronisées pour être regroupées dans une seule et même structure.

3.2.2.2 Optimisation de synchronisation

Sachant que le parallélisme induit un coût supplémentaire dans la répartition des tâches et des données ainsi que dans la synchronisation des threads, nous avons également cherché à les optimiser. Le temps de répartition des données entre les threads est réduit en ne transmettant au thread que la première adresse correspondant à leur tantième de liste respectif ainsi que le nombre d'éléments qu'ils ont à traiter. Ils peuvent ainsi déduire eux-mêmes l'ensemble des adresses des éléments qu'ils ont à calculer. Afin de réduire au minimum le temps de synchronisation nécessaire à la mise en commun des résultats de calcul des différents threads, seules les adresses des données locales des threads sont regroupées. Cela évite d'avoir à regrouper l'ensemble des données calculées par les threads. Chacun d'entre eux venant de calculer les collisions entre des objets d'une même paire, les résultats ont été stockées localement dans des structures de données.

3.2.3 Équilibrage de charges

Pour assurer une exécution homogène et équi-répartie entre les différents cœurs, nous nous sommes basés sur une technique de vol de tâches pour équilibrer les calculs entre les threads. Cette technique permet à un thread s'exécutant sur un cœur de "voler" des calculs à un autre thread n'ayant, quant à lui, pas terminer les siens. Cette méthode se basant sur des règles très simples, offre un résultat très performant. Dans notre cas, les calculs sur les paires d'objets ne nécessitent aucune synchronisation avec d'autres threads, ce qui permet à un thread de voler n'importe quelle paire à un autre.

3.2.4 Évaluation des performances

Nous présentons, dans cette section, les résultats en termes de temps de calcul, obtenus par notre modèle multi-cœur de Narrow phase. Comme pour l'évaluation de la Broad phase, nous nous sommes basés sur des scénarios de test large-échelle munis d'objets très divers. Nous présentons les résultats obtenus sur l'environnement de test illustré par la Figure 3.12. Le test consiste à effectuer une simulation composée d'un grand nombre d'objets aux propriétés géométriques différentes. Nous utilisons un grand nombre d'objet car les bénéfices du parallélisme ne sont généralement appréciables qu'à grande échelle et que, comme nous l'avons présenté précédemment, le parallélisme induit un coût supplémentaire lors de la répartition des données. Ce coût explique d'ailleurs, en partie, l'impossibilité quasi-générale de diviser par deux le temps de calcul sur un processeur dual-core. À travers ce test, nous avons cherché à comparer le temps de calcul de l'exécution séquentielle à celle parallèle offerte par notre modèle de répartiteur pour la Narrow phase.

La simulation est donc composée de 10.000 objets allant de formes simples tels que des cubes et des sphères, en passant par des objets plus complexes et convexes, jusqu'aux objets non-convexes complexes (éléphant, volant, écrou, vis, chaise). Le graphique présenté sur la Figure 3.13 présente le temps de calcul de l'exécution séquentielle et parallèle. Les tests ont été effectués sur un processeur dual-core de type Intel Core2 Extreme CPU X7900 de 2,8 GHz. On voit clairement sur le graphique le gain offert

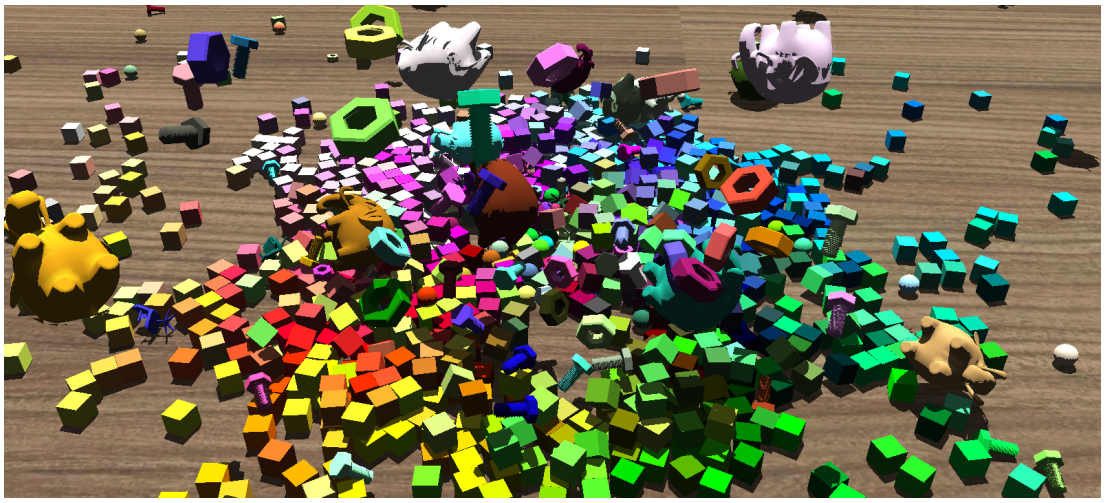


FIGURE 3.12 – Environnement utilisé pour tester notre modèle de répartiteur parallèle pour la Narrow phase.

par l'approche parallèle. Nous avons mesuré le temps de calcul pour le même scénario sur les 120 premiers pas de temps de la simulation. Les objets sont tous laissés tomber sur un plan au même moment. Le temps nul au début de la simulation s'explique par le fait que lors de la chute les objets sont relativement éloignés les uns des autres. La Broad phase ne les voit donc pas comme potentiellement en collision, par conséquent, la Narrow phase n'a rien à calculer. La courbe qui ensuite évolue, retrace le fait que les objets arrivent sur le sol et sont en collision entre eux. Le temps de calcul de l'exécution parallèle correspond en moyenne à 56,75% de celui de l'exécution séquentiel. Le temps est à peu de choses près divisé par deux en utilisant ce répartiteur parallèle.

3.2.5 Synthèse et perspectives

Nous avons présenté un nouvel algorithme parallèle de Narrow phase conçu pour une exécution sur architecture multi-cœur basé sur le concept de matrice algorithmique dynamique. Ce nouveau concept change la structuration interne du fonctionnement de la Narrow phase. Notre algorithme répond au critère de généricité en exécutant en parallèle autant de calcul de paires qu'il y a de cœurs disponibles. Les résultats montrent également que cet algorithme est plus rapide que l'approche séquentielle de Narrow phase. En utilisant la matrice algorithmique dynamique, cela permet également de pouvoir avoir différents algorithmes s'exécutant en parallèle en fonction des classes d'objets définies.

Nous avons testé notre modèle sur huit cœurs CPU au maximum, il serait intéressant d'évaluer ses performances sur un nombre supérieur de cœurs. Il serait également nécessaire d'établir un classement précis des algorithmes grâce à leurs évaluations réalisées pré-simulation.

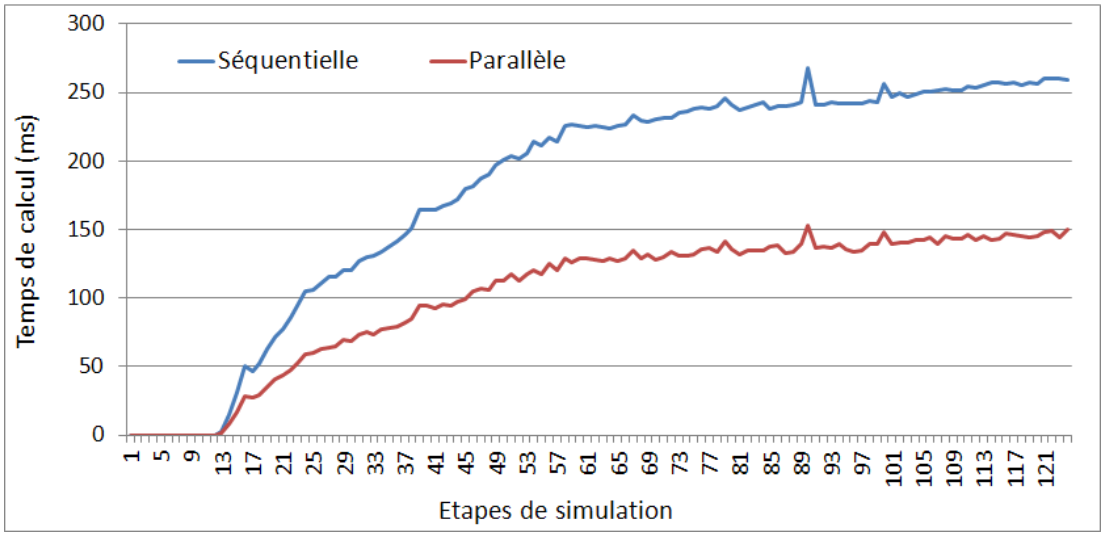


FIGURE 3.13 – Comparaison entre le temps de calcul du répartiteur séquentiel et parallèle.

3.3 Conclusion

A travers ce chapitre, nous avons présenté nos travaux sur l'optimisation et la mise en place d'algorithmes de concepts basés CPU. Nous avons présenté nos travaux sur la proposition de modèles basés multi-cœur pour la Broad phase et la Narrow phase. Le premier algorithme de Broad phase, basé sur le SaP de force brute, a permis de montrer que pour obtenir de meilleures performances, l'usage d'algorithmique plus simples et hautement parallélisables peut s'avérer être une solution performante. Parallélisé sur huit cœurs, l'algorithme parallèle de force brute s'avère plus performant que celui incrémental.

Le concept de matrice algorithmique permet de modifier la structure interne de fonctionnement des approches existantes. Les classes d'objets définies par l'utilisateur définissent désormais l'algorithme à utiliser pour détecter les collisions. Le répartiteur parallèle pour l'étape de Narrow phase s'avère également performant. Plusieurs algorithmes différents issus de la matrice algorithmique peuvent désormais être exécutés en parallèle.

Toutes ces approches basées multi-cœur répondent donc bien au critère de généricité car elles sont capables de s'exécuter sur un nombre quelconque de cœurs. Leur rapidité a également été montrée par les différents tests que nous avons effectués. Le concept de matrice algorithmique respecte, quant à lui, le critère de structuration en redéfinissant le fonctionnement interne de la Narrow phase.

Après s'être intéressés à proposer de nouveaux algorithmes parallèles performants basés CPU, nous présentons par la suite nos travaux sur la proposition d'algorithmes parallèles basées GPU.

Chapitre 4

Solutions Parallèles basées GPU

Nous présentons à travers ce chapitre la seconde partie de nos travaux ayant porté sur la proposition de nouveaux modèles génériques et performants basés GPU. L'architecture hautement parallèle qu'offrent les GPUs ouvre une nouvelle dimension dans la résolution des problématiques de temps de calcul des algorithmes. L'état de l'art a montré que la Broad phase est une étape essentielle du pipeline de détection de collision. Nous nous sommes ici intéressés aux critères de généricité et de rapidité pour les algorithmes impliqués dans l'étape de Broad phase. Nous présentons, dans un premier temps, un nouvel algorithme basé sur une approche topologique semi-brute [AGA11a] (cf. Section 4.1) exécuté sur GPU et apportant un gain significatif de temps de calcul. Nous proposons ensuite une nouvelle approche basée sur une optimisation mathématique d'un algorithme de force brute au sein duquel nous cassons la combinatoire par l'utilisation de subdivision spatiale de type grille (cf. Section 4.2). Le transfert de données CPU-GPU ayant une place prépondérante dans le goulot calculatoire, nous présentons également une optimisation de ces transferts. Les mesures de performance réalisées sur ces algorithmes de Broad phase montrent un gain significatif par rapport aux algorithmes existants et apportent la preuve de la réponse au critère de rapidité.

4.1 Broad phase topologique

Différentes approches, décrites dans le Chapitre 1, permettent de rapidement réduire le nombre de paires à manipuler. L'une des méthodes les plus utilisées au sein des simulations physiques, est celle du Sweep and Prune. Cet algorithme a déjà été notre point de départ pour nos solutions multi-cœur. Sa rapidité et simplicité de mise en œuvre en font un bon point de départ pour un algorithme de Broad phase parallèle. Nous présentons le positionnement de notre approche par rapport aux travaux existants ainsi qu'un chapitre préliminaire présentant les pistes possibles d'amélioration (cf. Section 4.1.1), nous décrivons ensuite notre méthode (cf. Section 4.1.2) suivie de l'évaluation des performances (cf. Section 4.1.4).

4.1.1 Positionnement de l'approche

Le Grand [LG07] est l'un des premiers à avoir proposé un modèle d'implémentation de l'algorithme du Sweep and Prune sur GPU. Son approche, afin de paralléliser l'algorithme, se base sur une subdivision spatiale de l'environnement en différentes cellules. Chaque cellule contenant ou non des objets, les tests sont effectués entre objets d'une même cellule et/ou avec les cellules voisines. Cette approche reste toutefois limitée à des objets de tailles quasi-similaires voire identiques. Comme le mentionne Le Grand, son approche se trouve limitée dans le cas d'objets de tailles très diverses. Il cite l'exemple d'une large planète autour de laquelle graviteraient des milliers de plus petits éléments. La grille résultant de son approche se résumerait à une seule et unique case englobant toute la scène. En effet, il définit la taille des cellules de sa grille uniforme comme la taille du plus grand volume englobant présent dans la scène. Son approche est très utilisée pour les simulations basées particules [Gre08, Gre10] où la taille des objets est relativement homogène.

Notre point de départ a donc été l'utilisation de l'algorithme du Sweep and Prune en essayant de proposer un nouveau modèle parallèle basé GPU indépendant de la taille des objets.

Différentes approches sont envisageables quant à la suppression de la dépendance du modèle de Le Grand sur la taille des objets. La première idée serait de cibler le problème sur la taille des cellules de la grille. Il serait ainsi envisageable d'utiliser le principe des grilles hiérarchiques introduites par Mirtich [Mir97]. Ces grilles hiérarchiques permettent de ne plus être dépendant de la taille des objets. Le principe consiste à prendre les tailles du plus petit et du plus grand volume englobant de la scène et de créer n grilles (nombre fixé à l'avance) entre ces deux valeurs. Chacune de ces grilles ayant des cases de taille fixe. Il n'y a donc plus une seule et unique grille à gérer mais n. Cette approche permet, certes, de se séparer de la dépendance envers la taille des objets mais augmente le nombre de grilles à prendre en compte. Les objets volumineux apparaissent donc dans plusieurs de ces grilles, le plus volumineux apparaissant dans chacune d'entre elles. Sachant que ces grilles doivent être mises à jour et calculées à chaque pas de temps, se défaire de la dépendance envers la taille des volumes en augmentant l'occurrence des objets dans les grilles ne semble pas être une approche performante et plus rapide.

Notre approche s'est donc basée sur le fait de ne plus utiliser de répartition spatiale de type grille pour paralléliser les calculs sur GPU, mais d'envisager un parallélisme au niveau du tri des listes représentant les axes de l'environnement et la recherche de paires similaires entre ces axes.

4.1.2 Description de l'algorithme

Notre algorithme du Sweep and Prune basé GPU est divisé en quatre parties : deux sont exécutées sur CPU et deux sur GPU. Les deux étant exécutés sur CPU étant les plus rapides et non significatives en termes de temps de calcul. L'algorithme peut être résumé comme suit :

1. Parcours parallèle et linéaire des objets pour mise à jour des AABBs (CPU)
2. Tri parallèle des bornes d'objets sur les trois axes (GPU)
3. Détection parallèle et linéaire des chevauchements sur les axes (CPU)
4. Recherche parallèle de paires communes aux trois axes (GPU)

Malgré le fait que les calculs effectués lors de la première étape sont totalement indépendants et donc, facilement parallélisable, elle est réalisée sur CPU car son temps de calcul est négligeable et croit de manière linéaire en fonction du nombre d'objets. La réaliser sur GPU impliquerait des transferts de données important, ce qui, dans notre cas, ne serait pas intéressant. La troisième étape est également effectuée sur CPU car pour déterminer les paires en chevauchement sur les axes, il est nécessaire de parcourir l'axe dans sa globalité ce qui n'est pas parallélisable. Cette étape est également très rapide et est indispensable pour faire le lien entre l'étape 2 et l'étape 4. Nous montrons dans la Section 4.1.4 que les temps de transfert CPU-GPU entre l'étape 2, 3 et 4 sont totalement négligeables comparés aux temps de calcul global des différentes étapes.

4.1.2.1 Parcours parallèle et linéaire des axes

Nous avons décrit, dans le chapitre précédent (cf. Chapitre 3 Section 3.1.2.1), l'algorithme du Sweep and Prune basé force brute. L'algorithme que nous utilisons peut être qualifié d'approche de force "semi-brute" car elle n'est ni basée sur une approche de force effectuant n^2 tests à chaque itération ni basée sur le maintien d'une structure incrémentale offrant une complexité en $nlog(n)$. Lors de la première étape, nous parcourons de façon linéaire tous les objets de l'environnement afin de récupérer les nouvelles valeurs de leur AABB respectif. Les bornes *min* et *max* de chaque objet sur les trois axes sont stockées dans une structure de type liste. Le remplissage de ces listes de bornes est effectué en parallèle. Les bornes ajoutées dans ces listes correspondent à des structures composées de trois attributs :

- L'identifiant de l'objet
- Un booléen indiquant s'il s'agit d'une borne *min* ou *max*
- La valeur numérique de la position sur l'axe

Le tableau suivant présente un exemple de listes obtenues après parcours de trois objets A, B et C d'un environnement. Les bornes *min* et *max* de chaque objet sont ajoutées au fur et à mesure du parcours des objets de la scène.

X	A_{minX}	A_{maxX}	B_{minX}	B_{maxX}	C_{minX}	C_{maxX}
Y	A_{minY}	A_{maxY}	B_{minY}	B_{maxY}	C_{minY}	C_{maxY}
Z	A_{minZ}	A_{maxZ}	B_{minZ}	B_{maxZ}	C_{minZ}	C_{maxZ}

4.1.2.2 Tri parallèle des axes

L'objectif est ensuite de trier ces listes afin d'obtenir l'ordre croissant des bornes sur les axes. Afin de réaliser cette étape de tri, nous utilisons une librairie développée par Nvidia appelée *Komrade*. Cette librairie, utilisée pour le développement d'application en CUDA, est très similaire à la librairie STL C++. Elle propose une gestion optimisée des structures vectorielles et des transferts CPU-GPU. Dans notre cas, l'intérêt majeur

de cette librairie est la présence de méthode de tri de liste sur GPU. Il existe différentes librairies permettant d'effectuer du tri parallèle sur GPU telle que *CUDPP*[1] ou *Thrust*[2].

La méthode de tri offerte par la librairie Komrade est une approche de type "*radix sort*" (tri par base). Cette méthode permet de trier une liste de clés (éléments) selon un ordre lexical avec une complexité en $O(nm)$, avec n correspondant au nombre d'éléments présents dans la liste et m à la taille moyenne des éléments. Plusieurs travaux ont montré que la technique du radix sort est la plus adaptée et la plus performante sur architecture GPU [GGKM06, SHG09]. Dans notre cas, les éléments à trier sont les positions des bornes sur leur axe respectif. Pour une bonne précision ces positions sont généralement représentées par des nombres à virgules flottantes codés sur 4 octets (32 bits). La méthode de tri utilisée manipule donc des clés de 32 bits.

Dans la continuité de l'exemple précédent, le tableau suivant présente les listes obtenues après l'étape de tri sur GPU.

X	A_{minX}	B_{minX}	A_{maxX}	B_{maxX}	C_{minX}	C_{maxX}
Y	A_{minY}	B_{minY}	C_{minY}	A_{maxY}	B_{maxY}	C_{maxY}
Z	A_{minZ}	B_{minZ}	A_{maxZ}	B_{maxZ}	C_{minZ}	C_{maxZ}

Le tri des trois listes des axes permet d'ordonner les positions des bornes sur les axes et ainsi de faire apparaitre les chevauchements entre les différents objets sur les axes.

4.1.2.3 Détection parallèle et linéaire des chevauchements

La troisième étape de l'algorithme consiste à parcourir les trois listes triées afin de créer les paires d'objets ayant les coordonnées de leurs bornes en chevauchement. La méthode permettant de déterminer si deux objets sont en chevauchement sur un axe consiste à parcourir linéairement la liste de cet axe en imitant le principe des piles. Dès que l'on rencontre une borne *min* cela signifie qu'il s'agit du début d'un objet λ_1, on conserve l'identifiant de cet objet et on poursuit notre parcours de la liste. Si la borne suivant correspond à la borne *max* du même identifiant d'objets, cela signifie que l'on a traversé l'objet λ_1 sans rencontrer d'autre borne. Cet objet ne chevauche pas d'autre objet sur cet axe. A l'inverse, si la prochaine borne rencontrée est une borne *min* d'un autre objet λ_2, cela signifie que l'on vient d'entrer dans un nouvel objet sans être sorti du premier. Les deux objets λ_1 et λ_2 sont donc potentiellement en collision car leurs bornes se chevauchent sur un axe. Dans ce cas, on crée la paire (λ_1, λ_2) en la stockant dans la liste des paires potentiellement en collision (PCS : *Potential Colliding Set*) sur cet axe. Un PCS est créé par axe. Comme pour la première étape, nous qualifions cette étape de parallèle car le calcul effectué sur chacun des axes est réalisé en parallèle. Les trois axes sont donc calculés simultanément.

Nous présentons par la suite, la continuité de l'exemple en parcourant un seul axe à la recherche de chevauchement afin de créer des paires à stocker dans le PCS.

1	X	•A_{minX}	B_{minX}	A_{maxX}	B_{maxX}	C_{minX}	C_{maxX}

1. http ://gpgpu.org/developer/cudpp
2. http ://gpgpu.org/2009/05/31/thrust

Objet courant	A
PCS	

2 | X | A_{minX} | $\bullet B_{minX}$ | A_{maxX} | B_{maxX} | C_{minX} | C_{maxX} |

Objet courant	A - B
PCS	(A, B)

3 | X | A_{minX} | B_{minX} | $\bullet A_{maxX}$ | B_{maxX} | C_{minX} | C_{maxX} |

Objet courant	B
PCS	(A, B)

4 | X | A_{minX} | B_{minX} | A_{maxX} | $\bullet B_{maxX}$ | C_{minX} | C_{maxX} |

Objet courant	
PCS	(A, B)

5 | X | A_{minX} | B_{minX} | A_{maxX} | B_{maxX} | $\bullet C_{minX}$ | C_{maxX} |

Objet courant	C
PCS	(A, B)

6 | X | A_{minX} | B_{minX} | A_{maxX} | B_{maxX} | C_{minX} | $\bullet C_{maxX}$ |

Objet courant	
PCS	(A, B)

4.1.2.4 Recherche parallèle de similitudes

La dernière étape de l'algorithme a pour objectif de déterminer les paires communes aux trois listes. Afin d'éviter un test séquentiel exhaustif en $O(n^3)$, nous utilisons le GPU pour massivement paralléliser les calculs. L'algorithme choisi est relativement simple pour bénéficier de l'architecture hautement parallèle du GPU. À partir des listes triées obtenues grâce aux étapes précédentes, on alloue , tout d'abord, trois tableaux dans la mémoire du GPU. Deux d'entre eux correspondent aux listes des axes X et Y, le troisième permet de stocker les résultats. Sa taille est celle du plus grand tableau précédent. On répète ensuite l'opération entre l'axe Z et le résultat précédent.

L'algorithme peut être résumé comme suit :

1. On récupère les trois listes de paires en chevauchement (sur X, Y ou Z).
2. Les listes de X et Y sont allouées dans la mémoire du GPU.
3. On alloue un tableau résultat de la taille maximale entre X et Y.
4. Chaque thread est en charge d'une paire de la liste maximale.
5. Les threads recherchent leur paire dans la seconde liste.
6. Le tableau résultat contient les paires communes aux deux listes.
7. On alloue ensuite la liste Z dans la mémoire du GPU.
8. On répète l'opération en comparant Z avec le résultat de l'étape précédente.

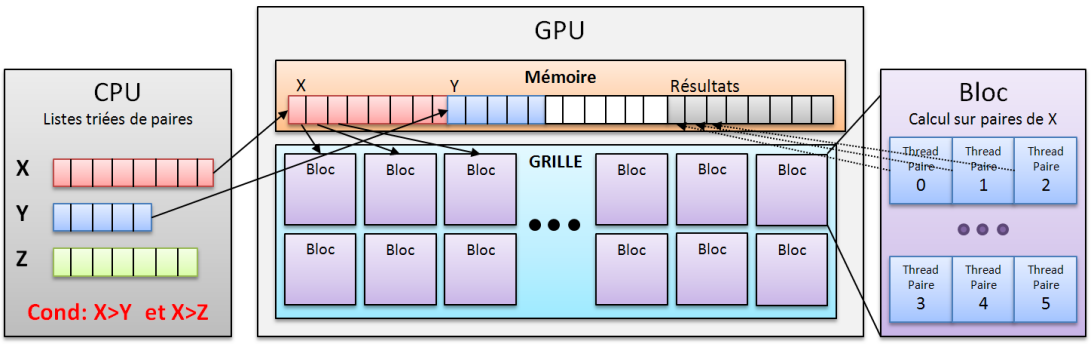


FIGURE 4.1 – Schéma global du modèle de Broad phase basé GPU.

La Figure 4.1 présente une simplification de la quatrième étape de l'algorithme. Les trois listes représentant les axes sont copiées du CPU vers le GPU, on répartit ensuite les paires à traiter entre les blocs de threads, les résultats des threads sont stockés dans la partie résultat. Il est important de noter que lors de la présence d'une conditionnelle dans le code parcouru par les threads, seuls ceux qui opèrent le même parcours sont exécutés en parallèle. Lors de la présence d'une conditionnelle de type *if-then-else*, si un thread du *warp* prend une autre branche que le reste de son groupe, tout le groupe exécute les deux branches. Cette double exécution engendre obligatoirement un coût supplémentaire et, anticiper le résultat des conditionnelles des threads afin de les regrouper par groupe serait inenvisageable vu le temps supplémentaire que cela prendrait. La séparation en deux groupes reste toutefois peu significative en termes de temps supplémentaire.

4.1.3 Implémentation GPU

Notre algorithme est réalisé en C++ et CUDA. Deux phases sont exécutées sur l'hôte (CPU) et deux autres sur le *device* (GPU). Les deux phases CPU sont des étapes couplant des étapes parallèles multi-cœur avec OpenMP et des phases séquentielles (entrées/sorties avec GPU). L'important est de limiter les interactions entre CPU et GPU pour augmenter les performances.

4.1.4 Évaluation des performances

Nous présentons dans cette section, les performances de notre nouveau modèle de Broad phase basé GPU. Nous avons comparé notre approche du Sweep and Prune sur GPU avec celle du moteur physique *Bullet* [Bul]. Nous avons testé avec un nombre croissant d'objets dans l'environnement. Le Tableau 4.1 présente les différents résultats obtenus en établissant une moyenne des temps de transfert et de calcul entre les deux étapes de calcul GPU. Pour des environnements composés d'un nombre inférieur à 100 objets, l'algorithme GPU de Bullet est plus performant. Au-delà de ce nombre d'objets, on constate que notre approche est plus performante que celle du moteur Bullet. Or, on constate également que dès que l'on dépasse les 1000 objets, les performances chutent complètement et s'éloignent fortement d'un temps interactif (30-50 Hz), ce qui

est loin d'être l'objectif recherché. Les temps de transfert du CPU vers le GPU et inversement restent cependant très faibles. Au vu des performances pour des environnements de quelques milliers d'objets nous n'avons pas effectué de simulations avec davantage d'objets.

Nb Obj	-> GPU(ms)	Noyau (ms)	-> CPU(ms)	Total(ms)	Bullet GPU(ms)
50	0,00882	0,09	0,0079	0,10672	**0,02**
100	0,01008	0,88	0,01295	0,90303	**0,16**
250	0,02369	1,783	0,05763	**1,86432**	4,47
500	0,03137	3,154	0,11587	**3,30124**	8,11
750	0,09475	12,08	0,39853	**12,57328**	17,22
1000	0,16132	56,96	0,67858	**18,7999**	29,915
2000	0,25269	143,08	1,08134	**38,41403**	61,42

TABLE 4.1 – Comparaison de notre approche du Sweep and Prune basé GPU avec celui de Bullet.

La Figure 4.2 illustre graphiquement la différence de temps de calcul des deux approches. Notre approche se révèle plus performante que celle de Bullet mais encore relativement inadapté aux véritables environnements large échelle.

Afin de déterminer la cause de cet accroissement du temps de calcul, nous présentons dans le Tableau 4.2 la répartition du temps de calcul entre les quatre étapes de notre algorithme. Nous constatons que l'étape de recherche de paires communes entre les trois listes des trois axes occupe en moyenne quasiment 90% du temps de calcul. La combinatoire de cette recherche s'avère trop importante dès qu'on atteint un nombre trop important d'objets. Cette stratégie apparait donc comme une solution efficace pour de petits environnements virtuels mais inefficace pour notre objectif du large échelle.

Nb Obj	Transferts	1)AABB	2)Tri	3)Scan Axes	4)Paires communes	Total
100	2,406%	5,28%	4,18%	0,344%	**87,78%**	100%
1000	1,448%	0,318%	6,77%	0,012%	**91,45%**	100%

TABLE 4.2 – Répartition du temps de calcul parmi les quatre étapes de notre modèle de Broad phase basé GPU selon le nombre d'objets et le temps de transfert.

4.1.5 Synthèse et perspectives

Nous avons présenté un premier modèle de l'algorithme du SaP sur architecture GPU. Notre approche se divise en quatre étapes dont les deux plus coûteuses en temps de calcul sont effectuées sur le GPU. Le fait de ne plus utiliser de subdivision spatiale de type grille a permis de se défaire de la limitation du modèle de Le Grand sur la dépendance de la taille des objets. Notre approche reste toutefois limitée en termes de nombre d'objets. La combinatoire de l'étape de recherche des paires communes aux différentes listes sur GPU reste encore trop importante. Nous avons constaté à travers

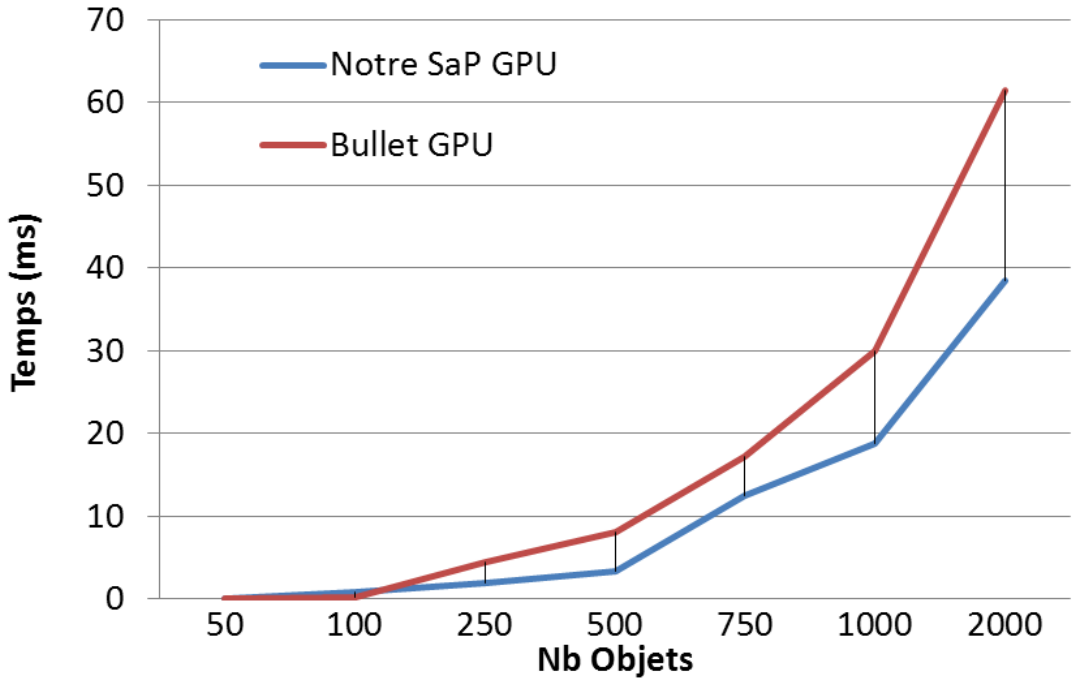


FIGURE 4.2 – Comparaison graphique de l'approche SaP basé GPU avec celle GPU de Bullet.

les résultats que cette nouvelle approche permet de réduire le temps de calcul quels que soient le type et la taille des objets simulés mais ne permet pas de simuler un très grand nombre d'objets en temps réel [AGA11a]. Notre objectif portant sur la simulation d'environnements large échelle, cette solution de recherche de similitude pour l'algorithme de Broad phase s'avère inadaptée.

Au vu des résultats peu intéressants découlant de cette approche, nous n'avons pas cherché à améliorer ou optimiser l'algorithme. Il reste cependant des voies à explorer tels qu'utiliser les librairies récentes de tri sur GPU (telles que Thrust [3]) plus rapides et plus efficaces que Komrade.

Nous présentons dans la suite de ce chapitre, un autre algorithme de Broad phase basé GPU utilisant une autre approche et, quant à lui, parfaitement adapté au large échelle.

4.2 Optimisation GPU mathématique pour approche force brute

Cette section présente une solution performante pour la détection de collision basée sur un algorithme de force brute couplé à la subdivision spatiale. Notre objectif est de démontrer qu'une algorithmique simple, adaptée à une architecture hautement parallèle,

3. http ://code.google.com/p/thrust/

apparait comme nettement plus performante que les meilleurs algorithmes actuels. La problématique est double, elle réside tout d'abord dans la simplification extrême de l'algorithme sans perte d'information, puis dans une implémentation optimisée sur GPU. Le but étant d'obtenir le plus faible temps de calcul possible tout en préservant la cohérence et précision des résultats. Nous présentons, par la suite, une introduction à la problématique (cf. Section 4.2.1), le descriptif des contraintes GPU (cf. Section 4.2.2), notre modèle basé force brute (cf. Section 4.2.3), les optimisations des transferts (cf. Section 4.2.4), le couplage avec la subdivision spatiale (cf. Section 4.2.6) et son évaluation (cf. Section 4.2.7).

4.2.1 Introduction

Dans le cas d'une simulation physique n-body, nous partons du principe qu'un algorithme dit de force brute effectue à chaque pas de temps n^2 calculs. Plus précisément il effectue $\frac{n^2-n}{2}$ (cf. Table 4.3). Les paires du type (n, n) ne sont pas testées (suppression de la diagonale de la matrice) ainsi que toutes les paires correspondant aux paires symétriques d'autres paires (x, y) et (y, x). La matrice est donc une matrice triangulaire supérieure privée de sa diagonale.

	1	2	3	4
1	x	1	2	3
2	x	x	4	5
3	x	x	x	6
4	x	x	x	x

TABLE 4.3 – Matrice des paires à tester dans un environnement composé de quatre objets : liste de 6 paires.

Le calcul effectué sur chacune de ces paires d'objets consiste à déterminer si les volumes englobants des deux objets composant une paire sont ou non en collision. Nous utilisons des volumes englobants alignés sur les trois axes de l'environnement x, y et z (AABB - cf. Section 1.3.2.4). Nous avons donc besoin, pour chaque objet, de six valeurs :

- Positions maximales sur les trois axes.
- Positions minimales sur les trois axes.

Le volume englobant d'un objet est donc représenté comme suit :

$$\begin{pmatrix} Xmin & Xmax \\ Ymin & Ymax \\ Zmin & Zmax \end{pmatrix}$$

L'algorithme utilisé pour déterminer s'il y a ou non collision entre deux volumes est l'Algorithme 2, présenté dans le Chapitre 3 Section 3.1.2.1. Cet algorithme permet de déterminer si l'une des bornes max d'un objet selon un axe est inférieure à la borne min de l'autre objet sur le même axe. Si c'est le cas, il existe un espace (de taille $max - min$) entre les deux objets. Il n'y a donc pas collision.

4.2.2 Implémentation GPU

Afin de massivement paralléliser le calcul sur GPU chaque thread est en charge d'une seule paire. Pour une simulation à n éléments, $\frac{n^2-n}{2}$ threads seront donc créés. Il faut donc fournir à chaque thread la paire d'objet qu'il a à traiter. Sachant que chaque objet est représenté par deux vecteurs (Min et Max), chacun composé de trois valeurs (x, y et z), il faut donc transmettre $2 * 2 * 3 = 12$ valeurs à chaque thread. La précision de ces valeurs étant primordiale pour assurer un résultat cohérent quant à la présence ou non d'une collision, il est indispensable que chacune de ces 12 valeurs transmises aux threads soit codées sur 4 octets (nombre flottant à virgule).

Il faut donc transmettre $12 * 4 = 48$ octets à chaque thread. Dans le cas d'une simulation à 1000 éléments, la taille du transfert sera donc de :

$$\frac{n^2-n}{2} * 48 = \frac{1000^2-1000}{2} * 48 = 499500 * 48 = 23Mo$$

N° paire	0	1	2	...	$\frac{n^2-n}{2}$
Paire	(1,2)	(1,3)	(1,4)	...	(n-1, n)

Sachant que chaque thread GPU possède un unique indice, le thread N°i s'occupe de la paire i, pour i allant de 0 à $\frac{n^2-n}{2}$).

Sachant que ce transfert CPU-GPU doit avoir lieu à chaque pas de temps, il n'est pas envisageable qu'il soit aussi important. Le principal goulet d'étranglement quant à l'obtention de bonnes performances lors d'une programmation GPU réside dans la taille du transfert de données entre le CPU et le GPU. Moins les données sont importantes et plus le gain de temps est conséquent.

Dans l'exemple précédent, chaque objet étaient transmis $n-1$ fois au GPU lors d'un pas de temps, nous proposons que chaque objet ne soit transmis qu'une seule fois. Or, il n'est désormais plus possible pour un thread de savoir quelle est la paire d'objet qu'il doit traiter. La première idée serait d'allouer dans la mémoire du GPU $\frac{n^2-n}{2}$ emplacements correspondant au indice des paires et donc des objets que les threads auraient à traiter. Travaillant sur des simulations à nombre fixe d'objets, il est possible, lors d'une phase d'initialisation pré-simulation, d'allouer en mémoire GPU cet espace pour y stocker le numéro des deux objets composant les paires.

Exemple : Appelons *memoire_paire* la zone allouée contenant des paires de numéro d'objets. Le thread N°k fait donc un accès mémoire au k^e emplacement de *memoire_paire* et lit deux valeurs (N° objet1 et N° objet2). Ces deux numéros lui permettent donc ensuite d'aller lire dans les deux emplacements mémoire $objet1$ et $objet2$ où les objets ont été stockés une seule et unique fois pour récupérer les coordonnées des volumes englobants des objets. Dans ce cas de figure, chaque thread fait donc deux accès mémoire pour savoir quels sont les objets qu'il doit traiter puis deux autres accès mémoire pour en récupérer les coordonnées des volumes englobants. Chaque thread fait donc au total quatre accès mémoire en lecture avant d'effectuer son calcul.

Afin de réduire au maximum les accès mémoire effectués par chaque thread, nous cherchons à savoir s'il est possible, connaissant le nombre d'objets n et l'indice k d'un

thread, de déterminer la paire (x, y) que le thread N°k doit traiter sans faire d'accès en mémoire.

Exemple : Simulation avec 10 objets, les paires à tester sont : (1,2),(1,3),(1,4), ... , (8,9),(8,10),(9,10) Comment déterminer uniquement grâce au calcul (sans accès mémoire) que le thread N°37 doit s'occuper de la paire (6,9) ? ou bien que le N°22 doit s'occuper de la paire (3,9) ? (cf. Table 4.4).

	1	2	3	4	5	6	7	8	9	10
1	X	0	1	2	3	4	5	6	7	8
2	X	X	9	10	11	12	13	14	15	16
3	X	X	X	17	18	19	20	21	22	23
4	X	X	X	X	24	25	26	27	28	29
5	X	X	X	X	X	30	31	32	33	34
6	X	X	X	X	X	X	35	36	37	38
7	X	X	X	X	X	X	X	39	40	41
8	X	X	X	X	X	X	X	X	42	43
9	X	X	X	X	X	X	X	X	X	44
10	X	X	X	X	X	X	X	X	X	X

TABLE 4.4 – Matrice des paires à tester dans un environnement composé de dix objets.

4.2.3 Simplification de la fonction d'accès

Nous présentons, dans cette section, une solution permettant de déterminer la paire à traiter étant donné le numéro d'identifiant du thread ainsi que le nombre d'objets. Il existe certainement d'autres moyens de parvenir à résoudre ce problème, notre approche n'est donc surement pas unique mais elle fonctionne parfaitement dans notre cas et contribue à l'obtention d'excellentes performances.

La numérotation des lignes et colonnes de la matrice des paires se fait de 1 à n (de haut en bas et de gauche à droite). On note x l'indice de la colonne et y l'indice de la ligne. Dans ce cas la première ligne contient $(n-1)$ nombres, la deuxième $(n-2)$,..., la y-ième $(n-y)$,... et la n-ième 0.

De la ligne 1 à la ligne $y-1$ il y a donc :

$$A_y = (n-1) + (n-2) + ... + (n-(y-1)) \text{ éléments}$$

Sachant que la numérotation des paires commencent à 0, A_y est donc le premier nombre de la ligne y (0, 9, 17, 24... sur la table 4.4.). A_y étant une suite, nous pouvons simplifier l'opération par :

$$A_y = (n-1) + (n-2) + ... + (n-(y-1)) = (y-1)n - (1+2+..+(y-1)) = (y-1)n - \frac{y(y-1)}{2} = \frac{(y-1)(2n-y)}{2}$$

$$A_y = \frac{(y-1)(2n-y)}{2}$$

Comme ce premier nombre est dans la colonne $y+1$, le nombre étant dans la colonne $x(>y)$ est donc $A_y+x-(y+1)$. Connaissant le numéro k du thread ainsi que le nombre total d'objets n, il faut déterminer les valeurs x et y représentant les deux objets de la paire à tester. L'inéquation à résoudre est la suivante :

$$A_y \leq k \leq A_{y+1}$$

$$\frac{(y-1)(2n-y)}{2} \leq k \leq \frac{((y+1)-1)(2n-(y+1))}{2}$$

Or, pour résoudre $A_y \leq k \leq A_{y+1}$, il n'est pas nécessaire d'utiliser les deux inégalités car il suffit de dire que :

– y est le plus grand entier tel que $A_y \leq k$
– ou bien que y est le plus petit entier tel que $k \leq A_{y+1}$

Ce qui, dans les deux cas, conduit à chercher la solution réelle de l'équation puis à prendre une partie entière. En utilisant la formule $A_y = k$, c'est à dire le cas où le numéro du thread correspond au début d'une ligne de la matrice, on obtient que :

$$\frac{(y-1)(2n-y)}{2} = k$$

et donc

$$-y^2 + y(2n+1) - 2n - 2k = 0$$

On est donc en présence d'un polynôme du second degré avec :

$$a = -1, b = 2n+1, c = -2n-2k$$

Les racines X_1 et X_2 à trouver sont donc :

$$X_1 = \frac{-(2n+1)+\sqrt{4n^2-4n+8k+1}}{-2} \; X_2 = \frac{-(2n+1)-\sqrt{4n^2-4n+8k+1}}{-2}$$

Lors de la résolution de ces racines, on constate que seulement X_1 est situé entre les bornes 1 à n. Nous allons donc résoudre X_1 afin d'obtenir la coordonnée y de la matrice. On a donc :

$$y = \left| \frac{-(2n+1)+\sqrt{4n^2-4n+8k+1}}{-2} \right|$$

Pour obtenir, la coordonnée x il suffit de résoudre :

$$x = (y+1) + k - Ay = (y+1) + k - (\frac{(y-1)(2n-y)}{2})$$

avec A_y correspondant à la première valeur de la ligne y.

Grâce à cette résolution de polynôme, il est donc possible pour chaque thread de déterminer à l'aide de n (nombre total d'objets) et de son indice k les deux objets qu'il doit traiter. Il n'est donc plus nécessaire d'allouer les paires en mémoire, ce qui fait gagner deux accès mémoires par thread et un espace assez conséquent de mémoire dans le GPU. Les seules et uniques données à transférer à chaque pas de temps sont donc les deux vecteurs Min et Max de chaque objet. Ce qui, pour une simulation de n objets représente :

$$n * 2 * 3 * 4 = (24 * n) octets$$

Dans le cas d'une simulation de 10.000 objets, la taille des données à transférer est donc de $24 * 10^4$ octets soit à peu près 234Ko. Ce qui peut se réaliser sur des architecture récentes en moins de 50 microsecondes soit 0,050 ms. Transfert quasi négligeable par rapport à notre problématique.

4.2.4 Transfert GPU-CPU

Il est également primordial d'optimiser au maximum le temps de transfert retour (GPU vers CPU) afin de récupérer les résultats calculés par le GPU. Sachant que cette récupération doit être effectuée à chaque pas de temps, plus elle sera rapide et plus le temps global de calcul sera intéressant. Chaque thread étant en charge de déterminer si deux objets sont en collision ou non, la réponse "oui" ou "non" peut être représenté sur un seul bit à 1 ou 0. Sachant que pour une simulation à n objets, il y a $\frac{n^2-n}{2}$ paires à tester, il suffit d'en allouer autant en bits. Le bit N°X valant 1 signifie donc que les objets de la paire N°X sont en collision. Dans le cadre de la programmation C++/CUDA, il est impossible de manipuler directement des paramètre ou attributs d'un seul bit. Même un booléen est représenté sur 1 octet (8bits). Nous décidons donc de travailler avec le plus petit format de données : le caractère (*char*) codé sur huit bits en allouant un bit par paire. Nous utilisons les huit bits d'un caractère pour stocker le résultat du test de collision de huit paires. Sachant qu'une allocation GPU ne peut se faire qu'en octet, le nombre d'octets nécessaires est donc de :

$nbOctet = N/8 + N\%8$ (N étant le nombre de paires et % l'opération modulo)

On alloue donc $(N/8 + N\%8)$ octets pour stocker les résultats. Lors d'un résultat positif au test de collision, un thread crée un caractère c égal à 1 et procède à un décalage binaire de $(Id\%8)$ bits au sein du caractère (Id correspondant à l'identifiant du thread). Le thread doit ensuite écrire en mémoire ce caractère à l'emplacement $(Id/8)$ de la chaine allouée. Pour cela, chaque thread récupère la valeur présente à l'emplacement $(Id/8)$ et procède à une opération de "OU" logique sur les deux chaines binaires. La lecture de cette chaine binaire transmise au CPU se fait grâce à l'opération logique : "ET".

4.2.5 Résultats préliminaires

Nous présentons les résultats préliminaires obtenus par ce modèle d'algorithme de Broad phase basé force brute. La Figure 4.3 illustre les différents environnements de test utilisés de 100 à 10.000 objet. Le tableau 4.5 illustre les différentes tailles des données en entrée et en sortie du GPU en fonction du nombre d'objets ainsi que les différents temps de transfert moyen. Le graphique illustré sur la Figure 4.4 résume l'évolution du temps noyau GPU en fonction du nombre d'objets ainsi que l'évolution de la taille des entrées/sorties du GPU. Nous constatons à travers ces résultats que les temps de transfert sont négligeables proportionnellement au temps de calcul mis par les threads au sein du noyau. Le transfert des données du CPU vers le GPU est linéairement croissant par rapport au nombre d'objets et atteint son maximum, pour nos tests, avec $53\mu s$

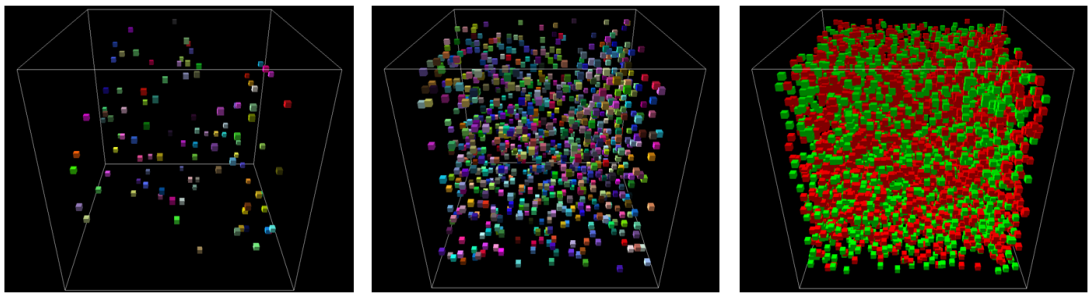


FIGURE 4.3 – Environnements de test de 100 à 10.000 objets utilisés pour évaluer l'algorithme de Broad phase GPU dans un environnement de taille (20,20,20). Les objets sont des cubes de taille 0,5.

pour 5000 objets. Celui de la récupération des résultats (GPU vers CPU) est, quant à lui, plus important mais toujours inférieur à 3ms pour des simulations de plus de 5.000 objets.

Nb Objets	Entrée	CPU->GPU	Temps Noyau	Sortie	GPU->CPU
100	2,34 Ko	8,89 μs	**0,03 ms**	0,60 Ko	7,87 μs
500	11,71 Ko	15,45 μs	**0,61 ms**	15,23 Ko	32,72 μs
1000	23,43 Ko	22,06 μs	**2,64 ms**	60,97 Ko	110,86 μs
2500	58,59 Ko	39,34 μs	**17,5 ms**	381,32 Ko	659,06 μs
5000	117,18 Ko	53,15 μs	**71,92 ms**	1525,57 Ko	2632,82 μs
10000	234,375 Ko	112,43 μs	**267,59 ms**	6102,91 Ko	7929,73 μs

TABLE 4.5 – Taille des données d'entrée et de sortie, temps de transfert et temps de calcul de l'approche basée force brute pour l'algorithme de Broad phase.

4.2.5.1 Comparaison

Pour les simulations possédant moins de 3000 objets, notre modèle dépasse les performances des approches les plus récentes. Si on compare la simulation de 2500 objets aux résultats de Liu et al. [LHLK10], on constate que notre approche est plus rapide de 20%. Leur approche est basée sur un SaP standard sur GPU couplé à un algorithme de choix du meilleur axe de projection. Comparativement à l'approche de Pabst et al. [PKS10] et Kim et al. [KHH+09] qui testent leur modèle hybride avec le benchmark n-body de l'UNC [4], notre algorithme de Broad phase est, respectivement, cinq fois et trois fois plus rapide. L'approche de Kim est basée sur un parcours d'arbre en parallèle et celle de Pabst est basée sur une subdivision spatiale uniforme couplée à un SaP de force brute non-optimisé sur les triangles.

4. http ://gamma.cs.unc.edu/

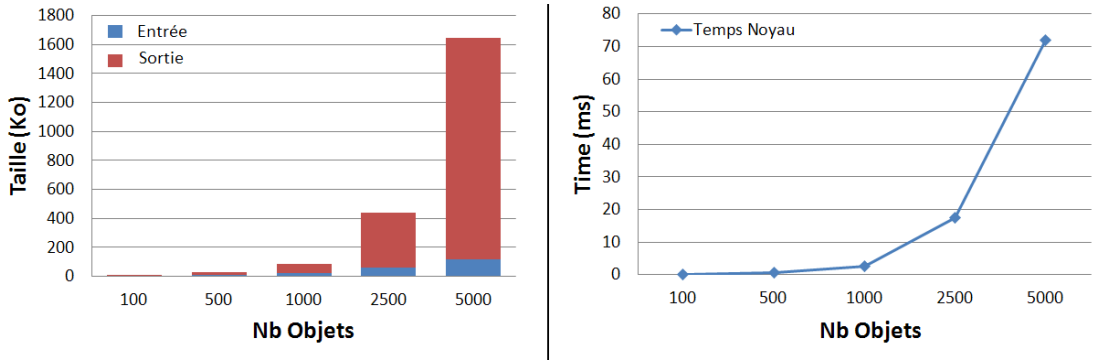


FIGURE 4.4 – Temps du noyau GPU de l'algorithme SaP force brute.

4.2.5.2 Synthèse préliminaire

L'objectif initial que nous nous étions fixé en montrant qu'une algorithmique très simpliste (force brute) correctement optimisée, pouvait s'avérer très performante, est en partie atteint. En effet, notre approche s'avère plus performante que les approches existantes pour des scènes inférieures à 3000 objets mais comme nous nous y attendions, au vu de la nature combinatoire de l'algorithme en $O(n^2)$, les résultats s'effondrent lorsque le nombre d'objets augmente. Nous allons donc couper l'effet combinatoire en couplant cette approche à une technique de subdivision spatiale. Nous allons continuer d'utiliser les optimisations proposées sur les tailles et temps de transfert ainsi que la méthode proposée permettant à chaque thread de calculer sa paire courante sans aucun accès mémoire. Notre objectif étant clairement la simulation d'environnements large échelle, il est indispensable de revoir ce modèle afin d'ajouter des conditions permettant d'élaguer le nombre de paires à tester.

4.2.6 Subdivision spatiale

Afin de réduire le temps de calcul de cette étape de Broad phase, nous couplons notre approche basée force brute à une approche de cohérence spatiale. En effet, nous proposons d'utiliser une grille 3D pour réduire le nombre de paires à calculer. Nous ne voulions initialement pas prétraiter les données de Broad phase afin de rester sur une approche où les données sont prises brutes et distribuées parmi les différentes unités de calcul. Nous avons précédemment constaté que cette stratégie s'avère payante pour de "petits" environnements mais totalement infructueuse pour de plus larges environnements. Il est donc utile de chercher à élaguer plus largement le nombre de paires candidates.

Pour cela nous mettons en place une structure de grille en 3D. Cette grille est construite au début de la simulation puis mise à jour à chaque pas de temps de simulation. Le nombre de cellules ainsi que leur tailles dépend du nombre d'objets et de leur taille. La Figure 4.5 présente l'évolution du nombre de cellules dans la grille en fonction du nombre d'objets dans la scène.

Nous détaillons, à travers cette section, les différents processus de construction (cf.

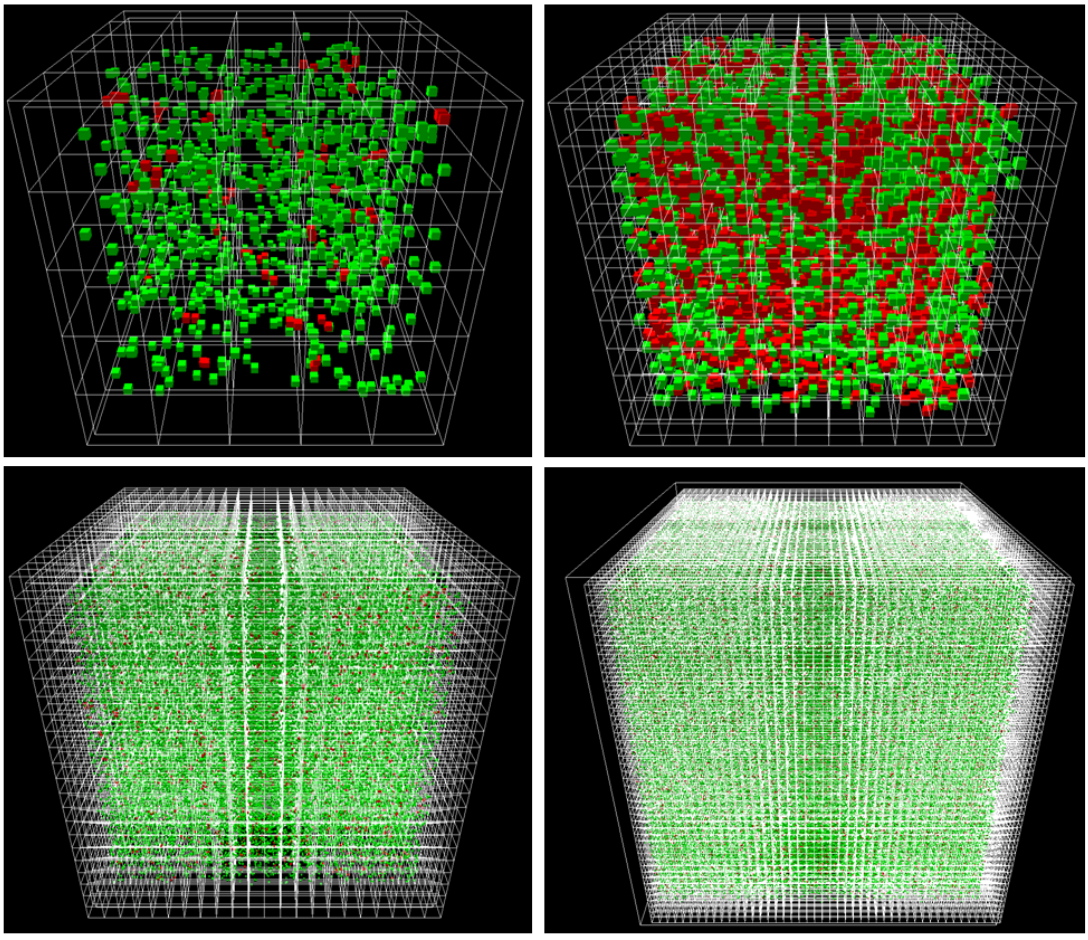


FIGURE 4.5 – Grille 3D variable en fonction du nombre d'objets dans la scène. De gauche à droite : 1000, 10K, 100K, 1M d'objets avec respectivement 125, 1331, 10648, 103823 cellules dans la grille.

Section 4.2.6.1) et de mise à jour (cf. Section 4.2.6.2) de la grille utilisée.

4.2.6.1 Construction

La construction de la grille se fait en deux étapes :
- Détermination du nombre et du volume des cellules de la grille.
- Affectation des objets aux cellules correspondantes.

Caractéristiques des cellules

Afin d'assurer une certaine homogénéité et cohérence, nous tenons compte de la densité des objets au sein de l'environnement pour déterminer les caractéristiques des cellules de la grille. Les dimensions de l'environnement et le nombre d'objets sont les deux attributs utilisés lors de la construction. Les dimensions de l'environnement sont utilisées pour déterminer les limites de la grille.

Dans un premier temps, nous procédons à l'évaluation de la densité des objets dans l'environnement. Pour cela, nous divisons le nombre d'objets par le volume cubique de l'environnement. La valeur obtenue représente la densité d'objets dans la scène. Nous fixons ensuite une valeur X afin de déterminer le volume idéal qu'une cellule doit avoir pour contenir X objets. On divise X par la densité de l'environnement pour obtenir ce volume optimal de cellule. On applique ensuite une racine cubique de ce volume pour déterminer la taille du côté d'une cellule. Dans notre approche nous utilisons une grille avec des cellules cubiques. Après avoir obtenu la dimension d'une cellule, on peut déterminer le nombre de cellules par axes et affiner la taille des cellules pour qu'elles s'ajustent parfaitement à la taille de l'environnement. La valeur X que nous fixons varie en fonction des simulations.

Remplissage de la grille

L'objectif est ensuite de lier les objets à leurs cellules respectives. Nous utilisons pour cela le GPU afin de paralléliser de façon massive les calculs. Chaque calcul consiste à déterminer dans quelle cellule de la grille un objet se situe. L'algorithme que nous utilisons est le suivant :

Algorithme 5 Calcul des identifiants de cellule des objets

```
Position de l'origine de la grille : OrigineGrille ;
Longueur du côté des cellules : TailleCellule ;
Pour Tout (Objets) Faire
  int x = (ObjetCourant.position.x - OrigineGrille.x)/ TailleCellule ;
  int y = (ObjetCourant.position.y - OrigineGrille.y)/ TailleCellule ;
  int z = (ObjetCourant.position.z - OrigineGrille.z)/ TailleCellule ;
  int id = z * NbCellulesSurXY + y * NbCellulesSurX + x ;
  grille[id].ajouter(ObjetCourant)
Fin Pour
```

Les calculs effectués pour chaque objet étant totalement indépendant des autres calculs, ils peuvent être réalisés en parallèle sur le GPU. Nous déterminons pour chaque objets la ou les cellules auxquelles ils appartiennent. L'identifiant principal de la cellule d'un objet est calculé grâce à l'algorithme présenté ci-dessus, les autres cellules voisines sont déterminées en fonction des dimensions de l'objet. A la fin de cette étape, chaque cellule de la grille est donc pourvue d'une liste d'indices d'objets. Nous créons ensuite toutes les combinaisons possibles entre ces paires pour tester si les objets sont ou non en collision. Nous réutilisons pour cela, l'algorithme de force brute présenté précédemment.

4.2.6.2 Mise à jour

Cette étape de mise à jour est une étape cruciale lors de l'utilisation de structure de type grille 3D. Le rôle de la grille étant d'élaguer plus largement les paires candidates à tester pour l'algorithme du Sweep and Prune, la mise à jour se doit donc d'être rapide. Cette étape est également réalisée sur GPU. On parcourt parallèlement tous les objets afin de prendre en compte leur nouvelle position et les affecter aux cellules correspondantes.

4.2.7 Résultats

Nous présentons dans cette section, les résultats obtenus par notre algorithme de Broad phase basé GPU utilisant un couplage entre une grille régulière 3D et l'algorithme du Sweep and Prune de force brute présenté précédemment. Dans nos environnements de tests, tous les objets sont en mouvement afin d'évaluer les performances pour les situations les plus extrêmes. Nous présentons, dans un premier temps, les valeurs numériques de nos résultats sous forme de tableau puis nous procédons à une comparaison avec les algorithmes existants.

4.2.7.1 Résultats numériques

Le Tableau 4.6 présente les différentes valeurs des temps de transfert et de calcul évalués durant différentes simulations au nombre croissant d'objets dans un environnement de taille (20, 20, 20). Tous les tests ont été effectués sur un Intel Core-Duo CPU X7900 de 2,8 Ghz avec 3 Go de Mémoire et une Quadro FX 3600M. Le tout sous Windows XP 32 bits.

La deuxième colonne des tableaux correspond à lŠétape de mise à jour des volumes englobants (ici AABB). La troisième étape indique le temps passé pour mettre la grille à jour. La quatrième colonne donne le temps nécessaire pour effectuer le Sweep and Prune sur GPU. Ce temps comprend les temps de transfert Entrée et Sortie ainsi que le temps de calcul du noyau.

On constate que lŠétape la plus consommatrice en temps de calcul est celle du Sweep and Prune sur GPU suivi par la mise à jour de la grille. Le Tableau 4.7 établit la comparaison de notre approche avec ou sans lŠutilisation de la grille 3D. LŠutilisation de la grille a un coût visible sur les petits environnements au sein desquels nous constatons quŠil nŠest pas utile de lŠutiliser. Ces petits environnements sont ceux composés de

Nb Objets	MàJ AABB	MàJ Grille	SaP GPU	Total
1K	0,03	0,13	0,09	**0,25 ms**
5K	0,16	0,74	0,19	**1,09 ms**
10K	0,31	1,45	0,59	**2,35 ms**
50K	1,67	6,17	8,78	**16,62 ms**
100K	3,38	16,12	18,41	**37,91 ms**
500K	7,02	57,45	101,02	**165,49 ms**
1M	14,26	184,91	192,72	**391,89 ms**

TABLE 4.6 – Temps de calcul de l'algorithme du Sweep and Prune GPU avec grille de calcul 3D dans un environnement de taille (20,20,20) avec des objets de tailles aléatoires de 0,01 à 1 suivant une loi de répartition uniforme.

moins de 1000 objets. Au-delà de ces 1000 objets, lŠutilisation dŠune grille 3D entraine des résultats plus performants.

Nb Objets	Sans Grille	Avec Grille
100	**0,046 ms**	1,02ms
500	**1,07 ms**	1,94 ms
1000	**2,772 ms**	3,04 ms
5000	74,60 ms	**7,88 ms**
10000	318,45 ms	**34,28 ms**

TABLE 4.7 – Comparaison de notre modèle de SaP GPU avec ou sans grille 3D.

4.2.7.2 Résultats comparatifs

Il est assez difficile de comparer nos résultats avec la littérature car il existe très peu d'approche ayant cherché à simuler plus d'un million d'objets de tailles diverses. Il est d'autant plus difficile de comparer les approches que les GPUs utilisés sont généralement différentes. Il existe plusieurs travaux étudiant les simulations basées particules avec plusieurs millions d'éléments mais cela s'éloigne de notre domaine. Les travaux de Tracy et al. [TBW09] se sont intéressés à améliorer le SaP sur CPU et leurs résultats présentent uniquement des environnements avec un faible pourcentage d'objet en mouvement. L'unique résultat qu'il indique avec 100% d'objets en mouvement est composé de 3000 Objets. Leur algorithme met 4ms tandis que le nôtre met 0,46ms (temps de transfert compris). Il y a donc un facteur de 10 entre les deux algorithmes. Des approches telles que celle de Liu et al. [LHLK10] ont testé leur approche sur un environnement composé de 960k sphères de trois tailles différentes. Cependant, ils ne donnent pas les valeurs numériques exactes de temps de calcul de leur algorithme, ce qui rend difficile la comparaison. Les seules données comparables sont celles de trois mesures de temps réalisées sur trois nombres de sphères différentes : 64K, 128K et 256K. Leur approche met respectivement 13.71 ms, 48.16 ms et 241 ms pour appliquer l'étape de SaP de

la Broad phase. Notre algorithme met respectivement 15.67 ms, 30.68 ms et 50.33 ms pour les mêmes conditions de simulation. Nous sommes donc légèrement en dessous des performances pour l'environnement de 64K sphères, 1.5 fois plus rapide pour celui des 128K sphères et cinq fois plus rapide pour le dernier.

D'autres approches telles que celle de Pabst et al. [PKS10] et Lauterbach et al. [LMM10] ne sont uniquement testées que sur de petits environnements n-body (=300 Objets). Les temps de leurs algorithmes de Broad et Narrow phase ne sont pas différenciés, seul le temps total est donné. Pour une simulation de 300 sphères et 4 cônes, l'approche de Pabst met 128,1 ms pour effectuer les calculs en mode discret et 184,8 ms en continu. Il est toujours très difficile d'évaluer le temps mis par la Broad phase dans une approche. En partant sur une Broad phase qui représenterait seulement 10%, 5%, 2% voire 1% du temps global, leur temps serait respectivement de 12.81 ms, 6.405 ms, 2.562 ms ou 1.281 ms. Si nous croisons nos résultats, notre algorithme de SaP-Grille sur GPU pour le même type d'environnement met exactement 0,41 ms. Même pour une Broad phase qui ne représenterait qu'1% de leur temps total, notre algorithme serait trois fois plus rapide. La difficulté supplémentaire de comparaison est que Lauterbach et al. ont ont effectué leurs tests sur une Nvidia GTX 285 tandis que Pabst et al. l'ont réalisé sur une Nvidia GTX 295 et nous l'avons effectué sur Quadro FX 3600M ce qui rend très difficile la comparaison.

4.2.8 Synthèse et perspectives

Le couplage entre une approche de subdivision spatiale de type grille 3D avec une approche optimisée de force brute basée sur le Sweep and Prune apparait donc comme une solution performante pour les environnements large échelle. L'optimisation logique des transferts de données a permis de réduire au minimum la taille et les temps de transfert. L'architecture hautement parallèle du GPU offre donc la possibilité aux deux principales étapes (grille et SaP) de s'exécuter en parallèle et de réduire le temps de calcul. Ce modèle est donc générique pour les architectures GPU, adaptatif car la grille évolue en fonction du nombre d'objets et plus performant que les approches récentes. Il répond donc aux différents critères établis quant à la réduction du goulet d'étranglement calculatoire de la détection de collision.

Diverses voies sont possibles quant à l'amélioration de ces algorithmes. L'étape de mise à jour de la grille est l'une des étapes les plus consommatrices de temps de calcul, il serait donc judicieux d'étudier l'impact de la taille des cellule dans la grille sur les performances. L'utilisation de cellules non cubiques est également une voie qui semble important à étudier.

4.3 Conclusion

Nous avons présenté, à travers ce chapitre, deux contributions portant sur de nouveaux algorithmes pour la Broad phase sur architecture GPU. L'approche topologique présentée dans la Section 4.1 a permis de se défaire de la dépendance à la taille des objets des approches précédentes. Elle répond bien au critère de rapidité mais reste cependant fortement limitée en termes de nombre d'objets simulables en temps interactif. Nous

avons également atteint en partie notre objectif sur le fait de montrer qu'une algorithmique simple et naïve de force brute peut, en étant correctement optimisée, conduire à des résultats génériques très intéressants et performants. Nous avons apporté une nouvelle simplification de la méthode d'accès quant à la gestion des n^2 paires permettant de réduire au maximum les allocations mémoire ainsi que les lectures et écritures faites par les threads. Les masques binaires appliqués lors de la récupération des résultats permettent également de réduire au maximum la taille des données à transférer. Comme la première approche, il reste cependant difficile d'atteindre le très large échelle avec ce type d'approche. L'ajout d'une structure de type grille 3D a permis de significativement casser la combinatoire et de réduire le temps de calcul. Les différentes comparaisons réalisées avec les travaux existants ont montré que cet algorithme était le plus performant pour des environnements large échelle. Il permet entre autre de réaliser des simulations à plusieurs millions d'objets différents en temps interactif raisonnable.

Ces trois contributions répondent donc à certains des critères essentiels pour la réduction du goulet d'étranglement de la détection de collision que sont : la généricité, l'adaptativité (grille adaptée en fonction des objets) et la rapidité (plus rapide que les algorithmes existants).

Nous présentons, dans le chapitre suivant, les évolutions de ces différents algorithmes leur permettant d'être exécutés sur des plateformes multi-CPU et multi-GPU.

Chapitre 5

Solution Hybride : Pipeline Multi-GPU/Multi-Cœur

Nous présentons, dans ce chapitre, des solutions algorithmiques pour les architectures hybrides multi-cœur et multi-GPU. L'objectif de ce chapitre est de présenter la manière dont le critère de généricité et rapidité ont été pris en compte pour permettre aux solutions multi-cœur et simple GPU, présentées précédemment, de pouvoir s'exécuter sur de plus larges architectures. Comparativement aux précédents chapitres, les algorithmes présentés peuvent être comparés à des "sur-couches" algorithmique permettant la répartition des données et des calculs. Le fait de disposer de solutions efficaces basées simple GPU et multi-cœur permet d'envisager une utilisation judicieuse de ces solutions pour profiter de la puissance des plateformes multi-GPU/multi-CPU. Nous nous sommes donc focalisés sur l'établissement d'algorithmes permettant de répartir les données et les calculs sur les différents cœurs et GPUs. Nous proposons une approche, où les deux étapes du pipeline de détection de collision peuvent s'exécuter sur un nombre quelconque de cœurs et de GPUs. La Section 5.1 présente un rappel sur le positionnement par rapport aux travaux existants et une approche préliminaire de notre modèle. La Broad phase est présentée en Section 5.2 et la Narrow phase en Section 5.3. Cette nouvelle approche de pipeline hybride répond aux critères de rapidité (réduction du temps de calcul) et de généricité (exécution sur un nombre quelconque de cœurs CPU et de GPUs).

5.1 Positionnement de l'approche

Comme décrit dans notre état de l'art (cf. Section 2.4), il existe très peu d'approches permettant l'exécution du pipeline de détection de collision sur architecture multi-CPU et multi-GPU. Les premiers à avoir proposé un tel modèle sont Kim et al. [KHH+09]. Leur modèle est basé sur une hiérarchie de volumes englobants, les cœurs CPU sont utilisés pour le parcours parallèle de la hiérarchie et les GPUs pour les tests élémentaires sur primitives. Techniquement, leur approche est censée s'exécuter sur un nombre quelconque de GPUs mais n'a, en pratique, été testée que sur deux GPUs. Les deux autres approches récentes sont celles de Pabst et al. [PKS10] et Herman et al.

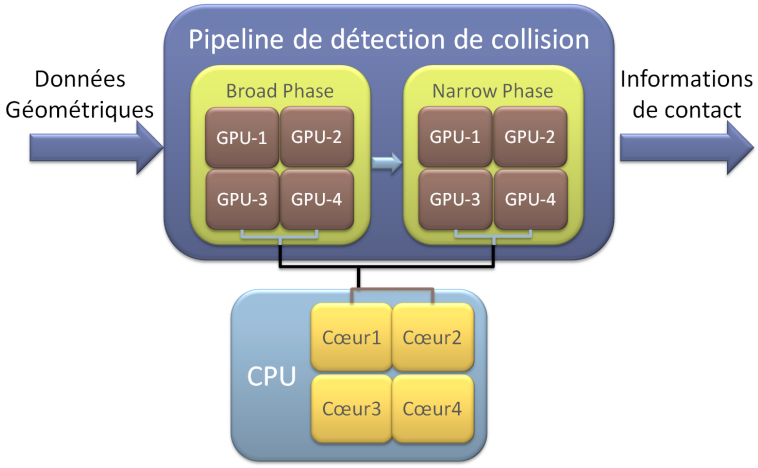


FIGURE 5.1 – Schéma global de notre modèle hybride basé multi-cœur et multi-GPU.

[HRF+10]. Pabst et al. se basent sur la subdivision spatiale et des tests d'intersections principalement basés sur des sphères pour détecter les collisions. Les calculs de sphères sont effectués sur GPU ainsi que les tests sommet/triangle et arête/arête. Herman et al. proposent une méthode basée sur une combinaison entre le partitionnement spatial et le vol de données pour répartir les calculs sur les différents cœurs CPU et GPU. Ils ont testé leur approche sur des architectures allant jusqu'à quatre bi-GPU (8 GPUs) en atteignant parfois un gain proche de sept en comparant avec l'utilisation d'un seul GPU.

Les principaux moteurs physiques existants tels que Bullet [Bul], PhysX [Phy] et Havok [Hav] n'utilisent pas, pour le moment, d'approche multi-GPU. Ils peuvent parfois s'exécuter sur de telles architectures mais leur stratégie consiste à utiliser les différents GPUs pour différentes tâches, telles qu'un GPU en charge du rendu et l'autre de la physique. Ils commencent de plus en plus à s'intéresser aux multi-cœur et les dernières librairies disponibles font état de plusieurs algorithmes physiques portés et adaptés aux architectures multi-cœur. Il n'y a donc pas, pour le moment, de réelle avancée en matière de modèles multi-GPU orientés calcul physique.

Notre méthode réside donc dans le fait de proposer un nouveau pipeline hybride pour la détection de collision. Ce pipeline se doit de respecter différents critères tels que la rapidité du temps de calcul et la généricité des architectures d'exécution (nombre de cœurs CPU et nombre de GPUs). Notre objectif est d'utiliser ce modèle hybride pour la simulation d'environnements large échelle, ce qui pour le moment n'a pas encore été réalisé.

La Figure 5.1 présente une vision simplifiée de notre pipeline hybride. Les deux étapes du pipeline sont exécutées sur un nombre quelconque de GPUs (ici quatre), gérées chacune par un thread sur un cœur CPU. Tous les threads sont exécutés en parallèle pendant les étapes de Broad et Narrow phases afin d'effectuer simultanément tous les calculs sur chacun des GPUs. Pour s'assurer de la stratégie de l'exécution d'un thread par GPU, nous avons testé différents scénarios mettant en scène la gestion des

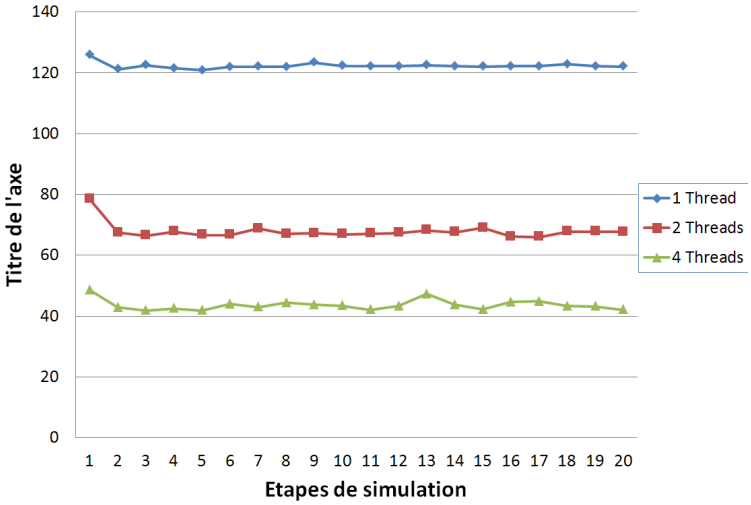


FIGURE 5.2 – Comparaison d'une plateforme munie de 4-GPUs gérée par 1, 2 ou 4 threads. Nous avons mesuré les dix premiers pas de temps de simulation pour cette comparaison.

GPUs par un nombre différent de threads CPU. La Figure 5.2 présente une comparaison d'une plateforme composée de 4 GPUs gérée, par 1, 2 ou 4 threads. Le test a été réalisé sur un ®Intel Xeon de 3,2 GHz (processeur bi-quad cœur), chaque thread est exécuté sur un cœur séparé. Le test de temps de calcul a été réalisé durant dix pas de simulations de l'étape de Broad phase dans un environnement composé de 20.000 cubes (cf. Figure 5.4). On constate que le temps optimal est obtenu en utilisant un thread par GPU disponible. Techniquement, ce travail a été réalisé en couplant des algorithmes développés en CUDA et C++ avec OpenMP pour fournir une parallélisation globale sur multi-GPU et multi-cœur.

Nous détaillons, par la suite, les deux étapes de notre pipeline hybride que sont la Broad phase (cf. Section 5.2) et la Narrow phase (cf. Section 5.3) suivies des résultats comparatifs (cf. Section 5.4).

5.2 Broad Phase

Le point de départ choisi pour élaborer notre pipeline hybride est notre algorithme de Broad phase GPU présenté dans le Chapitre 4 Section 4.1. Cet algorithme permet d'effectuer les calculs sur simple GPU. Nous étions arrivés à la conclusion que ce modèle simple GPU permettait de réduire le temps de calcul de l'étape de Broad phase mais restait encore trop coûteux en temps de calcul pour simuler de larges environnements. Nous proposons donc un nouvel algorithme permettant de répartir les calculs sur un nombre quelconque de GPUs. L'objectif étant de montrer, qu'exécuté sur une architecture multi-GPU, cet algorithme est plus performant pour les environnements large échelle. L'objectif sous-jacent, de porter cet algorithme sur multi-GPU, est de voir si

l'utilisation d'algorithmiques plus simples et plus facilement parallélisables permet de tirer plus de profit de ces architectures hautement parallèles.

5.2.1 Broad phase topologique GPU

Nous rappelons dans cette section les différentes étapes de l'algorithme que nous avons proposés dans le Chapitre 4 Section 4.1. Les étapes sont les suivantes :

1. Parcours parallèle et linéaire des objets pour mise à jour des AABBs (CPU)
2. Tri parallèle des bornes d'objets sur les trois axes (GPU)
3. Détection parallèle et linéaire des chevauchements sur les axes (CPU)
4. Recherche parallèle de paires communes aux trois axes (GPU)

Notre modèle permet d'exécuter les étapes 2 et 4 sur des architectures multi-GPU et les étapes 1 et 3 sur architectures multi-cœur. L'étape de tri a été réalisée en répartissant, lorsque cela était possible, le tri d'un axe sur un GPU. Dans le cas d'une architecture bi-GPU, un des GPUs effectue le tri de deux axes de l'environnement. À l'inverse, sur une architecture à 4 GPUs, seuls 3 GPUs font effectuer l'étape de tri. Nous présentons par la suite, l'étape 4, la plus coûteuse en termes de temps de calcul, adaptée pour exécution multi-GPU.

5.2.2 Répartition spatiale des données

Après avoir adapté l'algorithme du SaP sur simple GPU, nous présentons notre modèle permettant de l'adapter sur une architecture multi-GPU. Pour séparer les calculs entre les différents GPUs, pendant l'étape de Broad phase, nous utilisons une technique de subdivision spatiale dynamique et plus précisément on divise l'espace de l'environnement par le nombre de GPUs. La technique de subdivision utilisée n'est pas de type régulière comme les grilles ou les octree mais, comme la complexité de l'algorithme ne dépend que du nombre d'objets dans la scène, on peut subdiviser l'environnement en utilisant la densité des objets. Chaque GPU se retrouve ainsi avec le même nombre d'objets à calculer ce qui permet d'équilibrer le temps de calcul a priori de manière homogène entre les GPUs.

La Figure 5.3 présente la technique que nous proposons pour subdiviser l'environnement afin de répartir les calculs entre les GPUs. Nous vérifions parmi les axes celui qui possède le plus de paires qui se chevauchent, puis on le divise par le nombre de GPUs. Chaque GPU effectue donc désormais l'étape 4 de recherche de paires communes dans son propre sous-ensemble de données. Comme décrit dans la Section 5.1, chaque GPU est géré par un thread sur un cœur afin de fournir une parallélisation globale sur multi-GPU et multi-cœur. Un seul thread est créé par cœur et est en charge d'une partie de l'environnement et de son GPU qui exécute l'algorithme de Broad phase. Les collisions entre deux parties de l'environnement ne peuvent pas se produire car la subdivision est faite avec des paires qui se chevauchent et non pas avec des objets. Si un objet A est à la frontière entre deux subdivisions et qu'il est en chevauchement avec un objet B d'un côté et un objet C de l'autre, il apparaitra dans les listes de deux GPUs sous la forme de la paire (A, B) et (A, C). À la fin de l'étape de Broad phase multi-GPU

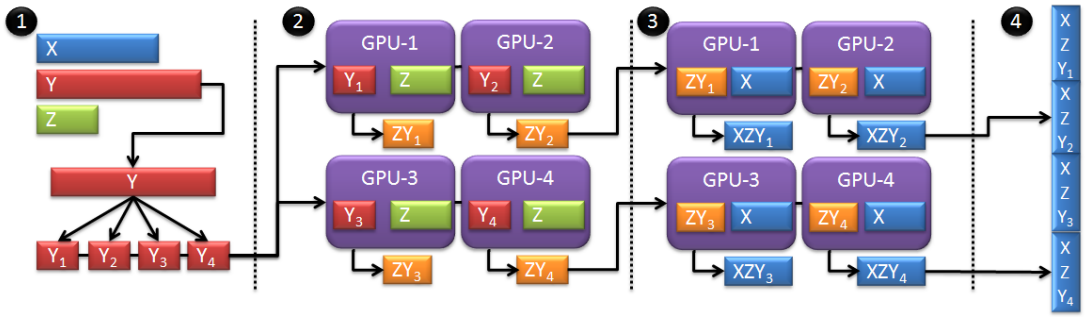


FIGURE 5.3 – **(1)** On crée trois listes de bornes min et max sur le CPU (X, Y et Z), la plus longue est sélectionnée pour être divisée par le nombre de GPUs. **(2)** Une partie de la liste divisée et celle d'un autre axe sont transférées dans la mémoire des GPUs (ici l'axe X et Y). L'algorithme du SaP GPU est ensuite appliqué sur chaque GPU. Pour chaque paire de la liste la plus longue un thread est créé et recherche une paire similaire dans l'autre liste. **(3)** On répète les calculs pour comparer l'axe restant avec le résultat précédent (ici l'axe Z). **(4)** On synchronise tous les résultats des GPUs afin de construire la liste globale des paires en chevauchement.

nous synchronisons tous les résultats des GPUs pour créer la liste des paires d'objets en potentielle collision pour la transmettre à la Narrow phase.

5.2.3 Mesures de performances

Nous présentons, dans cette section, les résultats obtenus par notre approche hybride de Broad phase. Nous l'avons testée avec différents scénarios allant d'environnements aux objets similaires à des scènes très hétérogènes. Les environnements de tests sont illustrés sur la Figure 5.4 et leurs propriétés géométriques justifiant leur utilisation sont détaillées dans la Table 5.1. Notre objectif est de tester notre approche avec des simulations de type n-body d'objets rigides. Tous les objets des environnements sont dynamiques, seuls le sol et le bol (pour l'environnement de l'alphabet) sont statiques. Les tests sont effectués sur une plateforme de 4 GPUs (2 Nvidia Quadro Plex composé chacun de 2 Nvidia Quadro FX 5800). Ces GPUs sont composés de 4 Go de mémoire et de 240 cœurs à 1,3 GHz. Le CPU en charge de ces GPUs est un Intel Xeon 5482 de 3,2 GHz sur Windows XP (64 bits) avec 64 Go de RAM. Ce processeur est un double quad-cœur avec un cache L1 de 32 Ko et L2 de 6144 Ko partagés par les cœurs sur chacun des deux processeurs. Nous présentons, par la suite, les valeurs numériques des temps de calcul obtenus suivies par une comparaison avec d'autres approches.

La Figure 5.5 montre les mesures de performance de l'étape de Broad phase avec l'environnement des 10K sphères. Nous avons reproduit quatre fois la même simulation, mais avec quatre algorithmes différents de CPU séquentiel à 1, 2 ou 4 GPU(s). Nous pouvons voir sur le graphe que les algorithmes ont les mêmes perturbations en temps de calculs tout au long de la simulation. Ces perturbations sont liées à l'évolution de la simulation et plus précisément aux variations du nombre de collisions entre les objets. La ligne horizontale au début de chaque courbe représente la chute des sphères avant

Modèles	Nb Objets	Nb Polygones	Propriétés
Cubes	20k	240k	Simple et convexe
Sphères	10k	10.8M	Complexe convexe
Tores	1k	3.4M	Similaire et non-convexe
Alphabet	2.6k	7.3M	Tous différents

TABLE 5.1 – Propriétés des quatre environnements de test. Leurs différentes propriétés couvrent un large choix de possibilités géométriques et numériques dans le domaine des objets rigides.

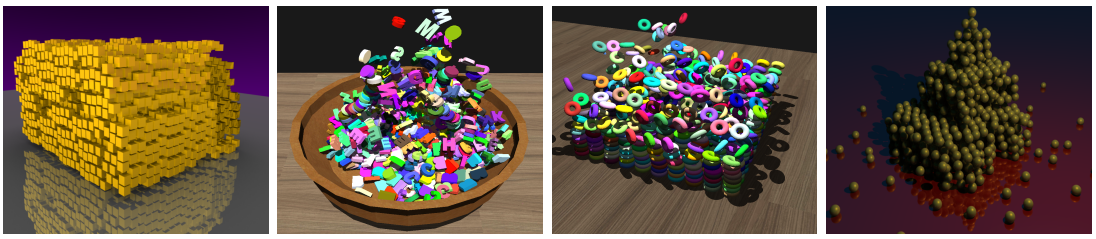


FIGURE 5.4 – Environnements de tests utilisés pour évaluer notre pipeline hybride. Les tests sont des simulations n-body d'objets rigides. De gauche à droite : 20K cubes, 2.6K lettres de l'alphabet, 1K tores et 10K sphères.

de tomber sur le plan. On constate que l'utilisation de 4 GPUs est 11 fois plus rapide que le CPU séquentiel et 3 fois plus performante qu'un seul GPU.

Le graphique à gauche de la Figure 5.6 présente la différence en pourcentage de temps entre l'exécution CPU et celle utilisant 1, 2 ou 4 GPU(s). On constate que le plus faible gain est obtenu pour l'environnement des tores qui n'est effectivement composé que de 1000 objets. La différence de temps de calcul entre l'algorithme CPU et GPU est plus faible. Ceci s'explique par la rapidité que met le processeur à calculer ces 1000 objets. Plus le nombre d'objets est important, plus le gain est également important. Par exemple, pour le premier environnement avec les cubes, l'utilisation d'un GPU réduit par 4,2 le temps de calcul, par 7,5 avec 2 GPUs et de 14,6 pour 4 GPUs. Tandis que pour le troisième environnement des tores, le gain est de 2,5 pour 1 GPU, de 3,6 pour 2 GPUs et de 4,3 pour 4 GPUs. Le peu d'objet explique à nouveau le peu d'avantage à utiliser 4 GPUs pour calculer l'étape de Broad phase sur 1000 objets. Les résultats montrent que l'utilisation d'un GPU réduit significativement le temps de calcul au cours du processus de Broad phase dans les environnements composés de plusieurs milliers d'objets.

Le graphique à droite de la Figure 5.6 présente le pourcentage de réduction du temps de calcul de l'utilisation de 2 et 4 GPUs par rapport à un seul GPU. Nous avons mesuré le temps passé par les trois algorithmes (1, 2 et 4 GPU(s)) sur les quatre environnements de tests. Comme pour le premier graphique, on constate que plus le nombre d'objets est élevé, plus le gain est important (cf. Cubes Vs. Tores). Les résultats montrent que la solution multi-GPU est parfaitement adaptée pour ce genre d'algorithme hautement

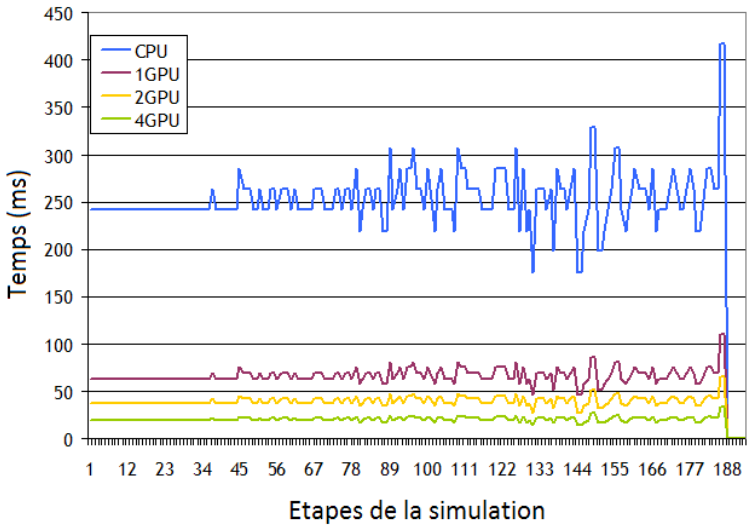


FIGURE 5.5 – Comparaison de l'algorithme de Broad phase sur 1, 2 ou 4 GPU(s) et CPU sur l'environnement composé de 10K sphères.

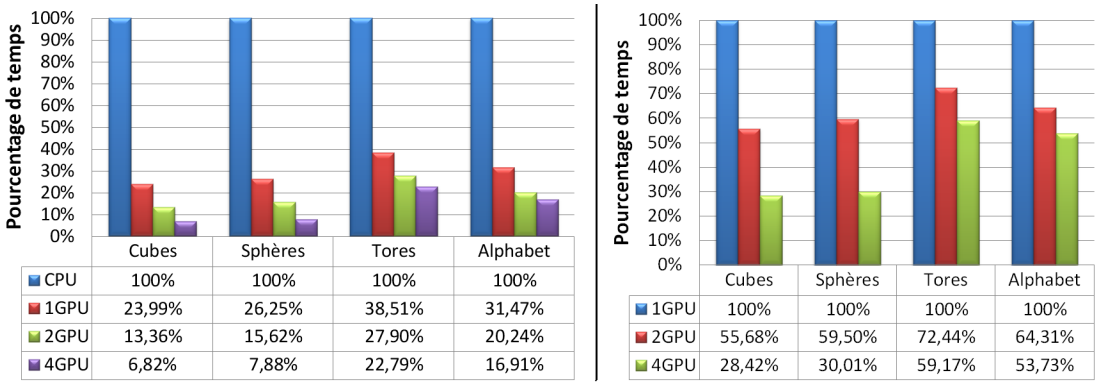


FIGURE 5.6 – **Gauche** : Comparaison de l'algorithme de Broad phase entre l'exécution sur CPU et l'utilisation de 1, 2 ou 4 GPU(s) sur les quatre environnements de test. **Droite** : Comparaison de l'utilisation de 2 ou 4 GPUs par rapport à l'utilisation d'un seul GPU.

parallélisables et réduit le temps de calcul sur les architectures composées de 2 ou 4 GPUs. Les résultats montrent également que l'utilisation du plus grand nombre de GPUs disponibles n'assure pas obligatoirement un gain aussi important que l'on pourrait l'espérer lors de l'utilisation d'environnements composés de peu d'objets.

5.3 Narrow Phase

Nous détaillons dans cette section notre approche hybride pour l'étape de Narrow phase. Nous avons choisi d'utiliser une approche basée image car cette famille d'algorithme utilise les capacités graphiques du GPU pour détecter les collisions. Les GPUs étant initialement conçues pour gérer le graphisme, utiliser les propriétés graphiques des cartes semble donc être une solution pour l'obtention de bonnes performances. Les approches basées images sont souvent utilisées pour la détection de collision et permettent d'obtenir de bonnes performances [FBAF08, AFC+10]. Il n'existe pour le moment aucune approche permettant d'utiliser ces algorithmes sur des architectures multi-GPU afin de détecter les collisions au sein de larges environnements virtuels. L'algorithme que nous utilisons pour cet étape est un algorithme basé image utilisant la technique du *Depth Peeling*. Cet algorithme parfaitement adapté à l'architecture GPU n'a jamais été porté sur multi-GPU. Il offre un compromis rapidité/précision permettant de choisir entre une vitesse de calcul très élevée, couplée à une précision faible ou une vitesse plus lente, due à une précision supérieure des résultats. Notre approche implémente une nouvelle technique d'équilibrage de charge entre les GPUs. Cette méthode permet, grâce des règles, d'uniformiser le temps de calcul entre les cartes quelques soient les objets à traiter.

5.3.1 Depth Peeling

Nous avons, dans un premier temps, développé un algorithme simple GPU implémentant la technique du Depth Peeling pour détecter les points de collision. Il ne nécessite pas de structure de données particulière ni de prétraitement. Il fonctionne avec des paires d'objets transmises par la Broad phase et considérées comme étant potentiellement en collision. Leurs surfaces sont rendues dans une texture selon un axe de vue choisi. La Figure 5.7 illustre l'algorithme en deux dimensions pour deux objets en chevauchement (cercle vert et carré bleu). Les polygones sont pixellisés (convertis d'une image vectorielle à une image de pixels) dans deux directions orthogonales et stockés dans plusieurs images (calques) afin de représenter la totalité de la géométrie. Chaque paire est composée de deux objets préalablement filtrés par la Broad phase et d'une boîte englobante où la collision pourrait survenir. L'algorithme de Depth Peeling est ensuite effectué sur cette boîte de sélection pour détecter la collision entre les objets.

Plus la résolution de l'image est grande, plus l'information de contact est précise mais plus le temps de calcul est long. Une résolution plus élevée est essentielle pour modéliser des contacts très précis. Pour l'utiliser avec des environnements large échelle, nous avons décidé d'éviter d'avoir à calculer des contacts très précis. Pour massivement paralléliser les calculs de paires sur le GPU, nous divisons le buffer image de 512x512 en sous-parties de taille égale. Cette division du buffer permet de calculer les collisions sur

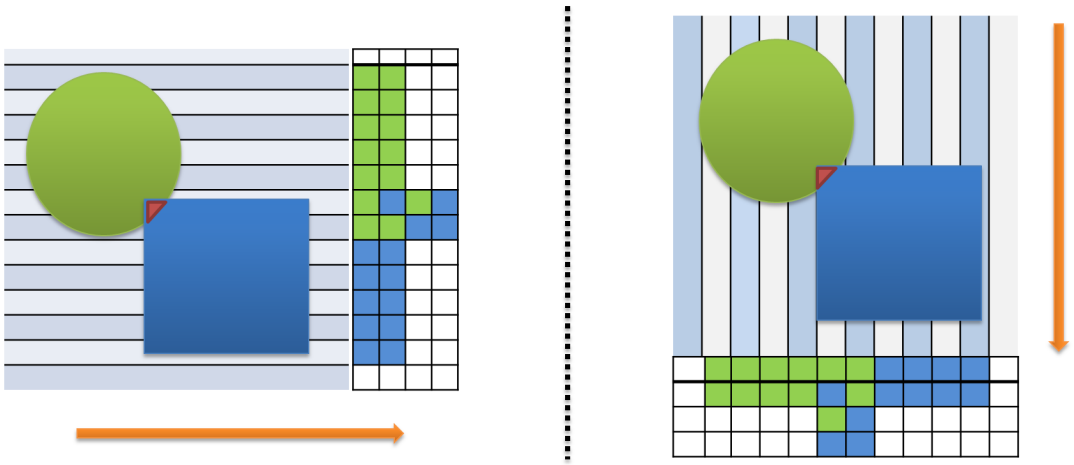


FIGURE 5.7 – Exemple 2D de l'algorithme du Depth Peeling utilisé pour détecter les collisions entre deux objets (cercle vert et carré bleu), la zone d'intersection est indiquée en rouge. **Gauche** : rasterization des surfaces selon l'axe horizontal. **Droite** : rasterization selon la direction verticale.

plusieurs paires d'objets en parallèle. Pour nos tests, nous avons divisé le buffer image en petits carrés de résolution 32x32, permettant ainsi de calculer en parallèle 256 paires d'objets. Cette faible résolution empêche l'obtention d'informations précises de contact mais détecte efficacement quand une collision se produit. Un système dynamique qui permettrait d'adapter la résolution et le nombre d'objets pourrait être envisageable et très utile.

5.3.2 Équilibrage de charge

L'objectif est de trouver une solution à l'utilisation conjointe de l'algorithme précédent et de l'architecture multi-GPU. Comment est-il possible de séparer les calculs entre les GPUs disponibles pendant le processus de Narrow phase ? Un premier élément de réponse est d'utiliser trois GPUs pour les calculs d'orientation différente. Il y a trois orientations orthogonales différentes à calculer (x, y et z) durant l'algorithme de Depth Peeling. Cette solution semble efficace en termes de réduction de temps de calcul mais tout à fait inappropriée pour plus de trois GPUs. Pour exploiter pleinement la puissance du GPU, nous avons besoin d'un algorithme permettant de séparer de manière équilibrée les calculs entre les GPUs. Nous savons que le temps de calcul du Depth Peeling est linéaire en fonction du nombre d'objets et du nombre de polygones existants dans les objets. Nous proposons d'utiliser une distribution dynamique des paires d'objets de la liste. Nous commençons en divisant la liste de paires par le nombre de GPUs, car nous ne connaissons pas la densité des objets et des polygones dans l'environnement. Le maintien d'une carte de densité de polygones au cours de la simulation serait très coûteux en termes de temps de calcul. Ainsi, chaque GPU commence par sa propre

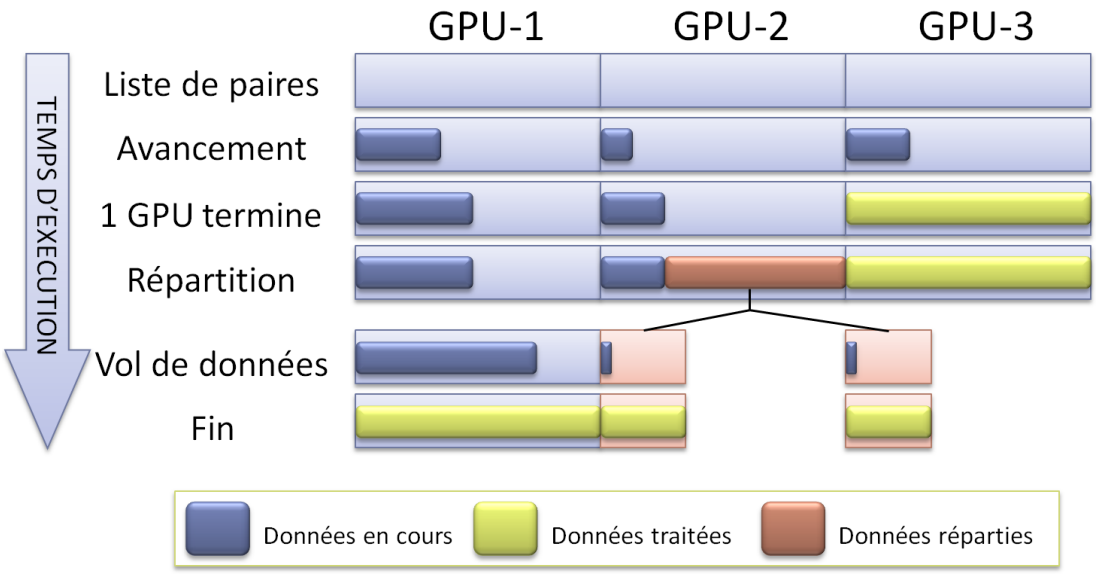


FIGURE 5.8 – Exemple de scénario de vol de données GPU durant l'exécution de 3 GPUs. La liste des paires est divisée en trois parties égales, transmises à chaque GPU. Quand un GPU termine le calcul de son ensemble de données avant les autres, la moitié des données restantes du plus occupé est divisée en deux parties égales et partagées entre les deux GPUs.

sous-partie de la liste. Le premier GPU qui termine, continue en déchargeant un autre GPU afin de terminer sa propre partie.

Cette technique pourrait être comparée à une méthode de vol de données car le GPU plus rapide "vole" des données au plus occupé. Elle peut aussi être assimilée à une technique d'équilibrage de charge [HBD96, KS95, KGR94]. Les Figures 5.8 et 5.9 illustrent un scénario possible de vol des données GPU sur une plate-forme composée de 3 GPUs. Techniquement, chaque GPU dispose de son propre index qui indique l'évolution des calculs sur son propre ensemble de données. Quand un calcul est effectué pour une paire d'objets, le thread en charge du GPU augmente l'indice de données. Pour obtenir de meilleures performances et comme pour la Broad phase, la gestion des GPUs est également gérée en parallèle sur architecture multi-cœur.

5.3.3 Mesures de performances

La Figure 5.10 montre le temps de calcul au cours du processus de Narrow phase sur les mêmes tests présentés précédemment (cf. Figure 5.4). Comme l'algorithme du Depth Peeling est une solution basée image, nous ne comparons pas avec le temps CPU, mais entre les différents temps passés par les trois algorithmes (1, 2 et 4 GPU(s)). Nous remarquons que le temps de calcul diminue de manière significative lorsque le nombre de GPUs augmente. Quel que soit l'environnement, le temps est en moyenne réduit de 1,64 en passant de 1 GPU à 2 GPUs et de 3,22 en passant à 4 GPUs. L'environnement

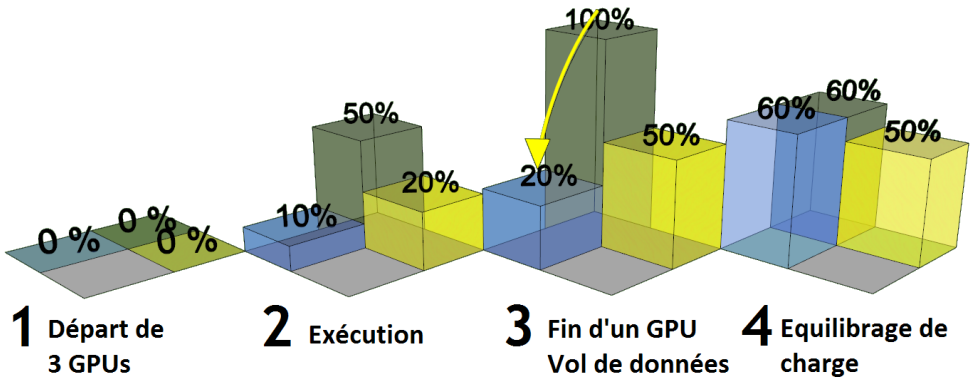


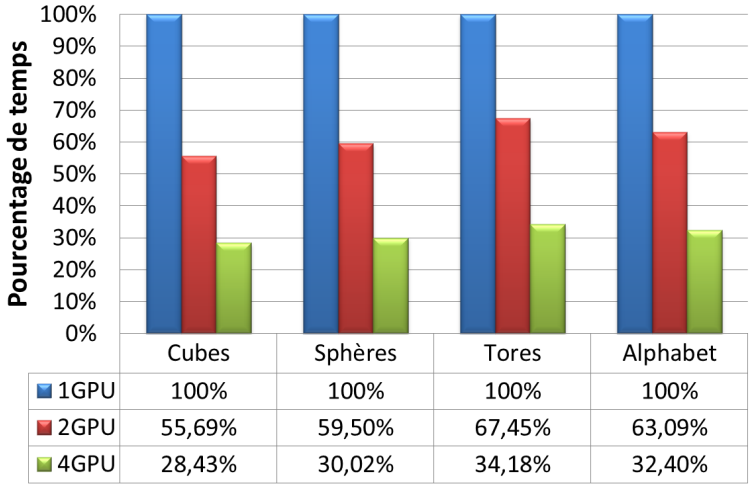
FIGURE 5.9 – Technique d'équilibrage de charge utilisée pendant le processus d'exécution GPU afin de maintenir un temps homogène de calcul entre les GPUs.

obtenant les meilleurs performances est celui des cubes, le temps est réduit de 1,80 sur 2 GPUs et de 3,52 sur 4 GPUs. Le grand nombre de paires d'objets à traiter et la présence de maillages très simples expliquent ces bonnes performances. Les données qu'un GPU vole à l'autre ne représentent pas beaucoup de temps de transfert. Tandis que sur l'environnement des tores, le gain n'est que de 1,48 sur 2 GPUs et de 2,93 sur 4 GPUs. Cela est dû au maillage complexe que possèdent les tores car lorsqu'un GPU a terminé ses calculs de paires et qu'il vole des données à un autre GPU, les données à transférer via le bus PCI sont volumineuses et le temps est important. Pour cette raison, il serait intéressant d'évaluer l'impact de la taille des données à transférer sur la quantité optimale que devrait voler un autre GPU pour obtenir un temps de calcul homogène entre les GPUs.

Les résultats montrent que notre solution multi-GPU est bien adaptée pour l'algorithme du Depth Peeling pour le processus de Narrow phase. Elle réduit le temps de calcul sur 2 et 4 GPUs, dans le meilleur des cas, le temps est divisé par 1,80 sur 2 GPUs et par 3,52 sur 4 GPUs.

5.4 Résultats comparatifs

Notre pipeline hybride détection de collision est implémenté pour s'exécuter sur un nombre quelconque de GPUs et de cœurs CPU. Nous avons montré que l'utilisation optimale d'une plateforme multi-GPU est la gestion, en amont, d'un GPU par un thread CPU s'exécutant sur un cœur. Partant de ce constat, notre pipeline hybride offre une gestion optimisée des GPUs par le processeur principal. Il reprend les deux étapes (Broad et Narrow) présentées précédemment en ajoutant une sur-couche multi-GPU / multi-cœur. Le principe est donc de coupler une méthode topologique (algorithme de Broad phase) à un algorithme optimisé pour GPU (algorithme Depth Peeling de Narrow phase). Les résultats montrent que ce nouveau pipeline hybride de détection de collision réduit significativement le temps de calcul.



	Cubes	Sphères	Tores	Alphabet
1GPU	100%	100%	100%	100%
2GPU	55,69%	59,50%	67,45%	63,09%
4GPU	28,43%	30,02%	34,18%	32,40%

FIGURE 5.10 – Comparaison de l'algorithme de Narrow phase entre 1, 2 et 4 GPU(s) (rapport en % à un GPU).

	Broad phase	Narrow phase	Total
Bullet	127.45	2415.03	2542.48
Our work	22.40	120.069	142.47

TABLE 5.2 – Comparaison de notre pipeline hybride avec le moteur physique Bullet. Nous mesurons le temps (ms) pour calculer la Broad et la Narrow phase dans l'environnement de l'alphabet.

Nous avons testé l'algorithme avec quatre environnements de tests différents illustrés sur la Figure 5.4. Leurs propriétés géométriques sont résumées dans la Tableau 5.1. Les résultats montrent que l'architecture multi-GPU peut être utilisée pour détecter des collisions et plus précisément que la Narrow phase et la Broad phase peuvent fonctionner sur ce type de plate-forme en réduisant les temps de calcul. Les résultats montrent que cette solution adaptée pour le multi-GPU et multi-cœur a de meilleures performances. Elle réduit le temps de calcul de près de 3,5-4X en passant d'un seul GPU à 4 GPU en grande échelle des environnements virtuels.

Nous avons comparé les résultats obtenus par notre nouveau pipeline hybride avec celui du moteur physique Bullet. Par souci d'égalité, nous utilisons les algorithmes GPU de Bullet développés en CUDA pour le comparer avec notre pipeline multi-GPU. La Table 5.2 présente le résultat obtenu en comparant les deux pipelines sur un environnement composé de 2600 objets représentant les 26 lettres de l'alphabet.

Nous avons également comparé notre pipeline hybride à celui de Kim et al. [KHH+09] sur l'environnement de type n-body. Cet environnement de test existe en deux qualités, une basse résolution sur laquelle leur pipeline hybride met 6,8 ms à calculer les collisions. Et une haute résolution sur laquelle leur approche met 53,8 ms. Nous n'avons pas

Nb GPU(s)	Notre algo	Kim [KHH+09]
1 GPU	63,47 ms	80 ms
2 GPUs	37,76 ms	54 ms
4 GPUs	19,07	non-testé

TABLE 5.3 – Comparaison de notre pipeline hybride avec celui de Kim [KHH+09].

comparé notre pipeline avec l'environnement de basse résolution car il s'éloigne trop de la qualité attendue pour les environnements large échelle.

Le Tableau 5.3 présente une rapide comparaison de notre pipeline avec celui de Kim et al. [KHH+09]. Notre algorithme hybride met 63,47 ms sur 1 GPU, 37,76 ms sur 2 GPUs et 19,07 ms sur 4 GPUs. Leur algorithme n'est testé que sur deux GPUs. Sur deux GPUs, notre algorithme est 1,42 fois plus rapide et 2,82 fois sur quatre GPUs. Le modèle de GPU utilisé par Kim est une Nvidia Geforce GTX 285 est le notre une Nvidia Quadro Fx 3600m.

5.5 Synthèse et perspectives

Nous avons présenté, à travers ce chapitre, une contribution globale portant sur la proposition d'un nouveau pipeline hybride de pipeline de détection de collision. Ce pipeline hybride est composé de deux nouvelles contributions axées sur le portage d'algorithmes de détection de collision sur des architectures multi-GPU. La première contribution est l'adaptation de notre algorithme de Broad phase GPU du Sweep and Prune sur multi-GPU en utilisant une technique de répartition spatiale des données selon les axes. La seconde contribution consiste en l'utilisation d'un algorithme basé image, implémentant la méthode du Depth Peeling, adapté aux architectures multi-GPU grâce à l'utilisation d'une technique d'équilibrage de charges basée sur le vol de données. Ce modèle hybride est le premier pipeline de détection de collision où les algorithmes sont entièrement exécutés sur multi-GPU. Ces deux algorithmes n'ont jamais été adaptés aux plateformes multi-GPU et multi-cœur. Les résultats montrent que ce nouveau pipeline de collision parallèle réduit le temps de calcul au sein des environnements virtuels large échelle. La gestion de l'ensemble des GPUs disponibles est également optimisée en parallélisant les threads sur processeur multi-cœur. Ce pipeline hybride répond donc aux critères de généricité et de rapidité.

Il serait désormais intéressant d'évaluer ce nouveau pipeline de détection de collision sur des architectures multi-GPU plus larges. Nous souhaiterions également l'évaluer sur des environnements de grande taille avec plusieurs centaines de milliers voire millions d'objets avec 8, 16, 32 ou plus de GPUs. Les résultats suggèrent une multitude d'orientations futures. Étudier les différentes techniques de répartition pour distribuer les données et les tâches entre les GPUs pourrait permettre de déterminer laquelle est la plus appropriée pour les plateformes multi-GPU. De nombreuses améliorations sont possibles lors de l'étape de Narrow phase pour la répartition des tâches et des données.

Toutes nos approches (multi-cœur, GPU et hybride), présentées précédemment, répondaient aux critères de généricité et de rapidité décrits lors de notre bilan de l'état de l'art. Nous présentons, dans la suite de ce manuscrit, nos travaux sur les critères de structuration et d'adaptativité pour réduire le goulet d'étranglement de la détection de collision.

Chapitre 6

Pipeline et Algorithmique Dynamique

Ce chapitre présente nos travaux sur les critères de structuration et d'adaptativité des solutions parallèles pour la détection de collision. Nous présentons, dans un premier temps, nos travaux sur la structuration du pipeline (cf. Section 6.1). Nous avons intégré la prise en compte de l'architecture d'exécution dans le pipeline et nous en présentons une version parallèle fonctionnelle munie d'une technique dynamique d'équilibrage du parallélisme. Nous présentons ensuite nos travaux sur l'adaptation algorithmique dynamique (cf. Section 6.2). Nous proposons une approche basée sur des simulations hors-ligne permettant de définir les domaines de performance des algorithmes afin d'adapter durant la simulation l'algorithme en fonction des conditions de simulation. Les critères de rapidité et de généricité sont également respectés et illustrés dans les deux sections de résultats (cf. Section 6.1.4 et 6.2.3).

6.1 Pipeline : Structuration Parallèle et Dynamique

Cette section est consacré à nos travaux sur le critère de structuration parallèle du pipeline de détection de collision. Lors de l'état de l'art, nous avons établi que ce critère était essentiel à prendre en compte quant à la réduction du goulet d'étranglement calculatoire. En effet, depuis Hubbard [Hub93, Hub95] le pipeline est séquentiel et généralement composé de deux étapes (parfois plus), qui possèdent une précision et un temps de calcul croissants. Depuis sa création, le pipeline est l'unique stratégie adoptée par tous les simulateurs physiques car il permet de réduire significativement le temps de calcul. Nous présentons, par la suite, nos travaux sur la structure du pipeline (cf. Section 6.1.1.1) avec, dans un premier temps, la prise en compte de l'architecture d'exécution [AGA09a] suivi, par notre modèle de pipeline parallèle [AGA10b]. Ce pipeline rompt avec la structure conventionnelle du pipeline séquentiel de détection de collision. Il offre la possibilité de bénéficier de deux niveaux de parallélisme en intégrant une première parallélisation globale de ses principales étapes (cf. Section 6.1.1) et un second niveau de parallélisme interne des algorithmes adapté par une gestion dynamique des threads lors de la simulation sur multi-cœur (cf. Section 6.1.2 et 6.1.3). Ce nouveau pipeline

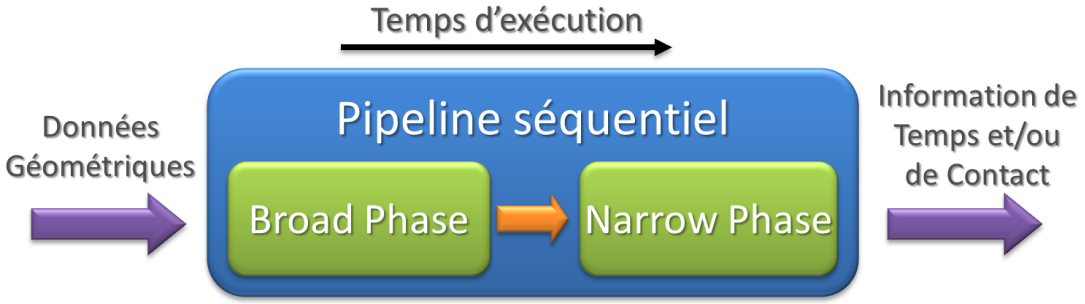


FIGURE 6.1 – Pipeline séquentiel de détection de collision [Hub93].

parallèle répond au critère de structuration tout en prenant en compte celui d'adaptativité (adaptation dynamique durant la simulation), de généricité (exécution possible sur un nombre quelconque de cœurs) et de rapidité (réduction significative du temps de calcul).

6.1.1 Nouvelle structure de pipeline

Le modèle de pipeline de détection de collision utilisé par tous les simulateurs physiques ne prend pas en compte l'évolution matérielle des architectures. Une représentation de ce pipeline est visible sur la Figure 6.1. Ce pipeline, dit séquentiel, montre la succession des différentes étapes et les algorithmes appliqués. Nous présentons, dans un premier temps, un nouveau concept permettant la prise en compte de l'architecture d'exécution dans le pipeline (cf. Section 6.1.1.1) suivi par le descriptif du premier niveau de parallélisme de notre modèle du pipeline parallèle (cf. Section 6.1.1.2).

6.1.1.1 Ajout de la dimension architecture

La nouvelle dimension ajoutée dans le pipeline permet la prise en compte de l'architecture d'exécution. Ce pipeline offre une nouvelle manière de représentation illustrant, à la fois, la problématique à laquelle est confrontée la communauté cherchant à améliorer les performances de la détection de collision, et la solution conceptuelle pour parvenir à la résoudre. Cette nouvelle dimension matérielle permet donc d'illustrer la couche d'exécution des algorithmes utilisés.

La figure 6.2 illustre cette nouvelle représentation du pipeline. Nous voyons un pipeline composé de deux étapes (Broad et Narrow) exécutées simultanément sur une architecture différente. Cette représentation du pipeline considère désormais l'architecture d'exécution comme essentielle à l'obtention de meilleures performances pour les algorithmes de détection de collision. La nouvelle dimension tient ici compte du nombre de CPUs et de GPUs ainsi que du nombre de cœurs présents au sein de ces unités de calcul pour répartir les données et les calculs entre les unités de calcul et paralléliser en interne les différentes étapes.

Nous proposons une exécution parallèle des deux phases du pipeline. Comme l'illustre

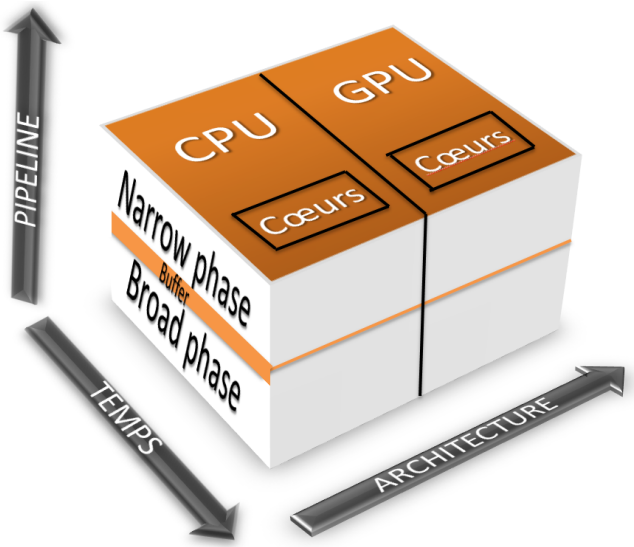


FIGURE 6.2 – Représentation du nouveau pipeline 3D de détection de collision

le pipeline 3D, on voit les deux étapes s'exécuter en parallèle et liée par une structure de type buffer permettant de partager leur résultats. Il existe tout de même un léger décalage entre les deux débuts de phase. Ce décalage est dû au fait que l'on ne peut vraisemblablement pas démarrer l'étape de Narrow phase avant que la Broad phase n'ait commencé à effectuer ses calculs.

6.1.1.2 Rupture de la séquentialité

Bien que la notion de pipeline soit intrinsinquement liée à un mode de parallélisme, le pipeline de détection de collision n'a jamais été parallélisé. Zachmann [Zac01] en a fait une évaluation théorique d'une exécution simultanée des phases du pipeline et a montré que si la densité de l'environnement est importante par rapport au nombre de processeurs, un gain significatif en termes de vitesse peut être constaté.

Une façon d'améliorer le pipeline de détection de collision est de revoir l'imbrication des phases entre elles. En d'autres termes, comment est-il possible de réagencer les étapes du pipeline pour en améliorer la performance ? Le pipeline est actuellement séquentiel et est conçu de la façon suivante : les données géométriques de l'environnement sont transmises en entrée et passent par une succession séquentielle de filtres ayant pour rôle de raffiner la précision du calcul. En sortie, on obtient les informations de contact et/ou de temps calculées entre les objets en collision. Nous proposons une exécution en parallèle de ces phases où la Broad phase et la Narrow phase sont exécutées simultanément lors d'une simulation. À l'inverse du pipeline séquentiel classique, l'exécution des deux phases est parallèle. Les principaux problèmes lors de la rupture de la séquentialité d'un algorithme sont, tout d'abord, d'assurer qu'il n'y a aucune perte de calculs et donc que les résultats sont toujours cohérents. Dans notre cas, nous gardons la séquentialité du processus fait sur une paire d'objets mais nous rompons le processus

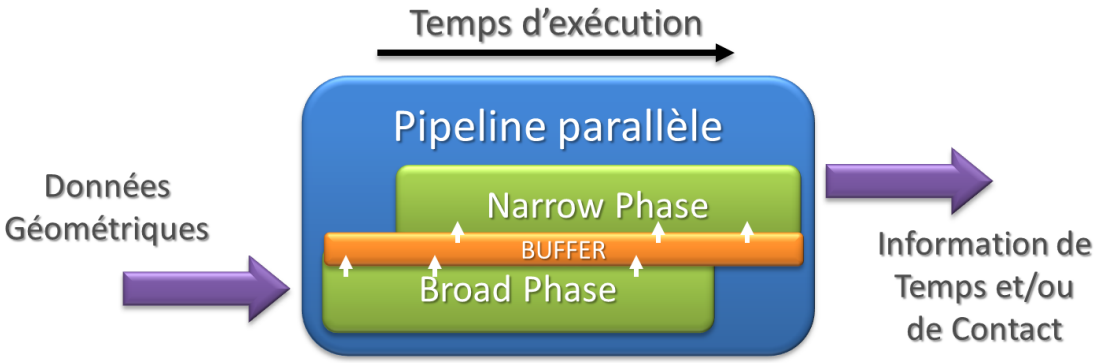


FIGURE 6.3 – Notre modèle de pipeline parallèle de détection de collision.

global séquentiel de l'ensemble des paires. En d'autres termes, une paire d'objets est d'abord utilisée par la Broad phase pour déterminer rapidement s'il existe des risques de collision. Si tel est le cas, la paire est transmise à la Narrow phase pour déterminer les points de contact précis. Le pipeline séquentiel attend la fin de la Broad phase pour commencer la Narrow phase.

Les deux phases sont liées par un buffer dans lequel la Broad phase stocke le résultat du traitement de ses données. La Narrow phase les utilise pour poursuivre la détection de collision. Il s'agit d'un modèle producteur-consommateur. Ce modèle est particulièrement adapté à notre contexte multi-thread. La Figure 6.3 présente ce nouveau pipeline parallèle et les différences avec celui séquentiel. Au lieu de commencer par exécuter la Broad phase puis la Narrow phase, deux threads maitres sont créés et exécutés en parallèle sur deux cœurs. Dès qu'une paire a été traitée par la Broad phase et considérée comme étant en potentielle collision, elle est stockée dans le buffer en prévision d'être utilisée par la Narrow phase. Le thread de la Narrow phase commence à la fin du traitement de la première paire de la Broad phase. Le buffer est un tableau de paires d'objets où la Broad phase est seulement autorisée à mettre le résultat du filtrage de ses paires et la Narrow phase à récupérer des données afin de poursuivre les calculs. Sur une architecture équipée d'un plus grand nombre de cœurs, les deux threads maitres (Broad et Narrow) sont en charge de la création et de la gestion de plusieurs threads fils. Le modèle développé est comparable à un modèle producteur/consommateur où les phases sont en exécution perpétuelle et se transmettent leur résultat via un *buffer* (mémoire tampon) partagé par les threads maitres de chaque phase. Cette nouvelle disposition du pipeline réduit en moyenne de 30 à 40% le temps de calcul (cf. Section 6.1.4.1).

6.1.2 Équilibrage dynamique

Après avoir décrit le premier niveau de parallélisme de notre pipeline consistant en l'exécution parallèle des deux étapes de Broad et Narrow phase, nous présentons, dans cette section, le deuxième niveau de parallélisme. Ce second niveau intervient lors de l'exécution du pipeline sur des architectures multi-cœur possédant plus de cœurs que de phases dans le pipeline. Le premier niveau de parallélisme, présenté précédemment,

est adapté à une architecture possédant un nombre de cœurs égal au nombre d'étapes dans le pipeline avec, par exemple, un thread en charge de la Broad phase et l'autre de la Narrow phase. Chacun d'entre eux fonctionnant sur un cœur différent. La question est de déterminer comment est-il possible de répartir les threads sur 4, 8 ou n cœurs ? Est-il préférable de répartir équitablement les ressources entre les différentes phases ou doit-on favoriser une phase plus que l'autre ? Nous présentons, par la suite, deux exemples illustrant la problématique du choix de parallélisme suivis de l'approche, que nous proposons, basée sur l'utilisation d'une méthode d'équilibrage de charge modifiant dynamiquement la répartition du parallélisme. Cette méthode est chargée de déterminer quelle étape est la plus coûteuse en temps de calcul de façon à lui donner plus de threads pour s'exécuter. Nous utilisons, pour cela, des algorithmes parallélisés sur multi-cœur que nous présentons dans la Section 6.1.3.

6.1.2.1 Exemples de temps de calcul

Nous présentons deux exemples illustrant les temps de calcul évolutifs des étapes du pipeline.

Exemple 1 : Prenons une simulation avec plusieurs millions d'objets en chute libre vers un sol. Durant la chute, les objets ne se touchent pas, ils sont relativement séparés les uns des autres. Dans ce cas, la Broad phase doit vérifier tous les objets à chaque pas de temps de la simulation tandis que la Narrow phase n'a aucun calcul à effectuer.

Exemple 2 : Prenons, cette fois, une simulation composée d'une dizaine d'objets, tous en collision les uns avec les autres, où chacun des objets est composé de milliers de polygones. Dans ce cas, le peu d'objets justifie à lui seul le temps de calcul négligeable de la Broad phase, tandis que le très grand nombre de polygones engendre un temps de calcul nettement plus important pour la Narrow phase.

Ces deux exemples opposés illustrent que le temps de calcul peut être différent entre les deux étapes durant une simulation. Ces deux scénarios peuvent également se retrouver dans une seule et même simulation, ce qui renforce le fait d'utiliser une technique d'adaptation dynamique. Les deux exemples précédents font également valoir le fait qu'il est difficile de séparer les threads entre les étapes sans savoir quel type de scénario possède la simulation et comment il peut évoluer.

6.1.2.2 Étape de contrôle dynamique

Nous proposons d'utiliser une technique dynamique d'équilibrage de parallélisme qui adapte la répartition des threads lors de la simulation. Cette répartition se fait grâce aux mesures des temps de calcul de la Broad et de la Narrow phase. Si une des phases se révèle nettement plus longue en termes de temps de calcul, on lui alloue plus de threads pour son exécution parallèle. Sachant qu'analyser le temps passé par les étapes à chaque pas de temps de simulation est coûteux, nous proposons d'utiliser une fréquence variable des contrôles. La fréquence est élevée au début de la simulation puis elle évolue en fonction du scénario de la simulation. La fréquence élevée est utilisée, au début, pour appréhender rapidement le comportement et les performances du scénario par rapport à l'architecture d'exécution. La fréquence est ensuite adaptée en fonction des changements opérés dans la répartition des threads. Lorsqu'un choix de nouvelle répartition

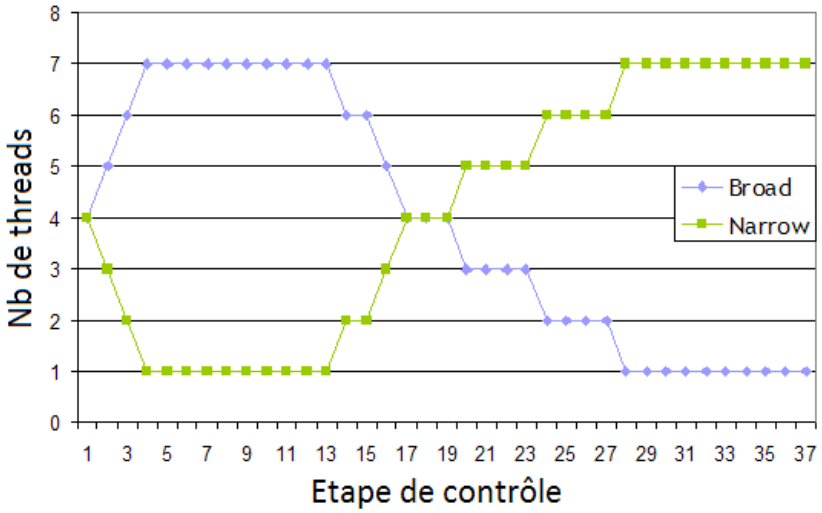


FIGURE 6.4 – Exemple de notre modèle de répartition dynamique de threads entre la Broad et la Narrow phase. Les deux phases commencent avec quatre threads. Au début, les objets tombent sur le sol sans aucun contact entre eux. Nous adaptons ensuite la répartition en fonction du temps de calcul.

est effectué, nous devons assurer que le temps passé par cette nouvelle répartition est meilleur que le précédent. Pour ce faire, après chaque changement de répartition, un contrôle du temps des phases est réalisé et la fréquence des contrôles est augmentée. À l'inverse, tant que la configuration reste inchangée la fréquence diminue jusqu'à un seuil à fixer par l'utilisateur.

La Figure 6.4 illustre un exemple de répartition dynamique des threads lors d'une simulation. Ce test a été effectué sur une architecture bi-quad-cœur (Intel Xeon 5482). Chaque phase commence avec quatre threads correspondant à la moitié du nombre de cœurs disponibles. Nous mesurons le temps passé par les phases avec quatre threads et comme la Broad phase est plus longue que la Narrow phase, un thread de la Broad phase est transmis à la Narrow phase. Au début, tous les objets tombent sur le sol sans contact, la Narrow phase n'a donc pas de calcul à effectuer. Autour du 18^e pas de contrôle, on constate une inversion dans la répartition des threads. Cette inversion signifie que les objets sont maintenant en contact avec le sol et donc que la Narrow phase a plus de calculs de collisions à effectuer. Son temps de calcul s'accroit et dépasse celui de la Broad phase, les threads de la Broad phase lui sont donc transmis au fur et à mesure que la condition est vérifiée.

Avec cette nouvelle technique d'équilibrage dynamique du nombre de threads durant le processus de détection de collision, nous sommes en mesure de fournir un temps de calcul proche de l'optimal tout au long de la simulation. Les exemples montrés dans la section précédente (cf. Section 6.1.2.1) peuvent être différenciés et traités de manière différente. Si ces deux exemples de scénarios sont dans la même simulation, nous sommes en mesure d'adapter dynamiquement la répartition du parallélisme pour

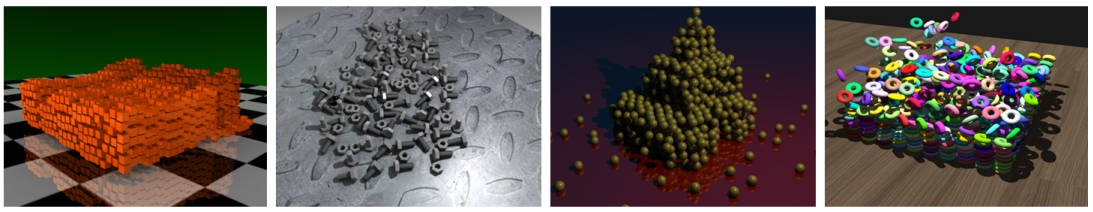


FIGURE 6.5 – Les quatre environnements de test utilisés pour évaluer notre modèle de pipeline parallèle.

faire face à l'évolution des scénarios. Dans le premier exemple, nous avons montré que notre technique d'équilibrage favorise une parallélisation plus massive de la Broad phase tandis que la Narrow phase sera favorisée dans le deuxième exemple.

6.1.3 Algorithmiques parallèles

Le pipeline est techniquement composé de deux phases. Pour pleinement bénéficier du parallélisme de l'architecture multi-cœur lors du second niveau de parallélisme de notre modèle, les deux étapes du pipeline sont parallélisées sur processeur multi-cœur.

6.1.3.1 Broad phase

Afin de paralléliser massivement l'algorithme de Broad phase, nous avons choisi l'approche parallèle semi-brute du SaP présentée précédemment (cf. Section 3.1). Cette technique ne met pas à jour une structure interne mais reprend à zéro à chaque pas de temps. L'algorithme consiste en la projection des bornes minimales et maximales des objets sur les trois axes de l'environnement et en la recherche de chevauchements entre ces coordonnées. Après projection sur les axes, il y a $\frac{(n^2-n)}{2}$ paires d'objets à calculer. Cette algorithmique est très bien adaptée au parallélisme grâce à l'indépendance des calculs entre eux. Ils peuvent être distribués sur un nombre quelconque de cœurs.

6.1.3.2 Narrow phase

Comme tous les objets dans une simulation peuvent être différents en termes de propriétés géométriques, nous utilisons notre approche de répartiteur parallèle présentée précédemment (cf. Section 3.2.1). Ce répartiteur est en charge de trouver et d'appliquer l'algorithme le plus approprié à une paire d'objets. La sélection est effectuée grâce à des classes d'objets définies comme des objets simples (cubes, sphères ...) ou complexes et convexes ou non-convexes. Ce répartiteur fonctionne en prenant l'ensemble des paires d'objets préalablement filtré par la Broad phase et applique l'algorithme le plus approprié. Son exécution peut être massivement parallélisée par le fractionnement et la distribution du tableau des paires d'objets entre les différents cœurs.

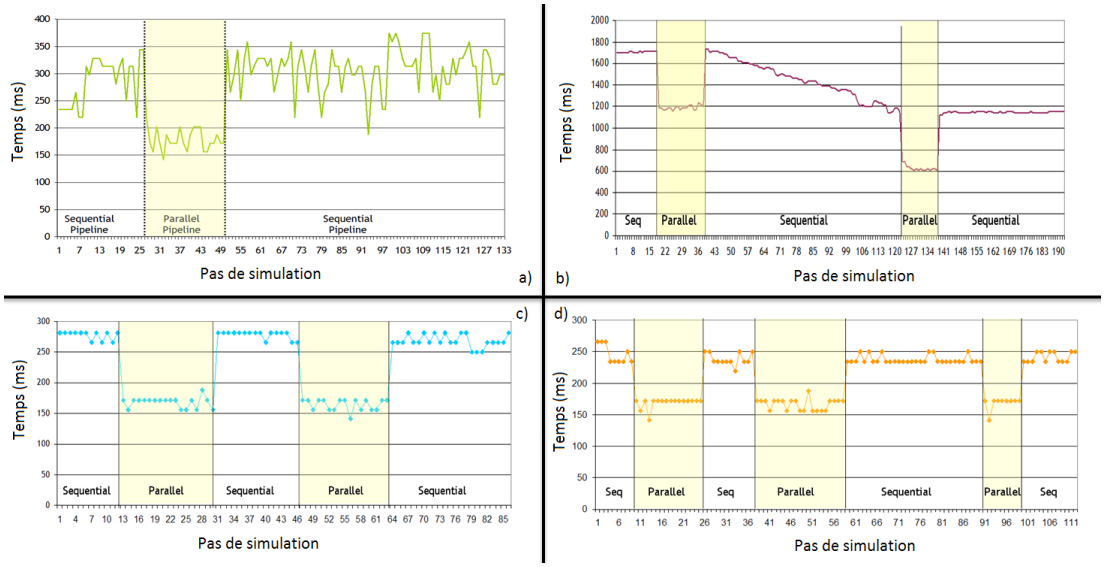


FIGURE 6.6 – Quatre exemples illustrant la différence entre le pipeline séquentiel et parallèle utilisés successivement durant des simulations. De haut en bas et de gauche à de droite : **a :** 500 vis et boulons dans un bol, **b :** 2000 tores dans un environnement conique, **c :** 3000 cubes sur un plan, **d :** 2500 sphères sur un plan.

6.1.4 Résultats et discussion

Nous avons testé notre nouveau pipeline parallèle de détection de collision avec différents scénarios de simulation, en passant par des objets semblables totalement indépendants à des scènes hétérogènes d'objets en collision (cubes, sphères, tores, vis et boulons). Les environnements utilisés sont illustrés sur la Figure 6.5. Dans la suite, nous présentons dans un premier temps les résultats de notre pipeline de détection de collision parallèle sur une architecture double-cœur (cf. Section 6.1.4.1), puis nous présentons les résultats de notre nouvelle méthode d'équilibrage du parallélisme sur une architecture composée de huit cœurs (cf. Section 6.1.4.2).

6.1.4.1 Résultats : pipeline parallèle

Les tests ont été effectués sur un processeur Intel Core-2 X7900 de 2,8 GHz sur Windows XP avec 3 Go de RAM. Nous comparons l'utilisation du pipeline de détection de collision séquentiel et notre nouveau modèle parallèle. Les algorithmes utilisés sont rigoureusement identiques entre les deux pipelines, la différence réside dans l'exécution simultanée ou consécutive des deux étapes. La Figure 6.6 présente les temps de calcul du pipeline tout au long des simulations durant lesquelles nous inversons le modèle de pipeline utilisé. Les différents scénarios de simulation que nous avons utilisé sont :

- a) 500 vis et boulons dynamiques dans un bol statique
- b) 2000 tores dynamiques dans un environnement de cônes statiques
- c) 3000 cubes dynamiques sur un plan statique

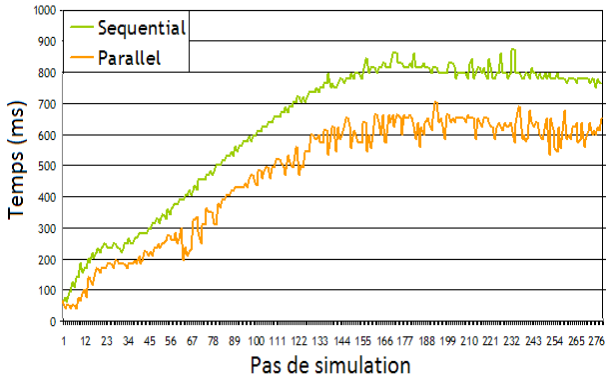


FIGURE 6.7 – Comparaison entre le temps de calcul du pipeline séquentiel de détection de collision et notre nouveau modèle parallèle. Le test a été réalisé avec 1000 Tores tombant sur un plan.

– d) 2500 sphères dynamique sur un plan statique

Ces quatre différentes simulations permettent de couvrir une palette de propriétés géométriques comme convexe ou non-convexe, très peu ou plusieurs milliers de polygones, dynamiques ou statiques, simples ou complexes. Par exemple, un tore est un objet non-convexe composé, dans notre cas, de 1152 polygones tandis qu'un cube est un objet convexe composé uniquement de 12 polygones.

Le premier graphique en haut à gauche de la figure 6.6 montre les résultats de la simulation faite avec des vis et des boulons et nous remarquons que le pipeline parallèle permet de réduire le temps de calcul de près de 40%. Sur la droite, les résultats de la simulation faite avec les tores illustre une réduction de 30% et une seconde de 40%. Une réduction du temps significative de plus de 40% est également constatée pour la simulation avec 3000 cubes dynamiques et de moins de 30% pour les sphères. Le temps de calcul est donc, en moyenne, réduit de 30 à 40% pour tous les environnements de test utilisés.

Une comparaison globale de l'exécution parallèle et séquentielle du pipeline est représentée sur la Figure 6.7. Le temps de calcul est mesuré à chaque pas de simulation. Nous constatons que quel que soit l'étape de simulation, le pipeline parallèle est plus performant que celui séquentiel.

Nous avons montré que dans tous les environnements de test, le nouveau modèle de pipeline parallèle de détection de collision réduit significativement le temps de calcul. Sur une architecture double cœur ce nouveau pipeline fournit de meilleures performances que la solution séquentielle existante.

6.1.4.2 Résultats : équilibrage dynamique

Nous présentons, dans cette section, les résultats de notre nouvelle méthode de répartition dynamique des threads. Nous l'utilisons sur une architecture composée de plus de deux cœurs afin de pleinement exploiter le parallélisme des cœurs. Les tests

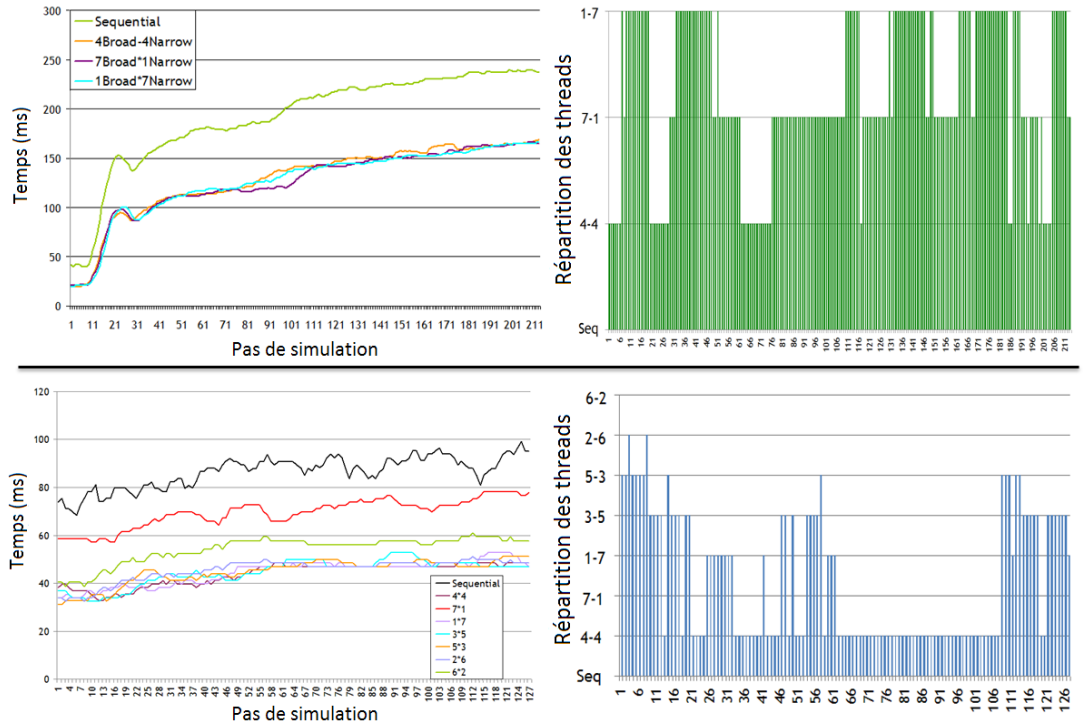


FIGURE 6.8 – Comparaison du pipeline séquentiel avec différentes configurations de notre technique de répartition des threads (4-4 signifie 4 threads pour la Broad et 4 pour la Narrow). Les tests sont effectués sur huit cœurs avec 500 Tores (**Haut**) et 1000 cubes (**Bas**). Les graphiques de **gauche** représentent les temps de calcul et ceux de **droite** présentent la meilleure configuration à chaque pas de temps.

ont été effectués sur un processeur Intel Xeon X5482 (bi-quad cœur) de 3,20 GHz sur Windows XP avec 64 Go de RAM. Nous présentons, dans un premier temps, une étude sur les principales différences entre plusieurs configurations possibles de répartition de thread afin de justifier l'utilisation d'une répartition dynamique. La Figure 6.8 compare les temps de calcul du pipeline séquentiel et de différentes configurations de répartition des threads entre la Broad et la Narrow phase. Au total, sept configurations différentes (1-7, 2-6, 3-5, 4-4, 5-3, 6-2 et 7-1) où $X - Y$ signifie X threads pour la Broad phase et Y pour la Narrow phase.

Comme nous l'avons présenté, la solution parallèle réduit significativement le temps de calcul or il apparaît que plusieurs configurations de répartition des threads ne sont pas appropriées pour le type de simulation. Par exemple, les configurations 7-1 et 6-2 dans la deuxième simulation sont plus longues que les autres. Cependant, lorsque nous cherchons la meilleure configuration à chaque pas de temps sur ces courbes, on remarque que le meilleur candidat n'est pas unique et change constamment. Les deux graphiques de droite illustrent ce phénomène en montrant à chaque pas de simulation, quelle répartition de threads est la plus rapide. Nous notons que l'algorithme séquentiel n'est jamais le meilleur algorithme. Selon l'évolution de la simulation, les phases ont

des besoins qui changent en termes de nombre de threads nécessaires pour réduire leur temps de calcul.

Ces deux exemples confirment le fait qu'il n'existe pas de répartition unique, meilleure que les autres dans une simulation. Comme les deux phases (Broad et Narrow) subissent des évolutions de temps de calcul, une adaptation dynamique est nécessaire. La Figure 6.9 montre les résultats que nous avons obtenus grâce à notre technique d'équilibrage dynamique de threads. La ligne noire indique le choix dynamique du nombre de threads obtenu par notre méthode. La zone bleue représente la meilleure configuration à chaque pas de temps. Pour obtenir cette zone bleue nous avons rejoué la même simulation avec différentes configurations fixes de répartition de threads en mesurant le temps de calcul. Nous avons ensuite croisé les résultats afin d'établir pour chaque pas de temps la configuration qui obtenait le meilleur temps. Notre solution dynamique suit de près les résultats optimaux et ne change pas ponctuellement comme les résultats calculés le font. La limitation de notre approche est également illustrée par la non anticipation des variations du temps de calcul. Par exemple, sur le graphique du haut à gauche, autour de la 67^e étape, nous voyons que l'adaptation prend quelques pas de temps pour changer la répartition des threads en raison de la dynamicité du pas de contrôle. Nous ne contrôlons pas le temps passé par les phases à chaque pas de temps. Quand une nouvelle configuration est choisie, la fréquence de contrôle devient maximale afin d'assurer que la nouvelle répartition des threads est plus rapide en temps de calcul que la précédente. Tant que la configuration reste inchangée, la fréquence diminue, ce qui explique le temps de réaction de l'adaptation. Sur le deuxième graphique, nous avons seulement autorisé le module de répartition à utiliser trois configurations possibles de répartition des threads (4-4, 7-1 et 1-7). La configuration 4-4 est utilisée lorsque les deux phases ont à peu de choses près le même temps de calcul, la configuration 1-7 favorise la Broad phase et la configuration 1-7 la Narrow phase.

Nous avons montré que notre nouvelle technique d'équilibrage de parallélisme permet d'adapter le nombre de threads en fonction du temps de calcul des deux phases. Comme les phases ont un temps de calcul évolutif, nous sommes en mesure de le détecter et d'en tenir compte en ajoutant ou en supprimant des threads. L'étape de contrôle dynamique permet d'assurer que les choix de la nouvelle répartition sont corrects et la fréquence variable des étapes permet de passer moins de temps à contrôler le temps de calcul des phases.

6.1.5 Synthèse et perspectives

Nous avons présenté le premier pipeline de détection de collision parallèle et adaptatif pour les architectures multi-cœur. Comparativement au pipeline séquentiel, les deux principales phases ont été modifiées et adaptées pour s'exécuter simultanément. Nous proposons d'utiliser un modèle producteur-consommateur entre la Broad et Narrow phase. Les données filtrées par la Broad phase sont stockées dans une mémoire tampon en prévision d'être utilisées par la Narrow phase. Dès qu'une paire a été calculée par la Broad phase, elle est stockée puis traitée par la Narrow phase. Ce nouveau pipeline est également couplé à une nouvelle technique d'équilibrage dynamique de parallélisme permettant d'adapter la répartition des threads entre les deux phases au cours de la

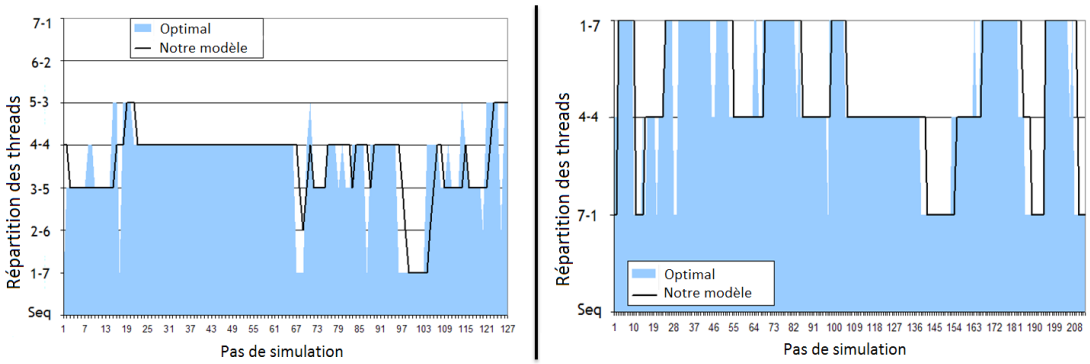


FIGURE 6.9 – Résultats de la technique d'équilibrage dynamique du parallélisme qui gère et adapte le nombre de threads entre les différentes étapes du pipeline durant une simulation. Le graphique de gauche a été obtenu avec 1000 cubes et celui de droite avec 500 tores.

simulation. Un pas de contrôle dynamique est réalisé afin de mesurer le temps de calcul et d'assurer que les nouvelles répartitions des threads sont meilleures que les précédentes. Cette technique d'équilibrage permet de faire face aux évolutions du scénario de la simulation et de réduire les temps de calcul des différentes phases.

L'exécution parallèle des phases du pipeline permet donc de répondre parfaitement au critère de structuration évoqués en synthèse de l'état de l'art. De plus, l'adaptation dynamique du pipeline durant la simulation répond au critère d'adaptativité. L'exécution possible sur un nombre quelconque de cœurs fait que le critère de généricité est également pris en compte et enfin, les mesures de performances réalisés affichant une réduction de 30 à 40% de moyenne du temps de calcul montre que le dernier critère de rapidité est également parfaitement intégré à ce nouveau pipeline. Tous les critères évoqués sont donc réunis et respectés au sein de notre nouveau pipeline parallèle de détection de collision.

Il serait intéressant de tester ce nouveau pipeline de détection de collision sur des architectures multi-cœur plus larges (16 ou 32 cœurs) afin d'analyser son comportement. Les solutions parallèles utilisées au sein de chacune des phases ne sont pas forcément les plus optimales, il serait donc judicieux de les optimiser au maximum pour évaluer leurs performances sur les architectures multi-cœur lors d'une utilisation du pipeline parallèle. Ce nouveau pipeline parallèle de détection de collision apporte une multitude d'orientations futures. L'une des plus intéressantes serait d'étendre les capacités parallèles de ce pipeline aux architectures GPU et multi-GPU. Cela signifierait une exécution des phases en parallèle sur multi-cœur et multi-GPU avec une adaptation dynamique des tâches et des données entre les unités de calcul.

6.2 Adaptation Algorithmique Dynamique

Cette section présente nos travaux sur l'adaptation dynamique d'algorithmes durant les simulations. Nous nous sommes intéressés au critère d'adaptativité et de généricité établis lors de notre état de l'art de la littérature. Nous avons, pour cela, travaillé sur des simulations au scénario évolutif. Ce type de simulation consiste en une variation très rapide et très importante des conditions environnementales. Une condition va concernée dans notre cas la variation du nombre d'objets en un intervalle de temps très court. Nous pouvons prendre en exemple une simulation avec un utilisateur pouvant, à sa guise, ajouter, supprimer, casser, lier des objets au sein de l'environnement virtuel (cf. Figure 6.10). Un autre exemple pourrait être les simulations faisant intervenir la fracture d'objets où un seul objet se transforme en une multitude de débris indépendants. Dans ces deux exemples, le nombre d'objets présents dans l'environnement peut varier en très peu de temps. Afin de palier le manque de dynamicité au sein des algorithmes de détection de collision pour faire face à l'évolution des simulations, nous proposons une approche d'adaptation algorithmique dynamique basée sur l'utilisation de scénarios de simulations hors-lignes [AGA11b]. Nous présentons, par la suite, le contexte technique de ces travaux (cf. Section 6.2.1) en présentant les algorithmes utilisés, le descriptif complet de notre approche (cf. Section 6.2.2) puis son évaluation (cf. Section 6.2.3).

6.2.1 Algorithmes utilisés

Nous présentons rapidement, dans cette section, les algorithmes que nous avons utilisés pour notre approche d'adaptation algorithmique dynamique de détection de collision. Nous détaillons le développement et l'adaptation de ces algorithmes sur architecture multi-cœur, simple GPU et multi-GPU. Ce sont tous des algorithmes de Broad phase basés sur le Sweep and Prune (SaP) [CLMP95] en utilisant l'approche dite de force semi-brute car c'est une très bonne candidate à la parallélisation due à l'indépendance des calculs. Nos algorithmes sont adaptés de manière générique afin de s'exécuter sur les unités de calcul disponibles (CPU séquentiel, multi-cœur, simple GPU et multi-GPU).

- **Algorithme CPU séquentiel**
  Nous utilisons la version incrémentale de l'algorithme du SaP. Plus précisément, la version manipulée est celle utilisée par le moteur physique Bullet [Bul].

- **Algorithme CPU multi-cœur**
  Nous utilisons l'algorithme que nous avons proposé pour paralléliser l'algorithme du SaP sur une architecture multi-cœur [AGA10a] (cf. Chapitre 3 Section 3.1). L'algorithme est en mesure de s'exécuter sur des architectures composées d'un nombre quelconque de cœurs. Selon les résultats obtenus, il permet de diviser le temps de calcul par 5X-6X sur une architecture 8-cœur et devient plus performant que le modèle incrémental de l'algorithme à partir d'un certain nombre de cœurs (six dans notre cas).

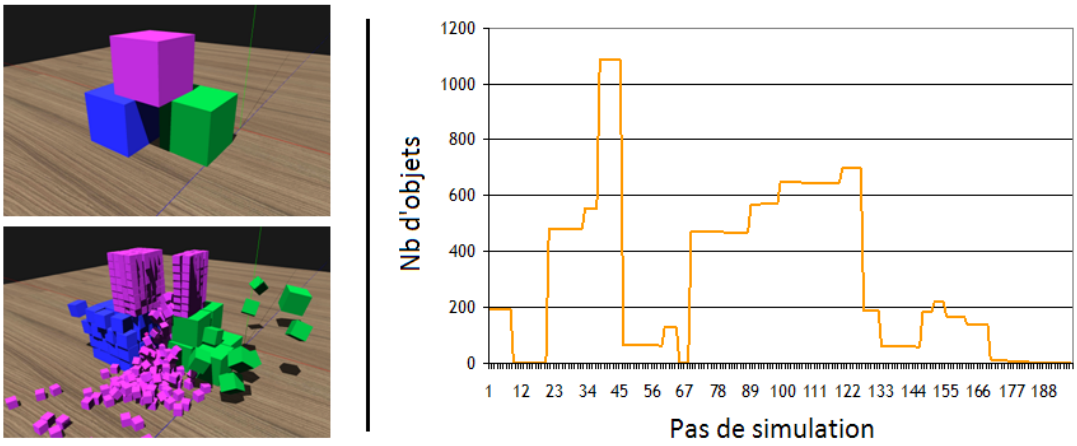


FIGURE 6.10 – Scénario évolutif en fonction du nombre d'objets. La simulation commence avec quelques cubes (en haut à gauche) et l'utilisateur peut ensuite diviser ou supprimer des cubes autant qu'il le souhaite (en bas à gauche). L'objectif est de faire varier le nombre d'objets dans la scène de quelques-uns à plusieurs centaines de milliers, comme illustré sur la courbe de droite.

– **Algorithme simple GPU**
  Nous utilisons l'algorithme du SaP sur GPU [AGA11a] présenté précédemment (cf. Chapitre 4 Section 4.1). Au vu des résultats, cet algorithme s'est révélé très performant pour des environnements composés de quelques milliers d'objets et meilleur que les algorithmes existants.

– **Algorithme multi-GPU**
  Et enfin, nous utilisons notre algorithme multi-GPU de Broad phase [AGA11a] (cf. Chapitre 5 Section 5.2). Cet algorithme est en mesure de s'exécuter sur un nombre quelconque de GPUs tout en réduisant le temps de calcul. Les résultats ont montré une réduction de 3,5-4 en passant de un à quatre GPUs.

6.2.2 Processus global

Un aperçu de notre approche est illustré sur la Figure 6.11, il est composé de deux parties : les calculs hors-lignes et leur utilisation au cours de l'exécution en ligne. Pour illustrer notre propos, la Figure 6.10 montre un exemple d'une évolution complexe de scénario. La simulation commence avec seulement quelques objets et évolue ensuite rapidement lorsque l'utilisateur ajoute, enlève ou brise des objets en plusieurs milliers de morceaux. Dans ce cas, le nombre d'objets n'est pas stable tout au long de la simulation. Les algorithmes doivent être adaptés pour faire face à ces perturbations. Il est convenu que l'algorithme basé GPU est, à coup sûr, le moyen le plus rapide pour calculer l'étape de Broad phase pour une simulation composée de milliers d'objets. Il est également convenu que l'algorithme séquentiel basé CPU est potentiellement le meilleur pour une

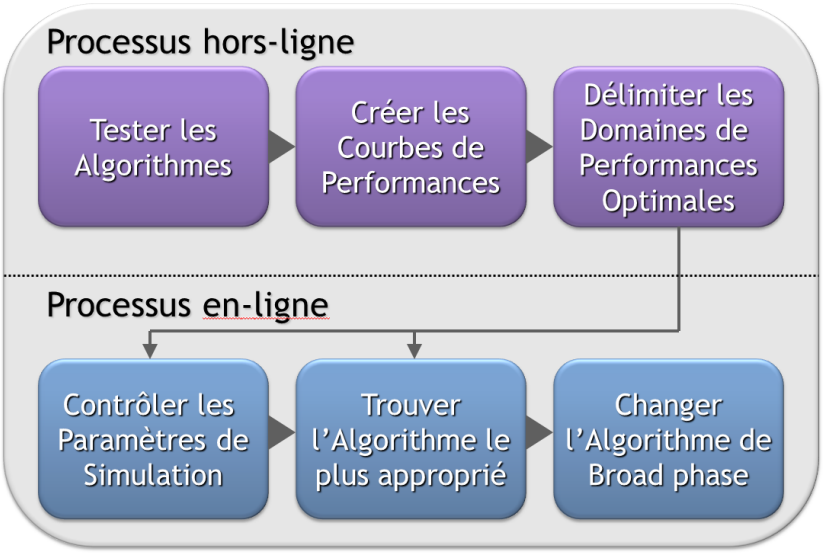


FIGURE 6.11 – Les calculs hors-lignes sont effectués pour évaluer les domaines de performance optimale de chaque algorithme. Les résultats sont ensuite utilisés lors du processus de l'exécution en ligne afin de choisir l'algorithme le plus approprié.

simulation de seulement quelques objets. Mais nous ne connaissons pas exactement le moment à partir duquel l'un devient meilleur ou pire que l'autre. Nous ne connaissons pas non plus les limites pour des architectures multi-GPU. En d'autres termes, quel est le nombre optimal de GPU permettant de fournir les meilleures performances et quand doit-on inclure ou exclure un GPU de la boucle de calcul ? Nous détaillons, par la suite, les deux parties principales de notre approche que sont : le processus de simulations hors-lignes (cf. Section 6.2.2.1) et son utilisation durant l'exécution en ligne (cf. Section 6.2.2.2).

6.2.2.1 Simulations hors-ligne

Notre analyse des performances algorithmiques hors-lignes consiste en une extraction des points caractéristiques afin de délimiter les domaines de performance optimale des algorithmes de Broad phase (nous cherchons à déterminer le point de croisement des courbes illustré par la valeur "X" sur la Figure 6.12). L'algorithme du SaP est seulement sensible au nombre d'objets dans l'environnement car il fonctionne avec les volumes englobants des objets. Le temps passé à calculer les chevauchements entre des objets simples ou complexes est exactement le même. Partant de ce constat, nous proposons d'analyser et d'évaluer le comportement de chaque algorithme en faisant varier le nombre d'objets. Ces évaluations sont effectuées grâce à des simulations hors-lignes. La Figure 6.12 illustre un exemple de résultat obtenu de temps de calcul d'un algorithme s'exécutant sur CPU et sur GPU en fonction du nombre d'objets. Cette simulation hors-ligne a été effectuée sur un double-cœur CPU X7900 de 2,9 GHz et sur une Quadro FX 3600M. Nous notons que chaque algorithme a son propre comportement et, en

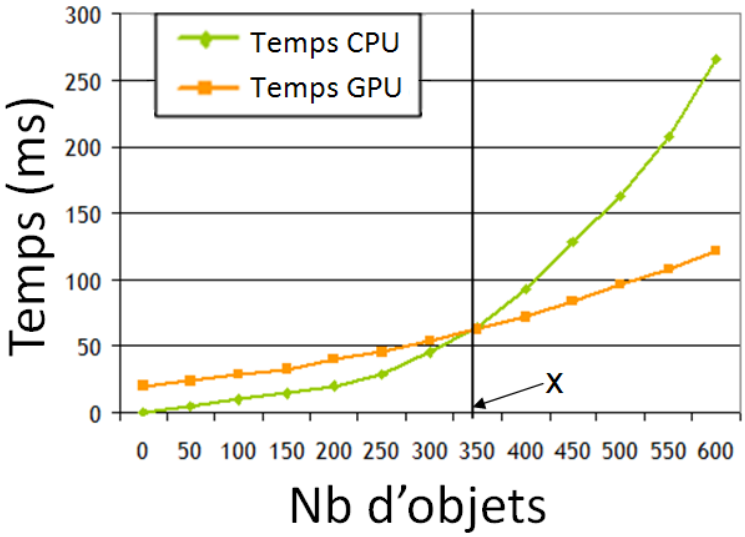


FIGURE 6.12 – Exemple de temps de calcul passé par l'algorithme de Broad phase s'exécutant sur CPU (ligne verte) ou GPU (ligne orange). Les domaines de performance optimale que nous obtenons sont représentés par les deux zones. Il suffit ensuite de déterminer la valeur X pour connaître précisément l'instant où la solution GPU devient meilleur, en termes de temps de calcul, que celle CPU.

fonction du nombre d'objets, l'un des deux ressort comme plus approprié car il utilise moins de temps pour calculer les paires qui se chevauchent. Aux alentours de 350 objets dans l'environnement, la solution algorithmique GPU devient plus performante que celle s'exécutant sur CPU. Basé sur un scénario de simulation imprévisible comme l'illustre la Figure 6.10, nous aimerions que le temps de calcul soit constamment le plus faible possible. En regardant le graphique, nous sommes bien conscients qu'il est impossible de déterminer quel algorithme est meilleur que l'autre. En revanche, il est possible de déterminer quel algorithme est le candidat le plus approprié pour un nombre spécifique d'objets.

Nous utilisons un procédé d'extraction des points d'intersection des courbes. Ces points délimitent les domaines de performance optimale de chaque algorithme. Ils sont ensuite utilisés lors de la simulation en ligne pour contrôler et changer, si besoin est, l'algorithme de Broad phase. Chaque simulation hors-ligne, effectuée pour tester un algorithme, génère un tableau de données avec des informations de temps de calcul. Nous procédons à une comparaison de l'information des temps de calcul pour construire les domaines de performance optimale.

6.2.2.2 Simulation en ligne

Les résultats obtenus par la détermination des domaines de performance optimale des algorithmes sont utilisés lors de la simulation en ligne. Les frontières de ces domaines

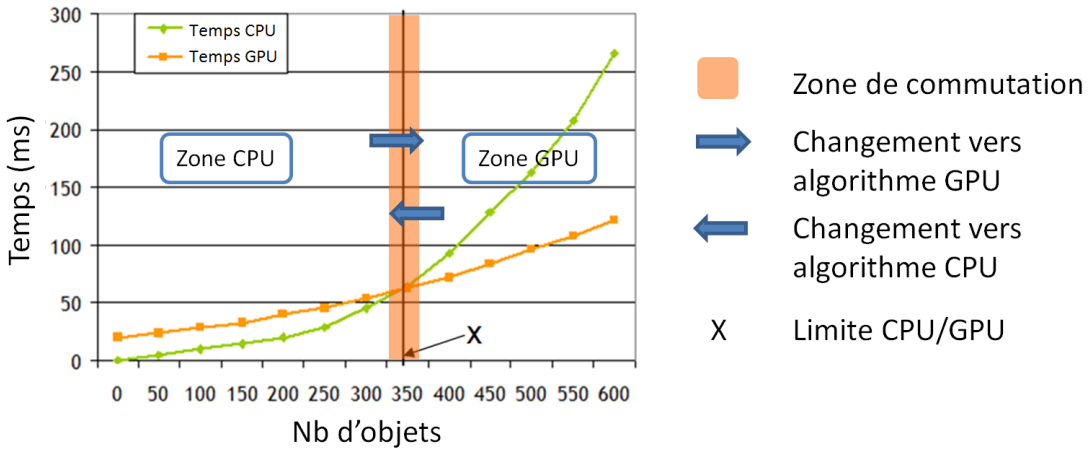


FIGURE 6.13 – Zone de commutation algorithmique englobant le point de croisement entre deux courbes de performance.

sont stockées pour être utilisées pendant le processus de commutation de l'algorithme Broad phase. Nous proposons de contrôler les paramètres de la simulation (ici le nombre d'objets dans l'environnement). Ce contrôle permet de déterminer s'il est nécessaire de changer l'algorithme utilisé, par un autre plus approprié lorsque les frontières des domaines de performance optimale sont franchies. Pour assurer une commutation en temps réel pendant le transfert algorithmique, chaque algorithme prend en entrée la même structure de données et donne en sortie les mêmes données résultat. Afin d'éviter des minima locaux qui peuvent apparaitre lors d'une oscillation autour de la frontière des domaines de performance optimale, nous utilisons une "zone de commutation" analogue à l'effet d'hystérèse. Cette zone englobe le point de croisement entre deux courbes permettant ainsi de redéfinir le moment où il faut changer d'algorithme. La Figure 6.13 schématise cette zone de commutation. La taille de la zone est totalement modifiable par l'utilisateur. Autrement dit, étant donné X le nombre d'objets représentant la limite entre deux zones de performance optimale de deux algorithmes A_1 et A_2 et v la valeur de commutation définie par l'utilisateur, on bascule de A_1 vers A_2 pour $X + v$ et inversement pour $X - v$. Le contrôle est inclus dans l'application globale et plus précisément dans la partie en charge de la physique. Notre approche de Broad phase dynamique est illustrée sur la Figure 6.14.

6.2.3 Résultats

Les simulations hors-lignes sont nécessaires en raison des performances algorithmiques tributaires de l'architecture d'exécution. Nous avons utilisé trois différentes plateformes pour tester notre solution, à savoir :

- Intel cœur(x2) CPUX7900 de 2,8 GHz - Quadro FX 3600M
- Intel Xeon(x4) CPUX5472 de 3 GHz - 2 Quadro FX 4600
- Intel Xeon(x8) CPUX5482 de 3,2 GHz - 4 Quadro FX 5800

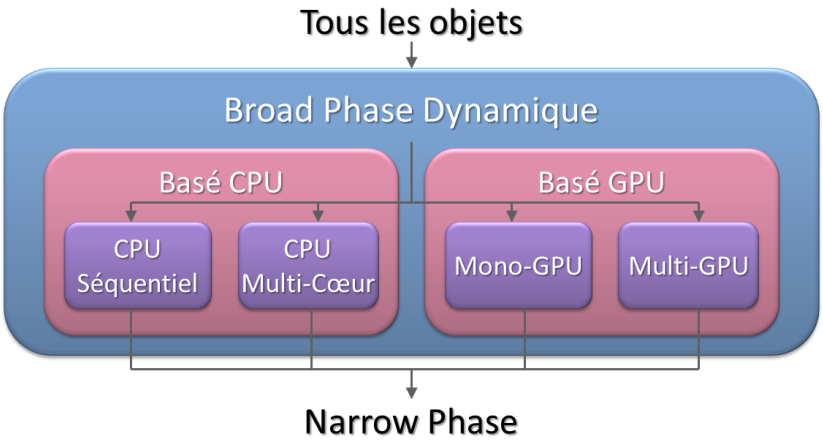


FIGURE 6.14 – Vue simplifiée de notre adaptation dynamique de Broad phase. Les données d'entrée sont orientées vers l'algorithme le plus efficace à l'instant donné.

La Figure 6.15 illustre un exemple de résultats obtenus sur la troisième plateforme GPU. Chaque courbe représente le temps de calcul par rapport au nombre d'objets. On peut remarquer qu'entre le début et la fin du graphique, l'ordre de la meilleure à la pire courbe de temps de calcul, est totalement inversé.

Chaque courbe a son propre comportement par rapport au nombre d'objets dans l'environnement. Sans surprise, l'utilisation de quatre GPUs pour calculer la Broad phase pour une centaine d'objets semble être la pire façon de procéder. De même, l'utilisation d'un seul GPU unique pour plus de 4.000 objets entraîne une perte de temps. Le point intéressant est de constater que chaque algorithme possède son moment d'utilisation. Après la mesure de ces résultats, nous pouvons croiser les données des courbes afin de déterminer les différents points d'intersection pour délimiter le domaine de performance optimale de chaque algorithme. Il est désormais possible, au cours de la simulation en ligne, de savoir quand inclure ou retirer un GPU de la boucle de calcul du processus de Broad phase.

La Figure 6.16 présente les résultats CPU obtenus sur la deuxième plateforme. Comme nous l'avons constaté dans les résultats GPU, l'ordre du meilleur au pire temps de calcul est inversé entre le début et la fin. Nous avons indiqué les domaines de performance optimale de chaque algorithme afin de révéler leur "meilleur moment d'utilisation". Cela montre qu'utiliser en permanence tous les cœurs disponibles n'est pas la solution optimale et recommandée. Il est nécessaire de désormais prendre en compte le nombre d'objets présents dans l'environnement.

Un couplage des résultats CPU et GPU est également possible, mais difficile à représenter de manière graphique. Cependant, ce couplage est effectué lors de simulations hors-lignes afin de déterminer les domaines de performance optimale de tous les algorithmes disponibles. Les résultats montrent que les algorithmes CPU et GPU sont différents en ce qui concerne la dépendance du temps de calcul avec le nombre d'objets.

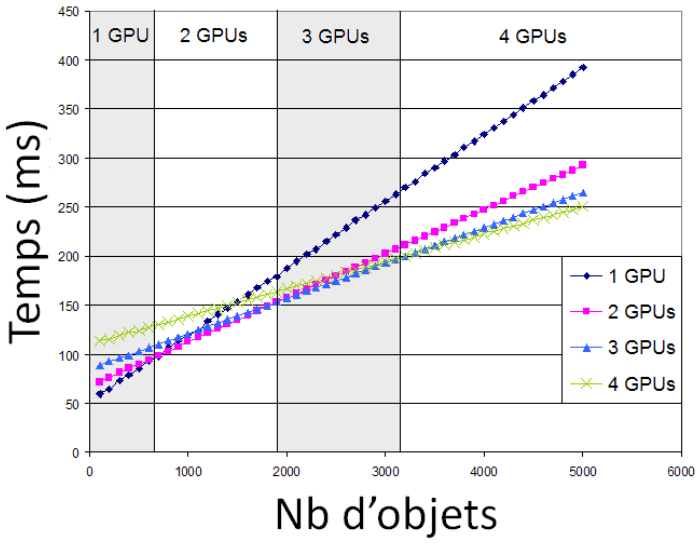


FIGURE 6.15 – Les résultats des simulations hors-lignes effectuées sur une plateforme composée de quatre Quadro FX 5800. Nous avons mesuré le temps de calcul en utilisant 1, 2, 3 et 4 GPU(s) en fonction du nombre d'objets dans l'environnement.

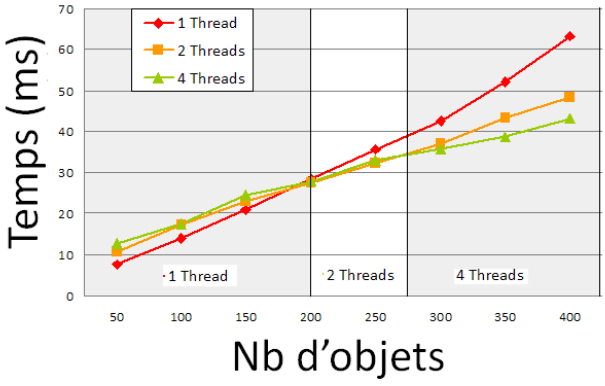


FIGURE 6.16 – Les résultats des simulations hors-lignes effectuées sur une plateforme composée de quatre cœurs. Nous avons mesuré le temps passé par les 1, 2 et 4 thread(s) en fonction du nombre d'objets dans l'environnement.

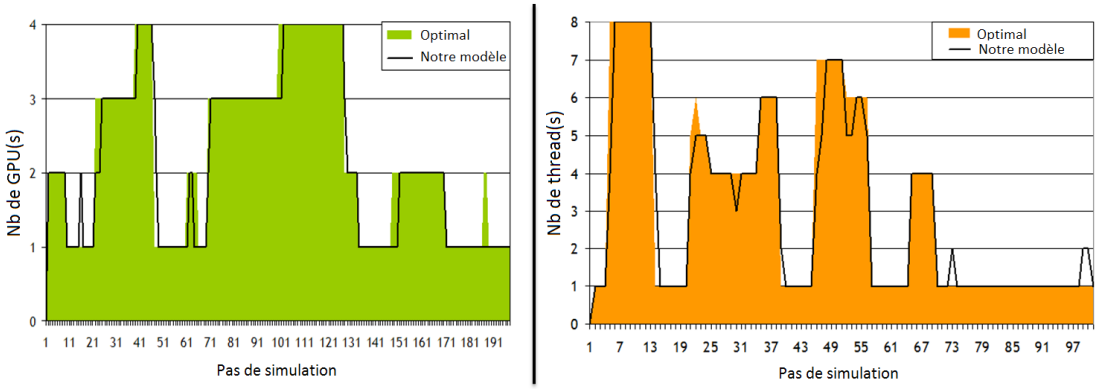


FIGURE 6.17 – Illustrations graphiques de notre approche d'adaptation algorithmique dynamique. Le graphique de **gauche** montre un résultat obtenu sur plateforme multi-GPU et celui de **droite** illustre un résultat sur multi-cœur.

La Figure 6.17 montre deux résultats de notre adaptation algorithmique dynamique. Le graphique de **gauche** montre un résultat obtenu sur plateforme multi-GPU. Celui de **droite** illustre un résultat sur multi-cœur. Le protocole d'expérimentation a consisté en l'enregistrement d'une simulation interactive où l'on mesurait les moments des actions de l'utilisateur ainsi que les variations du nombre d'objets. Nous avons ensuite rejoué cette simulation pour chaque configuration à évaluer. La ligne noire présente les choix effectués par notre modèle dynamique entre ajouter ou supprimer un GPU ou cœur CPU de la boucle de calcul au cours de la simulation. Les zones colorées présente les résultats optimaux obtenus lors du croisement des résultats des simulations rejouées. Les essais ont été effectués sur 4 * Quadro FX 5800 et sur un Intel Xeon 5482 cadencé à 3,2 GHz. On constate que notre modèle suit de très près les résultats calculés et que l'algorithme de Broad phase a été, à plusieurs reprises, modifié pour être remplacé par un autre algorithme disponible plus approprié. Nous observons tout de même quelques légères variations entre les deux résultats pouvant être dues à différentes phénomènes. La première raison possible est que, durant ces simulations, l'occupation du processeur n'est pas prise en compte et qu'elle peut fortement variée entre le calcul des domaines de performance optimale des algorithmes réalisé lors de simulations hors-lignes et leur utilisation durant la simulation en ligne. La seconde raison possible est plus d'ordre matériel et concerne l'utilisation passée des unités de calcul. En effet, un processeur multi-cœur n'ayant pas ou peu utilisé certains de ses cœurs depuis un certain temps sera légèrement moins performant qu'un processeur ayant utilisé à pleine puissance tous ces cœurs avant les calculs. L'activation d'un cœur peut mettre un certain temps avant d'être performante et c'est peut être l'explication des légères variations présentes sur les graphiques. Le phénomène est similaire pour les GPUs. Un GPU ayant été massivement utilisé avant que nous effectuions nos calculs hors-lignes offrira de meilleures performances qu'un autre GPU "froid" non-utilisé avant.

6.2.4 Synthèse et perspectives

Nous avons présenté une première manière d'adapter dynamiquement l'étape de Broad phase de la détection de collision lors de simulations aux scénarios évolutifs. Ces travaux répondent au critère d'adaptativité désormais indispensable pour les algorithmes de détection de collision pour faire face à l'évolution des architectures. Nous avons développé des algorithmes de Broad phase pouvant s'exécuter sur plusieurs types d'architectures différentes (CPU séquentiel, CPU multi-cœur, simple GPU et multi-GPU). Dans notre approche, nous proposons de déterminer les domaines de performance optimale pour chaque algorithme de Broad phase. Nous évaluons les performances des algorithmes sur des architectures différentes en fonction du nombre d'objets dans l'environnement. Des courbes de performance des algorithmes, nous extrayons les points d'intersection des courbes pour délimiter la zone où l'algorithme est optimal. Les résultats obtenus sont ensuite utilisés lors de la simulation en ligne pour commuter, en temps réel, l'algorithme de Broad phase par un algorithme plus approprié car plus rapide.

Le critère de généricité est respecté car n'importe quel nouvel algorithme peut être inclus dans la boucle de pré-calcul hors-ligne et comparé par rapport aux autres. Les résultats montrent que cette adaptation dynamique des algorithmes de Broad phase permet de déterminer et d'utiliser l'algorithme le plus efficace tout au long d'une simulation. Nous savions que la solution multi-GPU de détection de collision était totalement inadaptée aux petites simulations physiques tout comme l'est la solution CPU séquentielle pour des simulations à grande échelle, mais nous sommes désormais en mesure de savoir à partir de quand l'un devient meilleur ou pire que l'autre. Les mesures de performances effectuées permettent de valider le critère de rapidité car notre approche permet, désormais, d'offrir un temps de calcul quasi-optimal par rapport aux algorithme disponibles (critère de rapidité).

Il y a plusieurs voies d'optimisation dans l'adaptation dynamique des algorithmes de détection de collision. Les algorithmes de Broad phase peuvent encore être améliorés en termes de temps de calcul et d'efficacité. La répartition des tâches et des données est également un bon moyen d'amélioration de ces algorithmes s'exécutant sur multi-cœur ou architecture multi-GPU. Un autre moyen important et intéressant d'amélioration est l'adaptation dynamique des algorithmes au cours du processus de Narrow phase. Cette étape est plus complexe car de nombreux paramètres supplémentaires sont à prendre en compte lors de cette phase (complexité des objets, les hiérarchies, les propriétés, le nombre de polygones ...). Plusieurs analyses et mises en œuvre doivent encore être effectuées pour révéler le caractère essentiel de l'adaptation des algorithmes de détection de collision sur l'architecture d'exécution.

6.3 Conclusion

Nous avons présenté, à travers ce sixième chapitre, deux contributions (pipeline parallèle [AGA10b] et adaptation algorithmique dynamique [AGA11b]) respectant les quatre critères définis lors de l'état de l'art. Nous avions ainsi décrit que le respect de ces critères permettrait de réduire le goulet d'étranglement calculatoire critique de la détection de collision. Les résultats sur le pipeline parallèle (cf. Section 6.1.4) et l'adaptation

dynamique (cf. Section 6.2.3) illustrent et prouvent que les critères ont bien été respectés et que le goulet a bien été réduit. À travers les différentes synthèses, nous avons également montré que de nombreuses perspectives permettraient de poursuivre cette réduction de la combinatoire et d'accroitre la généricité et la rapidité des algorithmes.

Conclusion

Nous avons présenté, à travers ce manuscrit, nos travaux sur l'établissement de liens génériques et adaptatifs, entre la détection de collision et le calcul haute performance. L'objectif a été de bénéficier des nouvelles architectures pour réduire le goulet d'étranglement calculatoire de la détection de collision. Nous présentons, dans la suite de cette conclusion, le bilan de nos travaux quant aux contributions réalisées ainsi que différentes perspectives.

Bilan des contributions

L'étude du goulet d'étranglement calculatoire de la détection de collision (cf. Chapitre 1) ainsi que celle de l'évolution des architectures machines [AGA09b], nous a permis d'établir lors de l'analyse des solutions parallèles existantes (cf. Chapitre 2) un certain nombre de fonctionnalités et critères essentiels à la réduction du goulet d'étranglement au sein d'environnements large échelle. Ces critères, fixés au début de ce doctorat, étaient les suivants : *Généricité, Rapidité, Structuration et Adaptativité.*

Conception d'algorithmes parallèles génériques

Ce premier objectif, répondant principalement aux critères de généricité et de rapidité, a eu pour but de tirer profit d'un nombre quelconque d'unités parallèles de calcul pour nos algorithmes (multi-cœur, GPU et multi-GPU).

Nous avons proposé un premier algorithme parallèle de Broad phase basé sur le Sweep and Prune pour les architectures multi-cœur [AGA10a] (cf. Chapitre 3). Les tendances des nouveaux processeurs allant clairement vers cette direction d'accroissement du nombre de cœurs, notre algorithme parallèle est en mesure de s'intégrer parfaitement dans ce contexte en s'exécutant sur un nombre quelconque de cœurs tout en offrant de meilleures performances. Les résultats ont montré que cette nouvelle version parallèle de l'algorithme, basée sur une approche semi-brute, permettait de réduire le temps de calcul (réduction par 5 sur 8 cœurs). Notre algorithme devenait plus performant que la version séquentielle et incrémentale pour un certain nombre de cœurs dépendant du nombre d'objets.

Nous avons proposé un nouvel algorithme pour l'étape de Narrow phase basé sur un modèle de répartiteur parallèle (cf. Chapitre 3). Ce répartiteur parallèle est basé sur un nouveau concept générique de matrice algorithmique dynamique et généralisable aux autres étapes du pipeline. Cette matrice permet d'évaluer les algorithmes entre eux

avant une simulation selon différentes classes d'objets préalablement définies.

Nous avons également présenté deux algorithmes GPU (cf. Chapitre 4). Le premier est basé sur une approche topologique proche du Sweep and Prune [AGA11a] et le second est un couplage entre une subdivision spatiale et une approche de force brute optimisée. Les temps de transfert et de calcul ont été optimisés et réduits pour les deux modèles, permettant ainsi d'accroitre la fréquence de calcul pour des environnements virtuels large échelle.

La mise en place d'un nouveau pipeline hybride nous a permis d'appréhender et de révéler le caractère primordial des architectures multi-cœur et multi-GPU (cf. Chapitre 5). Ce pipeline répond aux critères de généricité et de rapidité car il est en mesure de s'exécuter sur un nombre quelconque de cœurs et de GPUs en réduisant le temps de calcul. Notre approche utilise deux techniques de répartition de données entre les différents GPUs lors des deux étapes de Broad et de Narrow phase. La gestion en amont des GPUs est parallélisée et gérée sur multi-cœur.

Nouvelle structuration du pipeline

Le critère de structuration a également été pris en compte pour nos travaux ayant portés sur des modifications de l'organisation du pipeline de détection de collision (cf. Chapitre 6). En effet, nous avons, dans un premier temps, proposé une modification du pipeline de détection de collision permettant de prendre en compte l'architecture d'exécution [AGA09a]. Puis, nous avons proposé un nouveau modèle de pipeline parallèle offrant la possibilité d'exécuter les différentes étapes du processus en parallèle [AGA10b]. Le respect d'un modèle producteur-consommateur a permis de rompre avec la séquentialité des phases tout en conservant celle du traitement d'une paire d'objet pour raffiner la précision des calculs. Ce nouveau pipeline englobe également les trois autres critères définis que sont :

- La rapidité, car l'analyse de nos évaluations ont illustré que ce nouveau pipeline était plus performant que le modèle séquentiel.
- La généricité, car il est basé sur des algorithmes capables de s'exécuter sur un nombre quelconque de cœurs.
- L'adaptativité, car il est muni d'une approche d'adaptation dynamique de la répartition des threads permettant un équilibrage de charge en fonction de l'évolution des simulations.

Adaptation dynamique des algorithmes

Ce dernier objectif a été pris en compte lors de nos travaux sur l'adaptation algorithmique dynamique de l'étape de Broad phase basée sur des précalculs hors-lignes [AGA11b] (cf. Chapitre 6). La généricité des architectures et la rapidité de l'approche sont également deux critères intégrés par ces travaux. En effet, ces précalculs permettent de définir les domaines de performance optimale des algorithmes disponibles. Ils offrent ainsi la possibilité de pouvoir, durant la simulation, changer dynamiquement l'algo-

rithme utilisé par un autre candidat plus rapide. Ces travaux sur l'adaptation algorithmique ont, dans un premier temps, permis de révéler le fait qu'il est impossible de repérer un algorithme meilleur que tous les autres pour une simulation dont le scénario est évolutif. Et, dans un second temps, le contrôle dynamique des différentes temps de calcul des phases du pipeline offre désormais la possibilité aux simulations 3D de fournir un temps de calcul très proche de l'optimal en contrôlant et adaptant leurs algorithmes sur l'architecture d'exécution.

Nos travaux de thèse ont donc permis de réduire la combinatoire des algorithmes de détection de collision au sein d'environnements large échelle en répondant aux quatre critères définis indispensables lors de l'analyse critique de l'état de l'art. L'utilisation d'algorithmes basés multi-cœur et multi-GPU inclus dans notre nouveau pipeline hybride, lui-même basé sur le modèle de pipeline parallèle, adaptant dynamiquement le parallélisme de ces phases en fonction de l'évolution de la simulation est, pour nous, la réalisation concrète et finale de nos travaux.

Perspectives

Le caractère récent de ce domaine du calcul haute performance pour la détection de collision explique, à lui seul, la multitude de voies possibles pour poursuivre ces travaux. Tout au long de ce manuscrit nous avons détaillé les diverses perspectives relatives aux contributions. Nous résumons, dans la suite, les principales perspectives envisageables.

1- Adaptation dynamique pour la Narrow phase

Une première perspective intéressante serait de poursuivre l'approche sur l'adaptation algorithmique de la Broad phase en s'attachant à l'étendre à l'étape de Narrow phase car l'élaboration d'une telle approche permettrait d'obtenir un pipeline de détection de collision entièrement adaptatif et générique. Il serait pour cela nécessaire de prendre en compte des paramètres supplémentaires tels que la complexité des maillages et les propriétés physiques des objets afin d'être en mesure d'adapter la répartition des données et/ou des calculs lors de cette étape en fonction de l'évolution de la simulation.

2- Modèle Architecture/Algorithme

Un point crucial à développer serait de définir un modèle à base de règles qui, grâce aux paramètres de l'architectures et des caractéristiques de la simulation (taille, nombre d'objets, complexité, propriétés physiques, etc.) serait en mesure d'établir la répartition optimale tant au niveau données qu'au niveau calcul du pipeline de détection de collision. Cette perspective est ambitieuse mais elle serait en mesure de bouleverser la gestion de la performance pour la détection de collision. La multitude de paramètres à prendre en compte et l'incertitude de certaines mesures seraient certainement la difficulté majeure de l'élaboration d'un tel système de règles.

3- Grilles et clusters

Une perspective à plus long terme serait de s'éloigner de l'utilisation d'un simple ordinateur et d'étudier l'utilisation d'architectures spécialement conçues pour le calcul haute performance. Ces architectures pourraient être de type clusters CPU, GPU ou hybrides voire même de type grille de calcul. La latence entre les différents sites géographiques d'une grille de calcul pourrait conduire à l'utilisation de méthodes prédictives basées sur cette latence. L'utilisation de clusters est d'ores et déjà effective au sein du domaine de la réalité virtuelle [RS06] mais est assez peu étudiée pour la détection de collision.

Ces différentes perspectives illustrent la grande diversité de voies possibles pour poursuivre ces travaux. Les architectures multi-cœur, GPU et multi-GPU sont, et vont être, des éléments clé du parallélisme des algorithmes de détection de collision. La conception de tels systèmes nécessite donc des analyses supplémentaires détaillées des techniques de répartition afin de tirer profit au maximum des performances de ces architectures d'exécution complexes.

Références de l'auteur

[AGA09a] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. Adaptation multi-coeur d'un algorithme de détection de collision et intégration dans pipeline 3d. In *4eme Journées de l'Association Française de Réalité Virtuelle (AFRV 2009), 9-11 December 2009, Lyon, France*, 2009.

[AGA09b] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. New trends in collision detection performance. In Simon Richir & Akihiko Shirai, editor, *Virtual Reality International Conference (VRIC) 2009*, pages 53–62, April 2009.

[AGA10a] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. A broad phase collision detection algorithm adapted to multi-cores architectures. In Simon Richir & Akihiko Shirai, editor, *Virtual Reality International Conference (VRIC) 2010*, pages 95–100, April 2010.

[AGA10b] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. Synchronization-free parallel collision detection pipeline. In *International Conference on Artificial Telexistence (ICAT) 2010*, December 2010.

[AGA11a] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. Collision detection : Broad phase adaptation from multi-core to multi-gpu architecture. *Journal of Virtual Reality and Broadcasting*, In Submission, 2011.

[AGA11b] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. Dynamic adaptation of broad phase collision detection algorithms. In *IEEE International Symposium on Virtual Reality Innovations (ISVRI) - in conjunction with IEEE VR 2011*, March 2011.

Références

[ACF+07] Jérémie Allard, Stéphane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Hervé Delingette, and Laurent Grisoni. Sofa an open source framework for medical simulation. In *Medicine Meets Virtual Reality (MMVR'15)*, Long Beach, USA, February 2007.

[ACT00] Sigal Ar, Bernard Chazelle, and Ayellet Tal. Self-customized BSP trees for collision detection. *CGTA : Computational Geometry : Theory and Applications*, 15, 2000.

[AFC+10] Jérémie Allard, François Faure, Hadrien Courtecuisse, Florent Falipou, Christian Duriez, and Paul G. Kry. Volume contact constraints at arbitrary resolution. *ACM Transactions on Graphics*, 29(4) :82 :1–82 :10, July 2010.

[AGHP+00] Agarwal, Guibas, Har-Peled, Rabinovitch, and Sharir. Computing the penetration depth of two convex polytopes in 3d. In *SWAT : Scandinavian Workshop on Algorithm Theory*, 2000.

[AGL+04] Jeremie Allard, Valerie Gouranton, Loick Lecointre, Sebastien Limet, Bruno Raffin, and Sophie Robert. FlowVR : A middleware for large scale virtual reality applications. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference (10th Euro-Par'04)*, volume 3149 of *Lecture Notes in Computer Science (LNCS)*, pages 497–505, Pisa, Italy, August-September 2004. Springer-Verlag (New York).

[AR06] Jérémie Allard and Bruno Raffin. Distributed physical based simulations for large vr applications. In *IEEE Virtual Reality Conference*, 2006.

[AS01] Ulf Assarsson and Per Stenstrom. A case study of load distribution in parallel view frustum culling and collision detection. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par 2001 : Parallel Processing, 7th International Euro-Par Conference (7th Euro-Par'01)*, volume 2150 of *Lecture Notes in Computer Science (LNCS)*, pages 663–673, Manchester, UK, August 2001. Springer-Verlag (New York).

[BAPH00] Kevin Brown, Steve Attaway, Steve Plimpton, and Bruce Hendrickson. Parallel strategies for crash and impact simulations. In *Computer Methods in Applied Mechanics and Engineering*, volume 184, page 375Ũ390, 2000.

[Bar89] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics (Proc. of SIGGRAPH '88)*, 23(3) :223–231, July 1989.

[Bar90] David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 19–28, August 1990.

[Bar91] David Baraff. Coping with friction for non-penetrating rigid body simulation. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4) :31–40, July 1991.

[Bar92] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, 1992.

[BCG+96] Gill Barequet, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. Boxtree : A hierarchical representation for surfaces in 3d. *Computer Graphics Forum*, 15(3) :387–396, August 1996. ISSN 1067-7055.

[BD92] Chanderjit L. Bajaj and Tamal K. Dey. Convex decomposition of polyhedra and robustness. *SICOMP : SIAM Journal on Computing*, 21, 1992.

[BdCRV05] Antonio Benitez, Maria del Carmen Ramírez, and Daniel Vallejo. Collision detection using sphere-tree construction. In *CONIELECOMP*, pages 286–291. IEEE Computer Society, 2005.

[BEG+99] Julien Basch, Jeff Erickson, Leonidas J. Guibas, John Hershberger, and Li Zhang. Kinetic collision detection between two simple polygons. *SODA '99 Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithm*, pages 102–111, 1999.

[Ber97] Gino Van Den Bergen. Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, 2(4) :1–13, 1997.

[Ber99] Gino Van Den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *J. Graph. Tools*, 4(2) :7–25, 1999.

[Ber01] Gino Van Den Bergen. Proximity queries and penetration depth computation on 3d game objects. In *Game Developper Conference*, 2001.

[BF79] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACMCS*, 11(4) :397–409, 1979.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus : stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3) :777–786, August 2004.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree : An efficient and robust access method for points and rectangles. In *SIGMODIC : ACM SIGMOD Interantional Conference on Management of Data*, 1990.

[BL85] Charles E. Buckley and Larry J. Leifer. A proximity metric for continuum path planning. *IJCAI*, pages 1096–1102, 1985.

[Boy79] John W. Boyse. Interference detection among solids and surfaces. *Communications of the ACM*, 22(1) :3–9, January 1979.

[Bru01] Herman Bruyninckx. Open robot control software : the OROCOS project. In *ICRA*, pages 2523–2528. IEEE, 2001.

[BT95] Srikanth Bandi and Daniel Thalmann. An adaptive spatial subdivision of the object space for fast collision detection of animated rigid bodies. *Comput. Graph. Forum*, 14(3) :259–270, 1995.

[Bul] Bullet. http ://bulletphysics.org/.

[BV05] Bedřich Beneš and Nestor Goméz Villanueva. Gi-collide : collision detection with geometry images. *Proceedings of the 21st spring conference on Computer graphics*, pages 95–102, 2005.

[BW04] George Baciu and Wingo Sai-Keung Wong. Image-based collision detection for deformable cloth models. *IEEE Trans. Vis. Comput. Graph*, 10(6) :649–663, 2004.

[Cam85] Stephen Cameron. A study of the clash detection problem in robotics. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 488 – 493, mar 1985.

[Cam90] S. Cameron. Collision detection by four-dimensional intersection testing. *IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION*, 6(3), June 1990.

[Cam97] Stephen Cameron. Enhancing gjk : computing minimum and penetration distances between convex polyhedra. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 3112 –3117 vol.4, apr 1997.

[Can86] John Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8 :200–209, 1986.

[CC86] S. A. Cameron and R. K. Culley. Determining the minimum translational distance between two convex polyhedra. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 591–596, 1986.

[CCH+07] Yen-Kuang Chen, Jatin Chhugani, Christopher J. Hughes, Daehyun Kim, Sanjeev Kumar, Victor Lee, Albert Lin, Anthony D. Nguyen, Eftychios Sifakis, and Mikhail Smelyanskiy. High-performance physical simulations on next-generation architecture with many cores. Technical Report 11, Intel Technology Journal, August 2007.

[Cha81] Bernard Chazelle. Convex decompositions of polyhedra. In *STOC : ACM Symposium on Theory of Computing (STOC)*, 1981.

[Cha84] Bernard Chazelle. Convex partitions of polyhedra : A lower bound and worst-case optimal algorithm. *SICOMP : SIAM Journal on Computing*, 13, 1984.

[CK86] R. K. Culley and K. G. Kempf. A collision detection algorithm based on velocity and distance bounds. In *IEEE International Conference on Robotics and Automation*, pages 1064–1069, 1986.

[CLMP95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-collide : An interactive and exact collision detection system for large-scale environments. In *SI3D*, pages 189–196, 218, 1995.

[CMHL11] Gabriel Cirio, Maud Marchal, Sébastien Hillaire, and Anatole Lecuyer. Six degrees-of-freedom haptic interaction with fluids. In *To appear in IEEE Transaction on Visualisation and Computer Graphics*, 2011.

[CP92] Chazelle and Palios. Decomposing the boundary of a nonconvex polyhedron. In *SWAT : Scandinavian Workshop on Algorithm Theory*, 1992.

[CS06] Daniel S. Coming and Oliver G. Staadt. Kinetic sweep and prune for multi-body continuous motion. *Computers & Graphics*, 30(3) :439–449, 2006.

[CS08] Daniel S. Coming and Oliver G. Staadt. Velocity-aligned discrete oriented polytopes for dynamic collision detection. *IEEE Trans. Vis. Comput. Graph*, 14(1) :1–12, 2008.

[CTM08] Sean Curtis, Rasmus Tamstorf, and Dinesh Manocha. Fast collision detection for deformable models using representative-triangles. In Eric Haines and Morgan McGuire, editors, *SI3D*, pages 61–69. ACM, 2008.

[CUD07] NVIDIA CUDA. Compute unified device architecture programming guide. Technical report, NVIDIA : Santa Clara, CA, 2007.

[CW96] Kelvin Chung and Wenping Wang. Quick elimination of non-interference polytopen in a virtual environment. In M. Göbel, J. David, P. Slavik, and J. J. van Wijk, editors, *Virtual Environments and Scientific Visualization '96*, pages 64–73. Springer-Verlag Wien, April 1996.

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp : A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[DHKS93] David Dobkin, John Hershberger, David Kirkpatrick, and Subhash Suri. Computing the intersection-depth of polyhedra. *ALGRTHMICA : Algorithmica*, 9, 1993.

[Din99] C. Dingliana. Real-time collision detection and response using sphere-trees. Technical report, Image Synthesis Group - Trinity College Dublin, March 02 1999.

[DK90] Dobkin and Kirkpatrick. Determining the separation of preprocessed polyhedra - a unified approach. *ICALP : Annual International Colloquium on Automata, Languages and Programming*, pages 400–413, 1990.

[DM97] Leonardo Dagum and Ramesh Menon. OPENMP : An industry standard API for shared memory programming. *IEEE Computational and Evolutionary Programming*, 5 :46–55, 1997.

[DZ93] Paul Dworkin and David Zeltzer. A new model for efficient dynamic simulation. In Annie Luciani and Daniel Thalmann, editors, *Computer Animation and Simulation '93*, Eurographics, pages 135–148. Eurographics, ISSN 1017-4656, September 1993. Proceedings of the Eurographics Workshop in Politechnical University of Catalonia, Spain, September 4–5, 1993.

[ED08] Elmar Eisemann and Xavier Décoret. Single-pass GPU solid voxelization for real-time applications. In Chris Shaw 0002 and Lyn Bartram, editors, *Graphics Interface*, ACM International Conference Proceeding Series, pages 73–80. ACM Press, 2008.

[EG07] Mathias Eitz and Lixu Gu. Hierarchical spatial hashing for real-time collision detection. In *Shape Modeling International*, pages 61–70. IEEE Computer Society, 2007.

[EHK+00] Bernd Eberhardt, Jens Hahn, Reinhard Klein, Wolfgang Straer, Andreas Weber, Wsi gris Arbeitsbereich, and Graphisch interaktive Systeme. Dynamic implicit surfaces for fast proximity queries in physically based modeling, June 28 2000.

[EL01] Stephen A. Ehmann and Ming C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Computer Graphics Forum*, 20(3), September 2001.

[Eri05] Christer Ericson. *Real-time Collision Detection*. Morgan Kaufmann, 2005.

[ES99] Jens Eckstein and Elmar Schömer. Dynamic collision detection in virtual reality applications. In V. Skala, editor, *WSCG'99 Conference Proceedings*, 1999.

[Eve01] Cass Everitt. Interactive order-independent transparency, September 01 2001.

[FB74] R. A. Finkel and J. L. Bentley. Quad trees : A data structure for retrieval on composite keys. *Acta Informatica*, 4(1) :1, 1974.

[FBAF08] François Faure, Sebastien Barbier, Jérémie Allard, and Florent Falipou. Image-based collision detection and response between arbitrary volume objects. In Markus H. Gross and Doug L. James, editors, *Symposium on Computer Animation*, pages 155–162. Eurographics Association, 2008.

[FF03] Christoph Fünfzig and Dieter W. Fellner. Easy realignment of k-dop bounding volumes. *Graphics Interface*, pages 257–264, 2003.

[FF04] Mauro Figueiredo and Terrence Fernando. An efficient parallel collision detection algorithm for virtual prototype environments. In *ICPADS*, pages 249–256. IEEE Computer Society, 2004.

[FL01] Susan Fisher and Ming C. Lin. Fast penetration depth estimation for elastic bodies using deformed distance fields. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2001*, volume 1, pages 330–336, July 17 2001.

[For89] Steven Fortune. Stable maintenance of point set triangulations in two dimensions. In *FOCS*, pages 494–499. IEEE, 1989.

[For94] MPI Forum. MPI : A message - passing interface standard. *The International Journal of Supercomputing and High Performance Computing*, pages 159–416, 1994.

[FS05] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In Michael Meißner and Bengt-Olaf Schneider, editors,

Graphics Hardware, pages 15–22, Los Angeles, California, 2005. Eurographics Association.

[Gan07] Shashidhara K. Ganjugunte. A survey on techniques for computing penetration depth. Duke University, 2007.

[GASF94] Alejandro Garcia-Alonso, Nicolàs Serrano, and Juan Flaquer. Solving the collision detection problem. *IEEE Computer Graphics & Applications*, 14(3) :36–42, 1994.

[GF90] Elmer G. Gilbert and Chek-Peng. Foo. Computing the distance between general convex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation*, 6(1) :53–61, February 1990.

[GGK06] Alexander Gress, Michael Guthe, and Reinhard Klein. GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum*, 25(3) :497–506, 2006.

[GGKM06] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTerasort : high performance graphics co-processor sorting for large database management. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, pages 325–336. ACM, 2006.

[GH89] Elmer G. Gilbert and S.M. Hong. A new algorithm for detecting the collision of moving objects. *IEEE*, 1989.

[GJK88] Elmer G. Gilbert, Daniel W. Johnson, and Sathiya S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4 :193–203, 1988.

[GKJ+05] Naga K. Govindaraju, David Knott, Nitin Jain, Ilknur Kabul, Rasmus Tamstorf, Russell Gayle, Ming C. Lin, and Dinesh Manocha. Interactive collision detection between deformable models using chromatic decomposition. *ACM Transactions on Graphics*, 24(3) :991–999, July 2005.

[GKLM07] Naga K. Govindaraju, Ilknur Kabul, Ming C. Lin, and Dinesh Manocha. Fast continuous collision detection among deformable models using graphics processors. *Computers & Graphics*, 31(1) :5–14, 2007.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. Obbtree : A hierarchical structure for rapid interference detection. In *Proceedings of the ACM Conference on Computer Graphics*, pages 171–180, New York, August 4–9 1996. ACM.

[GLM05a] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Fast and reliable collision detection using graphics processors. In *COMPGEOM : Annual ACM Symposium on Computational Geometry*, 2005.

[GLM05b] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Quick-cullide : fast inter- and intra-object collision culling using graphics hardware. In *SIGGRAPH '05 : ACM SIGGRAPH 2005 Courses*, page 218, New York, NY, USA, 2005. ACM.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI, Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999.

[GMD11] Loeiz Glondu, Maud Marchal, and Georges Dumont. Real-time simulation of brittle fracture using modal analysis. In *In submission in Siggraph SCA*, 2011.

[GME+00] Arthur D. Gregory, Ajith Mascarenhas, Stephen A. Ehmann, Ming C. Lin, and Dinesh Manocha. Six degree-of-freedom haptic display of polygonal models. In *IEEE Visualization*, pages 139–146, 2000.

[GO94] Elmer G. Gilbert and Chong Jin Ong. New distances for the separation and penetration of objects. In *ICRA*, pages 579–586, 1994.

[Got96] Stefan Gottschalk. Separating axis theorem. Tr96-024, Department of Computer Science, UNC Chapel Hill, 1996.

[Gre08] Simon Green. Cuda particles. Technical report, Nvidia, June 2008.

[Gre10] Simon Green. Particle simulation using cuda. Technical report, Nvidia, 2010.

[Gri05] Laurent Grisoni. *Vers une simulation physique temps réel multi-modèle (HDR)*. PhD thesis, Université des sciences et des technologies de Lille (LIFL), 2005.

[GRLM03] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. Cullide : Interactive collision detection between complex models in large environments using graphics hardware. In M. Doggett, W. Heidrich, W. Mark, and A. Schilling, editors, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 025–032, San Diego, California, 2003. Eurographics Association.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5) :14–20, 1987.

[Gut85] Antonin Guttman. R-Trees : A dynamic index structure for spatial searching. In Shamkant B. Navathe, editor, *Proceedings of the 10th ACM International Conference on Management of Data (SIGMOD)*, pages 47–57. ACM Press, 1985.

[GW07] Ilan Grinberg and Yair Wiseman. Scalable parallel collision detection simulation. *Proceedings of the IASTED International Conference*, pages 380–385, 2007.

[Hav] Havok. http ://www.havok.com/.

[HBD96] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *Performance Evaluation Review*, 24, 1996.

[HBZ90] Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. Geometric collisions for time-dependent parametric surfaces. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 39–48, August 1990.

[HDLM96] Merlin Hughes, Christopher DiMattia, Ming C. Lin, and Dinesh Manocha. Efficient and accurate interference detection for polynomial deformation. In *Computer Animation*, page 155. IEEE Computer Society, 1996.

[He99] Taosong He. Fast collision detection using quospo trees. In *SI3D*, pages 55–62, 1999.

[Her86] Martin Herman. Fast, three-dimensional collision-free motion planning. In *IEEE International Conference on Robotics and Automation*, pages 1056–1063, 1986.

[Her10] Everton Hermann. *Simulations Physiques Interactives sur des Architectures Multi-Core et Multi-GPU*. PhD thesis, Institut National Polytechnique de Grenoble - INPG, June 30 2010.

[HF07] M Hutter and A Fuhrmann. Optimized continuous collision detection for deformable triangle meshes. In *Proc. WSCG Š07*, pages 25–32, 2007.

[HFL07] Jesse C. Himmelstein, Etienne Ferre, and Jean-Paul Laumond. Swept volume approximation of polygon soups. In *ICRA*, pages 4854–4860. IEEE, 2007.

[HFR08] Everton Hermann, Francois Faure, and Bruno Raffin. Ray-traced collision detection for deformable bodies. In *GRAPP*, pages 293–299, 2008.

[HKM95] Martin Held, James T. Klosowski, and Joseph S. B. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. In *Canadian Conference on Computational Geometry*, 1995.

[HLC+97] Thomas C. Hudson, Ming C. Lin, Jonathan Cohen, Stefan Gottschalk, and Dinesh Manocha. V-collide : Accelerated collision detection for vrml. In Rikk Carey and Paul Strauss, editors, *VRML 97 : Second Symposium on the Virtual Reality Modeling Language*, New York City, NY, February 1997. ACM SIGGRAPH / ACM SIGCOMM, ACM Press.

[Hof89] C. M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, Cal., 1989.

[HRF09] Everton Hermann, Bruno Raffin, and François Faure. Interactive physical simulation on multicore architectures. In *Eurographics Workshop on Parallel and Graphics and Visualization, EGPGV'09, March, 2009*, Munich, Allemagne, 2009.

[HRF+10] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-GPU and multi-CPU parallelization for interactive physics simulations. *Europar*, September 2010.

[HTG03] Bruno Heidelberger, Matthias Teschner, and Markus H. Gross. Real-time volumetric intersections of deforming objects. In Thomas Ertl, editor, *VMV*, pages 461–468. Aka GmbH, 2003.

[HTG04] Bruno Heidelberger, Matthias Teschner, and Markus H. Gross. Detection of collisions and self-collisions using image-space techniques. In *WSCG*, pages 145–152, 2004.

[Hub93] Philip M. Hubbard. Interactive collision detection. In *Proceedings of the Symposium on Research Frontiers in Virtual Reality*, pages 24–32, San Jose, CA, USA, October 1993. IEEE Computer Society Press.

[Hub95] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3) :218–230, September 1995. ISSN 1077-2626.

[Hub96] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph*, 15(3) :179–210, 1996.

[HZG08] Qiming Hou, Kun Zhou, and Baining Guo. BSGP : bulk-synchronous GPU programming. *ACM Transactions on Graphics*, 27(3) :19 :1–19 :13, August 2008.

[JC98] David E. Johnson and Elaine Cohen. A framework for efficient minimum distance computations. In *ICRA*, pages 3678–3684, 1998.

[JMJC05] Andy M. Day Jose M. Juarez-Comboni. A multi-pass multi-stage multi-gpu collision detection algorithm. In *Graphicon 2005 Proceedings*, 2005.

[JP04] Doug L. James and Dinesh K. Pai. BD-tree : output-sensitive collision detection for reduced deformable models. *ACM Trans. Graph.*, 23(3) :393–398, 2004.

[JSL99] Ammar Joukhadar, Alexis Scheuer, and Christian Laugier. Fast contact detection between moving deformable polyhedra, October 1999.

[JTS+07] Lenka Jerabkova, Christian Terboven, Samuel Sarholz, Torsten Kuhlen, and Christian Bischof. Exploiting multicore architectures for physically based simulation of deformable objects in virtual environments. In *Virtuelle und Erweiterte Realität, 4. Workshop der GI-Fachgruppe VR/AR,*, 2007.

[JTT01] Pablo Jiménez, Federico Thomas, and Carme Torras. 3d collision detection : a survey. *Computers & Graphics*, 25(2) :269–285, 2001.

[KGL+98] S. Krishnan, M. Gopi, M. Lin, Dinesh Manocha, and A. Pattekar. Rapid and accurate contact determination between spline models using shelltrees. *Computer Graphics Forum*, 17(3) :315–326, 1998. ISSN 1067-7055.

[KGR94] Vipin Kumar, Ananth Y. Grama, and Vempaty Nageshwara Rao. Scalable load balancing techniques for parallel computers. *JPDC : Journal of Parallel and Distributed Computing*, 22, 1994.

[KGS97] Dong-Jin Kim, Leonidas J. Guibas, and Sung-Yong Shin. Fast collision detection among multiple moving spheres. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, pages 373–375, New York, June 4–6 1997. ACM Press.

[KHeY08] DukSu Kim, Jea-Pil Heo, and Sung eui Yoon. Pccd : Parallel continuous collision detection. Technical report, Dept. of CS, KAIST, 2008.

[KHH+09] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-Eui Yoon. HPCCD : Hybrid parallel continuous collision detection using CPUs and GPUs. *Comput. Graph. Forum*, 28(7) :1791–1800, 2009.

[KHI+07] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and Richard Rowe. Collision detection : A survey. *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 4046–4051, Oct. 2007.

[KLM02] Young J. Kim, Ming C. Lin, and Dinesh Manocha. Deep : Dual-space expansion for estimating penetration depth between convex polytopes. In *ICRA*, pages 921–926. IEEE, 2002.

[KMSZ98] James T. Klosowski, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1) :21–36, January 1998.

[KMW91] Ralph Kopperman, Paul R. Meyer, and Richard G. Wilson. A jordan surface theorem for three-dimensional digital spaces. *Discrete & Computational Geometry*, 6 :155–161, 1991.

[KOLM02] Young J. Kim, Miguel A. Otaduy, Ming C. Lin, and Dinesh Manocha. Six-degree-of-freedom haptic display using localized contact computations. In *Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pages 209–216, 2002.

[KP03] Dave Knott and Dinesh K. Pai. Cinder : Collision and interference detection in real-time using graphics hardware. In *Graphics Interface*, pages 73–80, 2003.

[KRR88] Vipin Kumar, Vempaty Nageshwara Rao, and K. Ramesh. Parallel depth first search on a ring architecture. *Proc. of the 1988 International Conference on Parallel Processing*, III, Algorithms and Applications :128–132, August 1988. U. Texas.

[KS95] Yoshifumi Kitamura and Andrew Smith. Parallel algorithms for real-time colliding face detection. *Robot and Human Communication*, pages 211–218, November 07 1995.

[KSM00] D. d'Aulignac K. Sundaraj and E. Mazer. A new algorithm for computing minimum distance, October 13 2000.

[KVLM04] Young J. Kim, Gokul Varadhan, Ming C. Lin, and Dinesh Manocha. Fast swept volume approximation of complex polyhedral models. *Computer-Aided Design*, 36(11) :1013–1027, 2004.

[LA04] Jyh-Ming Lien and Nancy M. Amato. Approximate convex decomposition of polygons. In *COMPGEOM : Annual ACM Symposium on Computational Geometry*, 2004.

[LA07] Jyh-Ming Lien and Nancy M. Amato. Approximate convex decomposition of polyhedra. In Bruno Lévy and Dinesh Manocha, editors, *Symposium on Solid and Physical Modeling*, pages 121–131. ACM, 2007.

[LAM01] Thomas Larsson and Tomas Akenine-Möller. Collision detection for continuously deforming bodies. *Eurographics*, February 04 2001.

[LAM03] Thomas Larsson and Tomas Akenine-Möller. Efficient collision detection for models deformed by morphing. In *The Visual Computer*, volume 19(2-3), pages 164–174. Springer, May 2003.

[LBCC01] Anatole Lécuyer, Jean-Marie Burkhardt, Sabine Coquillart, and Philippe Coiffet. "boundary of illusion : " an experiment of sensory integration with a pseudo-haptic system. In *VR*, pages 115–122, 2001.

[LC91] Ming C. Lin and John F. Canny. A fast algorithm for incremental distance calculation. Technical report, University of Berkeley, California, March 19 1991.

[LCAA08] Xavier Larrodé, Benoît Chanclou, Laurent Aguerreche, and Bruno Arnaldi. Openmask : an open-source platform for virtual reality. In *IEEE VR workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS) (2008)*, 2008.

[LCF05] Rodrigo G. Luque, João Luiz Dihl Comba, and Carla Maria Dal Sasso Freitas. Broad-phase collision detection using semi-adjusting BSP-trees. In Anselmo Lastra, Marc Olano, David P. Luebke, and Hanspeter Pfister, editors, *SI3D*, pages 179–186. ACM, 2005.

[LG98] Ming C. Lin and Stefan Gottschalk. Collision detection between geometric models : a survey. In Robert Cripps, editor, *Proceedings of the 8th IMA Conference on the Mathematics of Surfaces (IMA-98)*, volume VIII of *Mathematics of Surfaces*, pages 37–56, Winchester, UK, September 1998. Information Geometers.

[LG07] Scott Le Grand. Broad-phase collision detection with cuda. *GPU Gems 3 - Nvidia Corporation*, 2007.

[LGLM00] Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. Fast distance queries with rectangular swept sphere volumes. In *ICRA*, pages 3719–3726. IEEE, 2000.

[LGS+09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2) :375–384, 2009.

[LHLK10] Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J. Kim. Real-time collision culling of a million bodies on graphics processing units. *ACM Trans. Graph*, 29(6) :154, 2010.

[Lin93] Ming Chieh Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Graduate Division of the University of California at Berkeley, 1993.

[LK02] Orion Sky Lawlor and Laxmikant V. Kalée. A voxel-based parallel collision detection algorithm. In *Proceedings of the 16th International Conference on Supercomputing (ICS-02)*, pages 285–293, New York, June 22–26 2002. ACM Press.

[LM95] Ming C. Lin and Dinesh Manocha. Fast interference detection between geometric models. *The Visual Computer*, 11(10) :542–561, 1995.

[LM06] Mark Lewis and Berna L. Massingill. Multithreaded collision detection in java. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*

& Conference on Real-Time Computing Systems and Applications (PDPTA'06), volume 1, pages 583–592, Las Vegas, Nevada, USA, June 2006. CSREA Press.

[LMM10] C. Lauterbach, Q. Mo, and D. Manocha. gproximity : Hierarchical gpu-based operations for collision and distance queries. In *Computer Graphics Forum (EUROGRAPHICS Proceedings)*, volume 29, pages 419–428, June 2010.

[LTT91] Benoit Lafleur, Nadia Magnenat Thalmann, and Daniel Thalmann. Cloth animation with self-collision detection, April 26 1991.

[MAC+02] David Margery, Bruno Arnaldi, Alain Chauffaut, Stéphane Donikian, and Thierry Duval. Openmask : Multi-threaded | Modular animation and simulation Kernel | Kit : a general introduction. In *Virtual Reality International Conference*, 2002.

[Mey86] Walter Meyer. Distances between boxes : Applications to collision detection and clipping. *IEEE Robotics and Automation. Proceedings*, 3 :597–602, 1986.

[Mir97] Brian Mirtich. Efficient algorithms for two-phase collision detection. Technical report, Mitsubishi Electric Research Laboratories, December 1997.

[Mir98] Brian Mirtich. V-clip : Fast and robust polyhedral collision detection. *ACM Trans. Graph*, 17(3) :177–208, 1998.

[MKE03] Johannes Mezger, Stefan Kimmerle, and Olaf Etzmuß. Hierarchical techniques in collision detection for cloth animation. In *WSCG*, 2003.

[MKF03] Philippe Meseure, Abderrahmane Kheddar, and François Faure. *Détection des collisions et Calcul de la réponse*, 2003.

[Möl97] Tomas Möller. A fast triangle-triangle intersection test. *JGTOOLS : Journal of Graphics Tools*, 2, 1997.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8) :114–117, 1965.

[MRS08] T. Morvan, M. Reimers, and E. Samset. High performance gpu-based proximity queries using distance fields. *Comput. Graph. Forum*, 27(8) :2040–2052, 2008.

[MTES97] Junichiro Makino, Makoto Taiji, Toshikazu Ebisuzaki, and Daiichiro Sugimoto. GRAPE-4 : A massively parallel special-purpose computer for collisional N-body simulations. *Astrophysical Journal*, 480(1) :432–446, 1997.

[MW88] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4), August 1988.

[NAT90] Bruce Naylor, John Amanatides, and William Thibault. Merging BSP trees yields polyhedral set operations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 115–124, August 1990.

[Nay92] Bruce F. Naylor. Interactive solid geometry via partitioning trees. In *Graphics Interface '92*, pages 11–18, May 1992.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Proceedings of Eurographics 2004*, pages 21–51. Blackwell Publishing Ltd, 2007. Warning : the year was guessed out of the URL.

[OvdS96] Mark H. Overmars and A. Frank van der Stappen. Range searching and point location among fat objects. *ALGORITHMS : Journal of Algorithms*, 21, 1996.

[Ove92] Mark H. Overmars. Point location in fat subdivisions. *IPL : Information Processing Letters*, 44, 1992.

[Pen90] Alex P. Pentland. Computational complexity versus simulated environments. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 185–192, March 1990.

[PG86] M. Pratt and A. Geisow. *Surface/surface intersection problems*, chapter 2, pages 117–142. Clarendon Press, 1986.

[PG94] I. J. Palmer and R. L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2) :105–116, March 1994.

[Phy] Physx. http ://www.nvidia.com/object/physx_new.html.

[PKS10] Simon Pabst, Artur Koch, and Wolfgang Straβer. Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. In *Computer Graphics Forum*, volume 29, pages 1605–16212, July 2010.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry : an Introduction*. Springer-Verlag, 1985.

[Qui94] Sean Quinlan. Efficient distance computation between non-convex objects. In *ICRA*, pages 3324–3329, 1994.

[RCE05] Abdennour El Rhalibi, Steve Costa, and David England. Game engineering for a multiprocessor architecture. In *DIGRA Conf*, 2005.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'REILLY, 2007.

[RKC02a] Stéphane Redon, Abderrahmane Kheddary, and Sabine Coquillart. Fast continuous collision detection between rigid bodies. *Computer Graphics Forum*, 21(3) :279–288, September 2002.

[RKC02b] Stéphane Redon, Abderrahmane Kheddar, and Sabine Coquillart. Hierarchical back-face culling for collision detection. *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2002*, 3 :3036 – 3041, July 02 2002.

[RKLM04] Stéphane Redon, Young J. Kim, Ming C. Lin, and Dinesh Manocha. Fast continuous collision detection for articulated models. In *ACM Symposium on Solid Modeling and Applications (2004)*, April 15 2004.

[RL85] Nick Roussopoulos and Daniel Leifer. Direct spatial search on pictorial databases using packed R-trees. In *ACM SIGMOD*, May 1985.

[Roc70] R. T. Rockafellar. *Convex Analysis*. Princeton University Press, 1970.

[RS94] Alexander Reinefeld and Volker Schnecke. Work-load balancing in highly parallel depth-first search. *Scalable High Performance Computing Conference*, pages 773–780, 1994.

[RS06] Bruno Raffin and Luciano Soares. Pc clusters for virtual reality. *IEEE-VR*, pages 215–222, 25-29 March 2006.

[Sam90] Hanan Samet. *Applications of Spatial Data Structures*. Book, 1990.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee : a many-core x86 architecture for visual computing. *ACM SIGGRAPH'08 Transactions on Graphics*, 27(3), August 2008.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3) :66–73, May/June 2010.

[SGwHS98] Jonathan Shade, Steven J. Gortler, Li wei He, and Richard Szeliski. Layered depth images. In *SIGGRAPH*, pages 231–242, 1998.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS*, pages 1–10. IEEE, 2009.

[SHMA96] Yuichi Sato, Mitsunori Hirata, Tsugito Maruyama, and Yuichi Arita. Efficient collision detection using fast distance-calculation algorithms for convex and non-convex objects. In *Proc. IEEE Intern. Conf. on Robotics and Automation (Minneapolis, Minnesota 1996)*, pages 771–778. IEEE, 1996.

[SKTK95] Andrew Smith, Yoshifumi Kitamura, Haruo Takemura, and Fumio Kishino. A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. *Proc. IEEE Virtual Reality Annual International Symposium*, pages 136–145, March 1995.

[SLAA09] Jean Sreng, Anatole Lécuyer, Claude Andriot, and Bruno Arnaldi. Spatialized haptic rendering : Providing impact position information in 6DOF haptic simulations using vibrations. In *IEEE Virtual Reality*, pages 3–9, 2009.

[SLY99] C. J. Su, F. H. Lin, and B. P. Yen. An adaptive bounding object based algorithm for efficient and precise collision detection of CSG-represented virtual objects, February 20 1999.

[SOM04] Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha. Difi : Fast 3D distance field computation using graphics hardware. *Comput. Graph. Forum*, 23(3) :557–566, 2004.

[SP95] Vladimir Savchenko and Alexander Pasko. Collision detection for functionally defined deformable objects. In *Implicit Surfaces '95*, April 1995.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree : A dynamic index for multi-dimensional objects. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 507, Brighton, England, August 1987.

[SSIF09] Andrew Selle, Jonathan Su, Geoffrey Irving, and Ronald Fedkiw. Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction. *IEEE Trans. Vis. Comput. Graph*, 15(2) :339–350, 2009.

[Sun90] V. S. Sunderam. Pvm : A framework for parallel distributed computing. *Concurrency : Practice and Experience*, 2(4) :315–339, 1990.

[SWF+93] John M. Snyder, Adam R. Woodbury, Kurt Fleischer, Bena Currin, and Alan H. Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. *Computer Graphics*, 27(Annual Conference Series) :321–334, 1993.

[TB07] Bernhard Thomaszewski and Wolfgang Blochinger. Physically based simulation of cloth on distributed memory architectures. *Parallel Computing*, 33(6) :377–390, 2007.

[TBW09] Daniel J. Tracy, Samuel R. Buss, and Bryan M. Woods. Efficient large-scale sweep and prune methods with AABB insertion and removal. In *VR*, pages 191–198. IEEE, 2009.

[Tch10] Loïc Tching. *Traitement générique des interactions haptiques pour l'assemblage d'objets issus de CAO*. Computer science, INSA de Rennes, February 2010.

[Til84] Robert B. Tilove. A null-object detection algorithm for constructive solid geometry. *Communications of the ACM*, 27 :684–694, July 1984.

[TJBDS09] Kim Tae-Joon, Moon Bochang, Kim Duksu, and Eui Yoon Sung. Racbvhs : Random-accessible compressed bounding volume hierarchies. In *SIGGRAPH Talks*. ACM, August 2009.

[TKH+05] Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, Marie-Paule Cani, François Faure, Nadia Magnenat-Thalmann, Wolfgang Straßer, and Pascal Volino. Collision detection for deformable objects. *Comput. Graph. Forum*, 24(1) :61–81, 2005.

[TLW11] Chen Tang, Sheng Li, and Guopin Wang. Fast continuous collision detection using parallel filter in subspace. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 71–80, New York, NY, USA, 2011. ACM.

[TMLT11] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. Collision-streams : fast gpu-based collision detection for deformable models. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 63–70, New York, NY, USA, 2011. ACM.

[TMT08] Min Tang, Dinesh Manocha, and Ruofeng Tong. Multi-core collision detection between deformable models. In *Computers & Graphics*, 2008.

[TPB08] Bernhard Thomaszewski, Simon Pabst, and Wolfgang Blochinger. Parallel techniques for physically based simulation on multi-core processor architectures. *Computers & Graphics*, 32(1) :25–40, 2008.

[TTS06] Oren Tropp, Ayellet Tal, and Ilan Shimshoni. A fast triangle to triangle intersection test for collision detection. *Journal of Visualization and Computer Animation*, 17(5) :527–535, 2006.

[Tur89] Greg Turk. Interactive collision detection for molecular graphics. Master's thesis, University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, North Carolina, 1989.

[Van94] George Vaněček, Jr. Back-face culling applied to collision detection of polyhedra. *The Journal of Visualization and Computer Animation*, 5(1) :55–63, January–March 1994.

[VCC98] B. C. Vemuri, Y. Cao, and L. Chen. Fast collision detection algorithms with applications to particle flow. *Computer Graphics Forum*, 17(2) :121–134, 1998. ISSN 1067-7055.

[VT95] Pascal Volino and Nadia Magnenat Thalmann. Collision and self-collision detection : Efficient and robust solutions for highly deformable surfaces. In Dimitri Terzopoulos and Daniel Thalmann, editors, *Computer Animation and Simulation '95*, pages 55–65. Eurographics, Springer-Verlag, September 1995.

[Wal07] Ingo Wald. On fast construction of SAH-based bounding volume hierarchies, 2007.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1) :52–69, January 1984.

[WHQ01] Gao Shuming Wan Huagen, Fan Zhaowei and Peng Qunsheng. A parallel collision detection algorithm based on hybrid bounding volume hierarchy. *CAD/GraphicsŠ2001*, August 2001.

[YFR06] Thomas Y. Yeh, Petros Faloutsos, and Glenn Reinman. Enabling real-time physics simulation in future interactive entertainment, 2006.

[Zac95] Gabriel Zachmann. The boxtree : Enabling real-time and exact collision detection of arbitrary polyhedra. In *Informal Proc. First Workshop on Simulation and Interaction in Virtual Environments, SIVE 95*, pages 104–112, University of Iowa, Iowa City, July 1995.

[Zac98] Gabriel Zachmann. Rapid collision detection by dynamically aligned dop-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium ; VRAIS '98*, pages 90–97, Atlanta, Georgia, March 1998.

[Zac01] Gabriel Zachmann. Optimizing the collision detection pipeline. In *Proc. of the First International Game Technology Conference (GTEC)*, January 2001.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5) :126 :1–126 :11, December 2008.

[ZL03] Gabriel Zachmann and Elmar Langetepe. Geometric data structures for computer graphics. In *Siggraph 2003 Tutorial 16*, 2003.

Liste des Figures

Liste des Algorithmes

Liste des Tableaux

Résumé

Les environnements de réalité virtuelle devenant de plus en plus complexes et de très grandes dimensions, un niveau d'interaction temps-réel devient impossible à garantir. En effet, de par leur complexité, due à une géométrie détaillée et aux propriétés physiques spécifiques, ces environnements large échelle engendrent un goulet d'étranglement calculatoire critique sur les algorithmes de simulation physique. Nous avons focalisé nos travaux sur la première étape de ces algorithmes qui concerne la détection de collision, car les problématiques font partie intégrante de ce goulet d'étranglement et leur complexité peut parfois se révéler quadratique dans certaines situations.

Le profond bouleversement que subissent les architectures machines depuis quelques années ouvre une nouvelle voie pour réduire le goulet d'étranglement. La multiplication du nombre de cœurs offre ainsi la possibilité d'exécuter ces algorithmes en parallèle sur un même processeur. Dans le même temps, les cartes graphiques sont passées d'un statut de simple périphérique d'affichage graphique à celui de supercalculateur. Elles jouissent désormais d'une attention toute particulière de la part de la communauté traitant de la simulation physique.

Afin de passer au large échelle et d'être générique sur la machine d'exécution, nous avons proposé des modèles unifiés et adaptatifs de correspondance entre les algorithmes de détection de collision et les architectures machines de type multi-cœur et multi-GPU. Nous avons ainsi défini des solutions innovantes et performantes permettant de réduire significativement le temps de calcul au sein d'environnements large échelle tout en assurant la pérennité des résultats. Nos modèles couvrent l'intégralité du pipeline de détection de collision en se focalisant aussi bien sur des algorithmes de bas ou de haut niveau. Nos modèles multi-cœur, GPU et multi-GPU allient différentes techniques de subdivision spatiale à des algorithmes basés topologie ainsi que des techniques d'équilibrage de charge basées sur le vol de données. Notre solution hybride permet d'accroitre l'espace et le temps de calcul ainsi que le passage au large échelle. L'association de ces nouveaux algorithmes nous a permis de concevoir deux modèles d'adaptation algorithmique dynamique basés, ou non, sur des scénarios de pré-calcul hors-ligne. Enfin, il nous est apparu indispensable d'ajouter au pipeline de détection de collision une nouvelle dimension révélant la prise en compte des architectures pour une exécution optimale. Grâce à ce formalisme, nous avons proposé un nouveau pipeline de détection de collision offrant une granularité de parallélisme sur processeurs multi-cœur. Il permet une exécution simultanée des différentes étapes du pipeline ainsi qu'un parallélisme interne à chacune de ces étapes.

Abstract

Virtual reality environments are becoming increasingly large and complex and real-time interaction level is becoming difficult to stably insure. Indeed, because of their complexity, detailed geometry and specific physical properties, these large scale environments create a critical computational bottleneck on physical algorithms. Our work focused on the first step of the physical process : the collision detection. These algorithms can sometimes have a quadratic complexity. Solving and simplifying the collision detection problem is integral to alleviating this bottleneck.

Hardware architectures have undergone extensive changes in the last few years that have opened new ways to relieve this computational bottleneck. Multiple processor cores offer the ability to execute algorithms in parallel on one single processor. At the same time, graphics cards have gone from being a simple graphical display device to a supercomputer. These supercomputers now enjoy attention from a specialized community dealing solely with physical simulation.

To perform large scale simulations and remain generic on the runtime architecture, we proposed unified and adaptive mapping models between collision detection algorithms and the runtime architecture using multi-core and multi-GPU architectures. We have developed innovative and effective solutions to significantly reduce the computation time in large scale environments while ensuring the stability and reproducibility of results. Our models cover the global collision detection pipeline, focusing both high and low level algorithms. Our models based on multi-core, GPU and multi-GPU combine different techniques of spatial subdivision algorithms based on topology and load balancing techniques based on data stealing. Our hybrid solution enables us to increase space and computing time within large scale virtual environments. The coupling of these new algorithms led us to develop two models of dynamic algorithmic adaptation based (or not) on off-line precomputed scenarios. Finally, it became necessary to add a new dimension to the collision detection pipeline taking into account of the architecture for optimal execution. With this formalism, we proposed a new pipeline of collision detection with a granularity of parallelism on multicore processors or multi-GPU platforms. It enables simultaneous execution of different stages of the pipeline and a parallel internal to each of these steps.

INSA RENNES

Institut National des Sciences Appliquées de Rennes

20, Avenue des Buttes de Coësmes . CS 14315 . F-35043 Rennes Cedex

Tel : 02 23 23 82 00 – Fax : 02 23 23 83 96