HAL
open science

Détection de Collision : D'un Algorithme sur Multi-Cœurs à une Nouvelle Dimension dans le Pipeline 3D

Quentin Avril, Valérie Gouranton, Bruno Arnaldi

To cite this version:

Quentin Avril, Valérie Gouranton, Bruno Arnaldi. Détection de Collision : D'un Algorithme sur Multi-Cœurs à une Nouvelle Dimension dans le Pipeline 3D. AFRV, Dec 2009, Lyon, France. hal-00475524

HAL Id: hal-00475524

https://hal.science/hal-00475524v1

Submitted on 22 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Détection de Collision : D'un Algorithme sur Multi-Cœurs à une Nouvelle Dimension dans le Pipeline 3D

Quentin Avril *
INRIA-INSA de Rennes

Valérie Gouranton †
INRIA-INSA de Rennes

Bruno Arnaldi ‡
INRIA-INSA de Rennes

Abstract

Depuis quelques années les architectures machines subissent une impressionnante évolution. Les architectures processeurs évoluent d'un simple cœur aux multi et many cœurs, parallèlement les cartes graphiques confirment leur statut de "supercalculateur" en dépassant largement la puissance de calcul des processeurs. Face à cette évolution, les nouvelles tendances d'optimisation de la détection de collision consistent à proposer une solution qui s'adapte à l'architecture d'exécution. Nous présentons dans cet article, deux contributions dans le domaine de la détection de collision au sein de larges environnements virtuels. Nous présentons tout d'abord, une première façon de paralléliser l'étape initiale (*broad-phase*) du pipeline de détection de collision sur une architecture multi-coeurs. Nous décrivons ensuite une nouvelle manière de représenter le pipeline en prenant en compte l'architecture d'exécution. L'algorithme de broad-phase que nous avons utilisé est celui du "Sweep and Prune" [6], nous l'avons adapté à une utilisation *multi-thread*. Afin de manipuler au mieux un ou plusieurs *threads* par coeur, les sections critiques et l'attente des *threads* doivent être minimisées. Notre modèle fonctionne sur des architectures n-cœurs et divise par 3 le temps d'exécution sur une architecture 8-cœurs.

Index Terms: I.6.8 [Collision Detection]: Broad phase algorithms—Multi-cores Architecture

1 Introduction

Les maquettes numériques et les applications industrielles de réalité virtuelle deviennent de plus en plus dimensionnées. Le niveau de performance n'est plus satisfaisant pour une interaction de l'utilisateur en temps-réel, au sein de larges environnements virtuels. Depuis plus d'une trentaine d'années, la détection de collision fait partie intégrante des principaux goulots d'étranglement des applications de réalité virtuelle. Depuis peu, la recherche dans le domaine s'intéresse à l'évolution des architectures *hardware* et de leur simplicité croissante d'utilisation, afin d'accroitre la puissance de calcul des algorithmes de détection de collision. De récents articles traitent de ce nouveau type de problème : accélérer la détection en utilisant les spécificités de ces nouvelles architectures (Multi-cœurs et GPU) [16, 24]. Au sein du domaine des algorithmes de détection de collision, nous pouvons noter que les plus récentes contributions possèdent un très faible coût de calcul et offrent un niveau de détection très élevé. Mais utiliser ces algorithmes avec des millions d'objets dans un environnement large échelle, montre généralement qu'ils sont inappropriés pour une interaction temps réel. Prendre le plus efficace et le plus rapide des algorithmes de détection de collision et l'exécuter sur une architecture parallèle n'assurera pas les meilleures performances car les algorithmes doivent être adaptés. Il était encore possible il y a quelques années de s'appuyer sur la loi de Moore prévoyant la multiplication du nombre de transistors tous les deux ans. Nous savons désormais que cette multiplication a atteint certaines limites physiques et que la tendance est maintenant aux architectures multi et many cœurs. La stratégie des spécialistes *hardware* est d'augmenter le nombre de cœurs et d'en réduire la fréquence afin d'obtenir une faible consommation d'énergie et d'émission de chaleur. Par conséquent, une application de réalité virtuelle utilisant actuellement un seul et unique cœur va, au fur et à mesure des années, devenir de plus en plus longue à s'exécuter si nous ne nous soucions pas de son adaptation aux architectures parallèles. Les cartes graphiques sont également sujettes depuis plusieurs années à une impressionnante évolution. Il apparait donc essentiel qu'une approche basée temps réel pour la détection de collision ainsi centrée sur les nouvelles performances offertes par le *hardware* ne peut pas être ignorée. Nous avons étudié le pipeline de détection de collision en s'intéressant dans un premier temps à la broad-phase (qui est en charge de la réduction du nombre de paires d'objets à tester). Nous présentons en section 2 l'état de l'art de la détection de collision, celui de l'architecture machine est détaillée en section 3 et la section 4 présente le lien entre ces deux domaines. Au travers de la section 5 nous présentons notre algorithme de "Sweep and Prune" adapté à la parallélisation suivi par la nouvelle vue 3D du pipeline de détection de collision. Nos résultats sont illustrés dans la section 6 puis nous concluons en ouvrant la problématique sur de futurs travaux.

*e-mail: quentin.avril@irisa.fr
†e-mail: valerie.gouranton@irisa.fr
‡e-mail: bruno.arnaldi@irisa.fr

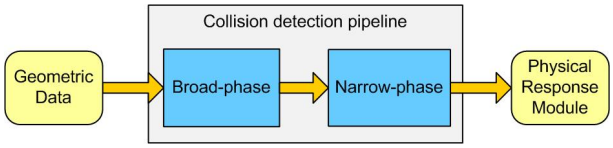

Figure 1: Pipeline de détection de collision [Hubbard 95].

2 Détection de collision

La détection de collision est un domaine très vaste traitant d'un problème en apparence simple : déterminer si deux (ou plusieurs) objets sont, ou vont être, en collision. Elle est utilisée dans des domaines diversifiés tels que celui des simulations basées physique, l'animation, la robotique, les simulations mécaniques (médical, biologique, industrie automobile), les applications haptiques et les jeux vidéos. Tous ces domaines applicatifs possèdent des contraintes différentes (temps-réel, efficacité, précision...); Ils engendrent donc un grand nombre de sous-problématiques (objets convexes ou concaves, simulation 2-Body ou N-Body, objets rigides ou objets déformables, méthodes discrètes ou continues...). Pour plus de détails, vous pouvez vous référés à d'excellents états de l'art sur le sujet [15, 18, 21, 25].

Etant donné n objets évoluant au sein d'un environnement virtuel, tester toutes les paires d'objets une à une consiste à effectuer n^2 tests. Et plus n est élevé, plus le calcul se transforme en goulot d'étranglement calculatoire. Afin de réduire cette complexité, les

algorithmes de détection de collision sont représentés et construits à l'aide d'un pipeline (cf Figure 1) [14]. Ce dernier est composé de deux parties principales: la broad-phase et la narrow-phase. Le but de ce pipeline est, en appliquant des filtres successifs, de réduire la $O(n^2)$ complexité. Ces filtres sont organisés de manière à permettre une efficacité et une précision croissante tout au long de la traversée du pipeline. Nous présentons par la suite les deux principales parties du pipeline, la broad phase en section 2.1 et la narrow phase en section 2.2.

2.1 Broad phase

La broad phase est en charge d'effectuer des tests rapides et efficaces sur les paires d'objets afin de déterminer si elles sont potentiellement en collision ou non. Les algorithmes de broad phase peuvent être classés entre quatre familles [18], la méthode de force brute, le partitionnement spatial, les méthodes topologiques et celles cinématiques.

Méthodes de force brute - Cette approche se base sur la comparaison de l'ensemble des volumes englobants des objets afin de savoir s'ils sont ou non en collision. Cette méthode est très couteuse car elle effectue pour n objets, n^2 tests. Dans la littérature, différents volumes englobants ont été proposés tels les sphères, les boites englobantes alignées sur les axes (AABB) [5], les boites englobantes orientées (OBB) [9] et divers autres (k-DOP, cônes, cylindres etc).

Méthodes de partitionnement spatial - Cette méthode se base sur le principe que deux objets situés dans des régions distantes de l'espace n'ont aucune chance d'entrée en collision dans les prochains pas de temps. Différentes méthodes ont ainsi été proposées pour diviser l'espace: les grilles régulières, les octree [3], les quadtree, les BSP, les K-d tree [4] ou encore les voxels.

Méthodes topologiques - Ces méthodes sont basées sur la position des objets les uns par rapport aux autres. Un couple d'objets étant trop loin l'un de l'autre est mis de côté. L'une des approches la plus connue et la plus utilisée, appelée *Sweep and Prune* [6], consiste à projeter les coordonnées des objets sur les axes et à en détecter des éventuels chevauchement. Un chevauchement des coordonnées de deux objets sur les trois axes (x, y, z) signifie que les volumes englobants sont en collision

Méthodes cinématiques - Ces approches tiennent compte du mouvement des objets, si deux objets se déplacent vers des directions opposées, ils ne peuvent pas rentrer en collision. Vanecek [26] a utilisé la cinématique des objets couplée à une méthode de filtrage des faces cachées pour accélérer la détection de collision.

2.2 Narrow phase

Les paires d'objets potentiellement en collision sont ensuite transmises à la narrow phase qui est en charge de l'exécution d'un test exact de collision. Nous pouvons classer ces algorithmes en quatre familles [18]: caractéristiques, simplex, espace image et volume englobants.

Algorithmes basés caractéristiques - Cette famille d'algorithmes travaille avec les primitives des objets (faces, arêtes et sommets). Le premier algorithme fut proposé en 1991 par Lin et Canny [20] appelé l'algorithme LC ou *"Voronoï marching"*. Il divise l'espace autour des objets en régions de Voronoï ce qui permet de pouvoir détecter quelles sont les primitives les plus proches entre deux polyèdres.

Algorithmes basés simplex - L'algorithme le plus célèbre de cette famille est celui du GJK [8] qui utilises la différence de Minkowski sur les polyèdres. Deux objets convexes sont en collision si et seulement si leur différence de Minkowski contient l'origine du repère dans lequel ils sont projetés.

Algorithmes basés image - Ce type d'algorithme fonctionne en utilisant les requêtes d'occlusion dans l'espace image, bien adaptés à l'utilisation sur carte graphique. Ils "rasterisent" les objets afin de procéder à des tests 2D ou 2.5D de chevauchement [2].

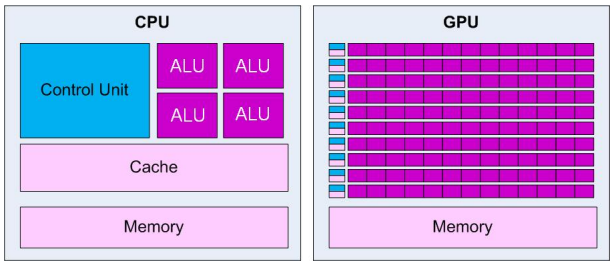

Figure 2: Comparaison de l'architecture CPU & GPU

Algorithmes basés volumes englobants - La plupart des stratégies utilise des hiérarchies de volumes englobants pour effectuer les tests de collision car cela augmente de façon notoire les performances et le temps de calcul. Ce type de hiérarchie permet de représenter les différents volumes englobant d'un objet sous forme d'arbre (arbre binaire, quadtree, octree ...) afin de réduire le nombre de tests à mettre en œuvre. Pour plus de détails, une description de ces hiérarchies et une comparaison de leur performance existent [18]. Les objets déformables représentent un vrai challenge pour les hiérarchies de volume englobant car ils nécessitent une constante mise à jour [5, 25].

3 Évolution de l'architecture

Depuis plusieurs années, les composants hardware bénéficient d'une impressionnante évolution comme le matériel graphique ou bien les architectures multicœurs. Afin de tirer pleinement les avantages de ces nouvelles architectures, des outils de simplification d'utilisation sont apparus permettant ainsi aux non-experts hardware de les utiliser. Nous présentons dans la section 3.1 l'évolution du hardware puis celle du software dans la section 3.2.

3.1 Évolution hardware

Nous présentons ici l'évolution du hardware pouvant être utilisé pour améliorer les algorithmes de détection de collision. Nous commençons par les cartes graphiques suivies des architectures multi et many cœurs.

3.1.1 Hardware graphiques

Contrairement au CPU, les "Graphics Processing Unit" (GPU) ont subi, ces dernières années, une importante évolution en terme de puissance de calcul. En partant d'unités à fonctions fixes, ils ont évolué vers un pipeline graphique de plus en plus programmable. La principale raison expliquant cette évolution par rapport au CPU réside dans le contrôle du flux de données. En effet, un CPU est un processeur généraliste (cf. Figure 2) qui manipules des données ordinaires possédant un fort niveau de dépendances. Plusieurs de ses composants sont en charge du contrôle du flux de données. A l'inverse les processeurs GPU sont très bien adaptés à des calculs hautement parallélisables du fait de l'indépendance des données qu'ils manipulent. Ils ne nécessitent donc pas de contrôle sophistiqué du flux de données. Par exemple, la carte Nvidia[1] Geforce GTX 295 peut développer 1.7 TFlops et l'ATI[2] Radeon HD 4890, 1.6 TFlops.

La bande passante sur un GPU est aussi supérieur à celle d'un CPU [22], mais un des problèmes fondamentale des calculs sur le GPU est la bande passante entre CPU et GPU. Un autre point essentiel pour une éventuelle voie d'optimisation est lié aux différents niveaux mémoire présents sur les cartes mémoires. En effet,

[1] Nvidia - http://www.nvidia.com/
[2] AMD - http://www.amd.com/us-en/

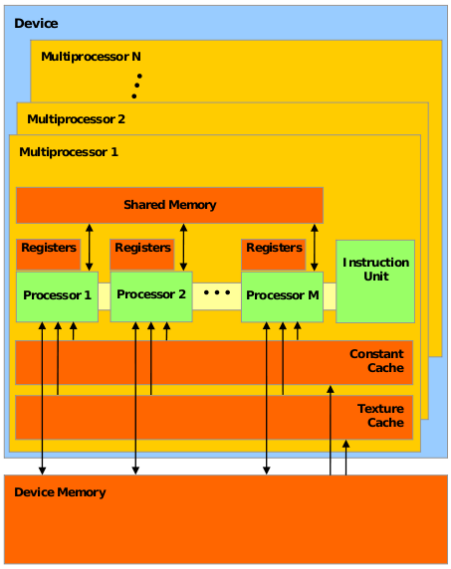

Figure 3: Niveaux de hiérarchie mémoire sur une Nvidia Geforce 8.

comme l'illustre la Figure 3, différents type de mémoire (registres, mémoires caches) sont partagés à plusieurs niveaux.

3.1.2 Architecture multi et many cœurs

Depuis 1965, les processeurs suivaient la fameuse loi de Gordon Moore prédisant, tous les deux ans, la multiplication du nombre de transistors sur un microprocesseur. Nous savons désormais que cette multiplication est bloquée par les limites physiques. Pour faire face à ce blocage, la tendance s'oriente vers la duplication des cœurs. Le premier ordinateur personnel muni d'un core-duo est arrivé en 2005 avec AMD. En 2006, Sun[3] a présenté son nouvel octo-cœurs baptisé *Niagara2*. En 2008, Intel[4] a annoncé un 32 cœurs appelé *Larrabee*. Cette rapide évolution du nombre de cœurs a laissé émerger un nouveau concept appelé le many-cœur qui permet une adaptation dynamique du nombre de cœurs actifs en fonction des besoins de l'utilisateur.

3.2 Software evolution

Les architectures machines étant en mesure de fournir un haut niveau de performance, Pour exploiter pleinement cette puissance, il est indispensable d'utiliser des outils adaptés. N'étant pas dans l'optique de proposer une nouvelle disposition d'architecture machine ni même une nouvelle stratégie de gestion de cache, nous nous sommes focalisés sur les outils disponibles permettant d'utiliser les nouvelles architectures hardware. Nous présentons rapidement par la suite le GPGPU et la programmation parallèle.

3.2.1 GPGPU

General-purpose Processing on Graphics Processing Unit (GPGPU) est la technique permettant d'effectuer sur les cartes graphiques des calculs traditionnellement réservés au CPU. Pour plus détails, nous nous sommes référés à un excellent état de l'art sur le domaine [22]. L'utilisation du matériel graphique permettant d'accélérer les calculs n'est pas récente mais la véritable révolution s'est produite après les années 2000 avec l'introduction des shaders permettant d'effectuer du calcul matriciel plus complexe. Depuis

[3]Sun - http://www.sun.com

[4]Intel - http://www.intel.com/

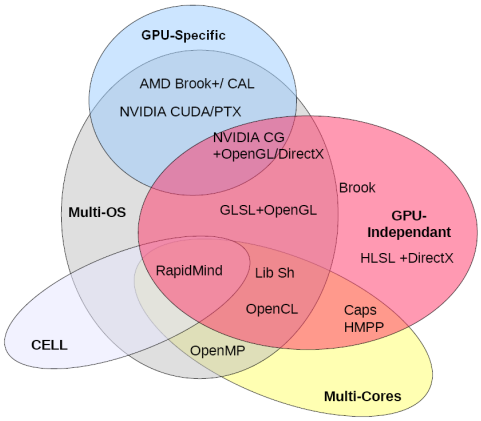

Figure 4: Différents niveaux de langage de GPGPU

2003 une section *SIGGRAPH*[5] est entièrement dédié au GPGPU. *Brook* (ATI) fut le premier language permettant d'utiliser le GPU comme un co-processeur pour les calculs parallèles. En 2007, Nvidia développa un langage et un outil appelé *CUDA*[6] (Compute Unified Device Architecture). Ce langage exploite la puissance du GPU en utilisant les principes de la programmation parallèles par threads. ATI/AMD a également développé son propre language *Brook+*. Sur la Figure 4, nous présentons différents langages utilisés pour le GPGPU.

OpenCL (Open Computing Language), mis en place par Khronos[7], est le premier standard libre pour le GPGU. C'est un ensemble de bibliothèques permettant d'écrire des programmes exécutables sur de nombreuses plateformes (CPUs, GPUs et autres processeurs). Les spécifications d'OpenCL 1.0 et les fichiers d'en tête sont disponibles mais ne sont pour le moment pas encore entièrement implémentés.

3.2.2 La Programmation Parallèle

La programmation parallèle a été très largement étudiée dans le domaine informatique. Sa première exploitation fut mis en place afin d'éviter qu'une tâche monopolise toute la machine pendant son exécution. Cela concernait les vieilles applications fonctionnant sur un seul processeur simple cœur sans réel gain significatif. La parallélisation s'est ensuite étendue à des architectures multi processeurs afin d'accroitre la vitesse de lourds calculs scientifiques. Depuis leur apparition, les architectures multicœurs ont donné une nouvelle dimension à la programmation parallèle.

OpenMP [7]est un standard mis en place par des industriels; Il cible trois langages: Fortran, C et C++. Son originalité réside dans la déclaration des blocs à paralléliser. L'implémentation n'y est pas réalisée dans le langage utilisé mais grâce à un ensemble de directives placées dans les zones à paralléliser. Les directives sont insérées à l'aide de "pragmas" dans le code et le compilateur prend ensuite le relais pour générer le code parallèle.

Intel Threading Building Blocks (TBB) [23] est une librairie C++ offrant une abstraction des détails de l'architecture et du mécanisme des threads. Intel TBB permet de tirer profit des architectures multi-cœurs sans en être un expert en multi-threading.

[5]ACM Siggraph - http://www.siggraph.org/

[6]Nvidia CUDA - http://www.nvidia.com/object/cudahome.html

[7]OpenCL - http://www.khronos.org/opencl/

4 Détection de Collision et Architecture

Les nouvelles contributions scientifiques dans le domaine de la détection de collision ne portent plus uniquement sur de nouveaux algorithmes mais aussi sur la modification d'algorithmes connus pour les adapter à l'architecture. Dans ce sens, nous présentons ici les algorithmes de détection de collision tenant compte de l'architecture machine sur laquelle ils s'exécutent. Nous pouvons les classer en deux familles principales, GPGPU et Multi-threads [1].

4.1 Algorithmes basés GPU

La famille d'algorithmes basés GPU est la plus ancienne. Nous l'avons brièvement présentée en section 2. Les GPUs sont, depuis de nombreuses années, utilisés pour des algorithmes de détection de collision à l'aide de méthode propre aux GPUs. Cependant, ils deviennent de plus en plus utilisés pour effectuer des tâches qui ne leur étaient pas attribuées auparavant. Les algorithmes basés image ont été proposés afin d'exploiter la puissance de calcul grandissante des cartes graphiques. Un avantage important réside en la non-nécessité de précalculer les données avant traitement. Plusieurs algorithmes utilisant l'espace image ont été proposés et comparés à une implémentation CPU [12]. Les résultats montrent que les GPUs accélèrent de façon notoire le temps de calcul de la détection de collision au sein d'environnement complexes. Cependant les CPUs fournissent plus de flexibilité et de meilleures performances dans des environnements légers. La broad phase est également implémentée sur GPU en utilisant les requêtes de visibilité de l'espace image [11]. Cinder [17] est un algorithme exploitant le GPU pour y implémenter une méthode de lancer de rayon pour détecter les collisions. Un algorithmes basé GPU pour détecter les "auto-collisions" pour l'animation de matériaux déformables a été proposé par Govindaraju et al. [10].

4.2 Algorithmes basé multi-threads

Depuis plusieurs années, différents travaux traitent de l'implémentation d'algorithmes multi-threads dans la détection de collision et plus précisément dans les simulations dynamiques de type particule. Lewis et al. [19] proposent un algorithme multi-threads pour simuler des amas de particules en rotation. Une évaluation théorique des performances de la fin du pipeline parallélisé a été réalisée par Zachmann [27] et montre que si la densité de l'environnement est suffisamment importante, alors une utilisation d'algorithmes multi-threads améliorera les performances.

Plus récemment de nouvelles contributions proposent des algorithmes de détection de collision portés sur architecture multi-cœurs. Tang & al. [24] proposent d'utiliser une représentation hiérarchique pour accélérer les requêtes de détection de collision et un algorithme incrémental exploitant la cohérence temporelle. Le tout est porté sur une architecture multi-cœurs. Ils obtiennent un gain de 4X-6X sur un 8-cores pour des modèles d'objets déformables. Kim & al [16] proposent d'utiliser une hiérachie de volume englobant basé sur les caractéristiques pour améliorer les performances de la détection de collision continue. Ils proposent également une nouvelle méthode de décomposition de tâches pour leur algorithme avec une méthode d'assignement dynamique des taches. Ils obtiennent un gain de 7X-8X en utilisant un 8-cores comparé avec un simple cœur. Hermann & al. [13] proposent une parallélisation de simulations physiques interactives, en obtenant un gain de 14X-16X sur un architecture 16-cores.

5 Contributions

Nous présentons deux contributions sur l'optimisation de la détection de collision. La première concerne la parallélisation de l'algorithme du *"Sweep and Prune"* et la seconde décrit une nouvelle façon de représenter le pipeline de détection de collision.

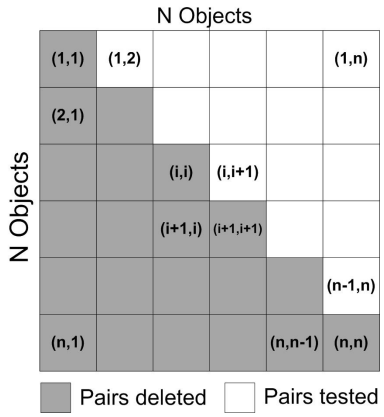

Figure 5: La matrice des paires d'objets à tester et à distribuer entre les cœurs.

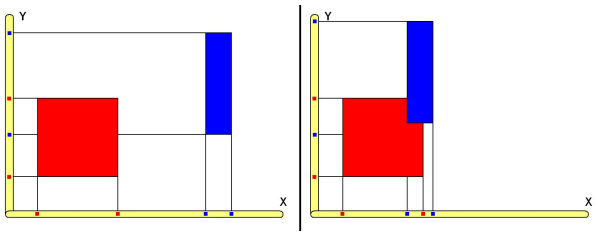

Figure 6: "Sweep and Prune" avec un non-chevauchement à gauche et un chevauchement à droite.

5.1 Parallélisation de Broad Phase

L'état de l'art a montré que la structure des algorithmes de détection de collision nécessite une amélioration pour faire face au problème de temps-réel dans le cas de l'utilisation de larges scènes virtuelles. Nous nous sommes focalisés sur une étape essentielle du pipeline appelée la broad phase en proposant une implémentation de l'algorithme de *"Sweep and Prune"*[6] sur une architecture multi-cœurs.

5.1.1 "Sweep and Prune"

L'algorithme de *Sweep and Prune* est l'un des plus utilisées dans la broad phase car il permet une élimination des paires très efficace et rapide. De plus, il est indépendant de la complexité des objets. La version séquentielle prend en entrée la totalité des objets de l'environnement et fournit en sortie la liste des paires d'objets ayant leurs volumes englobants en chevauchement. L'algorithme est divisé en deux parties, la première est en charge de la mise à jour du volume englobant de chaque objet et la seconde effectue le test de chevauchement de ces derniers. Les volumes englobants utilisés sont des AABBs [5] car ce sont de bons candidats pour la recherche de chevauchement des coordonnées sur les axes.Avec n objets dans l'environnement, $\frac{(n^2-n)}{2}$ paires doivent être testées (cf figure 5). Au vu du test, les paires restantes représentent les objets considérés comme potentiellement en collision. ces paires sont ensuite transmises à la Narrow Phase (section 2.2) pour un test plus précis.

5.1.2 Algorithme Proposé

Les architectures multi-cœurs permettent de séparer les calculs entre les cœurs disponibles. Afin d'en exploiter au maximum les capacités, les sections d'écriture critique, l'attente des threads et la

Algorithm 1 Sweep and Prune algorithm sequential

```
for all (objects) do
    if (current object is active) then
        Update AABB()
    end if
end for
for all (pairs of objects) do
    for all (Axis(x,y,z) do
        if (max Objet A < min Objet B) or
        (max Objet B < min Objet A) then
            Delete pair
        end if
    end for
    Keep pair
end for
```

Figure 7: "Sweep and Prune" séquentiel.

Algorithm 1 Sweep and Prune algorithm parallel

```
C: Number of cores
N: Number of objects
for all (C threads) do
    for (N/C Objects) do
        if (current object is active) then
            Update AABB()
        end if
    end for
end for
Synchronisation of threads()
for all (C threads) do
    for all ((N^2-N)/2 pairs of objects) do
        Calculate Overlap()
    end for
end for
```

Figure 8: "Sweep and Prune" parallèle.

synchronisation des cœurs doivent être prises en compte et minimisées voire évitées. Pour paralléliser cet algorithme nous avons utilisé OpenMP car l'utilisation de directives nous permet de conserver notre code séquentiel moyennant l'ajout de nouvelles structures de données. Malgré le fait qu'Intel TBB offre de meilleures performances, il est plus complexe à mettre en œuvre et il génère du code spécifique impossible à exécuter sans les librairies Intel.

Un schéma simplifié de notre modèle est visible sur la Figure 9 et l'algorithme est décrit sur la Figure 8. Nous y observons une parallélisation des deux étapes principales couplée à une synchronisation entre les deux. Le nombre de threads mis en place dépend du nombre de cœurs disponibles, car étant en charge de calcul géométrique et n'attendant pas de signaux précis, deux threads par cœurs augmenteront le temps de calcul. Dans la première partie de l'algorithme, chaque thread travaille sur $\frac{n}{c}$ objets où n est le nombre d'objets total dans l'environnement et c le nombre de cœurs. Il est possible de répartir les objets entre différents threads car la mise à jour des AABBs ne dépend pas de la complexité de l'objet, le temps passé par objet pour un thread est quasiment homogène. Comparé à la version séquentielle de l'algorithme où les nouveaux AABBs sont écrits à la volée dans la structure de données, nous ne pouvons utiliser le même schéma sans être confrontés à des sections d'écriture critiques entre les threads. L'introduction de plus petits espaces de stockage de données par thread est donc indispensable. Ces espaces sont des tableaux de pointeurs alloués dynamiquement en fonction du nombre de cœurs et d'objets. La synchronisation entre les deux étapes de l'algorithme est donc obligatoire pour regrouper tous les nouveaux AABBs dans la même structure. Nous ne

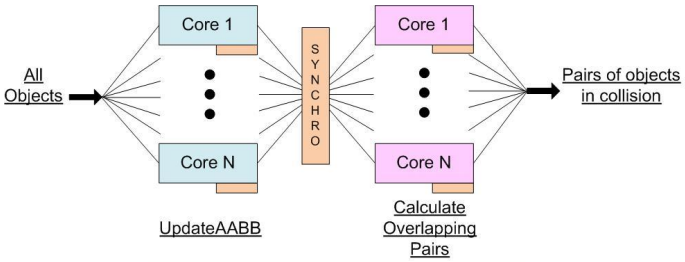

Figure 9: L'algorithme de broad phase parallèle.

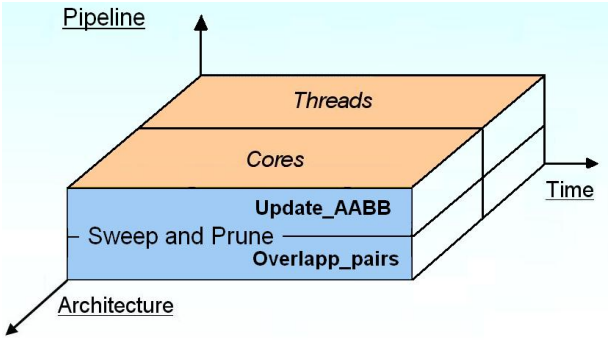

Figure 10: L'algorithme de "Sweep and Prune Algorithm" représenté à l'aide d'un nouveau pipeline.

regroupons uniquement que les pointeurs des tableaux des threads par souci de réduction minimale du temps de synchronisation.

Dans la seconde partie de l'algorithme, chaque thread se voit attribuer $\frac{(n^2-n)}{2}/c$ paires d'objets où c et n sont toujours le nombre de cœurs et d'objets. Le calcul effectué par les threads est un test de chevauchement entre les coordonnées des objets projetées sur les axes. Ce test ne dépendant pas de la complexité des objets et ayant, par conséquent, un temps de calcul quasi-constant, nous pouvons comme dans la première partie répartir les calculs parmi les cœurs disponibles. Afin d'éviter les mêmes problèmes de sections critiques en écriture, nous avons munis chaque thread de son propre espace de stockage pour y mettre les paires d'objets en chevauchement. Toutes les paires d'objets en collision sont ensuite regroupées pour créer la liste des paires objets en collision. Cette nouvelle liste est transmise à la Narrow Phase qui se charge d'effectuer un test exact de collision. Notre algorithme revisité peut être représenté à l'aide d'un nouveau pipeline (cf Figure 10). Nous pouvons y voir les deux principales parties de l'algorithme (AABB update et Overlapping Pairs Tests) mis en œuvre grâce aux multi-threading sur une architecture multi-cœurs.

5.1.3 Pipeline Revisité

Nous avons précédemment montré que les architectures multi-cœurs donnaient une nouvelle dimension à la programmation parallèle pour la réduction du temps de calcul. C'est pourquoi nous proposons d'ajouter cette nouvelle dimension au pipeline séquentiel d'Hubbard [14] afin de réaliser une correspondance exacte de notre algorithme. A travers ce modèle de nouveau pipeline, nous proposons une parallélisation globale du pipeline où chaque phase est représentée comme un bloc dépendant du temps, de sa place dans le pipeline et de l'architecture d'exécution (cf. Figure 11). Nous imaginons également une exécution continue des phases du pipeline, où chacune d'entre elles intègrent ses nou-

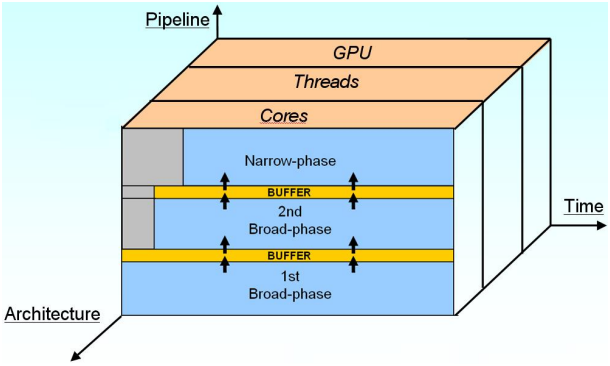

Figure 11: Pipeline de détection de collision revisité.

veaux résultats dans un buffer partagé par les phases voisines. Le principal but de ce pipeline modifié est de pouvoir s'adapter à tout type d'architecture machine. Une adaptation possible de ce pipeline serait d'exécuter la broad phase sur une architecture multi-cœurs à l'aide du multi-threading et d'exécuter la narrow phase sur plusieurs GPU en parallèle. L'adaptation peut également se répandre à l'intérieur même des phases en imaginant une partie d'un algorithme s'exécuter sur un type d'architecture et l'autre partie sur un autre type. Ce pipeline peut également être dimensionné pour une utilisation sur une grappe d'ordinateurs voire une grille de calcul. Le travail sur grille nécessite la prise en compte de plusieurs facteurs supplémentaires tels que le temps de communication entre les diverses entités de calcul et la volatilité des ressources. Nous pouvons tirer un avantage de ces nouveaux facteurs en imaginant des algorithmes prédictifs s'exécutant sur les machines géographiquement éloignées et ceux localement proches, effectuant les calcul exacts de collision.

6 Résultats

Nous présentons, dans cette partie, les principaux résultats obtenus en terme d'accélération du temps de calcul. Afin de faire apparaitre un gain de temps significatif nous travaillons avec plusieurs centaines de milliers d'objets à faible géométrie (cube de 6 polygones). Nous utilisons également un environnement dynamique afin d'assurer le fait que les objets sont continuellement en mouvement et donc considérés comme actifs. Nous avons travaillé sur une machine munie d'un processeur Intel Xeon (2*Quad) CPU X5482 de 3.20 GHz disposant de 64 GB de RAM avec Windows XP Professional x64 Edition Version 2003. Cette machine a servi de base pour les tests sur 1, 2, 4 ou 8 cœurs.

La Figure 12 montre le temps de calcul de la première étape de l'algorithme sur la mise à jour des AABBs. Les quatre différentes courbes représentent le temps de calcul en fonction du nombre de polygones en utilisant 1, 2, 4 ou 8 cœurs. Nous constatons que l'utilisation d'1 ou 2 cœurs réduit ostensiblement le temps de calcul (division par 2). La différence entre 2, 4 et 8 cœurs est moins significative, le temps de calcul y est toujours réduit mais n'y est pas divisé par 2. La seconde partie de l'algorithme est présentée sur la Figure 13 et nous y notons un gain similaire à la première partie. Nos résultats montrent que le gain principal est obtenu au passage de 1 à 2 cœurs, celui de 2 à 4 puis de 4 à 8 réduit également le temps de calcul mais de façon moins notoire. Nous constatons une stabilisation de la courbe ce qui laisse à penser qu'une exécution sur 16-cœurs n'engendrerait pas une grosse différence avec 8 cœurs. Le temps global d'exécution de l'algorithme est présenté sur la Figure 14 où est indiqué le temps d'exécution en fonction du nombre de cœurs. Ce test a été réalisé en utilisant 600k polygones pour 1 à 8 cœurs. Le temps diminue selon que le nombre de cœurs augmente mais la courbe suit une descente de plus en plus faible. Nous pouvons supposer, qu'en utilisant plus de cœurs, la courbe se stabilisera parfaitement quitte à remonter avec un trop grand nombre de cœurs.

Nous avons également mesuré le temps d'exécution mis par t threads réparti sur c cœurs. L'hypothèse fait au début sur le fait qu'utiliser un thread par cœur semblait être la meilleure solution s'est avérée correcte. Par exemple, le temps mis par 3 threads répartis sur 2 cœurs est plus long que d'en utiliser 2 mais meilleur qu'avec 1 seul thread. L'utilisation de plus d'un thread par cœur n'est donc pas justifiée et apparait comme étant moins efficace. La Figure 15 montre le gain de temps en fonction du nombre de cœurs utilisés.

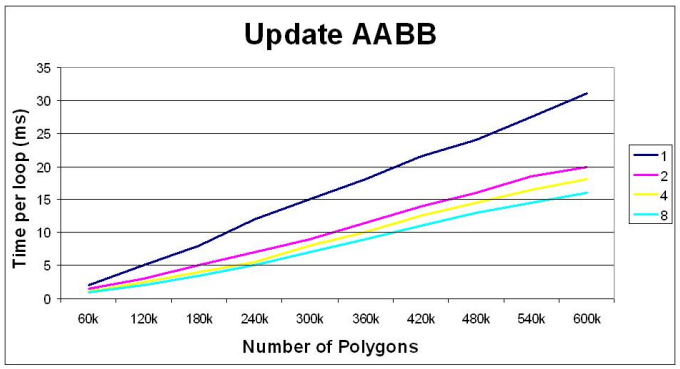

Figure 12: Le temps d'exécution de la mise à jour des AABBs en fonction du nombre de polygones.

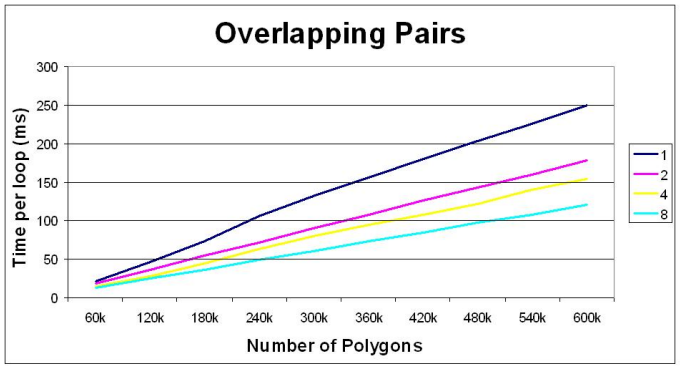

Figure 13: Le temps d'exécution du test de chevauchement des paires en fonction du nombre de polygones.

7 Conclusion

Nous avons présenté deux contributions sur l'optimisation de la détection de collision centrées sur les performances machines. Le modèle de pipeline revisité, présenté avec une nouvelle 3e dimension, permet de regrouper les composants hardware avec le pipeline de détection de collision. En partant de ce modèle, nous nous sommes focalisés sur la première étape du pipeline (broad phase) et nous avons proposé une première manière de parallélisation du célèbre algorithme de *Sweep and Prune*. Ce modèle prend en compte l'architecture d'exécution et plus précisément le nombre de cœurs disponibles. Les architectures nous permettent de répartir les calculs à l'aide du multi-threading. Les sections d'écriture critique et l'attente des threads ont été minimisés en introduisant une nouvelle structure de données pour chaque thread. La programmation

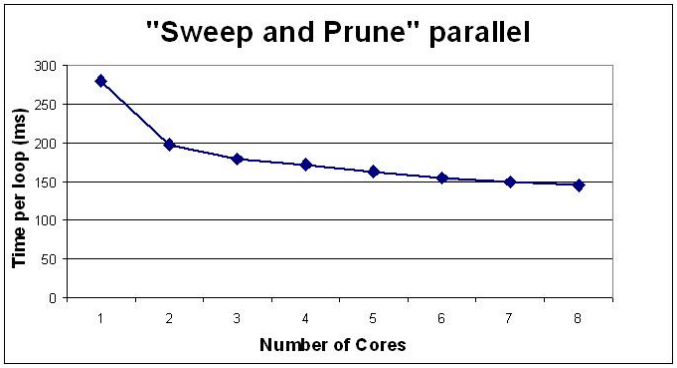

Figure 14: Le temps d'exécution global.

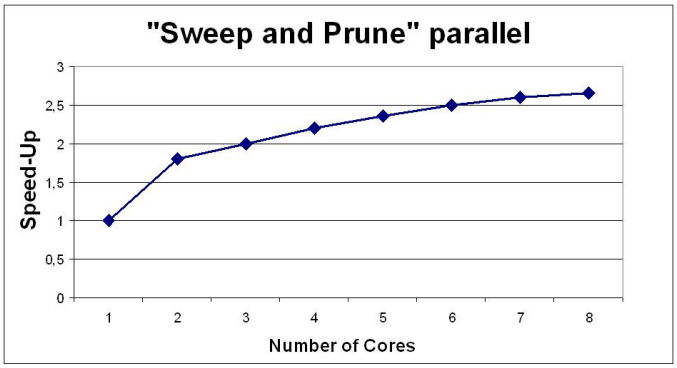

Figure 15: Accélération globale de la parallélisation.

par directives, comme OpenMP, apparait comme un bon compromis pour la portabilité du code. Nous avons également montré que paralléliser l'algorithme sur 2 cœurs est très efficace et réduit significativement le temps de calcul. Même si l'utilisation de 4 ou 8 cœurs engendre aussi une réduction du temps de calcul, celui-ci n'est pas très important. Les futurs algorithmes offrant une interaction temps-réel doivent être implémentés à travers l'utilisation du GPGPU, du multi-threading, des architectures multi ou many-cœurs avec gestion de mémoire principale et mémoire cache. On peut également imaginer travailler sur les infrastructures telles que des clusters ou des grilles de calcul. Le lien existant entre la réalité virtuelle et les performances machines doit être de plus en plus étudié et renforcé.

8 Futurs Travaux

Nous travaillons toujours sur l'optimisation du code du *"Sweep and Prune"* parallèle. Nous nous intéressons aussi au couplage de notre modèle multi-cœurs avec le hardware graphique afin d'accélérer encore l'algorithmes. Tous les blocs de notre pipeline revisité doivent également être optimisés et adaptés. Il faut pour cela définir les caractéristiques techniques et les contraintes temporelles de chaque blocs. Nous réfléchissons pour cela à un modèle générique de correspondance permettant de proposer une configuration d'exécution précise pour l'algorithme de détection de collision quelque soit l'architecture cible.

9 Acknowledgments

Ce travail est financé par la région Bretagne (Projet GRiRV N°4295). Les auteurs souhaitent remercier Florian Nouviale pour sa précieuse aide dans la programmation et le débuggage, Fabrice Lamarche pour ses explications et clarifications sur OpenMP et également Benoit Ronflette qui nous a permis de pouvoir automatiser l'animation au sein de nos environnements.

References

[1] Q. Avril, V. Gouranton, and B. Arnaldi. New trends in collision detection performance. In *Virtual Reality International Conference*, pages 53–62, 2009.

[2] G. Baciu and W. S.-K. Wong. Image-based collision detection for deformable cloth models. *IEEE Trans. Vis. Comput. Graph*, 10(6):649–663, 2004.

[3] S. Bandi and D. 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.

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

[5] G. V. D. Bergen. Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, 2(4):1–13, 1997.

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

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

[8] E. G. Gilbert, D. W. Johnson, and S. 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.

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

[10] N. K. Govindaraju, M. C. Lin, and D. Manocha. Quick-cullide: fast inter- and intra-object collision culling using graphics hardware. page 218, 2005.

[11] N. K. Govindaraju, S. Redon, M. C. Lin, and D. 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.

[12] B. Heidelberger, M. Teschner, and M. H. Gross. Detection of collisions and self-collisions using image-space techniques. In *WSCG*, pages 145–152, 2004.

[13] E. Hermann, B. Raffin, and F. Faure. Interactive physical simulation on multicore architectures. In *Eurographics Workshop on Parallel and Graphics and Visualization, EGPGV'09, March, 2009*, Munich, Allemagne, 2009.

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

[15] P. Jiménez, F. Thomas, and C. Torras. 3d collision detection: a survey. *Computers & Graphics*, 25(2):269–285, 2001.

[16] D. Kim, J.-P. Heo, and S. eui Yoon. Pccd: Parallel continuous collision detection. Technical report, Dept. of CS, KAIST, 2008.

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

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

[19] M. Lewis and B. L. Massingill. Multithreaded collision detection in java. In H. 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.

[20] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. Technical report, Mar. 19 1991.

[21] M. C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In R. 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, Sept. 1998. Information Geometers.

[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. 2007. Warning: the year was guessed out of the URL.

[23] J. Reinders. *Intel Threading Building Blocks*. O'REILLY, 2007.

[24] M. Tang, D. Manocha, and R. Tong. Multi-core collision detection between deformable models. In *Computers & Graphics*, 2008.

[25] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. pages 119–140, Sept. 2004.

[26] G. Vaněček, Jr. Back-face culling applied to collision detection of polyhedra. *The Journal of Visualization and Computer Animation*, 5(1), Jan.–Mar. 1994.

[27] G. Zachmann. Optimizing the collision detection pipeline. In *Proc. of the First International Game Technology Conference (GTEC)*, Jan. 2001.