THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Université Toulouse III - Paul Sabatier*

Discipline ou spécialité : *Informatique*

Présentée et soutenue par Jean-Patrick Roccia

Le 24 Mai 2013

Accélération de l'algorithme du lancer de rayons en environnement parallèle hétérogène

---

JURY

Rapporteurs : Xavier Granier
Dominique Houzet

Examinateurs : Stéphane Mérillou
Bernard Lecussan

Ecole doctorale : Mathématiques Informatique Télécommunications de Toulouse
Unité de recherche : Institut de Recherche en Informatique de Toulouse - UMR 5505
Directeur(s) de Thèse : Mathias PAULIN, Christophe COUSTET

Remerciements

Mes pensées vont à mes proches qui m'ont encouragé et soutenu tout au long de cette thèse. Je tiens à remercier les équipes d'HPC-SA et de VORTEX pour leur accueil chaleureux. Et plus particulièrement mon responsable scientifique côté entreprise, Christophe Coustet ainsi que mon directeur de thèse côté universitaire, Mathias Paulin.

Je dédie cette thèse à ma compagne et ma famille.

Table des matières

1

Introduction et motivations

HPC-SA fournit depuis plus de dix ans des outils de simulation hautes performances dans le domaine du lancer de rayons. Cette régularité, ainsi que le besoin de maintenir ces performances face à la concurrence, impliquent une participation active à la recherche dans le domaine du lancer de rayons. Cette thèse s'inscrit dans cette dynamique.
Avant d'aller plus avant, il est indispensable de présenter l'algorithme du lancer de rayons lui-même. Nous détaillerons, par la suite, les environnements d'exécutions ciblés par nos outils de lancer de rayons. Nous présenterons ensuite nos contraintes purement industrielles, avant de terminer cette introduction par une présentation de l'organisation générale de la thèse.

1.1 Lancer de rayons

En préambule à l'étude de son accélération, définissons l'algorithme du lancer de rayons ainsi que ses applications.
Comme le souligne Macey dans [Mac08], les fondements du lancer de rayons ont été introduits par René Descartes en 1637. Descartes utilisa en effet des rayons de lumières pour expliquer la formation d'arcs-en-ciel. Dans le domaine de l'informatique graphique, la première utilisation du lancer de rayon remonte à 1968 avec Arthur Appel. Depuis cette date, le lancer de rayons est un domaine de recherche important au sein de la communauté informatique. En effet, il permet d'obtenir une grande précision pour simuler des phénomènes de rayonnements physiques, et ce dans des domaines aussi variés que la thermique, l'optique ou encore l'acoustique.
Concrètement, le lancer de rayons peut être découpé en deux parties distinctes : la requête d'intersection et la physique associée. Ces deux parties permettent de définir le contour des fonctionnalités de notre bibliothèque de lancer de rayon. Le but étant d'une part de proposer une bibliothèque indépendante de la physique qu'elle permet de simuler,

et d'autre part de pouvoir travailler les problématiques physiques indépendamment du moteur de lancer de rayons. Détaillons brièvement chacune de ces parties.

1.1.1 Requête d'intersection

La requête d'intersection est purement géométrique. Elle constitue la partie centrale et commune à toutes les applications utilisant le lancer de rayons. La partie physique du lancer de rayon est cliente de cette requête. Celle-ci permet en effet d'évaluer les relations de visibilités entre objets dans un espace donné. Cette partie est typiquement majoritaire en terme de coût d'exécution. Elle est donc critique, et nécessite une attention particulière, dès lors que l'on cherche à accélérer l'exécution du lancer de rayons. Le coût de cette requête implique bien souvent des concessions au niveau de la partie physique afin de minimiser le nombre de requêtes d'intersections.

1.1.2 Physique

La partie physique du lancer de rayons s'appuie sur les modèles de simulations associés au domaine dans lequel il est utilisé. Cette partie est donc vraiment spécifique aux phénomènes simulés et peut totalement être indépendante du service lui assurant l'accès aux relations d'inter-visibilité. Concrètement, la physique permet de quantifier l'impact de l'émission de rayonnements dans une scène, et de propager ce rayonnement via des requêtes d'intersections géométriques établissant les relations de visibilité.

1.2 Architectures matérielles modernes

Suite au coup d'arrêt de la course aux fréquences de fonctionnement des CPU, la nouvelle source de gain de performance est désormais le nombre croissant de cœurs de calcul. Contrairement à l'augmentation de fréquence de fonctionnement qui permettait d'accélérer les exécutions sans aucune modification des codes, le parallélisme implique une refonte profonde des codes existants pour accélérer leur exécution. Ce parallélisme croissant au niveau des CPU est toutefois très éloigné des standards sur GPU. Ces derniers sont en effet récemment devenus des coprocesseurs généraux, qui posent eux aussi des problèmes en terme de parallélisation mais à une toute autre échelle. Détaillons les spécificités matérielles générales des CPU et des GPU.

1.2.1 Architecture parallèle CPU

Le nombre d'unités de calcul sur CPU, malgré sa constante augmentation, reste relativement faible. En effet, les processeurs grand public disposent généralement de quatre cœurs physiques au maximum. Leur nombre est artificiellement augmenté sur certains modèles en simulant la présence d'un cœur logique supplémentaire par cœur physique. Ces cœurs supplémentaires, bien qu'apportant un gain de performance en général, ont un impact bien moindre qu'un réel cœur de calcul physique.
La propriété des CPU est donc de disposer de peu de cœurs de calcul, mais ce faible nombre de cœurs est compensé par leur caractère versatile et autonome. Au niveau de l'accès aux données, les CPU ont un accès rapide et privilégié à la mémoire centrale. De plus, sur les dernières architectures, les unités de calcul disposent de deux niveaux de cache de données chacune, auxquels s'ajoute un troisième niveau de cache partagé par l'ensemble des cœurs. Une grande partie des transistors d'un CPU est dédiée à maximiser l'utilisation du pipeline d'instructions de chacun des cœurs de calcul. Pour cela des mécanismes de logiques et de contrôle sont mis en jeux, afin de prévoir des entrées dans le pipeline d'instruction de manière continue et efficace. En effet, une mauvaise prédiction des instructions à réaliser implique un vidage, puis rechargement du pipeline d'instructions ayant un fort impact sur les performances. Pour résumer, le CPU se concentre sur la maximisation de l'efficacité d'un nombre restreint de cœurs de calcul sur des flots de données importants. Ce qui justifie l'importance des parties logique et contrôle, ainsi que la taille et le nombre des caches mémoire, visibles sur la figure 1.2.
Cette capacité à s'adapter aux flux d'instructions, et la nature versatile des cœurs CPU liée à la maturité de leur écosystème en font des unités relativement simples à programmer.
Il est à noter que bien avant la disponibilité de CPU multi-cœurs, une forme de parallélisme avait déjà fait son apparition sur CPU sous la forme d'instructions prenant plusieurs données en entrée et leur appliquant le même traitement en parallèle. Ces instructions, de type SIMD ("Single Instruction Multiple Data"), ont perduré et évolué au travers de diverses extensions, avec parmi les plus marquantes et utilisées SSE, SSE2, AVX et bientôt AVX2. Chacun des cœurs des processeurs multi-cœurs est donc capable d'utiliser ce type d'instructions, ce qui implique un niveau de parallélisme supplémentaire à ne pas négliger.

1.2.2 Architecture parallèle GPU

La position des GPU est radicalement opposée à celle des CPU. Originellement, les GPU étaient des coprocesseurs graphiques au comportement entièrement câblé. Cette ri-

gidité dans le pipeline d'exécution leur permettait d'outrepasser les CPU pour des tâches de rendus massivement parallélisables en faisant l'impasse sur les unités de contrôle d'exécution. Leur architecture était donc composée d'une multitude d'unités de calcul, bien plus simples qu'un cœur CPU. Avec l'avènement du pipeline de rendu programmable à différents niveaux via les shaders, l'idée a germé d'utiliser les GPU pour des tâches plus générales. C'est ici qu'est né le concept de GPGPU, pour "General-Purpose computing on Graphics Processing Units".
Les shaders ouvraient la programmation des GPU sans pour autant briser totalement la notion de pipeline de rendu. Depuis 2006, les GPU ont évolué vers une architecture unifiée et donc n'exécutant plus un pipeline figé. Cette évolution matérielle a permis l'ouverture des GPU aux calcul généraux, mais il a fallu définir des interfaces de programmation pour pouvoir contrôler l'exécution d'un programme dans un environnement si massivement parallèle. CUDA [NVIb] puis, plus tard, OpenCL [GROa] ont permis d'obtenir cette capacité.
Le parallélisme sur un GPU est sans commune mesure avec celui d'un CPU. Les derniers GPU NVIDIA GeForce GTX 680 disposent en effet de 1536 cœurs de calcul, et une grande partie des transistors sont destinées aux unités de calcul elle-mêmes comme le présente la figure 1.2. Le modèle de programmation des GPU est de type SIMT ("Single Instruction Multiple Thread"), dérivé dans le principe des modèles SIMD. En effet, les cœurs de calcul sont organisés par blocs de 32 cœurs. Chacun de ces blocs est donc capable de gérer plusieurs groupes de 32 fils d'exécution. Les 32 fils d'exécution groupés sont totalement synchronisés, et le fait de pouvoir en attribuer plusieurs à un bloc permet de masquer les latences mémoire en intercalant les exécutions des différents groupes de 32 fils d'exécution au sein du bloc. En pratique, les 32 fils d'exécution ne divergent pas au niveau de l'exécution, et certains sont simplement masqués pour les portions de codes qu'ils ne doivent pas exécuter. Il est donc rentable en terme de performances de maintenir une cohérence d'exécution à l'échelle de ses 32 fils d'exécution afin d'éviter des masquages successifs résultants à un cumul des couts d'exécution de chacun des codes divergents.
Pour ce qui est de l'accès à la mémoire, chaque bloc a à sa disposition un certain nombre de registres mémoire et une zone mémoire très rapide locale au bloc. Les accès mémoire à la mémoire globale du GPU sont eux beaucoup plus coûteux, et doivent être organisés de manière coalescente dans la mesure du possible pour préserver les performances. Les accès mémoire sont réellement le point crucial de la programmation GPU. Inutile de compter sur les caches ou l'intelligence des unités de calcul pour masquer les accès mémoire. Augmenter le nombre de fils d'exécution gérés par un bloc permet de recouvrir en partie les latences mémoire mais impose de disposer d'assez de mémoire interne au bloc pour accueillir des fils d'exécution supplémentaires.

Une autre limitation du GPU concerne la taille de sa mémoire globale, généralement bien inférieure à celle disponible en mémoire centrale. Le GPU est de plus soumis aux limitations en termes de bande passante du bus PCI-EXPRESS pour toute communication avec la mémoire centrale.
Malgré sa puissance théorique considérable illustrée par le graphe 1.1, tirer partie efficacement d'un GPU pour du calcul général se révèle donc vite compliqué et requiert une très bonne compréhension du fonctionnement matériel pour l'obtention de performances correctes.

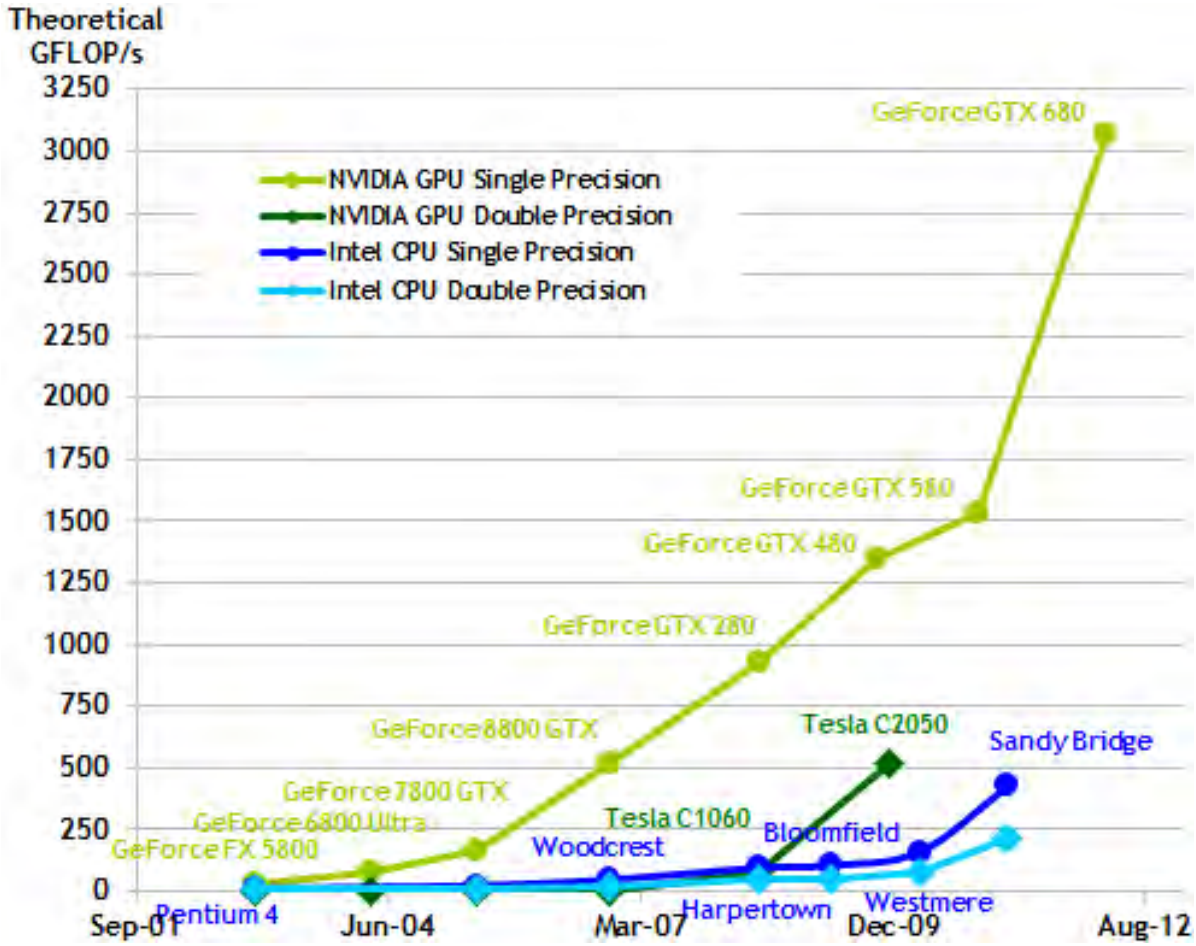


FIGURE 1.1 – Évolution des performances théoriques des CPU et des GPU, image issue de la documentation CUDA [NVIb].

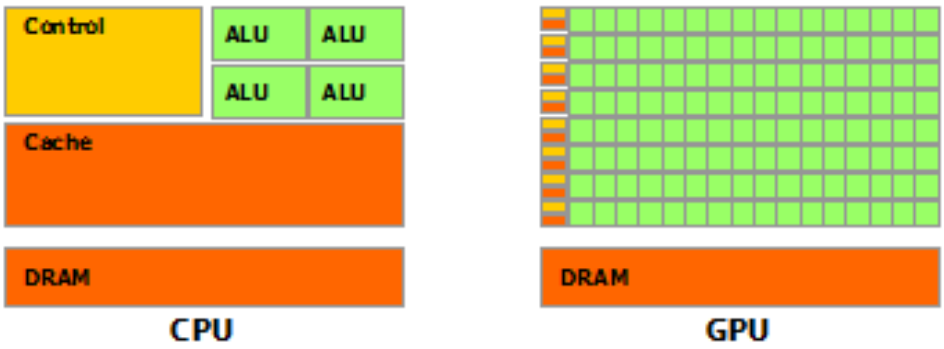


FIGURE 1.2 – Vue simplifiée d'un CPU et d'un GPU, image issue de la documentation CUDA [NVIb].

1.3 Contraintes industrielles

Cette thèse étant réalisée en étroite collaboration avec le milieu industriel, un certain nombre de contraintes ont guidé nos différentes expérimentations. En sus de la contrainte primordiale de performances générales, voici un aperçu des principales lignes directrices qui ont motivé nos choix tout au long de la thèse.

1.3.1 Adaptabilité à l'existant

Notre bibliothèque de lancer de rayons doit permettre une intégration rapide dans des logiciels existants. Cela permet d'une part de remplacer les versions précédentes de notre bibliothèque, mais aussi de convaincre des clients potentiels. En effet, il est très compliqué de convaincre un client potentiel de tester la bibliothèque si son intégration nécessite plus que quelques heures de développement. De plus, il est indispensable de pouvoir intégrer notre bibliothèque sans avoir à repenser la physique associé au lancer de rayons, qui a potentiellement était validée depuis des années.

1.3.2 Adaptabilité aux ressources

Il est indispensable que notre bibliothèque de lancer de rayons ne soit pas trop restrictive au niveau des configurations matérielles et logicielles prises en compte afin de pouvoir assurer son intégration dans nos produits, mais aussi pour d'éventuelles intégration dans des logiciels tiers. C'est pourquoi notre bibliothèque doit être capable de s'adapter dynamiquement au matériel disponible afin d'en tirer au mieux parti sans augmenter la configuration minimale. Notre configuration matérielle minimale cible n'impose qu'un CPU de type x86 récent (SSE2). Pour ce qui est de la partie logicielle, nous visons d'être

compatible avec les trois principaux systèmes d'exploitation du marché, à savoir Microsoft Windows, apple MAC-OS X et Linux.

1.3.3 Adaptabilité aux requêtes utilisateur

Nous aspirons à ce que notre interface de parallélisation soit utilisable sans connaissance à priori des problématiques internes au lancer de rayons. C'est pourquoi notre bibliothèque de lancer de rayons doit être capable de s'adapter aux requêtes de l'utilisateur sans présupposer des propriétés difficiles à appréhender pour l'utilisateur. L'interface avec l'utilisateur se veut simple et minimale pour une utilisation basique. Tous les cas problématiques, ainsi que les aspects parallélisation et répartition sur des unités de calculs hétérogènes doivent être détectés et gérés en interne.

1.3.4 Maintenance

Dernière contrainte et non des moindres, l'interface de notre bibliothèque de lancer de rayons doit être la plus claire possible afin de ne pas perdre les utilisateurs externes. De plus, il est indispensable de prendre en compte l'impact de l'intégration de notre bibliothèque sur la maintenance du code client. Il est en effet nécessaire de pouvoir debugger ou réviser le code client sans que notre bibliothèque soit une gène.

1.4 Organisation de la thèse

Afin de clôturer cette introduction, voici un aperçu général de l'organisation de la thèse. Dans une première partie consacrée à la requête d'intersection, nous ferons un état de l'art des méthodes permettant de l'accélérer avant de présenter nos travaux portant sur trois axes :

- Construction hybride CPU/GPU d'un KD-Tree optimal.
- Traversée hybride CPU/GPU d'un arbre de subdivision binaire.
- Test d'intersection rayon/triangle hybride CPU/GPU.

La seconde partie de la thèse cible la partie physique du lancer de rayons, et présente notre contribution sous forme d'interface de parallélisation générique permettant de répartir les calculs entre les CPU et les GPU sans intervention de l'utilisateur.
Nous conclurons en faisant un état des lieux sur la tenue des objectifs et le respect des contraintes, tout en ouvrant la discussion sur les travaux futurs.

Première partie

Vers une parallélisation hybride CPU/GPU du lancer de rayons

1

Introduction

Le principe de la requête d'intersection 1.1.1 s'énonce de manière très simple. Il suffit, en effet, d'être capable de répondre à la question suivante : étant donné un rayon, ayant une position de départ ainsi qu'une direction donnée dans l'espace, intersecte-t-il un objet dans l'environnement, et si oui quelle primitive géométrique et à quelle position ? La capacité à répondre à cette question suffit à être capable de simuler fidèlement des phénomènes de rayonnement plus ou moins complexes. Toutefois, malgré sa relative simplicité, cet algorithme pose de multiples problèmes lorsqu'il s'agit de le rendre performant. Avec une approche naïve, la requête d'intersection entre un rayon et une primitive géométrique représente environ 90% du temps de calcul dans l'algorithme du lancer de rayons. La primitive géométrique actuellement massivement adoptée pour la représentation de la géométrie est le triangle, pour des soucis de performances et de simplicité. C'est pourquoi nous utiliserons cette primitive pour illustrer nos travaux, même s'il est envisageable d'adapter nos recherches à d'autres primitives.
En l'état, une simulation, même très peu complexe, peut prendre plusieurs heures pour générer un résultat de faible qualité. La recherche en ce domaine est parvenue à mettre en place divers mécanismes qui permettent d'accélérer significativement la requête d'intersection géométrique, parmi lesquels nous retiendrons particulièrement :

- l'optimisation de la méthode de calcul d'intersection rayon/triangle elle-même.
- la maximisation de l'apport d'information par rayon.
- la construction d'une structure spatiale d'accélération.
- la parallélisation de la requête d'intersection.

2

Intersection rayon/triangle

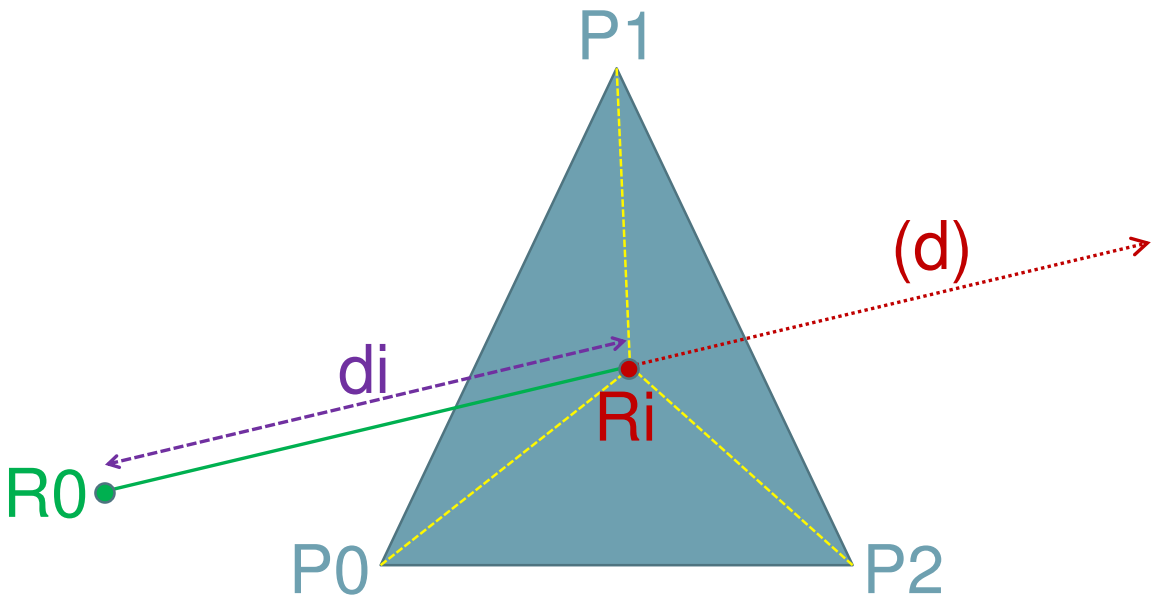


FIGURE 2.1 – Intersection Ri à une distance di entre un rayon d'origine $R0$ et de direction (d) et un triangle $P0P1P2$.

Étant donné la prépondérance de cette partie du code lors de l'exécution, son optimisation a fait l'objet de nombreuses recherches. Comme illustré par la figure 2.1, la

méthode d'intersection rayons/triangle doit être capable, à partir d'un rayon défini par son origine $R0$ et sa direction (d) et d'un triangle $P0P1P2$, de déterminer non seulement si le rayon intersecte le triangle, mais aussi de fournir des propriétés sur l'impact. En effet, il est très courant d'utiliser la distance rayon-impact di, la position de l'impact Ri et les coordonnées barycentriques de cet impact au sein du triangle lors du post-traitement des impacts. Les coordonnées barycentriques d'un impact, couramment nommées u, v et w, permettent la localisation de l'impact à l'intérieur du triangle grâce à ses trois sommets, $P0$, $P1$ et $P2$, via l'équation 2.1. Celles-ci servent par exemple à calculer les coordonnées de texture pour un impact donné, ou encore à obtenir la normale, ou la couleur au sommet interpolée à l'intérieur d'un triangle.

$$Ri = w * P0 + u * P1 + v * P2 \tag{2.1}$$

2.1 Méthodes d'intersection

Trois méthodes différentes se sont plus ou moins imposées, de par leur complémentarité. Voici un bref aperçu des diverses solutions. Toutes ces techniques d'intersection essaient de rejeter au plus vite les impacts invalides afin de minimiser les calculs. Ce rejet s'opère selon deux critères sur l'intersection entre le plan porteur du triangle et le rayon :

- l'intersection n'est pas incluse dans les limites spatiales du triangle, c'est-à-dire que les coordonnées barycentriques u, v et w ne sont pas comprises entre zero et un.

- la distance entre l'intersection et le rayon n'est pas située dans l'intervalle d'intersections valide.

2.1.1 Test d'intersection Möller-Trumbore

Ce test, proposé par Möller et Trumbore dans [MT97], repose sur la capacité à se placer dans un repère propice à l'évaluation rapide de l'intersection. Pour cela, la position du rayon est tout d'abord translaté afin de se placer dans un repère dont l'origine n'est autre que le premier sommet du triangle à intersecter. Une seconde opération géométrique est ensuite appliquée afin de se ramener dans un repère où le triangle est unitaire sur les axes Y et Z, et la direction du rayon alignée à l'axe X. La grande force de cette méthode d'intersection réside dans son efficacité sans le moindre pré-calcul. Il est toutefois encore possible de l'optimiser de deux manières. L'une consiste à simplement utiliser une certaine représentation mémoire des triangles : le premier sommet et les deux arêtes qui le relient aux deux autres sommets. Cela permet d'éviter de calculer ces deux arêtes nécessaires

à chaque test d'intersection. De plus, il est possible de pré-calculer le produit vectoriel entre ces deux arêtes, lui-aussi nécessaire au calcul d'intersection, afin d'accélérer encore le test. Au final si l'on utilise la représentation optimale des triangles pour cette méthode d'intersection, la structure de donnée est composée de 9 réels par triangles ou 12 réels par triangle si l'on pré-calcul le produit vectoriel entre les deux arêtes.

2.1.2 Test d'intersection Wald

L'idée à la base de ce test d'intersection, issu des travaux de Wald dans [Wal04], consiste à se ramener à une évaluation en deux dimensions des coordonnées barycentriques de l'intersection entre le rayon et le triangle. Afin d'atteindre cet objectif, chacun des triangles va être projeté sur un des trois plans alignés aux axes. Le choix du plan de projection est crucial pour la robustesse des calculs, il est nécessaire de choisir le plan dans lequel la surface du triangle projeté est la plus importante. Cette méthode se base sur le fait que projeter un triangle et une intersection sur un même plan conserve les coordonnées barycentriques. Il est donc aisé de calculer ses coordonnées barycentriques sur le plan de projection, et donc en deux dimensions, puis d'utiliser ces coordonnées pour en déduire la véritable intersection au sein du triangle. Cette méthode repose sur une phase de pré-calcul, stockant neuf réels et un entier par triangles, indispensable pour obtenir des performances. Concrètement, le test d'intersection consiste à intersecter le rayon avec le plan porteur du triangle, puis à projeter l'intersection sur le plan aligné choisi lors du pré-calcul pour enfin évaluer à partir des coordonnées barycentrique u, v et w obtenues si oui ou non le rayon intersecte le triangle.

2.1.3 Test d'intersection Shevtsov

Cette version du test d'intersection, proposée par Shevtsov dans [SSKN07], utilise les coordonnées de Plücker décrites dans [Sho98]. Celles-ci sont une manière alternative de décrire des droites dirigées dans l'espace, en les définissant avec six nombres. Elles permettent de déterminer, en seulement un produit scalaire à six dimensions, d'une part si les lignes s'intersectent(produit scalaire nul), mais aussi leurs relations spatiales, c'est-à-dire laquelle des deux droites passe au-dessus de l'autre grâce au signe du produit scalaire. Cette méthode d'intersection repose donc sur les relations entre le rayon et chacune des arêtes du triangle à intersecter. Si les trois produits scalaires, entre chacune des arêtes et la direction du rayon ont le même signe, alors le triangle est intersecté. Cette méthode repose elle aussi sur une phase de pré-calcul, stockant neuf réels et un entier par triangles, afin d'obtenir de meilleures performances.

2.2 Choisir une méthode d'intersection

Le choix de la méthode d'intersection dépend directement des contraintes imposées par l'environnement d'exécution. Si la compacité mémoire est critique, mieux vaut s'orienter vers une méthode ne nécessitant pas de pré-calcul pour être efficace. Dans le cas où la performance est le critère privilégié, il est préférable d'utiliser une version du test d'intersection accélérée par le pré-calcul de données. Selon le type de données en entrée, la méthode d'intersection la plus efficace peut varier. En effet, selon la scène et les contraintes de rejet d'intersection fixées par l'intervalle de validité des intersections, un rejet précoce par distance d'intersection peut s'avérer plus ou moins efficace qu'un rejet précoce par inclusion de l'impact au sein du triangle. Notre choix sera détaillé plus tard suite à diverses expérimentations.

3

Maximiser l'apport des rayons

Cette méthode d'accélération du lancer de rayons se base simplement sur une pré-étude de la quantité d'information qu'un rayon va pouvoir nous apporter. En effet, il est possible dans certains cas de maximiser la quantité d'information que va pouvoir nous apporter un rayon dans notre simulation. Afin d'évaluer le potentiel d'un rayon, plusieurs approches sont possibles. Certaines sont issues de pré-études faites en amont des simulations, d'autres repensent l'algorithme de génération des rayons, ou sont capables de réaliser un apprentissage durant la simulation.

La grande force du lancer de rayon en terme de synthèse d'image, se situe dans sa capacité à modéliser les effets globaux à la scène. En effet, en terme de rayons, il est très facile de modéliser les ombres, réflexions, réfractions et chemins lumineux. Dans la pratique, il est toutefois indispensable de bien penser l'algorithme de rendu sous peine d'obtenir de mauvaises performances.

Si l'on veut procéder par analogie avec la physique du photon, les sources lumineuses sont les origines des rayons lumineux qui vont venir apporter de l'énergie dans la scène à chaque intersection. Si l'on applique ce schéma tel quel pour générer une image, le temps nécessaire à obtenir un résultat stable et utilisable est potentiellement énorme. En effet, rien n'assure que chacun des rayons que va générer la lumière joue un rôle dans notre image finale, et une grande partie des calculs sont donc totalement inutiles puisque l'énergie émise par la lumière n'atteint pas la caméra dans notre scène. Afin de remédier à ces calculs superflus Whitted a proposé un algorithme de lancer de rayons maximisant l'apport d'informations par rayon dans [Whi80]. Pour cela, au lieu de chercher à générer les rayons de lumière qui vont atteindre la caméra, Whitted utilise la réciprocité des chemins optiques afin que la caméra devienne le point de départ des rayons. Le problème est donc inversé, et les rayons ne sont plus porteur de lumière mais à la recherche de

lumière. Seuls les rayons ayant la camera comme point de départ sont tracés et ont la possibilité de générer des rayons dits secondaires (réflexion, réfraction et ombre). Grâce à cet algorithme, les données d'illumination et les relations de visibilité avec les différentes sources lumineuses ne sont étudiées que pour des objets ayant un rapport avec des objets visibles par la caméra, et qui apportent donc une réelle contribution à l'image. Cette modification algorithmique permet de diminuer drastiquement le temps de calcul. Si la réciprocité des chemins n'introduit pas de biais dans les calculs, l'algorithme complet de Whitted ne traite que la composante spéculaire des matériaux. Il ne permet donc pas, tel quel, de prendre en compte une illumination globale réaliste incluant les réflexions diffuses et autres caustiques.

Les algorithmes d'illumination globale ont pour objectif de résoudre l'équation du rendu présentée par Kajiya dans [Kaj86] :

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_s(x, \omega_i \leftrightarrow \omega_o) L_i(x, \omega_i) |(N_x.\omega_i)| \mathrm{d}(\omega_i) \tag{3.1}$$

avec $L_o(x, \omega_o)$ la luminance sortante au point x selon la direction ω_o, $L_e(x, \omega_o)$ le rayonnement émis (source lumineuse), $L_i(x, \omega_i)$ la luminance entrante, $f_s(x, \omega_i \leftrightarrow \omega_o)$ la fonction de distribution bidirectionnelle de diffusion (BSDF) et N_x la normale au point x. Cette equation ainsi que le comportement d'une BSDF sont respectivement illustrés par les images 3.1 et 3.2.

Cette équation n'ayant pas dans le cas général de solution analytique, il est courant de recourir à des méthodes d'intégrations numériques et en particulier à la méthode Monte-Carlo publiée par Metropolis dans [MU49]. En effet, cela permet via l'introduction de tirages aléatoires de converger vers une estimation de la luminance entrante $L_i(x, \omega_i)$ issue globalement de la scène. Malheureusement, ce type de simulation a pour point faible sa vitesse de convergence en $\frac{1}{\sqrt{n}}$, ce qui signifie concrètement qu'améliorer la qualité du résultat d'un facteur quatre implique une augmentation du temps de calcul d'un facteur seize.
C'est pourquoi une approche très en vogue dans le domaine du rendu par méthode Monte-Carlo consiste à faire de l'échantillonnage préférentiel (importance sampling). Cela consiste en une meilleure sélection aléatoire des échantillons afin d'améliorer la vitesse de convergence vers la solution correcte. Différentes heuristiques ont en effet été développées afin de guider la génération aléatoire de chemin de lumière. Parmi ces techniques, Lawrence propose d'utiliser la BRDF pour influencer les chemins dans [LRR04], Agarwal suggère l'utilisation de cartes d'environnement dans [ARBJ03] et Jensen illustre l'intérêt de l'utilisation de cartes de photons dans [Jen95]. Chacune de ces méthodes donne des

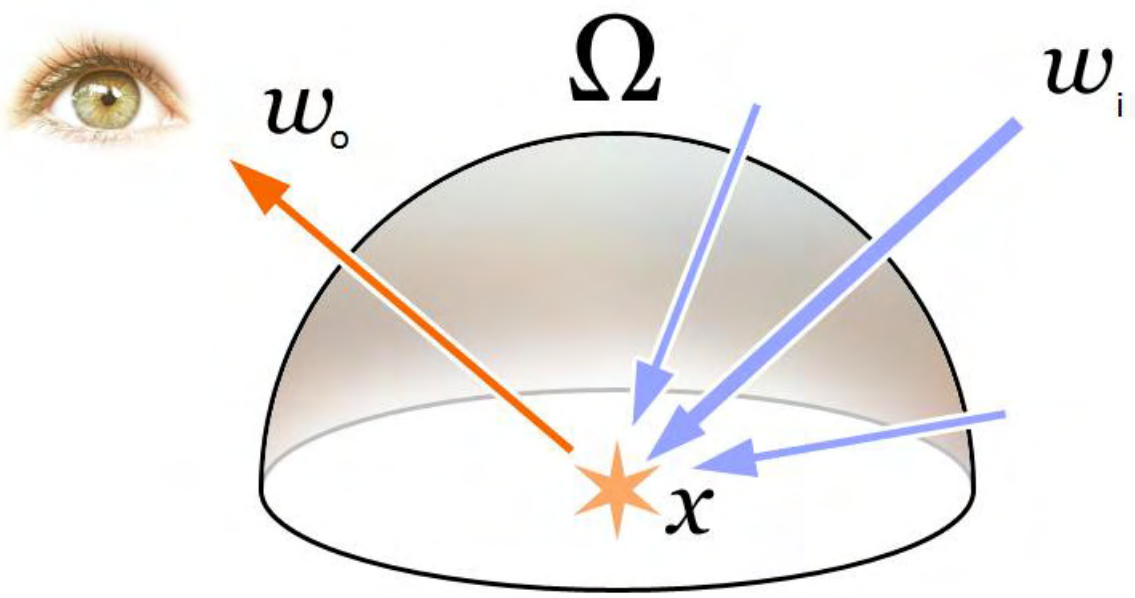


FIGURE 3.1 – Illustration de la luminance sortante en ω_o en direction du point d'observation et de la luminance entrante ω_i au sommet x.

résultats concluants dans certaines conditions mais impliquent des pertes de performances en terme de convergence dans d'autres configurations. C'est pourquoi l'idée de proposer un échantillonnage préférentiel multiple a émergée avec, par exemple, les travaux de Pajot dans [PBPP11]. Cette technique consiste à choisir automatiquement à la volée une méthode d'échantillonnage en fonction d'une estimation de sa capacité à répondre au mieux aux propriétés du matériaux. La mise en place de tels mécanismes amène une accélération non-négligeable en terme de vitesse de convergence.
L'algorithme de rendu de Whitted et l'échantillonnage préférentiel, même si leur contributions sont très éloignées, partagent un objectif commun : maximiser la quantité d'informations par rayons lors de la simulation afin d'accélérer les simulations. Les gains de performances apportés par ces différentes contributions rappellent combien il est important de garder à l'esprit le coût de lancement des rayons. Malgré toutes les optimisations effectuées au niveau de la requête géométrique elle-même, les démarches visant à réduire le nombre de rayons nécessaires à l'obtention d'un résultat donné peuvent tout de même s'avérer payante. Leur mise en place est toutefois bien plus délicate car totalement liée aux phénomènes simulés, et ne relève donc pas de la bibliothèque de lancer de rayons qui se veut générale et donc plus bas niveau.

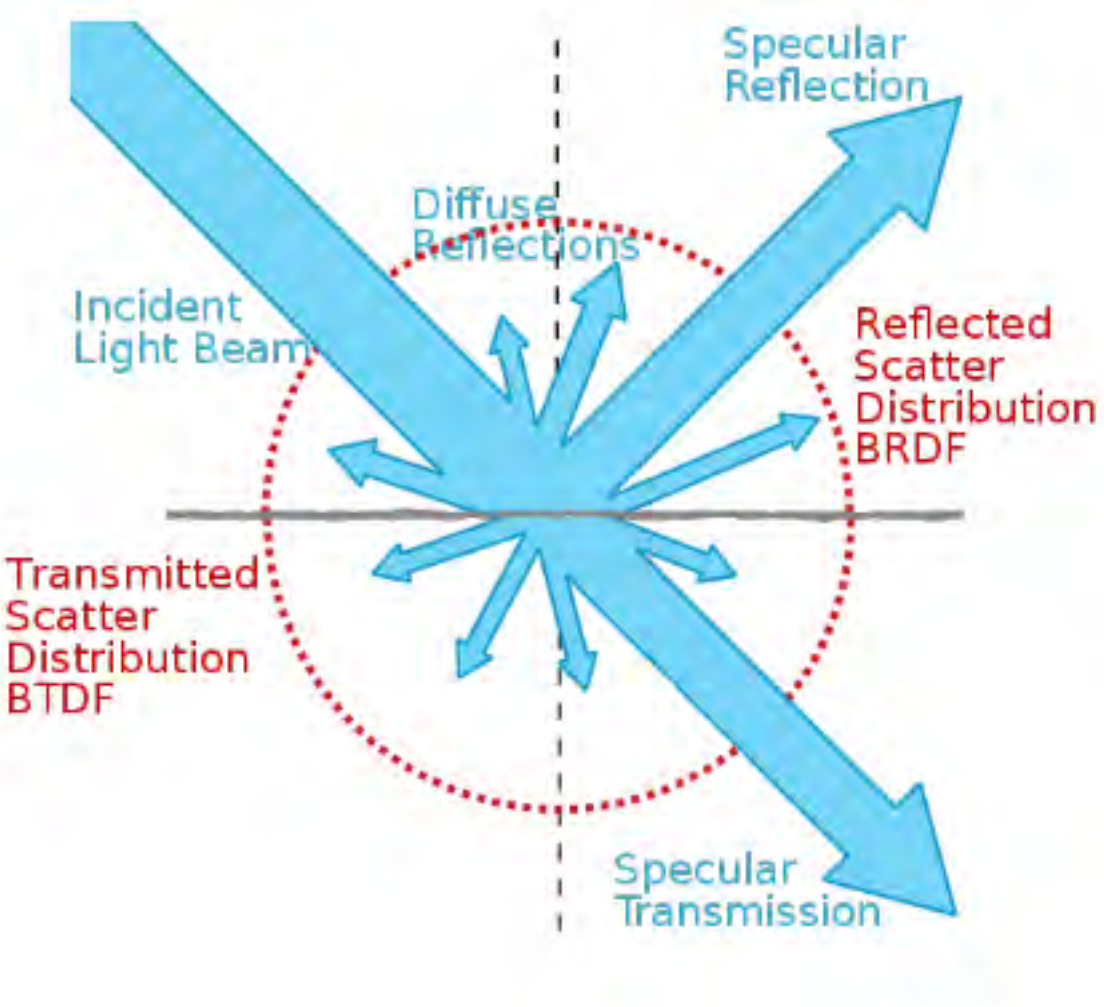


FIGURE 3.2 – Une BSDF décrit le comportement d'un matériau pour une luminance donnée. Elle se décrit de deux composantes, l'une réflective nommée BRDF, l'autre transmittive nommée BTDF.

4

Structures d'accélération

Le but des structures d'accélération est de réduire le nombre d'appels à la méthode d'intersection rayon/triangle à exécuter en ciblant plus rapidement les groupes de primitives géométriques ayant de fortes probabilités d'être intersectés par un rayon donné. Elles permettent théoriquement de réduire la complexité des calculs d'intersections rayon/-triangle de O(N) pour N triangles à O(log(N)). Bien que cette complexité soit une limite théorique ne correspondant à aucun modèle réaliste, les travaux d'Havran montrent que cette propriété est vérifiée pour la très grande majorité des modèles en synthèse d'images dans [Hav00]. La structure d'accélération est très importante dans le sens où elle va directement définir les performances des sessions de lancer de rayons, mais implique aussi un temps de construction avant de pouvoir réaliser une session de lancer de rayons. Faisons un tour d'horizon des structures d'accélération les plus couramment utilisées.

4.1 Grilles régulières

La grille régulière est surtout réputée pour sa simplicité, c'est en effet la plus simple des subdivisions spatiales. Celle-ci fit son apparition dans le domaine du lancer de rayon dans les années 80 notamment grâce à Cleary dans [CW88] et fut intégré à l'un des premiers système d'accélération du lancer de rayons, présenté par Fujimoto, nommé ARTS [FTI88]. La construction ainsi que la traversée d'une grille régulière sont très aisées. Elles montrent toutefois rapidement leurs limites sur des scènes contenant des géométries à différentes échelles, ou en terme de compacité mémoire.

4.1.1 Définition

Une grille régulière découpe régulièrement l'espace sur chacun des axes dimensionnels. Chacune des cellules, de mêmes volumes, représente donc de manière unique une partie

bornée de l'espace. Les cellules englobent strictement tout l'espace existant dans la scène. Chacune des cellules contient une liste des triangles contenus, totalement ou partiellement, dans l'espace qu'elle représente.

4.1.2 Construction

Le seul paramètre qui détermine la qualité d'une grille régulière se situe dans le choix du pas d'échantillonnage. Une fois ce pas choisi, les cellules sont générées à l'intérieur de la boîte englobante de la scène. Il suffit ensuite de calculer, pour chacun des triangles de la scène, les cellules qu'il intersecte afin de l'insérer dans la liste de chacune des cellules. La construction est détaillée dans l'algorithme 1 et imagée dans la figure 4.1.

Algorithm 1 Retourne la grille régulière GR construite pour la liste de triangles $listeTriangles$

function BUILDGRILLE($listeTriangles$)
  $GR \leftarrow allocGrille(boiteEnglobante(listeTriangles), pasEchantillonage)$
  for all $triangle$ dans $listeTriangles$ **do**
    $listeCellules \leftarrow cellulesIntersectees(triangle, GR)$
    for all $cellule$ dans $listeCellules$ **do**
      $ajouterTriangle(cellule, triangle)$
  return GR

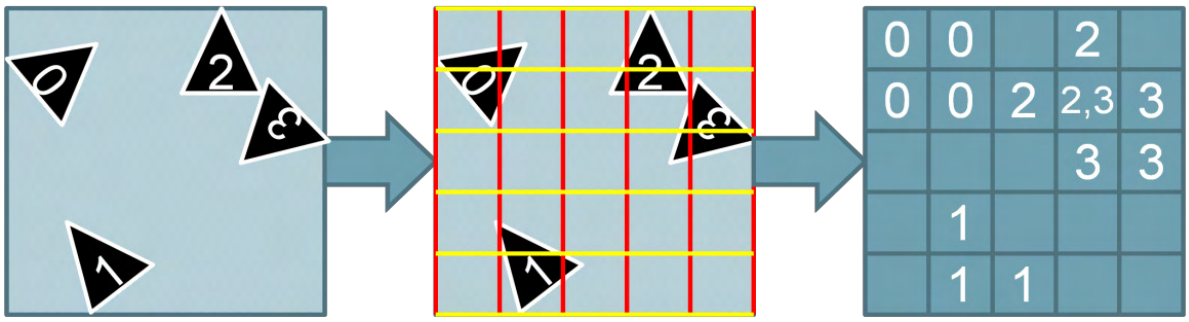


FIGURE 4.1 – Vue simplifiée du processus de construction d'une grille régulière. Découpage de l'espace en cellules régulières, pointant vers les triangles qu'elles contiennent.

4.1.3 Traversée

La traversée d'une grille régulière repose sur trois méthodes :

- *trouverCelluleDepart(rayon)* qui renvoie la cellule qui contient la position de départ du rayon passé en paramètre.

- *trouverProchaineCellule(rayon, cellule)* qui renvoie la prochaine cellule intersectée par le rayon en sortant de la cellule passée en paramètre, ou celluleInvalide si le rayon sort de la grille.

- *calculIntervalle(rayon, cellule)* qui calcule l'intervalle de distance composé de la distance à l'entrée de la cellule comme minimum, et de la distance de sortie de la cellule comme maximum. Cet intervalle permet d'assurer que l'intersection calculée soit bien située dans la cellule, et donc d'autoriser à mettre un terme à la traversée sans visiter d'autres cellules.

Le parcours se déroulant dans l'ordre des cellules traversées par le rayon, dès qu'une intersection est trouvée dans une cellule, il est inutile d'aller explorer la cellule suivante. La méthode de traversée est plus précisément décrite dans l'algorithme 2 et imagée par la figure 4.2.

Algorithm 2 Retourne l'intersection I la plus proche, pour le rayon R dans la grille régulière GR

function INTERSECTGRILLE(R, GR)
  $celluleCourante \leftarrow trouverCelluleDepart(R, GR)$
  $intervalleValide \leftarrow calculIntervalle(R, celluleCourante)$
  while $celluleCourante \neq celluleInvalide$ **do**
    $I \leftarrow intersectTriangles(celluleCourante, intervalleValide)$
    if $I \neq nonImpact$ **then**
      return I
    $celluleCourante \leftarrow trouverProchaineCellule(R, celluleCourante)$
    $intervalleValide \leftarrow calculIntervalle(R, celluleCourante)$
  return $nonImpact$

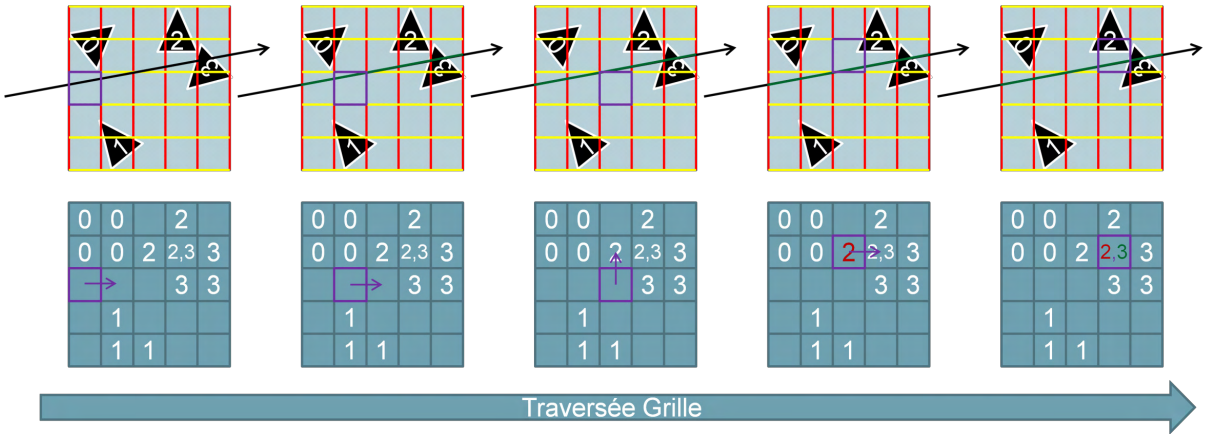


FIGURE 4.2 – Vue simplifiée du processus de traversée d'une grille régulière. Progression de cellule en cellule tant que la cellule courante de contient pas de triangle intersecté par le rayon.

4.2 KD-Tree

Le KD-Tree, proposé par Bentley [Ben75], a vu son intérêt dans le domaine du lancer de rayons croitre avec les travaux de Wald [WSBW01]. Celui-ci est réputé pour être une structure d'accélération très efficace pour le lancer de rayons. Cette efficacité est toutefois contrebalancée par un coût de construction conséquent et une occupation mémoire conséquente.

4.2.1 Définition

Un KD-Tree est un arbre de subdivision spatiale de l'espace. Chacun de ses nœuds correspond à un plan aligné à un axe, subdivisant l'espace. Ils contiennent, de plus, les liens vers leurs fils droit et gauche. Ses feuilles représentent donc une localisation spatiale contrainte par les plans de coupe définis par leurs ancêtres au sein de l'arbre. Appliqué au domaine du lancer de rayons, les plans de coupes des nœuds se limitent aux trois axes de l'espace, et les feuilles contiennent les données nécessaires à l'intersection des triangles effectivement contenus dans leur volume.

4.2.2 Construction

La construction récursive d'un KD-Tree est décrite dans l'algorithme 3 et imagée par la figure 4.3, en invoquant la méthode de construction avec la liste des triangles de la scène et leur boîte englobante comme arguments.

Algorithm 3 Retourne le nœud du KD-Tree construit pour la boîte englobante *boiteEnglobante* et la liste de triangles *listeTriangle*

```
function buildKDRec(listeTriangle, boiteEnglobante)
    if vide(listeTriangle) then
        return feuilleVide
    listePlansCandidats ← trouverCandidats(boiteEnglobante, listeTriangle)
    meilleurPlan ← trouverMeilleurCandidat(listePlansCandidats)
    coutSubdivision ← evalCoutSubdivision(meilleurPlan)
    coutCreationFeuille ← evalCoutCreationFeuille(boiteEnglobante, listeTriangle)
    if coutSubdivision < coutCreationFeuille then
        listeTriangleGauche ← trouverTrianglesGauche(listeTriangle, meilleurPlan)
        listeTriangleDroite ← trouverTrianglesDroite(listeTriangle, meilleurPlan)
        boiteEnglobanteGauche ← subdiviserGauche(boiteEnglobante, meilleurPlan)
        boiteEnglobanteDroite ← SubdiviserDroite(boiteEnglobante, meilleurPlan)
        filsGauche ← buildKDRec(listeTriangleGauche, boiteEnglobanteGauche)
        filsDroit ← buildKDRec(listeTriangleDroite, boiteEnglobanteDroite)
        return nœud(filsGauche, filsDroit, meilleurPlan)
    else
        return feuille(listeTriangle)
```

4.2.3 Optimalité

La qualité d'un KD-Tree se définie par les performances obtenues lors de son exploitation, c'est à dire sa capacité à accélérer le lancer de rayons dans le cas général. Il existe plusieurs points lors de la construction d'un KD-Tree sur lequel il est possible de réaliser des approximations afin de réduire le temps de construction. Ces approximations résultent en une perte de qualité, ce qui induit une perte de performance lors de l'exploitation du KD-Tree. Un KD-Tree dit "optimal" au sens de Wald et Havran [WH06] s'appuie sur trois points critiques lors de la construction du KD-Tree :

- l'heuristique d'évaluation des plans de subdivision.
- la sélection des plans candidats à la subdivision.
- le raffinement des limites des triangles.

Chacun de ses points va être abordé plus bas, et chacun d'eux peut être simplifié afin de favoriser la vitesse de construction au détriment de la qualité du KD-Tree.

Heuristique d'évaluation des plans de subdivision

Afin dévaluer si un nœud doit être subdivisé, et si oui par quel plan, il est nécessaire de pouvoir affecter un coût à tous les plans de coupe candidats. Plusieurs stratégies peuvent

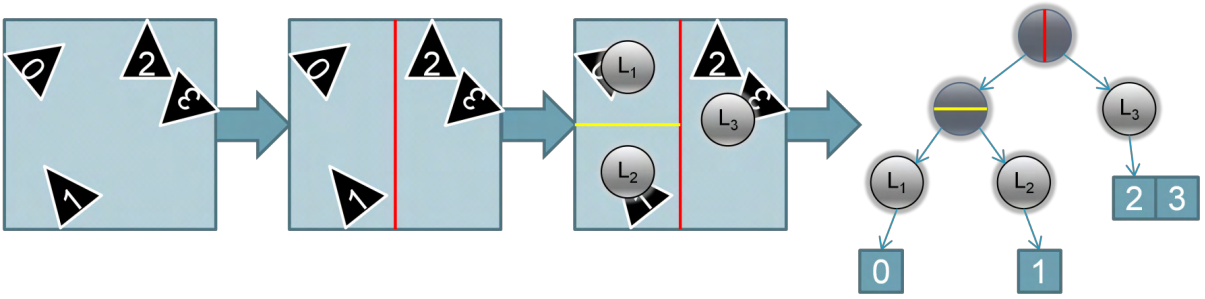


FIGURE 4.3 – Vue simplifiée du processus de construction d'un KD-Tree. Découpage de l'espace en feuilles, pointant vers les triangles qu'elles contiennent, définies par des plans de coupes alignés aux axes.

être mises en place, mais il est une heuristique proposée par MacDonald qui fait référence en terme de qualité de l'arbre final : "Surface Area Heuristic" [MB90]. Celle-ci est conçue pour être optimale dans le cas général, et donc pour du lancer de rayons incohérent. Concrètement, cette formule décrite par l'équation 4.1 permet d'obtenir un coût C_p pour un plan de coupe P_c à partir :

- d'une estimation du coût d'ajout d'un niveau de profondeur à l'arbre C_{trav}
- d'une estimation du coût d'un appel à la méthode d'intersection rayon/triangle C_{inter}
- de la répartition des triangles de chaque côté du plan de coupe T_{left} et T_{right}
- du rapport entre l'aire surfacique de la boîte englobante du nœud à subdiviser $SA(N)$ et l'aire surfacique de chacun de ces deux nœuds fils envisagés $SA(N_{left})$ et $SA(N_{right})$

Il est courant de rajouter un bonus à cette heuristique afin de privilégier la création de feuilles vides. Celles-ci ont pour propriété de ne pas générer d'appel à la méthode d'intersection rayon/triangle. Il est donc rentable de détecter au plus tôt qu'un rayon traverse un espace totalement vide, afin d'économiser des tests d'intersections.

$$C_p(P_c) = C_{trav} + C_{inter}\left(\frac{SA(N_{left})}{SA(N)}T_{left} + \frac{SA(N_{right})}{SA(N)}T_{right}\right) \tag{4.1}$$

Sélection des plans candidats à la subdivision

Les plans candidats sont très important pour la qualité finale du KD-Tree. En effet, ils représentent les uniques subdivisions envisagées et évaluées. Il est raisonnablement impossible d'évaluer toutes les possibilités de subdivision d'un espace donné. Une étude

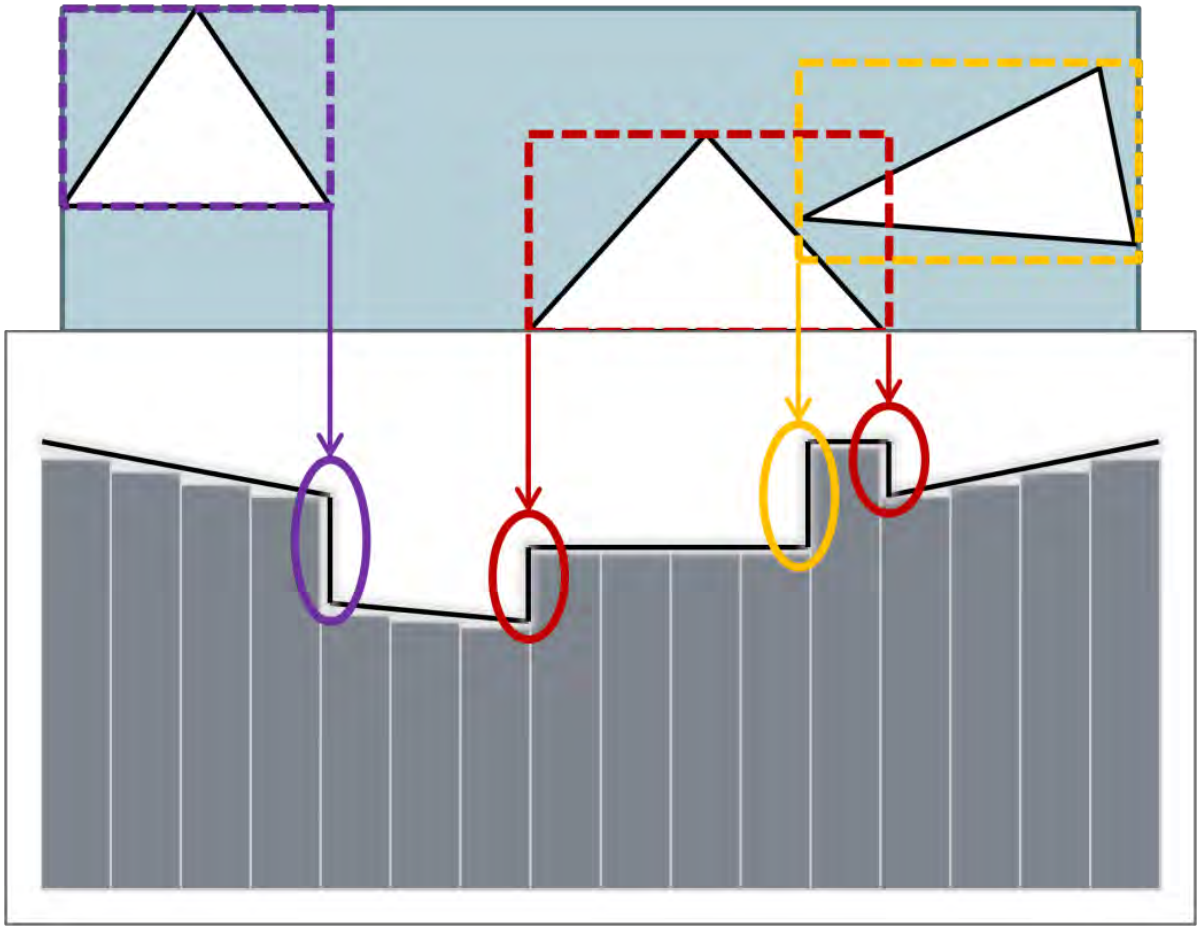


FIGURE 4.4 – Évolution de la fonction d'estimation du coût de la subdivision en fonction de sa position spatiale.

de l'évolution de la fonction de coût SAH, illustrée dans la figure 4.4, révèle que ses hautes fréquences se situent sur les frontières des triangles au sein de l'espace évalué. De plus, celle-ci évolue de manière linéaire entre deux frontières de triangles consécutives. Ces propriétés permettent de déduire que les extremum de cette fonction se situent à la frontière d'un triangle. Afin de minimiser cette fonction de coût, il est donc judicieux d'évaluer son comportement sur les limites des triangles.

Raffinement des limites des triangles

Lorsque qu'un nœud est subdivisé par un plan de coupe, il arrive que ce plan intersecte des triangles contenus dans le nœud. Une source d'amélioration de la qualité du KD-Tree consiste à raffiner les limites des triangles intersectés par un plan de coupe afin d'obtenir ses limites réelles au sein des deux nœuds fils générés par la subdivision. Cela permet :

- d'éviter à ce triangle d'apparaître inutilement dans des feuilles de l'arbre qui ne le

contiennent pas réellement, voir figure 4.5. Et donc d'éviter des appels à la méthode d'intersections rayon/triangle inutiles lors de la traversée.

- d'utiliser les limites raffinées pour la sélection de plans candidats dans les nœuds fils, et donc d'améliorer les estimations de coûts de subdivision pour les nœuds descendants.

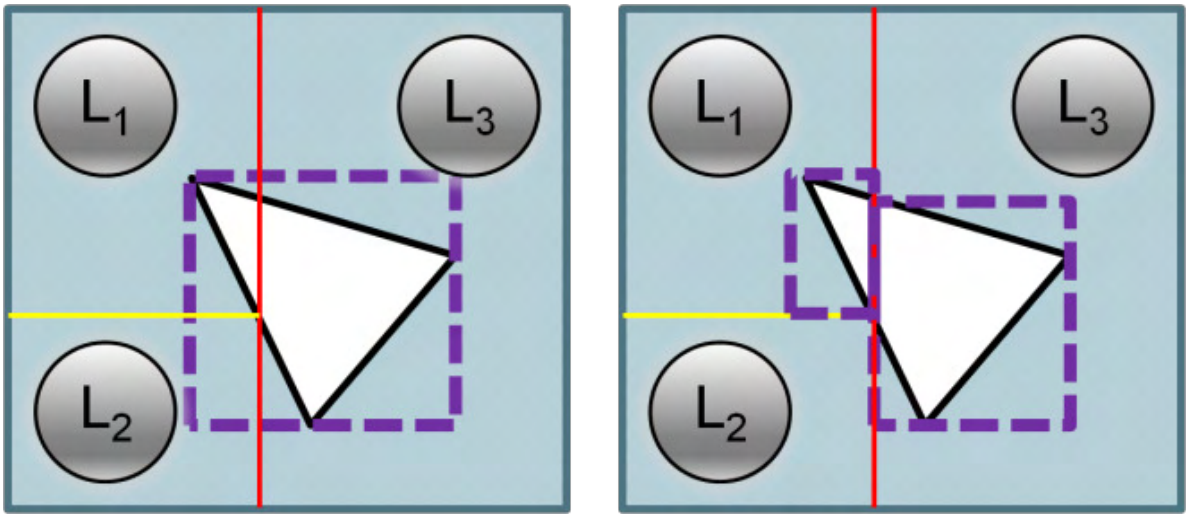


FIGURE 4.5 – Raffiner la boîte englobante du triangle au cours des subdivisions(image de droite) permet d'éviter qu'il n'apparaisse dans la feuille L2 du KD-Tree comme ce serait la cas si les limites étaient préservées(image de gauche).

4.2.4 Traversée

La traversée d'un KD-Tree se déroule de manière à garantir que la première intersection valide retournée par une feuille de l'arbre pour un rayon donné est la plus proche. Cela permet de stopper la traversée dès lors qu'une feuille retourne une intersection. Cette traversée fait appel à trois méthodes :

- *filsProche(nœud, rayon, intervalle)* qui va renvoyer le premier fils du nœud à être intersecté par le rayon et l'intervalle de validité du rayon dans ce nœud fils.

- *filsLoin(nœud, rayon, intervalle)* qui va renvoyer le second fils du nœud à être, ou pas, intersecté par le rayon et l'intervalle de validité du rayon dans ce nœud fils.

- *intersectionPossible(intervalle)* qui va retourner vrai si l'intervalle passé en paramètre est valide. Concrètement, ce test rejette le second nœud fils à visiter si

il n'y a aucune chance d'obtenir une intersection dans l'intervalle retourné par la méthode filsLoin.

Il existe plusieurs méthode pour réaliser cette traversée. La première, récursive, requiert d'être invoquée avec le nœud racine du KD-Tree en argument et est décrite dans l'algorithme 4. La seconde, directement issue de la dérécursivation de la première, utilise

Algorithm 4 Retourne l'intersection I la plus proche, dans l'intervalle In, pour le rayon R dans le nœud N

function INTERSECTKDREC(R, In, N)
  if N est une feuille vide **then**
    $I \leftarrow nonImpact$
    return I
  if N est une feuille **then**
    $I \leftarrow intersectTriangle(N, R, In)$
    return I
  // N est un nœud
  $Nproche, InProche \leftarrow filsProche(N, R)$
  $Nloin, InLoin \leftarrow filsLoin(N, R)$
  $I \leftarrow intersectKDRec(R, InProche, Nproche)$
  if $I = nonImpact$ and intersectionPossible(InLoin) **then**
    $I \leftarrow intersect(R, InLoin, Nloin)$
  return I

une pile de nœuds à visiter et est décrite dans l'algorithme 5 et imagée dans la figure 4.6.

Plus récemment, une nouvelle méthode de traversée a été proposée par Foley dans [FS05], afin de répondre au problème de la gestion d'une pile de traversée sur GPU. Celle-ci propose plusieurs variantes, mais l'idée principale repose sur la version dite "KD-Restart". Le principe est de relancer la traversée en modifiant l'intervalle de validité des intersections lorsque l'on atteint une feuille de l'arbre sans trouver d'intersection. Celle-ci est décrite dans l'algorithme 6.

Algorithm 5 Retourne l'intersection I la plus proche, dans l'intervalle In, pour le rayon R dans le KD-Tree KD

```
function intersectKDStack(R, In, KD)
    empiler(pile, racine(KD), In)
    while nonVide(pile) do
        NCourant, InCourant ← depiler(pile)
        while NCourant est un nœud do
            Nproche, InProche ← filsProche(NCourant, R, InCourant)
            Nloin, InLoin ← filsLoin(NCourant, R, InCourant)
            NCourant ← Nproche
            InCourant ← InProche
            if intersectionPossible(InLoin) then
                empiler(pile, Nloin, InLoin)
        if [ thensi NCourant est une feuille vide, ne rien faire]NCourant est une feuille
            I ← intersectTriangle(NCourant, R, InCourant)
            if I ≠ nonImpact then
                return I
    return nonImpact
```

Algorithm 6 Retourne l'intersection I la plus proche, dans l'intervalle In, pour le rayon R dans le KD-Tree KD

```
function intersectKDStackless(R, In, KD)
    InRestart ← In
    while intersectionPossible(InRestart) do
        NCourant ← racine(KD)
        InCourant ← InRestart
        while NCourant est un nœud do
            Nproche, InProche ← filsProche(NCourant, R, InCourant)
            Nloin, InLoin ← filsLoin(NCourant, R, InCourant)
            NCourant ← Nproche
            InCourant ← InProche
            if intersectionPossible(InLoin) then
                InRestart ← InLoin
        if [ thensi NCourant est une feuille vide, ne rien faire]NCourant est une feuille
            I ← intersectTriangle(NCourant, R, InCourant)
            if I ≠ nonImpact then
                return I
    return nonImpact
```

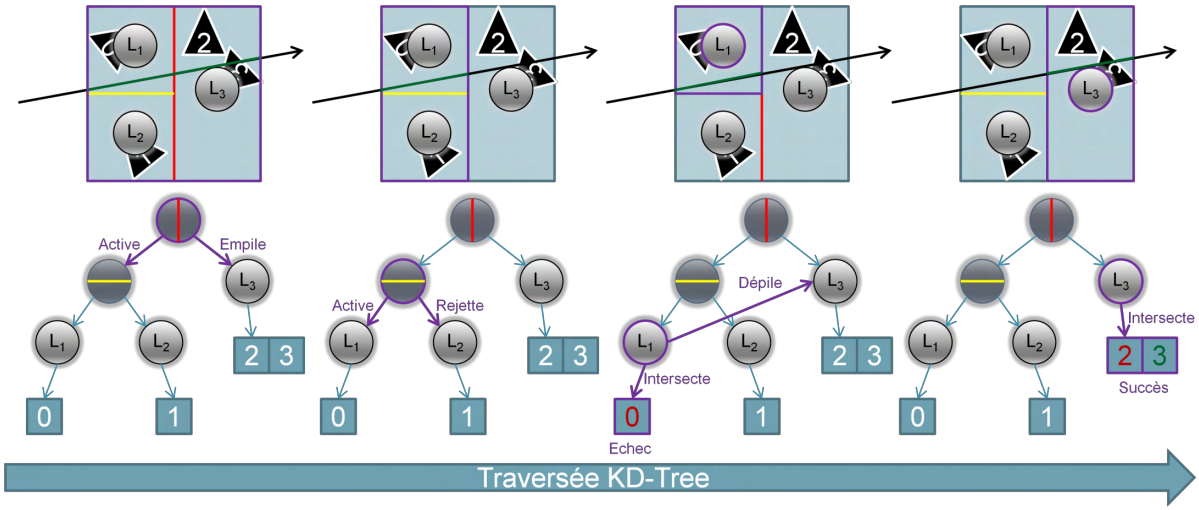


FIGURE 4.6 – Vue simplifiée du processus de traversée d'un KD-Tree. Progression de nœud en nœud tant que l'on n'atteint pas une feuille contenant un triangle intersecté par le rayon.

4.3 BVH

Mal noté et négligé dans les tests d'Havran dans [Hav00], les BVHs ont fait l'objet d'un intérêt nouveau suite aux travaux de Wald [WBS07] qui, en s'inspirant des améliorations apportées au Kd-Tree, prouvent l'intérêt des BVH en particulier pour des scènes dynamiques en raison de leur rapidité de construction. Outre le temps de construction, les BVH ont aussi pour propriété d'être une structure très compacte en mémoire.

4.3.1 Définition

Une BVH, hiérarchie de volumes englobants, est un arbre binaire de subdivision des primitives géométriques. Chacun de ses nœuds contient une boite englobante et les liens vers ses fils droit et gauche. Les nœuds d'une BVH ont pour propriété de pouvoir se chevaucher spatialement. Les feuilles, quant à elles, contiennent à la fois une boîte englobante et une liste de triangles totalement contenus dans la boîte englobante. Une propriété intéressante que peut garantir une BVH est l'unicité de la référence d'un triangle par une feuille. En contrepartie, une position spatiale donnée peut appartenir à plusieurs feuilles de la BVH.

4.3.2 Construction

La construction récursive d'une BVH est décrite dans l'algorithme 7 et imagée dans la figure 4.7, en invoquant la méthode de construction avec la liste des triangles de la scène

et leur boîte englobante comme arguments.

Algorithm 7 Retourne le nœud du BVH construit pour la boîte englobante *boiteEnglobante* et la liste de triangles *listeTriangle*

function BUILDBVHREC(*listeTriangle*, *boiteEnglobante*)
  $listePlansCandidats \leftarrow trouverCandidats(boiteEnglobante, listeTriangle)$
  $meilleurPlan \leftarrow trouverMeilleurCandidat(listePlansCandidats)$
  $coutSubdivision \leftarrow evalCoutSubdivision(meilleurPlan)$
  $coutCreationFeuille \leftarrow evalCoutCreationFeuille(boiteEnglobante, listeTriangle)$
  if $coutSubdivision < coutCreationFeuille$ **then**
    $listeTriangleGauche \leftarrow trouverTrianglesGauche(listeTriangle, meilleurPlan)$
    $listeTriangleDroite \leftarrow trouverTrianglesDroite(listeTriangle, meilleurPlan)$
    $boiteEnglobanteGauche \leftarrow boiteEnglobante(listeTriangleGauche)$
    $boiteEnglobanteDroite \leftarrow boiteEnglobante(listeTriangleDroite)$
    $filsGauche \leftarrow buildBVHRec(listeTriangleGauche, boiteEnglobanteGauche)$
    $filsDroit \leftarrow buildBVHRec(listeTriangleDroite, boiteEnglobanteDroite)$
    return $nœud(filsGauche, filsDroit, boiteEnglobante)$
  else
    return $feuille(listeTriangle, boiteEnglobante)$

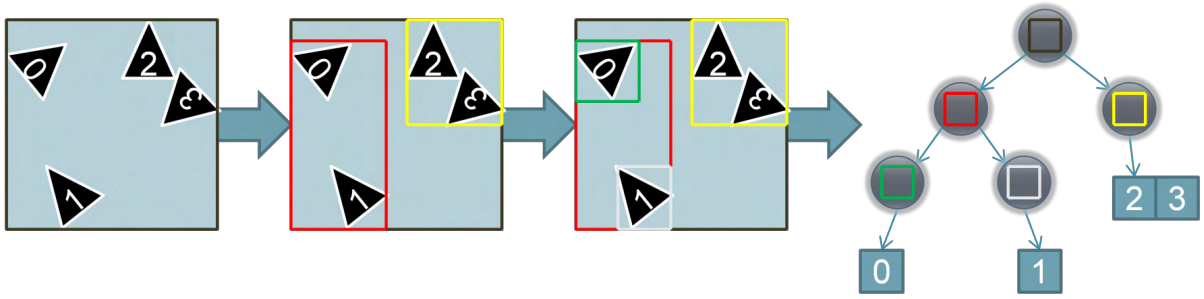


FIGURE 4.7 – Vue simplifiée du processus de construction d'une BVH. Découpage de l'espace par regroupements récursifs de primitives géométriques par proximité spatiale.

4.3.3 Traversée

La version récursive de la traversée du BVH requiert d'être invoquée avec le nœud racine du BVH et une intersection initialisée avec nonImpact en argument et est décrite dans l'algorithme 8. Cette méthode de traversée ne peut recourir à des optimisations permettant d'ignorer la traversée de nœuds suite à la découverte d'un impact. En effet, rien ne garantit que les boîtes englobantes des deux fils ne se chevauchent pas, et donc qu'un impact plus proche ne se situe pas dans les nœuds suivants.

De la même manière que pour le KD-Tree, une version non récursive de la traversée

Algorithm 8 Retourne l'intersection I la plus proche, pour le rayon R dans le nœud N

function INTERSECTBVHREC(R, N)
  if $Nestunnœud$ **then**
    $NGauche \leftarrow filsGauche(N)$
    $NDroit \leftarrow filsDroit(N)$
    if $intersectBVHRec(R, boiteEnglobante(NGauche))$ **then**
      $I \leftarrow plusProche(intersect(Nproche, R), I)$
    if $intersectBVHRec(R, boiteEnglobante(NDroit))$ **then**
      $I \leftarrow plusProche(intersect(NDroit, R), I)$
    return I
  else[N est une feuille]
    $I \leftarrow plusProche(intersectTriangles(N), I)$

du BVH utilise une pile de nœuds à visiter et est décrite dans l'algorithme 9 et imagée dans la figure 4.8.

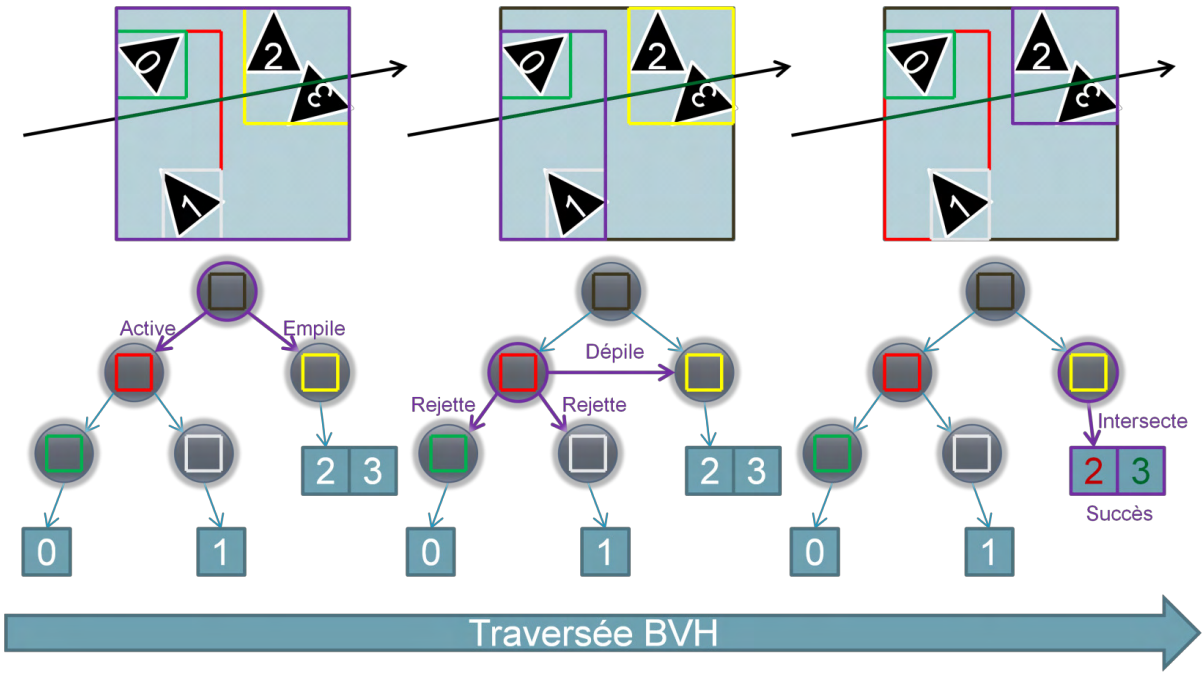


FIGURE 4.8 – Vue simplifiée du processus de traversée d'une BVH. Progression de nœud en nœud tant que l'on n'a pas visité toutes les feuilles intersectées par le rayon.

Algorithm 9 Retourne l'intersection I la plus proche, pour le rayon R dans la BVH BVH

```
function intersectBVHStack(R, BVH)
    empiler(pile, racine(BVH))
    I ← nonImpact
    while nonVide(pile) do
        NCourant ← depiler(pile)
        while NCourant est un nœud valide do
            NGauche ← filsGauche(N)
            NDroit ← filsDroit(N)
            intersectGauche ← intersect(R, boiteEnglobante(NGauche))
            intersectDroit ← intersect(R, boiteEnglobante(NDroit))
            NCourant ← NInvalide
            if intersectGauche et intersectDroit then
                NCourant ← NGauche
                empiler(pile, NDroit)
            else
                if intersectGauche then
                    NCourant ← NGauche
                if intersectDroit then
                    NCourant ← NDroit
        if NCourant est un feuille valide then
            I ← plusProche(intersectTriangles(NCourant), I)
    return I
```

4.4 Choisir une structure d'accélération

Le choix de la structure d'accélération doit prendre plusieurs facteurs en considération. Le premier concerne directement le type de géométrie que l'on souhaite pouvoir importer. En effet, la grille régulière peut se révéler extrêmement efficace en terme de temps de construction et de traversée, mais elle n'est pas résistante à tous les types de géométrie en terme de performances. Les subdivisions binaires de l'espace, que ce soit le KD-Tree ou la BVH, permettent de s'affranchir de ces considérations géométriques. La BVH a comme principale qualité son temps de construction, qui lui permet de gérer efficacement des scènes dynamiques et sa compacité mémoire. Le KD-Tree s'impose quant à lui comme la structure permettant d'atteindre les plus hautes performances dans le cas général en terme de vitesse de traversée, au prix d'un coût de construction et d'une occupation mémoire supérieure aux BVH. Le choix de la structure d'accélération dépend donc aussi au final du type d'application, et plus particulièrement du nombre de rayons qui vont être lancés dans la scène avant de reconstruire la structure. En effet, afin que l'accélération due à l'utilisation d'un KD-Tree soit rentable face à une BVH, il est nécessaire que le surcoût en terme de temps de construction soit amorti lors du lancer de rayons.

5

Parallélisation

La parallélisation de l'algorithme du lancer de rayons est depuis longtemps un sujet de recherche. En effet, tout a commencé sur des machines multiprocesseurs dans les années 90 avec les travaux de Green [GP89], [GP90]. Plusieurs architectures parallèles spécifiquement conçues pour le lancer de rayons ont émergées, comme l'architecture SaarCOR [SWS02] ou la carte dédié Caustic [com]. Toutefois, aucun de ces produits n'a aujourd'hui atteint l'étape de commercialisation de masse. La parallélisation du lancer de rayons a donc suivie l'évolution du matériel disponible sur le marché, en commençant par l'utilisation des instructions SIMD des CPU ainsi que de leur nombre de cœurs de calculs grandissant avant de franchir le pas de l'utilisation du GPU pour un parallélisme massif. Même si l'on était loin du parallélisme impliqué par les GPU actuels, les instructions SIMD permettant de traiter quatre données simultanément, huit sur les CPU les plus récents, bon nombre de concepts et de constats restent aujourd'hui valides. Voici un récapitulatif des diverses approches de parallélisation du lancer de rayons d'une part, puis de la construction de la structure d'accélération d'autre part.

5.1 Lancer de rayons

En soit, l'algorithme du lancer de rayons se prête plutôt bien à la parallélisation. En effet, une fois générés, les rayons sont indépendants les uns des autres et peuvent donc être traités parallèlement par des unités de calcul différentes. Cette méthode permet d'atteindre des performances proportionnelles avec le nombre d'unités de calcul, sauf en cas de conflit de caches mémoires si les unités partagent des caches mémoires.
Avec la démocratisation des instructions SIMD, l'idée de regrouper les rayons par groupes de quatre a émergée en particulier via les recherches de Wald [Wal04]. Quelque soit la structure d'accélération utilisée, cette organisation permet d'obtenir des facteurs d'accélération de l'ordre de x2-x3 mais seulement si les rayons mis en paquet sont cohérents. En effet,

les traiter par paquets implique qu'ils traversent la structure et intersectent les triangles de manière synchrone. Or si un seul des rayons doit intersecter un nœud donné, tous les autres seront mis en attente par un mécanisme de masque. La cohérence des rayons signifie qu'ils doivent globalement intersecter les mêmes nœuds, et donc présenter le moins possible de divergences directionnelles et positionnelles.
Afin d'amortir les coûts d'accès mémoire, démontrés facteur limitant principal par Wald [WSBW01], lors de la traversée et de l'intersection avec les triangles, Overbeck propose d'aller plus loin en construisant des faisceaux de rayons [ORM08]. Ceux-ci sont concrètement représentés par quatre rayons délimitant le volume englobant tous les rayons contenus dans le faisceau. Ces quatre rayons vont définir les nœuds à traverser dans la structure d'accélération et ainsi factoriser des chargements mémoire qui auraient eu lieu pour tous les rayons contenus dans le faisceau. Encore une fois, cela permet d'augmenter les performances mais impose une contrainte ferme de cohérence au sein des faisceaux. La cohérence d'un faisceau n'est pas simple à définir. Elle dépend de nombreux paramètres tels que la proximité et l'alignement des rayons entre eux, de l'agencement et des dimensions des surfaces dans une scène, et de la nature de la structure d'accélération. Cadet propose par exemple une heuristique permettant de juger de la cohérence d'un paquet afin de le générer au mieux dans [Cad08].
Cette cohérence des rayons reste un facteur majeure dans l'obtention de performances sur GPU. En effet, son modèle d'exécution SIMT souffre de la même limitation mais appliquée à des paquets de la taille d'une *warp*, c'est à dire trente-deux actuellement. Affecter un rayon à chaque thread GPU pose finalement des problèmes en terme de besoin de cohérence très proches de ceux rencontrés avec l'utilisation d'instructions SIMD sur CPU. En effet, les accès non coalescents de chacun des threads peuvent entraîner un surcoût non négligeable. De plus les threads étant synchronisés par paquets de trente-deux, tant que l'un des threads n'a pas terminé une tâche, les autres patientent. C'est pourquoi les faisceaux de rayons sur GPU ont aussi été explorés. En effet, les accès mémoires sur GPU se révèlent encore plus critiques que sur CPU. Afin de résoudre le problème de cohérence, Garanzha propose un pipeline de rendu très efficace comprenant une étape de tri des rayons selon un critère simplifié de cohérence [GL10].
Toutefois, ces méthodes sont la plupart du temps appliquées à des rendus comprenant des types de rayons dont il est possible d'extraire une cohérence certaine. Les rayons caméra ont par exemple tous la même origine et des directions très proches pour des pixels proches. Les rayons d'ombre peuvent facilement être assez cohérents. En effet, pour plusieurs pixels voisins il y a de fortes probabilités que les impacts soient proches, et donc que les origines des rayons d'ombres soient proches et, de plus, leurs directions seront aussi voisines car elles visent une source lumineuse précise. Dans le cas moins favorable de rayons for-

tement incohérents, tels que ceux qu'un solveur de type Monte-Carlo peut en produire, les méthodes précédentes, qui parient sur la l'extraction de groupes de rayons cohérents, sont vouées à l'échec. C'est pourquoi le moteur de rendu par lancer de rayons EMBREE dernièrement développé par Intel [INTa], qui utilise du Monte-Carlo pour générer des rendus, utilise les instructions SIMD pour réaliser des tests d'intersections entre un rayon et plusieurs triangles plutôt que l'inverse. Même si cette approche apporte un gain inférieur au gain maximal obtenu dans le cas favorable par la traversée conjointe de plusieurs rayons cohérents, elle a le gros avantage de permettre un gain stable, à l'inverse des méthodes par rayons cohérents qui, de par l'ajout d'une phase d'analyse de la cohérence des rayons, peuvent même conduire à une dégradation des performances.

5.2 Structures d'accélération

Tirer efficacement parti du parallélisme pour la construction d'une structure d'accélération peut se révéler très simple et efficace, comme c'est le cas pour les subdivisions régulières de l'espace et en particulier les grilles régulières. Mais cela peut aussi poser beaucoup plus de problèmes lorsque l'on s'attaque aux subdivisions binaires spatiales, et donc entre autres aux KD-Tree et BVH.

5.2.1 Subdivisions régulières de l'espace

Le gros avantage lors de la construction des subdivisions régulières de l'espace réside dans le fait que le partitionnement de l'espace n'est en aucun cas influencé par la géométrie qu'il va contenir. Cela permet donc, à partir d'un triangle, de connaître exactement les positions des cellules dans lesquelles il va devoir se positionner.
Cette propriété permet donc d'attribuer un paquet de triangles à chacune des unités de calcul, qui va évaluer les cellules concernées par chacun des triangles et aller y insérer les références vers les triangles. Sur CPU, il suffit de s'assurer que l'insertion dans une cellule par plusieurs threads simultanément soit sécurisée et suffisamment performante. Plusieurs solutions sont exposées et comparées dans les travaux de Lagae [LD08].
Sur GPU il est nécessaire de passer par une phase de comptage du nombre de références vers les triangles que chacune des cellules va contenir. En effet, il est impossible à ce jour d'allouer dynamiquement de la mémoire, il faut donc déléguer cette tâche au CPU entre la phase de comptage des références et leurs insertions, qui peuvent toutes deux être réalisées sur GPU comme le démontre Kalojanov dans [KS09].

5.2.2 Subdivisions binaires de l'espace

Le gros problème lors de la parallélisation de la construction d'une subdivision binaire de l'espace réside dans les nœuds proches de la racine. En effet, la façon la plus naturelle de procéder semble d'affecter un nœud à construire à chaque unité de calcul. Si cette méthode fonctionne très bien dès lors qu'une certaine profondeur de l'arbre est atteinte, beaucoup d'unités sont en attente d'obtention d'un nœud au début du processus. De plus, avec l'augmentation constante du nombre d'unités de calcul, la profondeur idéale de l'arbre pour exploiter toutes les unités grimpe, et donc de plus en plus de puissance de calcul est gaspillée. Il faut aussi prendre en compte le fait que plus un nœud est proche de la racine, plus il est probable qu'il contienne beaucoup de géométrie à traiter, et donc que sa construction soit longue.
Afin de réduire l'attente des unités de calculs, une approche courante consiste à dégrader la qualité des nœuds en haut de l'arbre afin de les traiter plus vite jusqu'à obtenir suffisamment de nœuds à traiter pour les unités de calcul et basculer en subdivisions de qualités. Une méthode courante de split rapide consiste à subdiviser le nœud de manière à répartir les triangles équitablement de chaque côté, ou simplement de diviser l'espace par le milieu comme celle proposée par Zhou [ZHWG08]. Ces approches ne sont toutefois pas pérennes avec l'augmentation du nombre d'unités de calcul. En poussant à l'extrême ce principe, l'arbre finirait par être totalement construit avec la méthode de subdivision de basse qualité. De plus, en fonction de la scène, la qualité de l'arbre peut rapidement se dégrader et avoir de fortes répercutions sur le lancer de rayons.
A contre-pied des méthodes précédentes, Choi propose une autre approche consistant à exploiter le parallélisme à deux niveaux dans [CKL+10]. Tout d'abord les nœuds sont classifiés en deux catégories, les larges et les fins en fonctions du nombre de triangles qu'ils contiennent. Le traitement des nœuds larges sera ainsi réparti sur plusieurs unités de calculs alors que les nœuds fins seront affectés à une seule unité de calcul. Cette approche a le mérite de résoudre tous les problèmes listés plus haut tout en assurant un parallélisme continu et efficace durant toute la durée de la construction.
Très récemment, Karras a présenté un algorithme hautement parallèle permettant de construire des structures spatiales [Kar12]. Cette méthode repose sur un arbre radix (ou PATRICIA) permettant de générer un ensemble de subdivisions en parallèle à partir d'une représentation simplifiée des triangles(centroïdes) et d'une approximation de leurs positions(code MORTON). Cet algorithme est parfaitement adapté à une augmentation du nombre de cœur de calcul car il brise la barrière habituelle de nœud racine dont dépendent les nœuds fils. Si les temps de calculs sont impressionnants, cet algorithme utilise une approximation grossière de la géométrie pour réaliser les subdivisions. La qualité de la subdivision spatiale résultante, non discutée dans la publication, se retrouve donc amoin-

drie. Cela destine donc cette approche à des applications ayant des contraintes fortes sur le temps de construction de la structure d'accélération, comme les scènes animées interactivement par exemple.
Un constat général que l'on peut tirer suite à l'étude des différentes méthodes parallèles de constructions d'arbres binaires de subdivisions spatiales, réside dans le cloisonnement entre les périphériques. En effet, chaque méthode se propose d'une manière ou d'une autre de tirer au mieux parti du parallélisme du CPU ou du GPU, mais aucune ne propose d'approche hybride.

5.3 Impacts sur le choix d'une structure d'accélération

La parallélisation de la traversée et de la construction ne modifie en rien le choix entre une grille régulière et un arbre de subdivision binaire de l'espace. En effet, le problème du type géométrique reste présent, et si la grille régulière était adaptée à l'application, sa version parallèle le sera tout autant.
Pour ce qui est du choix de l'arbre de subdivision binaire de l'espace, Cadet propose une étude détaillée sur le comportement des Kd-Tree et BVH en présence de paquets de rayons dans sa thèse [Cad08]. Il en ressort que la BVH s'avère très performante lors de l'utilisation de paquets de rayons cohérents. Allant même jusqu'à supplanter les performances du KD-Tree pour les rayons caméra, voire même pour les rayons d'ombres. Le KD-Tree conserve toutefois un net avantage dans le cas général, c'est-à-dire pour des rayons secondaires, issus de l'évaluation de réflexions ou réfractions, ou plus largement pour des rayons omnidirectionnels.
Selon le type d'applications, la BVH peut donc devenir une meilleure alternative au KD-Tree, après parallélisation, si les rayons sont globalement cohérents.

6

KD-Tree hybride CPU/GPU

Le choix de notre structure d'accélération a été guidé par plusieurs de nos contraintes industrielles. La première concerne la géométrie à importer, nous ne pouvons en effet faire aucune supposition sur sa répartition spatiale. C'est pourquoi, l'idée d'utiliser une grille régulière a tout de suite été rejetée. La décision d'orienter nos recherches sur le KD-Tree et non sur le BVH s'explique par une étude de nos applications courantes du lancer de rayons, mais aussi par notre volonté d'efficacité générale, sans contrainte sur les données en entrée. En effet, nos applications actuelles tendent vers l'utilisation de méthodes à base de Monte-Carlo, et donc de rayons extrêmement incohérents. De plus, les rayons naturellement cohérents sont par défaut très rapides à traiter et ne représentent pas un point chaud sur lequel notre attention doit se focaliser particulièrement. En outre, notre bibliothèque de lancer de rayons doit être capable d'accélérer simplement tous type d'application de lancer de rayons en s'adaptant à l'existant, y compris les applications lançant les rayons un à un, domaine privilégié du KD-Tree. C'est pourquoi le KD-Tree s'est imposé, pour ses hautes performances dans le cas général.
La première partie de nos travaux a porté sur la construction d'un KD-Tree optimal, tel que présenté dans la section 4.2.3, de manière hybride CPU/GPU [RPC12a]. Notre but étant de parvenir à rendre ce coût de construction plus facile à amortir lors de la phase de lancer de rayons. Nous avons choisi de ne faire aucune concession sur la qualité de la construction et de répartir les tâches de constructions et de traversées entre le CPU et/ou le GPU en fonctions de leurs disponibilités et occupations respectives. En effet, nos contraintes industrielles nous imposent de pouvoir nous adapter au matériel disponible, et donc de pouvoir nous passer de le présence d'un GPU utilisable, ou de soulager le CPU pour qu'il puisse accomplir d'autres tâches pendant le lancer de rayons et le tout de manière dynamique. C'est pourquoi notre mode hybride signifie à la fois que chacun des périphérique est capable individuellement de réaliser la tâche de construction ou de traversée, mais aussi que tous les périphériques sont capables de collaborer pour réaliser

ces tâches efficacement en parallèle.

6.1 Construction

Nous nous sommes appuyés sur la méthode de construction décrite par Wald et Havran dans [WH06]. Celle-ci décrit en effet un moyen de réduire la complexité de la construction d'un KD-Tree de $O(N^2)$ ou $O(N \log^2 N)$ à $O(N \log N)$ par un mécanisme de maintient du tri initial des limites des triangles. Pour ce qui est de la méthode de parallélisation, nous empruntons la vision développée par Choi dans [CKL+10] qui consiste à exploiter le parallélisme à deux niveaux en fonction de la taille des nœuds à construire. Ces références nous permettent de respecter à la fois nos objectifs en terme de qualité du KD-Tree, en ne dégradant pas sa qualité pour maximiser le parallélisme, mais aussi nos contraintes de performances, en utilisant un algorithme de construction efficace. Le coût de construction d'un KD-Tree est majoritairement dû :

- aux tris des limites spatiales des triangles (voir figure 4.5), que ce soit le tri initial ou le maintien du tri après raffinement des limites des triangles.

- à l'évaluation de l'heuristique de subdivision pour les millions de plans candidats que peut contenir un nœud.

- au nombre de nœuds à traiter, directement dépendants de la scène ; la profondeur de l'arbre binaire peut amener à traiter des millions de nœuds.

6.1.1 Contributions

Afin d'accélérer la construction du KD-Tree, nous avons directement ciblé les points chauds identifiés plus haut. Les limites des triangles, que nous nommerons événements dans la suite du document, font l'objet de notre première contribution. nous proposons une représentation des événements permettant à la fois d'accélérer leur tri mais aussi d'alléger la phase d'évaluation de l'heuristique d'évaluation. Notre seconde contribution consiste en un algorithme GPU permettant d'accélérer le traitement des nœuds contenant un nombre de plans candidats à la subdivision important. Cet algorithme s'intègre dans notre dernière contribution, qui apporte un modèle de collaboration entre le CPU et le GPU afin de traiter parallèlement et efficacement le nombre important de nœuds générés lors de la construction d'un KD-Tree.

Représentation des événements

Chaque triangle contenu dans un nœud génère six limites, deux par axes, marquant le début et la fin du triangle. Ces limites des triangles au sein d'un nœud, appelées événements, sont généralement constituées :

- de la valeur de la limite sur l'axe courant.
- d'un marqueur indiquant le type DEBUT/FIN de l'événement.
- d'un entier désignant l'index du triangle qui a généré cet événement.

Pour rappel, ces événements déterminent à la fois les plans candidats à la subdivision du nœud qui les contient mais servent aussi de marqueurs à la présence d'un triangle au sein du nœud. La liste des événements dans un nœud doit être maintenue triée au fur et à mesure de la construction afin de pouvoir effectuer des optimisations indispensables pour le calcul de l'heuristique de subdivision. A chaque subdivision validée, il faut être capable de retirer des triangles qui n'appartiennent plus au nœud fils, mais aussi d'insérer de manière ordonnée des événements mis à jour suite à l'intersection entre les triangles et le plan de coupe. De la représentation classique des événements découle un surcoût non négligeable à l'exécution. En effet, le test de comparaison entre deux événements pour le tri, doit assurer qu'un événement de DEBUT de triangle soit toujours inférieur à un événement FIN de triangle en cas de valeur égales. Ceci requiert deux accès mémoires supplémentaires lors du test de comparaison.

Notre représentation encode le type d'événement directement dans le float. Afin de garantir une perturbation minimale de la valeur, seul le bit de poids faible du float est impacté. Pour les événements de type DEBUT de triangle, le bit de poids faible se voit attribuer la valeur du bit de signe du float. Inversement pour les événements de type FIN de triangle, le bit de poids faible se voit attribuer l'inverse de la valeur du bit de signe du float. Notre représentation est donc constituée :

- d'un float représentant la valeur de la limite sur l'axe courant, dont le type DEBUT/FIN de triangle peut être extrait par une simple comparaison entre le bit de signe et le bit de poids faible. S'ils sont égaux c'est un événement DEBUT de triangle, sinon c'est une événement FIN de triangle.
- d'un int désignant l'index du triangle qui a généré cet événement.

Cette représentation, illustrée par des exemples dans le tableau 6.1, a un très grand nombre d'avantages :

Valeur initiale	décimale binaire	0.0f 0000...0000	0.1f 0011...1101	-0.1f 1011...1101
Événement début de triangle	décimale binaire	0.0f 0000...0000	0.099999994f 0011...1100	-0.1f 1011...1101
Événement fin de triangle	décimale binaire	1.401e-45f 0000...0001	0.1f 0011...1101	-0.099999994f 1011...1100

TABLE 6.1 – Exemples d'encodages d'événements pour différentes valeurs. Les premiers et derniers bits sont porteur du type d'événement, égaux(en vert) indique un début de triangle, différents(en rouge) indique une fin de triangle

- Les événements de triangles alignés à l'axe sont automatiquement ordonnés. En effet, l'usage du bit de signe pour l'encodage assure que la valeur de l'événement DEBUT de triangle sera strictement inférieur à son correspondant FIN de triangle.

- La consommation mémoire est diminuée, un bool de moins par événement.

- Les tris de type clef/valeur hautes performances sont directement utilisables, sur des types primitifs et sans complexifier le test de comparaison avec un accès aux types des événements comparés.

- La perturbation du KD-Tree est minimale. Dans le pire des cas, les plans de coupes sont décalés à la suivante ou précédente valeur représentable dans l'espace des float.

- L'extraction du type d'événement ne requiert pas d'accès mémoire à un bool supplémentaire, mais une comparaison du bit de signe et du bit de poids faible de la valeur de l'événement

Répartition des tâches de construction

Le but de notre répartition est d'organiser la collaboration entre le CPU et le GPU afin de tirer avantage de leurs spécificités respectives. Toutefois, et conformément à la contrainte listée précédemment, notre répartition reste applicable dans le cas où le CPU ou le GPU serait indisponible.

Pour y parvenir, les nœuds à construire sont classifiés en deux catégories : les nœuds larges et les nœuds fins. Cette classification dépend du nombre de triangles Nt qu'ils contiennent. Le but de cette classification est de traiter les nœuds larges sur GPU afin d'exploiter le parallélisme massif sur leurs grands nombres d'événements, pendant que les nœuds fins sont traités sur le CPU. Si Nt est supérieur à un seuil fixé par l'utilisateur, le nœud est classifié large, sinon il est classifié fin. Ce seuil doit être fixé en fonction de plusieurs facteurs :

- la profondeur maximale du KD-Tree.

- le nombre total de triangles de la scène.

- le ratio de puissance entre le CPU et le GPU.

Le GPU va de préférence créer le nœud racine, c'est à dire générer les événements de chacun des triangles sur les trois dimensions, puis effectuer le tri global de tous les événements. Ensuite le thread GPU va se mettre à traiter les nœuds larges, un à un, en profondeur d'abord afin de générer au plus vite des nœuds fins pour le CPU. Le fait de traiter un à un les nœuds larges réduit considérablement la consommation mémoire globales sur le GPU. En effet, une fois traité, le nœud est libéré de l'espace mémoire GPU, et le nœud suivant disposera de toute la mémoire GPU. Le tri initial est efficacement réalisé par un tri de type clef/valeur sur des types primitifs (float/int), grâce à notre représentation compacte des événements. Le thread GPU reste actif tant que la pile de nœuds larges n'est pas vide. Le CPU de son côté est exploité à raison d'un thread par cœur, qu'il soit physique ou logique. Ce groupe de threads se met en attente d'entrées dans la pile des nœuds fins. Chaque entrée dans cette pile envoie un signal qui réveille un thread, s'ils ne sont pas tous déjà occupés. Le traitement des nœuds fins est réalisé selon la méthode en quatre étapes décrites par Wald et Havran dans [WH06]. Chaque fois qu'un thread CPU a achevé la construction d'un nœud, il vient dépiler un nouveau nœud si la pile de nœud fin en contient, sinon il se met en attente d'insertion dans la pile de nœuds fins. Lorsque le thread GPU est inactif, le thread CPU qui achève la construction du dernier nœud fin met un terme à la phase de construction. Cette approche est très flexible en terme de répartition des calculs : si aucun GPU n'est disponible, le CPU peut construire entièrement l'arbre en considérant tous les nœuds comme des nœuds fins. De la même manière, tous les nœuds peuvent être considérés comme des nœuds larges, ce qui aura pour effet de ne pas réveiller de thread CPU, et donc de construire entièrement l'arbre sur GPU. De plus, si un nœud large venait à ne pas tenir en mémoire GPU, il suffit de l'empiler dans la pile des nœuds fins pour que le CPU le traite. En effet, les nœuds, fins ou larges, partagent tous exactement la même structure mémoire.

Implémentation

Toute la partie GPU de notre algorithme de construction de KD-Tree est implémentée entièrement avec l'API NVIDIA CUDA [NVIb]. Le traitement des nœuds larges est découpé en une phase d'initialisation du nœud racine et un processus en quatre étapes de construction d'un nœud. Chaque étape a été optimisée de telle sorte qu'elle tire parti de la parallélisation massive d'un GPU. L'image 6.1 est une vue globale de notre algorithme.

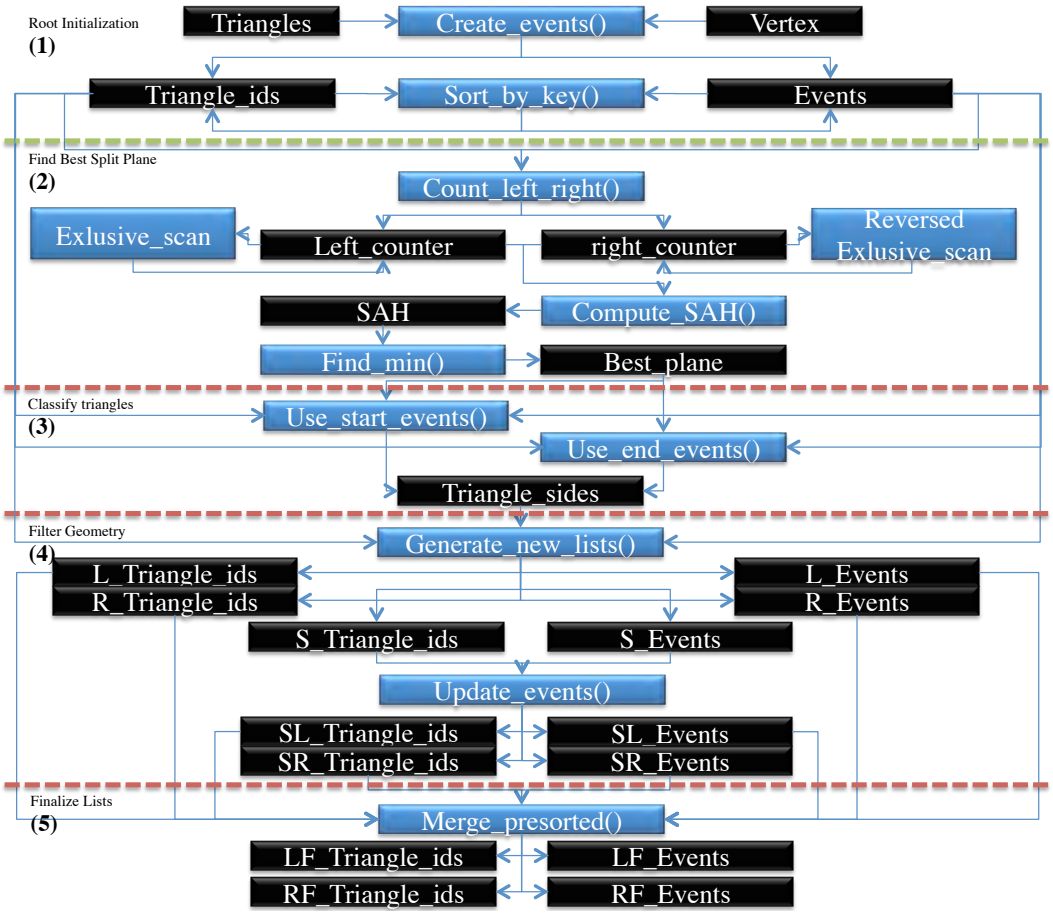


FIGURE 6.1 – Vue globale de notre algorithme de construction des nœuds large sur GPU. Les cellules bleues représentent les appels vers des primitives CUDA, les cellules noires représentent les différents tampons mémoires.

Initialisation du nœud racine (1)

Cette phase prend en entrée la liste des triangles présents dans la scène (figure 6.2). Un kernel CUDA est chargé, pour chacun des triangles de cette liste, de générer la liste des événements compacts qui le représentent. Pour cela, chaque thread CUDA charge un triangle et calcule sa boîte englobante (figure 6.3). Les types DEBUT/FIN de triangle se voient ensuite encodés dans les valeurs min/max de la boîte englobante pour chacun des axes. Ces événement compacts sont ensuite stockés dans une liste d'événements correspondant à leurs dimensions respectives. La dernière partie de l'initialisation consiste à effectuer le tri initial des listes d'événements sur chacun des axes (figure 6.4). Ce tri est effectué à l'aide du tri haute performance de type clefs/valeurs fourni dans la bibliothèque Thrust [HB10], désormais incluse dans le SDK CUDA. Les événements sont triés selon la valeurs flottante de leurs positions spatiales, qui représentent donc les clefs du tri, alors que les index des triangles sont les valeurs associées à ces clefs. Le nœud racine à construire

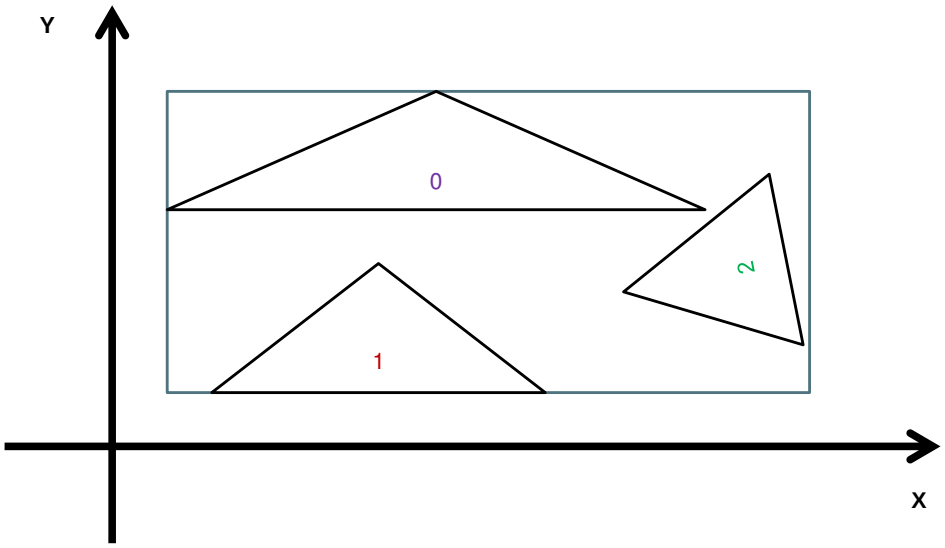


FIGURE 6.2 – Géométrie utilisée pour illustrer le déroulement de l'algorithme de construction du KD-Tree sur GPU.

peut désormais être créé grâce à la liste des événements triés, le nombre de triangles qu'il contient et la boîte englobante de la scène. Il est ensuite simplement empilé, dans la pile des nœuds larges ou fins selon le nombre de triangles qu'il contient, afin d'être traité par le périphérique le plus adapté.

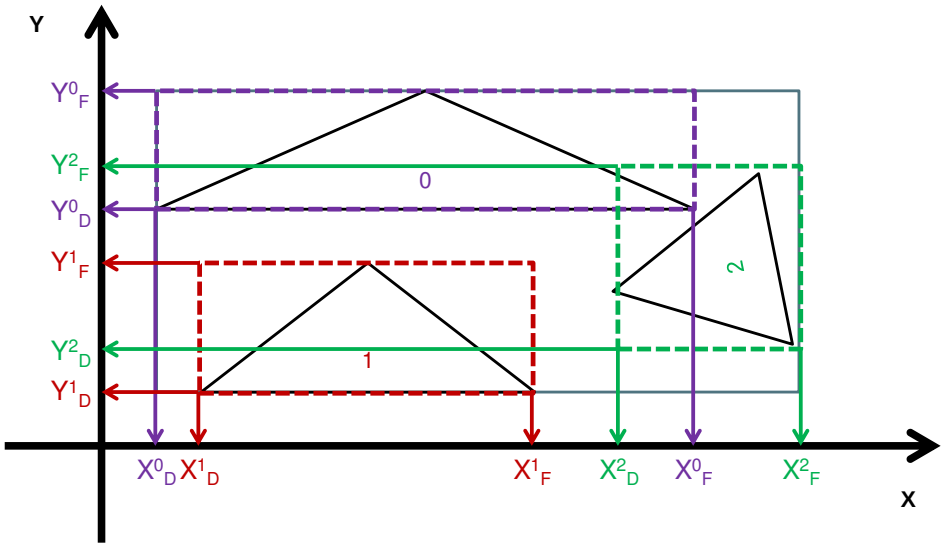


FIGURE 6.3 – Les boites englobantes de chacun des triangles de la scène sont calculés, afin de pouvoir générer des événements.

Events	X^0_D	X^0_F	X^1_D	X^1_F	X^2_D	X^2_F	Y^0_D	Y^0_F	Y^1_D	Y^1_F	Y^2_D	Y^2_F
Triangle	0	0	1	1	2	2	0	0	1	1	2	2

Sort_by_key()

Events	X^0_D	X^1_D	X^1_F	X^2_D	X^0_F	X^2_F	Y^1_D	Y^2_D	Y^1_F	Y^0_D	Y^2_F	Y^0_F
Triangle	0	1	1	2	0	2	1	2	1	0	2	0

FIGURE 6.4 – Les événements correspondants aux limites spatiales de chacun des triangles sont encodés avec notre représentation compact puis triés indépendamment sur chacun des axes.

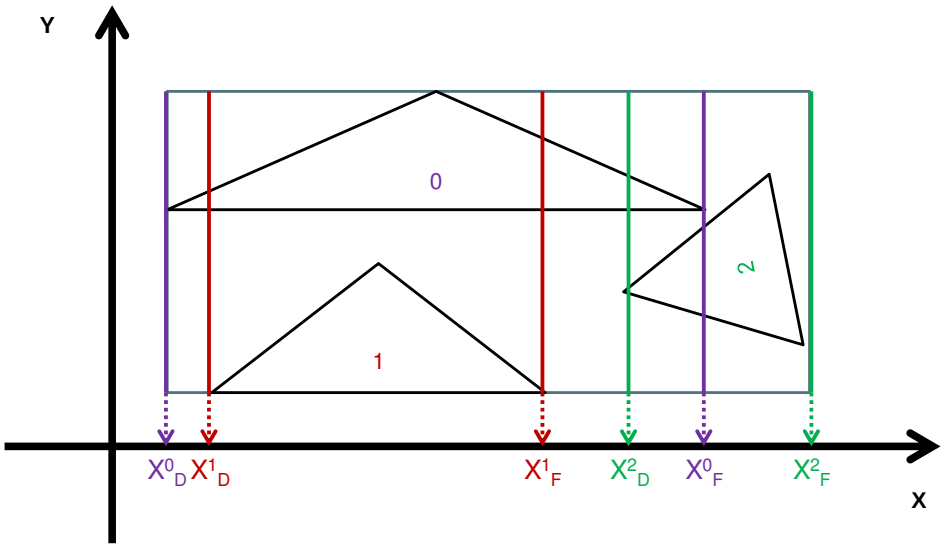


FIGURE 6.5 – Chacun des événements va être étudié en tant que candidat à la subdivision du nœud courant. Pour limiter la complexité de l'exemple, seuls les plans sur l'axe X seront considérés.

Sélection du meilleur plan de subdivision (2)

Chacun des événements présent dans le noeud large à construire va être étudié et servir de candidat à la subdivision spatiale (figure 6.5). Dès qu'un nœud large est disponible pour la construction, la première étape consiste à initialiser des compteurs de triangles de part et d'autre de chacun des plans candidats à la subdivision du nœud. Deux listes de compteurs sont nécessaires, une pour le nombre de triangles à gauche du plan, l'autre pour les triangles à droite du plan. Chacune des listes contient autant d'entrées que de plans candidats à évaluer. Un premier kernel CUDA est invoqué afin d'initialiser les compteurs de triangles associés à chaque événement contenu dans le nœud. Chaque thread CUDA charge un événement. Si l'événement est de type DEBUT de triangle, le compteur de triangle à gauche est initialisé avec 1 et le compteur de triangle à droite est initialisé avec 0. Inversement, si l'événement est de type FIN de triangle, le compteur de triangle à gauche est initialisé avec 0 et le compteur de triangle à droite est initialisé avec 1. Cette méthode d'initialisation des compteurs permet d'obtenir le nombre final de triangles à gauche et à droite de chacun des événements, et donc des plans candidats à la subdivision, en réalisant un "parallel exclusive scan" sur la liste de compteurs de triangles à gauche et un "reversed parallel exclusive scan" sur la liste de compteurs de triangles à droite (figure 6.6). L'évaluation de l'heuristique de subdivision peut désormais être réalisée en parallèle. Un nouveau kernel CUDA est invoqué pour réaliser cette évaluation. Chaque thread CUDA va réaliser le calcul de l'heuristique pour un événement. Pour cela, chaque

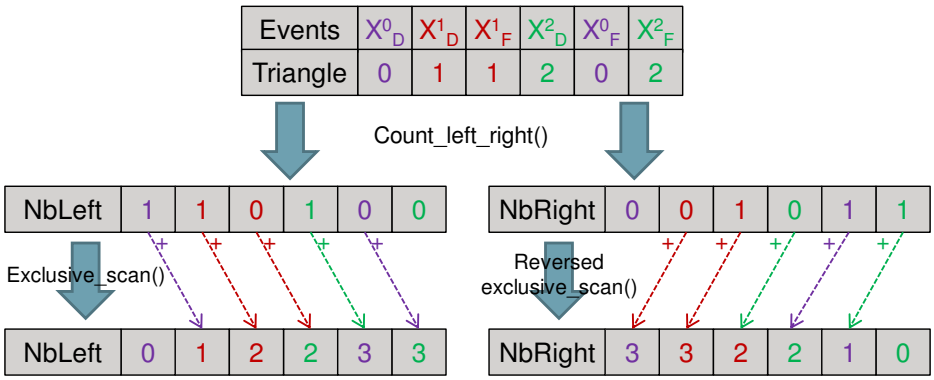


FIGURE 6.6 – Les événements de début de triangle (D) initialise les compteurs gauches à 1, et les événements de fin de triangle (F) initialise les compteurs droits à 1. Les scans exclusifs permettent d'obtenir par accumulation le nombre de triangles de chaque côté de chacun des événements.

thread a besoin de la boîte englobante du nœud courant, afin d'en déduire les boites englobantes des deux nœuds fils candidats. Une fois les aires surfaciques de chacune des boites englobantes calculées, l'estimation du coût de subdivision via l'heuristique SAH est évalué via la formule décrite par l'équation 4.1 et stocké dans une zone mémoire tampon. Finalement, le meilleur candidat est sélectionné suite à une recherche parallèle de valeur minimum sur GPU dans le tampon d'estimation de l'heuristique (figure 6.7). La création d'une feuille est aussi envisagée à cette étape. S'il s'avère que le coût estimé pour la création d'une feuille est inférieur à celui du meilleur candidat à la subdivision, la liste des triangles présents dans le nœud est extraite de la liste des événements, une feuille est insérée dans la structure finale et le traitement du nœud courant s'achève.

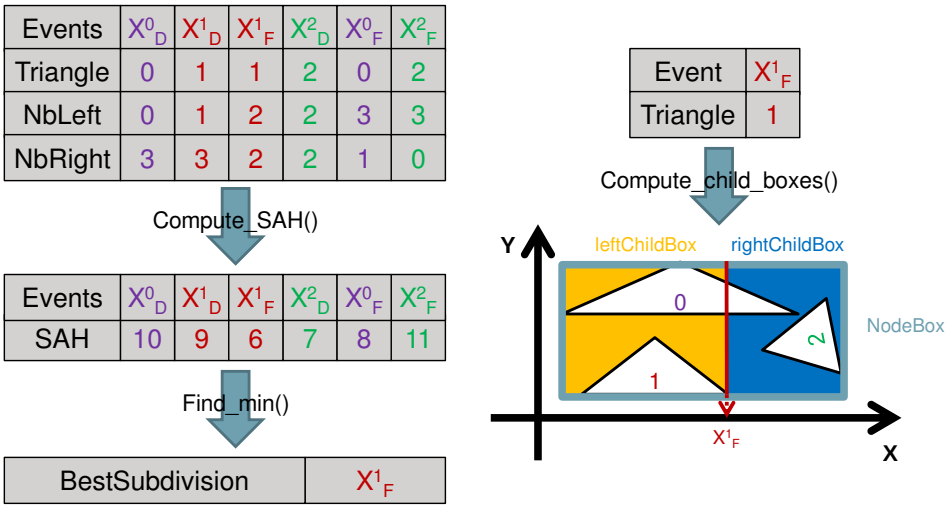


FIGURE 6.7 – Les données requise à l'évaluation de l'heuristique de subdivision SAH sont disponibles. Il suffit juste de calculer les boites englobantes de chacun des fils envisagés en coupant la boite du nœud courant avec l'événement évalué. Une fois les estimations de coût de subdivision calculées, une recherche de minimum est lancé afin d'identifier l'événement le mieux noté.

Classification des triangles (3)

Si un plan de subdivision a été sélectionné, il faut désormais répartir les triangles autour de celui-ci (figure 6.8). Pour cela, chacun des triangles va se voir attribuer la propriété GAUCHE, DROIT ou PARTAGE, symbolisant sa position par rapport au plan de subdivision. Ceci est réalisé par deux kernel CUDA distincts. Le premier ne traite que les événements de DEBUT de triangle. Pour chacun d'eux la position du triangle correspondant est initialisé avec GAUCHE ou DROITE selon la position de l'événement relativement au plan de subdivision. La seconde phase ne traite quant à elle que les événements de FIN de triangle situés à droite du plan de subdivision. Ainsi, les triangles associés à ces événements et précédemment classifiés à GAUCHE lors de la première passe sont réaffectés en PARTAGE (figure 6.9). Le découpage en deux kernels de cette phase s'explique par le fait qu'il est, à ce jour, impossible d'obtenir une synchronisation globale des threads CUDA. Il est donc impossible de garantir que les événements de type DEBUT de triangle ont bien été traités avant de travailler sur les événements de FIN de triangle en un seul Kernel. En sortie de cette phase, nous disposons d'une classification spatiale relative au plan de subdivision pour chacun des triangles présent dans le nœud.

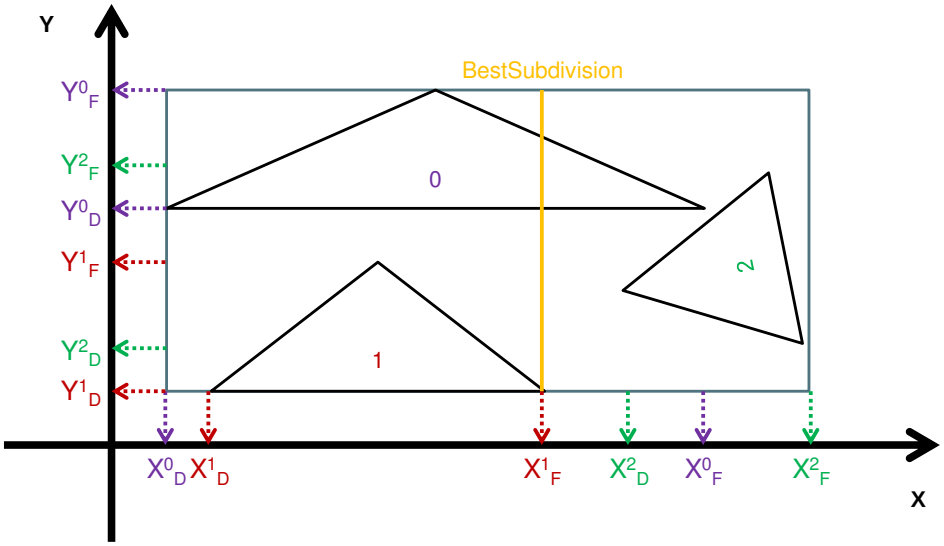


FIGURE 6.8 – Le plan destiné à la subdivision du nœud courant est désigné. Il faut maintenant déterminer quels triangles se situent strictement à gauche et à droite de ce plan, mais aussi ceux qui sont intersecté par le plan de subdivision.

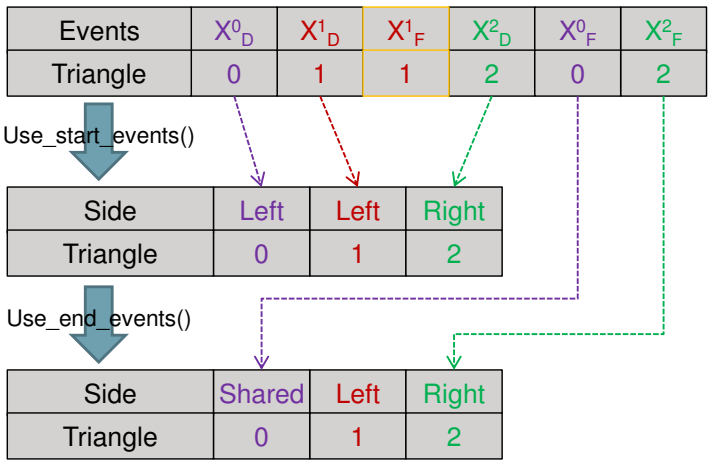


FIGURE 6.9 – Les événements de début de triangle permettent d'initialiser le côté du plan de subdivision auquel se situe chacun des triangles. L'étude des événements de fin de triangle permet de mettre à jour l'emplacement des triangles afin de détecter les triangles intersectés par le plan de subdivision.

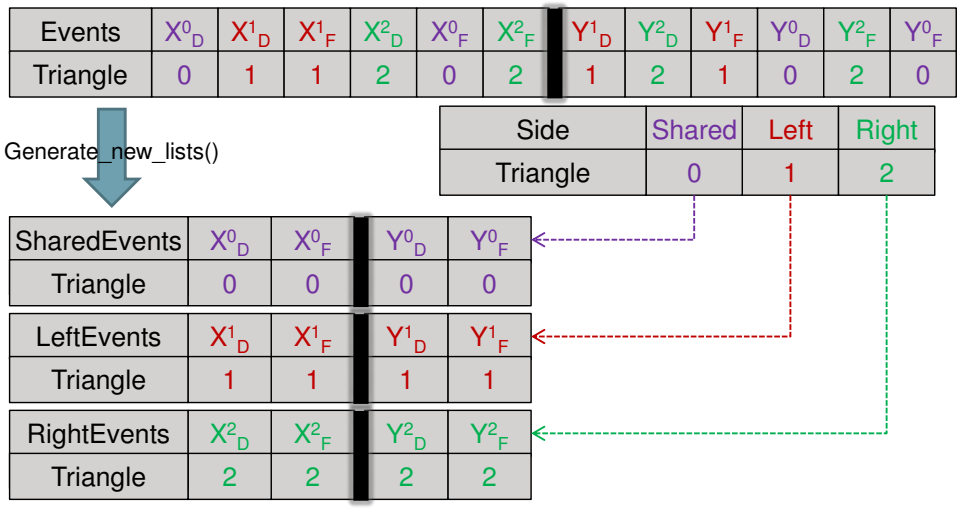


FIGURE 6.10 – La classification des triangles par localisation spatiale autour du plan de subdivision permet de générer directement trois listes d'événements distinctes. Deux listes représentent les événements destinés au fils gauche et droit qui seront réutilisés tels quel et sont maintenus triés, la troisième contient les événement partagés qui vont devoir être mis à jours pour chacun des fils.

Filtrage de la géométrie (4)

Cette étape va dans un premier temps générer les listes d'événements dont les triangles ont été strictement situés à gauche ou à droite du plan de subdivision par la phase précédente. Chacun de ces événements est en effet repris tel quel et recopié dans la liste, gauche ou droite, sans aucune modification et en maintenant l'ordre afin de préserver le tri initial. Pour les triangles partagés, la procédure est plus complexe. Ils sont dans un premier temps isolés(figure 6.10). Un kernel va s'occuper de générer pour chacun d'eux le nouvel événement correspondant pour chacun des côté du plan de subdivision. Pour la dimension sur laquelle le plan a été sélectionné, la mise à jour de l'événement est très simple. Il suffit en effet, de borner les valeurs des événements avec la valeur du plan de subdivision comme maximum pour le fils gauche, et comme minimum pour le fils droit. Sur les autres axes, il est nécessaire de déterminer l'intersection entre le triangle qui a généré l'événement et le plan de subdivision afin de déterminer ses nouvelles frontières (figure 6.11). Dans ce but, chacun des vertex est placé soit à gauche, soit à droite du plan via sa composante sur l'axe de subdivision. Cela permet d'isoler les arêtes du triangles qui intersectent le plan de subdivision. A partir des pentes de ces arêtes et des positions des sommets qui la composent, il est aisé d'obtenir la position de l'intersection entre le plan de subdivision et chacune des arêtes, et donc les nouveaux événements à utiliser. Le signe de la pente des arêtes permet de plus de déduire quelle paire d'événements attribuée au

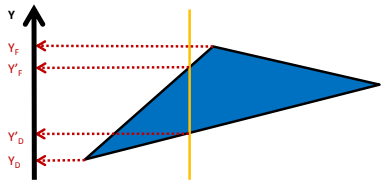
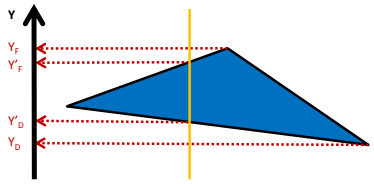
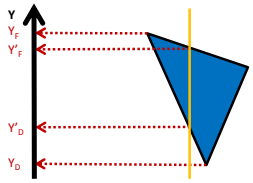
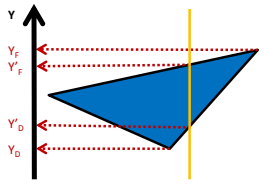
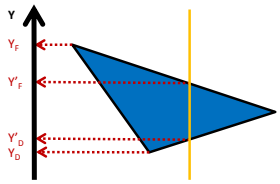
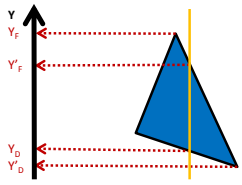
NBG/NBD/SP0/SP1	illustration	Evénements gauche/droit
1/2/+/+		$\{Y_D — Y'_F\}/\{Y'_D — Y_F\}$
1/2/+/-		$\{Y'_D — Y'_F\}/\{Y_D — Y_F\}$
1/2/-/-		$\{Y'_D — Y_F\}/\{Y_D — Y'_F\}$
2/1/+/+		$\{Y_D — Y'_F\}/\{Y'_D — Y_F\}$
2/1/+/-		$\{Y_D — Y_F\}/\{Y'_D — Y'_F\}$
2/1/-/-		$\{Y'_D — Y_F\}/\{Y_D — Y'_F\}$

TABLE 6.2 – Procédure de répartition des événements d'un triangle intersecté par un plan de subdivision. Fourni la répartition des anciens événements (Y_D et Y_F) et des nouveaux événements (Y'_D et Y'_F) entre les fils gauche et droit en fonction du nombre de sommets à gauche du plan (NBG), du nombre de sommets à droite du plan (NBD) et des signes des pentes des arêtes intersectés (SP0 et SP1).

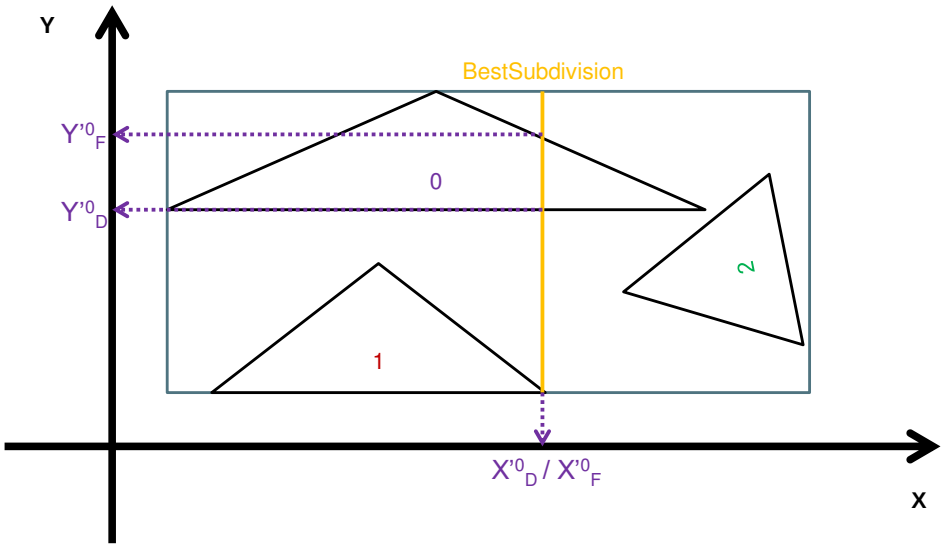


FIGURE 6.11 – Le plan de subdivision intersecte les triangles partagés, ce qui a pour effet de générer deux nouveaux événements par triangle et par axes qu'il va falloir répartir correctement entre les fils gauche et droit.

fils gauche et au fils droit parmi les deux anciens événements et les deux nouveaux fraichement générés. Le comportement a adopter est décrit dans le tableau 6.2 et ne dépend que du nombre de sommets du triangle de part et d'autre du plan de subdivision et du signe de la pente des arêtes du triangle intersectant le plan de subdivision. A la fin de cette passe, les événements sont classifiés en quatre zones mémoires (figure 6.11 et 6.12) :

- les événements des triangles strictement situés à gauche du plan de subdivision
- les événements des triangles strictement situés à droite du plan de subdivision
- les événements des triangles intersectant le plan de subdivision mis à jours pour le fils gauche
- les événements des triangles intersectant le plan de subdivision mis à jours pour le fils droit

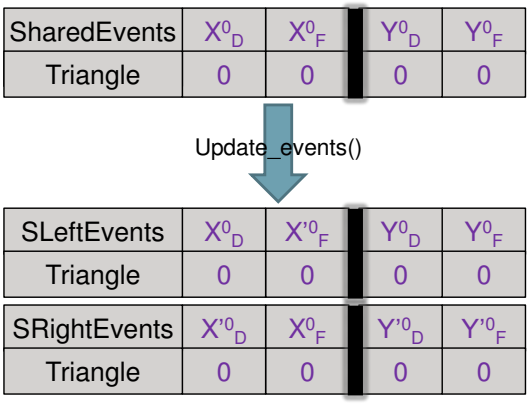


FIGURE 6.12 – Les événements partagés sont utilisés pour générer les nouveaux événements sur chacun des axes, issues de l'intersection entre le plan de subdivision et le triangle. L'affectation des nouveaux événements au sein des fils gauche et droit est géré grâce à la méthode décrite dans le tableau 6.2.

Finalisation des listes (5)

Cette dernière étape a pour seul objectif de fusionner les listes de triangles strictement d'un côté du plan de subdivision avec celle des événements partagés mis à jour pour ce côté. Pour atteindre cet objectif, la liste des événements partagés est tout d'abord trié, car la mise à jour a potentiellement détérioré l'ordre des événements. Nous appliquons ensuite un opérateur de fusion de listes triées fourni encore une fois par la bibliothèque Thrust (figure 6.13). Une fois ces listes fusionnées, les fils gauche et droit sont créés et empilés dans la pile des nœuds fins ou larges selon le nombre de triangles qu'ils contiennent.

FIGURE 6.13 – Tous les événements pour chacun des nœuds fils sont disponibles. Il ne reste plus qu'à fusionner les événements gauches/droits déjà triés et les événements partagés dans une seule liste tout en maintenant le tri sur chacun des axes.

6.2 Procédure de tests

Afin de tester, expérimenter et chiffrer nos différentes recherches, nous avons mis en place une collection de modèles et de points de vue qui seront utilisés tout au long de nos expérimentations sur le KD-Tree. Des rendus ainsi que des informations complémentaires sur chacune de ces scènes sont disponibles dans l'image 6.14 et le tableau 6.3.

Modèle	nombre de triangles	propriétaire
Bunny	69666	Stanford 3D Scanning Repository
fairy Forest	174117	Ingo Wald
Dragon	896816	Stanford 3D Scanning Repository
Happy	1087188	Stanford 3D Scanning Repository
Soda Hall	2156683	UC Berkeley Walkthrough Group
Hairball	2850000	Samuli Laine and Tero Karras, NVIDIA Research
Rungholt	6704264	Kescha
San-Miguel	7852595	Guillermo M. Leal Llaguno, Evolucién Visual

TABLE 6.3 – Informations complémentaires sur les modèles utilisés pour nos expérimentations. Chacun de ces modèle à travers ces propriétés permet de mettre à l'épreuve notre capacité à gérer efficacement tout type de géométries. Bunny, Dragon et Happy sont des modèles de complexité géométriques croissante permettant surtout de se comparer avec la communauté de l'informatique graphique du fait de leur utilisation universelle. Fairy Forest est un modèle contenant peu de triangles, des objets de différentes échelles ainsi que des textures et de la transparence. Soda Hall est un modèle conséquent ayant une géométrie globale alignée aux axes mais contenant un certain nombre d'objets de moindre échelle non-alignés aux axes. Hairball est un modèle très complexe géométriquement, à la fois très dense et courbé, et contenant des triangles effilés. Rungholt est un modèle massif ne contenant que des triangles alignés aux axes mais beaucoup de textures et de transparences. Quant à San-Miguel, il regroupe énormément de propriétés dans un seul modèle : massif, différentes échelles entre les objets, différentes densités de géométries, textures, transparences. Merci aux différents propriétaires pour leur partage.

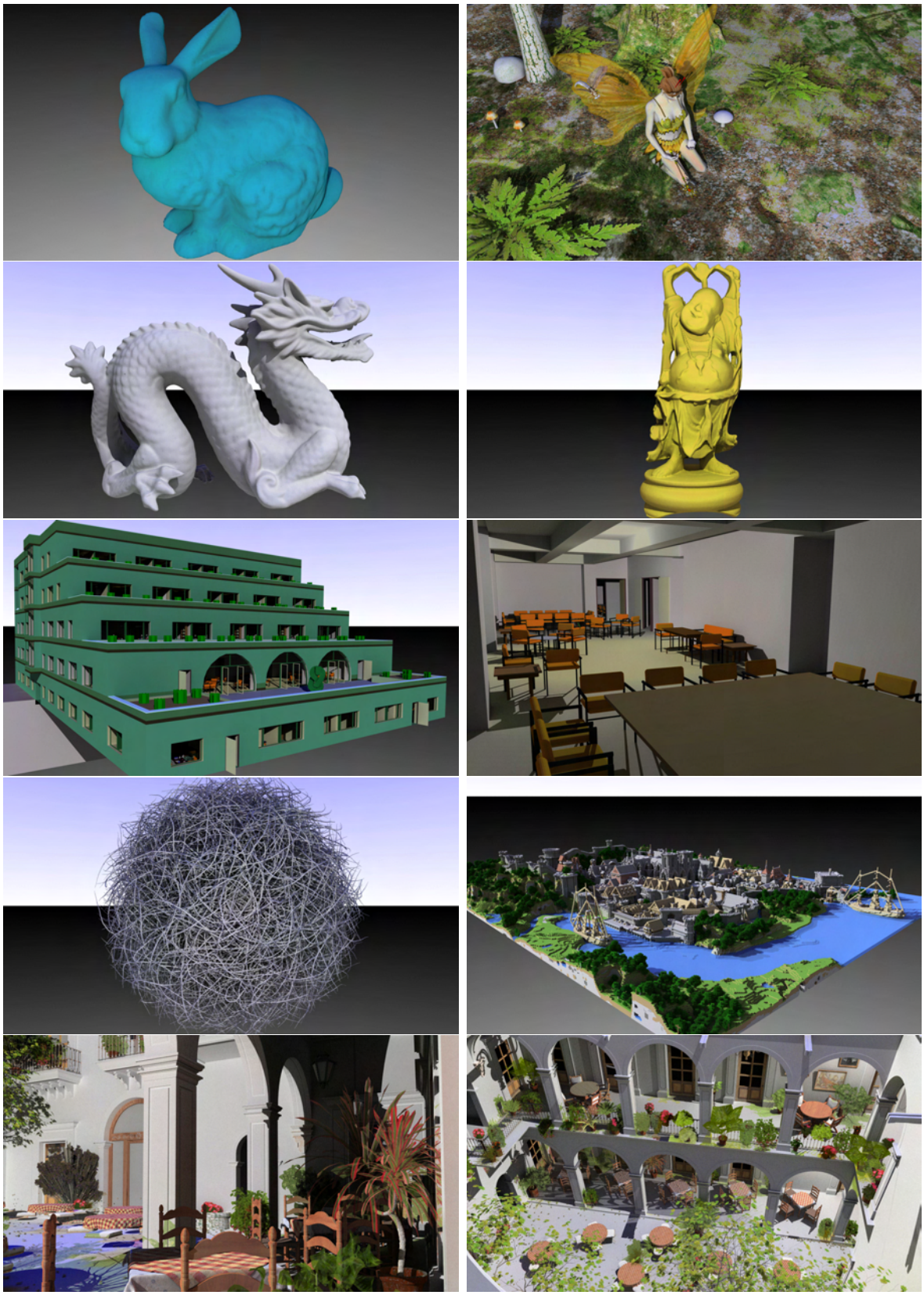


FIGURE 6.14 – Différents modèles et points de vue utilisés lors de nos expérimentations. Chacune de ces scènes contient une lumière ponctuelle, sauf Soda Hall qui en contient deux et San-Miguel qui en contient trois.

6.2.1 Résultats et discussions

La représentation des événements proposée est un point central de notre algorithme de construction du KD-Tree en terme de performances. Son influence sur les tris, illustrée dans les graphiques 6.15 et 6.16, ainsi que sa compacité mémoire accrue lui permettent de simplifier et accélérer les calculs lors de la construction du KD-Tree. Le tableau 6.4 permet d'illustrer l'impact de l'utilisation de cette représentation en se comparant à une implémentation haute performance utilisant une représentation classique des événements. Celle-ci nous permet de surpasser les précédents résultats de l'ordre de 30% à 50% à configuration égale.
De plus, comme illustré par le tableau 6.5, notre mode hybride CPU/GPU permet un gain supplémentaire de l'ordre de 50% sur les temps de construction obtenus en CPU seulement.
Enfin, le tableau 6.6 met en avant l'adaptabilité de notre algorithme au matériel en le comparant au tableau 6.5. En effet, une augmentation de l'ordre de 35% des performances du CPU seul allié à une augmentation de l'ordre de 30% des performances du GPU seul résulte en une augmentation de l'ordre de 32% des performances en mode hybride CPU/GPU sans changer aucun paramètre entre les deux configurations.

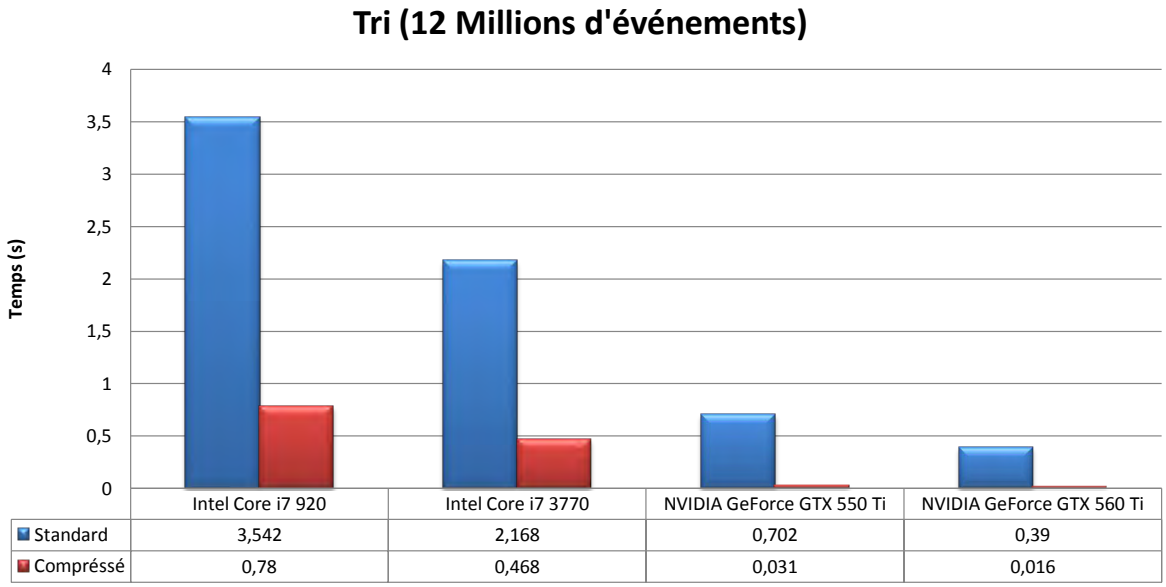


FIGURE 6.15 – Temps de tri pour 12 millions d'événements aléatoires standards et compressés avec notre méthode, sur différents périphériques.

Le graphe 6.17 expose l'évolution des temps de construction moyens de notre KD-Tree en fonction de l'utilisation de différentes ressources. Celui-ci permet de constater que l'ajout d'une unité de calcul produit systématiquement un gain en terme de temps

	[CKL+10]		notre algorithme	
Modèle	1-thread CPU	8-threads CPU	1-thread CPU	8-threads CPU
Dragon	2.65	1.48	1.40	0.70
Happy	3.22	1.86	1.70	0.84
Soda Hall	5.98	3.43	4.00	2.34

TABLE 6.4 – Temps de construction du KD-Tree obtenus pour différents modèles avec notre algorithme et celui issu des travaux de Choi proposé dans [CKL+10] dont le code source est disponible sur le dépôt [CKL+], CPU : Intel core I7-920

Modèle	1-thread CPU	8-threads CPU	GPU	Hybride CPU-GPU
Dragon	1.40	0.70	2.10	0.43
Happy	1.70	0.84	2.27	0.51
Soda Hall	4.00	2.34	2.24	1.13

TABLE 6.5 – Temps de construction du KD-Tree avec notre algorithme pour différents modèles, CPU : Intel core I7-920, GPU : NVidia GeForce GTX 550 TI

Modèle	1-thread CPU	8-threads CPU	GPU	Hybride CPU-GPU
Dragon	0.87	0.43	1.43	0.29
Happy	1.06	0.53	1.54	0.34
Soda Hall	2.54	1.51	1.56	0.76

TABLE 6.6 – Temps de construction du KD-Tree avec notre algorithme pour différents modèles, CPU : Intel core I7-3770, GPU : NVidia GeForce GTX 560 TI

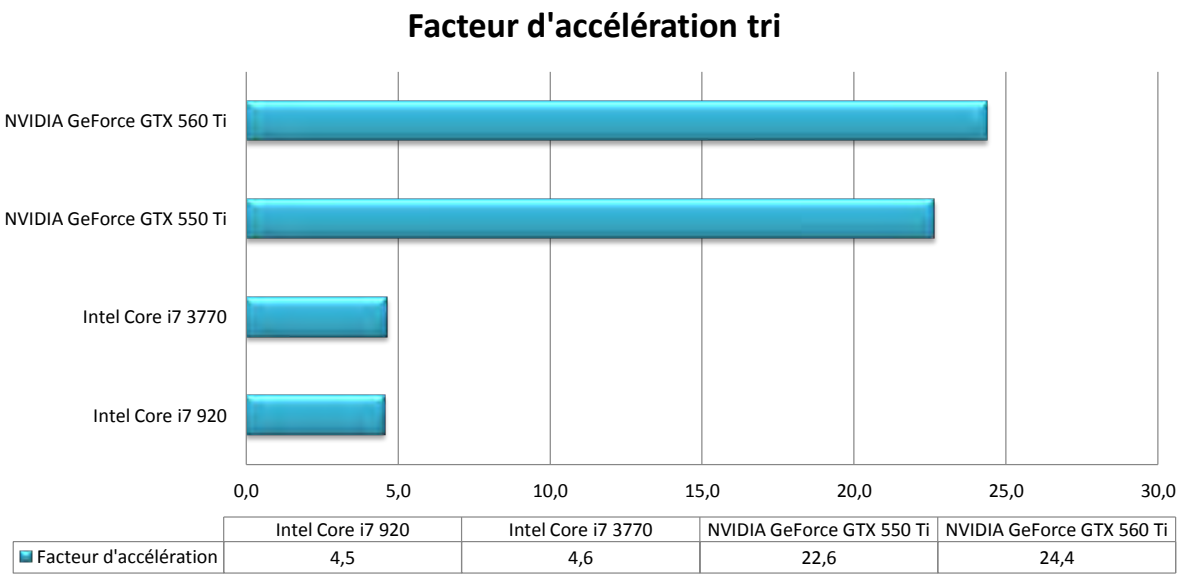


FIGURE 6.16 – Facteurs d'accélération des tris sur différents périphériques.

de calcul. Il permet de plus de tirer quelques conclusions :

- Les cœurs logiques de nos processeurs Intel apportent très peu de performances supplémentaires. Ceci est toutefois aussi le cas dans la publication d'Intel [CKL+10]. Ces derniers ayant choisis de ne finalement pas utiliser les cœurs logiques dans leurs expérimentations, cela ne semble donc pas être lié à notre implémentation.

- La méthode de construction GPU uniquement est plutôt lente, de l'ordre du temps de construction pour un seul thread CPU, car la méthode de traitement des nœuds par le GPU uniquement n'est pas du tout optimale dès lors que nous disposons d'un grand nombre de nœuds contenant très peu de triangles.

Comme indiqué sur le graphe 6.18, l'activation des quatre cœurs de nos différents CPU permet de réduire le temps de construction de moitié. Cette limite s'explique par le fait que le traitement des premier nœuds larges sur CPU n'exploite aucun parallélisme actuellement. L'adjonction d'un GPU pour le traitement des nœuds large permet de minimiser cette attente de parallélisme sur CPU et parvient à réduire encore de moitié le temps de construction.

Afin d'aller plus loin, plusieurs axes méritent notre attention. La fixation du seuil de classifications des nœuds en larges ou fins en fait parti. Une mauvaise valeur pour ce seuil a pour effet d'accroître :

- Le temps d'attente du CPU avant l'arrivée du premier nœud fin.

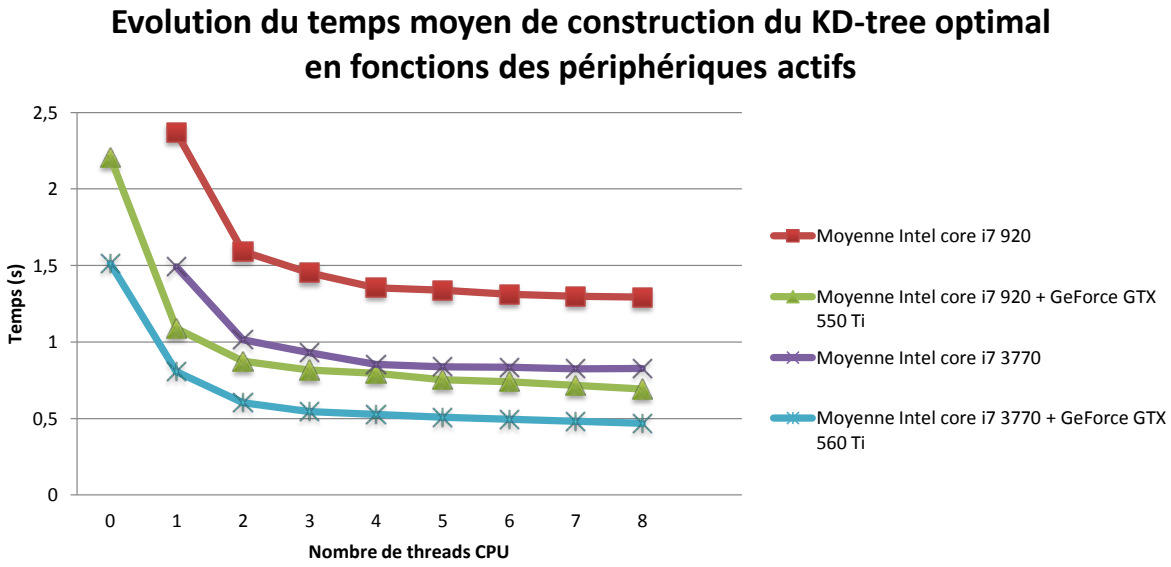


FIGURE 6.17 – Moyenne des temps de construction du KD-Tree sur nos scènes de tests, en fonction des périphériques et du nombre de threads CPU actifs.

- Le temps d'attente du GPU avant que le CPU n'aie terminé le traitement des nœuds fins.

Nous pensons que le meilleur moyen de résoudre à la fois

- la barrière du passage de relais CPU/GPU entre les nœuds fins et les nœuds larges
- le manque de parallélisme en mode CPU uniquement dans les premières étapes de construction
- la difficulté de définition du seuil de classification des nœuds en larges ou fins

consiste à pousser encore plus loin le côté hybride de la construction. Pour cela la solution qui semble s'imposer consiste à ajouter une méthode de traitement des nœuds larges pour le CPU, à la manière de l'implémentation de Choi décrite dans [CKL+10], ainsi qu'un traitement parallèle par paquets des nœuds fins par le GPU.

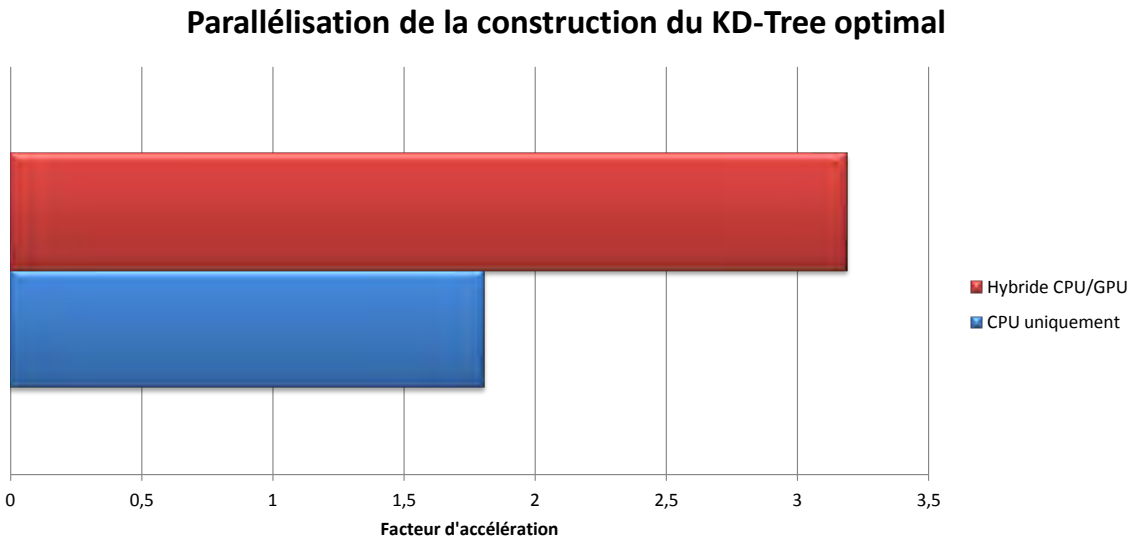


FIGURE 6.18 – Facteurs d'accélération moyen du temps de construction du KD-Tree en parallèle en mode CPU uniquement et en mode hybride CPU/GPU.

6.3 Traversée

La traversée d'un arbre binaire de subdivision spatiale est un domaine déjà fortement étudié que ce soit sur CPU par Havran et Wald avec [Hav00] et [Wal04], ou encore sur GPU par Foley et Aila avec [FS05] et [AL09]. Notre problématique se situe au niveau de l'hybridation de cette méthode de traversée. Nous nous reposons directement sur les algorithmes de traversée décrits dans ces études afin d'en tirer une technique de traversée performante, que ce soit sur CPU ou GPU, permettant de tirer partie des ressources de chacun des périphériques. Nous avons ensuite expérimenté plusieurs méthodes d'intersections rayon/triangle et proposons une organisation permettant d'augmenter les performances pour un surcoût mémoire quasi-nul.

6.3.1 Contributions

Nos contributions s'articulent autour de deux axes. Le premier consiste à définir une méthode de traversée efficace pour toutes les unités de calculs, CPU ou GPU. Pour cela nous avons réalisé une étude des performances pour plusieurs méthodes de traversées, avec différents tests d'intersections et enfin différentes organisations mémoires. Cette étude nous a permis de proposer une nouvelle organisation mémoire pour les données d'intersection rayon/triangle répondant à nos critères en terme de performance et de compacité mémoire.

Notre seconde contribution apporte le côté hybride de notre application de lancer de rayons. En effet, elle permet de superviser la collaboration entre les unités de calcul afin de distribuer efficacement et dynamiquement les rayons à lancer.

Étude des performances de la traversée

Sur CPU, la méthode de traversée de Wald avec pile a déjà prouvé son efficacité et est massivement utilisée [Wal04]. Nous avons donc sélectionné cette méthode et l'avons opposée à la méthode sans pile développée spécialement pour les GPU décrite par Foley dans [FS05]. La question de l'utilisation d'une pile de traversée sur GPU se pose du fait du caractère statique des allocations mémoires. En effet, chaque fil d'exécution doit posséder une pile de la même profondeur que le KD-Tree, car il est impossible de faire varier sa taille durant l'exécution. De plus, cette pile consomme trop de mémoire pour pouvoir être placée en mémoire partagée par un block de calcul. Elle doit donc être placée dans la mémoire la plus lente : la mémoire globale. Nous avons donc implanté ces deux méthodes, à la fois sur CPU et sur GPU, afin de mesurer leurs efficacités respectives sur chacune des unités de calcul. Comme l'on peut le constater dans le tableau 6.19, la méthode de traversée avec pile supplante la version sans pile dans tous nos tests. Avec une moyenne de 25% de performances supplémentaire, nous avons naturellement sélectionné cette technique de traversée, que ce soit sur CPU ou GPU.

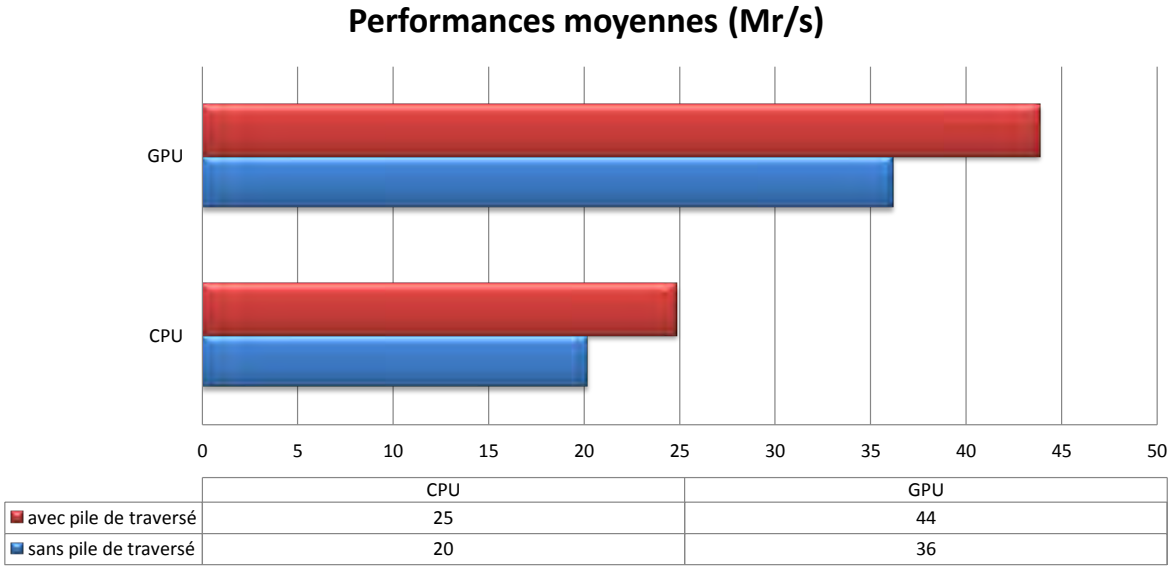


FIGURE 6.19 – Moyenne des performances de la traversée sur nos scènes de tests, en fonction de l'utilisation ou non d'une pile de traversée sur CPU et GPU.

Organisation des données d'intersection rayon/triangle

Afin de pouvoir insérer notre bibliothèque de lancer de rayons de manière non intrusive, nous n'imposons pas de représentation mémoire particulière pour les triangles en entrée. L'utilisateur fournit une méthode retournant les trois sommets du triangle et nous les convertissons directement dans notre format de données d'intersections afin de les stocker de manière contiguë en mémoire. Nous avons donc choisi d'utiliser une méthode d'intersection avec pré-calcul de données. Le surcoût mémoire étant déjà requis pour maîtriser la forme et l'organisation mémoire des données, autant profiter de cette étape pour augmenter les performances.
Une feuille du KD-Tree contient en général une liste d'indices pointant les triangles à intersecter. Afin de chiffrer le coût de cette indirection, nous avons expérimenté une version où chacune des feuilles contient directement les données d'intersection en lieu et place des indices des triangles à intersecter. Cette organisation augmente la consommation mémoire mais permet aussi de construire des paquets de triangles dans les feuilles. Nous avons donc implémenté quatre versions pour chacune des méthodes d'intersection décrites plus haut 2.1.1, 2.1.2 et 2.1.3 :

- une méthode indirecte, les feuilles du KD-Tree contiennent une liste d'index de triangles à intersecter, pointant vers les données d'intersection rayon/triangle associé de manière unique à chacun des triangles.

- une méthode directe, les feuilles du KD-Tree contiennent des données d'intersections rayon/triangle, potentiellement dupliqués lorsque plusieurs feuilles partagent le même triangle.

- une méthode directe SIMD utilisant les instructions SSE2, les feuilles du KD-Tree contiennent des données d'intersections rayon/triangles par paquets de quatre triangles à la fois.

- une méthode directe SIMD utilisant les instructions AVX, les feuilles du KD-Tree contiennent des données d'intersections rayon/triangles par paquets de huit triangles à la fois.

Nous avons, de plus, distingué deux versions pour la méthode d'intersection de Möller-Trumbore (2.1.1) afin de chiffrer la pertinence du pré-calcul du produit vectoriel entre les deux arêtes.
La méthode indirecte ne permet pas de grouper les triangles par paquets afin de bénéficier des instructions SIMD. En effet, les triangles étant potentiellement partagés par plusieurs feuilles, il est impossible d'extraire des groupement en paquets efficaces sans les dupliquer.

Or l'indirection mémoire via l'index du triangle est justement l'atout de cette méthode car il permet de ne pas dupliquer les données d'intersection rayon/triangle. Une autre approche pour bénéficier des instructions SIMD consisterait à grouper les rayons par paquets pour intersecter les triangles uns à uns, mais cela requiert une cohérence des rayons pour être efficace. Nous avons donc écarté cette approche.
Les instructions SSE2 et AVX, documentées ici [INTb], n'étant pas supportées par les GPU, nous avons implémenté une surcouche aux types SIMD. Celle-ci utilise directement les intrinsics SIMD [INTc] côté CPU et émule le comportement de ces intrinsics pour le GPU.

Performances brutes

Comme indiqué dans le graphique 6.20, la suppression de l'indirection pour l'accès au triangle augmente les performances, entre 10% et 30% pour toutes les versions du test d'intersection, que ce soit pour le CPU ou le GPU. Les méthodes utilisant les instructions SIMD ont un apport significatif pour le CPU, jusqu'à 450% pour la version SSE2 et 660% pour la version AVX. Les versions SIMD n'apportent rien en terme de performances pour le GPU, mais le léger surcoût dû à l'émulation des intrinsics SIMD est amorti par le bénéfice de la suppression de l'indirection mémoire pour l'accès aux données d'intersections. Il est à noter que la méthode d'intersection de Möller-Trumbore (2.1.1), en retrait en terme de performance sur CPU pour les versions indirecte et directe, se révèle particulièrement efficace non seulement sur GPU, mais bénéficie énormément des instructions SIMD sur CPU. A tel point qu'elle se révèle être la version la plus performante que ce soit sur CPU ou GPU dès lors que les instructions SSE2 ou AVX sont activées. Pour ce qui est du pré-calcul du produit vectoriel entre les deux arêtes, son apport est situationnel. Si sur CPU ce pré-calcul peut apporter 10-20% de performances, excepté pour la version AVX, il se révèle pénalisant sur GPU. Ces pertes de performances s'expliquent par l'occupation mémoire supplémentaire provoquée par le stockage du produit vectoriel. Sur GPU l'accès mémoire se révèle plus coûteux que le calcul du produit scalaire, alors qu'en configuration AVX, la taille des données d'intersections provoque des défauts de caches à répétition fortement nuisibles aux performances.

Compacité mémoire

Le KD-Tree ayant pour particularité la possibilité d'indexation multiple d'un même triangle dans plusieurs feuilles, il est légitime de se préoccuper du surcoût mémoire impliqué par la suppression de l'indirection mémoire dans les feuilles. En effet, dupliquer

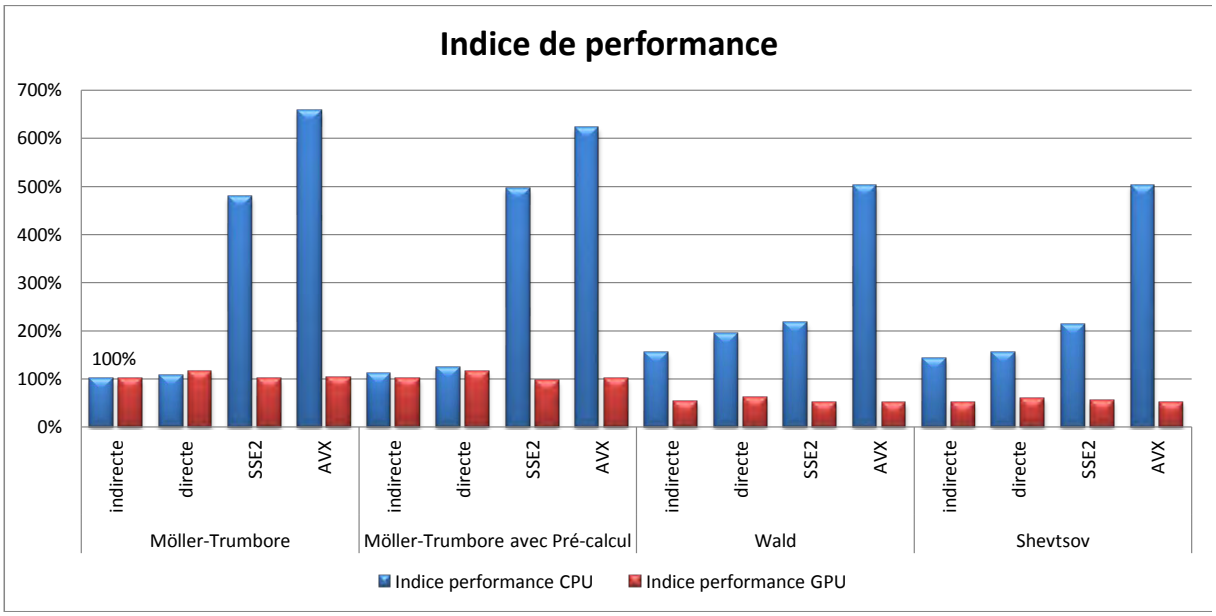


FIGURE 6.20 – Comparatif sur CPU et GPU des performances pour différentes méthodes d'intersection en utilisant différentes organisations mémoires. La valeur de référence (100%) est fixée par la méthode Moller-trumbore en accès indirect (plus la valeur est élevée, plus la méthode est efficace).

des données d'intersection implique des consommations mémoires bien supérieures à la duplications d'indices de triangle. Ces consommations mémoires sont exposées dans le graphique 6.21. Si la suppression de l'indirection mémoire pour l'accès au triangle se révélait payante au niveau performances, son surcoût en terme d'occupation mémoire s'avère difficilement acceptable. En effet, la consommation mémoire pour les données d'intersection se retrouve, en moyenne, doublée, voire triplée. Ce surcoût s'explique part :

- la duplication des données d'intersection, pour les triangles contenus dans plusieurs feuilles du KD-Tree.
- les données d'intersection SIMD partiellement remplis, en effet les variables SIMD ont une taille fixe regroupant quatre (SSE2) ou huit (AVX) données, or le nombre de triangles dans la feuille n'est pas forcement un multiple de cette taille.

Si ce surcoût mémoire peut se révéler contraignant côté CPU, il est très pénalisant côté GPU. En effet, si l'espace mémoire du GPU ne permet pas de recevoir l'ensemble du KD-Tree et des données d'intersection, celui-ci n'est plus éligible pour le lancer de rayons. Cela signifie qu'en l'état, utiliser les méthodes d'intersections directes conduit à une désactivation du GPU pour des modèles deux à trois fois moins imposants que pour la méthode indirecte.

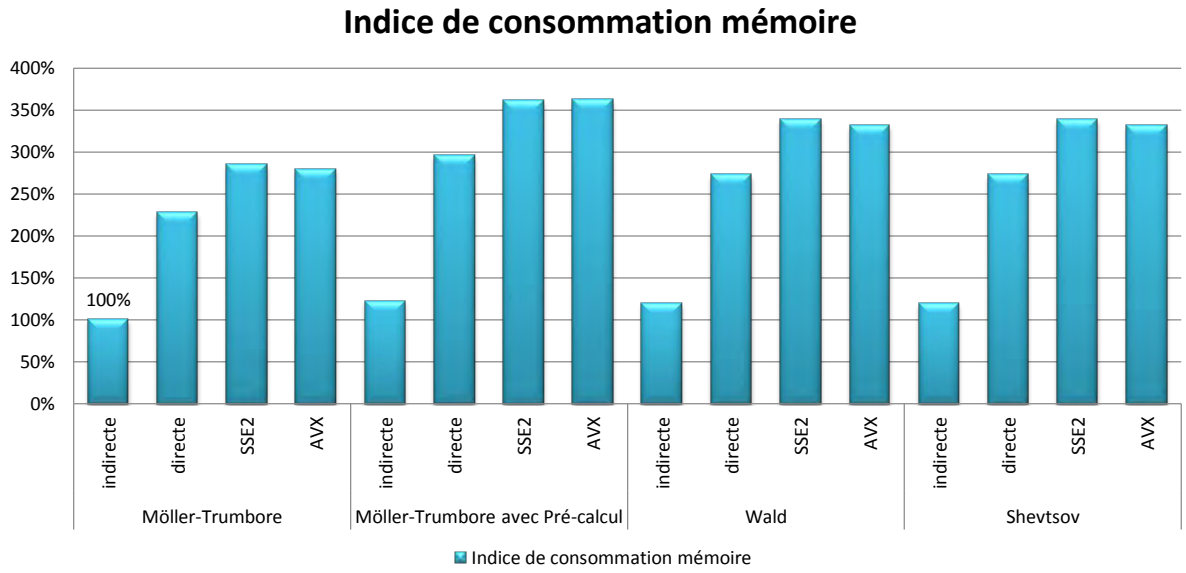


FIGURE 6.21 – Comparatif sur CPU et GPU de l'occupation mémoire pour différentes méthodes d'intersection en utilisant différentes organisations mémoires. La valeur de référence (100%) est fixée par la méthode Moller-trumbore en accès indirect (plus la valeur est élevée, plus la consommation mémoire est importante).

Organisation mixte

Afin de tendre vers les performances obtenues sans indirections mémoire pour l'accès au triangles, tout en contenant l'empreinte mémoire, nous proposons une organisation mixte des données d'intersection. Celle-ci repose sur le suivi de la répartition des triangles au sein des feuilles du KD-Tree. Si un triangle n'apparait que dans une seule feuille, alors ses données d'intersection sont éligibles à être directement copiées dans la feuille de l'arbre. Dans le cas inverse, la feuille ne contiendra qu'un indice d'indirection pour accéder au triangle. Concrètement, les feuilles contiennent à la fois des indices de triangles pointant des données d'intersection externes, ainsi que des données d'intersection directement résidentes dans la feuille. Comme décrit dans les graphes 6.22 et 6.23, cette organisation nous permet de conserver, en moyenne, 88% des gains de performances constatés en supprimant l'indirection mémoire, tout en limitant l'extension de l'empreinte mémoire à une moyenne de 10%. Les graphes 6.24 et 6.25 permettent d'illustrer l'efficacité de chacune des approches en fonction de leur consommation mémoire. Nos organisations mixtes nous permettent d'obtenir des facteurs de performances par occupation mémoire jusqu'à cinq fois supérieurs à leurs homologues non-mixtes sur CPU et jusqu'à deux fois supérieurs sur GPU. Ces résultats illustrent la qualité du compromis entre performances et compacité mémoire de nos approches mixtes.

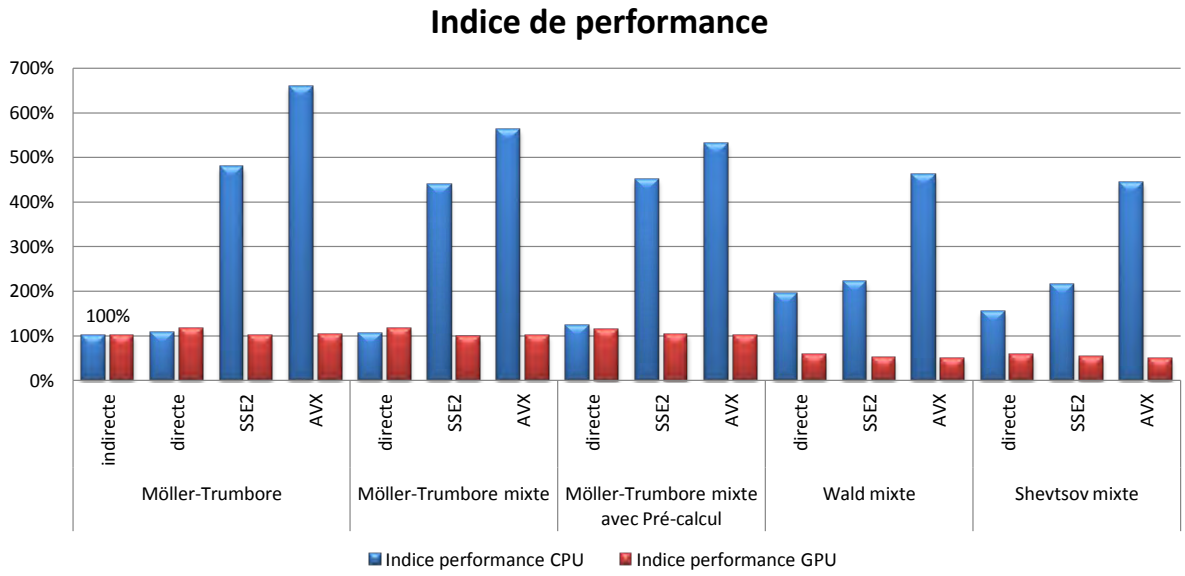


FIGURE 6.22 – Comparatif sur CPU et GPU des performances pour différentes méthodes d'intersection en utilisant notre organisation mémoire mixte. La valeur de référence (100%) est fixée par la méthode Moller-trumbore en accès indirect.

Robustesse

En marge des critères de performance et de compacité mémoire, nous avons rajouté un critère à notre sélection de la méthode d'intersection. Nous avons en effet déjà fait l'expérience de problèmes d'imprécision lorsque la représentation des nombres réels est poussée dans ses limites. Notre procédure de test inclut donc une scène, poussant les méthodes d'intersections aux limites des nombres flottants afin d'évaluer leur robustesse. Les images 6.26 générées à l'aide des trois méthodes d'intersections pour des scènes identiques sont sans appel. L'algorithme d'intersection de Möller-Trumbore s'en sort parfaitement bien lorsque les méthodes de Shevtsov et de Wald manquent des intersections, créant des trous dans la géométrie et donc des artefacts visuels.

Organisation finale

C'est naturellement la méthode mixte de Möller-Trumbore qui est actuellement utilisée par notre bibliothèque de lancer de rayons. Ses performances à la fois sur CPU et GPU, sa compacité mémoire, sa robustesse et enfin la simplicité de sa représentation mémoire en ont fait le choix le plus pertinent. En effet, la description géométrique des triangles est directement disponible sous la forme d'un sommet et deux arêtes. Cela apporte une souplesse non négligeable, afin de pouvoir réaliser des calculs sur ces triangles en interne sans faire appel aux données de l'application hôte.

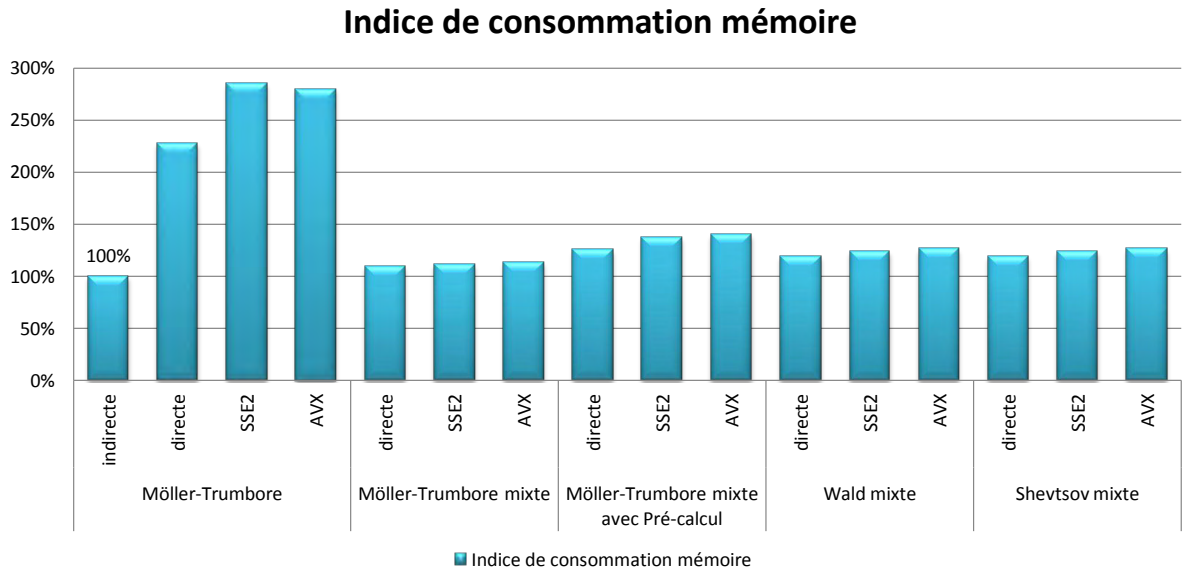


FIGURE 6.23 – Comparatif sur CPU et GPU de l'occupation mémoire pour différentes méthodes d'intersection en utilisant notre organisation mémoire mixte. La valeur de référence (100%) est fixée par la méthode Moller-trumbore en accès indirect.

Répartition des tâches de traversée entre périphériques

Afin de répartir les tâches de manière souple lors du lancer de rayons, le besoin d'un manager de périphériques de calculs s'est vite fait sentir. Celui-ci s'occupe dans un premier temps de scanner tous les périphériques disponibles sur la machine. Pour les CPU il va déterminer le nombre de threads idéal afin de coller au mieux au matériel, un par cœur physique ou logique généralement. Pour les GPU il va s'occuper de détecter ceux qui sont éligibles CUDA, leur créer un thread CPU chacun pour la gestion des transferts mémoires et les appels de kernel, et enfin leur affecter un identifiant qui dépend de leur classement en terme de puissance de calcul. Par défaut, le manager va activer toutes les ressources disponibles. Toutefois, l'utilisateur est libre d'aller restreindre les ressources à utiliser, en désactivant un certain nombre de threads CPU ou certains GPU.
Nous avons ensuite créé un manager de tâches de lancer de rayons. Celui-ci est client du manager de périphériques. En effet, il sollicite le manager de périphériques afin d'obtenir des ressources matérialisées par des threads, typés CPU ou GPU. Une fois ces ressources obtenues, le manager de tâches va lancer les différents threads de lancer de rayons, que nous appellerons par la suite des traceurs, sur la version du code de traversée qui est associé au type de périphérique du thread. Ce manager contient une structure qui peut

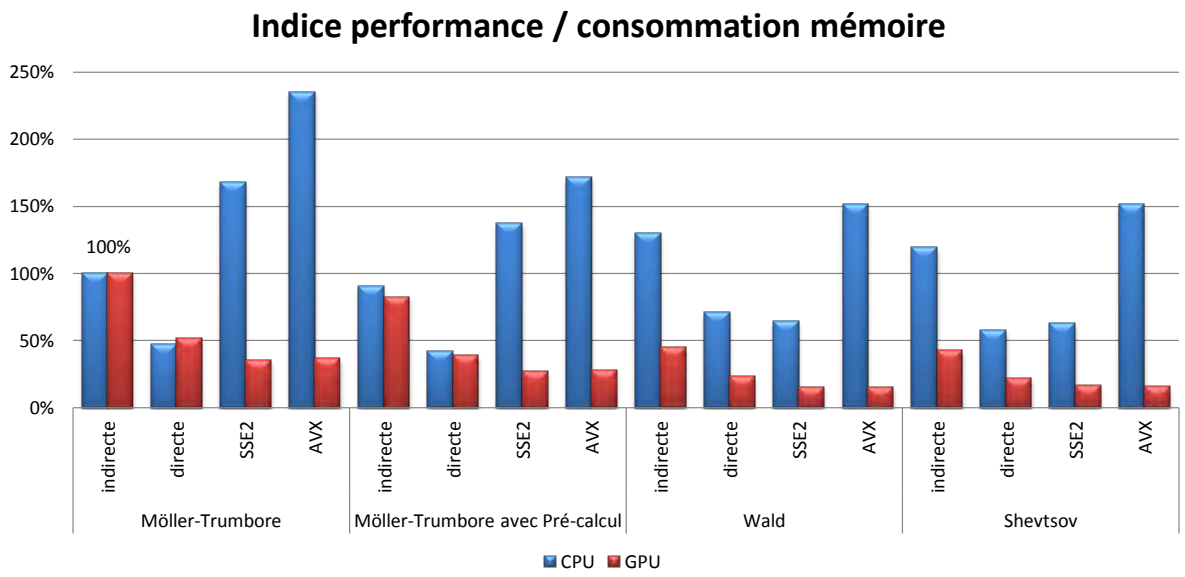


FIGURE 6.24 – Comparatif sur CPU et GPU des performances par rapport à leurs coûts mémoires pour différentes méthodes d'intersection en utilisant différentes organisations mémoires. La valeur de référence (100%) est fixée par la méthode Moller-trumbore en accès indirect.

être vue comme un compteur de rayons restant à traiter, accompagné d'un pointeur vers les données en entrée : les rayons à tracer, ainsi qu'un pointeur vers les données en sortie : les impacts associés. Chaque tracé a la possibilité de demander au manager l'obtention d'un certain nombre de rayons. Le manager accordera, en fonction du nombre de rayon restants, la totalité des rayons demandés, une partie seulement ou aucun si le compteur est vide. Lorsque le compteur de rayons restant tombe à zero, le manager de tâches se met dans un premier temps en attente des threads qui n'ont pas terminé leur tâches, puis se charge de rendre les ressources au manager de périphériques. Il est à noter que les threads CPU sont prioritaires lors de l'affectation de rayons à lancer. Cela permet de ne pas faire appel au GPU pour lancer quelques rayons isolés, dont le nombre n'est pas suffisant pour amortir le coup de lancement et des transferts mémoire d'une méthode sur GPU. Le premier appel à une tâche de traversée GPU déclenche la copie mémoire du KD-Tree accompagnée de ses données d'intersection sur le GPU. Durant cette copie, le CPU continue son activité et le GPU ne s'attribue pas de rayons à lancer. S'il s'avère que le GPU n'est pas en mesure de recevoir toutes les données nécessaires à l'exécution de la requête d'intersection, celui-ci est désactivé pour la tache courante.

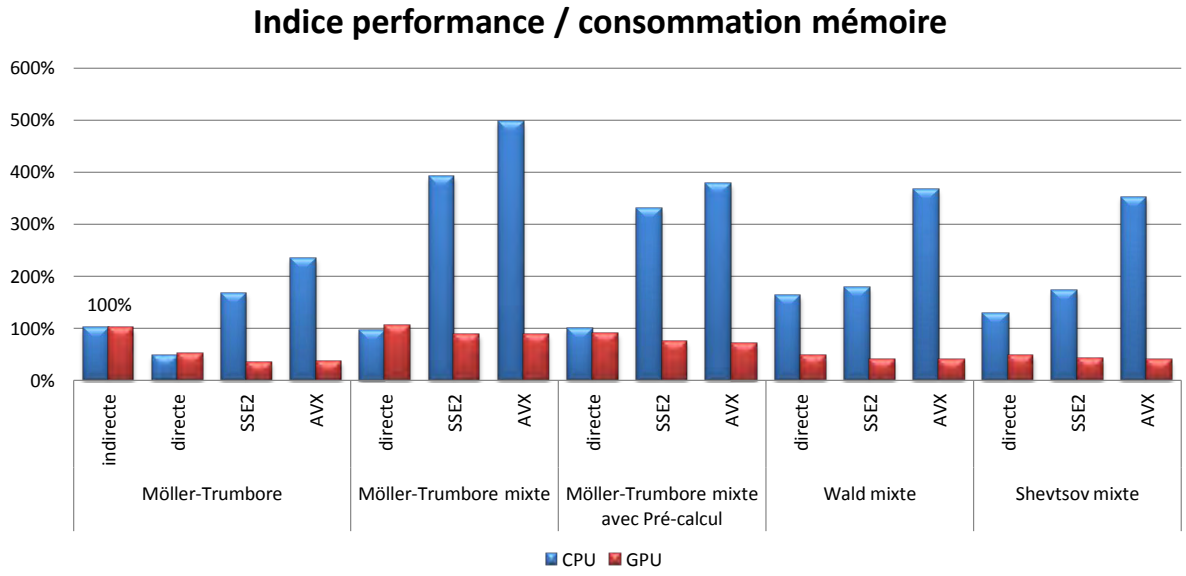
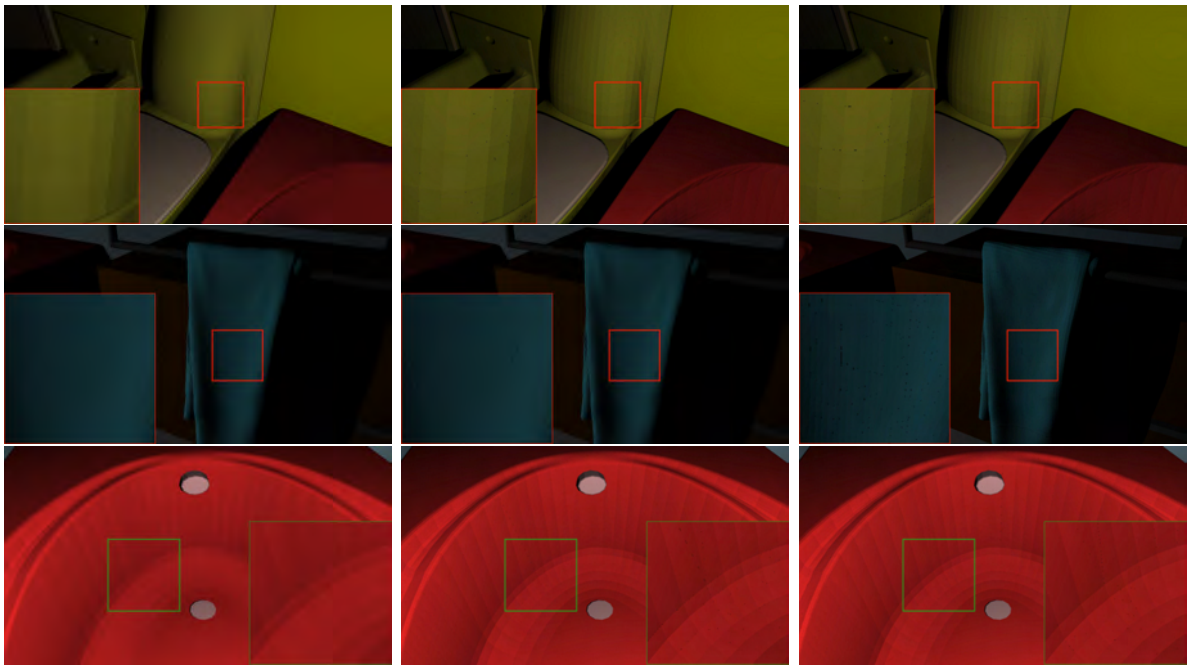


FIGURE 6.25 – Comparatif sur CPU et GPU des performances par rapport à leurs coûts mémoires pour différentes méthodes d'intersection en utilisant notre organisation mémoire mixte. La valeur de référence (100%) est fixée par la méthode Moller-trumbore en accès indirect.

Implémentation

La traversée en elle-même utilise directement l'implémentation décrite par Wald dans sa thèse [Wal04]. Quelques ajustements de codes ont étés effectué afin de pouvoir appeler la même méthode sur CPU et sur GPU. En effet, le GPU utilise une partie du travail d'implémentation réalisé par Aila dans [AL09], et en particulier les threads persistants et la traversée de type "while-while". C'est à dire que les rayons au sein d'une *warp* se synchronisent quand ils ont tous terminé leur descente dans le KD-Tree, puis lorsqu'ils ont terminé d'intersecter leurs triangles respectifs. Nous avons expérimentés diverses approches afin d'accélérer cette traversée, en particulier en permettant à une *warp* de recharger des rayons pour certains de ses threads seulement. Même en passant par un cache de rayons et d'impacts en mémoire partagée afin de garantir des accès coalescents en mémoire centrale, aucune de nos solutions ne s'est avérée payante.
Pour ce qui est de l'implémentation des différents tests d'intersection, nous utilisons les algorithmes décrits dans les publications sans modification majeures. Seules les versions SIMD ont nécessité une attention particulière. Surtout pour les méthodes de Shevtsov et Wald qui utilisent un axe majeur pour les triangles, pouvant varier au sein d'un paquet de triangles. Cet axe majeur implique aussi des divergences sur les données issues des rayons en entrée. Il a donc été nécessaire de mettre en place un mécanisme de masques afin de placer correctement les composantes majeures des rayons dans les registres SIMD. La



Möller-Trumbore Shevtsov Wald

FIGURE 6.26 – Images issues de la procédure de test de la robustesse des méthodes d'intersections. La caméra ainsi que la géométrie sont situées dans des plages de valeurs à la limite des représentations flottantes 32 bits des nombres réels. Les méthodes d'intersection de Wald et de Shevtsov présentent énormément d'artefacts (pixels noirs) alors que la méthode de Möller-Trumbore présente des résultats satisfaisants.

surcouche SIMD destiné au GPU simule le comportement exact des *intrinsics* SIMD, en particulier au niveau des opérations de comparaisons et d'opérateurs logiques. En effet, il est nécessaire qu'un masque SIMD généré par un périphérique soit utilisable correctement par tous les périphériques. Techniquement, cette surcouche fait appel à une fonctionnalité de l'API CUDA permettant de déclarer une méthode comme exécutable à la fois par le CPU et le GPU via les symboles "__host__" et "__device__". De plus, le symbole "__CUDA__ARCH__" permet de générer du code uniquement GPU et autorise donc des comportements divergents entre le CPU et le GPU au sein même de ces fonctions pourtant partagées par les deux entités.

Afin de suivre l'évolution des répartitions de triangles dans les feuilles, nous utilisons une fonctionnalité de l'API CUDA : la mémoire dite "zero-copie". Cette mémoire est allouée à l'aide de l'allocateur CUDA dans la mémoire centrale mais est directement mappée dans l'espace mémoire des GPU. Bien que très pratique et efficace, cette mémoire doit toutefois être utilisée avec parcimonie. En effet, celle-ci est allouée de manière à ne pas être paginable par le système d'exploitation. Chaque allocation consomme donc réellement et directement la mémoire de manière globale sur la machine hôte. Dans notre construction,

une liste de statuts de triangles est allouée en mémoire "zero-copie". Lorsqu'un triangle est partagé par un plan de subdivision durant une phase de construction d'un nœud du KD-Tree, le thread de construction, qu'il soit CPU ou GPU, accède à ce statut. Si le triangle a le statut non-partagé, le thread de construction met à jour ce statut et affecte un identifiant unique d'indirection à ce triangle. Le thread peut en effet modifier ce statut sans risque d'accès concurrent car si le triangle n'est pas déjà partagé, il est forcement le seul à le posséder dans sa phase de construction. Si au contraire le triangle est déjà partagé, il est inutile de réaliser la moindre action. Lors de la construction d'une feuille du KD-Tree, la liste de statuts nous permet de créer des paquets de données d'intersection pour les triangles non-partagés qu'elle contient et de les concaténer avec les indices uniques d'indirection affectés aux triangles partagés par plusieurs feuilles qu'elle contient aussi.
La répartition des tâches de traversée profite grandement de la mémoire "zero-copie". Elle nous permet en effet d'utiliser nos pointeurs vers les données en entrée/sortie tels quels pour tous les traceurs, que ceux-ci soient des traceurs CPU ou GPU. De plus, grâce à cette mémoire, les données sont automatiquement *streamées* que ce soit lors de la copie du CPU vers le GPU des rayons, ou lors du retour des impacts du GPU vers le CPU. La seule contrainte réside dans le fait que l'allocation des zones mémoire contenant les rayons et les impacts doit être maitrisé afin d'utiliser l'allocateur CUDA adéquat. C'est pourquoi l'interface de notre API de lancer de rayon requiert de passer par notre type de tampon mémoire de lancer de rayon, qui encapsule l'allocation de manière transparente.

6.3.2 Résultats et discussions

Notre implémentation de la traversée du KD-Tree, alliée à notre organisation des données de la méthode d'intersection rayon/triangle nous permettent d'atteindre de hautes performances.Ces performances sont illustrées par des comparaisons avec une application de référence en terme de performances dans le milieu de la recherche académique, à savoir EMBREE [INTa]. Comme l'indiquent nos mesures sur le graphe 6.27, nos performances CPU sont légèrement supérieures à EMBREE. Cette courte avance est toutefois largement confortée dès lors que nous utilisons notre capacité à faire collaborer plusieurs périphériques. D'autre part, notre répartition des tâches de traversée entre périphériques apporte une souplesse et une capacité d'adaptation à la fois aux ressources disponibles, mais aussi au type de requêtes utilisateur. Comme indiqué dans le graphique 6.28, chaque périphérique actif est utilisé et apporte un gain de performances. La seule contrainte à l'obtention des performances maximum se résume à allouer un thread CPU, physique ou logique, par GPU actif. En effet, activer toutes les unités du CPU ainsi que les GPU

provoque des pertes de performances importantes. Cela s'explique par le fait que même si le gros des calcul est effectué par un GPU, il n'en demeure pas moins qu'un thread CPU est nécessaire pour gérer les allers-retours mémoires et lancer les kernels CUDA. Comme nous pouvons le constater sur le graphe 6.29, la perte de cette unité de calcul CPU est toutefois largement compensée par l'apport d'un GPU sur tous nos cas tests.
La seule difficulté à la mise en œuvre efficace de notre traversée réside dans l'affectation des nombres de rayons que chacun des périphériques demande au manager de rayons. Nos expérimentations nous ont amenées à fixer ce seuil à 1024 pour le CPU, et 256000 pour le GPU. Ces valeurs ont prouvé leur efficacité sur nos deux configurations de tests, mais pourraient être remises en cause sur des configurations moins équilibrées au niveau du ratio de puissance entre le CPU et le GPU.

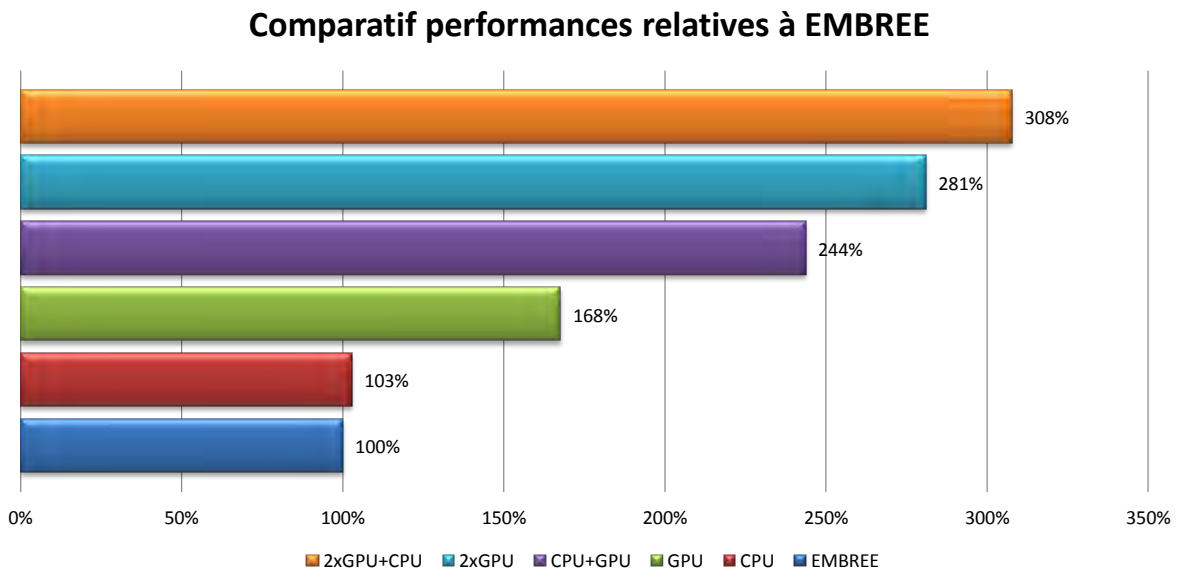


FIGURE 6.27 – Comparatif des performances de notre bibliothèque de lancer de rayon, face à l'application haute performance développée par Intel : EMBREE [INTa]. Les performances sont mesurées selon la même procédure pour les deux applications : path tracing avec des chemins à deux rebonds. Les résultats sont moyennés sur l'ensemble de nos scènes de tests, figure 6.14 table 6.3

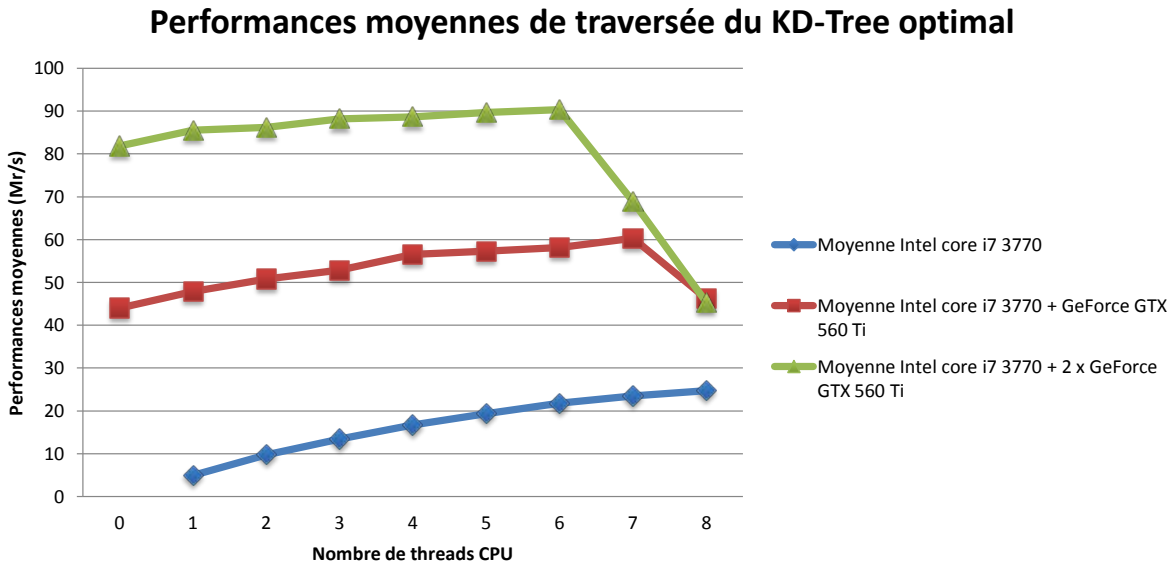


FIGURE 6.28 – Moyenne du nombres de millions de rayons par secondes traversant le KD-Tree sur nos scènes de tests, en fonction des périphériques et du nombre de threads CPU actifs.

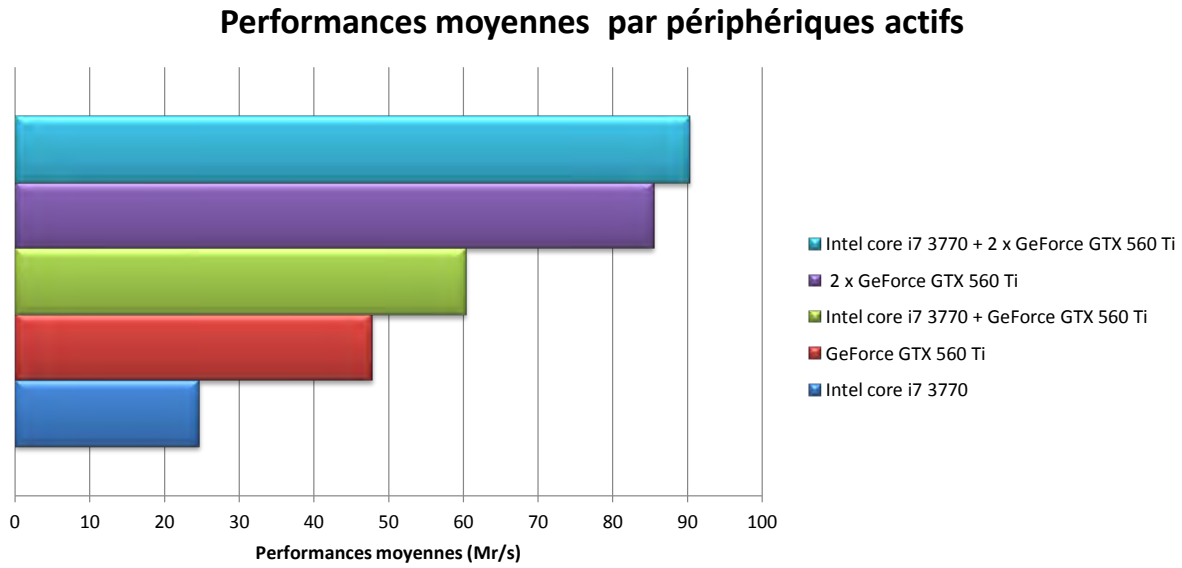


FIGURE 6.29 – Évolution moyenne des performances en fonction de l'activation des différents périphériques sur nos scènes de test.

7

Conclusion

Nos différentes contributions, que ce soit au niveau de la construction de la structure d'accélération ou de sa traversée, nous permettent d'atteindre des performances supérieures ou égales aux standards en terme de performances que ce soit sur CPU ou GPU.
Nous avons, de plus, atteint cet objectif en terme de performance sans perdre de vue nos contraintes industrielles. La plus grande force de notre bibliothèque de lancer de rayons se situe en effet dans sa grande souplesse d'utilisation ainsi que dans sa capacité à tirer au mieux parti des périphériques actifs en fonctions des requêtes de l'utilisateur.
Nous n'imposons qu'un minimum de contraintes afin de s'intégrer dans une application existante :

- Nos types de rayons et d'impacts sont imposés à l'application hôte en entrée et sortie.

- Notre bibliothèque accède aux données de la scène via une méthode que doit fournir l'utilisateur.

De plus, même si l'application hôte n'a pas vocation à lancer des paquets de rayons suffisamment importants pour justifier l'utilisation du GPU, notre gestionnaire des tâches de traversée se chargera de répartir les rayons à lancer sur le CPU en priorité. Ceci a pour effet de maintenir des performances optimales, sans nécessité d'intervention de l'utilisateur. Industriellement, ce facteur est très important. En effet, le temps requis pour intégrer notre bibliothèque dans une application est moindre s'il ne nécessite pas une refonte complète de l'application hôte pour constater une accélération.
Cette adaptabilité aux données en entrée est de plus totalement dynamique, et paramétrable par l'utilisateur via notre manageur de périphériques. Ceci est indispensable afin de laisser à l'application hôte le pouvoir de se réserver certaines ressources pour

d'autres tâches.
Enfin, notre interface de parallélisation est suffisamment souple pour permettre l'utilisation de périphériques extérieurs. Il serait tout à fait possible d'y greffer une couche réseaux permettant de lancer des calculs sur des CPU et GPU distants.

Deuxième partie

Parallélisation générique

1

Introduction et objectifs

Suite à nos travaux sur la requête d'intersections permettant de répartir les calculs sur tous les périphériques de manière transparente, nous nous sommes rapidement interrogés sur notre capacité à fournir à l'utilisateur une interface permettant de paralléliser de la même manière les traitements physiques affectés aux rayons. En effet, il est tout à fait naturel de vouloir affecter une phase de calcul à chacune des requêtes d'intersection. En général cette phase simule la physique associée aux phénomènes de rayonnements. Cette interface de parallélisation est motivée par la complexité du développement sur GPU, mais aussi par la difficulté à répartir efficacement des tâches entre CPU et GPU. Notre but est donc d'encapsuler toute cette partie complexe afin de fournir à nos clients une manière efficace et transparente de répartir leurs calculs sur les unités de calcul compatibles. Nos objectifs s'articulent autour de quatre points principaux que nous allons détailler.

1.1 Abstraction matérielle

Le premier objectif consiste à définir une interface de parallélisation permettant d'exécuter des tâches de calcul parallèles sur plusieurs types d'unités de calcul. Concrètement, cette abstraction matérielle doit permettre d'utiliser la même définition de la tâche de calcul pour toutes les unités de calcul, c'est à dire éviter d'écrire du code spécifique à chacune des unités de calcul. En environnement industriel, cette abstraction a une valeur ajoutée non-négligeable car elle permet de ne maintenir qu'une seule version du code parallèle pour toutes les différentes unités de calcul.

1.2 Hautes performances

L'objectif principal consiste bien sûr à obtenir de hautes performances via le mécanisme de parallélisation générique. Cet objectif est triple :

- Le surcoût dû à l'utilisation de la parallélisation générique doit être facile à amortir, même sur une configuration matérielle grand public.
- La parallélisation générique doit permettre d'écrire des tâches de calcul performantes pour l'ensemble des unités de calcul qu'elle est capable de gérer.
- La collaboration des différentes unités de calcul doit permettre une accélération des temps d'exécution en rapport avec la puissance de chacune des unités.

Si cet objectif de performance n'était pas atteint, l'utilisation de l'interface de parallélisation proposée aurait du mal à être justifié.

1.3 Robustesse

Dans l'optique d'une utilisation en milieu industriel, il est important que l'interface de parallélisation soit capable d'assurer à l'utilisateur que ses tâches de calcul ne seront réparties que sur des unités de calculs ayant la capacité à les recevoir sans défaut d'exécution prévisible.

1.4 Simplicité d'utilisation

Le dernier point essentiel consiste à ne pas perdre de vue que la parallélisation générique a pour objectif principal de simplifier la programmation hétérogène. C'est pourquoi la définition d'une tâche de calcul parallèle doit être relativement simple, et son lancement aisé. La capacité à debugger les tâches définies via la parallélisation générique proposée est un autre aspect de cette simplicité qui ne doit être négligé. Se retrouver face à un bogue spécifique à une unité de calcul sans aucun outil permettant de simplifier sa résolution n'est pas une solution viable à terme.

2

État de l'art

La parallélisation automatique ou assistée de code est un domaine largement étudiée. Parmi les solutions existantes, nous retiendrons deux approches courantes :

- La parallélisation via primitives de compilation

- La parallélisation via déclaration de tâches

2.1 Parallélisation via primitives de compilation

Cette approche, utilisée dans [Grob], consiste à incorporer des directives de compilation dans un code non parallèle afin de modifier le code généré pour tenter de le rendre parallèle automatiquement. Les avantages de cette approche résident dans son abstraction totale du parallélisme, et dans le fait qu'aucune connaissance des interfaces bas niveau utilisées n'est requise. Elle a par contre comme inconvénients de faire facilement perdre de vue la parallélisation lors de la conception. Ce qui peut amener à une parallélisation automatique très peu efficace.
Il est de plus complexe de notre côté de fournir un appel à notre requête d'intersection qui soit parallélisable via ce procédé sur différents périphériques. Enfin, la répartition des charges entre les périphériques n'est pas automatique, et il est nécessaire de spécifier un type d'accélérateur pour chacune des tâches parallélisés automatiquement.

2.2 Parallélisation via déclaration de tâches

Cette approche, utilisée dans [tea], consiste à déclarer des tâches de calcul qui peuvent être réalisées de manière concurrente. Cela permet de garder la parallélisation au centre de la conception en définissant les tâches de manière indépendante. De notre côté, il

est possible de fournir une requête d'intersection pour chacune des interfaces bas niveau utilisés par l'interface de parallélisation.
Cette méthode a pour inconvénient de nécessiter l'apprentissage des interfaces bas niveaux car elle utilise les implémentations fournit par l'utilisateur. De plus, la répartition des charges entre les périphériques doit être préparée par l'utilisateur, qui doit découper ses tâches de calcul avec la bonne granularité et fournir des implémentations pour chacun des périphériques qu'il souhaite utiliser.

2.3 Problématique

Les deux approches de parallélisation précédemment explicités ne permettent pas de répondre en totalité à notre contrainte principale, à savoir la répartition transparente des calculs entre plusieurs périphériques. C'est pourquoi nous avons défini une approche quelque peu différente des solutions existantes. Elle propose de déclarer des tâches parallèles à répartir entre les périphériques ayant comme fil conducteur :

- Un parallélisme explicite, qui reste central dans la conception
- Un détachement total des interfaces bas niveaux utilisées
- Une répartition inter-périphériques implicite et automatique
- Un accès privilégié à notre requête d'intersection
- Un respect du code utilisateur, qui ne sera pas altéré lors du processus de parallélisation

3

Abstraction du parallélisme

Notre interface de parallélisation générique peut être assimilée à une couche d'abstraction du parallélisme. Nous allons dans un premier temps détailler la modélisation logique de cette abstraction du parallélisme avant de présenter notre implémentation de cette modélisation.

3.1 Modélisation

Les modèles d'exécution GPU sont très particuliers, et étroitement liés au matériel qu'ils permettent de programmer. En effet, les notions de blocs de calcul, de *warps* ainsi que les différents niveaux de mémoire (global, shared, local) n'ont pas véritablement de sens pour un CPU. Il est toutefois difficile de s'en abstraire totalement sans réduire à néant le bénéfice de l'utilisation d'un GPU. C'est pourquoi, nous proposons un modèle d'exécution à la fois proche de celui des GPU en terme de conception mais beaucoup moins lié au matériel. La modélisation proposée s'appuie donc sur un concept de fil d'exécution hybride utilisant des données ayant certaines propriétés afin de réaliser une tâche globale de manière parallèle. Cette tâche sera subdivisée en sous-tâches afin d'être distribuée sur toutes les unités de calcul. Tous ces concepts sont détaillés ci-dessous. Malgré leur caractère générique, ces concepts ont pour objectif de permettre une utilisation efficace du GPU en particulier. En effet, si un CPU se révèle très souple avec ses appels virtuels performants, ses différents niveaux de cache et son accès direct à la mémoire centrale, ce n'est pas le cas du GPU actuellement. Il est donc raisonnable de s'adapter aux contraintes fondamentales des GPU, qui n'altèrent en rien l'efficacité des CPU très malléables.
Le tableau 3.1 illustre globalement la modélisation du parallélisme proposée et l'interconnexion entre tous les concepts en jeu.

3.1.1 Fil d'exécution

Le fil d'exécution est un concept essentiel de notre modélisation du parallélisme. Celui-ci est l'unité atomique en terme de répartition sur les différentes unités de calcul. Cela signifie qu'un fil d'exécution affecté à une unité de calcul sera entièrement traité par celle-ci. Une tâche de calcul parallèle est donc définie par un certain nombre de fils d'exécution ayant un identifiant unique. Le nombre de fils d'exécution est directement lié au nombre de données à traiter en parallèle dans la tâche de calcul, un par pixel pour générer une image par exemple.

3.1.2 Gestion des données

Afin de tirer partie de la puissance du GPU, il est indispensable de maximiser la contiguïté des données pour deux raisons :

- Optimiser les transferts mémoire afin de lancer des copies mémoire de zones mémoire étendues. En effet le coût de l'appel à la copie et la latence impliquée par une copie mémoire entre le CPU et le GPU ne sont possibles à amortir que pour un nombre relativement important de données contiguës copiées.

- Optimiser les temps de calcul. Le premier critère d'optimisation sur GPU se situe au niveau de l'accès aux zones mémoire. Il est en effet indispensable de maximiser les accès cohérents à la mémoire, mais aussi de maximiser leur coalescence.

Il est donc indispensable d'orienter l'utilisateur, le plus naturellement possible, vers un parallélisme basé données. C'est pourquoi notre modélisation du parallélisme fait appel à des tampons mémoire. Ceux-ci peuvent résider dans différentes zones mémoire, adressables directement ou pas par les différentes unités de calcul.

Les tâches de calcul parallèle font appel à ces zones mémoire afin d'associer les données en jeux avec chacun des fils d'exécutions.

3.1.3 Descripteur de tâche

Le descripteur de tâche réunit tous les concepts précédents. Il défini :

- Les opérations à effectuer par chacun des fils d'exécution.

- Les tampons mémoire nécessaires aux fils d'exécution.

- La manière d'accéder aux tampons mémoire.

- Le nombre de données à traiter.

Concrètement, les opérations à effectuer sont symbolisées par un noyau de code exécutable parallèlement et indépendamment sur plusieurs jeux de données. Les tampons mémoire utilisés et la manière d'y accéder permettent de définir pour un fil d'exécution donné, le sous ensemble de données qui doivent être adressables en lecture et/ou écriture. La manière d'accéder à un tampon mémoire se définit en fonction de deux propriétés distinctes :

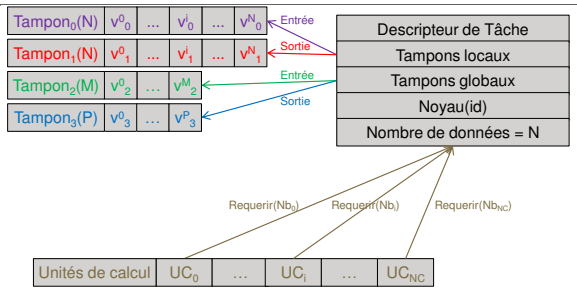
- Le degré de disponibilité du tampon mémoire. Celui-ci peut être :
    - local, ce qui signifie que seul le fil d'exécution ayant l'identifiant id au sein de la tache accèdera à la donnée à l'emplacement id du tampon mémoire.
    - global, ce qui signifie que tous les fils d'exécution peuvent potentiellement accéder à toutes les données du tampon mémoire.
- Le rôle joué par le tampon mémoire dans la ligne d'exécution. Celui-ci peut être :
    - une donnée en entrée, c'est-à-dire constante et donc accessible seulement en lecture.
    - une donnée en sortie, c'est à dire accessible uniquement en écriture.
    - une donnée en entrée/sortie, c'est à dire accessible à la fois en lecture et en écriture.

Ces tampons mémoire représentent les seules données ayant une durée de vie supérieure à celle du fil d'exécution.

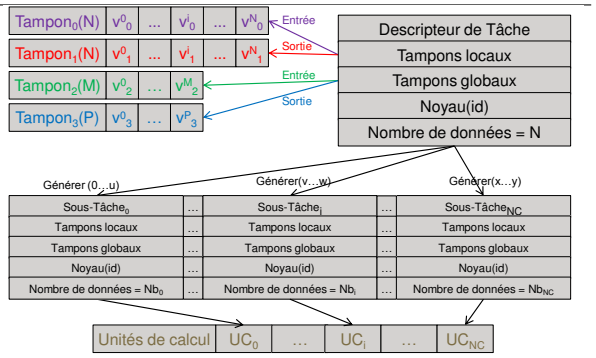
Le noyau de calcul d'une tâche peut en outre retourner un résultat. Cette valeur de retour pour chacun des fils d'exécution sera composée afin d'obtenir un résultat unique d'exécution de la tache dans sa globalité.

3.1.4 Sous-tâches de calcul

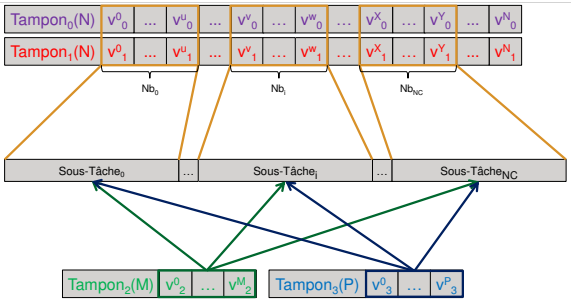
Le découpage en sous-tâches de calcul permet de répartir simplement les calculs entre les unités de calcul. Tant que la tâche courante n'est pas terminée, chacune des unités a la capacité de s'approprier une sous-partie des calculs en créant une sous-tâche de calcul. La création de cette sous-tâche utilise les informations contenues dans le descripteur de tâche afin de s'assurer que les éventuelles recopies mémoire entre les différents espaces d'adressage soient limitées aux données strictement nécessaires pour l'unité de calcul qui la reçoit.



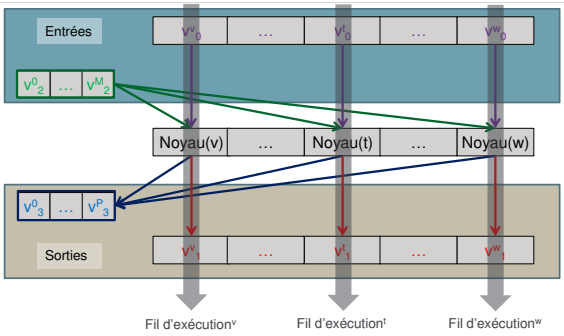
Un descripteur de tâche fait le lien entre les données et les unités de calcul. Chacune de ces unités a la capacité à demander une sous-tâche de calcul. Les types de tampons mémoire ainsi que leur mode d'accès permettent d'automatiser de manière efficace les transferts mémoire entre les différentes unités de calcul.



Chaque requête d'obtention de calculs par une unité de calcul génère une sous-tâche ayant un intervalle de données à traiter. Cet intervalle couvre le nombre de données demandé par l'unité s'il en reste suffisamment. Si la totalité des données sont attribuée, la sous-tâche générée est vide, ce qui conduit l'unité de calcul à se retirer de la tâche courante.



La répartition des données dans les sous-tâche de calcul tient compte des différents espaces d'adressage des unités de calcul ainsi que des propriétés des zones tampon. Les tampons mémoire locaux sont alloués et accessibles uniquement dans l'intervalle de données défini par la sous-tâche de calcul. Les tampons mémoire globaux sont alloués et accessibles en totalité par toutes les sous-tâches de calcul.



Chacun des fils d'exécution a accès à une "colonne" de données locales. Cette colonne de données est renforcée par un accès total aux données globales et doit permettre de réaliser l'exécution du noyau pour le fil d'exécution courant. Les données en entrée seulement ne sont copiées dans l'espace de l'unité de calcul qu'avant le lancement de la tâche. Les données en sortie seulement ne sont recopiés dans l'espace mémoire initial qu'à la fin de l'exécution de la sous-tâche.

TABLE 3.1 – Vue globale de la modélisation du parallélisme proposée et de tous les concepts auquel elle fait appel.

3.2 Implémentation

Afin de mettre à l'épreuve notre modélisation de la parallélisation, nous en proposons une implémentation. Nous détaillerons dans une première partie les choix techniques que nous avons effectués avant d'exposer l'interface utilisateur et enfin de préciser son fonctionnement interne.

3.2.1 Choix techniques

Notre implémentation repose en grande partie sur l'API CUDA. Ce choix se justifie du fait de la maturité de cette API, que ce soit en terme de fonctionnalités mais aussi en terme de performances et d'écosystème de développement. Pour ce qui est du langage de programmation choisi pour le code à répartir entre les CPU et GPU nous avons arrêté notre choix sur le sous-ensemble du langage C++ supporté par l'API CUDA. En effet, le C++ est un langage à la fois souple, performant, abouti et massivement utilisé dans le monde l'informatique graphique.

3.2.2 Interface utilisateur

Tout appel à la bibliothèque CUDA est totalement masqué par notre interface, mais malgré cela nous sommes dépendants d'un certain nombre de contraintes, directement issues de l'utilisation de CUDA en interne. Nous présenterons dans un premier temps ces contraintes, ainsi que l'impact de l'utilisation de notre implémentation de la parallélisation générique sur la chaine de compilation d'une application.

Contraintes de langage

Le cœur des tâches de calcul doit être écrit de manière à être exécutable à l'aide de CUDA. C'est pourquoi, il est nécessaire de restreindre les fonctionnalités C/C++ utilisables dans le noyau de calcul au sous-ensemble disponible dans CUDA. Parmi ces restrictions, décrites dans la documentation CUDA [NVIa], deux sont prépondérantes afin de mener à bien la conception des tâches de calcul :

- Utilisation "statique" de la mémoire. Il est en effet impossible d'utiliser une allocation dynamique sur GPU.

- Non-utilisation de la bibliothèque standard C++. Elle n'est en effet pas supportée sur GPU.

Ces contraintes sont évidentes pour des programmeurs ayant eu à côtoyer la programmation de GPU, mais doivent être assimilées au plus tôt car elles définissent en grande partie la façon de penser une application parallèle sur GPU.

Compilation

Les coeurs de calcul ainsi que toutes les méthodes auxquelles ils font appel doivent être disponibles à la fois sur CPU et sur GPU. Pour cela, nous faisons appel aux fonctionnalités du compilateur CUDA : NVCC [NVIc]. Les règles imposées pour la compilation sont les suivantes :

- NVCC doit être présent dans la chaine de compilation. Son intégration est toutefois automatisée pour Microsoft Visual Studio sur Microsoft windows ou Eclipse sur APPLE mac OS et Linux grâce au SDK CUDA [NVIb].

- Chacune des méthodes à répartir sur CPU et GPU doit être précédée du flag de compilation "HYBRID". Ce flag est indispensable pour spécifier à NVCC les méthodes qu'il doit répartir entre compilation CPU et GPU.

- Les méthodes hybrides doivent être déclarées et définies, soit directement dans un fichier d'entête (*.h), soit dans un fichier CUDA associé (*.cu) au lieu d'un fichier C/C++ (*.c/*.cpp).

Déclaration des données de calcul

Les données utilisables dans les noyaux de calcul de notre interface de parallélisation doivent nécessairement être déclarées à notre manageur de données. Celui-ci est un singleton, c'est à dire qu'il est unique et global à l'application. Il est capable d'allouer les zones mémoire ou de recevoir des pointeurs vers des zones mémoire du client. Une fois déclarée, la zone mémoire se voit affectée un identifiant unique que le client peut utiliser pour initialiser les tâches de calcul parallèles. Le manageur mémoire peut être utilisé en collaboration avec le manageur de périphérique, qui est lui capable de fournir un identifiant unique pour chacun des périphériques. Ainsi, notre manageur de données accepte cinq types de requêtes pouvant être effectuées pour n'importe quel périphérique grâce à son identifiant unique :

- Allocation mémoire : permet d'allouer une zone mémoire d'une taille spécifiée par l'utilisateur sur le périphérique désigné et d'obtenir son identifiant unique en plus de son pointeur.

- Lier une zone mémoire : permet d'enregistrer une zone mémoire du client pour la rendre éligible à notre interface de parallélisation via un identifiant unique.

- Modification externe : permet de signifier qu'un certain périphérique a modifié cette zone mémoire sans passer par les mécanismes de la parallélisation générique.

- Délier une zone mémoire : permet de rendre l'identifiant unique précédemment attribué à une zone mémoire du client, et met donc fin à son utilisation via notre interface de parallélisation.

- Libération mémoire : permet de rendre l'identifiant unique précédemment attribué à une zone mémoire allouée via le manageur de données. La libération mémoire est effectuée par le manageur, et met donc fin à son utilisation via notre interface de parallélisation.

Il est à noter que notre manageur de données est capable de gérer des espaces mémoire partagés, et donc adressables, par plusieurs périphériques. C'est le cas pour la mémoire "zéro-copie" de l'API CUDA. Il est donc possible d'allouer ou de lier des zones mémoire partagées entre plusieurs périphériques. Cela permet des optimisations internes lors du lancement des tâches de calculs. Une fois les données déclarées, l'utilisateur n'a plus à gérer quoi que ce soit pour ces données. Leur disponibilité et leur accès au sein de la tâche de calcul parallèle sont garantis, que ce soit sur CPU ou GPU.

Définition d'une tâche

La définition d'une tâche de calcul générique passe par un héritage C++. Il est nécessaire de définir une structure représentant la tâche, en héritant du type de tâche que l'on souhaite exécuter. Les types de tâches sont répartis selon deux critères :

- Présence d'un type de retour global, issu de la composition des différents retours de chacune des fils d'exécution.

- Capacité à lancer des rayons au sein de la tâche de calcul.

En fonction de ces critères, la tâche devra hériter d'une structure précise. La structure minimale de tâche parallèle, à savoir celle sans valeur de retour et sans capacité à lancer des rayons, se contente de déclarer les propriétés minimums suivantes :

- Le nombre de tampons mémoire locaux via un template C++.

- Le nombre de tampons mémoire globaux via un template C++.

- Le noyau de calcul via une méthode membre respectant nos contraintes de langage et de compilation et ayant pour en-tête :

```
HYBRID void compute(const unsigned int &globalId) const;
```

Hériter de la structure de tâches avec type de retour imposera de plus :

- La définition du type de retour RETURN_TYPE du noyau via un template C++.
- La modification de la signature du noyau de calcul en :

```
HYBRID RETURN_TYPE compute(const unsigned int &globalId)
    const;
```

- La définition de l'opérateur de composition binaire des résultats de chacun des fil d'exécution ayant pour en-tête :

```
HYBRID RETURN_TYPE binaryOperator( const RETURN_TYPE &left ,
    const RETURN_TYPE &right ) const;
```

Pour ce qui est de la capacité à lancer des rayons dans une tâche de calcul parallèle, hériter de cette structure va imposer :

- Le passage d'une scène de lancer de rayons lors du lancement du calcul parallèle d'une tâche. Celle-ci est en effet nécessaire pour initialiser en interne les données de lancer de rayons.

Dans le noyau de calcul, les accès aux données se font par des appels à des fonctions membres prédéfinies retournant des références vers les données. Ces fonctions membres ont les signatures suivantes :

- accès aux données locales

```
HYBRID DATA_TYPE &getLocalData<DATA_TYPE>(const unsigned int
    &dataId , const unsigned int &globalId) const;
```

avec :

  - dataId l'indice de la donnée au sein de la tâche.
  - globalId l'indice du fil d'exécution courant relativement à la tâche dans son ensemble.

- accès aux données globales

```
HYBRID DATA_TYPE &getGlobalData<DATA_TYPE>(const unsigned
    int &globalDataId, const unsigned int &id) const;
```

avec :

- globalDataId l'indice de la donnée globale au sein de la tâche.
- id l'indice de la donnée dans la donnée globale.

Exécution d'une tâche parallèle

Une fois les données déclarées au manageur de données et la tâche de calcul définie, il reste une dernière étape avant de pouvoir lancer l'exécution de la tâche de calcul parallèle. Il est nécessaire de fournir les identifiants uniques des zones mémoire liées à tâche de calcul. Pour cela la tâche de calcul doit être remplie avec les identifiants uniques de zones mémoire fournis par le manageur de données. L'ordre dans lequel ces identifiants sont envoyés dans la tâche de calcul est déterminant pour leur accès dans le noyau du calcul, puisque les valeurs dataId/globalDataId requises par les méthodes getLocalData/getGlobalData ne sont autres que les indices des données locales/globales au sein de la tâche de calcul. Pour remplir les identifiants de données, les tâches de calcul parallèles disposent de deux méthodes :

- renseigner les données locales

```
void setLocalData(const unsigned int &dataId, const unsigned
    int &uniqueID);
```

avec :

- dataId l'indice de la donnée au sein de la tâche.
- uniqueID l'indice unique fourni par le manageur de données.

- renseigner les données globales

```
void setGlobalData(const unsigned int &globalDataId, const
    unsigned int &uniqueID);
```

avec :

- globalDataId l'indice de la donnée globale au sein de la tâche
- uniqueID l'indice unique fourni par le manageur de données.

Le branchement des données terminé, il ne reste plus qu'à faire appel à la méthode statique de lancement de tâches parallèles correspondant au type de tâche :

- si la tâche n'a ni type de retour, ni possibilité de lancer des rayons

```
void parallelCompute(const parallelTask &userParallelTask);
```

- si la tâche a un type de retour mais ne lance pas de rayons

```
RETURN_TYPE parallelCompute<RETURN_TYPE>(const parallelTask
    &userParallelTask);
```

- si la tâche n'a pas de type de retour et peut lancer des rayons dans cette scène

```
void parallelCompute(const parallelTask &userParallelTask,
    const tracingScene &scene);
```

- si la tâche a un type de retour et peut lancer des rayons dans cette scène

```
RETURN_TYPE parallelCompute<RETURN_TYPE>(const parallelTask
    &userParallelTask, const tracingScene &scene);
```

A la manière de la répartition automatique des tâches de traversée décrite plus haut pour le lancer de rayons (6.3.1), notre manageur de périphériques permet d'activer ou de désactiver des périphériques pour la tâche qui va être lancée. Celle-ci sera donc répartie automatiquement entre les périphériques actifs sans aucune intervention de l'utilisateur. Celui-ci peut décider d'attendre les résultats de manière synchrone ou asynchrone. C'est-à-dire que le client peut choisir d'attendre la fin de l'exécution de la tâche pour reprendre la main, ou de continuer son exécution en ayant la possibilité d'interroger la tâche sur son statut.

3.2.3 Fonctionnement interne

L'encapsulation totale de l'utilisation de différents types de périphériques afin d'accomplir des tâches parallèles distribuées pose plusieurs problèmes auxquels nous avons répondu par divers mécanismes. Le premier consiste à être capable de s'adapter à l'environnement matériel disponible. La gestion des transferts mémoire est une contrainte majeure, en particulier à cause des différentes mémoires associées et adressables par un sous ensemble des périphériques. Enfin la dernière problématique majeure consiste à répartir efficacement les sous tâches de calcul sur les différentes unités de calcul. Nous allons présenter les pièces maitresses de notre organisation qui nous permettent d'outrepasser les problématiques énoncées.

Manageur d'unités de calcul

Le manageur d'unités de calcul est directement hérité de nos travaux sur la répartition des tâches de traversée lors du lancer de rayons 6.3.1. Celui-ci a plusieurs rôles :

- Détecter toutes les unités de calcul éligibles.
- Créer et affecter un thread de gestion par unité de calcul.
- Centraliser et gérer les requêtes d'affectation de ressources matérielles.

Notre manageur d'unités de calcul détecte comme éligible au calcul chaque cœur, physique ou logique, des CPU ainsi que tout GPU NVIDIA ayant des capacités de calcul CUDA supérieures ou égales à 2.0. Le CPU se voit attribuer les premiers identifiants uniques à raison de un par cœur. Tous les GPU compatibles sont ensuite évalués et classés par puissance décroissante pour l'attribution des identifiants uniques suivants. Notre manageur peut ensuite être vu comme une cagnotte de ressources. Il est possible de lui soumettre des requêtes d'attribution de ressources, qui seront honorées par une mise à disposition des threads de gestion associés aux unités de calcul demandées. L'accès systématique et centralisé à ce manageur assure que les unités de calcul ne sont pas agressées par différentes de nos tâches, que ce soit du lancer de rayons ou une quelconque tâche de parallélisation. En effet, il est nuisible en termes de performance, de lancer plus de threads ayant un fort besoin en puissance de calcul, qu'il n'y a de ressources effectives.

Manageur de données

Le rôle du manageur de données est crucial dans notre répartition transparente des tâches inter-périphériques. Les allocations mémoire et les transferts de données entre les différents espaces d'adressage se révèlent être bien souvent aux premiers postes en termes de temps d'exécution en environnement hétérogène. Notre manageur de données utilise les informations collectées par le manageur d'unités de calcul afin d'identifier le nombre d'espaces d'adressage différents mais aussi afin de détecter la présence ou pas d'un environnement CUDA. Si des GPU sont actifs, nos fonctions d'allocation et de recopie mémoire seront rebranchées dynamiquement vers les versions CUDA. En effet, même pour allouer des zones mémoire CPU, il est préférable d'utiliser des fonctions CUDA afin d'obtenir des performances supérieures lors des recopies. Le manageur de données vérifie aussi la présence d'espaces partagés entre plusieurs périphériques, dans notre cas seulement la mémoire dite "zéro-copie" de l'API CUDA.
Une fois ces informations en sa possession, le manageur de données est capable d'honorer les requêtes externes ou internes. Toutes les requêtes suivent une logique paresseuse afin

de minimiser les appels aux allocations mémoire ainsi qu'aux recopies mémoire. Lors de l'allocation ou de la liaison d'une zone mémoire à l'aide de notre manageur de données, celui-ci attribue non pas un unique pointeur pour cette zone mémoire, mais un pointeur pour ces données dans chacun des espaces mémoire différents. L'espace mémoire original est gardé en mémoire afin de le distinguer de ses éventuelles copies dans d'autres espaces mémoire. Lorsqu'une tâche de calcul requiert une donnée, en partie ou en totalité et pour un certain périphérique, le manageur vérifie d'abord si cette donnée a déjà été allouée dans cet espace mémoire. Si ce n'est pas le cas, l'allocation de la taille demandée est réalisée dans l'espace mémoire adéquat et les données demandées recopiées. S'il s'avère que cette donnée a déjà donné lieu à une allocation, la taille de la précédente allocation est vérifiée, et si elle s'avère suffisante pour honorer la requête, l'allocation précédente est réutilisée et la copie lancée sans ré-allocation mémoire. Ce modèle général est toutefois optimisé dans quelques cas :

- Si la donnée à transférer est située dans un espace partagé entre les périphériques source et destination, le pointeur de données est recopié tel quel et la totalité des données est marquée comme accessible par ce périphérique.

- Dans le cas où la totalité des données est accessible sur le périphérique destination, aucune recopie mémoire n'est lancée et le pointeur de donnée renvoyé correspond juste au pointeur alloué précédemment auquel un offset est ajouté pour atteindre le début de la zone mémoire réclamée.

Dans le cas où une allocation mémoire se voit refusée pour cause de mémoire insuffisante, il est possible de définir la stratégie à appliquer par le manageur de données :

- Supprimer toutes les allocations dans l'espace mémoire rempli.

- Désactiver le périphérique associé à cet espace mémoire pour la tâche courante.

- Appeler une méthode utilisateur fournissant les identifiants uniques des données à supprimer de cet espace mémoire.

Dans tous les cas, deux échecs successifs à l'allocation pour la tâche courante se solderont par une désactivation du périphérique associé à l'espace mémoire rempli pour la tâche courante.

Lorsqu'une tâche de calcul signale la finalisation de ses calculs sur un tampon mémoire, le manageur de données vérifie si l'espace d'adressage original de ce tampon mémoire diffère ou n'est pas partagé avec l'espace d'adressage de l'unité de calcul ayant généré le résultat. Dans ce cas, une recopie mémoire des données résultat est lancée à destination de l'espace mémoire original. C'est de cette dernière étape que découle une limitation sur

les types de données globales en sortie. En effet, les recopies mémoire étant organisées par blocs, il est actuellement impossible de gérer efficacement des accès en écriture concurrents et aléatoires par différentes unités de calcul, dans différents espaces mémoire sur la même zone mémoire finale. C'est pourquoi, les données globales en sortie sont actuellement limitées aux espaces mémoire adressables par tous les périphériques actifs pour la tâche de calcul.

Base de donnée des sous-tâches de calcul

Dès qu'une tâche parallèle est lancée, une base de données de sous-taches de calcul est créée. Celle-ci permet d'avoir un point de synchronisation pour les différentes unités de calcul intervenantes dans l'accomplissement des calculs. Il est possible de lui demander l'obtention d'un certain nombre de fils d'exécutions contigus. La base de donnée se charge de retourner l'intervalle de données finalement attribué suite à une requête. Lorsque elle ne dispose plus de données, elle rejette les requêtes, ce qui a pour effet de désactiver les différents intervenants dans le calcul. La base de donnée ne gère en aucun cas les nombres de données attribuées. C'est à l'intervenant de juger du nombre de données qu'il requiert.

Exécuteurs

Les exécuteurs sont les ressources allouées par le manageur d'unités de calcul qui vont effectivement réaliser les calculs. C'est ici que les différences en fonction des types d'unités de calcul sont les plus marquées. Les exécuteurs CPU et GPU ont en commun la phase d'initialisation des données. Lors de leur création, ils se voient attribuer une granularité, symbolisant le nombre de fils d'exécution contigus qu'ils vont traiter sans refaire appel à la base de donnée.
Lors de la première étape, chacun d'eux va faire appel au manageur de données afin d'allouer et copier les données globales, et allouer avec une certaine granularité les donnée locales de la tâche courante sur un espace d'adressage accessible par son unité de calcul. Si cette initialisation est réussie, l'exécuteur requiert un certain nombre de données (défini par sa granularité) à la base données des sous-tâches de calcul. Si celle-ci honore totalement ou en partie la requête, le manageur de données est encore une fois mis à contribution pour rendre disponible les données locales associées à l'intervalle de fils d'exécution consécutifs obtenu.
Les exécuteurs CPU se contentent ici de boucler sur l'index du fil d'exécution, à raison de un par passe, afin d'appeler la méthode compute sur chacun des fils. Si un type de retour est défini, l'opérateur binaire est systématiquement appelé entre le nouveau résultat et le résultat issu de l'application de l'opérateur binaire entre tous les résultats précédents.

Les exécuteurs GPU lancent un kernel CUDA, associant à chaque *thread* CUDA un fil d'exécution. Si une valeur de retour est définie, l'opérateur de composition est appliqué de manière optimisée sans aller-retour entre le CPU et le GPU grâce à une réduction parallèle spécialement conçu pour le GPU proposée par Harris dans [Har07]. A chaque fois qu'un exécuteur termine une sous-tâche de calcul, il valide ses résultats auprès du gestionnaire de données et requiert de nouvelles données à la base de donnée des sous-tâches de calculs. Dès lors que la base de donnée rejette la requête d'obtention d'une sous-tâche de calcul, l'exécuteur retourne la valeur de retour si la tâche en définit une puis se termine. La terminaison de tous les exécuteurs déclenche le réveil de l'appelant qui peut composer les résultats obtenus par les différentes unités de calcul si un type de retour est défini.

4

Applications

Afin de mesurer les performances, évaluer la simplicité d'utilisation et illustrer les utilisations possibles de notre interface de parallélisation, nous avons mis au point un jeu de tests. Ceux-ci sont conçus pour utiliser tous les types de tâche parallèle disponibles. Les résultats seront confrontés à une implémentation séquentielle de chacune des tâches, pour divers jeux de périphériques activés. Tous nos tests sont réalisés sur la même configuration. Le CPU utilisé dans nos mesures est un Intel Core I7-3770 et les deux GPU utilisés sont des NVidia GeForce GTX 560 TI.

4.1 Filtrage d'image

Cet exemple met à l'épreuve notre parallélisation sur une tâche de calculs sans type de retour ni capacité à lancer des rayons. Nous avons implémenté un filtre moyenne de taille 17x17, illustré sur la figure 4.1.

La tâche de calcul parallèle utilise deux zones mémoire, qui doivent être déclarées à notre manageur de données :

- une image source.

- une image destination.

L'image source est déclarée donnée globale en entrée pour la tâche, car le noyau de calcul doit accéder aux pixels voisins. L'image destination est déclarée donnée locale en sortie, en effet le noyau de calcul accède seulement au pixel associé à son indice global en écriture seulement. Le code source associé à la tâche de calcul parallèle est disponible sur la figure 4.2 et le code source permettant de lancer cette tâche répartie est détaillé sur la figure 4.3. Ces codes permettent de se rendre compte de la simplicité d'utilisation de l'interface de parallélisation. Les performances obtenues en fonction des périphériques actifs sont

détaillées sur le graphe 4.4. Le GPU est particulièrement efficace sur cette tâche, et les gains constatés lors de l'activation des différents périphériques sont en accord avec leurs capacités respectives. Les facteurs d'accélération obtenus par l'activation des différents GPU sont détaillés dans le graphe 4.5.

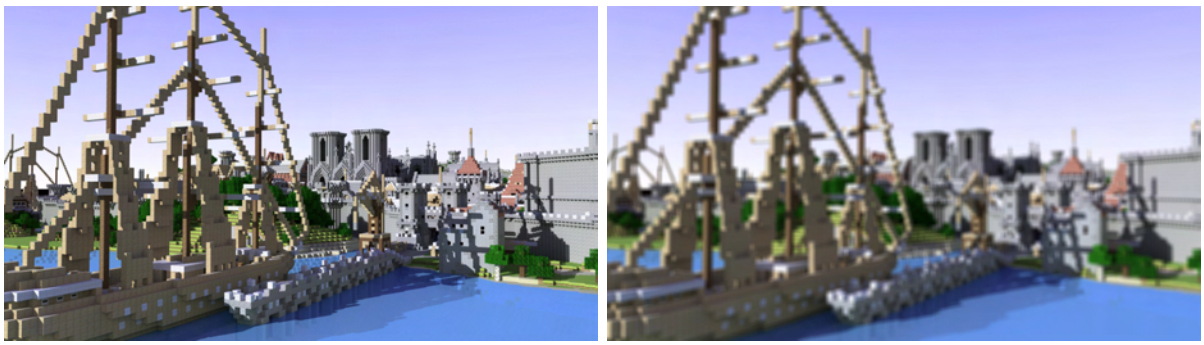


FIGURE 4.1 – Filtrage moyenne 17x17 appliqué à un rendu de Rungholt Full HD réalisé avec notre path tracer.

```
template<int IMG_WIDTH, int IMG_HEIGHT, int FILTER_SIZE>
struct filterImageParallelTask : public voidParallelTask<1, 1>
{
public:

  HYBRID void compute(const unsigned int &globalIndex) const
  {
    //compute coordinates
    const int width  = globalIndex / IMG_WIDTH;
    const int height= globalIndex - width * IMG_WIDTH;
    const int offset_boundary = FILTER_SIZE / 2;
    rawColor rawPixelColor = rawBlack;
    //add each pixels values in raw color
    for (int pixelOffsetX = -offset_boundary; pixelOffsetX <= offset_boundary; pixelOffsetX++)
    {
      const int currentWidth = __min(__max(0, width + pixelOffsetX), IMG_HEIGHT-1);
      for (int pixelOffsetY = -offset_boundary; pixelOffsetY <= offset_boundary; pixelOffsetY++)
      {
        const int currentHeight = __min(__max(0, height + pixelOffsetY), IMG_WIDTH-1);
        rawPixelColor += getGlobalData<color>(getTaskInputImageGlobalId(), currentWidth * IMG_WIDTH + currentHeight);
      }
    }
    //store divided and compressed final color
    color &finalColor      =      getLocalData<color>(getTaskOutputImageLocalId(), globalIndex);
    finalColor             =      rawPixelColor.compress(rawPixelColor, powf(2 * offset_boundary + 1, 2));
  }

  HYBRID unsigned int getTaskInputImageGlobalId() const
  {
    return 0;
  }

  HYBRID unsigned int getTaskOutputImageLocalId() const
  {
    return 0;
  }
};
```

FIGURE 4.2 – Définition de la tâche de calcul parallèle pour application du filtrage moyenne 17x17 via notre interface de parallélisation.

```
void filterImage(const dataId     &inputImageDataId,  const dataId  &outputImageDataId)
{
  filterImageParallelTask<1920, 1080, 17> task;
  //local
  task.setLocalData(outputImageDataId, task.getTaskOutputImageLocalId(), OUTPUT);
  //global
  task.setGlobalData(inputImageDataId, task.getTaskOutputImageLocalId(), INPUT);
  //set nb results
  task.setNbElements(1920 * 1080);
  //launch
  parallelCompute(task);
}
```

FIGURE 4.3 – Lancement d'une tâche de filtrage parallèle répartie automatiquement entre les unités de calcul.

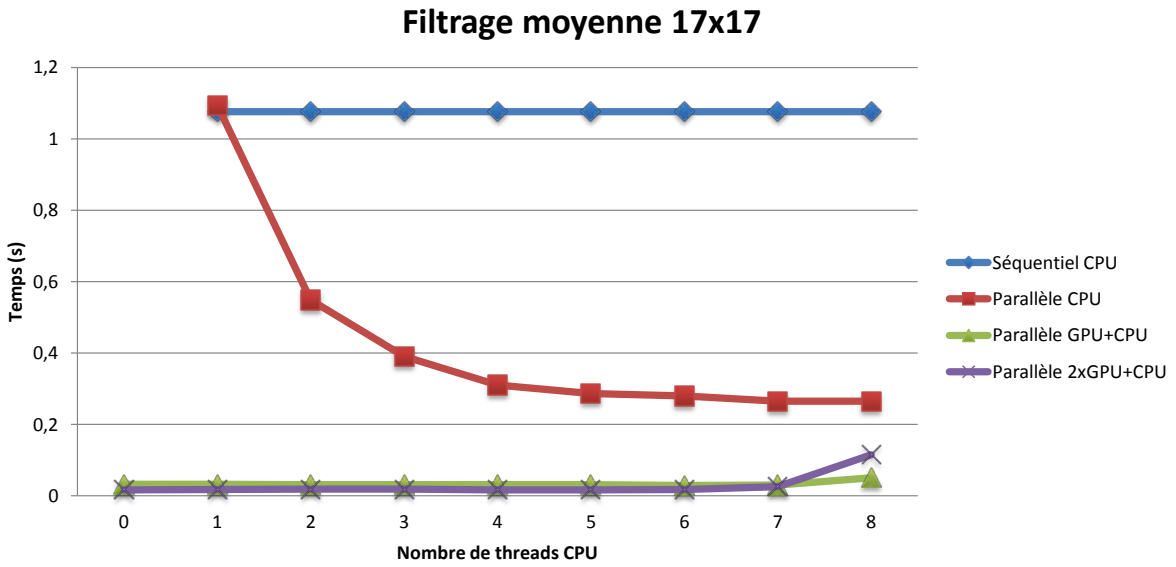


FIGURE 4.4 – Moyenne des temps de filtrage de l'image Full HD, en fonction des périphériques et du nombre de threads CPU actifs.

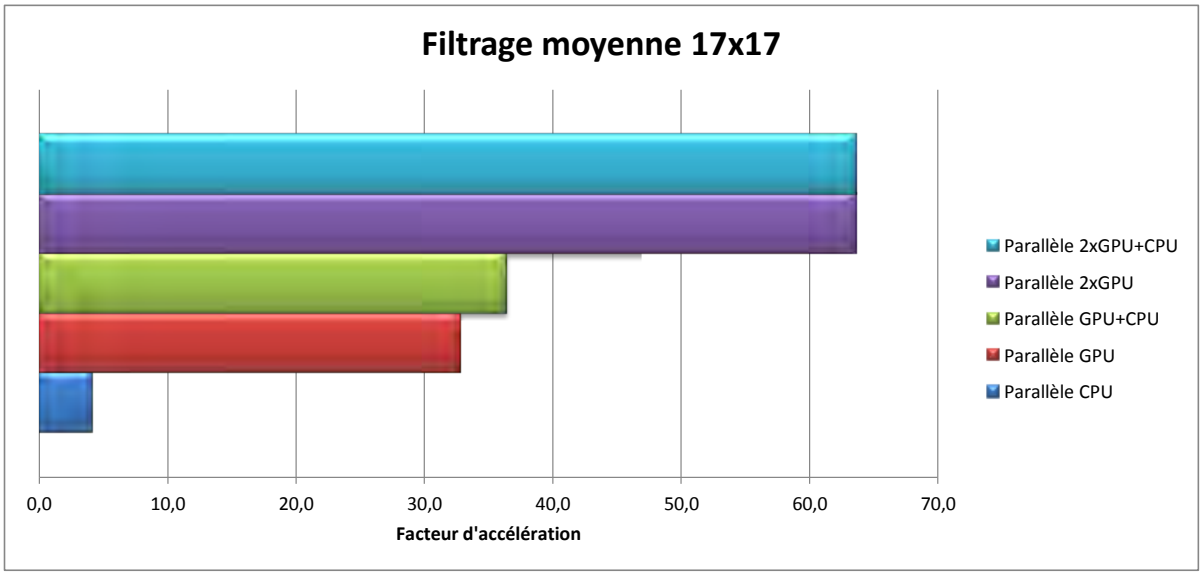


FIGURE 4.5 – Facteurs d'accélération du temps de filtrage pour une image Full HD selon les périphériques activés, relativement à la version séquentielle.

4.2 Filtrage d'image avec retour des extremums de luminosités

Cet exemple a pour but d'illustrer les performances et la simplicité d'obtention d'un résultat global d'exécution. Cette tâche avec type de retour se propose de réaliser exactement le même filtrage que la tâche de test précédente 4.1 en y ajoutant le retour à la fois du minimum et du maximum de luminosité sur l'ensemble des pixels de l'image. La tâche de calcul utilise exactement les mêmes données que la tâche de test précédente. Les modifications effectuées au niveau du code sont disponibles sur les figures 4.6 et 4.7. Encore une fois, le code reste compact et lisible. Les résultats obtenus en terme de performances en fonctions des périphériques sont détaillés sur les graphes 4.8 et 4.9. Les résultats sont globalement similaires aux résultats précédents, ce qui prouve que notre composition de résultats est suffisamment performante pour passer totalement inaperçue face aux calculs définis par la tâche.

```
struct extremum
{
public :
  HYBRID extremum(const float &minValue = FLT_MAX, const float &maxValue = -FLT_MAX)
  {
  min = minValue;
  max = maxValue;
  }
  float min;
  float max;
};
template<int IMG_WIDTH, int IMG_HEIGHT, int FILTER_SIZE>
struct filterImageReturnParallelTask : public ParallelTask<1, 1, extremum>
{
public:

  HYBRID extremum compute(const unsigned int &globalIndex) const
  {
      //compute coordinates
    const int width        = globalIndex / IMG_WIDTH;
    const int height= globalIndex - width * IMG_WIDTH;
    const int offset_boundary = FILTER_SIZE / 2;
    rawColor rawPixelColor = rawBlack;
    //add each pixels values in raw color
    for (int pixelOffsetX = -offset_boundary; pixelOffsetX <= offset_boundary; pixelOffsetX++)
    {
      const int currentWidth = __min(__max(0, width + pixelOffsetX), IMG_HEIGHT-1);
      for (int pixelOffsetY = -offset_boundary; pixelOffsetY <= offset_boundary; pixelOffsetY++)
      {
        const int currentHeight = __min(__max(0, height + pixelOffsetY), IMG_ WIDTH-1);
        rawPixelColor += getGlobalData<color>(getTaskInputImageGlobalId(), currentWidth * IMG_WIDTH + currentHeight);
      }
    }
    //store divided and compressed final color
    color &finalColor      =      getLocalData<color>(getTaskOutputImageLocalId(), globalIndex);
    finalColor             =      rawPixelColor.compress(rawPixelColor, powf(2 * offset_boundary + 1, 2));

    return extremum(finalColor.getLuminance(), finalColor.getLuminance());
  }

  HYBRID unsigned int getTaskInputImageGlobalId() const
  {
    return 0;
  }

  HYBRID unsigned int getTaskOutputImageLocalId() const
  {
    return 0;
  }

  HYBRID extremum getInitialValue() const
  {
    return extremum();
  }

  HYBRID extremum binaryOperator(const extremum &left, const extremum &right) const
  {
    return extremum(__min(left.min, right.min), __max(left.max, right.max));
  }
};
```

FIGURE 4.6 – Définition de la tâche de calcul parallèle pour application du filtrage moyenne 17x17 avec retour des extremums de luminosité via notre interface de parallélisation.

```
extremum filterImageReturn( const dataId &inputImageDataId,  const dataId  &outputImageDataId)
{
  filterImageParallelTask<1920, 1080, 17> task;
  //local
  task.setLocalData(outputImageDataId, task.getTaskOutputImageLocalId(), OUTPUT);
  //global
  task.setGlobalData(inputImageDataId, task.getTaskOutputImageLocalId(), INPUT);
  //set nb results
  task.setNbElements(1920 * 1080);
  //launch
  return parallelCompute<extremum>(task);
}
```

FIGURE 4.7 – Lancement d'une tâche de filtrage parallèle avec retour des extremums de luminosité répartie automatiquement entre les unités de calcul.

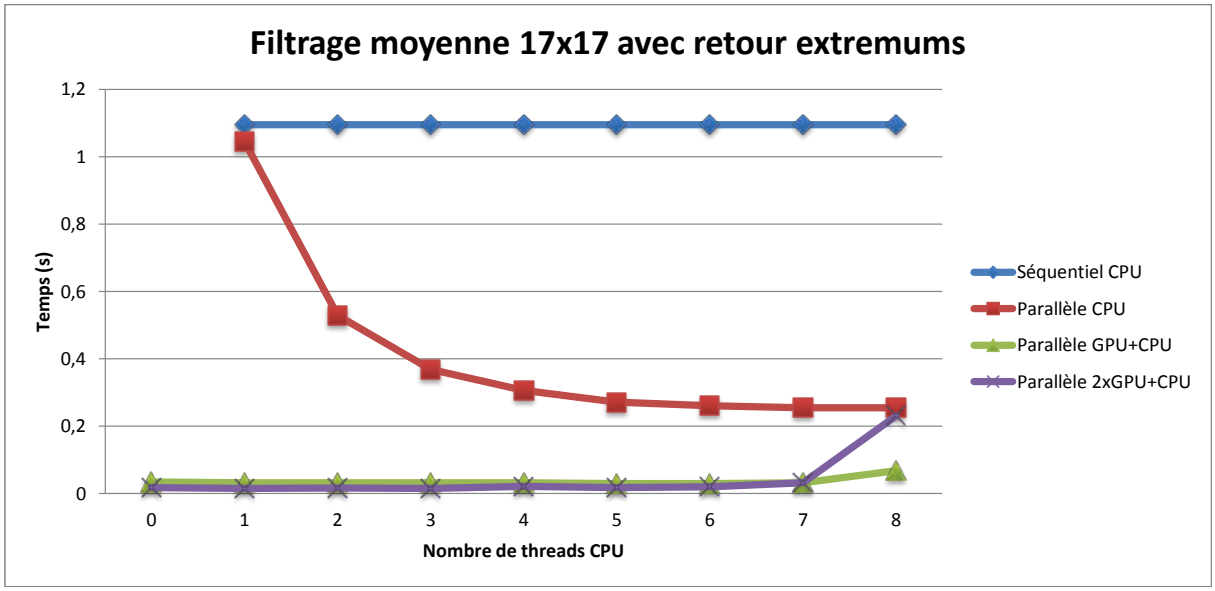


FIGURE 4.8 – Moyenne des temps de filtrage et de retour des extremums de luminosités de l'image Full HD, en fonction des périphériques et du nombre de threads CPU actifs.

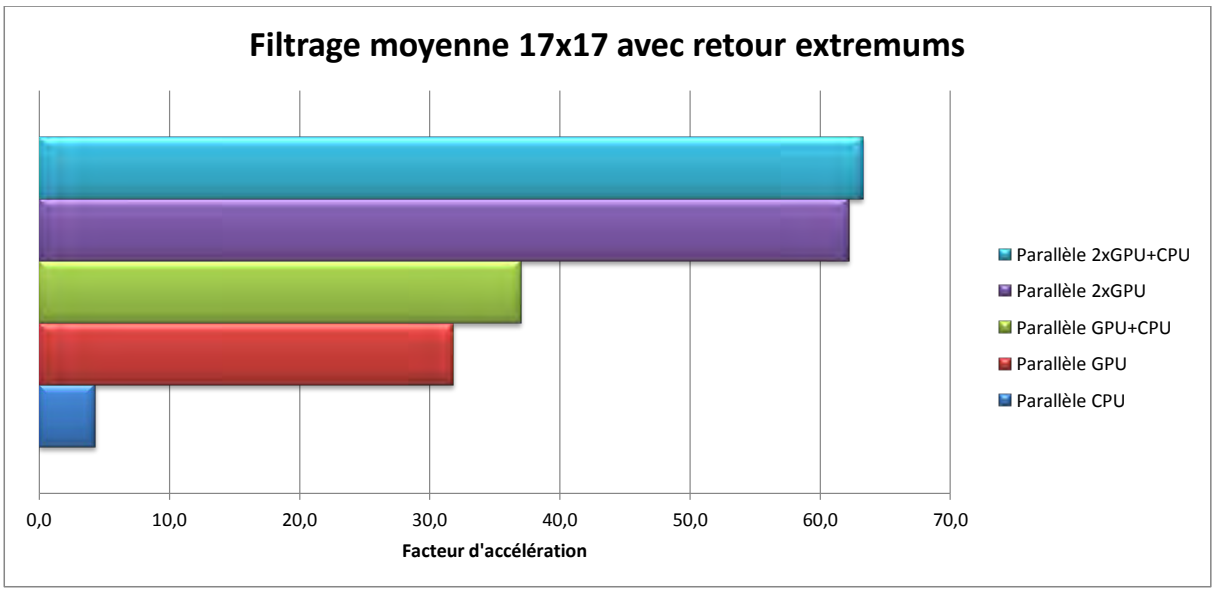


FIGURE 4.9 – Facteurs d'accélération du temps de filtrage et de retour des extremum de luminosités pour une image Full HD selon les périphériques activés, relativement à la version séquentielle.

4.3 Lancer de rayons minimal

Cet exemple démontre l'utilité de pouvoir lancer des rayons au sein d'une tâche parallèle. Nous avons implémenté un lancer de rayons minimal, se contentant de lancer les rayons caméra et se limitant à un modèle simpliste de matériaux pour lequel la couleur n'est liée qu'à l'incidence du rayon et à la normale du triangle intersecté. Cet algorithme est composé de trois étapes :

- La génération des rayons cameras.
- L'appel à la requête d'intersection parallèle.
- Le traitement des impacts, c'est-à-dire le calcul des matériaux simplifiés et le remplissage de l'image.

Nous avons écrit deux versions de cet algorithme :

- La première utilise la répartition classique des tâches de lancer de rayons définie plus haut 6.3.1. Cela implique la définitions de deux tâches parallèles entrecoupées par un appel à la requête d'intersection parallèle. Les rayons sont donc générés dans une tâche parallèle, puis intersectés de manière répartie, et enfin une dernière tâche de calcul parallèle évalue les matériaux et les stocke dans une image. Le code associé à ces deux tâches est décrit dans la figure 4.10. La fonction permettant de générer l'image est quant à elle disponible à la figure 4.11.
- La seconde réalise les trois étapes directement au sein d'une même tâche de calcul parallèle ayant la possibilité de lancer des rayons. Le code de la tâche de calcul est décrit dans la figure 4.12, le code permettant de lancer la génération de l'image est exposé dans la figure 4.13.

Chacune des versions a été testée sur le modèle Rungholt, pour produire l'image 4.14. Les performances de ces deux méthodes sont illustrées par les graphes 4.15 et 4.16 pour la version en deux passes, et les graphes 4.17 et 4.18 pour la version en une passe. la différence de performances est sans appel. Si le mécanisme de répartition entre les périphériques se comporte de manière satisfaisante sur les deux méthodes, les facteurs d'accélération bruts atteints par la méthode en une passe sont deux à trois fois supérieurs à ceux atteints par la méthode en deux passes. Cela confirme l'intérêt, non seulement en terme de simplicité du code mais aussi en terme de performance de la possibilité d'écrire des tâches de calculs parallèles ayant la possibilité de lancer des rayons.

```
template<int IMG_WIDTH, int IMG_HEIGHT>
struct rayTraceSimpleParallelFirstPass : public voidParallelTask<1, 1>
{
public:
  HYBRID void compute(const unsigned int &globalIndex) const
  {
    const int height  = globalIndex / IMG_WIDTH;
    const int width   = globalIndex - height * IMG_WIDTH;
    ray &currentRay   = getLocalData<ray>(getTaskRayLocalId(), globalIndex);
    const camera &cam = getGlobalData<camera>(getTaskCameraGlobalId, 0);
    cam.generateRay(width, height, currentRay);
  }
  HYBRID unsigned int getTaskRayLocalId()
  {
    return 0;
  }
  HYBRID unsigned int getTaskCameraGlobalId()
  {
    return 0;
  }
};

struct rayTraceSimpleParallelSecondPass : public voidParallelTask<3, 0>
{
public:
  HYBRID void compute(const unsigned int &globalIndex) const
  {
    const ray &currentRay    = getLocalData <ray>(getTaskRayLocalId(), globalIndex);
    const hit &currentHit    = getLocalData <hit>(getTaskHitLocalId(), globalIndex);
    color &finalColor = getLocalData <color>(getTaskOutputImageLocalId(), globalIndex);
    if (currentHit.hitSomething())
    {
      const float colorFactor = pscal(currentRay.direction, currentHit.normal);
      finalColor = color(colorFactor, colorFactor, colorFactor);
    }
    else
    {
      finalColor = black;
    }
  }
  HYBRID unsigned int getTaskRayLocalId()
  {
    return 0;
  }
  HYBRID unsigned int getTaskHitLocalId()
  {
    return 1;
  }
  HYBRID unsigned int getTaskOutputImageLocalId()
  {
    return 2;
  }
};
```

FIGURE 4.10 – Définition des tâches de calcul parallèle pour le rendu simplifié en deux passes via notre interface de parallélisation.

```
void rayTraceSimpleTwoPasses(const dataId  &cameraDataId ,const dataId &outputImageDataId,  const dataId &rayDataId
       ,      const dataId         &hitDataId
       ,      const tracingScene   &rayTracingScene
       )
{
  rayTraceSimpleParallelFirstPass<1920, 1080> firstTask;
  firstTask.setLocalData(rayDataId, firstTask.getTaskRayLocalId(), OUTPUT);
  firstTask.setGlobalData(cameraDataId, firstTask.getTaskCameraGlobalId(), INPUT);
  firstTask.setNbElements(1920 * 1080);
  parallelCompute(firstTask);
  rayTracingScene.trace(rayDataId, hitDataId);
  rayTraceSimpleParallelSecondPass secondTask;
  secondTask.setLocalData(rayDataId, secondTask.getTaskRayLocalId(), INPUT);
  secondTask.setLocalData(hitDataId, secondTask.getTaskHitLocalId(), INPUT);
  secondTask.setLocalData(outputImageDataId, secondTask.getTaskOutputImageLocalId(), OUTPUT);
  secondTask.setNbElements(1920 * 1080);
  parallelCompute(secondTask);
}
```

FIGURE 4.11 – Lancement des deux tâches de calcul parallèle et de la passe de lancer de rayons pour le rendu simplifié. Les tâches et le lancer de rayons sont répartis automatiquement entre les unités de calcul.

```
template<int IMG_WIDTH, int IMG_HEIGHT>
struct rayTraceSimpleOnePass : public voidTracingTask<1, 1>
{
public:

  HYBRID void compute(const unsigned int &globalIndex) const
  {
    const int height = global_index / IMG_WIDTH;
    const int width = global_index - height * IMG_WIDTH;

    ray currentRay;
    hit currentHit;

    const camera &cam = getGlobalData<camera>(getTaskCameraGlobalId, 0);

    cam.generateRay(width, height, currentRay);

    trace(currentRay, currentHit);

    color &finalColor             = getLocalData<color>(getTaskOutputImageLocalId(), globalIndex);

    if (currentHit.hitSomething())
    {
      const float colorFactor = pscal(currentRay.direction, currentHit.normal);
      finalColor = color(colorFactor, colorFactor, colorFactor);
    }
    else
    {
      finalColor = black;
    }
  }

  HYBRID unsigned int getTaskCameraGlobalId()
  {
    return 0;
  }

  HYBRID unsigned int getTaskOutputImageLocalId()
  {
    return 0;
  }
};
```

FIGURE 4.12 – Définition de la tâche de calcul parallèle pour le rendu simplifié en une passe via notre interface de parallélisation.

```
void rayTraceSimpleOnePass(
              const dataId         &cameraDataId
       ,      const dataId         &outputImageDataId
       ,      const tracingScene   &rayTracingScene
       )
{
  rayTraceSimpleOnePass<1920, 1080> task;
  //local
  task.setLocalData(outputImageDataId, task.getTaskOutputImageLocalId(), OUTPUT);
  //global
  task.setGlobalData(cameraDataId, task.getTaskCameraGlobalId(), INPUT);
  //set nb results
  task.setNbElements(1920 * 1080);
  //launch
  parallelCompute(task, rayTracingScene);
}
```

FIGURE 4.13 – Lancement de la tâche de calcul parallèle pour le rendu simplifié en une passe.

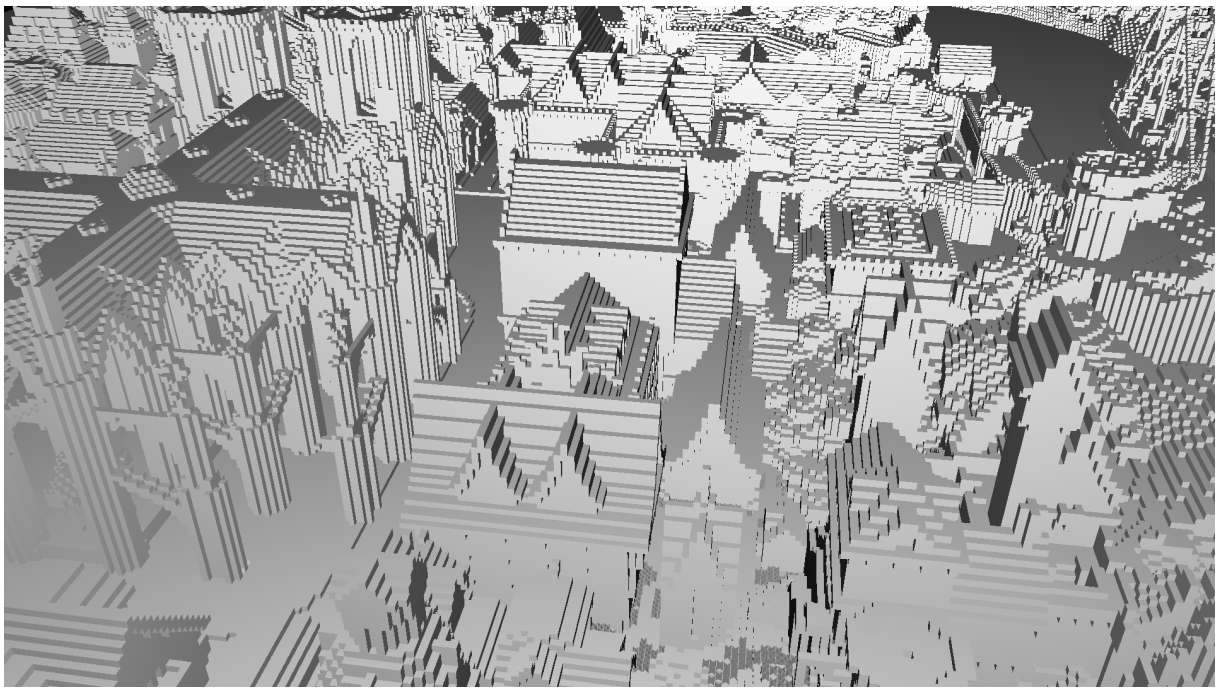


FIGURE 4.14 – Scène de test (Rungholt) rendue avec le lancer de rayons simplifié.

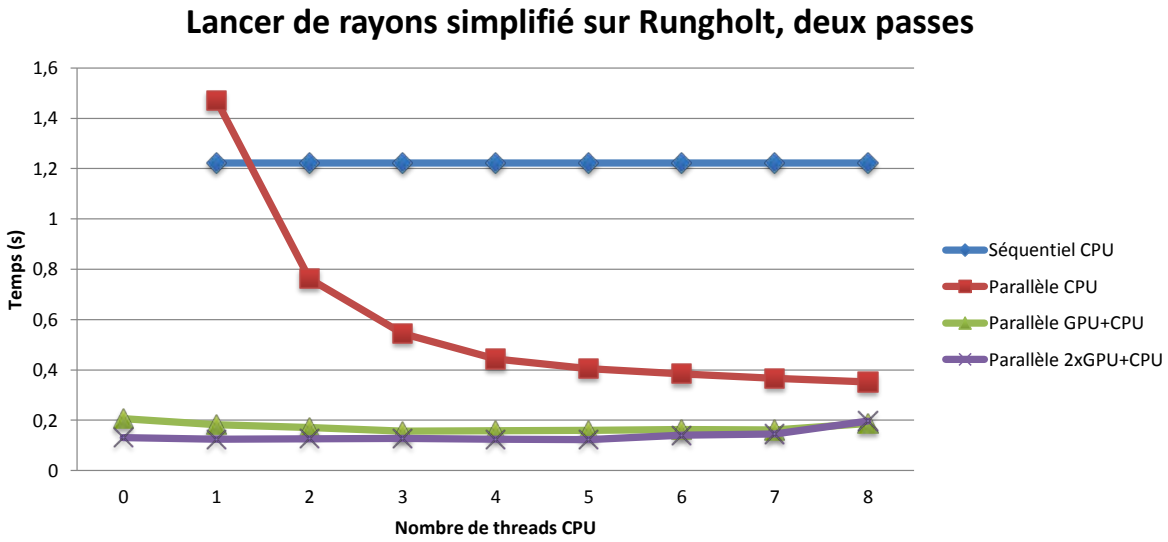


FIGURE 4.15 – Temps pour le rendu simplifié de Rungholt en Full HD avec l'algorithme en deux passes, en fonction des périphériques et du nombre de threads CPU actifs.

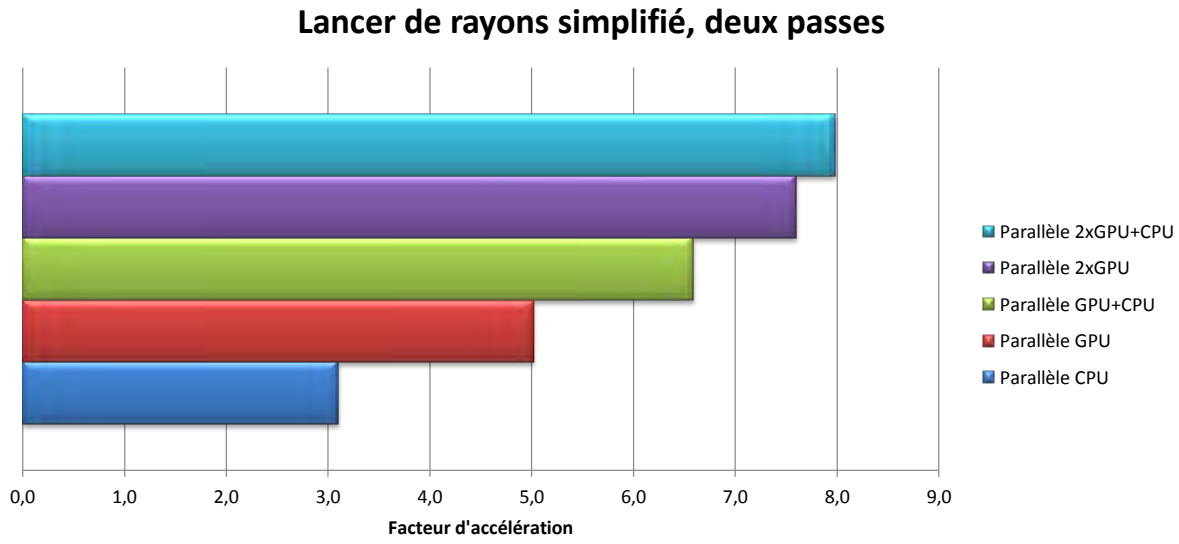


FIGURE 4.16 – Facteurs d'accélération pour le rendu simplifié sur nos scènes de test en Full HD avec l'algorithme en deux passes selon les périphériques activés, relativement à la version séquentielle.

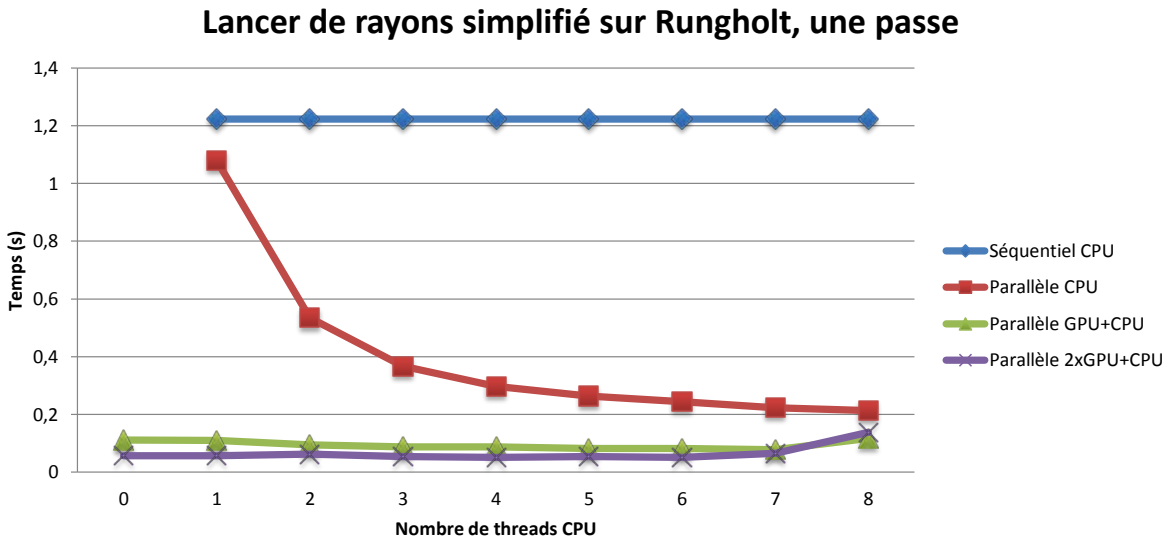


FIGURE 4.17 – Temps pour le rendu simplifié de Rungholt en Full HD avec l'algorithme en une seule passe, en fonction des périphériques et du nombre de threads CPU actifs.

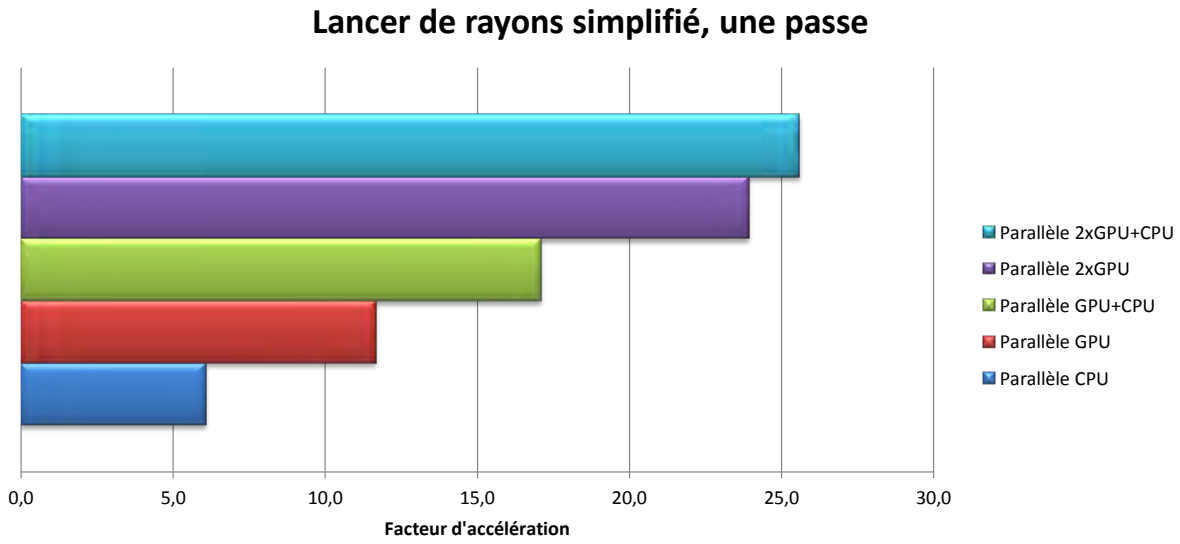


FIGURE 4.18 – Facteurs d'accélération pour le rendu simplifié sur nos scènes de test en Full HD avec l'algorithme en une passe selon les périphériques activés, relativement à la version séquentielle.

4.4 Path tracing

Notre dernier test a consisté à réaliser un path tracer entièrement parallélisé grâce à notre interface de parallélisation générique. Le principe du path tracing est de ne pas générer seulement des rayons caméras pour calculer une image, mais de générer des chemins de rayons partant de la caméra et rebondissant dans la scène. Cela permet de ramener de la lumière de manière indirecte, c'est-à-dire sans que la source ne soit forcement visible par l'impact du rayon camera. L'utilisation de nos tâches de calcul parallèles avec lancer de rayons nous permet de gérer dynamiquement les transparences, sans imposer le lancement de passes supplémentaires pour gérer les traversées de transparence. Notre path-tracer gère de plus les textures, pour les couleurs de matériaux mais aussi pour les transparences ainsi que l'utilisation de normales interpolées afin d'éliminer les effets polygonaux sur les objets arrondis. Les rendus de nos scènes de test disponibles sur la figure 6.14 ont étés réalisés avec notre path tracer. De la même manière que pour les autres tests, les performances du path tracer en fonction des périphériques actifs sur deux modèles sont disponibles dans les graphes 4.19 et 4.20. Les facteurs d'accélération moyens obtenus sur l'ensemble de nos scènes de test sont illustrés sur le graphe 4.21. Le CPU et le GPU se révèlent ici très proches en termes de performances. En effet, d'une part l'incohérence globale des chemins d'exécution mais aussi l'incohérence des rayons à tracer et enfin le surcoût des transferts mémoire entre le CPU et le GPU limitent l'avance massive que pouvait avoir le GPU en terme de performance sur des applications plus "simples". Encore une fois, notre interface de parallélisation se montre capable de gérer de manière efficace les différents périphériques disponibles, et atteint des facteurs d'accélération de l'ordre de x5-x7 par périphérique actif comparativement à la version séquentielle.

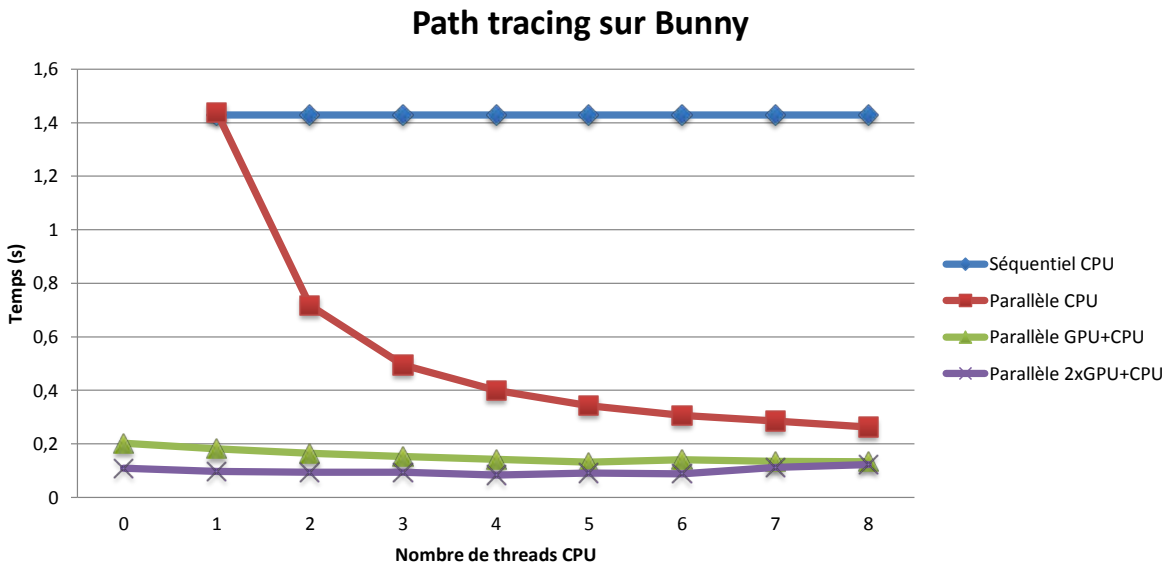


FIGURE 4.19 – Temps de rendu d'une image pour Bunny en Full HD avec notre path tracer pour des chemins de longueur 4, en fonction des périphériques et du nombre de threads CPU actifs.

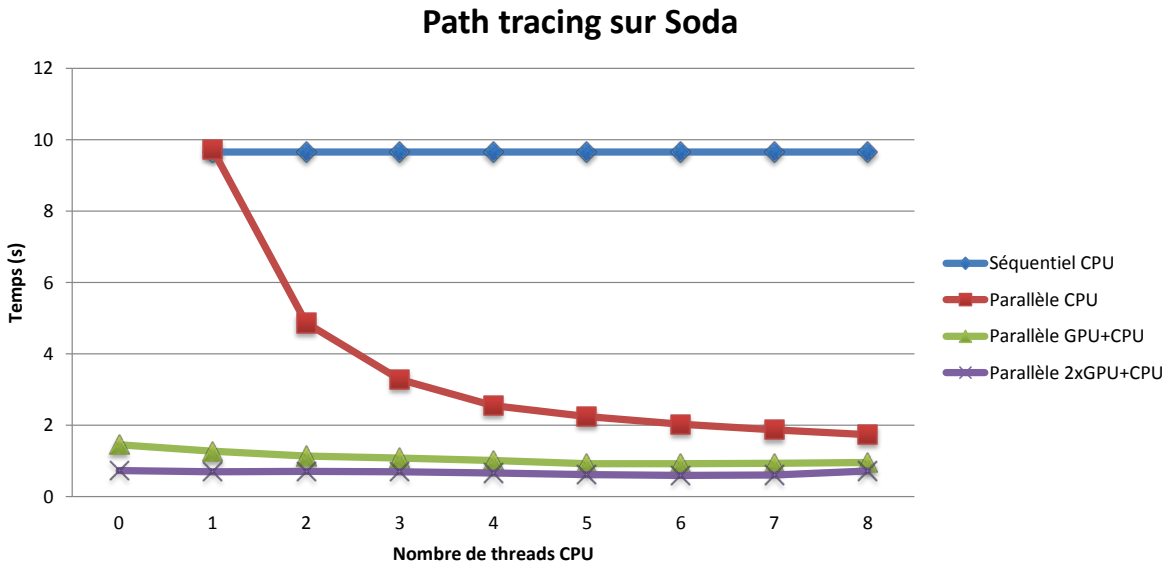


FIGURE 4.20 – Temps de rendu d'une image pour Soda Hall (vue intérieure) en Full HD avec notre path tracer pour des chemins de longueur 4, en fonction des périphériques et du nombre de threads CPU actifs.

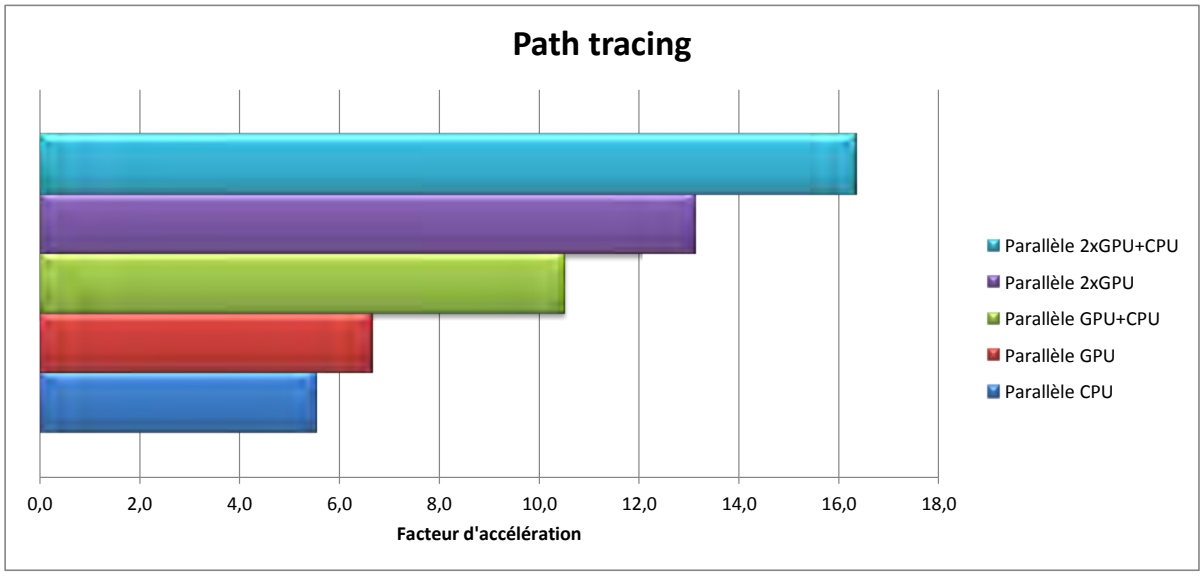


FIGURE 4.21 – Facteurs d'accélération moyens du temps de rendu par path tracing pour des chemins de longueur 4 sur nos scènes de test en Full HD selon les périphériques activés, relativement à la version séquentielle.

5

Conclusion

Comme nous pouvons le constater au travers des diverses applications que nous avons explorées pour notre interface de parallélisation, notre objectif de performances est bien respecté. Selon les affinités avec les unités de calcul, notre interface de parallélisation tire plus ou moins parti de l'activation du CPU ou du GPU, mais ne bride en aucun cas les performances en surexploitant un périphérique plus lent. Chaque type d'unité de calcul apporte ce dont il est capable sans bloquer les autres périphériques. Cette bonne évolution en fonction des unités activées est de plus confortée par les facteurs d'accélération bruts obtenus. En effet, l'utilisation de notre interface de parallélisation générique sur nos cas tests se traduit par des gains moyens de x5 en environnement CPU, et de x25 dès lors que le CPU collabore avec le GPU. Comme constaté lors de la parallélisation du lancer de rayons, il est indispensable de réserver un thread CPU par GPU actif afin de maximiser les performances. De plus, la comparaison avec les algorithmes séquentiels démontre que le surcoût lié à l'utilisation de notre interface de parallélisation générique est négligeable. En effet, la totalité du traitement d'une tâche est généré par des mécanismes de template C++, et tient dans un code très restreint. Cela permet une optimisation maximale du code généré durant la phase de compilation, au prix d'un temps de compilation plus important. C'est pourquoi une implémentation spécifique de la parallélisation hybride n'aurait pas vraiment de sens en terme de performance et donc très peu d'intérêt de manière générale. Au delà des performances, notre interface de parallélisation se révèle plutôt facile à utiliser pour un développeur classique, même si elle ne peut masquer totalement les contraintes imposées par l'utilisation d'un GPU pour les calculs. Les codes des tâches de calcul ainsi que de lancement de tâches de calcul parallèles sont clairs pour un développeur habitué au C++, et très loin des considérations matérielles liées aux groupes, *warps* ou autres niveaux de caches imposés par CUDA. Notre interface est de plus compatible avec le debugger NVidia NSight, ce qui permet de debugger du code purement GPU de manière confortable.

Une autre force de notre parallélisation générique réside dans sa capacité à gérer les espaces mémoire CPU et GPU automatiquement, sans imposer leurs localités. En effet, nos tests utilisent tous la mémoire centrale comme zone mémoire de résidence, mais si les zones mémoire déclarées au manageur de données sont situées sur un GPU, la parallélisation fonctionnera exactement de la même manière tout en minimisant les aller-retours CPU-GPU. Ainsi, il est parfaitement possible d'écrire le résultat d'un rendu réparti entre plusieurs CPU et GPU directement dans une image dans l'espace mémoire d'un des GPU, pour un affichage via openGL par exemple. Il faut toutefois garder à l'esprit que la quantité de mémoire disponible sur un GPU est généralement bien plus limitée que la mémoire centrale, ce qui désigne celle-ci comme mémoire de résidence privilégiée afin de pouvoir gérer de gros volumes de données. C'est en effet grâce à cela que notre interface de parallélisation générique nous permet non seulement d'ouvrir, mais aussi de tirer parti des GPU sur des modèles imposants, dont la totalité des données ne tiendrait pas sur le GPU. En outre, ce contrôle global des tampons mémoire a permis la mise en place d'un mécanisme de sécurisation du lancement des tâches sur les unités de calcul. En effet, en dehors des bogues de programmation dans le noyau de calcul lui-même, le manque de mémoire résidente au sein d'un GPU est une cause probable d'échec d'exécution d'une sous-tâche de calcul. Prévoir cet échec et mettre en place des alternatives paramétrables permet d'éviter un crash GPU qui peut se révéler extrêmement problématique car il provoque potentiellement la perte de l'affichage chez le client. Le dernier point, et non des moindres, concerne la maintenance du code. Grâce à notre interface de parallélisation, le code CPU et le code GPU sont identiques. Cela implique une simplification non négligeable de la maintenance. En effet, les évolutions du code, que ce soit suite à du debug ou du développement, sont ainsi automatiquement compilées et exécutables pour le CPU et le GPU. Cela assure une non-divergence de codes entre les périphériques. Maintenir deux versions de chaque algorithme parallélisable afin de pouvoir s'adapter aux périphériques disponibles se révèle rapidement problématique et source de perte de temps lors de l'exploitation de l'application. Cela permet tout simplement de réduire le taux de duplication de codes, une règle de base pour la bonne marche d'un projet industriel.

1

Conclusion

1.1 Résumé des contributions

Nos contributions s'étalent sur l'ensemble des éléments clefs pour l'accélération de l'algorithme du lancer de rayons. La première permet d'accélérer la construction d'une structure d'accélération de haute qualité : un KD-Tree optimal. Cette méthode permet d'accélérer la construction du KD-Tree sur CPU, et se compare avantageusement aux méthodes précédemment publiées. Elle est de plus capable de profiter de l'activation d'un GPU, afin d'améliorer encore une fois les performances.
Notre seconde contribution concerne la répartition des tâches de traversée du KD-Tree optimal entre les CPU et GPU. Celle-ci est transparente pour l'utilisateur et permet de faire participer chacun des périphériques de manière parallèle, à hauteur de ses capacités de calcul, afin de minimiser les temps de calcul.
Notre troisième contribution concerne le test d'intersection rayon-triangle appliqué au KD-Tree. Nous avons évalué les approches les plus souvent mentionnées dans la littérature afin de retenir celle qui est la plus adaptée à notre environnement. Nos mesures nous ont également permis d'arbitrer entre consommation mémoire et facteur d'accélération issu de pré-calculs géométriques. Notre organisation mémoire mixte nous permet de maximiser l'utilisation des instructions SIMD des CPU lors de l'intersection tout en maitrisant la consommation mémoire et préservant les performances sur GPU.
Enfin, notre dernière contribution est plus générale, et permet de répartir automatiquement de manière parallèle et transparente des calculs entre les cœurs CPU et les GPU activés. Même si son application est plus générale, l'objectif premier de cette interface de parallélisation est de permettre aux utilisateurs d'accélérer leurs calculs physiques, associés au relations de visibilité calculées à l'aide de notre requête d'intersection.

1.2 Objectifs et contraintes

Notre objectif principal, à savoir la performance, a été atteint au vu des différentes mesures et comparaisons effectuées. Pour ce qui est de l'adaptabilité à l'existant, nos différents niveaux d'interface nous permettent d'intégrer notre bibliothèque en plusieurs étapes, en garantissant de hautes performances pour chacun des niveaux d'intégration. En effet, si notre interface de parallélisation peut nécessiter un certain temps pour le portage à partir d'une application existante, il est possible de procéder simplement au branchement de notre requête d'intersection en lieu et place de l'existant dans un premier temps et de constater un premier gain de performances. Cette requête d'intersection se connecte de manière non intrusive au logiciel hôte, et n'impose aucune représentation mémoire pour la géométrie.
Les ressources sont gérées de manière transparente, et notre bibliothèque de lancer de rayons est capable de tirer parti de chacun des périphériques éligibles de bout en bout lors de l'exécution d'une session de lancer de rayons. Notre chaîne de compilation fonctionne sur les systèmes d'exploitation que nous avions ciblés, et des outils de développement compatibles GPU sont disponibles sur chacune de ces plateformes.
De plus, notre bibliothèque de lancer de rayons n'est pas optimisée pour un type particulier de tâches, et se veut très performante dans le cas général. Les problématiques de gestion des périphériques hétérogènes sont entièrement masquées et il est tout à fait possible de ne jamais se préoccuper des périphériques actifs lors des simulations. En effet, par défaut, les calculs seront répartis sur le maximum de périphériques de manière optimale. Les problématiques de gestion de taille des mémoires associées aux différents périphériques sont masquées, et l'utilisateur n'a pas à intervenir si un modèle se révélait trop gros pour tenir dans la mémoire d'un GPU. Celui-ci serait désactivé pour cette session et redeviendrait éligible dès lors qu'une tâche satisferait les contraintes mémoires d'exécution. Cette alternative se révèle robuste, même si elle peut conduire à constater un gain nul à l'activation du GPU sur des jeux de données massifs.
Notre dernière contrainte, à savoir faciliter la maintenance du code, est assurée par notre manière de forcer l'existence d'un seul et même code pour les CPU et les GPU. Cette manière de déclarer les tâches parallèles assure une cohérence totale entre les versions CPU et GPU du code. De plus, la présence d'outils sérieux de debug et de développement autour de la plateforme CUDA sur tous nos systèmes d'exploitation cibles assure une facilité d'évolution et de maintenance des codes générés à l'aide de notre bibliothèque de lancer de rayons.

1.3 Travaux futurs

Plusieurs pistes restent à explorer autour de nos contributions. La construction du KD-Tree par exemple, est la seule de nos contributions qui n'utilise pas le même algorithme sur CPU et GPU. Et même s'il collabore, le côté hybride n'est pas aussi poussé que dans nos autres contributions. Il serait intéressant d'utiliser notre interface de parallélisation pour faire intervenir le CPU et le GPU parallèlement dans chacune des deux étapes, à savoir le traitement des nœuds fins et le traitement des nœuds larges.
La partie répartition des calculs utilise pour l'instant des seuils fixes sur les nombres de données qu'elle demande à traiter par passes. Ces seuils permettent d'obtenir des performances idéales sur nos configurations de tests. Il serait toutefois sûrement judicieux de tenir à jour une affinité de périphérique avec une tâche afin d' ajuster dynamiquement ces seuils lors des exécutions successives.
Notre organisation des données d'intersection rayon/triangle doit pouvoir permettre de mettre en place efficacement un mécanisme de mail-boxing, tel que décrit par Wald dans [Wal04] ou encore par Shevtsov dans [SSKN07]. Le mail-boxing consiste à mettre en cache les résultats d'intersections rayon/triangle infructueux. Les triangles pouvant être présents dans plusieurs feuilles voisines du KD-Tree, cela permet de ne pas tester ces mêmes intersections rayon/triangle infructueuses dans toutes les feuilles pour un même rayon. Il se trouve que notre organisation des données d'intersection rayon/triangle cloisonne les triangles en fonction de leur appartenance totale à une feuille. Elle permet donc de remplir le cache d'intersections déjà calculées seulement pour les triangles ayant une chance d'être intersectés à nouveau dans les feuilles voisines. Cela devrait avoir pour effet d'optimiser le taux de succès du cache d'intersections.
Une des particularités de notre interface de lancer de rayons, consiste à laisser la possibilité de déclarer des données résidentes dans plusieurs espaces mémoire différents. Cela permet de faire résider les données d'une application entièrement dans l'espace GPU par exemple. Certaines applications obtiennent des facteurs d'accélération impressionnants mais résider totalement en mémoire GPU implique de grosses contraintes en termes d'espace mémoire. En effet, un GPU dispose généralement de beaucoup moins de mémoire qu'il n'y en a en mémoire centrale. Dans un soucis de robustesse, un GPU peut se retrouver mis à l'écart dès lors qu'une tâche utilise trop de mémoire. Si cela est toujours plus satisfaisant que de provoquer un crash du GPU, ce n'est pas totalement satisfaisant dans le sens où le GPU n'apporte aucun gain. Il serait donc intéressant de faire tendre notre gestion des zones mémoires vers un modèle dit "out-of-core". Ce type de modèle est capable de gérer dynamiquement une requête d'accès à n'importe quelle zone mémoire, sur n'importe quel périphérique durant l'exécution. Globalement, une unité de calcul se

voit affecter un cache mémoire résident d'une taille en rapport avec ses capacités, et un gestionnaire mémoire est chargé de rapatrier intelligemment les données dans ce cache au gré des requêtes mémoires. Dans le domaine du GPU, nous avons exploré quelques pistes en terme de modèle "out-of-core" en nous basant sur les travaux de Stuart [SCO10] et de Crassin [Cra11]. Nos expérimentations se sont pour l'instant heurtées à des problèmes de performances, ou de limitations imposées sur certains systèmes d'exploitations. En effet, les requêtes mémoires telles que présentées par Stuart monopolisent un thread CPU pour les allocations et les copies mémoires, et génèrent des latences inacceptables à l'échelle des threads CUDA. Quant à la méthode proposée par Crassin, même si ses performances sont satisfaisantes, elle prend comme hypothèse que la mémoire zéro-copie de l'API CUDA est disponible de manière quasi-illimitée. Ceci est le cas :

- sur tous les systèmes d'exploitation si la carte de calcul utilisée est une déclinaison professionnelle, et n'utilise pas de sortie vidéo.

- sur les systèmes Linux x64 et Microsoft Windows XP x64.

Ne pouvant ni restreindre les systèmes d'exploitations supportés, ni cibler exclusivement les GPU professionnels dédiés au calcul, le modèle out-of-core reste un sujet ouvert pour nos applications.
Enfin, nous sommes actuellement en train d'explorer la possibilité de mise en place de plusieurs back-end à notre interface de parallélisation générique. En effet, le modèle de parallélisme utilisé permet de répondre efficacement aux contraintes de programmation GPU via CUDA, qui sont très restrictives. Il est donc possible de s'adapter assez facilement à des modèles de programmation parallèle proches, ou moins contraignants. C'est pourquoi une version OpenCL est en cours de réalisation via le SDK AMD et ses extensions C++, et qu'une version OpenMP est envisagée.

Bibliographie personnelle

[JBR+11] D. Joseph, D. Braga, J. P. Roccia, E. Gengembre, A. Thuillier, B. Ladevie, and J. J. Bezian. Calcul des flux solaires pour le Bâtiment par méthode de Ray Tracing. *Société Française de Thermique*, 2011.

[RPC12a] J. P. Roccia, M. Paulin, and C. Coustet. Hybrid CPU/GPU KD-Tree Construction for Versatile Ray Tracing. In *Proceedings EUROGRAPHICS Short Papers*, pages 13–16, 2012.

[RPC+12b] J. P. Roccia, B. Piaud, C. Coustet, C. Caliot, E. Guillot, G. Flamant, and J. Delatorre. SOLFAST, a Ray-Tracing Monte-Carlo software for solar concentrating facilities. *Journal of Physics Conference Series*, 369(1) :012029, June 2012.

Bibliographie

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[ARBJ03] Sameer Agarwal, Ravi Ramamoorthi, Serge Belongie, and Henrik Wann Jensen. Structured importance sampling of environment maps. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 605–612, New York, NY, USA, 2003. ACM.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9) :509–517, September 1975.

[Cad08] Gilles Cadet. *Accélération de le Requête d'Intersection par la Réorganisation des Rayons*. PhD thesis, SUPAERO, TOULOUSE, 2008.

[CKL+] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. Source code - "parallel sah k-d tree construction". https ://github.com/bchoi/ParKD/.

[CKL+10] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. Parallel sah k-d tree construction. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 77–86, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[com] Caustic Graphics company. Ray tracing accelerator boards. https ://www.caustic.com/.

[Cra11] Cyril Crassin. *GigaVoxels : A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, UNIVERSITE DE GRENOBLE, July 2011. English and web-optimized version.

[CW88] JohnG. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4 :65–83, 1988.

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, pages 15–22, New York, NY, USA, 2005. ACM.

[FTI88] A. Fujimoto, Takayuki Tanaka, and K. Iwata. Tutorial : computer graphics ; image synthesis. chapter ARTS : accelerated ray-tracing system, pages 148–159. Computer Science Press, Inc., New York, NY, USA, 1988.

[GL10] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum*, 29(2), May 2010.

[GP89] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Comput. Graph. Appl.*, 9(6) :12–26, November 1989.

[GP90] StuartA. Green and DerekJ. Paddon. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, 6 :62–73, 1990.

[GROa] KHRONOS GROUP. Opencl - the open standard for parallel programming of heterogeneous systems. http ://www.khronos.org/opencl/.

[Grob] OpenACC Group. Openacc application program interface. http ://www.openacc-standard.org/.

[Har07] M. Harris. Optimizing parallel reduction in cuda. *http ://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf*, 2007.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HB10] Jared Hoberock and Nathan Bell. Thrust : A parallel template library, 2010. Version 1.3.0.

[INTa] INTEL. Embree - photo-realistic ray tracing kernels. http ://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels-0.

[INTb] INTEL. INTEL Developer Zone. http ://software.intel.com/fr-fr.

[INTc] INTEL. INTEL Intrinsic guide. http ://software.intel.com/fr-fr/avx.

[Jen95] Henrik Wann Jensen. Importance driven path tracing using the photon map. In *in Eurographics Rendering Workshop*, pages 326–335. Springer-Verlag, 1995.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[Kar12] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *High Performance Graphics*, pages 33–37, 2012.

[KS09] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 23–28, New York, NY, USA, 2009. ACM.

[LD08] Ares Lagae and Philip Dutré. Compact, fast and robust grids for ray tracing. SIGGRAPH 2008 Talk, SIGGRAPH 2008, Los Angeles, USA, August 2008.

[LRR04] Jason Lawrence, Szymon Rusinkiewicz, and Ravi Ramamoorthi. Efficient brdf importance sampling using a factored representation. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 496–505, New York, NY, USA, 2004. ACM.

[Mac08] Jon Macey. Ray-tracing and other rendering approaches. http ://nccastaff.bournemouth.ac.uk/jmacey/CGF/slides/RayTracing4up.pdf, 2008.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3) :153–166, May 1990.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics, gpu, and game tools*, 2(1) :21–28, 1997.

[MU49] Nicholas Metropolis and Stanislaw M. Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247) :335–341, September 1949.

[NVIa] NVIDIA. CUDA Toolkit Documentation. http ://-docs.nvidia.com/cuda/index.html.

[NVIb] NVIDIA. CUDA Zone - NVIDIA Developer Zone. http ://developer.nvidia.com/category/zone/cuda-zone.

[NVIc] NVIDIA. NVIDIA CUDA Compiler Driver NVCC. http ://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[ORM08] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. Large Ray Packets for Real-time Whitted Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing (IRT)*, pages 41–48, Aug 2008.

[PBPP11] Anthony Pajot, Loic Barthe, Mathias Paulin, and Pierre Poulin. Representativity for robust and adaptive multiple importance sampling. *IEEE Transactions on Visualization and Computer Graphics*, 17(8) :1108–1121, August 2011.

[SCO10] Jeff A. Stuart, Michael Cox, and John D. Owens. Gpu-to-cpu callbacks. In *UCHPC 2010 : Proceedings of the Third Workshop on UnConventional High Performance Computing (Euro-Par 2010 Workshops)*, UCHPC 2010 : Proceedings of the Third Workshop on UnConventional High Performance Computing (Euro-Par 2010 Workshops). Springer, August 2010.

[Sho98] Ken Shoemake. Plücker coordinate tutorial. Ray Tracing News, 1998.

[SSKN07] Maxim Shevtsov, Alexei Soupikov, Alexander Kapustin, and Nizhniy Novorod. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *In procedings of GraphiCon'2007*, Moscow, Russia, June 2007.

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor : a hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 27–36, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[tea] Inria RUNTIME team. Starpu : A unified runtime system for heterogeneous multicore architectures. http ://runtime.bordeaux.inria.fr/StarPU/.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[WH06] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, September 2006.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6) :343–349, June 1980.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS)*, 20(3) :153–164, 2001.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, page 126, 2008.

Résumé

Depuis son introduction en 1968 par Arthur Appel, le lancer de rayons est un domaine de recherche privilégié au sein de la communauté informatique. En effet, il permet d'obtenir une grande précision pour simuler des phénomènes de rayonnements physiques, et ce dans des domaines aussi variés que la thermique, l'optique ou encore l'acoustique.
Toutefois, malgré sa relative simplicité, celui-ci pose de multiples problèmes lorsqu'il s'agit de le rendre performant. Avec une approche naïve, la requête d'intersection entre un rayon et une primitive géométrique représente environ 90% du temps de calcul. Divers mécanismes permettent d'accélérer la requête d'intersection géométrique, parmi lesquels l'optimisation de la méthode de calcul d'intersection rayon/triangle elle-même, mais aussi la construction d'une structure spatiale d'accélération. Celles-ci permettent de réduire le nombre de requêtes d'intersection à exécuter en ciblant plus rapidement les groupes de primitives géométriques ayant de fortes probabilités d'être intersectés par le rayon.
Les architectures matérielles modernes proposent un parallélisme croissant, symbolisé par une augmentation du nombre de cœurs de calculs disponibles, que ce soit sur CPU avec les hexa-cœurs ou sur GPU avec leurs centaines de cœurs.
L'algorithme du lancer de rayons se doit de tirer parti de cette puissance de calcul disponible, et se prête plutôt bien à la parallélisation. En effet, au lieu de traiter les rayons un à un de manière séquentielle, les traiter parallèlement permet d'augmenter significativement les performances.
Les applications modernes du lancer de rayons, à base de Monte-Carlo par exemple, requièrent de lancer un nombre important de rayons dans la même scène. Cependant, la cohérence des rayons est rarement suffisante pour permettre une mise en paquet simple et efficace. Néanmoins, toutes les applications ne nécessitent pas un grand nombre de rayons. Sélectionner un objet dans une scène ne requiert, par exemple, qu'un seul rayon et il est donc important de pouvoir lancer efficacement des rayons isolés.
Nos contributions s'étalent sur l'ensemble des éléments clefs pour l'accélération de l'algorithme du lancer de rayons. La première permet d'accélérer la construction d'une structure d'accélération : un KD-Tree de haute qualité. Cette méthode permet d'accélérer la construction du KD-Tree sur CPU, et se compare avantageusement aux méthodes précédemment publiées. Elle est de plus capable de profiter de l'activation d'un GPU, afin d'améliorer encore une fois les performances.
Notre seconde contribution concerne la répartition des tâches de traversées du KD-Tree

optimal entre les CPU et GPU. Celle-ci est transparente pour l'utilisateur et permet de faire participer chacun des périphériques de manière parallèle, à hauteur de ses capacités de calculs, afin de minimiser les temps de calculs.
Notre troisième contribution concerne le test d'intersection rayon-triangle appliqué au KD-Tree optimal. Notre organisation mémoire nous permet de maximiser l'utilisation des instructions SIMD des CPU lors de l'intersection tout en maitrisant la consommation mémoire et préservant les performances sur GPU.
Enfin, notre dernière contribution est plus générale, et permet de répartir automatiquement de manière parallèle et transparente des calculs entre les cœurs CPU et les GPU activés. Même si son application est plus générale, l'objectif premier de cette interface de parallélisation est de permettre aux utilisateurs d'accélérer leurs calculs physiques, associés au relations de visibilités calculés à l'aide de notre requête d'intersection.

Abstract

Since its introduction in 1968 by Arthur Appel, ray tracing is a prime area of research within the computing community. Indeed, it allows to obtain high accuracy to simulate physical phenomena of radiation, and in areas as diverse as heat transfer, optics , acoustics.
However, despite its relative simplicity, it poses many problems when it comes to the performance. With a naive approach, the application of intersection between a ray and a geometric primitive is about 90 % of the computation time. Various mechanisms allow to accelerate the computation of geometric intersection, including the optimization of the method for calculating ray/triangle intersection itself, but also the construction of a spatial acceleration structure. These can reduce the number of ray/triangle intersection queries to be executed by first targeting groups of geometric primitives with a high probability of being intersected by the ray.
Modern hardware architectures offer a growing parallelism, symbolized by a growing number of available cores. Whether on CPU with hexa-core or GPU with hundreds of cores. The ray tracing algorithm should take advantage of the available computing power, and lends itself quite well to parallelization. Indeed, instead of treating the rays one by one sequentially, parallel processing can significantly increase performance.
Modern applications of ray tracing, based on Monte Carlo for example, require to trace a large number of rays in the same scene. However, the coherency of the rays is rarely sufficient for bundling them simply and effectively. However, all applications do not require a large number of rays. To pick an object in a scene, for example, only requires a single

ray and it is therefore important to effectively trace single rays.
Our contributions are spread over all the main elements for ray tracing algorithm acceleration. The first one speeds up the construction of an high quality acceleration structure : an optimal KD-Tree. This method can speed up the construction of the KD-Tree on CPU, and outperforms previously published method. It is also able to take advantage of enabling a GPU to improve performance even further.
Our second contribution concerns the distribution of the optimal KD-Tree traversal steps between CPU and GPU. This is transparent to the user and allows each device to participate, up to its computing capacity in parallel to minimize computation time.
Our third contribution concerns the ray-triangle intersection test applied to the optimal KD-Tree. Our memory organization allows us to maximize the use of the SIMD instructions on CPU in the intersection test while controlling memory usage and maintaining performance on GPU.
Finally, our last contribution is more general, and spreads automatically and transparently parallel computations between enabled CPU cores and GPU. Even if its application is more general, the goal of this parallelization interface is to allow users to accelerate physics calculations, associated to the visibilities calculated using our primitive intersection.