



Université
de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par **l'Institut Supérieur de l'Aéronautique et de l'Espace**
Spécialité : STIC Sûreté de logiciel et calcul de haute performance

Présentée et soutenue par **Lancelot PERROTTE**
le 10 juin 2011

Algorithmes robustes de lancer de rayons pour calcul de dose interactif

JURY

M. Dominique Houzet, président, rapporteur
M. Bruno Arnaldi, rapporteur
M. Laurent Chodorge
M. Cheikh M'Backé Diop
M. Bernard Lécussan, directeur de thèse
M. Mathias Paulin

École doctorale : **Mathématiques, informatique et télécommunications de Toulouse**
Unité de recherche : **Équipe d'accueil ISAE-ONERA CSDV**
Directeur de thèse : **M. Bernard Lécussan**

Remerciements

JE souhaiterais tout d'abord remercier chaleureusement Laurent Chodorge, qui m'a encadré pendant ces trois années de thèse au CEA. Merci à lui de m'avoir proposé cette thèse passionnante et de m'avoir toujours soutenu et encouragé. Je n'oublierai pas non plus les nombreuses discussions techniques dans son bureau, sources de très nombreuses pistes de travail. Je tiens aussi à joindre à ces remerciements mon directeur de thèse Bernard Lécussan pour ses précieux conseils et sa disponibilité dans les moments décisifs.

Je voudrais également remercier les membres de mon jury de thèse, en particulier les rapporteurs de cette thèse, Bruno Arnaldi et Dominique Houzet, ce dernier ayant en outre accepté de présider mon jury, pour leur lecture attentive de mon manuscrit de thèse et leurs remarques pertinentes. Enfin, je souhaiterais remercier Cheikh M'Backé Diop et Mathias Paulin pour avoir examiné mon travail.

Je tiens maintenant à remercier très vivement Guillaume Saupin pour son aide technique et pour sa gentillesse tout au long de ces trois années. Il est difficile pour moi d'imaginer ce que cette thèse aurait pu être sans son aide tant elle a été essentielle. De même, je remercie Bruno Bodin pour ses nombreux et précieux conseils lors de la deuxième partie de ma thèse. Je n'oublie pas non plus ses multiples relectures attentives, qui m'ont été extrêmement utiles.

De manière générale, je souhaiterais remercier l'équipe du Laboratoire de Simulation Interactive de Fontenay-aux-Roses : j'ai vraiment apprécié au cours de ces quatre années évoluer dans une atmosphère de travail aussi saine et détendue. En particulier, je remercie ceux qui étaient présents lors de mon arrivée au LSI pour leur accueil chaleureux, mais je n'oublie pas ceux qui sont arrivés après moi, dont un certain tandem italo-toulousain qui a rendu la fin de ma thèse bien plus agréable.

Enfin, je souhaiterais remercier tous ceux qui rendent au quotidien ma vie personnelle plus agréable, qui m'ont offert un grand soutien et des moments de détente essentiels. Tout d'abord, je voudrais remercier ma famille pour l'enfance formidable qu'elle m'a offerte et pour tout ce qu'elle continue à m'apporter. Je remercie aussi mes amis, qui sont essentiels dans ma vie et avec qui je continue à passer tant de bons moments. Pour finir, je souhaiterais remercier Paloma pour m'avoir soutenu et encouragé dans les nombreux moments de doute que j'ai connus. Ce manuscrit n'aurait sans doute pas pu exister sans sa présence.

Paris, 26 juillet 2011.

Table des matières

Table des matières	i
Liste des figures	v
Liste des tableaux	vii
Introduction	1
1 État de l'art	3
1.1 Le calcul du débit de dose	4
1.1.1 L'ingénierie de la radioprotection	4
1.1.2 Méthodes de calcul de la propagation des rayonnements	5
1.1.3 Méthode d'atténuation en ligne droite avec facteurs d'accumulation	6
1.2 Exploitation des cartes graphiques	9
1.2.1 La rastérisation	10
1.2.1.1 Présentation	10
1.2.1.2 Analogie avec la requête exhaustive d'intersections	10
1.2.1.3 Détournement des unités de rastérisation	11
1.2.2 Calcul générique sur Processeurs graphiques (GPGPU)	13
1.2.2.1 Structure des processeurs de cartes graphiques	13
1.2.2.2 Présentation de l'API CUDA	14
1.2.2.3 Algorithmes parallèles récurrents	15
1.3 Lancer de rayons	16
1.3.1 Présentation de l'algorithme	17
1.3.2 Les structures d'accélération	19
1.3.2.1 Les grilles	19
1.3.2.2 Le kd-tree	20
1.3.2.3 Le BVH	21
1.3.2.4 Comparaisons entre le kd-tree et le BVH	21
1.3.3 Critères de qualité d'un arbre kd-tree ou BVH	22
1.3.4 Techniques d'accélération pour des groupes de rayons cohérents	24
1.3.4.1 Sur CPU	24
1.3.4.2 Techniques d'accélération sur GPU	25
1.3.5 Gestion des scènes dynamiques	26
1.3.5.1 Adapter la structure d'accélération	27
1.3.5.2 Construction optimisée de structures d'accélération sur CPU	27
1.3.5.3 Construction optimisée de structures d'accélération sur GPU	28
1.3.6 La grille en perspective	30
1.3.6.1 Présentation de la grille en perspective	30
1.3.6.2 Les avantages de la grille en perspective pour notre algorithme	30

TABLE DES MATIÈRES

1.4 Conclusion	32
2 Triangle-tracing	33
2.1 Modifications de l'algorithme traditionnel	34
2.1.1 Description de l'organisation des données	34
2.1.2 Gestion de l'espace mémoire	35
2.1.3 Efficacité de l'algorithme de construction de la grille	35
2.2 Construction de la grille de rayons	37
2.2.1 Identification de la position des rayons dans la grille en perspective	38
2.2.2 Regroupement des rayons par indice de cellule	38
2.2.3 Décompte du nombre de rayons par cellule non vide	39
2.2.4 Génération du nombre de rayons par cellule	39
2.2.5 Génération de la version finale de <i>débutCelluleIds</i>	39
2.3 Triangle-tracing	40
2.3.1 Préparation du stockage des informations associées à chaque ligne	40
2.3.2 Stockage des informations sur les lignes associées aux triangles	42
2.3.3 Préparation du stockage des indices de cellule	42
2.3.4 Stockage des couples triangle/cellule	42
2.3.5 Préparation du stockage des résultats des tests d'intersections	42
2.3.6 Génération des couples d'indices rayon/triangle à tester	43
2.3.7 Réalisation des tests d'intersection rayon/triangle	43
2.3.8 Génération des résultats finals	43
2.3.9 Tri des intersections	43
2.4 Résultats	46
2.4.1 Temps de construction des deux grilles	47
2.4.2 Temps de calcul global	49
2.4.3 Comparaison avec l'algorithme traditionnel	51
2.4.4 Variations de la résolution de la grille en perspective	53
2.5 Sources d'optimisation supplémentaires	54
2.5.1 Construction de la grille de rayons	55
2.5.2 Rastérisation des triangles	55
2.5.3 Meilleure utilisation de la mémoire	57
3 Gestion de la précision	63
3.1 Problèmes de précision posés par l'arithmétique flottante	64
3.1.1 Difficultés liées à l'arithmétique flottante près des frontières des triangles	64
3.1.2 Choix d'un autre test d'intersection	66
3.1.3 Déterminant 2D	66
3.1.4 Algorithmes adaptatifs pour l'arithmétique exacte	67
3.2 Résolution des problèmes de précision sur GPU	68
3.2.1 Difficultés à porter les algorithmes adaptatifs de Shewchuk sur GPU	68
3.2.2 Solution proposée	69
3.2.3 Tests d'intersection 3D	70
3.3 Utilisation de la topologie du maillage	71
3.3.1 Principes	71
3.3.2 Implémentation	71
3.3.3 Cas des rayons passant près de la frontière d'un objet	73
3.3.4 Transfert des informations sur la nature des intersections vers le CPU	73
3.4 Problèmes posés par la projection des triangles	74

3.4.1	Le clipping	75
3.4.2	Changement de topologie	76
3.4.3	Clipping sur GPU	77
3.4.4	Prise en compte de la topologie du maillage initial	79
3.5	Résultats	80
3.5.1	Robustesse des calculs	80
3.5.2	Mesures de performance	81
3.5.3	Surconsommation de mémoire	84
4	Stratégies de traitement de multiples paquets de rayons	87
4.1	Introduction	88
4.1.1	Configurations de test	88
4.1.2	Les différentes stratégies	89
4.1.2.1	Stratégie “globale”	89
4.1.2.2	Stratégie “locale”	92
4.2	Optimisation des calculs pour la stratégie “locale”	93
4.2.1	Les différentes étapes	93
4.2.2	Optimisation de l’étape de transformation des triangles	93
4.2.3	Réduction du nombre de triangles étudiés	96
4.2.4	Impact de la phase de tri/transfert des données	100
4.3	Résultats	101
4.3.1	Stratégie “globale”	101
4.3.2	Stratégie “locale”	103
4.3.3	Comparaison des deux stratégies	105
4.3.4	Augmentation du nombre de paquets	108
4.3.5	Performances globales de la stratégie “locale”	110
4.4	Discussion	112
4.4.1	Optimisations supplémentaires de l’algorithme actuel	112
4.4.2	Vers une structure hiérarchique ?	113
Conclusion et perspectives		117
A Appendix		121
A.1	L’équation de transport de Boltzmann	121
A.1.1	Sections efficaces	121
A.1.2	Formulation de l’équation	122
A.2	Configuration matérielle	125
A.3	Publications associées à la thèse	127
Bibliographie		129

Liste des figures

1.1	Interactions de rayonnements γ avec la matière	6
1.2	Parcours d'un rayon γ à l'intérieur de plusieurs objets successifs	8
1.3	Paquet de rayons cohérents vu en deux dimensions	9
1.4	Schéma simplifié d'une carte graphique	13
1.5	Algorithme récursif de lancer de rayons	18
1.6	Parcours de rayon dans une grille régulière	19
1.7	Kd-tree sur une scène simple	20
1.8	BVH sur une scène simple	21
1.9	Grille en perspective vue en trois dimensions	30
1.10	Comparaison entre le BVH et la grille en perspective	31
2.1	Organisation des données pour décrire la grille en perspective	35
2.2	Grille en perspective sur une scène simple	37
2.3	Description de l'algorithme de construction de la grille en perspective indexant les rayons	38
2.4	Description de l'algorithme de lancer de triangles	41
2.5	Stratégie de rastérisation <i>scanline</i> sur deux triangles différents	41
2.6	Méthode de stockage des informations conjointes indice de rayon/coordonnée d'intersection	44
2.7	Visualisation "radio" de certaines scènes usuelles	47
2.8	Performances détaillées de l'algorithme de lancer de triangles	50
2.9	Évolution des performances de l'algorithme de lancer de triangles en fonction des variations de la résolution de la grille en perspective utilisée, sur la scène Conference	53
2.10	Évolution des performances de l'algorithme de lancer de triangles en fonction des variations de la résolution de la grille en perspective utilisée, sur la scène Sully .	54
2.11	Principes de la rastérisation hiérarchique	56
2.12	Réorganisation des écritures en mémoire pour une meilleure cohérence d'accès	58
2.13	Stratégie efficace de remplissage du vecteur de couples cellule/triangle	58
3.1	Test d'intersection rayon-triangle optimisé	65
3.2	Déterminants 2D associés à un triangle	67
3.3	Validation de la cohérence des intersections au niveau d'un sommet	72
3.4	Intersection sur la frontière d'un objet	73
3.5	Opération de clipping utilisant un plan $Z = \epsilon$	75
3.6	Découpe d'un triangle suivant les quatre plans du cône de vision	76
3.7	Algorithme de Sutherland-Hodgman	77
3.8	Modification de la topologie d'un maillage par l'étape de clipping	78
3.9	Clipping "integral" des triangles de la scène	78
3.10	Répartition des temps d'exécution suivant les différentes phases propres à l'algorithme de gestion de précision (tests d'intersections 3D)	82

3.11 Augmentation des problèmes de précision lors de l'éloignement du point de départ d'un rayon	83
4.1 Visualisation des interactions rayon-matière sur la scène nuclearCase	88
4.2 Schéma de deux des six grilles en perspective couvrant l'intégralité de l'espace autour du point de mesure du débit de dose	89
4.3 Arête de deux triangles conjoints commune à deux grilles en perspective	90
4.4 Mauvaise gestion d'un paquet de rayons de faible volume par la stratégie "globale"	91
4.5 Point de mesure entouré par diverses sources de radiations	92
4.6 Clipping d'un triangle donnant lieu à sept nouveaux sommets de triangles	94
4.7 Répartition des performances lors de la phase de transformation des triangles (sans tests de "rejet rapide")	95
4.8 Tests de "rejet rapide" des triangles	96
4.9 Plan "de profondeur maximale" associé à un faisceau de rayons	97
4.10 Répartition des performances lors de la phase de transformation des triangles (avec tests de "rejet rapide")	99
4.11 Performances de la stratégie "globale" lors du traitement de 4 cuves de grande dimension	102
4.12 Performances de la stratégie "globale" lors du traitement de 4 cuves de dimension "moyenne"	102
4.13 Performances de la stratégie "globale" lors du traitement de 4 cuves de faible dimension	102
4.14 Performances de la stratégie "locale" lors du traitement de 4 cuves de grande dimension	104
4.15 Performances de la stratégie "locale" lors du traitement de 4 cuves de dimension "moyenne"	104
4.16 Performances de la stratégie "locale" lors du traitement de 4 cuves de faible dimension	104
4.17 Comparaison des stratégies "globale" et "locale" lors du traitement de 4 cuves de grande dimension	105
4.18 Comparaison des stratégies "globale" et "locale" lors du traitement de 4 cuves de dimension "moyenne"	105
4.19 Comparaison des stratégies "globale" et "locale" lors du traitement de 4 cuves de faible dimension	106
4.20 Comparaison des stratégies "globale" et "locale" lors du traitement de 8 cuves de dimension "moyenne"	108
4.21 Comparaison des stratégies "globale" et "locale" lors du traitement de 12 cuves de faible dimension	108
4.22 Évolution du ratio nblIntersections/seconde en fonction de la densité des paquets de rayons	111
4.23 Paquets de rayon se "chevauchant"	114

Liste des tableaux

2.1	Temps de construction de la grille en perspective indexant les rayons de la scène	47
2.2	Temps de génération des couples cellule/triangle pour une scène vue en perspective	48
2.3	Performances de l'algorithme de lancer de triangles	49
2.4	Statistiques de performance de l'algorithme global de triangle-tracing, lors du lancer de 1024 * 1024 rayons primaires	51
2.5	Comparaison des performances obtenues entre les versions classique et inversée de l'algorithme global sur la grille en perspective	51
2.6	Performances détaillées de la phase d'initialisation, pour l'algorithme traditionnel de lancer de rayons	52
3.1	Modifications des performances après gestion des problèmes de précision	81
3.2	Comparaison de la précision des versions 2D et 3D des tests d'intersections rayon-triangle	83
3.3	Surconsommation de mémoire due aux algorithmes de gestion de précision	84
4.1	Performances comparées des tests de "rejet rapide" sur 4 cuves de grande dimension	98
4.2	Performances comparées des tests de "rejet rapide" sur 4 cuves de dimension "moyenne"	98
4.3	Performances comparées des tests de "rejet rapide" sur 4 cuves de faible dimension	98
4.4	Impact sur les performances de l'utilisation d'un plan "de profondeur maximale" .	99
4.5	Impact sur les performances de l'utilisation de transferts de mémoire asynchrones	100
4.6	Comparaison du nombre total de triangles traités par chaque stratégie	106
4.7	Comparaison du nombre total de couples triangle-cellule "inutiles" générés par chaque stratégie	107
4.8	Comparaison du nombre de tests effectués en fonction de la stratégie choisie . .	107
4.9	Évolution du ratio nbIntersections/seconde avec la diminution de la densité des paquets de rayons	109
4.10	Performances globales de la stratégie "locale".	110

Introduction

L'actualité récente a replacé le débat sur l'énergie nucléaire sur le devant de la scène. Dans ce contexte difficile, il est plus que jamais nécessaire de travailler à la sécurité des acteurs du monde nucléaire, suivant le principe ALARA (en anglais *As Low As Reasonably Achievable*), sorte de principe de précaution appliqué à la sphère nucléaire. Entre autres, il est essentiel de garantir une exposition minimale aux opérateurs intervenant sur des sites irradiés. Afin de préparer ces interventions sur des scènes de travail, on utilise classiquement des codes de calcul permettant d'évaluer rapidement le débit de dose radioactive reçu par des opérateurs ou par certains matériaux. Ces outils peuvent servir au dimensionnement des espaces de travail, mais aussi à la préparation de scénarios d'intervention (déplacements de l'opérateur dans la scène, positionnement de différents écrans...). Afin d'étudier rapidement divers scénarios, et diverses configurations de scènes, il est essentiel de disposer d'outils permettant un calcul de débit de dose interactif. Il faut cependant impérativement réaliser cette accélération des performances sans sacrifier la précision des calculs mis en place, afin de conserver une marge d'erreur faible sur l'estimation du débit de dose, et de respecter le principe ALARA.

L'objectif de cette thèse est donc de développer des algorithmes géométriques permettant l'évaluation rapide du débit de dose via la méthode classique de radioprotection nommée "méthode d'atténuation en ligne droite avec facteurs d'accumulation". Cette méthode consiste en une modélisation simplifiée du rayonnement, selon laquelle les différentes sources de rayonnement émettent des radiations "en ligne droite", sans subir de déviations à l'occasion de la rencontre de différents matériaux. Ces déviations sont cependant prises en compte via l'introduction de ces "facteurs d'accumulation". Concrètement, l'évaluation du débit de dose via cette méthode est limitée par la lenteur des calculs géométriques devant permettre d'identifier l'ensemble des surfaces intersectées (on considère que le rayonnement traverse chaque surface) le long de chaque rayon arrivant au point de mesure de la dose. Le problème à résoudre est donc un problème de lancer de rayons, sous quelques contraintes spécifiques : tout d'abord, comme nous l'avons dit, la recherche des intersections rayon-triangle est ici exhaustive, à savoir que toutes les intersections le long de chaque rayon de la scène doivent être identifiées, triées, puis conservées en mémoire. En outre, lors du calcul de dose en un point de la scène, l'intégralité des rayons considérés partent d'un point source de radiations pour arriver vers ce point de mesure. Tous ces rayons convergent donc en un même point, ce qui peut permettre diverses optimisations qui seront mises en place dans ce manuscrit. L'objectif de cette thèse est donc de garantir le traitement interactif de ce problème, sur des scènes industrielles importantes (près d'un million de triangles, le maillage résultant pouvant évoluer dans l'espace et subir des modifications de sa topologie au cours du temps), avec un grand nombre de rayons radiatifs (un ou plusieurs paquets de rayons, pour un total proche du million de rayons), tout en garantissant une certaine robustesse des calculs.

Le chapitre 1 constitue une synthèse de l'état de l'art suivant trois axes majeurs. En premier lieu, nous introduisons les méthodes physiques de calcul de dose utilisées dans le domaine de l'ingénierie de la radioprotection. Nous présentons ensuite les processeurs de cartes graphiques (GPUs), que nous souhaitons majoritairement utiliser dans nos travaux, ainsi que les interfaces de programmation associées, comme CUDA. Enfin, nous effectuons

une présentation détaillée de l'algorithme de lancer de rayons et des différentes techniques d'accélération des calculs associés. Cet état de l'art nous permet d'introduire la grille en perspective, structure particulièrement adaptée au traitement de rayons convergeant en un même point, que nous utilisons dans le reste du manuscrit.

Deux algorithmes GPU optimisés de construction et de parcours de grille en perspective sont ensuite détaillés dans le chapitre 2. Nous expliquons aussi dans ce chapitre pour quelles raisons nous avons choisi d'inverser l'ordre habituel d'exécution de l'algorithme de lancer de rayons, et de construire une structure d'accélération indexant les rayons de la scène, et non les triangles. Cette inversion permet une meilleure gestion de la mémoire, et rend possible l'exploitation de scènes aux dimensions très importantes. Elle permet également un gain de temps notable sur l'ensemble de l'algorithme de recherche exhaustive des intersections rayon-triangle à l'intérieur de la scène. Ces diverses optimisations rendent possible le traitement d'un paquet d'un million de rayons au sein d'une scène complexe (statique ou dynamique) en des temps proches de la centaine de millisecondes.

Nous présentons ensuite dans le chapitre 3 des techniques de gestion de la précision des calculs en arithmétique flottante, permettant de contrôler les diverses approximations effectuées lors des tests d'intersections rayon-triangle. Ces méthodes permettent le bon traitement de rayons passant à proximité des frontières d'un triangle (sommets ou arêtes), en donnant la garantie de compter une et une seule intersection avec le maillage de triangles pour de tels rayons. Les techniques proposées reposent sur des algorithmes adaptatifs permettant d'affiner la précision des calculs mis en jeu pour connaître le signe d'un déterminant 2D (ou 3D). La première phase de ces algorithmes est effectuée sur GPU, puis les suivantes, si besoin est, sur CPU. En couplant cette méthode à un algorithme de prise en compte de la topologie du maillage, nous pouvons garantir la robustesse de nos algorithmes sans avoir besoin de régler la valeur de certains coefficients propres à la scène. Ces techniques sont facilement couplées aux algorithmes présentés dans le chapitre 2 et n'entraînent qu'une dégradation de performance minime sur la globalité de l'algorithme (toujours inférieure à 10 % sur nos cas d'études).

Enfin, dans le dernier chapitre de ce manuscrit (chapitre 4), nous présentons l'exploitation de ces algorithmes dans un contexte nucléaire, à savoir la gestion de multiples paquets de rayons, donc de multiples sources de radiations, convergeant vers un même point de mesure. Nous comparons dans ce chapitre deux stratégies différentes de gestion des paquets, la plus rapide consistant à construire une grille en perspective associée à chacun de ces paquets. L'optimisation de cette méthode passe par des algorithmes efficaces (adaptés au GPU) d'évincement des triangles n'intersectant pas un faisceau de rayons donné, que nous présentons aussi dans ce chapitre. Grâce à ces optimisations, nous parvenons à traiter douze paquets de 100 000 rayons chacun, générant plus de 13 millions d'intersections avec une scène composée de plus de 700 000 triangles, en moins d'une demi-seconde. Ces algorithmes sont cependant nettement moins efficaces dans le cadre du traitement de paquets à moins forte concentration de rayons, mais en très grand nombre. Nous présentons à la fin de ce chapitre et en conclusion de cette thèse les perspectives futures de recherche, dans le but, entre autres, de manipuler efficacement des paquets de rayons moins homogènes.

État de l'art

1

Sommaire

1.1	Le calcul du débit de dose	4
1.1.1	L'ingénierie de la radioprotection	4
1.1.2	Méthodes de calcul de la propagation des rayonnements	5
1.1.3	Méthode d'atténuation en ligne droite avec facteurs d'accumulation	6
1.2	Exploitation des cartes graphiques	9
1.2.1	La rastérisation	10
1.2.2	Calcul générique sur Processeurs graphiques (GPGPU)	13
1.3	Lancer de rayons	16
1.3.1	Présentation de l'algorithme	17
1.3.2	Les structures d'accélération	19
1.3.3	Critères de qualité d'un arbre kd-tree ou BVH	22
1.3.4	Techniques d'accélération pour des groupes de rayons cohérents	24
1.3.5	Gestion des scènes dynamiques	26
1.3.6	La grille en perspective	30
1.4	Conclusion	32

Le calcul interactif des quantités de radiations reçues par un opérateur (on peut aussi vouloir calculer le débit de dose reçu par du matériel, des composants électroniques par exemple) se fait en plusieurs étapes essentielles. Premièrement, il faut déterminer un modèle physique suffisamment complexe pour rendre compte de manière fiable du risque encouru par l'opérateur. Nous introduisons donc dans la première partie de ce chapitre (1.1) les notions de physique nucléaire impliquées dans la modélisation de ce phénomène de radiations. Nous présentons ensuite la méthode d'atténuation en ligne droite avec facteurs d'atténuation, classiquement utilisée en radioprotection. Nous en déduisons ainsi la nature des requêtes géométriques auxquelles nous devons répondre afin de calculer le débit de dose recherché.

Nous pouvons alors observer que les requêtes géométriques associées à différents rayons sont indépendantes les unes des autres (même si l'on peut, dans un souci d'optimisation, mettre en commun certains calculs entre différents rayons). Cette propriété rend ce problème intrinsèquement parallèle, et nous incite donc à vouloir exploiter l'architecture hautement parallèle des processeurs de cartes graphiques, ou GPUs (pour Graphics Processing Unit). Dans la partie 1.2, nous commençons par étudier la rastérisation, dont les objectifs sont très proches des nôtres. Nous expliquons pourquoi les implantations matérielles actuelles sur GPU de l'algorithme de rastérisation ne nous permettent pas de résoudre nos problèmes. Il nous faut donc avoir recours à nos propres algorithmes, et les planter sur GPU. Nous étudions ainsi dans un second temps la structure des cartes graphiques actuelles et les interfaces de programmation dédiées au GPU. Ces interfaces se sont considérablement développées au cours des dernières années, rendant la programmation sur GPU beaucoup plus abordable.

La partie suivante (1.3) passe en revue les différents travaux de la communauté du lancer de rayons (ray-tracing), qui cherche à résoudre des problèmes très similaires aux nôtres, bien que nos deux contextes de travail impliquent quelques différences essentielles, entraînant des choix d'algorithmes distincts. Nous introduisons à la fin de cette partie la grille en perspective (en anglais *perspective grid*), dont nous présenterons l'exploitation propre à nos algorithmes dans le chapitre suivant (voir Chapitre 2).

1.1 Le calcul du débit de dose

En premier lieu, nous présentons dans cette partie l'ingénierie de la radioprotection, afin d'expliquer notre contexte de travail. Cette partie est très largement inspirée de l'ouvrage d'Henri Métivier [Mét06] (en particulier les chapitres 6 et 10) sur le sujet.

1.1.1 L'ingénierie de la radioprotection

Il y a plus d'un siècle, au mois de janvier 1896, le physicien allemand Wilhelm Conrad Röntgen découvre les rayons X. Peu de temps après, en mars de la même année, Henry Becquerel décrit pour la première fois la radioactivité. Durant le siècle précédent, les conséquences de ces découvertes ont été immenses : la radiographie médicale a fait considérablement avancer la médecine, tandis que l'énergie nucléaire a permis à un pays comme la France d'acquérir un taux d'indépendance énergétique proche de 50%.

Depuis les travaux de Röntgen, de nombreuses recherches ont montré l'omniprésence des rayonnements naturels dans l'environnement (citons, par exemple, un radionucléide comme l'uranium 238, contenu dans l'écorce terrestre, et dont la désintégration induit des rayonnements α , β , γ ...). En addition à ces rayonnements naturels, de nombreuses installations créées par l'homme génèrent également des rayonnements (accélérateurs de particules, appareils radiologiques, réacteurs nucléaires producteurs d'électricité...).

Dès le début du vingtième siècle, les effets néfastes des rayonnements sur l'homme ont été observés. La nécessité de la mise en place d'une radioprotection a donc vite été comprise. Les premières commissions internationales non gouvernementales, chargées d'étudier les effets des rayonnements et de définir les règles de radioprotection, ont été créées vers la fin des années 1920. La première organisation professionnelle de radioprotection a pour sa part vu le jour à Chicago au mois de décembre 1942. Dès 1942, "la radioprotection était définie comme l'étude des radiations ionisantes sur l'homme et son milieu et la recherche des moyens de l'en protéger, de façon qu'il puisse bénéficier pleinement des applications des radiations ionisantes" [BM69]. Depuis, la *Protection*, ou *ingénierie de la radioprotection* (en anglais *Radiation Shielding*) s'est considérablement développée, via des travaux de référence [Roc56, Jae68] et l'organisation de plusieurs grandes conférences internationales (l'ICRS, *International Conference on Radiation Shielding*, a ainsi lieu tous les 4 à 5 ans).

Les études de protection jouent un rôle essentiel dans toutes les étapes de la vie d'un réacteur nucléaire et sont à considérer aussi bien pendant les phases de conception, d'exploitation, que de démantèlement du réacteur. Elles ont pour but (citation de [Mét06]) :

"- de quantifier, à l'aide de méthodes de physique-mathématiques et de codes de calcul - *codes protection* - des grandeurs macroscopiques pertinentes permettant d'évaluer l'effet des rayonnements sur la matière organique et la matière inerte (les matériaux) et de prendre les mesures appropriées éventuelles pour s'en protéger ;

- la conception et le dimensionnement de dispositifs permettant de ramener l'intensité des rayonnements à un niveau n'affectant ni la santé des personnes, ni l'intégrité des matériaux."

De manière générale, ces études de protection se divisent en deux grandes phases : tout d'abord, il faut identifier les sources de rayonnements et évaluer leur distribution (en espace, en énergie et en angle). Une fois ces sources de rayonnement connues, la deuxième phase de l'étude consiste à étudier la propagation de ces rayonnements, afin d'en évaluer l'impact en un point ou une zone de calcul.

1.1.2 Méthodes de calcul de la propagation des rayonnements

Pour notre part, nous nous intéressons à la seconde phase de ces calculs, à savoir l'étude de la propagation des rayonnements. Avant toute chose, il convient de rappeler que deux grandes catégories de rayonnement sont à distinguer : les particules neutres (neutrons, γ) et les particules chargées (électrons β^- , positons β^+ , particules α ...). À énergie égale, les particules neutres pénètrent la matière sur des distances bien plus importantes que les particules chargées (plusieurs centimètres ou dizaines de centimètres contre quelques millimètres au plus). Une approximation courante consiste alors à considérer que les particules chargées s'arrêtent à leur arrivée dans la matière. Naturellement, cette approximation n'est pas valable pour des milieux peu denses comme l'air. Cependant, dans un contexte d'intervention sur des sites nucléaires, cela signifie que ces particules chargées peuvent facilement être "arrêtées", par exemple via l'utilisation de combinaisons adéquates. Pour ce genre de simulations, il est donc a priori plus pertinent d'étudier la propagation des particules neutres, à savoir les neutrons et particules γ .

Pour ces particules neutres, le flux de particules, qui caractérise la population de particules en propagation, est régi par l'*équation du transport*, forme linéaire de l'équation de Boltzmann (voir en annexe A.1 la formulation explicite de cette équation). Cette équation peut être résolue de manière précise, par exemple par la méthode aux ordonnées discrètes ou par la méthode de Monte Carlo (codes de calcul de référence MCNP [FG85] et TRIPOLI-4 [DPD+06]), cette dernière permettant également d'étudier la propagation de particules chargées. Ces méthodes permettent d'évaluer très finement les quantités de radiations reçues par un opérateur ou par du matériel. Cependant, les temps de calcul demandés pour obtenir une estimation du débit de dose sont extrêmement importants et ne permettent donc pas la mise en place et la comparaison rapide de différents scénarios d'intervention.

En phase d'avant-projet, on utilise donc fréquemment d'autres méthodes, simplifiées, fournissant des résultats moins précis, mais nécessitant beaucoup moins de calculs. La méthode d'atténuation en ligne droite est la méthode que nous utilisons pour nos calculs (nos calculs géométriques sont couplés au code de calcul NARMER, successeur du code MERCURE-6, développé et maintenu par le SERMA, Service d'Études de Réacteurs et de Mathématiques Appliquées, basé au centre CEA de Saclay). Cette méthode permet d'étudier des configurations tridimensionnelles tout en présentant des temps de calcul bien plus faibles que les méthodes plus exactes. Naturellement, les approximations effectuées font qu'elle n'est pas adaptée à tous les problèmes, en particulier à l'étude de configurations lacunaires présentant de nombreux vides. Néanmoins, elle peut fournir de très bonnes approximations pour des cas où l'on souhaite évaluer l'efficacité d'un ou plusieurs écrans afin de freiner la propagation du rayonnement. Il est important de noter que cette méthode s'applique pour l'étude de la propagation des particules γ , et non des neutrons (il est cependant possible de l'utiliser pour les neutrons rapides).

1.1.3 Méthode d'atténuation en ligne droite avec facteurs d'accumulation

En principe, l'évaluation du débit de dose par une méthode de calcul “exacte” suppose le calcul explicite du flux de particules en tout point de l'espace des phases (là aussi, voir l'annexe A.1 pour plus de détails). Ce calcul du flux en tout point permet de prendre en compte l'intégralité des interactions entre les rayonnements γ et la matière, en particulier les “chocs”, provoquant une déviation des γ . Dans le cadre de la méthode d'atténuation en ligne droite, nous négligeons ces chocs et considérons uniquement le cas des particules suivant un chemin en ligne droite, traversant donc les différents écrans de la scène sans être déviés de leur trajectoire avant d'arriver au point de mesure du débit de dose. Sur la figure 1.1, nous pouvons observer le cas très simple d'une source ponctuelle et d'un point de mesure du débit de dose, seulement séparés par un objet “écran”. Normalement, le débit de dose reçu en un point M devrait dépendre des particules venant de la source S , parties “dans la direction de M ”, et n'ayant pas subi de choc au cours de leur propagation (rayon bleu sur la figure). Le calcul du débit de dose au point M devrait aussi prendre en compte le cas des particules parties dans une autre direction (on considère le cas d'une source isotrope), “revenues” vers M après une ou plusieurs interactions avec la matière (voir sur la figure 1.1 les rayons de couleur rouge).

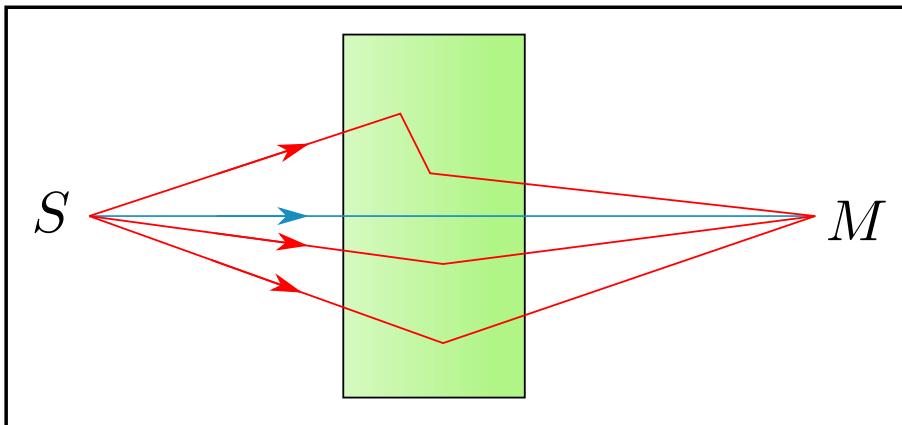


Figure 1.1 – Exemple de parcours de “particules” γ participant au débit de dose reçu au point M venant de la source ponctuelle isotrope S . Les lignes rouges représentent le parcours de “particules” subissant des “chocs”, et donc des déviations au cours de leur trajet. La méthode d'atténuation en ligne droite ne tient compte que du flux sans choc (cas des particules non déviées, en bleu sur le schéma).

Dans le cadre de la méthode d'atténuation en ligne droite, nous négligeons donc ces particules, et ne considérons que le *flux sans choc* des γ provenant de la source. Afin d'obtenir une évaluation réaliste du débit de dose, ces autres particules “déviées” doivent cependant être prises en compte, via l'expression d'un facteur d'accumulation, dépendant de nombreux paramètres, tels le numéro atomique du milieu, l'énergie des sources considérées, la géométrie du milieu traversé (fini ou infini, plan ou sphérique), ou encore la configuration du milieu (milieu homogène unique ou succession de milieux homogènes). Dans la pratique, ces facteurs d'accumulation sont évalués en amont et donc connus pour de nombreuses valeurs des paramètres de référence. Pour chaque cas de calcul de débit de dose, des lois d'interpolation appropriées permettent d'évaluer le facteur d'accumulation pour la situation considérée, à partir de ces valeurs connues, regroupées dans des tables de référence. La précision du calcul

de débit de dose obtenu via la méthode d'atténuation en ligne droite dépendra donc majoritairement de la qualité de ces facteurs d'accumulation, et des méthodes d'interpolation utilisées (voir [SC05, TSP07] pour des développements assez récents). Bien que cette méthode ne soit pas adaptée à tous les cas de figure (le calcul des facteurs d'accumulation dépend d'hypothèses simplificatrices sur la géométrie de la scène autour des zones d'interaction avec la matière), elle convient néanmoins assez bien à la propagation des rayonnements γ , subissant, en moyenne, très peu de “chocs aux grands angles” (déviations fortes de leur direction après interaction avec la matière).

L'évaluation de ces facteurs d'accumulation étant effectuée par le code de calcul NARMER, nous devons seulement, de notre côté, fournir les calculs géométriques suffisant à l'évaluation de ce “flux sans choc”. Considérons donc maintenant le cas d'une source ponctuelle isotrope émettant S_0 γ d'énergie E_0 par unité de temps, située en un point S . Le flux sans choc ϕ_0 , en un point M , situé à une distance ρ de cette source, s'exprime ainsi :

$$\phi_0(M) = \frac{S_0}{4\pi\rho^2} e^{-\delta(E)} \quad (1.1)$$

où le terme $\frac{S_0}{4\pi\rho^2}$ représente le flux de gamma en l'absence de matière (propagation dans le vide), et le terme $e^{-\delta(E)}$ la probabilité de non-interaction avec la matière sur la distance ρ . Le terme $\delta(E)$ est pour sa part défini par la relation :

$$\delta(E) = \sum_j (\sum_k \sigma_k(E) N_{j,k}) d_j$$

avec :

- j : indice sur les objets intersectés
- k : indice sur les différents corps simples (composition atomique) de chaque objet j
- $N_{j,k}$: nombre d'atomes (par unité de volume) du corps simple k dans l'objet j
- $\sigma_k(E)$: section efficace microscopique (voir annexe A.1)
- d_j : distance d'interaction du rayon avec l'objet j

Les calculs géométriques que nous devons effectuer afin d'évaluer ce débit de dose se limitent donc au calcul des distances d_j d'intersection avec la matière. En d'autres termes, cette distance d_j est la distance parcourue par le rayon liant le point S au point M dans l'objet d'indice j . Afin d'évaluer le débit de dose au point M , il nous faut donc trouver l'ensemble des intersections entre ce rayon et les objets de la scène, mais aussi les trier. En effet, un “objet” (cet objet pouvant être un matériau, comme du béton, ou un fluide, comme de l'eau) peut être contenu dans d'autres objets, par exemple dans le cas d'un tube calorifugé présentant plusieurs couches d'épaisseurs, ou d'une cuve contenant un fluide, lui aussi modélisé. Il nous faut alors pouvoir reconstruire le parcours du rayon dans chaque différente portion d'objet, et nous devons pour cela disposer de cette liste d'intersections triées (voir figure 1.2).

Le traitement de sources de radiations surfaciques ou volumiques est un peu plus complexe : tout d'abord, l'objet irradiant est maillé (mailles cubiques, par exemple, pour une source volumique), suivant un nombre de mailles N défini par l'utilisateur du logiciel de calcul de débit de dose. L'utilisateur définit également un nombre de “tirages” aléatoires T par maille, qui seront autant de sources ponctuelles de rayonnement (au nombre, donc, de $N * T$ au total). Des rayons, partant de ces sources et arrivant vers le point de mesure du débit de dose M , sont alors lancés. La contribution de chacune de ces sources au débit de dose global reçu en M est alors calculée suivant une expression similaire à la formule 1.1 et demande donc un “lancer de rayon” partant de la source, arrivant en M , pour lequel toutes les intersections

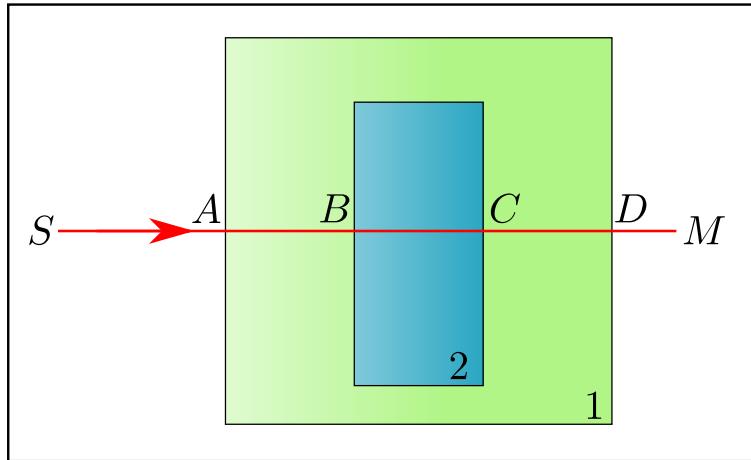


Figure 1.2 – Exemple de parcours d'un rayon γ à l'intérieur de plusieurs objets successifs, l'objet 2 étant encastré dans l'objet 1. Le rayon intersecte d'abord l'objet 1 sur une distance AB , puis l'objet 2 sur une distance BC , et l'objet 1 sur une distance CD . Afin de correctement détecter ces parcours dans la matière, il nous faut donc trier les intersections trouvées le long du rayon SM .

avec la matière doivent être identifiées et triées. L’ “addition” de la contribution de chacune de ces sources permet ensuite d’évaluer le débit de dose au point M .

À partir de ce débit de dose, une nouvelle répartition des sources ponctuelles de radiations, dépendant des résultats précédents, est réalisée (sans entrer dans les détails : on maille plus finement les zones contribuant de manière plus importante au débit de dose au point M . Les zones “moins importantes” sont quant à elles maillées plus grossièrement, de manière à conserver un nombre total de sources ponctuelles constant). Cette nouvelle répartition des sources de radiations donne lieu à une nouvelle évaluation du débit de dose au point M (cette évaluation tient aussi compte de la valeur précédemment trouvée). Ce processus est ainsi répété un certain nombre de fois (un nombre maximum d’interactions est défini par l’utilisateur), jusqu’à ce que le débit de dose semble converger (un écart-type minimal est également défini par l’utilisateur). Pour un nombre i d’itérations effectuées, nous effectuons donc en tout i lancers successifs de $N * T$ rayons. De par leur définition, ces $N * T$ rayons présentent a priori une certaine “cohérence” entre eux, puisque leurs points de départ appartiennent à une même zone de l’espace. Lors du calcul du débit de dose pour plusieurs de ces sources volumiques ou surfaciques, nous devons donc manipuler plusieurs paquets de rayons “cohérents”, correspondant à chaque source de radiations.

Au final, cette étape de calculs géométriques est, dans de nombreux cas, l’étape limitante du calcul de débit de dose. C’est pourquoi nous souhaitons optimiser fortement ces requêtes d’intersections, en gardant à l’esprit deux caractéristiques propres à notre problème. En premier lieu, comme nous l’avons déjà dit, notre calcul d’intersections se doit d’être exhaustif : pour chaque rayon joignant le point de mesure du débit de dose à une source ponctuelle de radiations, nous cherchons l’intégralité des interactions avec la matière, donc l’ensemble des intersections entre le rayon considéré et les triangles de la scène (nous nous limitons en effet dans ce manuscrit au traitement de scènes 3D représentées sous formes de maillages de triangles). La deuxième caractéristique propre à notre problème constitue plutôt un avantage en notre faveur : comme nous nous limitons ici au calcul du débit de dose en un seul point de l’espace à la fois, l’intégralité des rayons lancés pour calculer le débit de dose converge

en ce point de mesure. Nous verrons plus loin que cette propriété peut être avantageusement utilisée afin de réduire considérablement les temps de calcul.

Le problème auquel nous devrons répondre dans ce manuscrit est donc le suivant : étant donné un groupe de triangles et plusieurs groupes de rayons cohérents, nous devons calculer l'intégralité des intersections de chaque rayon avec les triangles de la scène. De plus, pour chacun de ces rayons, il nous faut trier ces intersections le long du rayon. Nous devons ensuite calculer les distances parcourues dans chaque objet à partir de cette liste d'intersections associées au rayon. Rappelons, enfin, que nous souhaitons pouvoir travailler sur des scènes contenant des objets mobiles (les sources, comme les écrans de protection, peuvent se déplacer), pouvant même présenter des changements de topologie au cours du temps (objet introduit dans la scène en cours d'intervention, prélèvement de matière entraînant une modification des objets de la scène, découpe d'objets durant une opération de démantèlement...). Nous souhaitons donc pouvoir mettre en place des algorithmes sur des scènes "dynamiques", pouvant évoluer très fortement entre deux pas de temps de la simulation.

1.2 Exploitation des cartes graphiques

Après avoir présenté notre contexte d'étude, nous allons maintenant pouvoir évaluer les différentes pistes de travail pouvant nous permettre d'optimiser ces nombreuses requêtes d'intersections. Pour la suite de ce chapitre, et jusqu'au chapitre 4, nous nous limiterons à un cas particulier, qui concentre cependant la grande majorité des difficultés auxquelles nous avons à faire face : plutôt que de considérer plusieurs paquets de rayons cohérents, nous ne traiterons le cas que d'un seul paquet de rayons cohérents. Par "cohérents", nous entendons ici que tous les rayons de ce paquet devront être situés du même côté d'un plan donné (voir Figure 1.3) : de manière plus concrète, si l'on se place au point de mesure du débit de dose, donc au point de départ (ou d'arrivée, selon le point de vue) de tous les rayons, il doit être possible de voir tous les rayons d'un seul regard, en dirigeant ce regard vers une direction appropriée (en considérant ici que notre champ de vision est de 180 degrés). Par rapport au cas général, ces hypothèses ne sont pas très contraignantes, et présentent l'intérêt de faciliter les analogies avec les travaux de la communauté du rendu graphique.

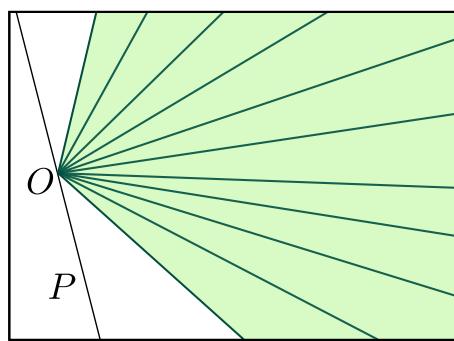


Figure 1.3 – Paquet de rayons cohérents répondant à nos critères vu en deux dimensions.
Tous les rayons du paquet partent du point 0 et sont situés du même côté du plan P .

Plaçons-nous donc dans le cadre du traitement d'un paquet de rayons cohérents : au sein d'un même paquet, le nombre de rayons est potentiellement très grand, et les requêtes d'intersection propres à des rayons différents peuvent tout à fait être réalisées de manière indépendante. Ces caractéristiques font de notre problème un problème intrinsèquement parallèle, et donc a priori adapté aux architectures hautement parallèles. Depuis quelques an-

nées, les processeurs de cartes graphiques (ou GPUs) représentent une solution d'architecture hautement parallèle à faible coût, et leur utilisation a été grandement facilitée. Il est ainsi désormais possible de développer assez aisément ses propres codes sur GPU, et d'obtenir des gains importants en performance. Néanmoins, lorsque cela est possible, il reste plus intéressant d'exploiter les algorithmes implantés matériellement sur les cartes graphiques, comme l'algorithme de rastérisation, procédé majoritairement utilisé pour le rendu 3D d'objets triangulés. Or, comme nous allons le voir, une analogie évidente peut être effectuée entre notre problème et l'algorithme de rastérisation. Dans les pages qui viennent, nous allons donc développer cette analogie, et étudier les possibilités de détourner l'utilisation classique des unités de rastérisation pour nos besoins.

1.2.1 La rastérisation

1.2.1.1 Présentation

La rastérisation [Bre65] consiste en la conversion d'une image vectorielle (ensemble de points, ligne, courbes, polygones...) en une image matricielle, soit une matrice de pixels destinée à être affichée sur un écran. Dans le langage courant, le terme rastérisation s'étend à l'algorithme de rendu permettant d'afficher une scène 3D sur un écran (les scènes 3D étant la plupart du temps stockées de manière vectorielle, sous forme de groupes de polygones). La rastérisation présente l'avantage d'être une méthode beaucoup plus rapide que le lancer de rayons (dont nous parlerons dans la partie suivante) en terme de calcul. Dans sa forme basique (pas de calcul d'ombres ni de traitement des textures), elle se compose des étapes suivantes :

1. Projection des sommets des triangles dans le repère de la caméra, en 2D
2. Filtrage des sommets. Ce filtrage consiste à considérer la position des sommets par rapport au cône de vision de la caméra. Si, pour un triangle donné, ses trois sommets sont hors du champ de vision de la caméra, le triangle n'a plus à être considéré. Dans d'autres cas, un triangle sera conservé tel quel, ou tronqué. Dans ce dernier cas, on ne garde que la portion de triangle intersectée par le cône de vision (opération de clipping), ce qui peut provoquer la création de nouveaux triangles.
3. Rastérisation à proprement parler des triangles restants : on calcule pour chaque triangle les pixels qu'il recouvre. Pour chacun de ces pixels, on calcule alors la couleur à afficher, et la profondeur du triangle associé. Si ce pixel a déjà été identifié comme recouvert par un triangle, il est déjà associé à la couleur d'une autre primitive. On compare alors les profondeurs de ces deux triangles à la position donnée par le pixel (utilisation du Z-buffer [Cat74]), et on ne garde que la couleur correspondant au triangle le plus proche de la caméra.

1.2.1.2 Analogie avec la requête exhaustive d'intersections

La rastérisation est donc très proche de notre algorithme : en effet, si l'on assimile chaque pixel de l'écran 2D à un rayon lumineux arrivant dans l'œil, cet algorithme devient équivalent à la recherche de la première intersection de chacun de ces rayons avec les triangles de la scène. De plus, comme on peut le voir dans la troisième étape de l'algorithme, on considère successivement, pour chaque pixel (donc chaque rayon), tous les triangles intersectés. On ne conserve via le Z-buffer que le premier intersecté, mais tous les triangles intersectants sont calculés (on calcule aussi au passage la distance entre le point de départ du rayon et le point d'intersection du rayon avec le triangle, donnée dont nous avons aussi besoin).

Cette analogie entre l'algorithme de rastérisation et notre problème nous amène à envisager l'utilisation de la rastérisation pour nos algorithmes. En effet, la rastérisation étant à l'heure actuelle la méthode de choix pour le rendu 3D, elle bénéficie d'une implantation matérielle sur tous les modèles courants de cartes graphiques, et peut donc atteindre des performances impressionnantes en termes de temps de calcul. Néanmoins, nous avons deux problèmes majeurs à résoudre pour pouvoir utiliser ce matériel :

1. La première différence entre nos besoins et les possibilités offertes par l'utilisation habituelle du “pipeline graphique” (succession des traitements réalisés par la carte graphique afin d'afficher les données) provient du stockage de toutes les intersections. En effet, la quantité de mémoire disponible pour chaque pixel est limitée et constante, si bien qu'en principe, il n'est pas possible de stocker plusieurs intersections. Certes, l'intégralité des intersections entre les rayons et les triangles est calculée au cours du processus de rendu, mais au final, une seule intersection est conservée par pixel.
2. Ensuite, la rastérisation, telle qu'implantée sur les cartes graphiques, suppose que les rayons (ou pixels) suivent une répartition régulière sur la surface de l'écran. Cette condition permet de simplifier les calculs lors de plusieurs étapes de l'algorithme de rendu, et donc de bénéficier d'une accélération matérielle plus efficace. Cette contrainte n'est guère étonnante, car elle correspond à la très grande majorité des besoins dans le domaine du rendu de scènes 3D. Néanmoins, dans notre cas, les rayons d'un même groupe partent dans des directions assez proches, mais ils ne suivent aucun schéma régulier (rappelons qu'ils sont imposés par le calcul du débit de dose réalisé par le code de calcul NARMER). Il nous faut donc trouver un moyen de contourner cette difficulté. Même si cette limitation n'est pas propre à l'algorithme de rastérisation, et est liée à l'optimisation des architectures de cartes graphiques pour des besoins précis, elle reste une limitation importante, car elle nous empêche, a priori, d'utiliser ce matériel pour nos algorithmes.

Si nous voulons exploiter le pipeline graphique pour nos algorithmes, il nous faut donc contourner ces deux difficultés majeures.

1.2.1.3 Détournement des unités de rastérisation

Notre premier besoin, à savoir stocker plusieurs intersections par pixel, est en réalité un besoin récurrent dans la communauté graphique, car il permet de traiter correctement l'affichage d'objets transparents, et peut également servir à régler des problèmes d'aliasing.

La première solution pour stocker un nombre illimité d'intersections pour chaque pixel consiste naturellement à modifier le matériel existant. Plusieurs propositions ont été faites à ce sujet, la première étant le A-buffer [Car84], dans lequel tous les fragments (un fragment peut être vu comme un couple pixel-triangle intersectant) étaient conservés en mémoire sous forme de listes chaînées. D'autres propositions ont suivi [Wit01, JC99], comme le F-buffer [MP01], dont une implémentation logicielle, restreinte aux cartes de la marque ATI, a été proposée en 2005 [HPS].

Comme nous souhaitons faire fonctionner nos algorithmes sur un matériel générique, nous ne pouvons pas exploiter ces solutions consistant à modifier la structure des cartes graphiques utilisées. Il nous faut donc nous tourner vers des solutions logicielles et explorer les divers contournements du pipeline réalisables pour stocker tous les niveaux de profondeur rencontrés par chaque pixel. La technique du depth-peeling [Eve01] est certainement la première solution venant à l'esprit, car la plus intuitive : elle consiste tout simplement à multiplier les passes de rastérisation, de manière à récupérer successivement les épaisseurs de matière intersectée.

Il suffit pour cela d'exécuter une première passe, puis de faire la deuxième en faisant des tests de profondeur prenant en compte les résultats de la passe précédente. En répétant ce processus n fois, on peut ainsi récupérer les n premières intersections le long de chaque rayon lumineux. Cette solution n'est toutefois pas vraiment satisfaisante, car elle impose de réaliser un nombre de passes possiblement très élevé, et très difficile à prévoir en amont.

Depuis quelques années a été introduite la possibilité d'effectuer un rendu dans plusieurs textures, via les MRTs (Multiple Render Targets). Concrètement, cet ajout est utile pour associer de nombreuses informations à un pixel donné, et peut donc entre autres permettre de stocker plusieurs fragments par pixel. En exploitant ces améliorations, le k-buffer [BCL⁺⁰⁷] permet de conserver en mémoire k intersections triées par pixel. Néanmoins, cette méthode devient beaucoup moins efficace pour $k > 8$, ce qui implique là encore que nous exécutons de multiples passes pour récupérer toutes les intersections d'une scène. Le k-buffer peut être exploité de manière à prendre en compte toutes les intersections de la scène [BCL⁺⁰⁷], mais il faut alors fusionner certains fragments (les intersections très proches les unes des autres sont transformées en une seule intersection), ce que nous ne pouvons faire dans le cadre de notre application. Cette technique, bien que plus rapide, souffre donc des mêmes problèmes que le depth-peeling, mais présente aussi des défauts majeurs : comme elle repose sur le fait que chaque nouveau fragment récupère la liste des fragments associés au pixel actuel pour la modifier (par exemple pour insérer le nouveau fragment dans la liste triée des fragments), et ensuite la recopier en mémoire, elle peut poser des problèmes de concurrence entre différentes unités de calculs souhaitant modifier cette liste au même instant. Les solutions proposées dans ce papier provoquent malheureusement une perte notable de performance. Ces problèmes sont résolus dans une proposition d'amélioration de la technique du depth-peeling effectuée par Liu [LHLW09] : en exploitant à nouveau les MRTs, jusqu'à 32 intersections par pixel peuvent être sauvées. Néanmoins, avec ce modèle, des intersections trop proches l'une de l'autre peuvent être ratées. Là encore, nous ne pouvons donc pas conserver une telle solution pour résoudre nos problèmes.

En conclusion, aucune de ces techniques ne permet de calculer et de conserver en mémoire toutes les intersections pour chaque pixel de l'écran. De plus, l'opération de rastérisation de chaque triangle n'est à l'heure actuelle pas configurable, ce qui pourrait provoquer la non-détection de certains triangles occupant une infime partie de la surface d'un pixel. La seconde différence entre la rastérisation et nos besoins (répartition régulière des pixels sur l'écran, alors que nos rayons ne suivent aucun schéma de répartition) n'est hélas pas non plus contournable. Des solutions ont été proposées, sous le nom de rastérisation irrégulière [JLBM05, AL04], mais elles consistent toutes deux en des propositions de modification de l'architecture des cartes graphiques, et non en une réelle implantation logicielle de ces algorithmes.

Néanmoins, la plupart de ces incompatibilités résultent plus de choix d'architecture que de réelles difficultés théoriques. Ces choix d'architecture correspondent à des optimisations utiles dans une majorité de cas d'applications, mais ils ne sont pas incompatibles avec les fondements algorithmiques de la rastérisation. La structure hautement parallèle des processeurs de cartes graphiques reste a priori particulièrement adaptée à notre problème. Il y a quelques années, il était encore extrêmement difficile de tirer partie de la puissance de calcul des cartes graphiques pour d'autres besoins que ceux du rendu. Depuis quelques temps, la tendance est à un fort accroissement de la programmabilité des cartes graphiques, entre autres via le développement d'interfaces de programmation dédiées, rendant la programmation beaucoup plus intuitive. Ainsi, de nombreuses applications scientifiques dans des domaines extrêmement variés ont pu bénéficier de ces architectures hautement parallèles à faible coût (citons la dynamique moléculaire [SPF⁺⁰⁷], l'astrophysique [BBZ08] ou la détection de collision en réalité virtuelle [TMLT11]). Ces développements sont regroupés sous le terme de calcul géné-

rique sur processeurs graphiques, en anglais GPGPU (pour General-Purpose Processing on Graphics Processing Units).

1.2.2 Calcul générique sur Processeurs graphiques (GPGPU)

1.2.2.1 Structure des processeurs de cartes graphiques

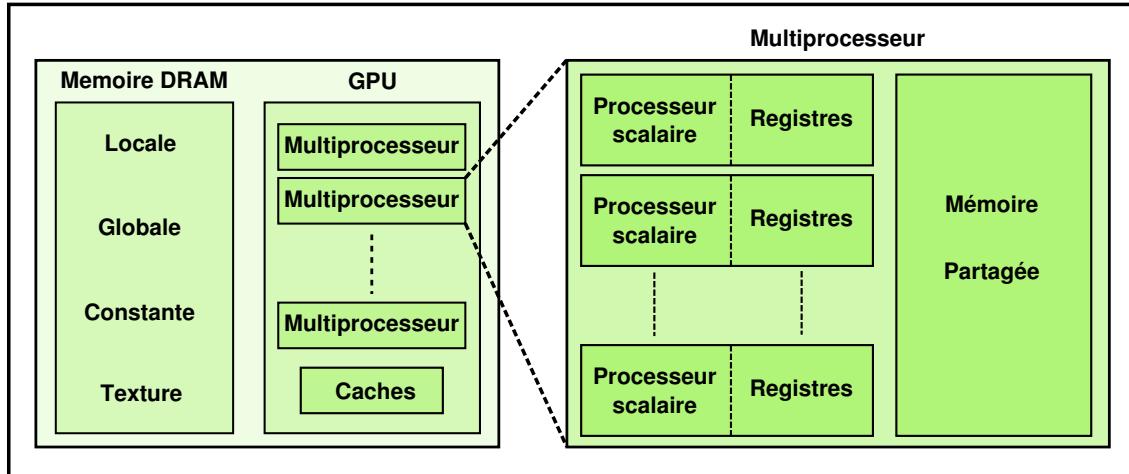


Figure 1.4 – Schéma simplifié d'une carte graphique. Sur la gauche du schéma, vision globale de la carte (mémoire DRAM + GPU). Sur la droite, description plus détaillée d'un multiprocesseur.

En effet, les processeurs de cartes graphiques, ou GPUs, présentent des architectures hautement parallèles, qui, lorsqu'elles sont correctement exploitées, permettent de très importants gains de performance. Concrètement, un GPU est constitué de l'union de plusieurs multiprocesseurs (voir le schéma d'une carte graphique Figure 1.4). Chacun de ces multiprocesseurs regroupe, en plus de certaines unités logiques, plusieurs processeurs scalaires (en anglais ALU, pour Arithmetic Log Unit). Ces processeurs scalaires permettent l'exécution simultanée d'un groupe SIMD (Single Instruction Multiple Data) de threads. Chacun de ces groupes SIMD de threads peut être vu comme une machine PRAM de type Lecture Concurrente, Ecriture Concurrente (CRCW), avec un nombre de processeurs constant et une résolution arbitraire des conflits en écriture.

Du point de vue de la mémoire, les multiprocesseurs se partagent l'accès à une mémoire globale (latence importante) et à des textures (latence moyenne). Au sein d'un même multiprocesseur, chaque thread a accès à un certain nombre de registres qui lui sont propres, et à une mémoire dite "partagée", propre au multiprocesseur, et commune au groupe SIMD de threads auquel il appartient. Cette mémoire partagée présente elle aussi une latence très réduite, similaire à celle d'un cache de niveau L1. Afin de fournir quelques ordres de grandeur, notons que sur la carte NVIDIA GTX 470, de dernière génération, on compte 14 multiprocesseurs et 32 processeurs par multiprocesseur, soit un total de 448 processeurs scalaires sur la carte (la tendance par rapport à la génération précédente de cartes est à la diminution du nombre de multiprocesseurs, mais à l'augmentation de leur force de calcul). Deux groupes de 32 threads peuvent être exécutés "simultanément" (via un système de pipeline) sur un multiprocesseur.

La grande force de l'architecture de ces cartes graphiques vient du fait que chaque multiprocesseur comporte une unité de gestion des threads qui permet de maintenir en parallèle un

grand nombre de groupes SIMD de threads. Par exemple, lorsqu'un groupe SIMD fait appel à une donnée en mémoire, un autre groupe SIMD est immédiatement exécuté, et l'exécution du premier groupe SIMD reprendra lorsque le transfert de données aura eu lieu. Ce système permet donc de cacher en grande partie les latences dues à des accès mémoire coûteux en performance, et donne l'impression de manipuler un nombre de processeurs scalaires beaucoup plus important que le nombre de processeurs réellement disponibles sur la carte. En reprenant l'exemple de la NVIDIA GTX 470, un multiprocesseur est capable de gérer les contextes d'exécution de 48 groupes de threads, soit $48 * 32 = 1536$ threads, pour un total de 21504 threads gérés en même temps sur la carte (il ne faut pas confondre ici ce nombre de threads dont le contexte est géré simultanément avec le nombre de threads réellement exécutés en même temps, beaucoup plus faible). Le temps de création d'un thread étant quasi-nul, et la gestion de ces groupes SIMD extrêmement rapide (souvent exécutée directement par le matériel), ces architectures sont particulièrement adaptées à des problèmes présentant un grain très fin de parallélisme. Néanmoins, en adaptant intelligemment les algorithmes, et en exploitant correctement les interfaces de programmation dédiées, il est possible d'accélérer les temps de calcul de nombreux algorithmes.

1.2.2.2 Présentation de l'API CUDA

Afin d'utiliser efficacement les processeurs de cartes graphiques, les fabricants de cartes ont depuis quelques années mis à la disposition des développeurs des interfaces de programmation facilitant grandement la communication avec les cartes graphiques. L'interface ayant rencontrée le plus de succès au cours des années précédentes est très certainement l'API CUDA [NVI09, NBGS08] proposée par NVIDIA. Le fabricant concurrent ATI a de son côté choisi la solution OpenCL [Ope09], fonctionnant suivant les mêmes principes, mais ayant le double avantage d'être multi-plateforme (possibilité d'utiliser des architectures multi-CPUs ou des cartes NVIDIA) et hétérogène (possibilité de contrôler simultanément des CPUs et des GPUs). Nous avons néanmoins choisi d'utiliser CUDA, que nous avons jugé plus mature, et qui bénéficie aujourd'hui d'un meilleur support. Nous avons ainsi beaucoup utilisé Thrust [HB09], librairie libre d'algorithmes parallèles associés à une interface similaire à la Standard Template Library (STL) du langage C++ (cette interface de type STL est possible grâce au support des templates en CUDA, absent de la norme OpenCL).

CUDA permet de communiquer avec une carte graphique via une API en langage de type C. Le code exécuté sur GPU est appelé via des fonctions appelées *kernels*. De manière générale, chaque appel de kernel sur le CPU correspond à l'exécution d'un grand nombre de threads, chacun de ces threads étant exécuté par un des processeurs scalaires présentés dans le paragraphe précédent. Les groupes SIMD de threads exécutés parallèlement sur un multiprocesseur sont ici appelés *warps*. Sur les cartes actuelles, les warps sont constitués de 32 threads. L'unité dite SIMT (Single Instruction Multiple Threads) de chaque multiprocesseur s'occupe de créer, d'ordonner et d'exécuter ces warps à tour de rôle. Lorsqu'un warp fait appel à un accès mémoire, un autre warp est immédiatement exécuté. Si, au sein d'un même warp, certains threads sont amenés à diverger, on exécutera séquentiellement les deux branches correspondant à cette divergence. Il faut donc éviter au maximum ce genre de comportements afin d'optimiser les performances.

Le warp n'a pas d'existence au sein du modèle CUDA (mais il est néanmoins primordial de garder en tête son existence lors de la programmation). En effet, au sein de l'API CUDA, les threads sont regroupés par *blocs*. Un bloc correspond à l'exécution de plusieurs warps de threads sur un même multiprocesseur. Au sein d'un bloc, les warps peuvent se synchroniser entre eux via un appel à la fonction `_syncthreads()`. Des threads de blocs différents n'ont a

priori aucun moyen de synchronisation, hormis l'appel à des opérations atomiques en mémoire globale, ou tout simplement, l'exécution d'un nouveau kernel. Il est important de noter que la sérialisation des calculs en présence de branches divergentes n'est vraie qu'au sein d'un warp. Des warps d'un même bloc peuvent tout à fait diverger entre eux sans perte de temps, ce qui constitue une amélioration conséquente par rapport aux anciens modèles de GPUs qui ne présentaient qu'un large modèle SIMD. Une caractéristique très importante de ces blocs de threads est qu'ils partagent l'accès à une mémoire partagée (*shared memory*) de très faible latence, qui permet la communication entre les threads d'un même bloc.

Ces blocs de threads sont ensuite regroupés dans une grille de blocs, à une ou plusieurs dimensions (tout comme les blocs de threads peuvent être à une ou plusieurs dimensions, de manière à faciliter l'analogie avec certains algorithmes, par exemple dans le domaine de la synthèse d'image). Lorsqu'un kernel est lancé, une unité de répartition du travail compte les blocs de threads nécessaires et les répartit sur les multiprocesseurs (un même multiprocesseur peut exécuter plusieurs blocs simultanément). Quand les blocs de threads terminent leur travail, l'unité de répartition lance de nouveaux blocs sur les multiprocesseurs disponibles. Une des grandes richesses de ce modèle de programmation vient du fait que les dimensions des grilles et des blocs de threads peuvent différer entre les kernels successifs. On peut donc faire varier facilement le grain de parallélisation entre les différents kernels, ce qui apporte un avantage très appréciable par rapport aux architectures SPMD (Single Program Multiple Data) classiques.

En terme de mémoire, chaque thread dispose d'un nombre limité de registres. Lorsque la quantité de mémoire nécessaire à un thread est trop importante, on fait appel à la mémoire dite *locale*, qui est en fait située sur la mémoire DRAM de la carte, et qui présente donc une latence aussi élevée que la mémoire globale. Il peut parfois être utile d'utiliser alors la mémoire partagée des blocs de threads pour stocker certaines variables, permettant ainsi un accès à un nombre "virtuel" plus élevé de registres. Le principal intérêt de la mémoire partagée (outre de servir d'espace d'échange de données entre les threads d'un même bloc) est néanmoins de stocker localement des données venant de la mémoire globale dont on sait qu'on y accèdera souvent. En effet, les échanges avec la mémoire globale sont à limiter au maximum, en particulier lorsque les données chargées au sein d'un même warp sont situées à des emplacements différents en mémoire : si les 32 threads d'un même warp font appel à des données très dispersées, ils peuvent générer jusqu'à 32 appels mémoire successifs, là où on peut n'en faire qu'un lorsqu'on charge des données très cohérentes et correctement alignées.

La bonne exploitation d'une carte graphique sous CUDA dépendra donc principalement de la parallélisation des tâches, mais aussi de la bonne gestion de la mémoire : il faut faire en sorte que les accès mémoire soient les plus cohérents possibles au sein d'un même warp, et également paralléliser au maximum les calculs au sein des blocs afin de recourir le moins possible à la mémoire globale.

1.2.2.3 Algorithmes parallèles récurrents

Afin de terminer cette présentation de la programmation sur processeurs graphiques, il nous faut signaler que de nombreux travaux ont été effectués ces dernières années afin d'optimiser le portage sur GPU de certains algorithmes extrêmement récurrents que l'on exploitera dans ce manuscrit. La brique de base de ces algorithmes est certainement l'algorithme parallèle de scan : on fournit en entrée d'un scan un vecteur de données et un opérateur binaire associatif \oplus doté d'un élément identité Id . Si le vecteur d'entrée est de la forme $[a_0, a_1, a_2 \dots a_n]$, un scan exclusif sur ce vecteur produira en sortie le vecteur $[Id, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \dots, a_0 \oplus a_1 \oplus a_2 \oplus \dots \oplus a_{n-1}]$. Un scan inclusif, quant à lui, produira

le vecteur $[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \dots, a_0 \oplus a_1 \oplus a_2 \oplus \dots \oplus a_n]$. Dans le cas, très commun, où l'opérateur binaire \oplus est l'opérateur de somme, un scan exclusif consiste simplement à produire à la place i un vecteur donnant la somme de tous les éléments précédant l'élément a_i dans le vecteur d'origine. Si le scan est inclusif, il faut inclure l'élément a_i dans cette somme. Dans le cadre d'un algorithme parallèle, un tel algorithme de scan peut être très utile, par exemple dans le cas où chaque thread doit produire en mémoire un vecteur de taille différente. Il faut alors commencer par déterminer où chaque thread doit écrire son résultat en mémoire. Comme cet emplacement dépend du nombre d'éléments générés par les threads précédents, une stratégie classique dans ce cas consiste à exécuter un premier kernel dans lequel chaque thread compte le nombre d'éléments qu'il va générer, puis à faire un scan sur le vecteur résultant afin de déterminer où chaque thread devra écrire son résultat. Enfin, un nouveau kernel sera lancé, afin que chaque thread procède réellement à l'écriture des données, à l'emplacement déterminé par l'étape de scan. Cet algorithme de scan est donc très utile dans un contexte d'algorithmes hautement parallèles, et l'on comprend aisément qu'il nous sera très utile dans un contexte où on ne se limite pas à la première intersection le long de chaque rayon, mais où l'on souhaite calculer l'intégralité des intersections le long de chaque rayon.

Un algorithme parallèle de scan est complexe à mettre en place, puisque la valeur d'un élément en sortie dépend de la valeur de tous les éléments placés avant lui en entrée. Néanmoins, des algorithmes parallèles extrêmement efficaces ont été récemment conçus [SHZ07, MG09] pour adapter le traitement de ce problème au GPU. Ils reposent principalement sur une optimisation fine du scan au sein d'un bloc de threads, via la réutilisation d'algorithmes adaptés pour les machines PRAM [HS86, Ble90], et sur une minimisation des écritures en mémoire globale (néanmoins nécessaires afin de communiquer les informations entre les différents blocs de threads). De la même manière, des solutions efficaces ont été trouvées pour paralléliser l'opération de scan segmenté, où les scans “repartent à zéro” en certains emplacements du vecteur.

D'autres algorithmes essentiels ont également bénéficié de recherches poussées afin d'optimiser leur exécution sur des machines parallèles. Le tri, autrefois plus lent sur GPU que sur CPU, peut désormais être effectué plus rapidement sur GPU que sur les processeurs classiques [SHG09, MG10], ce qui permet d'éviter des transferts très coûteux entre le CPU et le GPU lorsque des données doivent être triées : on peut désormais également effectuer cette étape sur GPU sans perte de performance. Enfin, il est intéressant de citer le cas du “compactage de flux” (en anglais *stream compaction*), permettant d'enlever certains éléments d'un vecteur ne vérifiant pas une certaine propriété, qui, là aussi, a reçu un traitement particulier sur les processeurs de cartes graphiques [BOA09]. Comme pour le cas de l'algorithme de scan, ces implantations extrêmement efficaces reposent sur une optimisation poussée des calculs au sein d'un bloc de threads et sur une exploitation maximale de la mémoire partagée.

En résumé, la programmation sur carte graphique est aujourd’hui grandement simplifiée et permet de profiter assez facilement d’architectures hautement parallèles. De plus, de nombreux algorithmes génériques ont bénéficié récemment d’implantations optimisées sur GPU. Nous allons donc dans la suite de ce manuscrit essayer d’exploiter ces processeurs de cartes graphiques afin d’améliorer grandement nos performances.

1.3 Lancer de rayons

Après avoir présenté les processeurs de cartes graphiques et notre volonté d’exploiter leur puissance, il nous faut revenir à notre problématique initiale : calculer l'intégralité des

intersections entre un grand nombre de rayons et un grand nombre de triangles, tous ces rayons partant d'un même point. Rappelons que, de plus, nous souhaitons pouvoir traiter des scènes dynamiques, dans lesquelles des objets peuvent bouger, apparaître, disparaître, tandis que le point de mesure peut lui aussi évoluer dans la scène au cours de la simulation. Ce problème de la requête d'intersections entre un groupe de rayons et un groupe de triangles est un problème abondamment abordé par la littérature, essentiellement par la communauté du lancer de rayons, ou ray-tracing. Dans cette partie, nous allons en premier lieu présenter cet algorithme de rendu (1.3.1), avant d'étudier les différentes solutions algorithmiques conçues pour obtenir des performances interactives dans ce domaine (1.3.2, 1.3.4 et 1.3.5). Enfin, nous présenterons la grille en perspective (1.3.6), que nous utiliserons dans la suite de ce manuscrit.

1.3.1 Présentation de l'algorithme

L'algorithme de lancer de rayons est une technique de rendu présentée pour la première fois à la fin des années 60 [App68]. Dans sa première version, il consiste à lancer des rayons partant de l'œil de l'utilisateur (donc d'une caméra virtuelle), chacun de ces rayons correspondant à un pixel de l'image 2D devant finalement être affichée à l'écran. On simule en réalité le parcours de la lumière en sens inverse, puisque, “physiquement”, les rayons vont vers l'utilisateur, et ne partent pas de son œil. Mais simuler le parcours de la lumière en faisant partir les rayons des sources de luminosité de la scène généreraient une quantité de calculs bien plus importante, et ne serait pas utile : il ne sert à rien de calculer l'éclairage d'une partie d'une scène 3D si cette partie n'est pas regardée par l'utilisateur.

La couleur de chaque pixel de l'image à afficher est donc la couleur du premier objet intersecté par le rayon associé à ce pixel. Avant d'afficher la couleur de cet objet, il faut s'assurer que cet objet est bien éclairé par une source lumineuse à l'intérieur de la scène virtuelle : il faut donc lancer un rayon d'ombre (*shadow ray*) entre le point d'intersection trouvé et la source lumineuse de la scène, et vérifier si un autre objet n'apparaît pas sur ce chemin. Si tel est le cas, le point d'intersection n'est pas éclairé, et le pixel associé est de couleur noire. Cette première version du lancer de rayons est une version assez basique, puisqu'elle ne considère que des sources ponctuelles, et des ombres parfaites. De plus, elle ne tient pas compte de possibles réflexions/réfractions subies par les rayons lumineux.

Cette première lacune a été résolue par Whitted [Whi80], qui a introduit un algorithme récursif permettant de prendre en compte des effets plus complexes. Ainsi, un rayon lumineux peut subir, lorsqu'il arrive à la surface d'un objet, une modification de son parcours, comme une réflexion, ou une réfraction. Il faut alors lancer un ou plusieurs rayons (dits “secondaires”) partant de ce point d'intersection (voir figure 1.5 pour un schéma de l'algorithme récursif). Pour chacun de ces rayons, on devra calculer la couleur du premier objet rencontré. Là encore, de nouveaux rayons lumineux pourront être générés, et ainsi de suite (c'est ici qu'apparaît la nature récursive de l'algorithme) jusqu'à aboutir à un calcul de la somme de toutes ces influences dans la couleur finale du pixel à afficher. D'autres raffinements de cet algorithme sont également apparus dans la littérature [CPC84, SM03], mais la primitive de base du lancer de rayons est toujours restée la même : chercher les intersections entre des rayons (dans une très grande majorité des cas, on se limite à la première intersection le long de chaque rayon) et des triangles.

Dans sa forme classique, le lancer de rayons présente deux différences majeures avec nos algorithmes. Premièrement, comme on l'a dit, la plupart des algorithmes de rendu s'arrêtent à la première intersection, ou aux premières intersections (prise en compte de la transparence de certains objets) le long de chaque rayon. Le besoin de stocker, dans notre contexte, l'intégralité des intersections, implique que l'on ne peut connaître à l'avance la quantité de mémoire

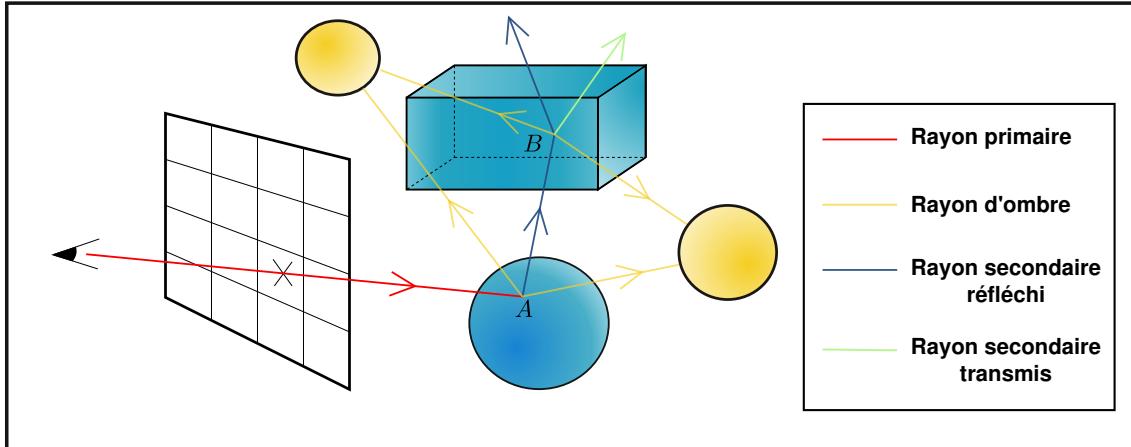


Figure 1.5 – Exemple de parcours d'un rayon dans la version récursive de l'algorithme de lancer de rayons. Le rayon primaire rouge est réfléchi par la sphère bleue (génération d'un rayon secondaire en *A*). Ce rayon réfléchi génère lui-même deux nouveaux rayons secondaires après intersection avec la boîte bleue (au point *B*). Deux rayons d'ombre sont générés en *A* et *B* afin d'évaluer l'intensité lumineuse en ces points.

nécessaire au stockage des intersections. Pour des architectures peu souples, comme celles du GPU, cela crée des contraintes non négligeables. La deuxième différence entre nos besoins et ceux de la communauté du lancer de rayons est plus à notre avantage : dans notre cas, les rayons d'ombre, ainsi que les rayons secondaires, n'existent pas. Les seuls rayons que nous prenons en compte lors de notre calcul de débit de dose sont des rayons arrivant au point de mesure du débit de dose, donc extrêmement “cohérents” d'un point de vue géométrique. Ils sont donc l'analogue des rayons partant de l'œil en lancer de rayons, dits “primaires”. Comme on le verra plus loin, cette différence nous permettra de mettre en place de nombreuses optimisations impossibles dans le cas classique.

Le lancer de rayons, même lorsqu'il se limite au traitement des rayons primaires, est néanmoins un algorithme extrêmement gourmand en ressources. En effet, lorsque les groupes de rayons et de triangles sont volumineux, l'accumulation des requêtes d'intersection provoque des temps de calcul importants. Bien que modélisant beaucoup plus fidèlement les lois de la physique, et donc générant des images de bien meilleure qualité, le lancer de rayons est ainsi resté jusqu'au début des années 2000 une technique hors-ligne, utilisée pour certaines applications spécifiques demandant une grande qualité d'image, mais à laquelle on préférerait systématiquement les techniques de rastérisation pour l'affichage temps-réel de scènes 3D. Néanmoins, les travaux de Wald en 2001 [WBWS01] ont montré qu'en concevant des algorithmes novateurs et en exploitant au maximum le matériel disponible, il était possible d'afficher plusieurs images par seconde d'une scène assez complexe (plusieurs centaines de milliers de triangles). Depuis cette publication, la communauté du lancer de rayons a été très active, cherchant à amener les performances du lancer de rayons au niveau de celles de la rastérisation, en particulier pour les scènes dynamiques, typiques, par exemple, des jeux vidéos. Dans les pages à venir, nous présentons donc les étapes importantes de ces recherches, sources majeures d'inspiration pour nos travaux.

1.3.2 Les structures d'accélération

Avant de parler des diverses possibilités d'accélération des calculs dans le contexte du lancer de rayons, il convient de rappeler qu'une approche naïve, consistant à tester, pour chaque rayon de la scène, s'il intersecte chaque triangle de la scène, amènerait des performances catastrophiques. Même sur une architecture hautement parallèle comme un GPU, la multiplication des tests d'intersection deviendrait impossible à gérer. Pour une scène constituée d'un million de rayons et de cent mille triangles, cent milliards de tests d'intersections seraient à réaliser, ce qui est aujourd'hui impossible en des temps compatibles avec une utilisation interactive. Il nous faut donc mutualiser un maximum de calculs pour améliorer nos performances. La première source d'optimisation des calculs vient d'une idée très naturelle : si un rayon part vers la partie "gauche" d'une scène, il ne sert à rien de chercher s'il intersecte les triangles de la partie "droite" de la scène. Afin d'accélérer les calculs, il est donc naturel de vouloir créer une hiérarchie sur les triangles de la scène, appelée structure d'accélération. Une bonne structure d'accélération doit ainsi permettre d'écartier un maximum de triangles en un minimum de temps.

1.3.2.1 Les grilles

La structure d'accélération la plus simple consiste à diviser régulièrement l'espace géométrique de la scène. On parle alors d'une grille régulière sur l'ensemble des triangles [AW87]. Pour le cas d'une grille 3D, l'espace est donc divisé en un ensemble de cubes, ou voxels. La construction de cette grille consiste à repérer, pour chaque triangle, les cubes de la grille qu'il intersecte. On crée ainsi, pour chaque cube de la grille, la liste des triangles qu'il contient. Ensuite, pour chaque rayon, la traversée de la grille consistera à identifier les cubes intersectés par ce rayon, puis à effectuer les tests d'intersection avec les triangles contenus par chacun de ces cubes (voir Figure 1.6). Il existe de nombreux algorithmes de parcours optimisés d'une grille [FTI86, AW87, Spa89], la plupart se rapprochant de l'algorithme de tracé de segment de Bresenham [Bre98].

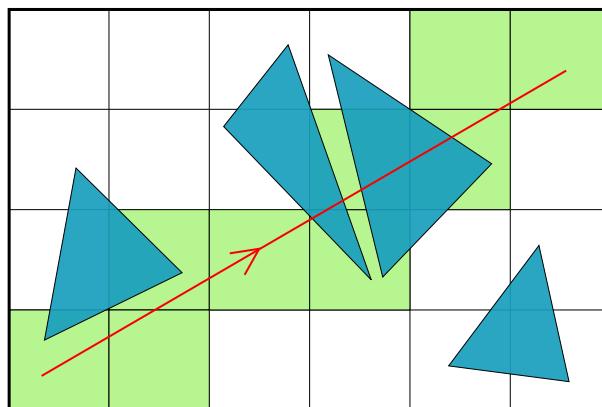


Figure 1.6 – Parcours d'un rayon dans une grille régulière. Pour chaque case verte parcourue par le rayon, on réalise les tests d'intersection entre le rayon et les triangles intersectant la case. Il est possible de ne pas refaire les tests d'intersection avec un même triangle tombant dans deux cases différentes (technique de *mailboxing*).

La grille régulière présente le grand avantage d'être extrêmement rapide à construire, mais sa structure régulière fait qu'elle n'est pas adaptée pour traiter des scènes de type *teapot in a*

stadium [KS97] (théière dans le stade), présentant des objets d'échelle très différentes en son sein. Afin de pallier ces défauts intrinsèques, de nombreuses améliorations ont été proposées [AK89], telles les hiérarchies de grilles [PPL⁺99, RSH00] ou les octrees [Gla88]. Ces nouvelles structures permettent d'obtenir des temps de traversée bien plus rapides, mais ne permettent pas d'atteindre les performances des deux structures que nous allons maintenant présenter.

1.3.2.2 Le kd-tree

La première de ces deux structures est le kd-tree [Ben75, FS88], qui a permis les premières démonstrations temps-réel de lancer de rayons [WBWS01]. Ces performances remarquables obtenues par Wald lui ont valu d'être la structure la plus utilisée au début des années 2000 (également en raison des travaux d'Havran [Hav00], qui la présentaient comme la structure la plus adaptée au lancer de rayons). Le kd-tree est un cas particulier de l'arbre BSP (Binary Space Partitioning) [FKN80]. Le BSP s'obtient en subdivisant récursivement l'espace en deux à l'aide d'un plan séparateur et en stockant la liste des objets contenus dans chaque feuille de l'arbre. Dans le cas du kd-tree, on se limite aux plans séparateurs alignés avec les axes du repère, ce qui simplifie grandement les calculs, et améliore donc nettement les temps de parcours de la structure. Une des principales caractéristiques du kd-tree vient du fait que lorsqu'un objet se situe de part et d'autre d'un plan séparateur, il faut dupliquer son identifiant dans les deux nœuds fils ainsi créés (voir Figure 1.7 pour un exemple de kd-tree).

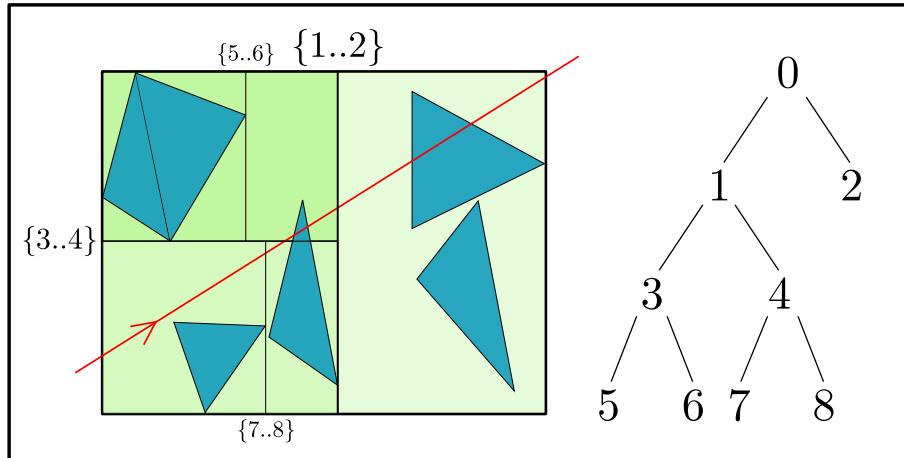


Figure 1.7 – Exemple de kd-tree sur une scène simple (6 triangles). Le rayon rencontre les feuilles 7, puis 8, 6 et 2. $\{i..j\}$ indique le plan séparateur des nœuds i et j . Le triangle chevauchant le plan séparateur des nœuds 3 et 4 doit être répertorié dans les feuilles numérotées 6 et 8.

La traversée d'un rayon se fait ensuite de manière récursive. En premier lieu, il convient de tester si le rayon intersecte la boîte englobante des triangles de la scène. Si tel est le cas, le parcours du kd-tree peut démarrer. Trois cas sont alors à identifier : soit le rayon intersecte uniquement le fils gauche de la racine de l'arbre, soit uniquement le fils droit, soit les deux à la fois. S'il n'intersecte qu'un seul des deux fils, la décision est simple : il poursuit la traversée avec le fils en question. S'il intersecte les deux, il faut calculer lequel des deux fils il parcourt en premier, calcul très rapide car le plan séparateur entre les deux fils est aligné avec les axes du repère. Supposons ici que le fils gauche soit le premier intersecté. On met alors le fils droit dans une pile, et on descend dans le fils gauche. C'est une descente en profondeur (*depth-first*), puisqu'on va effectuer tout le parcours du fils gauche avant de passer au fils droit. Pour chaque nouveau nœud rencontré, il va falloir appliquer les mêmes règles de parcours,

et donc, potentiellement, empiler des nœuds à parcourir ultérieurement. Lorsque le rayon rencontre un nœud qui est en fait une feuille de l'arbre, on effectue les tests d'intersection avec les triangles qu'elle contient. Si une intersection est trouvée, on peut arrêter le parcours de l'arbre (on se place ici dans le cas classique qui consiste à ne chercher que la première intersection le long du rayon). Sinon, on dépile le dernier nœud de la pile de nœuds restants à traiter, et on reprend le parcours de l'arbre. Cette traversée est dite “ordonnée”, puisqu'on parcourt les nœuds dans le même ordre que le rayon parcourt la scène. Cette propriété est très utile, car elle permet d'arrêter au plus tôt la traversée de l'arbre lorsqu'une intersection avec un triangle est trouvée.

1.3.2.3 Le BVH

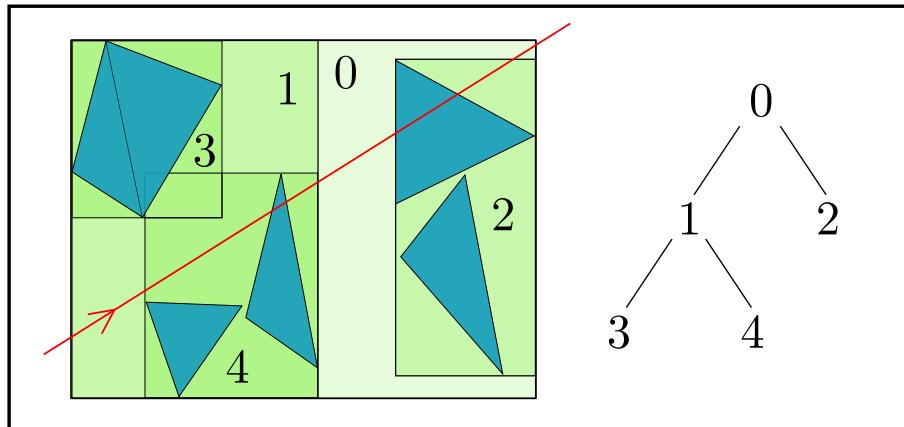


Figure 1.8 – Exemple de BVH sur la même scène que la figure 1.7. Le rayon parcourt ici les feuilles 4 et 2.

Le BVH, ou Bounding Volume Hierarchy [RW80, KK86], consiste également en une hiérarchisation de la scène, mais cette dernière n'est pas spatiale. La hiérarchisation se fait ici sur les objets de la scène. On l'utilise en général sous sa forme binaire, qui consiste donc à diviser les objets de la scène en deux groupes aussi “disjoints” que possible, et ainsi de suite, jusqu'à parvenir aux feuilles de l'arbre. Chacun de ces groupes d'objets est en principe représenté par sa boîte englobante, la sphère englobante étant parfois également usitée. Contrairement au cas du kd-tree, un objet ne peut pas ici se retrouver dans les deux fils d'un même nœud parent. La contrepartie est naturellement qu'il existe souvent une redondance d'espace entre deux noeuds voisins (voir figure 1.8). Le parcours du BVH se fait d'une manière très similaire à celui du kd-tree, avec, entre autres, l'utilisation d'une pile et un parcours la plupart du temps en profondeur. Néanmoins, comme deux noeuds fils d'un même parent peuvent se “chevaucher” géographiquement, il n'est en général pas possible de dire dans quel noeud va passer un rayon en premier. Il faut donc normalement parcourir les deux fils avant de produire la première intersection trouvée le long du rayon.

1.3.2.4 Comparaisons entre le kd-tree et le BVH

Après avoir effectué ces comparaisons sommaires, il convient déjà de comparer les différents défauts et qualités de ces deux structures “reines” dans le domaine du lancer de rayons, en particulier dans notre contexte de calcul de débit de dose. Le principal avantage du kd-tree réside dans son temps de traversée très rapide, dû au fait que les nœuds de l'arbre ne sont représentés que par des plans séparateurs. Le calcul d'intersection entre un rayon et ce plan

est très rapide, d'autant plus que ce plan est aligné avec les axes du repère. L'autre principal avantage de cette structure est qu'elle est extrêmement performante dans le cadre de la recherche de la première intersection, comme on l'a expliqué précédemment. Dans notre cadre, néanmoins, cet avantage n'existe plus. En ce qui concerne ses faiblesses, la profondeur d'un kd-tree est bien plus grande que celle d'un BVH, et nécessite donc davantage d'étapes de traversée de l'arbre. Son plus grand défaut provient cependant certainement de la nécessité de partager certains triangles entre plusieurs noeuds, ce qui rend le kd-tree beaucoup plus gourmand en mémoire, et peut rendre plus difficile sa construction.

De son côté, le BVH présente surtout le défaut d'étapes de traversée plus longues au niveau des noeuds intérieurs de l'arbre, puisqu'au lieu de tester l'intersection avec un plan, il convient de tester l'intersection avec deux boîtes englobantes filles. Cet important désavantage (la traversée non ordonnée n'en étant pas un dans notre cas) est compensé en partie par une occupation mémoire moins importante (dans un rapport de trois à quatre d'après [GPSS07]) et bornée (puisque une même primitive ne peut pas être référencée plusieurs fois). Un BVH permet également d'évincer plus vite des parties vides de l'espace, ce qui peut être extrêmement utile dans les parties hautes de l'arbre.

Il est intéressant de noter qu'une solution hybride [WK06] a été proposée pour conjuguer les qualités de ces deux structures. Si elle présentait des temps de construction remarquables, les performances obtenues en termes de temps de traversée n'étaient malheureusement pas assez bonnes.

1.3.3 Critères de qualité d'un arbre kd-tree ou BVH

Bien entendu, la manière de construire un kd-tree ou un BVH a un impact considérable sur les temps de parcours des rayons dans l'arbre. Diviser intelligemment un noeud en ses deux fils est un problème complexe, ayant reçu beaucoup d'attention dans la littérature. Il est également important de définir une stratégie d'arrêt de la construction de l'arbre, car il est rarement optimal de construire un arbre associant chaque triangle à une feuille différente. Pour les deux arbres, les critères de qualité finalement retenus par la communauté graphique sont très similaires, et nous les traiterons donc conjointement dans cette partie.

La première stratégie, sans doute la plus "naïve", consiste à diviser récursivement l'espace occupé par un noeud père suivant le critère de la médiane. Dans le cas d'une scène extrêmement homogène, cette division produit un arbre équilibré, et peut donc mener à des performances très honorables. Cependant, pour une très large majorité de scènes, cette division régulière de l'espace produit des arbres de très faible qualité, car peu équilibrés, et ne tenant pas du tout compte de la configuration géométrique particulière de la scène. Habituellement, le critère d'arrêt d'arbres construits selon ce modèle repose sur la définition d'un nombre maximal de triangles par feuille. Lorsqu'un noeud contient un nombre de triangles inférieur à cette quantité, il est immédiatement transformé en feuille, et la construction cesse.

Afin d'améliorer la qualité de l'arbre, il est donc nécessaire de choisir les plans de coupe de la structure d'accélération de manière beaucoup plus attentive, et de prendre en compte les particularités géométriques de la scène. Une stratégie réputée efficace consiste à maximiser l'espace vide dans les hauts niveaux de l'arbre, de manière à réduire considérablement le temps de parcours des rayons dans ces zones vides de primitives, donc "inutiles" pour notre problème. Cependant, ce genre de stratégie, comme de nombreuses autres, repose sur la définition de certaines constantes dépendantes de la scène. Or, nous voulons éviter ce genre de méthodes, et proposer des heuristiques permettant une automatisation des traitements sans qu'un utilisateur externe ait à indiquer des coefficients dépendants de la scène. La stratégie presque unaniment appliquée pour le domaine du lancer de rayons exploite le critère du

SAH, ou Surface Area Heuristic. Des recherches abondantes sur ce critère [GS87, MB90, Hav00] ont montré qu'il pouvait permettre la construction d'arbres extrêmement optimisés et adaptés au lancer de rayons. Bien que ce critère ne permette qu'une estimation de la qualité de l'arbre (il a d'ailleurs récemment été remis en question dans [PGDS09]), il reste la référence dans le domaine du ray-tracing, et la quasi-totalité des algorithmes de construction tentent de produire des arbres minimisant leur coût suivant ce critère.

Le critère SAH repose sur l'hypothèse que les rayons de la scène sont uniformément répartis dans l'espace, partant potentiellement de tous les points de la scène, et pouvant partir dans toutes les directions. Suivant ce principe, étant donnée une boîte mère B et sa boîte fille B_{fille} au sein de l'arbre, la probabilité qu'un rayon intersecte B_{fille} sachant qu'il intersecte B est donnée par :

$$P(B_{fille}|B) = \frac{S(B_{fille})}{S(B)} \quad (1.2)$$

où $S(B)$ désigne la surface de la boîte B , plus exactement la surface de son enveloppe. Le coût de subdivision d'une boîte B en deux sous-boîtes B_g et B_d (non forcément disjointes) est alors donné par la formule suivante :

$$C_{sub}(B, B_g, B_d) = C_{trav} + P(B_g|B).C(B_g) + P(B_d|B).C(B_d) \quad (1.3)$$

où C_{trav} représente le coût de traversée de la boîte, et $C(B_g)$ et $C(B_d)$ les coûts respectifs d'intersection des boîtes B_g et B_d . Il est à noter que le coût C_{trav} n'est pas le même pour un kd-tree et pour un BVH : comme on l'a vu, pour un kd-tree, la traversée d'un noeud consiste à calculer l'intersection du rayon avec un plan, alors que pour un BVH, il faut calculer l'intersection du rayon avec une boîte. Cette formule signifie tout simplement que le temps de parcours d'un rayon dans la boîte B est égal à son coût de traversée de la boîte, ajouté à la somme des temps de parcours de chacune de ses filles, pondérés par leur probabilité d'intersection.

A partir de ces deux formules, il est possible d'évaluer le coût associé à un arbre complet noté A :

$$C(A) = \sum_{n \in Noeuds} \frac{S(B_n)}{S(B_S)} C_{trav} + \sum_{f \in Feuilles} \frac{S(B_f)}{S(B_S)} |f| C_{inter} \quad (1.4)$$

où $S(B_S)$ est la surface de la boîte englobante de l'ensemble de la scène S , C_{inter} le coût d'intersection entre un rayon et un triangle, et $|f|$ le nombre de triangles au sein d'une feuille f . On considère ici que le coût d'intersection entre un rayon et une feuille f se réduit à $|f|$ tests d'intersections rayon-triangle, sans tenir compte du coût d'accès à la feuille, a priori négligeable. De plus, prendre en compte ce coût conduit apparemment à des arbres de moins bonne qualité [WH06]. Le meilleur arbre suivant le critère SAH est donc un arbre permettant de minimiser ce coût global. Cette recherche de minimum global est un problème de type NB-Complet [Hav00]. Pour des raisons évidentes, il est impossible d'imaginer considérer l'ensemble des noeuds et l'ensemble des feuilles possibles dans le cadre d'une simulation interactive.

La simplification communément acceptée consiste à construire un arbre de haut en bas, et à rechercher, pour chaque nouveau noeud à diviser, un coût minimum local de parcours. La simplification consiste à supposer que les deux fils potentiels résultant d'une division du noeud considéré sont des feuilles de l'arbre. La formule 1.3 devient alors :

$$C_{sub}(B, B_g, B_d) = C_{trav} + \frac{S(B_g)}{S(B)}.|B_g|.C_{inter} + \frac{S(B_d)}{S(B)}.|B_d|.C_{inter} \quad (1.5)$$

où $|B_g|$ et $|B_d|$ indiquent respectivement le nombre de triangles dans les nœuds gauches et droits issus de la subdivision de B . Cette approximation locale du coût de subdivision tend à une surestimation de ce coût, puisque les fils gauches et droits du nœud considéré pourraient eux aussi être subdivisés. Néanmoins, cette approximation, pouvant paraître majeure, donne généralement d'excellents résultats, et est abondamment utilisée par les chercheurs de la communauté du lancer de rayons.

En minimisant ce coût de subdivision pour chaque nouveau noeud à traiter, on crée ainsi un arbre d'excellente qualité pour le lancer de rayons. De plus, ce critère permet aussi de comparer ce coût de subdivision à un coût de non-subdivision (faire du nœud à traiter une feuille), et fournit ainsi un critère d'arrêt de construction de l'arbre plus pertinent que les habituelles heuristiques (profondeur maximale de l'arbre ou nombre de triangles par feuille inférieur à une quantité arbitraire).

Pour un kd-tree, il est facile de démontrer [Hav00] que les plans de coupe permettant un coût de subdivision minimal correspondent à des frontières de triangle (“début” ou “fin” d’un triangle suivant l’un des trois axes du repère). Pour un BVH, comme la séparation n'est pas spatiale, mais se fait sur les objets, le minimum de coût est plus difficile à atteindre. Pour simplifier les calculs, on se limite généralement à tester des plans de coupe potentiels répartis suivant chacun des trois axes du repère, et à identifier la position des triangles par rapport à ces plans de coupe (normalement via la position de leur centre de gravité) pour déterminer s'ils doivent appartenir au fils gauche ou droit du nœud étudié. Une fois le plan optimal trouvé, on calcule les boîtes englobantes des deux groupes de triangles trouvés afin de poursuivre la construction de l'arbre.

Bien que discutée, la construction d'un arbre BVH ou kd-tree via la recherche d'un coût minimal en regard du critère SAH reste abondamment utilisée par la communauté du lancer de rayons, car elle permet la construction d'arbres prenant bien en considération la configuration de la scène, et ne repose sur aucune hypothèse quant à la répartition des rayons au sein de la scène. Néanmoins, une fois ces structures construites, de nombreuses optimisations sont encore réalisables au cours du parcours de l'arbre.

1.3.4 Techniques d'accélération pour des groupes de rayons cohérents

1.3.4.1 Sur CPU

En effet, en étudiant la construction de ces structures d'accélération, nous avons en fait montré comment il était possible de mutualiser certaines informations sur des triangles proches géographiquement, en les regroupant ensemble, et en faisant un test d'intersection sur le groupe (nœud de l'arbre), plutôt que sur les triangles. La deuxième technique est celle qui a réellement amené le lancer de rayons vers des temps de calcul interactifs, et consiste pour sa part à mutualiser les calculs sur les rayons proches géographiquement : l'idée est que pour un groupe de rayons dits “cohérents”, la traversée de la structure d'accélération doit être très similaire, voire même identique. On va donc chercher à traiter ces rayons par groupe, et non plus à les considérer tous séparément. On espère ainsi économiser des calculs, mais également des accès récurrents à la mémoire.

La première contribution majeure dans ce domaine est celle de Wald [WBWS01], qui lance les rayons primaires par groupes de quatre dans un kd-tree, en utilisant le jeu d'instructions SSE de sa machine pour réaliser les calculs en parallèle sur un tel groupe. Concrètement, les rayons d'un même groupe effectuent exactement la même traversée de l'arbre. Si, à un moment de la traversée, le parcours de ces rayons diverge, on marque comme inactifs les rayons n'intersectant pas le nœud actuellement parcouru. Ils font alors les mêmes calculs

que les autres rayons du groupe, mais leurs données d'intersections ne sont pas mises à jour si une intersection est trouvée. Ce comportement peut créer une perte de temps non négligeable lorsque des rayons non cohérents composent ce groupe de quatre, mais dans le cadre de rayons cohérents, ces divergences sont rares (un facteur d'accélération de 3.5 est obtenu sur des paquets de rayons cohérents). Wald parvient ainsi à rendre la scène référence de la Conférence (274 000 triangles) à près de 2 fps (*frames per second*, images par seconde).

Ces travaux sont poursuivis en 2005 [RSH05], avec l'algorithme de MLRTA (Multi-Level Ray Tracing Algorithm). Cet algorithme repose principalement sur une technique de frustum-culling consistant à utiliser un groupe de 4 rayons englobant un grand groupe de rayons. Ces 4 rayons servent à éliminer rapidement les noeuds n'intersectant pas l'intérieur du faisceau. En fonction du parcours effectué dans l'arbre, ces faisceaux peuvent ensuite être subdivisés, voire ramenés à des rayons individuels. La scène de la Conférence est ici rendue à 9.5 fps avec un seul cœur, et à 15.6 fps avec deux. Les performances obtenues sont cependant difficiles à reproduire [CL07], en raison d'une optimisation poussée du code pour l'architecture utilisée. Ces deux publications majeures représentent néanmoins, encore aujourd'hui, la référence du lancer de paquets de rayons dans un kd-tree.

Le kd-tree n'est cependant pas la seule structure adaptée à une mise en commun des calculs effectués au sein d'un paquet de rayons. Des algorithmes très performants ont ainsi également été proposés pour le parcours d'un paquet de rayons dans une grille [WIK⁺06], mais surtout dans un BVH [LYM06, WBS07]. Dans ce dernier papier, Wald propose des algorithmes adaptés à des groupes de rayons bien plus gros ($8 * 8$, $16 * 16$), faisant du BVH une structure encore plus adaptée au parcours de paquets de rayons que le kd-tree. Il s'appuie sur la structure particulière du BVH, qui permet entre autres de faire descendre un rayon dans un noeud d'une hiérarchie sans avoir à calculer son intersection exacte avec le noeud. Cette optimisation n'est pas possible avec un kd-tree, où un rayon doit être mis à jour au fur et à mesure de son parcours de la structure d'accélération. En utilisant cette propriété du BVH, Wald peut ainsi éviter un très grand nombre de tests d'intersection rayon-boîte, quitte à devoir faire quelques calculs excessifs vers les niveaux bas de l'arbre. Alliée à plusieurs autres optimisations, cette innovation permet à Wald d'atteindre des performances très proches de celles du kd-tree (9.3 fps sur la Conférence), meilleures, même, pour les rayons primaires [CL07]. Elle a surtout permis au BVH de devenir la structure d'accélération privilégiée pour les algorithmes de lancer de rayons depuis quelques années (pour le traitement des scènes dynamiques en particulier).

Face à la tendance actuelle, les chercheurs ont ensuite souhaité adapter ces algorithmes à des architectures plus largement parallèles (travaux de Benthin, par exemple, sur les architectures CELL[BWSF06] ou Larrabee[BW09], ou algorithmes de *stream filtering* [WGBK07, GR08] pour des architectures parallèles futures). En particulier, de nombreux travaux ont été menés sur GPU, dont certains particulièrement remarquables dans les dernières années.

1.3.4.2 Techniques d'accélération sur GPU

Les premières tentatives de portage de lancer de rayons sur GPU remontent au début des années 2000. Carr[CHH02] détourne ainsi le pipeline graphique pour effectuer les tests d'intersection rayon-triangle, mais réalise le reste de l'algorithme sur le CPU. Les temps de transfert de données entre le CPU et le GPU sont alors beaucoup trop élevés pour espérer atteindre des temps de calcul interactifs. Pour remédier à ce problème, Purcell [PBMH02] propose d'implémenter l'intégralité de l'algorithme sur GPU. Il utilise une grille régulière comme structure d'accélération, mais reste très loin des performances CPU de l'époque.

On observe les premiers algorithmes de descente de kd-tree sur GPU en 2005 [FS05] (codage en Brook for GPU [BFH⁺04]). La construction du kd-tree se fait toujours sur CPU, limitant ces traitements au cas des scènes statiques. Afin d'éviter d'avoir à utiliser la pile par rayon propre au parcours du kd-tree, Foley propose de nouveaux algorithmes de parcours, dont le *kd-restart*, consistant simplement à reprendre le parcours à la racine de l'arbre lorsqu'un rayon vient d'effectuer les tests d'intersections avec les triangles d'une feuille de l'arbre, qu'aucune intersection n'a été trouvée, et que le rayon doit donc continuer sa traversée.

Il faut attendre 2007 pour voir les premières implémentations rivalisant réellement avec le CPU. [HSHH07] propose une modification de l'algorithme de kd-restart de Foley, exploite les paquets de rayons, et utilise une pile de taille réduite. Sur la scène de la Conférence, il obtient des performances de l'ordre de 5 fps (ses performances globales ne sont pas clairement indiquées car, s'il est excellent pour le lancer des rayons primaires, il est nettement moins efficace pour le traitement des rayons générés après réflexion sur un objet). Concurrentement, [PGSS07] propose un algorithme sans pile, exploitant également les paquets, implémenté en CUDA. Les feuilles de l'arbre sont ici reliées à des cordes (*ropes*), permettant de connaître les noeuds de sortie vers lesquels reprendre la traversée de la structure d'accélération, plutôt que de repartir de la racine de l'arbre, comme pour l'algorithme kd-restart (6.7 fps sur la Conférence). Toujours afin de contourner le problème de la pile, il a été également proposé d'utiliser une pile (allouée en mémoire partagée) commune à tous les threads d'un même bloc [GPSS07], pour une performance de 6.1 fps sur la scène de la Conférence.

En 2009, une étude particulièrement intéressante (voir [AL09]) montre que cette manière d'implémenter la pile n'est pas optimale, et qu'il est plus utile de maintenir une pile par rayon, de manière à économiser de nombreux calculs. La pile est ici stockée pour chaque rayon en mémoire locale, qui, comme on l'a vu, est en réalité aussi peu rapide que la mémoire globale, et contient toutes les données propres au thread qui ne peuvent pas être contenues dans les registres ou en mémoire partagée. Grâce à une implémentation particulièrement optimisée sous CUDA, [AL09] présente ainsi les premières performances GPU dépassant clairement leur équivalent CPU.

La publication qui fait aujourd'hui référence dans le domaine du lancer de paquets de rayons sur GPU [GL10] consiste en fait en une inversion du problème, qui permet de ne plus avoir besoin de cette pile qui complique le portage de l'algorithme sur GPU. De manière générale, le parcours d'arbre pour le domaine du lancer de rayons se fait en profondeur. Le parcours en largeur [Han86] a lui été beaucoup moins utilisé, limité à certains besoins précis, comme la limitation de l'espace mémoire occupé par un BVH pour de très grosses scènes [MW04], ou le stream filtering [GR08]. Ici, le parcours de l'arbre en largeur est effectué par les groupes de rayons. Tous les paquets de rayons parcourront le BVH en même temps pour un niveau de profondeur donné, un nouveau kernel étant lancé pour chaque nouveau niveau. Au niveau des feuilles de l'arbre, on reconside les rayons individuellement, et on effectue les tests d'intersections avec les triangles des feuilles. Bien que cette méthode puisse potentiellement faire parcourir de nombreux niveaux à un rayon qui aurait déjà dû arrêter sa traversée (puisque c'est le groupe de rayons qui "décide" pour lui), elle s'avère extrêmement efficace pour les rayons primaires (69 millions de rayons traités par seconde sur la scène de la Conférence, contre 46 pour [AL09]).

1.3.5 Gestion des scènes dynamiques

En combinant des arbres optimisés pour le lancer de rayons à ces méthodes de lancer de paquets de rayons cohérents, il est donc possible d'obtenir des performances interactives, même pour des scènes complexes. Cependant, nous avons jusque là négligé un point essentiel, à

savoir que les objets d'une scène peuvent être en mouvement, voire apparaître ou disparaître de la scène (par exemple, lors de l'explosion d'un objet dans un jeu vidéo). La structure d'accélération doit alors être capable d'évoluer à chaque pas de temps de la simulation. Il y a une dizaine d'années, les structures d'accélération associées à une scène moyenne demandaient plusieurs secondes pour être reconstruites intégralement, rendant le traitement d'une scène dynamique en temps réel absolument impossible. Le sujet a néanmoins été abondamment étudié au cours des précédentes années, en particulier dans le contexte de la programmation hautement parallèle sur GPU.

1.3.5.1 Adapter la structure d'accélération

Alors que les temps de construction des structures d'accélération étaient absolument prohibitifs dans le cadre d'un traitement temps-réel, les premières solutions proposées ont naturellement consisté en une mise à jour de la structure lors du déplacement des objets d'une scène (plutôt qu'une reconstruction totale pour chaque nouveau pas de temps). Les premiers essais ont consisté à mettre à jour la structure la plus facile et rapide à construire, à savoir la grille [RSH00, LLAm01]. Des essais similaires ont été effectués pour le kd-tree, mais ils se limitaient au traitement d'un mouvement hiérarchique ou connu à l'avance [WBS03, GFW⁺06]. En comparaison du kd-tree, le BVH paraît beaucoup plus simple à modifier : en effet, lorsqu'un triangle bouge dans une scène, il suffit de mettre à jour la boîte englobante de la feuille le contenant (cette feuille étant unique dans le cas du BVH), puis de mettre à jour, si besoin est, les boîtes englobantes des noeuds parents. Cette stratégie a été proposée conjointement dans deux publications [LYM06, WBS07], pour traiter efficacement des scènes dynamiques avec des objets en mouvement. Néanmoins, ce simple traitement peut conduire à des BVHs de qualité très dégradée après un certain nombre de pas de temps. [LYM06] propose donc une heuristique afin d'évaluer la dégradation de l'arbre. Lorsque l'arbre est considéré comme trop dégradé, il est reconstruit. Cette stratégie produit néanmoins des ralentissements notables du rendu au moment de la reconstruction du BVH. Pour pallier ce problème, on peut alors, dans un contexte multi-CPU, effectuer une reconstruction asynchrone de l'arbre [IWP] : si l'on dispose de N threads en simultané, pendant que $(N - 1)$ threads s'occupent du rendu et du parcours du BVH, le dernier thread s'occupe de la reconstruction de l'arbre.

Toutes ces solutions peuvent apporter des performances intéressantes, mais ne conviennent qu'à des scènes particulières, avec des objets en mouvement. Pour des scènes moins cohérentes, avec des explosions d'objets et des apparitions/disparitions de primitives, il faut alors être capable de reconstruire intégralement la structure d'accélération en un temps le plus court possible (certains algorithmes [Gar08] ont néanmoins été proposés afin de déterminer quelles parties d'un BVH doivent être complètement reconstruites et lesquelles peuvent juste être mises à jour). Dans les paragraphes à venir, nous présentons ces stratégies, multi-CPU comme GPU. La plupart reposent sur une construction rapide d'arbres légèrement dégradés vis-à-vis du critère SAH (cette dégradation étant parfois à peine perceptible au niveau des performances du lancer de rayons).

1.3.5.2 Construction optimisée de structures d'accélération sur CPU

De nombreuses stratégies ont été proposées pour optimiser la construction d'un arbre suivant le critère du SAH. Des algorithmes extrêmement optimisés ont été proposés pour une construction de kd-tree prenant en compte l'ensemble des plans potentiels de coupe [Sze03, WH06], en triant en amont les "débuts/fins" de triangles (nommés *événements*) suivant chaque axe du repère, et en balayant cet ensemble au cours de la recherche du plan

optimal de subdivision. Une fois la subdivision effectuée, il est possible de répartir les primitives dans le nouveau sous-arbre en maintenant ce tri pré-établi, et d'économiser ainsi de nombreux calculs.

Néanmoins, pour une approche temps-réel, cette évaluation exhaustive des plans de coupe potentiels reste trop gourmande en temps de calculs (15 secondes pour construire un kd-tree sur la scène de la Conférence chez [WH06]), et des approches alternatives ont ainsi été développées afin de construire des arbres de qualité “dégradée”, mais aux temps de construction très rapides, et permettant des temps de traversée proches des arbres construits suivant des méthodes plus classiques.

La solution choisie pour légèrement dégrader la qualité de l'arbre consiste souvent à séparer la construction des niveaux hauts et bas de l'arbre. En effet, l'évaluation exhaustive des niveaux hauts de l'arbre est particulièrement lourde en termes de temps de calculs. On propose donc la plupart du temps d'évaluer un certain nombre N de plans régulièrement espacés suivant les axes du repère. Pour cela, on cherche pour chaque primitive dans quel intervalle (entre deux plans consécutifs) elle est située. Une fois cette opération faite, il est possible de balayer efficacement ces informations (nombre de triangles par intervalle) afin de calculer, pour chaque plan, le nombre de triangles situés à gauche ou à droite de ce plan. En comparant les coûts calculés pour chaque plan, on obtient ensuite facilement le plan de coupe optimal. Cette stratégie, lorsqu'elle se limite aux niveaux hauts de l'arbre, ne dégrade que légèrement la qualité de l'arbre, et permet des gains importants en temps de construction. De plus, elle est applicable aussi bien au kd-tree [PGSS06, SAA07] qu'au BVH [Wal07]. Elle est aussi très bien adaptée à des approches multi-CPU, puisqu'il est possible de paralléliser efficacement ce tri par intervalles pour les niveaux hauts de l'arbre, en répartissant les primitives suivant les différents processeurs, avant de fusionner les différentes quantités calculées (grâce à ces techniques, Wald [Wal07] est capable de construire, sur une machine avec 8 cœurs, un BVH sur la scène de la Conférence en 26 ms seulement, pour une perte de performance de 14% par rapport à un BVH construit via la méthode classique).

Une fois les niveaux hauts de l'arbre traités, on peut revenir à un critère SAH plus exact pour les niveaux bas de l'arbre (chez [SAA07], mais pas chez [Wal07], car l'évaluation de plans régulièrement espacés reste pertinente pour un BVH), en attribuant le traitement de chaque nouveau noeud à un thread disponible. Il est à noter qu'il est également possible de majorer l'erreur faite via l'utilisation de cet échantillonnage de plans potentiels (par rapport à une évaluation exhaustive des plans de coupe), en commençant par utiliser un espace régulier pour estimer les premiers coûts, et en ré-échantillonnant plus finement au niveau des intervalles présentant une trop grosse variation de coûts SAH [HMS06]. Il est au passage très intéressant de noter qu'avec l'augmentation des forces de calcul parallèle, la profondeur à partir de laquelle le nombre de noeuds à traiter devient plus faible que le nombre de processeurs disponibles grandit elle aussi, impliquant une dégradation de plus en plus importante de la qualité de l'arbre. Partant de ce constat, [CKL⁺10] propose un algorithme de construction de kd-tree multi-CPU très optimisé, et ne faisant pas appel à un critère SAH simplifié pour les niveaux hauts de l'arbre (116 ms sur une machine avec 32 cœurs pour construire un kd-tree sur la scène Fairy, comprenant 172 000 triangles).

1.3.5.3 Construction optimisée de structures d'accélération sur GPU

Naturellement, depuis quelques années, plusieurs travaux ont également porté sur la construction de telles structures d'accélération sur GPU. Cette question demandait un soin particulier, car jusqu'en 2007, les structures d'accélération étaient construites sur CPU, puis transférées sur GPU avant parcours de la structure d'accélération, rendant impossible un

rendu temp-réel des scènes dynamiques. Néanmoins, là aussi, des solutions très performantes ont également été trouvées, dépassant même les résultats obtenus sur des architectures multi-CPU. On peut ainsi noter un algorithme très efficace de construction de grille [KS09], très similaire à celui que nous avons conçu en parallèle (et que nous présenterons en détails dans le chapitre suivant). Cet algorithme a très récemment été étendu pour la création d'une structure d'accélération à deux niveaux de grille, avec une grille à gros grain, et, à l'intérieur des cases de cette grille, une grille plus fine, la résolution de cette grille fine variant suivant le nombre de triangles trouvés dans la case en question [KBS11]. Cette structure présente l'avantage d'être extrêmement rapide à construire (17 ms sur la scène de la Conférence), et de présenter des temps de parcours bien meilleurs que ceux de la simple grille régulière.

Concernant les structures d'accélération plus populaires comme le kd-tree ou le BVH, des stratégies ont également été proposées ([ZHWG08] pour le kd-tree, [LGS⁺09] puis [PL10] pour le BVH). Toutes reposent sur le même principe que pour les architectures multi-CPU, avec des stratégies différentes pour les niveaux hauts et bas de l'arbre. Pour les niveaux bas de l'arbre, le critère SAH est utilisé, en confiant le traitement d'un nœud à un bloc de threads. Une solution robuste et simple à mettre en place consiste à lancer un kernel par niveau, en disposant d'une liste de nœuds en entrée, et d'une liste de nœuds en sortie (deux fois la taille de la liste en entrée, car chaque nœud peut générer deux nœuds fils). Pour chaque niveau de l'arbre, les nœuds en entrée sont traités en parallèle. Si un bloc traitant un nœud génère des nœuds fils, il les écrit dans la liste de nœuds en sortie. Cette liste est ensuite "nettoyée" via une opération de compactage pour éliminer les nœuds vides, et l'opération est renouvelée pour le niveau suivant.

Pour les niveaux hauts de l'arbre, le problème est plus difficile : comme il avait été noté dans [Wal07], l'algorithme de construction du haut de l'arbre utilisant le tri par intervalles ne présentait pas une très bonne extensibilité (*scalability* en anglais) pour un nombre de coeurs croissant. [Wal07] suggérait d'utiliser la subdivision via le critère de la médiane pour ces niveaux hauts de l'arbre, pour des performances moins bonnes, mais une extensibilité bien meilleure. Fort logiquement, étant donné la structure hautement parallèle du GPU, [ZHWG08] et [LGS⁺09] utilisent ce critère de la médiane pour les niveaux hauts de l'arbre, avec un tri astucieux utilisant les codes de Morton chez [LGS⁺09]. Ce tri a été amélioré par [PL10], avec un critère SAH simplifié pour les niveaux très hauts de l'arbre (là où la qualité de l'arbre devrait être la plus importante), pour des performances surprenantes, un BVH de bonne qualité sur la scène Stanford Dragon, comprenant 871 000 triangles, pouvant être construit en 95 ms.

En conclusion de cet état de l'art sur le lancer de rayons et les structures d'accélérations associées, nous pouvons dire que des performances impressionnantes ont pu être atteintes au cours de ces dernières années, grâce à la conception d'algorithmes innovants adaptés aux architectures parallèles et hautement parallèles. Le BVH, de par sa rapidité de construction et son traitement efficace de paquets de rayons cohérents, semble être le meilleur candidat pour notre problème. De plus, comme on l'a déjà dit, l'avantage important du kd-tree pour la recherche de la première intersection le long des rayons est inexistant dans notre cas. Néanmoins, pour plusieurs raisons que nous allons expliquer maintenant, nous avons choisi de ne pas chercher à exploiter le BVH pour nos algorithmes, et de nous concentrer sur une autre structure d'accélération, la grille en perspective, introduite par Hunt en 2008 [HM08b].

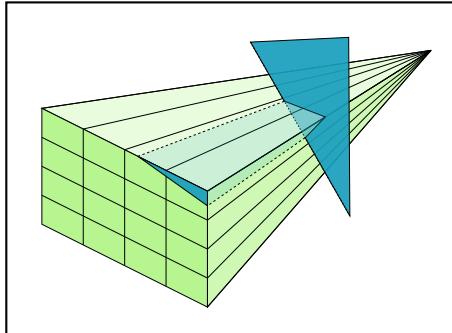


Figure 1.9 – Schéma d'une grille en perspective vue en trois dimensions, montrant la projection d'un triangle sur la grille régulière.

1.3.6 La grille en perspective

1.3.6.1 Présentation de la grille en perspective

La grille en perspective (voir Figure 1.9) est une grille régulière dans un espace en perspective centré en un point de projection. Dans le cas de rayons partageant le même point de départ, comme les rayons primaires, ce point de projection est tout simplement le point de convergence des rayons. Pour une description plus intuitive, on peut dire que la grille en perspective est le résultat de la division régulière de l'image vue à travers une caméra centrée en ce point de projection de l'espace. Pour décrire tout l'espace 3D autour d'un point donné, il faut donc utiliser conjointement six grilles, qui correspondent en fait aux six images que l'on pourrait voir sur les six faces d'un cube virtuel autour du point de projection. À cette grille 2D, on peut aussi ajouter une information de profondeur, et ainsi créer une grille 3D. Cette structure est en soi une structure d'accélération, au même titre que la grille régulière classique, le kd-tree ou le BVH. De la même manière, un rayon peut traverser cette grille, et on peut concevoir un algorithme permettant de trouver les différentes cases de la grille parcourues par le rayon. S'il est donc possible d'utiliser la grille en perspective pour toute sorte de rayons [HM08a], elle reste néanmoins nettement plus adaptée au traitement des rayons primaires. Nous allons maintenant étudier plus précisément pour quelles raisons cette structure est particulièrement adaptée à notre problématique et pourquoi nous avons donc décidé de l'utiliser au cours de nos travaux de recherche.

1.3.6.2 Les avantages de la grille en perspective pour notre algorithme

Les structures d'accélération classiques, telles que le BVH ou le kd-tree, ne se basent sur aucune hypothèse relative à la disposition des rayons ou des triangles de la scène. Cette propriété permet d'accorder un traitement équivalent à tous les types de rayons, mais elle empêche de mettre en place un parcours optimisé pour certains rayons aux trajectoires prévisibles, comme les rayons primaires. De son côté, la grille en perspective tient compte de l'organisation géographique particulière des rayons primaires pour leur offrir des temps de parcours minimaux. Sur la figure 1.10, on a comparé le traitement d'une scène simple constituée de 5 triangles, dans laquelle le rayon rouge intersecte deux triangles. Pour le cas du BVH, il va falloir tester l'intersection du rayon avec les deux boîtes, puis, pour chacune de ces boîtes, faire les tests d'intersections avec les triangles qu'elles contiennent. On obtient donc 2 tests d'intersections rayon-boîte et 5 tests d'intersections rayon-triangle, contre seule-

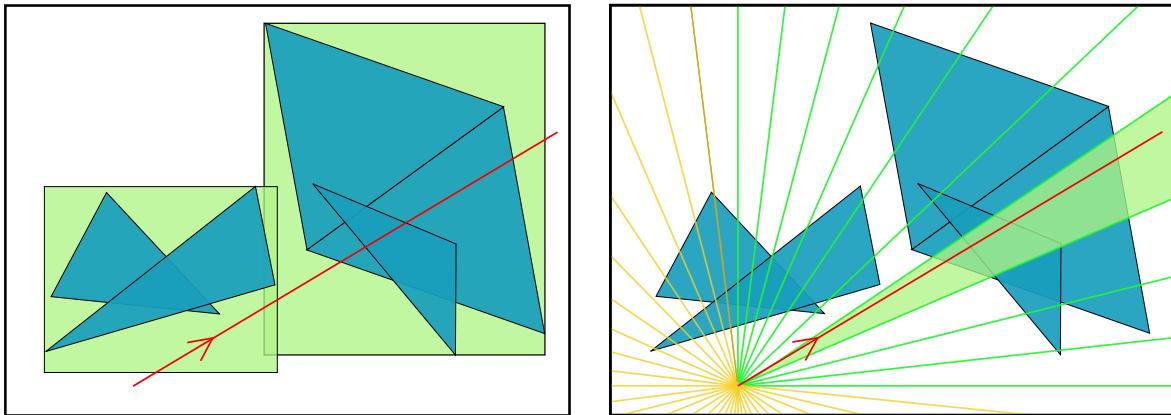


Figure 1.10 – Comparaison entre le parcours effectué par un rayon dans un BVH et dans une grille en perspective centrée au point de départ du rayon.

ment 2 tests d’intersections rayon-triangle pour le cas de la grille perspective, puisqu’on sait déjà les triangles que la case associée au rayon contient. L’orientation de cette structure d’accélération en fonction de la direction des rayons de la scène permet donc un traitement des rayons primaires extrêmement efficace, bien plus performant que les structures classiques. Il est d’ailleurs à noter que cette propriété est majoritairement due à la nature “en perspective” de notre grille, et qu’un BVH ou un kd-tree dans cet espace en perspective permettraient eux aussi des gains de temps importants par rapport à leur équivalent classique. Néanmoins, comme, dans notre cas, au sein d’un groupe de rayons, la répartition des rayons est assez régulière, nous pensons que la grille n’est pas fortement désavantagée par rapport aux structures plus classiques.

Surtout, l’utilisation de la grille régulière, par rapport à une structure plus complexe, permet de réduire à néant, ou presque, l’étape de parcours de la structure d’accélération, puisqu’il suffit, pour un rayon tombant dans une cellule donnée, de consulter la structure d’accélération et de récupérer la liste de triangles intersectant cette cellule. Dans notre contexte de recherche exhaustive (et surtout de stockage) des intersections au sein de la scène, cette propriété est particulièrement importante. Avec une structure plus complexe, dans le cas d’un traitement classique “1 thread = 1 rayon”, il faudrait effectuer un premier parcours de la structure d’accélération pour chaque rayon, compter le nombre de tests d’intersections à effectuer, puis faire une somme cumulée sur ces informations pour savoir où stocker les résultats des tests d’intersections pour chaque rayon. Là, il faudrait à nouveau effectuer le parcours de la structure d’accélération pour chaque rayon, pour enfin pouvoir effectuer les tests d’intersections, et stocker les résultats. Avec un BVH ou un kd-tree, même dans un espace en perspective, il nous faudrait donc répéter deux fois le parcours du rayon dans la structure d’accélération pour chaque rayon, ce que nous évitons avec la grille. Cette différence est réellement propre à notre recherche exhaustive des intersections de la scène, puisque dans le cas où l’on se limiterait à la première intersection le long de chaque rayon, il nous suffirait de mettre peu à peu à jour l’information sur l’intersection la plus proche rencontrée.

Outre ces grandes qualités vis-à-vis de notre problème, la grille en perspective présente néanmoins un défaut majeur : lorsque le point de départ des rayons, dans notre cas le point de calcul du débit de dose, évolue, elle doit elle aussi évoluer. Cela nous force donc à devoir reconstruire la structure d’accélération à chaque pas de temps, même pour des scènes statiques. Comme on va le voir plus loin, il est néanmoins possible de reconstruire cette structure en des temps très faibles (plus rapidement qu’un BVH ou un kd-tree, c’est là encore un autre atout). Cet inconvénient de la grille en perspective devient alors un avantage : comme le traitement

des scènes statiques et dynamiques est le même, le fait de pouvoir reconstruire cette structure très rapidement, et donc de traiter des scènes statiques en des temps interactifs, nous permet également de traiter des scènes dynamiques en des temps interactifs.

Nous présenterons dans le chapitre 2 notre algorithme GPU optimisé de construction/parcours de la grille en perspective, qui nous permettra de traiter les scènes statiques comme dynamiques en des temps interactifs.

1.4 Conclusion

Dans cet état de l'art, nous avons introduit les principes fondamentaux de la radioprotection et présenté la méthode d'atténuation en ligne droite avec facteurs d'accumulation, permettant d'estimer rapidement, durant des phases d'avant-projet, les quantités de radiations reçues par des opérateurs intervenant sur des sites irradiés. Cette méthode repose sur une modélisation simplifiée du rayonnement, s'effectuant ici en "ligne droite", les facteurs d'accumulation calculés devant permettre de corriger les approximations ainsi effectuées. Géométriquement, l'utilisation de cette méthode implique que le calcul du débit de dose s'effectue à partir de la création de rayons partant des sources de radiations de la scène, et arrivant tous au point de mesure du débit de dose. Le calcul de ce débit de dose demande l'identification et le tri de toutes les intersections le long de chaque rayon généré.

Ce problème étant intrinsèquement parallèle, nous avons décidé d'utiliser les architectures ultra-parallèles des processeurs de cartes graphiques (GPUs) afin de le résoudre. Nous avons d'abord démontré que l'exploitation du pipeline graphique, directement implanté au niveau du matériel sur ces cartes, ne nous permettait pas de répondre à nos besoins. Cette contrainte nous impose de concevoir nos propres algorithmes adaptés aux GPUs, et donc d'utiliser les interfaces de programmation dédiées développées au cours des années précédentes. Nous avons ainsi présenté l'interface de programmation CUDA, dédiée aux matériels NVIDIA, que nous utilisons pour nos travaux.

Nous avons ensuite présenté les principales techniques utilisées afin d'accélérer les temps d'exécution de l'algorithme de lancer de rayons, sur CPU comme sur GPU. Nous avons déduit de cette étude que des structures d'accélération hiérarchiques comme le BVH ou le kd-tree permettaient d'obtenir des performances très intéressantes, mais n'étaient pas assez adaptées à notre contexte d'utilisation, dans lequel tous les rayons de la scène partagent le même point de départ. Nous avons alors présenté la structure d'accélération que nous exploiterons dans les prochains chapitres de ce manuscrit, à savoir la grille en perspective. De par sa forme particulière, cette structure permet un traitement optimisé des rayons dits "primaires", correspondant à ceux que nous devons traiter.

Nous présenterons donc dans les chapitres suivants nos méthodes optimisées de construction et de parcours de ces grilles en perspective, ainsi que les techniques nous permettant de garantir la cohérence de nos résultats lors du travail dans un espace projectif.

Triangle-tracing

2

Sommaire

2.1	Modifications de l'algorithme traditionnel	34
2.1.1	Description de l'organisation des données	34
2.1.2	Gestion de l'espace mémoire	35
2.1.3	Efficacité de l'algorithme de construction de la grille	35
2.2	Construction de la grille de rayons	37
2.2.1	Identification de la position des rayons dans la grille en perspective	38
2.2.2	Regroupement des rayons par indice de cellule	38
2.2.3	Décompte du nombre de rayons par cellule non vide	39
2.2.4	Génération du nombre de rayons par cellule	39
2.2.5	Génération de la version finale de <i>débutCelluleIds</i>	39
2.3	Triangle-tracing	40
2.3.1	Préparation du stockage des informations associées à chaque ligne	40
2.3.2	Stockage des informations sur les lignes associées aux triangles	42
2.3.3	Préparation du stockage des indices de cellule	42
2.3.4	Stockage des couples triangle/cellule	42
2.3.5	Préparation du stockage des résultats des tests d'intersections	42
2.3.6	Génération des couples d'indices rayon/triangle à tester	43
2.3.7	Réalisation des tests d'intersection rayon/triangle	43
2.3.8	Génération des résultats finals	43
2.3.9	Tri des intersections	43
2.4	Résultats	46
2.4.1	Temps de construction des deux grilles	47
2.4.2	Temps de calcul global	49
2.4.3	Comparaison avec l'algorithme traditionnel	51
2.4.4	Variations de la résolution de la grille en perspective	53
2.5	Sources d'optimisation supplémentaires	54
2.5.1	Construction de la grille de rayons	55
2.5.2	Rastérisation des triangles	55
2.5.3	Meilleure utilisation de la mémoire	57

Nous avons présenté dans la partie précédente les diverses structures d'accélération classiquement utilisées dans le domaine du ray-tracing. Il est ressorti de cette présentation que la structure d'accélération la plus adaptée à notre problème était la grille en perspective. Néanmoins, la particularité de notre lancer de rayons (recherche exhaustive des intersections, et connaissance *a priori* des rayons) n'est pas sans impact sur l'utilisation de la grille en perspective.

Ainsi, notre contexte d'utilisation implique certaines contraintes d'espace mémoire, mais rend également possible certaines optimisations irréalisables dans le domaine traditionnel

du lancer de rayons. Une utilisation plus efficace de la grille en perspective est possible, en inversant l'algorithme classique : plutôt que de construire une grille indexant les triangles de la scène et de parcourir ensuite le groupe de rayons étudié, nous construisons la grille indexant les rayons de la scène et parcourons l'ensemble des triangles afin de chercher les intersections potentielles.

Dans la première partie de ce chapitre (section 2.1), nous expliquons plus en détail les motivations de cette inversion de l'algorithme traditionnel. Nous présentons ensuite notre algorithme parallèle de construction de la grille indexant les rayons de la scène (section 2.2), puis celui de parcours de la grille (section 2.3). Les résultats obtenus sont ensuite discutés (section 2.4). Enfin, nous présentons diverses optimisations pour des implémentations futures (section 2.5).

2.1 Modifications de l'algorithme traditionnel

Classiquement, l'utilisation d'une grille en perspective comme structure d'accélération dans un algorithme de lancer de rayons se fait en trois étapes :

1. Construction de la grille : pour chaque cellule de la grille, on construit la liste des triangles intersectant la cellule ;
2. Pour chaque rayon, on cherche la cellule correspondante (cette cellule est en effet unique dans le cas d'un rayon partant du centre de projection de la scène). On obtient ainsi pour chaque rayon une liste de triangles potentiellement intersectés ;
3. Pour chaque rayon, on effectue les tests d'intersection entre le rayon et ces triangles potentiels.

Nous allons voir dans cette partie que cet algorithme peut être amélioré dans notre contexte d'utilisation.

2.1.1 Description de l'organisation des données

Avant toute chose, présentons la structure choisie pour le stockage de cette grille. D'un pur point de vue logique, la structure que nous cherchons à créer est une liste de listes : pour chaque cellule de la grille, nous cherchons à construire la liste des triangles qu'elle intersecte. Afin d'obtenir une représentation en mémoire compacte et adaptée au GPU, nous utilisons la même organisation des données que [LD08] : les indices des triangles intersectant des cellules de la grille sont stockés de manière contiguë, dans un unique tableau, noté *trgIds*. Ces indices de triangles sont triés par indice de cellule croissant. Un deuxième tableau, noté *débutCelluleIds*, va servir d'index au premier tableau, de manière à savoir où commence dans le tableau *trgIds* la liste des indices de triangles associés à une certaine cellule (voir Figure 2.1 pour un exemple de ces deux vecteurs sur une scène simple).

Prenons sur cette figure l'exemple de la cellule d'indice 1 : cette cellule intersecte deux triangles, nommés *C* et *B*. Ces deux indices de triangles sont stockés de manière contiguë dans le tableau *trgIds* et commencent à être énumérés à la position *débutCelluleIds[1]* dans ce tableau. Ainsi, les deux indices de triangles *C* et *B*, associés à la cellule 1, sont à trouver dans le tableau *trgIds*, de la position *débutCelluleIds[1] = 3* à la position *débutCelluleIds[1 + 1] = 5* (non incluse). Naturellement, ces indices *C* et *B* apparaissent à d'autres positions dans le tableau *trgIds*, puisqu'ils sont également associés à d'autres cellules (cellules 0 et 3 sur la figure 2.1).

Au final, la création de ces deux tableaux suffit à décrire la grille en perspective.

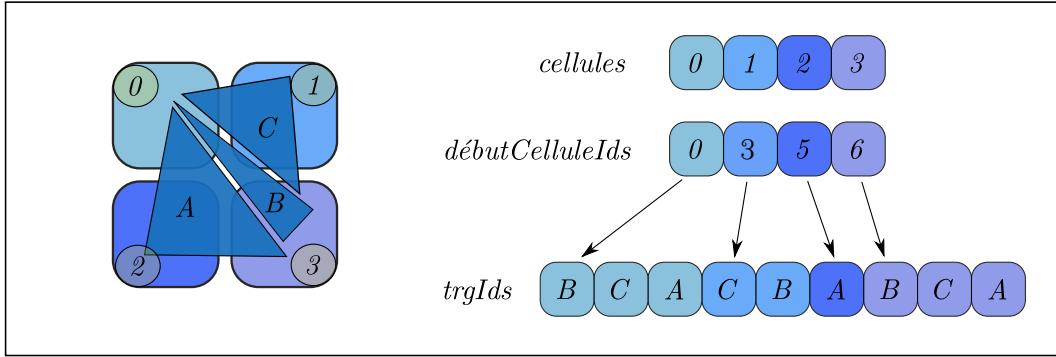


Figure 2.1 – Exemple d'une scène simple, composée de trois triangles, vue en perspective. Les deux vecteurs décrivant la grille en perspective sont *débutCelluleIds* et *trgIds*. *débutCelluleIds*[*i*] indique à quelle position dans le vecteur *trgIds* commence la liste de triangles touchant la cellule d'indice *i*.

2.1.2 Gestion de l'espace mémoire

Deux points très importants doivent être abordés au sujet de l'utilisation de cette grille en perspective dans notre contexte. Tout d'abord, la construction de cette grille requiert une occupation de l'espace mémoire conséquente, ce qui peut vite devenir un problème dans le contexte de la programmation sur GPU. En effet, les cartes du commerce ne disposent pas, dans de nombreux cas, d'un espace mémoire supérieur à 1 Go. Dans le cas de scènes complexes, cette limitation peut vite devenir préoccupante : pour une grille assez fine ($1000 * 1000$ cellules), le nombre de couples cellule-triangle peut devenir très important (plusieurs dizaines de millions de couples), menant facilement à des tailles de grille de l'ordre de plusieurs centaines de Mo. L'espace mémoire disponible sur une carte graphique est donc normalement suffisant pour contenir la grille. Mais une importante quantité d'espace est également nécessaire afin de conserver les coordonnées des triangles (2D et 3D) et des rayons de la scène. D'autres informations, liées à la topologie du maillage (voir chapitre 3) par exemple, peuvent aussi demander un espace mémoire important. De plus, dans notre contexte de travail, où toutes les intersections doivent être stockées (chaque intersection étant en outre associée à plusieurs informations), le problème de l'espace mémoire disponible sur GPU est particulièrement important, et se présente plus vite que pour les applications traditionnelles du raytracing. Le problème est néanmoins cité dans certains travaux de la communauté [KS09] (sans qu'une solution concrète soit détaillée), mais ne se présente que pour des scènes très volumineuses.

De manière plus concrète, sur une scène de travail que nous avons construite (nommée *nuclearScene*), assez volumineuse (738 000 triangles), nous avons dû faire face, pour certaines configurations, à la création de plusieurs dizaines de millions de couples cellule-triangle, pour près d'une dizaine de millions de tests d'intersection et d'intersections effectives. L'exécution de l'algorithme nécessitait alors un espace mémoire légèrement supérieur au Go, et n'était donc pas possible sur notre précédent modèle de carte (NVIDIA GTX 295).

2.1.3 Efficacité de l'algorithme de construction de la grille

Outre ces problèmes d'occupation mémoire, notre contexte d'utilisation nous incite à aborder un autre point, cette fois avantageux. En effet, comme nous le verrons, et comme cela fut déjà précisé dans [KS09], la majeure partie du coût de construction de la grille réside

dans le tri des couples triangle-cellule trouvés : après avoir identifié, pour chaque triangle, les indices des cellules rencontrées, il faut trier les couples triangle-cellule ainsi constitués par indice de cellule croissant. Or, ce nombre de cellules est potentiellement énorme, puisqu'un triangle peut intersecter un grand nombre de cellules, parfois même l'intégralité des cellules de la grille.

Cette nouvelle difficulté nous amène à proposer la solution suivante : plutôt que de construire une grille en perspective indexant les triangles de la scène, nous proposons de construire une grille indexant les rayons de la scène. Par rapport à l'algorithme décrit au début de la partie 2.1, nous inversons donc simplement les étapes 1 et 2 de l'algorithme. Il est à noter que la construction de la grille indexant les rayons n'est possible que parce que les rayons sont connus à l'avance (et même imposés par le code de calcul NARMER), ce qui n'est (en général) pas le cas pour les applications traditionnelles de raytracing. Cette inversion présente deux avantages majeurs.

En premier lieu, un gain de temps important va ainsi pouvoir être réalisé. En effet, dans notre cas d'utilisation, tous les rayons sont issus du même point, qui est le centre de projection de la scène. Par conséquent, la projection d'un rayon dans la grille en perspective se limite à un point. Le nombre de couples rayon-cellule à trier pour construire la grille est donc exactement égal au nombre de rayons intersectant le cône de vision associé à la grille. Au pire, il est égal au nombre de rayons contenus dans le paquet étudié. Or, lorsque les scènes gagnent un peu en complexité géométrique, ce nombre devient très vite largement inférieur au nombre de couples triangle-cellule que l'on doit trier dans le cas traditionnel. L'étape du tri est donc beaucoup plus rapide, et le temps de construction de la grille largement inférieur. Ce gain ne se fait néanmoins pas sans concession : en inversant ainsi l'algorithme, on obtient à la fin des calculs une liste d'intersections triées par indice de triangle et non plus par indice de rayon. Nous aurons donc à trier les intersections trouvées par indice de rayon croissant à la fin des calculs. Néanmoins, pour nos scènes, le nombre d'intersections trouvées est toujours bien inférieur au nombre de couples générés, et le gain de performance reste donc assez important.

L'autre avantage de cette technique est qu'elle résout en partie les problèmes d'espace mémoire précédemment exposés. En effet, il est désormais possible de traiter les triangles par "paquets". Après avoir construit la grille indexant les rayons, nous pouvons calculer les intersections avec un premier groupe de triangles, sauvegarder les intersections trouvées, libérer le reste de l'espace mémoire utilisé pour ce groupe de triangles, puis traiter les paquets suivants. Certes, pour de très grosses scènes, le problème de l'espace mémoire existe encore (en particulier pour le stockage d'un très grand nombre d'intersections), mais pour une grande majorité de scènes, déjà assez volumineuses, le problème décrit par [KS09] ne se présente plus.

Avant de décrire plus précisément l'implémentation de notre algorithme sur GPU, il est tout de même important de noter que cette inversion est en réalité une manière de reproduire le comportement des algorithmes de rastérisation (ce qui est logique, car dans ce domaine, les rayons sont également connus à l'avance, et issus du centre de la caméra). En effet :

1. La construction de la grille régulière revient à un pavage régulier de l'écran ;
2. Le parcours des triangles, la détermination des cellules intersectées, puis les tests d'intersection rayon-triangle peuvent être vus comme une phase de rastérisation des triangles (on essaiera d'ailleurs plus loin de s'inspirer des techniques de rastérisation pour effectuer certaines de ces étapes) ;
3. Enfin, lorsque les tests de profondeur sont désactivés, l'intégralité des intersections le long de chaque rayon (dans le cas graphique, équivalent à un pixel de l'écran) est générée.

Fondamentalement, cet algorithme se rapproche encore davantage de ceux proposés pour mettre en place une rastérisation irrégulière, évoquée dans la partie précédente. Il est également intéressant de noter que les premières tentatives de portage d'algorithmes de raytracing sur GPU reposaient sur la même idée [CHH02] : là aussi, l'algorithme était inversé, et on traitait les triangles en second lieu.

Néanmoins, les algorithmes que nous allons présenter dans la suite de ce chapitre nous permettent de conserver en mémoire l'intégralité des intersections trouvées, ce qui n'est pas possible avec les méthodes existantes. De plus, comme on l'a dit, il n'est pas possible d'utiliser un rastériseur physique pour des rayons à positions irrégulières dans l'espace.

Après avoir présenté les raisons de cette modification de l'algorithme traditionnel, nous pouvons maintenant détailler les méthodes choisies pour construire et exploiter la grille de rayons.

2.2 Construction de la grille de rayons

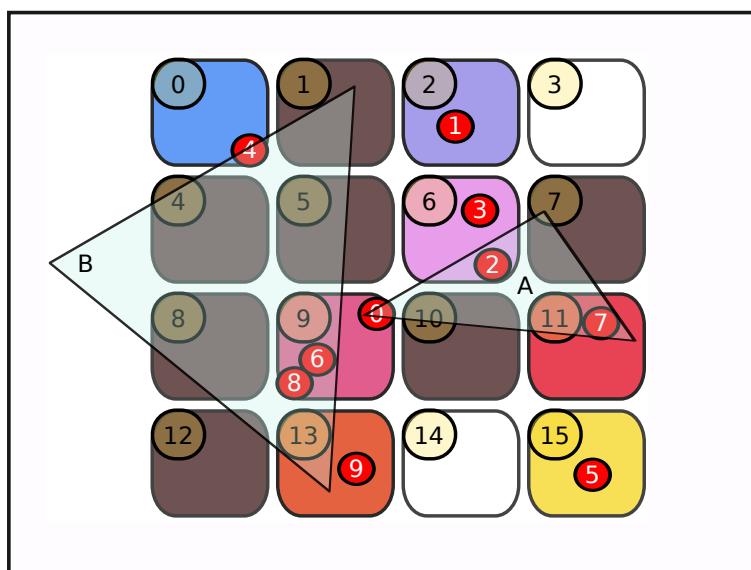


Figure 2.2 – Exemple de grille en perspective de résolution 4*4 sur une scène simple, composée de 2 triangles A et B, et de 10 rayons. Les indices de cellules sont représentés dans le coin supérieur gauche de chaque cellule. Les rayons, numérotés de 0 à 9, sont représentés par de petits cercles rouges. Les cellules colorées de la grille sont celles au sein desquelles une intersection rayon-triangle se produit. Leur couleur permet de les repérer dans les figures 2.3 et 2.4.

L'algorithme que nous présentons ici est très similaire à celui rencontré dans [KS09]. Il a cependant été conçu parallèlement à ces travaux. Un exemple concret d'une grille 2D sur une scène très simplifiée est présenté sur la figure 2.2. La figure 2.3 présente les différentes étapes de construction de la grille en perspective associée à ce cas-test. Les deux vecteurs décrivant la grille sont *rayIds* et *débutCelluleIds*, en suivant les mêmes conventions que pour la grille indexant les triangles de la scène (voir 2.1). Les différentes étapes de l'algorithme sont restituées sur la figure 2.3 ; elles sont numérotées comme les différentes sous-parties de cette partie.

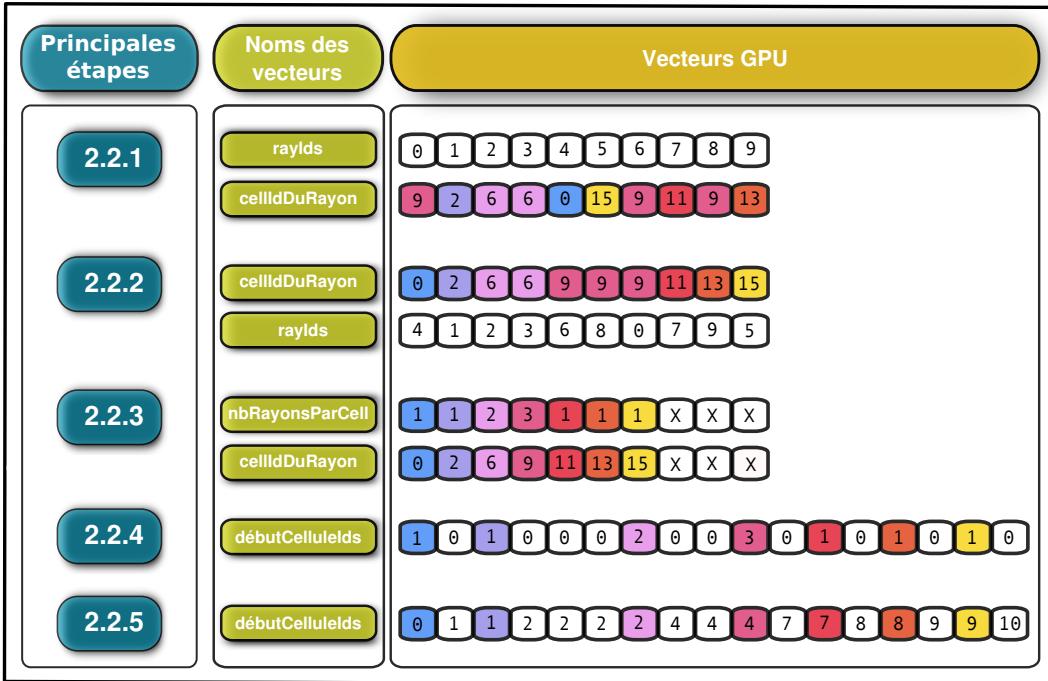


Figure 2.3 – Description de l’algorithme de construction de la grille en perspective indexant les rayons. Les valeurs associées aux vecteurs correspondent au cas présenté sur la Figure 2.2 (idem pour les couleurs de cellules utilisées). Les étapes de l’algorithme sont numérotées de la même manière que les paragraphes décrivant cet algorithme.

2.2.1 Identification de la position des rayons dans la grille en perspective

La première étape de l’algorithme est évidente : elle consiste simplement à trouver dans quelle cellule de la grille chaque rayon tombe (on considère ici que tous les rayons “tombent” dans la grille, c’est-à-dire appartiennent au faisceau de rayons défini par la grille en perspective, car cela ne modifie pas profondément le problème. Naturellement, nous traitons cette difficulté dans nos implémentations). $nbRayons$ threads sont donc lancés, chacun travaillant sur un rayon et ayant pour fonction de déterminer dans quelle cellule ce dernier tombe.

2.2.2 Regroupement des rayons par indice de cellule

La seconde étape de l’algorithme consiste à trier les couples rayon-cellule (formés à partir des deux vecteurs $rayIds$ et $cellIdDuRayon$) par indice de cellule croissant. Après cette étape, les rayons appartenant à une même cellule occuperont donc des espaces contigus en mémoire. Il est à noter que le tri peut ici être optimisé : en effet, pour une taille de grille de $1024 * 1024$, 20 bits suffisent pour décrire les indices des cellules. Du temps peut donc être économisé en utilisant un algorithme de tri capable de ne prendre en compte que les n premiers bits (au lieu des 32 normalement utilisés pour un entier) pour le vecteur de “clés”. Nous avons utilisé le tri issu de l’implémentation de Thrust [HB09] : cette dernière exploite les plus récents résultats de l’état de l’art (par exemple, [SHG09] et [MG10]) et présente entre autres cette optimisation consistant à exploiter uniquement le nombre de bits nécessaire.

Comme nous l’avons déjà précisé, cette étape constitue le goulot d’étranglement du processus de construction de la grille. Il est également intéressant de noter qu’à l’issue de cette

étape, le tableau $rayIds$ est déjà tel que nous souhaitons le produire à la fin de l'algorithme. Les étapes suivantes vont donc avoir pour but de construire correctement le tableau $débutCelluleIds$ associé.

2.2.3 Décompte du nombre de rayons par cellule non vide

L'objectif final de cet algorithme est de construire le tableau $débutCelluleIds$. Pour cela, nous allons initialiser toutes les valeurs de ce tableau à 0. Pour remplir correctement ce tableau, il nous faudra ensuite connaître les indices des cellules non vides (contenant au moins un rayon), puis, pour chacune de ces cellules, le nombre de rayons qu'elle contient.

Sur un processeur classique, il serait très simple de générer ces informations (le tableau $cellIdDuRayon$ étant déjà trié par indice de cellule), via une simple boucle sur le tableau $cellIdDuRayon$ généré à l'étape 2.2.2. Il n'est pas possible de réaliser une telle boucle sur GPU, car elle produirait de nombreux conflits d'écriture et se révèlerait donc tout à fait inadaptée à l'architecture du GPU. Nous passons donc ici par une "réduction segmentée", avec les indices de cellules pour clés, et un tableau rempli de 1 comme tableau de valeurs à traiter. Cette opération générique a également été implémentée de manière très efficace dans Thrust [HB09] (voir `Thrust::reduce_by_key`).

Plus concrètement, cette opération consiste en un compactage du tableau $cellIdDuRayon$ réalisé en remplaçant chaque séquence de valeurs identiques (correspondant à des rayons intersectant une même cellule) par un couple de valeurs : l'indice de cette cellule et le nombre de rayons qu'elle intersecte. Après cette opération de réduction segmentée, les "doublons" au sein du tableau $cellIdDuRayon$ sont donc effacés. Pour chacun des indices restants, on trouve, à la même position dans le tableau $nbRayonsParCell$, le nombre de rayons correspondant. Ainsi, observons, dans le cas de la figure 2.3, la séquence de valeurs 9, situées entre la cinquième et la septième position du tableau $cellIdDuRayon$, correspondant donc aux trois rayons intersectant la cellule d'indice 9 (revoir la figure 2.2 si besoin est). Après l'opération de "réduction segmentée" et l' "effacement" des valeurs identiques successives, on retrouve ce 9 en quatrième position dans le tableau $cellIdDuRayon$. À cette même quatrième position, dans le tableau $nbRayonsParCell$, nous retrouvons bien le chiffre 3, nombre de fois que le chiffre 9 apparaissait, après l'étape 2.2.2, dans le tableau $cellIdDuRayon$ (correspondant, comme on l'a déjà dit, au nombre de rayons dans cette cellule 9). Cette étape de l'algorithme de construction de grille nous permet donc de disposer de la liste des cellules intersectées par des rayons, et de connaître, pour chacune de ces cellules, le nombre de rayons qu'elle intersecte.

2.2.4 Génération du nombre de rayons par cellule

Grâce à l'étape précédente de l'algorithme, il nous suffit maintenant d'écrire les valeurs contenues dans le tableau $nbRayonsParCell$ dans le tableau $débutCelluleIds$, aux positions indiquées par le tableau $cellIdDuRayon$. On obtient ainsi dans le tableau $débutCelluleIds$ le nombre de rayons contenus par chaque cellule.

2.2.5 Génération de la version finale de $débutCelluleIds$

Comme le nombre de rayons pour chaque cellule est maintenant connu, il nous suffit maintenant d'effectuer un scan exclusif (ou somme cumulée) sur le tableau $débutCelluleIds$ pour produire le tableau que nous souhaitions générer. Ainsi, pour le cas, par exemple, de la

cellule, d'indice 9, on peut vérifier, à la fin de cette étape, que le nombre de rayons qu'elle intersecte est donné par la valeur $\text{débutCelluleIds}[10] - \text{débutCelluleIds}[9] = 7 - 4 = 3$. En outre, dans le tableau $rayIds$, généré à la fin de l'étape 2.2.2, on trouve bien, entre les positions $\text{débutCelluleIds}[9] = 4$, et $\text{débutCelluleIds}[10] = 7$ (non incluse), les valeurs 6, 8 et 0, correspondant aux indices de rayons que la cellule d'indice 9 intersecte (revoir la figure 2.2).

2.3 Triangle-tracing

Après cette phase d'initialisation vient la phase que nous appelons triangle-tracing, qui consiste, pour chaque triangle, à identifier les cellules recouvertes, à en déduire les rayons intersectant potentiellement le triangle, puis à effectuer les tests d'intersection. On suppose dans cette phase que les triangles sont déjà projetés dans le repère 2D associé à la grille. La figure 2.4 décrit l'application de notre algorithme au cas-test décrit sur la figure 2.2. Comme dans la partie précédente, les différentes étapes de l'algorithme sont numérotées de la même manière que les paragraphes associés.

Les quatre premières étapes de cet algorithme vont en réalité consister en une construction partielle de la grille en perspective référençant les triangles de la scène. L'objectif est ici de construire la liste des couples triangle-cellule de la grille. Cette étape correspond donc seulement à l'étape 2.2.1 présentée précédemment, sans les suivantes (notamment la lourde étape du tri). La difficulté vient ici du fait qu'un même triangle peut intersecter plusieurs cellules, et qu'on ne connaît pas le nombre de ces cellules par avance.

2.3.1 Préparation du stockage des informations associées à chaque ligne

Nous procédons à la rastérisation des triangles (recherche de toutes les cellules qu'un triangle intersecte) via un algorithme de *scanline*. Cet algorithme consiste à balayer l'ensemble des lignes entre le “haut” et le “bas” de chaque triangle (le triangle est déjà projeté en 2D) et, pour chaque ligne, à trouver les indices de la première et de la dernière cellule que le triangle intersecte (voir Figure 2.5). Le nombre de cellules chevauchées peut varier énormément en fonction du triangle, et, dans un souci de répartition de la charge des calculs, il n'apparaît donc pas judicieux de confier à chaque thread le traitement intégral d'un triangle. Afin de mieux répartir les calculs (sans toutefois complètement empêcher les problèmes de répartition de la charge), nous avons choisi de diviser le traitement de chaque triangle en deux étapes. Tout d'abord, chaque thread travaille sur un triangle différent et identifie les lignes de la grille qu'il occupe. Ensuite, chaque thread travaille sur une ligne au sein d'un triangle et désigne les cellules intersectées.

La première étape de cet algorithme consiste donc à faire traiter chaque triangle par un thread, qui doit générer et stocker en mémoire la liste des lignes que le triangle intersecte. Bien évidemment, le nombre de lignes occupées par chaque triangle varie entre les différents triangles, et il n'est donc pas possible de savoir à l'avance à quel emplacement dans le vecteur de sortie un thread va devoir commencer à écrire les indices des lignes qu'il a calculés. Nous nous retrouvons ici face au problème soulevé dans le paragraphe 1.2.2.3, à savoir que chaque thread génère une quantité de données différentes et qu'il faut donc déterminer la quantité de données générées par les threads précédents avant de pouvoir commencer à écrire en mémoire. Afin de résoudre notre problème, nous allons donc utiliser la méthode décrite dans ce paragraphe 1.2.2.3 (notons au passage que dans sa proposition de rendu suivant l'architecture

Principales étapes	Noms des vecteurs	Vecteurs GPU
2.3.1	nbLignes nbLignes	2 4 0 0 2 6
2.3.2	débutFin	6;7;A 9;11;A 0;1;B 4;5;B 8;9;B 12;13;B
2.3.3	nbCellsParLigne nbCellsParLigne	2 3 2 2 2 2 X 0 2 5 7 9 11 13
2.3.4	celluleIds triangleIds	6 7 9 10 11 0 1 4 5 8 9 12 13 A A A A A B B B B B B B
2.3.5	nbTests nbTests	2 0 3 0 1 1 0 0 0 0 3 0 1 X 0 2 2 5 5 6 7 7 7 7 10 10 11
2.3.6	rayIdsATester trgIdsATester	2 3 6 8 0 7 4 6 8 0 9 A A A A A A B B B B B B
2.3.7	résultats	1 0 0 0 1 1 1 1 1 0 0
2.3.8	sommeRésultats rayIdsFinal trgIdsFinal	0 1 1 1 1 2 3 4 5 6 6 2 0 7 4 6 8 A A A B B B

Figure 2.4 – Description de l’algorithme de lancer de triangles. Les valeurs associées aux vecteurs correspondent également au cas présenté sur la Figure 2.2. Les étapes de l’algorithme sont aussi numérotées de la même manière que les paragraphes décrivant cet algorithme.

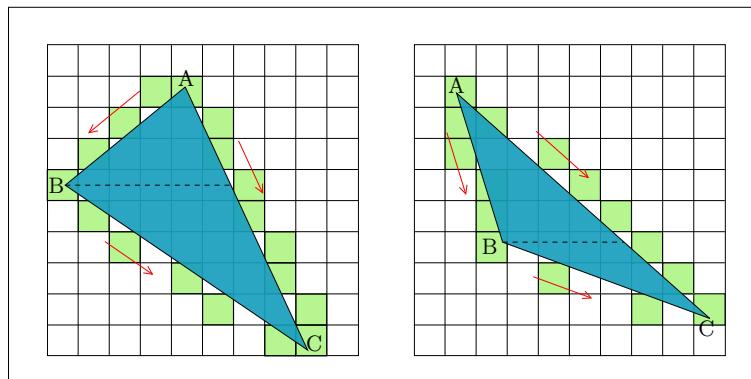


Figure 2.5 – Stratégie de rastérisation *scanline* sur deux triangles différents. L’algorithme part du sommet haut *A* chaque triangle. Pour chaque nouvelle ligne, les coordonnées des cellules de début et de fin du triangle (en vert) sont calculées.

Reyes sur GPU, Zhou [ZHR⁺09] utilise la même technique pour gérer les micropolygones au sein d'un pixel, dont il ne connaît pas le nombre au départ).

En premier lieu, nous commençons donc simplement par lancer un thread par triangle, qui se contente de compter le nombre de lignes occupées par le triangle. Un scan exclusif sur le vecteur produit nous permet ensuite d'identifier, pour chaque triangle, le nombre de lignes occupées par les triangles le précédent. Cette information nous permettra de savoir où chaque thread devra écrire en mémoire les informations qu'il calculera.

2.3.2 Stockage des informations sur les lignes associées aux triangles

Grâce à ce travail, nous pouvons maintenant stocker les informations sur les lignes associées à chaque triangle. Pour chaque ligne, cette information se résume à trois entiers : l'indice de la première cellule rencontrée par le triangle sur cette ligne, l'indice de la dernière cellule, et l'indice du triangle en question. Nous pourrions donc stocker cette information sous forme d'une petite structure de trois entiers, et créer un vecteur de structures *débutFin*, où $débutFin[i]$ serait la réunion des trois informations associées à la ligne d'indice i . Cependant, une telle organisation de la mémoire génère des accès mémoire peu cohérents, puisque des threads voisins n'accèdent alors pas à des données voisines. Dans une telle situation, on dit qu'il faut préférer les structures de tableaux (*structure of arrays*) aux tableaux de structures (*array of structures*) lorsqu'on programme sur GPU. Afin d'obéir à ce principe, nous créons donc trois vecteurs différents en mémoire : les trois entiers associés à une ligne i seront situés à la $i^{\text{ème}}$ position dans chacun de ces vecteurs. Dans un souci de clarté, nous utilisons néanmoins le vecteur *débutFin* dans la figure 2.4.

Le kernel que nous lançons ici consiste donc à assigner chaque thread à un triangle et à écrire en mémoire les informations associées à chacune des lignes de la grille que le triangle occupe. Chaque thread i devra commencer à écrire en mémoire à l'emplacement $nbLignes[i]$, calculé au cours de la phase précédente de l'algorithme.

2.3.3 Préparation du stockage des indices de cellule

On effectue ici le même travail que lors des étapes précédentes : pour chaque ligne d'un triangle, il s'agit de stocker en mémoire les cellules associées, le nombre de cellules variant en fonction de la ligne. Nous calculons donc ici le nombre de cellules pour chaque ligne, puis, via une somme cumulée, nous générerons les emplacements en mémoire où chaque thread devra commencer à écrire lors de l'étape suivante de l'algorithme.

2.3.4 Stockage des couples triangle/cellule

Grâce à cette étape, chaque thread peut maintenant travailler sur une ligne et stocker en mémoire les couples triangle/cellule associés à la ligne en question.

2.3.5 Préparation du stockage des résultats des tests d'intersections

Nous devons maintenant procéder aux tests d'intersections. Pour chaque couple triangle/cellule généré, il nous faut identifier le nombre de rayons que la cellule contient. Ce nombre de rayons nous donne le nombre de tests d'intersections à effectuer pour le couple triangle/cellule considéré. Pour la première fois, nous utilisons ici la grille de rayons que nous

avons générée à la partie 2.2. En effet, le nombre de rayons qu'une cellule i contient est donné par la simple formule $\text{débutCelluleIds}[i + 1] - \text{débutCelluleIds}[i]$.

Nous lançons donc un premier kernel, dans lequel chaque thread traite un couple triangle/cellule et écrit en mémoire le nombre de rayons dans la cellule du couple. Ensuite, une somme cumulée est exécutée sur le vecteur en sortie afin de savoir à quel emplacement en mémoire devront être écrits les indices des couples rayon/triangle à tester.

2.3.6 Génération des couples d'indices rayon/triangle à tester

Grâce à l'étape précédente, on peut désormais demander à chaque thread de traiter un couple triangle/cellule et d'écrire en mémoire les couples rayon/triangle à tester. Cette stratégie est plus efficace que de faire directement effectuer les tests d'intersections par chaque thread pour chaque couple triangle/cellule, car le nombre de rayons par cellule est très variable. Une telle stratégie mènerait donc facilement à de lourds problèmes de répartition de charge.

2.3.7 Réalisation des tests d'intersection rayon/triangle

Nous pouvons enfin procéder aux tests d'intersection rayon/triangle, chaque thread effectuant un test. Si le thread trouve une intersection, il écrit 1 à l'emplacement correspondant dans le vecteur *résultats*, 0 sinon. Chaque thread génère au passage la profondeur d'intersection, à savoir la valeur t , telle que, si M est le point d'intersection et \vec{OP} le rayon, alors $\vec{OM} = t\vec{OP}$. La nature des algorithmes utilisés pour ces tests d'intersections rayon-triangle sera détaillée dans le chapitre suivant (voir 3.1).

2.3.8 Génération des résultats finals

Il nous faut maintenant réduire les vecteurs *rayIdsATester* et *trgIdsATester* (mais également le vecteur de profondeurs d'intersections *Profondeurs*) aux couples rayon/triangle pour lesquels une intersection a été trouvée. Pour réaliser cela, nous commençons simplement par compter le nombre de 1 dans le vecteur *résultats*. Nous obtenons par la même occasion le nombre total d'intersections trouvées. Nous pouvons alors allouer de l'espace mémoire pour trois vecteurs *rayIdsFinal*, *trgIdsFinal* et *profondeursFinal*, correspondant respectivement aux indices de rayons, aux indices de triangles et aux profondeurs d'intersection pour chaque intersection trouvée. Afin de remplir ces trois vecteurs, il est ensuite nécessaire d'effectuer un scan exclusif sur le vecteur *résultats* (donnant le vecteur *sommeRésultats*), dans le but de connaître, pour chaque intersection dans le vecteur *résultats*, la place qui lui est associée dans les trois vecteurs de sortie. Enfin, nous pouvons alors remplir les trois vecteurs d'intersections, via une opération d'éparpillement (*scattering*), puisque nous voulons écrire les données situées en position i dans les trois vecteurs de départ aux positions *sommeRésultats*[i] dans les vecteurs de sortie. Cependant, nous ne voulons effectuer cette opération que pour les positions i vérifiant *result*[i] = 1. Pour réaliser cette opération, nous avons donc fait appel à l'opération *scatter_if* de la librairie Thrust.

2.3.9 Tri des intersections

Nous disposons maintenant de l'ensemble des intersections au sein de la scène, et des informations qui leur sont associées (indices de rayons, de triangles et profondeurs d'inter-

sections). La dernière étape de l'algorithme (avant de transférer ces données vers le CPU) consiste à trier ces informations par indice de rayon croissant, puis, à rayon égal, par profondeur croissante. Pour cela, plusieurs possibilités s'offrent à nous. La première consiste à trier les vecteurs *rayIdsFinal* et *profondeursFinal* en voyant ces deux vecteurs comme un seul vecteur de couples (*rayId*, *t*). Il faut alors définir une simple relation d'ordre sur ces couples pour pouvoir les trier. L'instauration de cette relation d'ordre n'est néanmoins pas sans effet, puisqu'elle rend impossible l'utilisation d'un tri par base (*radix sort*), utilisé en général sur GPU car très facile à paralléliser (le tri par base repose sur plusieurs tris successifs, chacun reposant sur seulement d bits de chaque donnée d'origine. Pour $d = 3$, par exemple, on a seulement $2^3 = 8$ valeurs possibles, et on peut donc se contenter d'identifier à laquelle de ces 8 valeurs est associée chaque donnée pour définir sa position de sortie). Le meilleur candidat sur GPU devient alors le tri fusion (*merge sort*), qui est néanmoins nettement moins efficace.

La deuxième solution consiste à trier les données d'intersections par indice de rayon, puis à faire travailler un thread sur chaque rayon, qui doit trier les intersections associées à ce rayon par profondeur. Cette stratégie n'est pas non plus optimale, car elle peut souffrir de problèmes de répartition de charge, le nombre d'intersections par rayon pouvant grandement varier. La troisième solution serait de d'abord trier les données par profondeur d'intersection, puis d'effectuer un second tri (stable, celui-ci) par indice de rayon. Hélas, le tri stable est plus lent en temps de calcul que le premier. De plus, le fait de réaliser un premier tri des données par profondeur d'intersection provoque un éparpillement des données associées à un même rayon, et rend le second tri d'autant plus long.

Aucune de ces solutions ne nous paraissant vraiment satisfaisante, nous avons donc élaboré une dernière technique permettant d'optimiser ce tri. Comme nous allons le voir, cette dernière solution n'a pas fonctionné comme nous l'attendions, mais nous pensons néanmoins qu'il est intéressant de la détailler, de manière, entre autres, à introduire les problèmes d'approximations générés par l'arithmétique flottante, que nous contournerons, dans le cadre des tests d'intersection rayon-triangle, dans le chapitre 3.

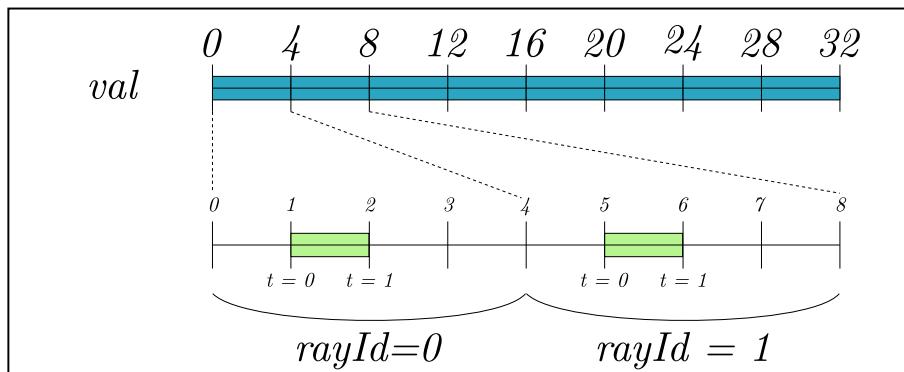


Figure 2.6 – Méthode de stockage utilisée afin de représenter simultanément, pour chaque intersection, le rayon intersectant *rayId* et la profondeur d'intersection *t*. Sur la ligne du haut, nous voyons l'ensemble des intersections, représentées par la valeur unique *val*. Un agrandissement est effectué pour les valeurs de *val* situées entre 0 et 8 sur la ligne du bas. Ces valeurs sont associées aux rayons d'indices *rayId* = 0 et *rayId* = 1. Les zones correspondant aux variations de *t* sont représentées en vert.

L'élaboration de cette technique part d'une volonté de ne garder qu'un seul vecteur pour décrire à la fois les indices de rayons et les profondeurs *t* pour chaque intersection. En effet, nous avons vu que les profondeurs d'intersection *t* étaient définies par la relation $\overrightarrow{OM} = t \cdot \overrightarrow{OP}$, où \overrightarrow{OP} désigne le rayon et *M* le point d'intersection. Cela signifie qu'une profondeur

d'intersection t sera toujours comprise entre 0 et 1. Un couple $(rayId, t)$ peut donc être identifié par l'unique valeur (non entière) $rayId + t$. L'indice du rayon concerné pourra être retrouvé en calculant la partie entière de cette valeur, et la profondeur t sera alors simplement donnée par la formule $(rayId + t) - t$. De manière à ne pas faire d'erreur pour des intersections se produisant à proximité de $t = 0$ ou $t = 1$ (des ambiguïtés peuvent alors exister quant à l'identité du rayon), il nous faut néanmoins nous assurer que les valeurs associées à des rayons différents sont bien disjointes. Pour régler ce problème, notre première idée a été d'utiliser la valeur $rayId + 0.5 * t + 0.25$ comme représentation d'un couple $(rayId, t)$. Toutes les intersections associées à un rayon $rayId$ sont alors représentées par des valeurs comprises entre $rayId + 0.25$ et $rayId + 0.75$, ce qui garantit le bon traitement des cas $t = 0$ ou $t = 1$. Cependant, afin de minimiser les erreurs d'arrondis, nous avons choisi d'utiliser la valeur suivante (voir la figure 2.6 pour une représentation plus intuitive):

$$val = 4 * rayId + t + 1$$

L'opération $4 * rayId + 1$ porte uniquement sur des entiers et ne souffre donc pas de problèmes d'approximation pour des valeurs raisonnables. Il suffit alors d'y ajouter le flottant t afin d'obtenir la valeur souhaitée. L'indice de rayon peut alors être retrouvé via :

$$rayId = \lfloor 4 * rayId + t + 1 \rfloor / 4$$

en utilisant une division entière, car $4 * rayId + t + 1$ est strictement compris entre $4 * rayId$ et $4 * (rayId + 1)$ (le fait d'avoir choisi la valeur 4 nous permet même de réaliser cette division entière via un simple décalage de bits). Une fois $rayId$ identifié, la valeur t peut quant à elle être retrouvée via la formule triviale

$$t = (4 * rayId + t + 1) - (4 * rayId + 1)$$

À titre d'exemple, considérons une intersection entre le rayon d'indice 5, pour la profondeur $t = 0.8$. Cette intersection sera alors représentée par la valeur $4 * 5 + 1 + 0.8 = 21.8$. Nous retrouvons bien alors :

$$rayId = \lfloor 21.8 \rfloor / 4 = 21 / 4 = 5$$

car la division est entière, puis

$$t = 21.8 - (20 + 1) = 0.8$$

Cette astuce devait donc nous permettre de trier les intersections en ne réalisant qu'un seul tri (de plus, le tri est effectué sur des données numériques disposant d'une relation d'ordre évidente, rendant possible un tri par base). En outre, elle pouvait nous permettre de minimiser la quantité de données à transférer du GPU vers le CPU (nous aurions alors pu attendre la phase de calcul sur CPU avant de “décrypter” ces données, et de retrouver les indices de rayons et profondeurs d'intersections).

Néanmoins, après implémentation, nous avons observé que l'utilisation de cette valeur unique afin de représenter le couple de données $(rayId, t)$ générait, après tri, un vecteur de distances d'intersections t différent de celui attendu. Après analyse, il s'est avéré que la simple addition entre l'entier $4 * rayId + 1$ et le flottant t générait une troncature très importante de l'information concernant t . Ainsi, pour des valeurs relativement élevées (nous avons par exemple constaté des erreurs pour des valeurs de $rayId$ proches de 75000), le terme $4 * rayId + 1$ devient trop grand devant t . L'addition de ces deux nombres suivant les règles de l'arithmétique flottante provoque donc une perte d'information importante concernant t . L'opération $(4 * rayId + t + 1) - (4 * rayId + 1)$ ne permet donc pas de retrouver la valeur

exacte t de départ (les décalages sont très importants, nous avons ainsi observé, pour des valeurs de $rayId$ proches de 75000, un passage de la valeur $t = 0.53$ à la valeur $t = 0.41$). Nous avons alors pensé n'utiliser la valeur val que pour le tri des intersections, et non pour le “décryptage” des valeurs $rayId$ et t . Hélas, la troncature des informations sur t fait que certaines intersections, associées à des valeurs de t normalement différentes, sont représentées par des valeurs val égales. Ainsi, le tri effectué sur les données val ne donne pas le résultat attendu.

Notre dernière tentative afin de contourner ce problème a été de “normaliser” les valeurs de $rayId$, en les remplaçant par des valeurs flottantes situées entre 0 et 1. Nous espérions ainsi, en réalisant des additions entre des valeurs de même ordre de grandeur, ne plus souffrir de ces approximations. Nous avons en effet fortement réduit le nombre d’erreurs effectuées via cette méthode, mais nous n’avons pu les faire disparaître en intégralité. Nous avons donc dû revenir vers les trois méthodes de tri présentées plus haut : la solution la plus rapide s’est avérée celle consistant à trier les intersections par indice de rayon, puis à lancer un kernel permettant de trier les intersections associées à un même rayon par coordonnée de profondeur. Nous avons ainsi mis en place un kernel peu optimisé, consistant à faire travailler chaque thread sur une intersection différente. Ce thread s’occupe, pour cette intersection $(rayId_0, t_0)$, de comparer la valeur t_0 avec les autres valeurs de t pour le rayon $rayId_0$. Il en déduit ainsi la place de l’intersection étudiée dans la liste des intersections pour le rayon $rayId_0$. Une fois ce travail effectué pour chaque thread, on peut donc recopier l’intersection considérée à sa bonne place dans les vecteurs $rayIdsFinal$ et $profondeursFinal$.

Cette technique de tri offre des performances satisfaisantes, mais devient très peu adaptée lors du traitement de scène dans lesquelles les rayons peuvent intersecter un grand nombre de surfaces. Nous verrons dans le chapitre 4 (voir 4.2.4) une autre solution de tri permettant de meilleurs résultats pour ce genre de scènes.

Après cette description détaillée des algorithmes GPU de construction et de parcours de grille en perspective, nous pouvons donc désormais étudier les résultats obtenus sur des scènes variées.

2.4 Résultats

Avant de présenter les performances de nos algorithmes, précisons que nous avons travaillé sur plusieurs scènes usuelles de la communauté graphique, représentant divers niveaux de complexité : Erw6, Fairy Forest, Conference et Sully. À ces quatre scènes, nous avons ajouté une cinquième scène, nommée NuclearCase, que nous avons construite nous-même. Cette scène a l'avantage d'être volumineuse (plus de 700 000 triangles) et d'être “propre” : elle est composée de plusieurs objets distincts, chacun de ces objets étant un groupe de triangles “sans trous”. Cette propriété nous permet de disposer d'une scène “exacte”, en comparaison des scènes habituellement utilisées par la communauté graphique, souvent constituées de modèles incohérents d'un point de vue géométrique/topologique. La scène nuclearCase nous permet ainsi de calculer correctement les épaisseurs parcourues par chaque rayon dans la matière, ce qui n'est pas possible avec des modèles classiques, où l'entrée dans un objet peut ne pas être associée à une sortie du même objet. Le modèle de carte graphique utilisé pour ces expériences, comme pour l'ensemble des tests décrits dans ce manuscrit, est une carte NVIDIA GTX 470 (voir l'annexe A.2 pour une description détaillée du matériel de test).

Pour chacune de ces expériences, nous avons lancé un groupe d'un million de rayons primaires (en réalité $1024 * 1024$ rayons), partant donc d'un même point de départ, et régulièrement répartis dans l'espace, chacun de ces rayons correspondant à un pixel sur une

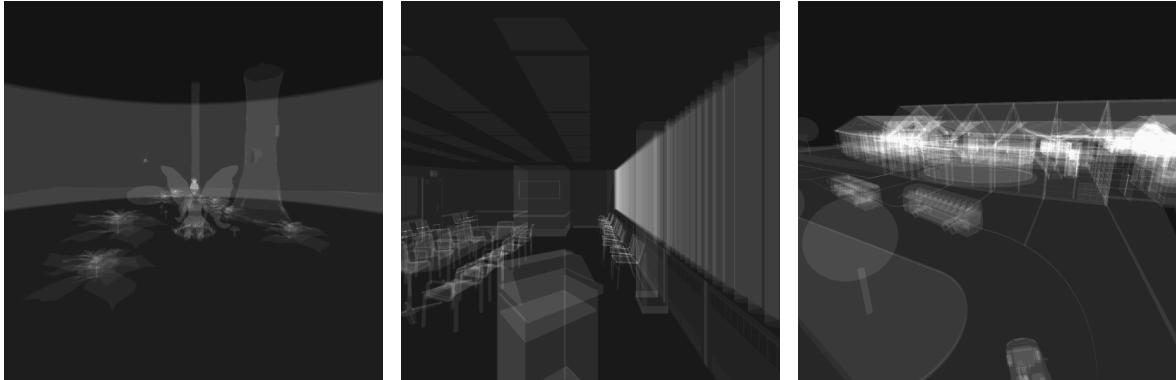


Figure 2.7 – Visualisation “radio” de certaines scènes usuelles de la communauté du lancer de rayons (Fairy Forest, Conference et Sully). 1024*1024 rayons sont lancés (chacun correspondant à un des pixels affichés). L’intensité de l’éclairage pour chaque pixel est proportionnelle au nombre d’intersections entre le rayon primaire associé au pixel et les triangles de la scène.

image vue d’une caméra centrée au point de départ des rayons (voir la figure 2.7 pour une visualisation “radio” de certaines scènes étudiées, réalisée à partir de nos algorithmes). Nous n’exploitons cependant jamais cette propriété dans nos calculs. Dans un premier temps, nous considérons des grilles de résolution fixe (1024 * 1024) : nous étudierons à la fin de cette partie l’influence de ce choix et les variations de performances observées lorsque la résolution des deux grilles varie.

2.4.1 Temps de construction des deux grilles

Temps par phase	0.4	2.4	2.5	0.2	0.4
Pourcentage temps global	6.8 %	40.7%	42.4 %	3.4 %	6.7 %

Tableau 2.1 – Comparaison des temps nécessaires à l’exécution de chaque étape de l’algorithme de construction de grille présenté en 2.2 (le tri correspond à la deuxième colonne du tableau). Chaque phase est représentée par une colonne. Les temps sont donnés en millisecondes.

La répartition géographique des rayons étant la même pour toutes ces expériences, le temps de construction de la grille de rayons a été constant : 5.9 millisecondes pour notre groupe de rayons (voir la répartition des temps en fonction des phases de l’algorithme dans le tableau 2.1). Avec les dernières améliorations de l’algorithme de tri apportées par la librairie Thrust, le tri des couples rayon/cellule ne semble plus représenter le goulot d’étranglement de l’algorithme de construction ¹. La troisième étape de l’algorithme, celle de réduction segmentée, semble désormais demander un temps équivalent à celui nécessaire au tri des couples rayon/cellule de la grille. Néanmoins, le temps global d’exécution de cet algorithme de construction de grille reste très faible, et ne constitue donc pas aujourd’hui une priorité du point de vue de l’optimisation des performances.

L’étude du temps de construction de la grille de triangles (construction partielle, puisqu’on ne procède pas au tri des couples triangle/cellule) est beaucoup plus intéressante à effectuer. En effet, comme on l’a vu dans la partie précédente, un triangle peut chevaucher plusieurs cellules de la grille, et il est donc beaucoup plus complexe de générer les couples triangle/cellule

1. voir la deuxième colonne du tableau, l’étape de tri prend 2.4 millisecondes sur les 5.9 de l’algorithme total. Lors de nos premières implémentations, sur le modèle de carte NVIDIA GTX 295, cette étape de tri demandait 17 millisecondes de calcul.

Scène	nbTrgs	nbTrgsTraités	nbCouples	Temps	T	C
Erw6	804	810	1.3M	2.2	2 716	1.7
Fairy	174K	169K	3.4M	3.2	18.9	0.9
Conference	283K	221K	4.3M	6.1	27.6	1.4
Sully	804K	406K	9.0M	10.3	25.4	1.1
NuclearCase	738K	738K	9.1M	7.6	10.3	0.8

Tableau 2.2 – Temps de génération des couples cellule/triangle pour une scène vue en perspective. *nbTrgsTraités* indique le nombre de triangles réellement traités par l’algorithme de triangle-tracing, car non éliminés par la phase de clipping. *nbCouples* indique le nombre de ces couples cellule/triangle. Le temps de génération est donné en millisecondes. *T* et *C* indiquent respectivement les temps de calcul nécessaires par million de triangles traités et par million de cellules.

que les couples rayon/cellule de la scène. Nous présentons nos performances dans le tableau 2.2. Ce tableau prouve clairement que les temps de construction de la grille ne dépendent pas du nombre de triangles de la scène. L’influence du nombre de couples triangle/cellule semble beaucoup plus importante : entre deux scènes de taille identique en termes de nombre de triangles, la scène présentant les triangles de plus grande “taille” demandera le plus long traitement. Pour vérifier cette affirmation, nous avons calculé la valeur de deux coefficients, *T* et *C*, indiquant respectivement les temps de calcul nécessaires (en millisecondes) par million de triangles traités et par million de couples triangle/cellule. Les variations de *T* observées dans le tableau sont en effet bien plus nettes que celles de *C*. Notons au passage que nous considérons les triangles “traités” et non les triangles initialement présents dans le modèle 3D de la scène. En effet, certains triangles, repérés comme n’intersectant pas l’enveloppe du groupe de rayons, sont évincés avant la phase de triangle-tracing. Cette phase d’élimination est la phase de clipping, que nous détaillerons dans les chapitres 3 (voir 3.4) et 4 (voir 4.2.2).

Il faut préciser que la notion de “taille” de triangle est ici assez particulière, puisqu’elle fait plutôt appel à une taille du triangle dans l’espace projectif : plus un triangle s’éloignera du point de départ des rayons (point de mesure du débit de dose dans notre contexte), plus il paraîtra petit. L’efficacité de la construction de la grille de triangles variera donc elle-même au sein d’une même scène en fonction de la position du centre de projection de la scène. C’est certainement là une des limitations imposées par le travail dans l’espace projectif. Dans le même temps, une scène très complexe, demandant un très long temps de construction de structure d’accélération dans le cas classique, sera beaucoup plus simple à traiter dans le cas où le point de mesure du débit de dose s’éloignera des parties “massives” de la scène.

Le fait que le coefficient *C* ne soit pas constant sur l’ensemble des scènes s’explique par le fait que le nombre de couples triangle/cellule n’est pas tout à fait le seul critère à prendre en compte pour expliquer la performance de notre algorithme. En effet, la répartition de ces couples entre les triangles de la scène a également une forte influence sur les performances observées. Pour deux scènes présentant le même nombre total de couples triangle/cellule, la scène présentant la plus grande homogénéité du point de vue de la taille de ses triangles sera avantageée. Nous devons ici faire face à un problème de répartition de la charge : deux triangles de tailles très différentes demandant des temps de traitement très différents pourront ralentir considérablement les performances s’ils sont traités par deux threads d’un même warp. Bien que nous ayons décidé de séparer le traitement des triangles par ligne (étapes 2.3.1 à 2.3.4 de notre algorithme), nous ne pouvons totalement éviter ce genre de difficultés (notons tout de même que le problème des larges triangles existe également dans le cadre d’applications plus classiques de lancer de rayons). Une solution pourrait être de découper les larges triangles en triangles plus petits [EG07], certaines solutions permettant de ne pas modifier la topologie

Scène	nbTests	nbInters	init	lancerTriangles	tri-transfert	total
Erw6	1.3M	1.2M	9.1	8.4	8.4	25.9
Fairy	3.4M	2.2M	17.2	17.4	14.7	49.3
Conference	4.3M	2.7M	18.9	24.8	16.7	60.4
Sully	9.0M	3.7M	24.8	42.7	25.8	93.3
NuclearCase	9.1M	5.7M	30.9	40.7	35.7	107.3

Tableau 2.3 – Temps d’exécution de l’algorithme de lancer de triangles pour différentes scènes. 1024 * 1024 rayons primaires sont lancés. *nbTests* et *nbInters* indiquent respectivement le nombre de tests d’intersections effectués et le nombre d’intersections finalement trouvées. *lancerTriangles* indique le temps de calcul nécessaire à l’exécution de l’algorithme de lancer de triangles décrit dans la partie 2.3 (privé de la phase de tri 2.3.9). *total* donne le temps total d’exécution de la simulation. Les temps sont donnés en millisecondes.

du maillage [DK08]. Ces solutions nécessitent néanmoins de définir des coefficients de travail dépendants de la scène, ce que nous souhaitons éviter. De plus, comme la “taille” de nos triangles est ici variable, il nous faudrait sans doute réeffectuer ces travaux de découpe au cours de la simulation. Il faudrait alors veiller à ce que ces temps de découpe ne soient pas trop longs, problème qui n’est pas évoqué dans les deux travaux que nous citons, cette découpe étant considérée comme un pré-calcul réalisé en amont. Néanmoins, nous aimerions tester ces solutions dans le futur, chose que nous n’avons pu réaliser, par manque de temps. D’autres sources d’optimisations seront présentées dans la partie 2.5.

2.4.2 Temps de calcul global

Après avoir étudié le temps nécessaire à la construction de nos deux grilles (la construction de la grille de triangles n’étant que partielle), nous souhaitons désormais étudier la performance de l’algorithme de “lancer de triangles” dans son ensemble. En effet, nous avons jusque là négligé le temps nécessaire à la réalisation des tests d’intersections et au tri des résultats. Nous ajoutons ici à la phase de tri des intersections la phase de transfert des intersections du GPU vers le CPU. Cette phase de tri-transfert des intersections reste cependant largement dominée par l’étape de tri, comme nous le verrons dans le chapitre 4 (voir 4.2.4). Les résultats obtenus sont présentés dans le tableau 2.3.

À la lecture de ce tableau, nous notons déjà le temps important pris par la phase d’initialisation par rapport aux autres phases. Cette phase d’initialisation, comprenant la création de la grille en perspective, mais aussi la phase de clipping et les étapes de projection des triangles, sera abordée en détails dans le chapitre 4 (voir 4.2). Nous verrons d’ailleurs à cette occasion que cette phase d’initialisation constitue également une étape limitante de notre algorithme global de calcul d’intersections, lorsqu’il est appliqué à de multiples paquets de rayons. En plus de cette phase d’initialisation importante, nous relevons également une phase de tri/transfert des intersections particulièrement coûteuse. Comme nous l’avons vu plus haut (voir 2.3.9), cette phase de tri est en effet assez complexe, et pénalise fortement la globalité de l’algorithme. Nous espérons pouvoir dans le futur élaborer une deuxième phase de tri (tri sur les coordonnées de profondeur) beaucoup plus efficace. Enfin, nous observons que la phase de lancer de triangles (privée de l’étape de tri/transfert des intersections) demande donc au final un temps modéré en regard des autres parties de l’algorithme. Il est néanmoins intéressant d’étudier plus précisément la répartition des performances lors de ces calculs.

La Figure 2.8 détaille ainsi la répartition des temps de calcul entre les différentes phases de l’algorithme de lancer de triangles, la phase de création de la liste de couples triangle/cellule (ou construction partielle de la grille de triangles) englobant les étapes 2.3.1 à 2.3.4, et la phase

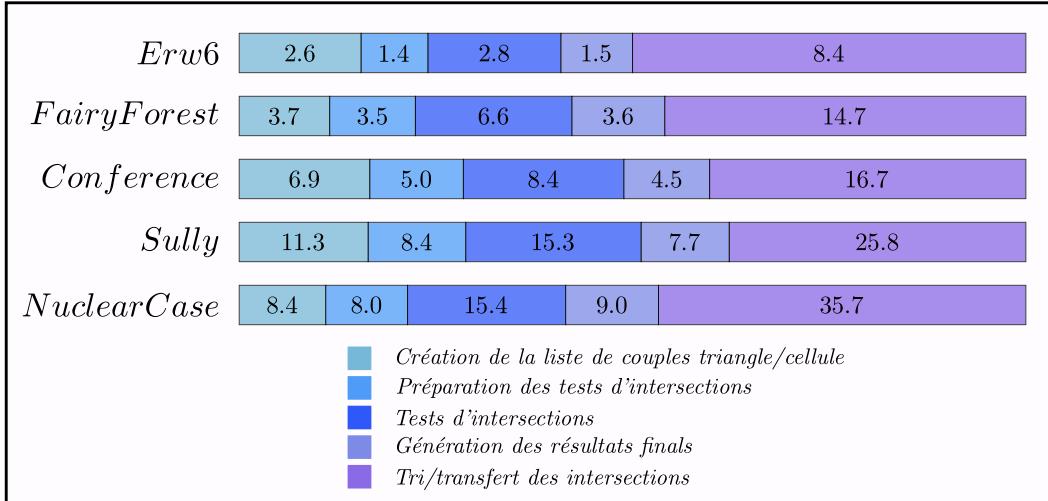


Figure 2.8 – Répartition des temps de calcul suivant les différentes phases de l'algorithme de lancer de triangles. Les temps sont donnés en millisecondes.

de préparation des tests d'intersection les étapes 2.3.5 et 2.3.6. Plusieurs observations doivent être effectuées au vu de ces résultats : tout d'abord, en dehors de la phase de tri /transfert des intersections, largement dominante, il n'existe pas réellement de goulet d'étranglement au sein de l'algorithme. Certes, la phase d'exécution des tests d'intersection rayon/triangle demande plus de temps que les autres, mais dans une majorité des cas, la différence de performance entre cette phase et les autres ne semble pas considérable. On pourrait d'ailleurs être surpris par cette faible différence entre cette étape et les autres. En réalité, bien que cette phase soit assez gourmande en calculs, elle demande une charge de travail quasi-constante entre les différents threads. De plus, les threads d'un même warp réalisent souvent des tests d'intersections sur les mêmes triangles (ou sur des triangles contigus en mémoire). Les données à aller récupérer en mémoire concernant les triangles sont donc de manière générale cohérentes au sein d'un même warp. Les données associées aux rayons sont elles aussi assez cohérentes, puisque pour un même triangle, les rayons testés sont généralement assez proches en mémoire (idem pour des triangles voisins). Grâce aux mécanismes de cache disponibles sur les cartes de dernière génération, nous parvenons ainsi à une utilisation de la bande-passante très correcte (75 Go/s). En comparaison, les autres phases peuvent également paraître un peu lourdes, même si la phase de préparation des tests d'intersection, autrefois bien trop lourde, a récemment bénéficié d'un facteur d'accélération important (facteur 4), grâce à une meilleure répartition de la charge de travail. Ces améliorations seront abordées dans la partie suivante (voir 2.5.3 pour une description détaillée de l'algorithme employé), ainsi que quelques pistes de travail pour la phase de construction de la liste de couples triangle/cellule, via des algorithmes de rastérisation plus adaptés au GPU.

Au final, même si certaines parties de l'algorithme mériteraient une optimisation plus poussée, nous pouvons surtout noter que notre algorithme permet des performances interactives sur des scènes complexes, paraissant très correctes en comparaison de l'état de l'art. En effet, pour évaluer plus précisément les performances de notre algorithme, nous avons mesuré le nombre de rayons traités par seconde (voir colonne *ratioRays* du tableau 2.4). Il apparaît que nous traitons plus de 17 millions de rayons par seconde sur la scène de la Conférence (rappelons les 69 millions de rayons primaires traités par seconde par [GL10] sur cette même scène). Or, il convient de rappeler que le stockage de toutes les intersections au sein de la scène nous impose une phase de préparation des tests d'intersection non négligeable, ainsi

Scène	nbInters	total	ratioRays	ratioInters
Erw6	1.2M	25.9	40.5	46.3
Fairy	2.2M	49.3	21.3	44.6
Conference	2.7M	60.4	17.4	44.7
Sully	3.7M	93.3	11.2	39.7
NuclearCase	5.7M	107.3	9.8	53.1

Tableau 2.4 – Statistiques de performance de l’algorithme global de triangle-tracing, lors du lancer de 1024×1024 rayons primaires. *total* indique le temps total pour l’exécution de la simulation, en millisecondes. La colonne *ratioRays* donne le nombre de millions de rayons traités par seconde. Le nombre de millions d’intersections traitées par seconde est repris dans la colonne *ratioInters*.

qu’une étape de tri/transfert des intersections très coûteuse. Surtout, nous reconstruisons ici l’intégralité de notre structure d’accélération pour chaque pas de temps, ce qui nous permet de traiter des scènes extrêmement “dynamiques” sans ralentissement (là où les 69 millions de rayons primaires annoncés par [GL10] ne tiennent pas compte de la phase de construction du BVH de la scène).

Devant la nature particulière de notre problème, il paraît cependant plus pertinent d’évaluer le nombre de millions d’intersections traitées par seconde par notre algorithme (voir colonne *ratioInters* du tableau 2.4). Nous observons alors sur ce tableau que nous atteignons sur la scène NuclearCase un ratio de 53 millions d’intersections traitées par seconde, tandis que sur les autres scènes, nous dépassons quasiment à chaque fois les 40 millions d’intersections par seconde. Nous notons surtout que ce ratio semble bien plus constant que le précédent, montrant sans doute sa plus grande pertinence. Malgré ces résultats satisfaisants, nous gardons cependant à l’esprit que toute comparaison rigoureuse avec les travaux de la littérature reste ici compliquée, nos besoins distincts nous amenant vers des solutions fondamentalement différentes.

2.4.3 Comparaison avec l’algorithme traditionnel

Si cette comparaison avec l’état de l’art est difficile à entreprendre, nous pouvons néanmoins étudier la pertinence de notre choix d’inversion de l’algorithme traditionnel, en nous comparant avec nos propres performances. En effet, afin de valider notre algorithme, nous avons implémenté l’algorithme traditionnel, à savoir une construction entière de la grille de triangles, et un lancer de rayons classique lui succédant. Les résultats de ces tests sont présentés dans le tableau 2.5.

Scène	init	lancerRayons	tri - transfert	total	totalAlgoInversé
Erw6	11.1	7.1	4.8	23.0	26.4
Fairy	27.8	16.2	8.9	52.9	49.3
Conference	34.6	18.0	10.2	62.8	60.4
Sully	58.5	48.3	17.0	123.8	93.3
NuclearCase	64.2	42.7	23.3	130.2	107.3

Tableau 2.5 – Comparaison des performances obtenues entre les versions classique et inversée de l’algorithme global sur la grille en perspective. *lancerRayons* donne le temps nécessaire à l’exécution de la phase de lancer de rayons (similaire au lancer de triangles), sans tenir compte de la phase de tri/transfert des intersections. *total* indique le temps global d’exécution de la simulation (version classique de l’algorithme de lancer de rayons). *totalAlgoInversé* rappelle le temps global d’exécution pour la version utilisant l’algorithme de lancer de triangles. Les temps sont donnés en millisecondes.

Avant de commenter ce tableau, signalons qu'avec notre ancien modèle de carte graphique (NVIDIA GTX 295), il n'avait pas été possible, pour la scène NuclearCase, de réaliser la construction de la grille de triangles dans le cadre de l'algorithme traditionnel. En effet, la quantité de mémoire disponible sur cette carte n'était pas suffisante pour contenir les 985 Mo de mémoire requis pour la simulation. Bien que ce problème ne se reproduise plus sur la carte NVIDIA GTX 470, de génération plus récente, il ne fait aucun doute qu'il pourrait facilement réapparaître pour des scènes légèrement plus complexes. Le second motif d'introduction de l'algorithme de lancer de triangles était un gain de performance venant d'une réduction de la quantité de données sur laquelle le tri, étape prédominante de la construction de la grille, était effectué. Comme on peut le voir sur ce tableau, l'objectif de gain de temps est réalisé. Il est néanmoins bien moins important que lors de nos premières implémentations, en raison des progrès réalisés sur l'implémentation du tri dans la librairie Thrust. Sur des scènes générant un nombre de couples triangle/cellule très important, comme la scène Sully, la différence de performance est cependant notable (l'algorithme classique est "32 % plus lent").

En comparant ce tableau avec le tableau 2.3, on observe, sans surprise, que la phase d'initialisation (comportant la création de la grille en perspective) est ici bien plus longue que pour la version "inversée" de l'algorithme (près de deux fois plus de temps pour la version classique sur la scène de la Conférence, ce facteur augmentant encore pour les scènes Sully et NuclearCase). Le tableau 2.6 permet d'observer plus en détails la répartition des temps de calculs lors de cette étape d'initialisation. On note ainsi que la création de la grille contribue pour une large part à la phase d'initialisation. Cependant, on observe (comme pour le cas de la grille indexant les rayons de la scène) que le tri demande moins de la moitié du temps de construction de la grille. Comme pour le cas de la "grille de rayons", la réduction segmentée (revoir 2.2.3 si besoin est), s'effectuant ici sur un vecteur beaucoup plus grand (de la taille du nombre de couples triangle/cellule identifiés après rastérisation), est l'autre phase dominante de cet algorithme de création de la grille en perspective "classique".

Scène	tri couples	création grille	total init
Erw6	4.0	8.6	11.1
Fairy	7.1	16.7	27.8
Conference	9.3	22.2	34.6
Sully	17.7	39.8	58.5
NuclearCase	17.8	37.3	64.2

Tableau 2.6 – Performances détaillées de la phase d'initialisation, pour l'algorithme traditionnel de lancer de rayons. Les temps sont donnés en millisecondes.

Toujours en comparant les tableaux 2.3 et 2.5, on observe que les phases de lancer de rayons et de lancer de triangles sont assez équivalentes en termes de durée, même si, pour les scènes Sully et NuclearCase, de manière assez surprenante, le lancer de rayons est plus lent que le lancer de triangles. Cette différence est due à des problèmes de cohérence mémoire, plus fréquents dans le cadre de l'algorithme traditionnel. En effet, un triangle intersectant plusieurs rayons le fait en principe avec des rayons assez proches géographiquement (donc, dans de nombreux cas, proches en mémoire). En revanche, un rayon intersectant plusieurs triangles intersecte normalement des triangles figurant dans des parties de l'espace différentes. Lors de calculs associés à des rayons successifs en mémoire (par exemple les tests d'intersection), les données à aller chercher en mémoire sont ainsi bien moins cohérentes (45 Go/s de bande passante durant les tests d'intersection pour l'algorithme classique sur la scène Sully, contre 70 pour notre version inversée de l'algorithme).

Néanmoins, on pourrait penser que le tri supplémentaire sur les indices de rayons intersectés que nous effectuons pour l'algorithme "inversé" devrait compenser le gain de temps

réalisé sur les premières étapes de l'algorithme. En réalité, dans la plupart des cas, le nombre de couples triangle/cellule est bien plus important que le nombre réel d'intersections observées. Le tri sur l'ensemble des couples triangle/cellule, effectué durant la création de la grille, demande donc plus de temps (en tout cas pour les scènes de taille importante) que le tri sur les couples rayon/cellule, suivi du tri sur l'ensemble des intersections (n'oublions pas non plus la phase de réduction segmentée, bien plus rapide lors de la construction de la “grille de rayons”). En conclusion, nous observons donc, pour des scènes complexes, un gain réel de performance, qui valide notre choix d'inverser l'ordre d'exécution de l'algorithme traditionnel.

2.4.4 Variations de la résolution de la grille en perspective

Afin de conclure cette partie de présentation des résultats obtenus, nous souhaitons étudier l'impact du choix de la valeur de la résolution de la grille en perspective utilisée. Dans ce but, nous avons mesuré, pour de nombreuses résolutions différentes, l'évolution des performances de l'algorithme de lancer de triangles (nous ne tenons pas compte de la phase de tri, indépendante de la résolution choisie). Pour chaque nouvelle résolution, nous avons ainsi mesuré les temps d'exécution des quatre phases majeures de cet algorithme. Les résultats, respectivement associés aux scènes Conference et Sully, sont visibles sur les figures 2.9 et 2.10.

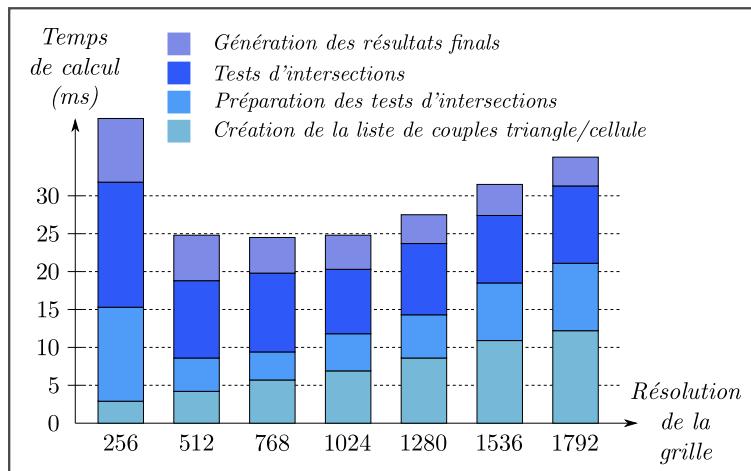


Figure 2.9 – Évolution des performances de l'algorithme de lancer de triangles en fonction des variations de la résolution de la grille en perspective utilisée, sur la scène Conference.

En premier lieu, ces figures nous permettent d'observer qu'en dehors des résolutions extrêmes, les performances de l'algorithme de lancer de triangles restent convenables pour une large plage de résolutions. Ainsi, bien que la résolution 768 * 768 semble optimale pour les deux scènes, les résolutions 512*512, 1024*1024 et 1280*1280 donnent des résultats assez similaires. Nous pouvons cependant remarquer que la résolution 1280*1280 donne des résultats moins satisfaisants que les trois autres résolutions pour le cas de la scène Conference, tandis que pour la scène Sully, c'est au contraire la résolution 512*512 qui produit les moins bonnes performances (toujours parmi ces quatre résolutions). Cette différence de comportement s'explique par le fait que les étapes 2.3.5 à 2.3.8 de l'algorithme deviennent vite dominantes lors du traitement de grandes scènes, et souffrent donc assez vite d'une diminution de la résolution de la grille, impliquant certes moins de couples cellule/triangle créés, mais aussi beaucoup plus de tests d'intersection rayon/triangle effectués. Ainsi, on observe, de manière générale,

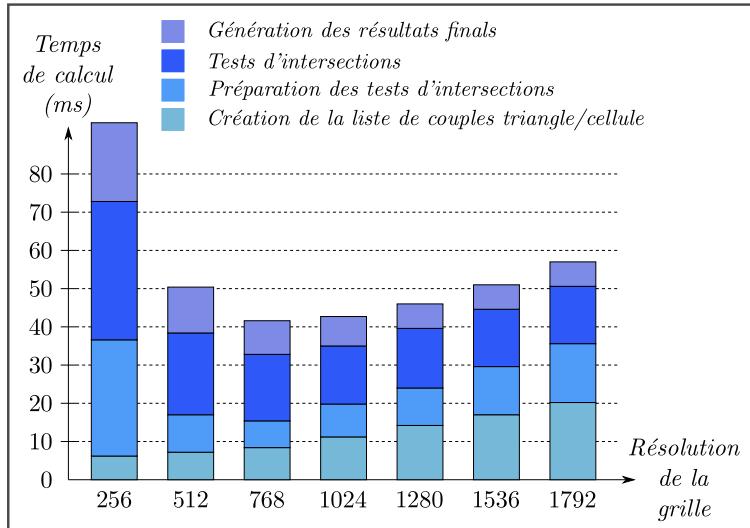


Figure 2.10 – Évolution des performances de l’algorithme de lancer de triangles en fonction des variations de la résolution de la grille en perspective utilisée, sur la scène Sully.

que les résolutions très faibles permettent une création de la liste de couples cellule/triangle très rapide, mais que le nombre de tests d’intersections générés devient vite trop néfaste à la suite de l’algorithme. Pour des résolutions très importantes, le nombre de tests d’intersections est moins élevé, et les phases 2.3.6 à 2.3.8 sont donc plus rapidement exécutées. Cependant, la phase de “rastérisation” (construction des couples) est naturellement bien plus lente, tout comme la phase de préparation des tests d’intersections, elle aussi liée au nombre de couples cellule/triangle à traiter. Le choix d’une résolution optimale de la grille en perspective demande donc d’effectuer un compromis en fonction des variations des deux quantités “nombre de couples” / “nombre de tests”. Comme nous l’avons cependant déjà remarqué, la plage de résolutions permettant de trouver un bon équilibre entre ces deux valeurs semble assez large. Nous avons pour notre part choisi d’utiliser la résolution $1024 * 1024$ pour les tests de performance réalisés dans ce chapitre, en raison du bon compromis qu’elle représente, et de son caractère “naturel” : en effet, lors du traitement de $1024 * 1024$ rayons primaires, cette résolution semble particulièrement adaptée à l’algorithme, puisqu’elle permet normalement d’associer chaque cellule de la grille à un rayon différent. Ainsi, si la plupart des triangles de la scène sont d’une taille raisonnable, la liste des triangles intersectés par un rayon est censée être quasiment équivalente à la liste de triangles qu’intersecte la cellule dans lequel il tombe. Nous étudierons dans le chapitre 4 (voir 4.3) les conséquences des variations de la résolution de la grille en perspective utilisée pour le cas de paquets moins homogènes, et nous verrons que cette règle consistant à choisir une résolution de la grille équivalente à la résolution du groupe de rayons donne, de manière générale, des résultats très satisfaisants.

Ces dernières mesures de performances nous permettent ainsi de terminer la présentation des résultats apportés par nos deux algorithmes de construction et de parcours de grille en perspective sur GPU.

2.5 Sources d’optimisation supplémentaires

Bien que nos méthodes de calculs nous permettent d’obtenir des performances interactives, nous pensons qu’il est encore possible d’optimiser certaines parties de nos algorithmes.

Nous présentons donc dans cette partie des pistes de travail pour des optimisations futures (certaines de ces optimisations ayant en réalité déjà été effectuées, voir [2.5.3](#)).

2.5.1 Construction de la grille de rayons

En premier lieu, d'après [\[ZM10\]](#), qui propose une implémentation CUDA de l'algorithme de rastérisation irrégulière, il semblerait que la construction de la grille de rayons que nous utilisons (via un tri sur les indices de rayons) pourrait être avantageusement remplacée par l'exploitation d'opérations atomiques. Ainsi, [\[ZM10\]](#) effectue un décompte du nombre de rayons par cellule, suivi d'un scan parallèle sur le vecteur rassemblant ces décomptes. Il génère ainsi le vecteur *débutCelluleIds* de notre algorithme. Pour effectuer ce décompte, il fait appel aux opérations atomiques permises par CUDA, qui permettent à chaque rayon d'incrémenter le compteur associé à la cellule dans laquelle il tombe, sans générer de conflits avec les autres threads : si conflit d'écriture il y a entre deux threads, le compteur est alors proprement incrémenté de deux unités. Au passage, lors de cette opération atomique, chaque thread récupère une information sur le nombre de rayons déjà comptés dans la cellule avant lui (avant, donc, qu'il n'incrémente ce nombre d'une unité). Cette information permet de connaître la place du rayon dans la cellule (cette place peut être quelconque, ce procédé permet seulement de s'assurer que deux rayons différents tombant dans la même cellule ne vont pas avoir la même place finale). Si un rayon tombe dans la cellule d'indice *c*, et qu'il est le rayon numéro *i* dans sa cellule, il devra donc écrire son indice à la position *débutCelluleIds[c] + i* dans le vecteur *rayIds* (naturellement, après la phase de scan parallèle sur le vecteur *débutCelluleIds*).

Les opérations atomiques en mémoire globale s'avérant assez lourdes, [\[ZM10\]](#) propose une manière astucieuse de réaliser une grande majorité de ces opérations atomiques en mémoire partagée, où elles sont effectuées beaucoup plus rapidement (dans les grandes lignes, les compteurs associés aux cellules sont incrémentés localement aux blocs en mémoire partagée, puis on fait appel aux opérations atomiques en mémoire globale pour mettre en commun ces informations locales). Grâce à cette implémentation, [\[ZM10\]](#) divise presque son temps de construction de grille par deux. Nous aimerions donc dans le futur tester cette optimisation potentielle. Cependant, la réussite d'une telle stratégie n'est pas du tout garantie dans notre cas, puisque [\[ZM10\]](#) met à profit le fait que la résolution de la grille qu'il utilise est limitée (seulement 128*64) afin d'implémenter ses opérations atomiques en mémoire partagée (il peut ainsi séparer cette grille en quatre parties, et faire tenir chacune de ces parties en mémoire partagée). Dans notre cas, le traitement d'une grille de résolution 1024*1024 ne pourrait donc pas bénéficier de ces opérations atomiques en mémoire partagée. Néanmoins, cette technique pourrait être utilisée pour un premier tri des indices de cellules suivant leurs bits de poids fort (ce qui permettrait ainsi d'économiser quelques étapes du tri *radix*) avant de revenir à un tri plus classique pour les bits de poids faible.

2.5.2 Rastérisation des triangles

Pour des travaux futurs, nous aimerions également envisager une stratégie différente pour la rastérisation des triangles. En effet, comme nous l'avons vu, nous avons opté pour une stratégie relativement simple à implémenter, à savoir la méthode *scanline*, qui consiste à parcourir les triangles de haut en bas, et à identifier, pour chaque ligne qu'occupe le triangle, les cellules que le triangle chevauche. Pour cela, nous partons du sommet haut du triangle, puis nous identifions, pour la ligne contenant ce sommet haut, les cellules de début et de

fin du triangle. Ces cellules sont obtenues à partir de l'identification de l'abscisse du point d'intersection entre la frontière basse de la ligne en question et chacune des deux arêtes (nous nous plaçons ici dans le cas le plus simple, une autre configuration peut être trouvée sur la droite de la figure 2.5). Nous descendons ensuite le long de chaque arête partant de ce sommet. Pour chaque nouvelle ligne, il nous suffira d'incrémenter cette abscisse de la pente de l'arête associée, et, pour trouver les cellules de début et de fin associées, de prendre les parties entières de ces abscisses (pour une grille 1024*1024, nous générerons des coordonnées de points et de triangles entre -512 et $+512$, de façon que les frontières de cellules correspondent à des coordonnées entières dans les axes du repère de la grille). Il est à noter qu'au moment de calculer les parties entières, nous “gonflons” toujours artificiellement les abscisses calculées, afin d'éviter les problèmes de précision dus à l'accumulation des erreurs lors de l'ajout de la pente pour chaque nouvelle ligne étudiée. Nous garantissons ainsi de ne jamais “rater” une cellule intersectée par un triangle.

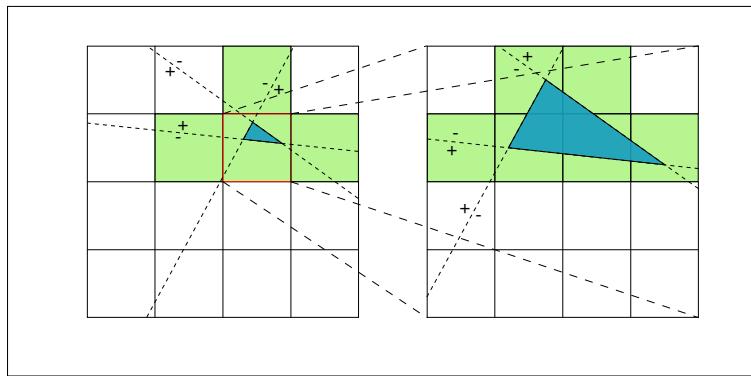


Figure 2.11 – Schéma montrant les deux premières étapes d'une rastérisation hiérarchique pour un triangle donné. Au premier niveau (gauche du schéma), seules quatre cellules (en vert) sont conservées. La décision de non-conservation se base sur les déterminants 2D associés aux arêtes du triangle. Si les quatre sommets d'une cellule sont du côté extérieur d'une arête, la cellule n'est pas conservée. Le traitement de la cellule rouge lors de la deuxième étape de la rastérisation hiérarchique est illustré sur la droite du schéma.

Cette méthode peut demander de nombreux calculs afin d'identifier les indices de cellules de début et de fin sur chaque ligne, mais surtout, elle peut créer d'importants déséquilibres de la charge de travail entre les différents processeurs scalaires, certains triangles pouvant occuper beaucoup plus de lignes, et certaines lignes pouvant être beaucoup plus longues que d'autres. Nous aimerions donc dans le futur essayer d'évaluer des stratégies de rastérisation différentes, en particulier une stratégie hiérarchique (qui remplacerait donc les phases 2.3.1 à 2.3.4 de notre algorithme), comme celle proposée pour l'architecture Larrabee [SCS⁰⁸]. Une telle stratégie (voir figure 2.11) consiste à effectuer la rastérisation sur une grille à grain très grossier et, pour le cas où un triangle touche une cellule de cette grille grossière, à subdiviser cette nouvelle cellule, et ainsi de suite jusqu'à arriver, si besoin est, à la vraie résolution de la grille. La rastérisation, pour les premiers niveaux de la grille hiérarchique (grain grossier), peut être remplacée par des tests simples (dits de “rejet trivial”) permettant d'identifier rapidement une majorité de cas pour lesquels un triangle ne touche pas une cellule. La simplicité de ces tests fait qu'il est possible de subdiviser une cellule qui n'est pas du tout touchée par un triangle, et donc de générer une charge de travail inutile. Cependant, la perte de performance occasionnée par ces erreurs est en principe compensée par le gain de temps lors de l'exécution des tests de “rejet trivial”. Sur GPU, une stratégie possible serait, pour le premier niveau de la grille hiérarchique, de faire traiter chaque triangle par un warp de threads, chaque thread

étant associé à une cellule de la grille. Cette première étape permettrait d'identifier pour chaque triangle les cellules de la grille grossière qu'il occupe. Il serait alors possible de générer une première liste de couples triangle/cellule de la grille grossière, puis d'appliquer la même stratégie pour les niveaux plus fins de résolution. Cette méthode présente l'avantage d'être naturellement parallèle, mais le traitement des gros triangles pourrait également demander beaucoup de travail inutile : en effet, pour un triangle très large, il devrait normalement être possible de voir que ce dernier recouvre intégralement certaines cellules de la grille grossière. Il serait par conséquent inutile de subdiviser de telles cellules.

Une séparation des stratégies entre les “gros” et les “petits” triangles pourrait alors être utile, comme elle pourrait d'ailleurs l'être pour notre stratégie actuelle : en effet, nous pourrions, dans le cadre de notre implémentation actuelle, envisager de faire traiter les gros triangles ou les grandes lignes par des groupes de threads plutôt que par un seul thread. Reste alors à voir comment réorganiser ce travail et comment différencier les “gros” et les “petits” triangles. Une solution intéressante pourrait peut-être également consister en une subdivision des gros triangles en groupes de petits triangles (tout en préservant la topologie du maillage), ce qui permettrait de garantir une certaine homogénéité entre les différents triangles traités, et d'ouvrir la voie vers des stratégies de rastérisation très adaptées au traitement de tels polygones [FLB⁺09, EL10]. Le problème reste donc encore très ouvert, ce qui n'est guère étonnant, étant donné la multitude de stratégies proposées par le passé pour l'algorithme de rastérisation.

2.5.3 Meilleure utilisation de la mémoire

Le dernier point d'optimisation que nous souhaitons aborder rejoint en partie le précédent : en effet, comme nous l'avons vu, les phases 2.3.2, 2.3.4 et 2.3.6 de notre algorithme de lancer de triangles consistent toutes à faire écrire par chaque thread un certain nombre de données en mémoire, ce nombre variant suivant les threads, et de telle manière que deux threads consécutifs écrivent dans deux zones mémoire consécutives. Malheureusement, dans le cas où un thread doit écrire un grand nombre de données, le thread suivant devra écrire dans un endroit très éloigné en mémoire, ce qui va générer des accès en mémoire non coalescents (incohérents), et grandement diminuer nos performances. Nous avons donc dû revoir l'organisation de ces accès mémoire, ce qui nous a permis de diviser par 4 les temps de calcul associés à la phase 2.3.4 (de 6 à 1.5 millisecondes sur la scène Sully), consistant à écrire en mémoire les couples triangle/cellule, en faisant travailler chaque thread sur une ligne de triangle différente. Chaque ligne pouvant vite contenir un grand nombre de cellules, cette phase de l'algorithme souffrait en effet considérablement d'accès de mémoire non coalescents (bande passante de 15 Go/s). Les solutions que nous avons apportées à ce problème nous ont permis de mieux répartir la charge de travail des différents threads, et de grandement améliorer la coalescence des accès en mémoire (105 Go/s de bande passante). Afin de décrire l'algorithme mis en place, nous nous concentrerons ici sur la description de l'optimisation de cette phase 2.3.4 (bien que cette méthode puisse assez aisément être appliquée aux phases 2.3.2 et 2.3.6 de l'algorithme).

Une première solution envisageable afin de résoudre ce problème pourrait être de conserver l'organisation globale des calculs (un thread pour chaque ligne), mais de mettre en place une organisation différente des écritures, de manière à ce que les threads d'un même groupe écrivent leur première donnée consécutivement en mémoire (voir Figure 2.12), puis leur deuxième donnée côté à côté, et ainsi de suite jusqu'à ce que chaque thread n'ait plus de données à écrire.

Si un tel schéma améliore grandement la cohérence de nos écritures en mémoire, il im-

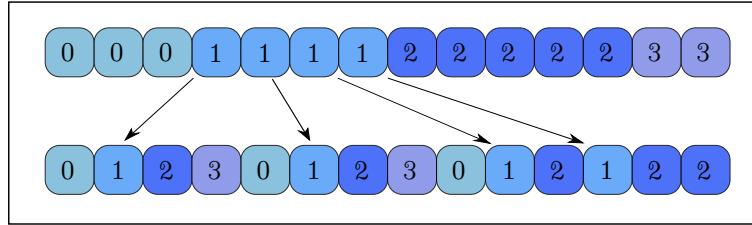


Figure 2.12 – Exemple de réorganisation des écritures en mémoire permettant des accès plus cohérents entre les différents threads. Les indices des cellules donnent les numéros des lignes à traiter (donc des threads qui s'en occupent). Sur la ligne du haut, les écritures effectuées par un thread sont contiguës. Sur la ligne du bas, les écritures sont entrelacées, de manière à ce que les $i^{\text{ème}}$ données écrites par les threads soient contiguës pour toute valeur de i .

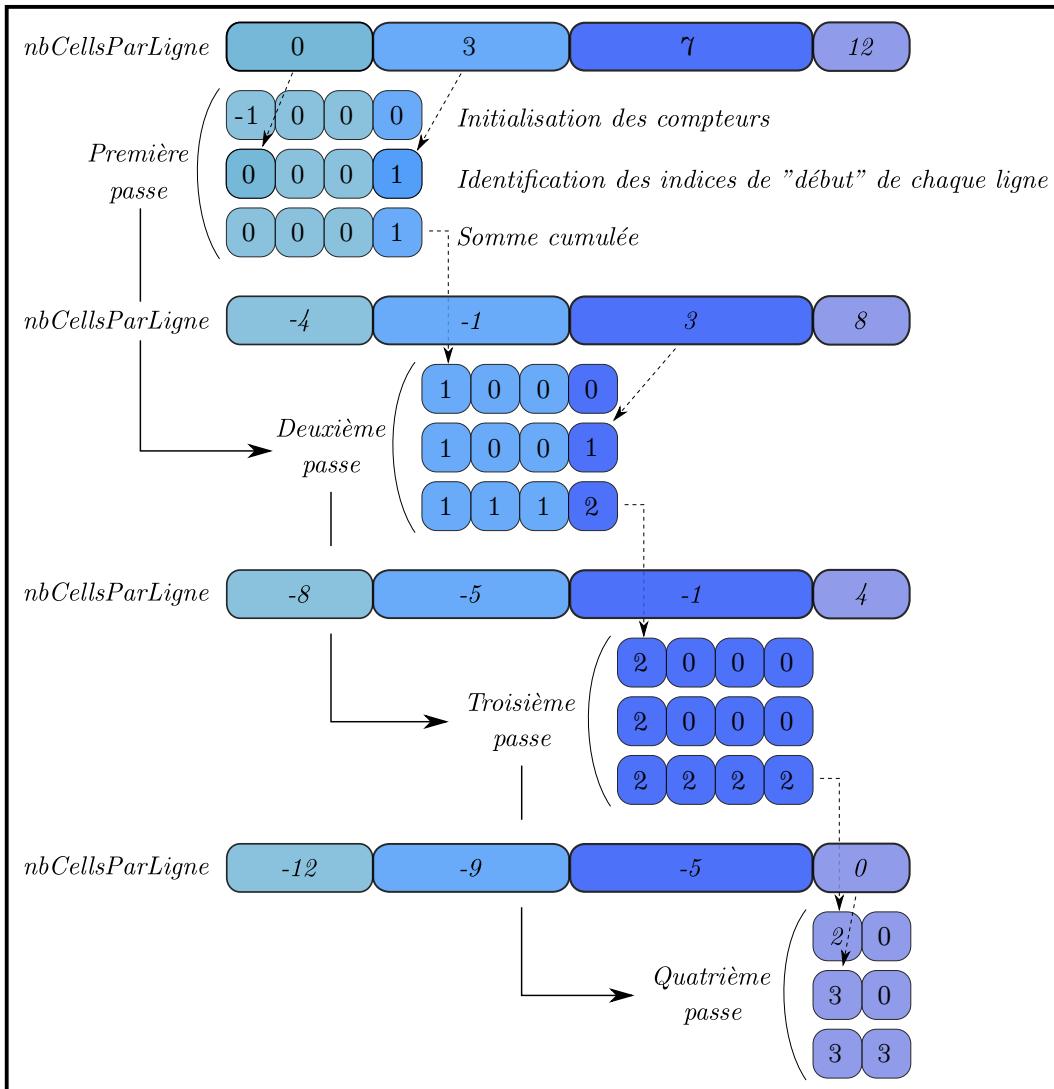


Figure 2.13 – Stratégie efficace de remplissage du vecteur de couples cellule/triangle (phase 2.3.4 de l'algorithme de triangle-tracing). Les données utilisées sont les mêmes que pour la figure 2.12. Les différentes couleurs des cellules correspondent aux couleurs des lignes de triangles traitées. Pour chaque passe de calculs, nous affichons les valeurs successives du tableau de compteurs situé en mémoire partagée (trois étapes). Chaque passe est également associée à une mise à jour du tableau $nbCellsParLigne$, d'où la répétition de ce tableau sur la figure.

plique également l'apparition de nouvelles difficultés. En effet, une telle stratégie provoque l'entrelacement de données propres, par exemple, à des triangles différents, ce qui peut perturber la cohérence des accès mémoire pour les parties ultérieures de l'algorithme. Ainsi, dans la version actuelle de l'algorithme, les tests d'intersection effectués par des threads successifs concernent la plupart du temps le même triangle et donc des rayons potentiellement proches. Dans cette nouvelle version, ces mêmes tests seraient effectués sur des triangles différents, et des rayons en principe moins proches géographiquement, ce qui pourrait considérablement réduire la cohérence de nos accès mémoire. De plus, chaque thread d'un bloc restant associé à une ligne de triangle, nous ne résolvons pas réellement les problèmes de répartition de la charge au sein d'un warp de threads, puisque certains threads peuvent finir le traitement de la ligne qui leur est attribuée bien plus rapidement que les autres threads du même warp.

Afin de réellement améliorer la répartition de la charge de travail et d'éviter l'entrelacement de données cohérentes, nous avons donc mis en place une autre méthode, sans modification des positions où les données sont écrites : cette stratégie consiste toujours à faire travailler un bloc de n threads sur n lignes différentes. En revanche, chaque thread de ce bloc ne travaille plus sur une ligne différente, mais sur une cellule à la fois (lors de l'exécution d'un kernel, un même thread traitera en revanche, en principe, plusieurs cellules). Afin de traiter toutes les cellules correspondant aux n lignes manipulées par le bloc, il nous faut donc réaliser plusieurs passes de calculs sur le groupe de threads. L'algorithme finalement retenu est représenté sur la figure 2.13. Cette figure décrit le traitement du cas simple observé sur la figure 2.12, pour lequel 4 lignes sont à traiter, les lignes 0, 1, 2 et 3 comptant respectivement 3, 4, 5 et 2 cellules (14 cellules en tout). Le tableau *nbCellsParLigne* figurant en haut de la figure 2.13 est le tableau produit à la fin de l'étape 2.3.3 et représente donc la somme cumulée du nombre de couples par ligne à traiter. Nous faisons ici la supposition que le bloc de threads traitant ces 4 lignes n'est composé que de 4 threads. Afin de traiter l'ensemble des cellules associées aux 4 lignes, il nous faut donc réaliser $(14/4 + 1 = 4)$ passes de calculs.

La première passe de calculs sera donc dédiée à l'écriture en mémoire de 4 couples triangle/cellule, ces 4 couples étant contigus en mémoire, et le premier de ces couples étant le premier couple associé à la première ligne traitée par le bloc de threads. Nous voulons donc pour cette étape que chaque thread écrive en mémoire une de ces données, de façon que les écritures réalisées soient parfaitement coalescentes. Chaque thread ne travaillera donc plus du tout sur une ligne différente. Pour cette première passe de calculs (voir figure 2.13), les trois premiers threads travailleront sur la ligne 0, tandis que le quatrième et dernier thread traitera le premier couple cellule/triangle de la ligne 1. Lors de cette première passe de calculs, la difficulté n'est pas de déterminer où chaque thread doit écrire, puisque les threads écrivent dans des espaces contigus en mémoire. En revanche, il est plus compliqué de savoir quelle donnée chacun de ces threads doit écrire en mémoire. En effet, chaque thread doit d'abord identifier sur quelle ligne il travaille pour cette passe de calculs. Dans ce but, chaque thread se voit attribuer un compteur, dont nous allons maintenant étudier la manipulation (l'allocation du tableau de compteurs en question se fait en mémoire partagée, en raison des échanges nécessaires entre les différents threads).

Afin d'identifier la ligne sur laquelle il doit travailler, chaque thread doit prendre en compte le nombre de couples associés à chaque ligne traitée par des threads le précédent. Sur notre exemple, les trois premiers threads travaillant sur la ligne 0, le thread 4 travaillera sur la ligne 1. Pour identifier cette information, il est nécessaire que le thread 4 sache sur quelle ligne les précédents threads ont travaillé. Pour cela, nous allons donc effectuer une somme cumulée, permettant de connaître précisément la ligne associée à chaque thread.

L'identification de cette ligne associée à chaque thread se fait ainsi en trois étapes. Tout d'abord, chaque compteur associé à un thread est initialisé à 0, sauf le premier compteur,

initialisé à -1. L'étape suivante est la seule étape durant laquelle chaque thread est associé à une ligne différente et non plus à une cellule. Lors de cette étape, le thread i lit en mémoire la valeur $nbCellsParLigne[i]$. Si cette valeur est inférieure à n (nombre de threads du bloc et de lignes à traiter, ici égal à 4), cela signifie qu'au moins un des threads du bloc va devoir travailler sur cette ligne (pour cette première passe uniquement, chaque thread pouvant travailler sur des lignes différentes lors de passes ultérieures). Pour marquer cette information, nous incrémentons de 1 (via une opération atomique en mémoire partagée) le compteur associé au thread $nbCellsParLigne[i]$ (voir, pour chaque passe, les flèches en pointillés sur la figure 2.13). Ainsi, sur l'exemple de la figure 2.13, le thread 1 va lire la valeur $nbCellsParLigne[1] = 3$. Il va donc incrémenter le compteur numéro 3 d'une unité. De même, le thread 0 va incrémenter le compteur $nbCellsParLigne[0] = 0$ d'une unité (le fait d'avoir initialisé le compteur 0 à la valeur -1 est dû au fait que le premier couple associé à la ligne 0 est toujours situé en position 0). Après cette étape, chaque thread recommence à travailler sur un couple triangle/cellule et non sur une ligne de triangle.

Nous effectuons ensuite une somme cumulée, qui nous permet d'obtenir, pour chaque thread, la ligne du couple sur lequel il doit travailler. Nous implementons cette somme cumulée au niveau du bloc, via la stratégie de Brent-Kung [BK82], suivant l'algorithme PRAM développé par Blelloch [Ble90]. Sur la figure 2.13, nous observons bien alors qu'à l'issue de cette opération de somme cumulée, les trois premiers threads ont leur compteur à 0, tandis que le quatrième a son compteur à la valeur 1.

Afin de réaliser l'opération d'écriture qui lui est attribuée, chaque thread doit maintenant calculer l'indice du couple sur lequel il doit travailler à l'intérieur de la ligne. Pour cela, si nous nommons *compteurs* le tableau de compteurs considéré, le thread i doit seulement calculer la valeur $i - nbCellsParLigne[compteurs[i]]$. En effet, la valeur $compteurs[i]$ désignant la ligne à laquelle il est associé, $nbCellsParLigne[compteurs[i]]$ désigne l'indice du premier thread devant travailler sur cette ligne. La différence entre i et cette valeur donne donc bien la position du couple traité par le thread vis-à-vis de la ligne étudiée.

Une fois cette première passe effectuée, les passes suivantes se déroulent quasiment de la même manière. La principale différence provient du fait que toutes les valeurs du tableau $nbCellsParLigne$ sont décrémentées de n (rappelons ici que $n = 4$) avant chaque passe (cette opération permet de prendre en compte les écritures déjà réalisées lors de passes précédentes). De plus, la phase d'initialisation des compteurs est légèrement différente : les compteurs autres que celui d'indice 0 sont là aussi initialisés à 0. En revanche, le premier compteur est initialisé à la valeur du dernier compteur issu de la phase précédente (voir les flèches en pointillés entre les passes successives sur la figure 2.13). Cette initialisation nous permet de prendre en compte les couples déjà traités pour le calcul de nos sommes cumulées et donc d'évaluer correctement la ligne associée à chaque thread. En raison de la modification du tableau $nbCellsParLigne$, il faut ici, au moment de la deuxième phase de travail sur les compteurs, ne procéder à l'incrément de compteur que si la valeur $nbCellsParLigne[i]$ est inférieure à n (ce qui était déjà le cas lors de la première phase), mais également supérieure à 0. Ces précautions nous permettent de réaliser ensuite l'ensemble des passes de l'algorithme et de garantir en sortie la production d'un tableau de couples identique à celui généré suivant la stratégie classique “un thread = une ligne”.

Comme nous l'avons déjà souligné, cet algorithme nous a permis de grandement améliorer nos performances lors de l'exécution de la phase 2.3.4. Nous n'avons pas eu le temps nécessaire pour appliquer ces méthodes à d'autres phases de l'algorithme, mais nous pensons que la démarche globale consistant à répartir de manière homogène les opérations d'écriture entre les différents threads d'un bloc pourrait s'avérer utile dans de nombreux cas.

Conclusion

Nous avons présenté dans ce chapitre deux algorithmes de création et de parcours de la grille en perspective sur GPU. Ces algorithmes reposent sur une inversion préalable de la structure traditionnelle de l'algorithme de lancer de rayons. Plutôt que de construire une grille indexant les triangles de la scène, nous répertorions dans notre grille en perspective les rayons constituant le groupe étudié. Nous faisons ensuite parcourir cette grille à l'ensemble des triangles de la scène, dans une phase que nous nommons “lancer de triangles”, par analogie avec le “lancer de rayons”. Cette inversion nous permet de diminuer nos temps de calcul, mais aussi de faciliter le traitement de très gros modèles, pouvant ainsi être traités par “passes” successives.

Les deux algorithmes GPU que nous avons conçus reposent sur l'utilisation récurrente d'algorithmes parallèles fondamentaux, comme le scan parallèle (somme cumulée) ou le tri. Nous avons également présenté plusieurs optimisations possibles propres à certains de nos kernels, certaines d'entre elles ayant déjà été mises en place. Ces algorithmes et les optimisations déjà réalisées nous ont permis d'assurer le calcul, le tri et le transfert vers le CPU de toutes les intersections entre un grand groupe de rayons primaires (plus d'un million) et une scène volumineuse (plusieurs centaines de milliers de triangles) en une centaine de millisecondes.

Ces algorithmes ont fait l'objet d'une publication dans les proceedings du workshop *GPUScA 2011*. Une version étendue de ce papier a été acceptée pour publication dans la revue *International Journal of High Performance Computing Applications (IJHPCA)*.

Gestion de la précision

3

Sommaire

3.1 Problèmes de précision posés par l'arithmétique flottante	64
3.1.1 Difficultés liées à l'arithmétique flottante près des frontières des triangles	64
3.1.2 Choix d'un autre test d'intersection	66
3.1.3 Déterminant 2D	66
3.1.4 Algorithmes adaptatifs pour l'arithmétique exacte	67
3.2 Résolution des problèmes de précision sur GPU	68
3.2.1 Difficultés à porter les algorithmes adaptatifs de Shewchuk sur GPU	68
3.2.2 Solution proposée	69
3.2.3 Tests d'intersection 3D	70
3.3 Utilisation de la topologie du maillage	71
3.3.1 Principes	71
3.3.2 Implémentation	71
3.3.3 Cas des rayons passant près de la frontière d'un objet	73
3.3.4 Transfert des informations sur la nature des intersections vers le CPU	73
3.4 Problèmes posés par la projection des triangles	74
3.4.1 Le clipping	75
3.4.2 Changement de topologie	76
3.4.3 Clipping sur GPU	77
3.4.4 Prise en compte de la topologie du maillage initial	79
3.5 Résultats	80
3.5.1 Robustesse des calculs	80
3.5.2 Mesures de performance	81
3.5.3 Surconsommation de mémoire	84

Nous avons jusque là négligé un aspect extrêmement important de notre problème, qui nous différencie une fois de plus de la communauté du rendu graphique, à savoir la gestion des problèmes de précision. En effet, notre contexte de travail nous demande de garantir la robustesse des résultats de notre calcul de débit de dose. Dans le domaine graphique, une erreur de calcul, entraînant par exemple un affichage erroné de la couleur d'un pixel, n'a pas de conséquences comparables avec le fait de fournir une mauvaise estimation du débit de dose reçu par un opérateur (surtout dans le cadre d'un affichage temps-réel). Ce problème a donc naturellement été assez peu abordé par la communauté graphique.

Avant d'aller plus loin dans notre analyse du problème, il convient de préciser ce que nous entendons par "problèmes de précision". Il ne s'agit pas forcément de garantir des résultats "exacts", car la modélisation physique utilisée pour notre calcul de débit de dose est déjà elle-même une simplification de la réalité physique. De plus, les maillages que nous utilisons pour représenter les objets de la scène épousent au mieux les surfaces réelles des objets physiques, mais ne peuvent être qualifiés d'équivalents "exacts" de ces surfaces. Par "problèmes de

précision”, nous entendons plutôt un résultat fiable, à savoir que nous ne pouvons accepter de fournir une estimation de débit de dose fortement erronée. Or, le débit de dose calculé en un point de mesure est la “somme” des débit de dose reçus de chaque source ponctuelle de radiations. Par conséquent, si un calcul de débit de dose associé à une seule de ces sources est très inexact, le débit de dose global estimé au point de mesure peut s'avérer très différent de la valeur attendue. Lors de notre recherche d'interactions rayon-matière, une erreur sur le comportement d'un seul rayon d'un gros groupe peut donc avoir des conséquences désastreuses sur l'estimation du débit de dose qui en découle. Malheureusement, lorsque l'on se contente d'appliquer les formules classiques de tests d'intersection rayon-triangle et de faire confiance à la précision des calculs en arithmétique flottante, il peut arriver que l'on “rate” l'intersection d'un rayon avec une matière traversée. On peut alors ne pas détecter l'entrée ou la sortie d'un rayon dans un objet. Dans le pire des cas, l'estimation du débit de dose fourni à l'opérateur peut être bien inférieure au débit de dose réellement subi. Nous ne pouvons naturellement pas accepter un tel comportement et devons fournir des solutions robustes, afin d'éviter que l'intersection réelle entre un rayon et un objet de la scène ne soit pas détectée.

Dans la première partie de ce chapitre (partie 3.1), nous détaillons les raisons des difficultés liées aux calculs en arithmétique flottante et nous présentons quelques pistes de résolution apportées par la littérature. Nous expliquons dans la partie suivante (partie 3.2) comment nous résolvons ces problèmes de précision dans le cadre d'une implémentation GPU. Ces solutions tiennent compte de la topologie du maillage (partie 3.3), qui est modifiée lors de l'étape de clipping permettant de projeter correctement les triangles dans un espace à deux dimensions. Nous introduisons dans la partie 3.4 ce problème et les solutions mises en place pour le résoudre, différentes sur CPU et sur GPU. Enfin, pour clore ce chapitre, nous présentons et discutons les résultats obtenus (partie 3.5).

3.1 Problèmes de précision posés par l'arithmétique flottante

3.1.1 Difficultés liées à l'arithmétique flottante près des frontières des triangles

Afin de présenter clairement les problèmes liés à la précision des calculs effectués en arithmétique flottante, il nous faut étudier de plus près les opérations mises en jeu lors des tests d'intersection rayon-triangle. Durant les précédentes décennies, de nombreux tests d'intersections rayon-triangle optimisés ont été proposés par les communautés du lancer de rayons [Gla89, Bad90, Mol97] ou de la détection de collision [RSF01], le plus populaire restant certainement celui de Wald [Wal04]. Pour fixer les idées, concentrons-nous sur ce dernier test et voyons les erreurs de comportement qu'il peut engendrer.

Ce test repose principalement sur le calcul des coordonnées barycentriques du point d'intersection entre le rayon concerné et le plan dans lequel réside le triangle (après projection suivant un plan aligné aux axes, ce qui simplifie les calculs, mais ne modifie pas les coordonnées barycentriques du point d'intersection). A partir des coordonnées barycentriques de ce point, il est possible de déduire si ce point réside ou non dans le triangle, et donc de savoir si le rayon intersecte ou non le triangle considéré (voir figure 3.1). Considérons maintenant la question du test d'intersection entre un rayon et deux triangles conjoints (non coplanaires) partageant une même arête. Comme les plans de ces deux triangles sont différents, les deux points d'intersection entre le rayon et les deux plans des triangles (respectivement notés I_1 et I_2) sont donc eux aussi différents. Supposons maintenant que l'on fasse bouger le rayon de façon qu'il intersecte les deux triangles à la fois, en passant par cette arête qu'ils partagent. Comme cette arête constitue l'intersection entre les plans des deux triangles, les deux points

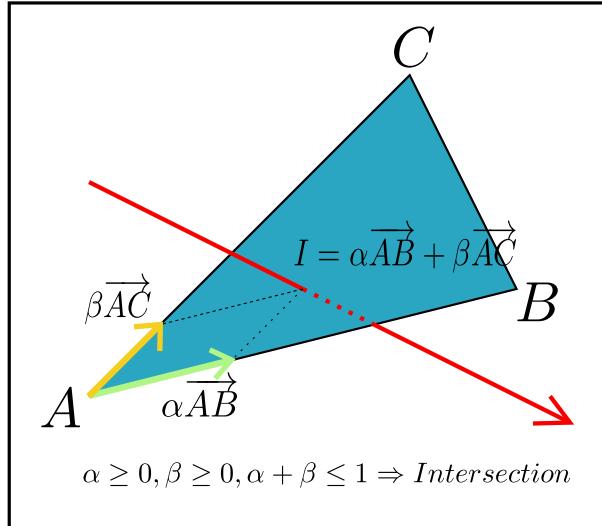


Figure 3.1 – Test d’intersection rayon-triangle optimisé, basé sur le calcul des coordonnées barycentriques, proposé par Wald.

d’intersections I_1 et I_2 devraient cette fois-ci être les mêmes. Mathématiquement, en posant le calcul à la main des formules que doit appliquer le processeur, on trouverait en effet que ces deux points sont identiques. Numériquement, cependant, ces deux points vont différer.

En effet, comme les deux triangles ne sont pas coplanaires, leurs normales sont différentes. Les formules pour calculer les coordonnées des points I_1 et I_2 ne sont donc pas les mêmes. Or, lors de calculs accomplis via l’arithmétique flottante, les arrondis successifs effectués sur les valeurs calculées peuvent entraîner de fortes imprécisions sur les valeurs finales d’une expression. Pour le calcul des coordonnées des points I_1 et I_2 , plusieurs arrondis vont donc certainement être effectués, sur des données distinctes de part et d’autre, menant, au final, à des valeurs numériques différentes pour les coordonnées de I_1 et de I_2 . La conséquence directe de ces approximations est que les coordonnées barycentriques des deux points considérés peuvent alors ne pas être cohérentes entre les deux triangles. “Par chance”, il peut arriver que l’on ne trouve qu’une seule intersection sur l’ensemble des deux triangles, mais on peut également trouver une intersection pour chaque triangle, voire, bien pire, ne trouver aucune intersection pour les deux triangles ! Nous rencontrons alors le cas introduit précédemment où une intersection avec un objet est “ratée”. Ce problème d’incohérence des tests d’intersections, dû au manque de précision des calculs en arithmétique flottante, peut se présenter sur toutes les “frontières” de triangles (arêtes ou sommets).

De manière à évaluer la fréquence de telles erreurs, nous avons simplement modélisé une sphère, que nous avons ensuite maillée (248 triangles). Nous avons alors généré dix points régulièrement espacés sur chaque arête du maillage. En ajoutant ces points aux sommets de ces 248 triangles, nous avons obtenu une liste de 4466 points, tous situés sur l’arête ou le sommet d’un triangle. Nous avons ensuite généré 4466 rayons, partant du centre de la sphère, chacun passant par un de ces 4466 points, puis mesuré les résultats des tests d’intersections rayon-triangle effectués. Pour ces tests, nous avons donc utilisé le code de Wald [Wal04], en considérant la version “inclusive” du test, à savoir qu’un point situé sur l’arête d’un triangle devrait normalement être compté comme appartenant à ce triangle.

Sur les 4466 rayons ainsi lancés, 720 ont renvoyé un nombre d’intersections différent de 1 (à noter qu’il est assez naturel que pour de nombreux rayons, on ne trouve qu’une seule intersection, malgré la nature “inclusive” du test d’intersection. En effet, les points situés sur

les arêtes de triangles ont tous été générés numériquement, et sont donc eux aussi le fruit d'approximations, ne se trouvant, la plupart du temps, pas “exactement” sur une arête de triangle). Sur les 146 rayons devant croiser un sommet de triangle, 8 rayons n'ont détecté aucune intersection, tandis que 103 ont renvoyé plus d'une intersection trouvée. Sur les 4320 rayons censés passer sur une arête de triangle, 99 rayons n'ont renvoyé aucune intersection, tandis que 510 en ont renvoyé deux. Sur les 4466 rayons générés, nous obtenons donc $99 + 8 = 107$ rayons pour lesquels aucune intersection n'a été trouvée. Ce test simple prouve bien à quel point le problème est réel et mérite d'être étudié. Nous allons donc maintenant étudier les différentes méthodes rendant plus “robuste” le test d'intersection rayon-triangle provoquant ces erreurs.

3.1.2 Choix d'un autre test d'intersection

La première solution naturelle à ce problème est tout simplement de concevoir un autre test d'intersection. En effet, cette difficulté, bien que moins contraignante pour la communauté graphique, a également été relevée. Elle peut être contournée [DSD⁺09] en adoptant un test d'intersection 3D qui se base sur les volumes formés par les extrémités des rayons et les arêtes du triangle [RSF01, KS06]. L'intérêt de ce test est de tenir compte de l'arête pour prendre sa décision, et donc de fournir des résultats cohérents entre deux triangles conjoints. Dans notre cas, où les triangles sont déjà projetés en 2D au moment du test d'intersection (nécessaire pour la création de la grille en perspective), un test équivalent, lui aussi “orienté arête”, consisterait tout simplement à tester si le rayon passe à gauche ou à droite de chacune des arêtes, en utilisant des arêtes “orientées”, de manière à définir la gauche et la droite d'une arête.

Si ce genre de solutions est simple à mettre en œuvre, et résout un grand nombre de problèmes, elle n'est malheureusement pas suffisante dans le cas de rayons croisant un triangle en son sommet. En effet, si les tests sont garantis d'être cohérents de part et d'autre d'une arête, il n'y a pas de garantie sur le fait qu'ils vont l'être “globalement”, sur l'ensemble des arêtes se rencontrant en ce sommet. Il nous faut donc, afin de garantir la robustesse de nos tests d'intersection, disposer d'un moyen simple et efficace de calculer l'information exacte (identique à celle que donnerait le résultat si on posait l'opération sur le papier) quant à la position d'un rayon par rapport à une arête. En effet, si cette information est exacte et non plus seulement cohérente entre des triangles conjoints, alors la cohérence des tests au niveau d'un sommet sera elle aussi garantie (une intersection au niveau d'un sommet résultant de l'intersection avec toutes les arêtes se rencontrant en ce sommet).

3.1.3 Déterminant 2D

Avant de décrire plus en détails les solutions possibles pour réaliser des calculs exacts, arrêtons-nous juste sur le calcul que nous voulons rendre exact. Comme nous l'avons déjà fait remarquer, nos triangles devant être projetés pour la construction de la grille en perspective, nous pouvons nous servir de cette projection 2D pour réaliser nos tests d'intersections rayon-triangle. Naturellement, il nous faut revenir aux données 3D pour calculer à quelle distance du point de départ du rayon a lieu l'intersection avec le triangle, mais si l'on souhaite savoir rapidement si l'intersection existe, on peut se contenter de ce test 2D.

Ce dernier repose sur la définition d'un déterminant 2D, nommé *edge-function* par Pineda [Pin88]. Étant donnés deux points $A(xA, yA)$ et $B(xB, yB)$, le déterminant E s'exprime en

un point (X, Y) suivant la formule :

$$E(X, Y) = (X - x_A) * (y_B - y_A) - (Y - y_A) * (x_B - x_A) \quad (3.1)$$

Cette fonction est positive si le point (X, Y) est situé “à gauche” de la droite orientée (AB) , négative s’il se trouve “à droite” d’ (AB) , nulle s’il est exactement sur (AB) . A partir de cette définition, on peut décrire un triangle comme étant l’union de trois déterminants 2D, chacun associé à une arête du triangle. Pour savoir si un rayon 3D, représenté dans notre espace en perspective par un simple point, intersecte un triangle, il suffira de calculer la valeur des trois déterminants associés au triangle en ce point, et d’étudier leurs signes (voir la figure associée 3.2).

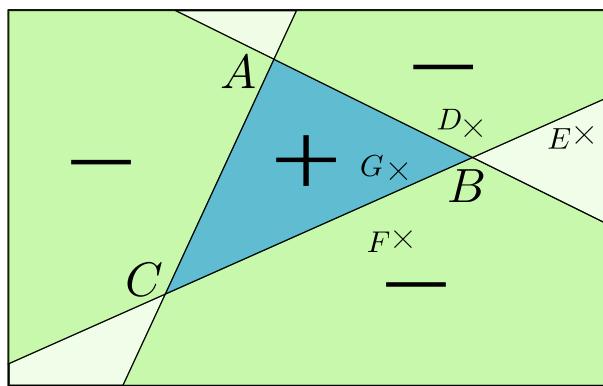


Figure 3.2 – Schéma d’un triangle ABC vu en deux dimensions. Les trois déterminants 2D liés au triangle sont positifs en G ; le point G est donc à l’intérieur du triangle. En D et F , un des déterminants est négatif, en E , deux de ces déterminants sont négatifs. Ces trois points sont donc à l’extérieur du triangle.

Le calcul de ce déterminant 2D est naturellement lui aussi sujet à approximations et peut donc renvoyer des résultats erronés. Pour le cas d’un rayon représenté par le point (X, Y) et passant à proximité d’une arête $[AB]$, le déterminant 2D associé à $[AB]$ doit en effet être quasi nul en (X, Y) et les termes $(X - x_A) * (y_B - y_A)$ et $(Y - y_A) * (x_B - x_A)$ quasi égaux en ce point. Les arrondis effectués pour évaluer ces deux termes peuvent cependant générer une erreur telle que le signe de la différence entre les deux termes va s’inverser. Le point sera alors identifié du mauvais côté de l’arête. Pour les simples problèmes de cohérence à proximité des arêtes, ce problème peut être résolu en s’assurant qu’on oriente de la même manière la droite (AB) dans les deux triangles partageant l’arête. Cependant, comme on l’a déjà dit, à proximité d’un sommet, ces problèmes de précision ne peuvent être aussi simplement résolus. Pour remédier à ces difficultés, nous voulons donc disposer d’un test nous permettant d’obtenir un résultat numérique identique au résultat mathématique renvoyé par le calcul du déterminant 2D (précisons bien que nous ne souhaitons pas évaluer exactement la valeur de ce déterminant 2D, mais seulement son signe, de manière à identifier avec précision les intersections rayon-triangle dans la scène).

3.1.4 Algorithmes adaptatifs pour l’arithmétique exacte

En premier lieu, il convient de préciser qu’une méthode classique pour affiner la précision des calculs, consistant à augmenter le nombre de décimales utilisées pour représenter les nombres flottants (précision quadruple-double, par exemple, dans [HLB01]), ne serait pas ici une solution. En effet, une telle méthode servirait juste à réduire le nombre d’erreurs, sans

garantir leur absence. Il nous faut donc chercher d'autres méthodes de calcul permettant une évaluation exacte du signe de nos déterminants 2D, c'est-à-dire garantissant la certitude sur le signe de l'expression donnée par la formule 3.1 (en considérant la nullité comme un signe à part entière). L'arithmétique des intervalles [BBP98, Hay03] pourrait sans doute nous permettre de résoudre ce problème. Il est également possible d'effectuer les calculs en arithmétique exacte, en reportant les approximations effectuées au fur et à mesure des calculs pour arriver à la valeur exacte de l'expression considérée. La librairie CGAL [cga] fournit une implémentation C++ efficace de ces algorithmes. Nous avons pour notre part choisi d'utiliser les algorithmes adaptatifs de Jonathan Shewchuk [She96], qui permettent de bénéficier de la précision des calculs en arithmétique exacte, mais de ne faire appel à ces méthodes coûteuses que pour les cas où le signe de l'expression étudiée reste incertain.

Concrètement, supposons que l'on veuille connaître le signe exact d'une expression comme celle de la formule 3.1. Tout d'abord, cette expression va être calculée de manière classique, via l'arithmétique flottante. Shewchuk fournit alors un estimateur A calculé à l'exécution, dépendant de la machine utilisée et de la précision demandée des calculs. En comparant la valeur absolue de l'expression calculée à cet estimateur A, il est alors possible de savoir si le signe de l'expression peut être déterminé avec certitude ou non. Si le doute persiste, le calcul de l'expression est affiné, en prenant mieux en compte les différentes approximations effectuées par l'arithmétique flottante. Il existe ainsi, pour le cas de l'évaluation exacte du signe d'un déterminant 2D, trois niveaux de raffinement possibles (le nombre de ces niveaux dépend de l'expression étudiée), qui permettent de calculer au fur et à mesure une valeur de plus en plus proche de la valeur exacte de l'expression. Le quatrième niveau de calcul consiste en un calcul de l'expression via l'arithmétique exacte, où une réponse finale quant au signe de l'expression étudiée est toujours donnée. Ces quatre niveaux de précision permettent de n'effectuer les calculs lourds dûs à l'arithmétique exacte (approximativement 10 fois plus longs qu'en arithmétique flottante lorsqu'on pousse les calculs à la quatrième étape) que pour un nombre très faible de cas. En effet, le premier estimateur A est déjà très sélectif et permet d'écartier un très grand nombre de cas. Dans la plupart des situations, il n'y a donc pas besoin de pousser la précision des calculs plus loin que la précision normale des calculs en arithmétique flottante.

Pour notre cas précis, le problème de la formule 3.1 a été explicitement résolu par Shewchuk (une légère amélioration a récemment été présentée [OOR09]), le déterminant 2D étant nommé dans ses travaux *orient2d*. Shewchuk présente donc une solution extrêmement efficace pour résoudre nos problèmes de précision, mais qui n'est hélas pas adaptée au fonctionnement du GPU. Nous expliquons dans la suite de ce chapitre pourquoi cette solution n'est pas adaptée au GPU, puis nous proposons une solution afin de résoudre ce problème.

3.2 Résolution des problèmes de précision sur GPU

3.2.1 Difficultés à porter les algorithmes adaptatifs de Shewchuk sur GPU

En premier lieu, il convient de préciser que les difficultés rencontrées pour porter les algorithmes de Shewchuk sur GPU n'ont aucun rapport avec un support seulement partiel de la norme IEEE-754 garanti par les GPUs actuels. En effet, pour les opérations simples, telles l'addition et la multiplication, il est possible, pour les cartes de la génération précédente (comme la NVIDIA GTX 295), d'imposer au GPU le respect de la norme IEEE (via l'utilisation de `_fadd` et `_fmul` par exemple, voir à ce sujet le *Programming guide CUDA* [NVI09]). Pour les cartes plus récentes, comme la NVIDIA GTX 470 que nous utilisons, le respect de

la norme IEEE est garanti pour ces opérations basiques. Or, les prédictats de Shewchuk font uniquement appel à ce genre d'opérations simples (dans le cas du prédictat *orient2d* en tout cas).

Le problème posé par les algorithmes de Shewchuk sur GPU est plutôt lié à la nature adaptative de ces algorithmes : en effet, dans l'approche de Shewchuk, les fonctions successivement appelées pour affiner les calculs sont de plus en plus lourdes en termes de calculs et peuvent donc, dans l'optique d'une implantation GPU où chaque thread serait en charge d'un test d'intersection, provoquer de forts problèmes d'équilibre de charge. Au sein d'un même warp, des threads peuvent diverger et provoquer un net ralentissement, mais à plus haut niveau, des blocs de threads peuvent également demander des temps de calculs très différents et provoquer un grand déséquilibre dans la répartition de la charge de calculs. De plus, dans le cas où les quatre étapes successives de raffinement des calculs doivent être effectuées, l'espace mémoire nécessaire à l'exécution du code peut devenir très important. Chaque bloc de threads risque donc d'avoir besoin d'une quantité de mémoire importante pour exécuter ce code. Le nombre de blocs exécutables simultanément par multiprocesseur risque donc d'être minimal, entraînant une occupation (*occupancy*) très faible. Il nous faut donc trouver comment contourner ces difficultés afin de porter les algorithmes de Shewchuk sur GPU.

3.2.2 Solution proposée

La solution que nous proposons pour ce portage est en réalité très simple et très intuitive. Comme dit plus haut, pour obtenir un résultat certain au prédictat *orient2d*, l'algorithme consiste à calculer un estimateur A, et, si besoin est, des estimateurs B, C puis D. Notre approche consiste simplement à ne procéder qu'à la première étape de l'algorithme de Shewchuk (calcul de A) sur GPU : si le résultat est incertain, le test d'intersection rayon-triangle correspondant est noté comme douteux. Une fois tous les rayons traités, ces tests d'intersection douteux sont réexécutés sur CPU. Les résultats de ces tests sont ensuite remontés au GPU, pour pouvoir réaliser les étapes [2.3.8](#) et [2.3.9](#) de l'algorithme de lancer de triangles présenté dans le chapitre précédent. La réussite de cette stratégie dépend de la sélectivité du premier critère du test de Shewchuk. Si, pour la plupart des cas, le premier niveau de précision suffit pour connaître le signe du déterminant 2D, alors les performances de l'algorithme global ne devraient être que faiblement dégradées. Or, nous verrons dans la partie [3.5](#) que ce test est extrêmement sélectif et nous permet donc de minimiser grandement la proportion de tests effectués sur CPU.

Il est néanmoins important de détailler l'exécution de cette phase où les tests d'intersections sont réexécutés sur CPU. En effet, afin de connaître les tests à effectuer sur CPU, il faut transférer au CPU l'identité des couples rayon/triangle pour lesquels les tests doivent être réexécutés. Il convient alors d'effectuer sur CPU le même test d'intersection 2D que celui ayant mené à un doute sur GPU. Afin d'éviter de transférer les coordonnées 2D des trois sommets de chacun de ces triangles, il pourrait paraître intéressant de recalculer sur CPU les coordonnées 2D des triangles concernés. Il faudrait alors, en premier lieu, avoir accès à l'identité du triangle 3D ayant donné naissance à ce triangle 2D (comme on le verra dans la partie [3.4](#), un même triangle 3D peut en effet, au cours de l'étape de “clipping”, être subdivisé en plusieurs nouveaux triangles 3D, afin de garantir des projections correctes en 2D). Une première difficulté serait déjà, à partir de ce triangle 3D disponible sur CPU, de retrouver lequel de ses fils (après l'étape de clipping) a donné naissance au triangle 2D concerné.

En dehors de ce premier problème, nous nous heurtons surtout, avec une telle solution, à des problèmes de précision. En effet, la projection 2D des triangles passe par l'exécution de

plusieurs divisions, pour lesquelles le respect de la norme IEEE sur GPU n'est pas garanti. Dans le cas où nous garantissons que nous ne manquons pas d'intersections avec un objet, les approximations engendrées ne posent pas de problème majeur, puisque, comme nous l'avons vu, nous travaillons déjà avec des objets légèrement différents de ceux rencontrés dans la réalité. En revanche, les divisions sur le CPU étant effectuées suivant cette norme IEEE-754, nous allons avoir des triangles 2D numériquement différents, selon que l'on considère leur version CPU ou GPU. Ce problème peut être sensible, puisque les triangles en question sont ceux pour lesquels le signe d'un déterminant 2D associé à une arête est incertain. En modifiant numériquement les coordonnées du triangle, même très légèrement, on risque donc souvent de modifier le résultat "exact" attendu et ainsi de générer des résultats incohérents entre les différents triangles. Il est important de noter que nous avons effectué des tests pour vérifier si ce problème pouvait se produire, et nous avons en effet rencontré de nombreux cas pour lesquels les résultats de tests d'intersections pour des triangles 2D censés être identiques différaient. Nous avons donc finalement décidé de transférer les coordonnées 2D des triangles du GPU vers le CPU, malgré les transferts mémoire occasionnés (rappelons néanmoins que seuls les triangles correspondant à des tests "douteux" doivent être transférés, ce qui en réalité se produit très rarement, comme nous le verrons dans la partie 3.5).

3.2.3 Tests d'intersection 3D

Néanmoins, il existe d'autres possibilités, moins naturelles, pour éviter ces transferts de mémoire. Une première idée consisterait à repérer les tests d'intersection rayon-triangle douteux, puis à effectuer sur CPU, pour ces cas précis, non plus le test d'intersection avec le triangle 2D importé depuis le GPU, mais avec le triangle 3D "père" du triangle 2D en question. En effet, comme nous l'avons dit, des tests d'intersection efficaces et cohérents, basés sur le calcul de signes de volumes de certains tétraèdres, existent aussi en 3D [DSD+09]. Or, il est assez facile de transformer ces calculs de volumes en un cas particulier du prédictat *orient3d* proposé par Shewchuk et donc de garantir, là aussi, l'obtention de résultats exacts pour ces tests.

Cette première idée mène cependant à des résultats incohérents, là aussi pour des problèmes liés aux approximations effectuées par l'arithmétique flottante. En effet, les triangles 2D résultent de certaines approximations, inhérentes aux calculs en virgule flottante, effectuées lors des calculs utilisant les coordonnées 3D des triangles. Les tests 2D et 3D pour un même triangle peuvent donc amener des résultats différents (à noter que cette difficulté n'est ici aucunement liée au GPU et apparaît également dans une implémentation purement CPU.) Les maillages 2D et 3D de la scène ne sont donc pas cohérents entre eux, et l'on peut alors parfois, avec une telle implémentation, "rater" l'intersection entre un rayon et un triangle. Nous avons cependant implanté cette stratégie, afin de calculer la fréquence d'apparition de telles erreurs : comme nous le verrons dans la partie 3.5, ce genre de cas est très rare sur des cas réels, mais apparaît néanmoins. Dans notre contexte d'utilisation, nous ne pouvons accepter le moindre comportement de ce type, et nous avons donc dû rejeter cette solution.

La dernière possibilité s'offrant à nous consiste alors à faire directement les tests d'intersection en 3D sur GPU. Comme la sélection des triangles à intersecer (phase de rastérisation présentée dans le chapitre précédent) repose toujours sur leur projection 2D, on pourrait se demander si le problème précédent ne risque pas de se présenter à nouveau, à savoir manquer les tests d'intersection 3D à effectuer. Néanmoins, lors de notre phase de rastérisation, nous "gonflons" un peu les triangles 2D, afin de ne surtout pas rater de rayons potentiellement intersectés. Nous verrons (partie 3.5) que cette précaution nous permet de garantir la cohé-

rence de nos calculs, mais que cette stratégie s'avérera moins efficace que celle consistant à effectuer les tests d'intersection en 2D.

3.3 Utilisation de la topologie du maillage

3.3.1 Principes

Maintenant que nous sommes capables d'effectuer des calculs de signes de déterminant 2D (ou 3D) fiables sur GPU, nous pouvons savoir si un rayon intersecte ou non un triangle, mais aussi distinguer les cas où ce rayon intersecte le triangle sur une arête, un sommet (intersection avec deux arêtes) ou à l'intérieur du triangle. Dans le cas où une intersection se produit sur l'arête d'un triangle, par exemple, cette intersection sera donc comptée deux fois, dans chacun des triangles se partageant l'arête. Afin de correctement calculer les distances parcourues par un rayon à l'intérieur des objets de la scène, nous avons besoin de ne compter qu'une seule intersection pour un tel cas. Une solution usuelle est alors de comparer les intersections trouvées le long d'un rayon et de fusionner celles qui sont trop proches, à un "epsilon" près. Cette stratégie nous force cependant à définir un "epsilon" dépendant de la scène, alors que nous souhaitons absolument garantir le calcul du débit de dose sans l'intervention d'un utilisateur externe qui devrait régler la valeur de ce genre de facteurs dépendants de la scène.

Afin de résoudre ce problème, nous allons donc employer une technique mathématiquement robuste, consistant à exploiter les informations dont nous disposons sur la topologie du maillage. Comme nous ne travaillons qu'avec des maillages propres (donc fermés), nous avons la garantie qu'une arête appartient toujours à deux et seulement deux triangles et que, par conséquent, toute intersection avec une arête doit être associée à une intersection avec une autre arête. De la même manière, une intersection sur un sommet partagé par n triangles doit apparaître n fois dans la liste des intersections.

Après calcul de toutes les intersections entre les rayons et les triangles de la scène, il convient donc de s'assurer que ces propriétés sont vérifiées par les intersections trouvées. Cette étape étant difficilement parallélisable, nous ne l'avons implémentée que sur CPU. Néanmoins, comme ce travail de vérification n'est nécessaire que pour les intersections sur une arête ou un sommet, et que ces intersections sont très rares dans une scène classique, cela n'entraîne pas de dégradation importante des performances.

3.3.2 Implémentation

L'implémentation que nous avons réalisée pour mettre en place ce traitement est assez simple et pourrait sans doute être optimisée. Elle consiste simplement à réaliser une boucle sur l'ensemble des intersections ayant lieu sur une frontière (arête ou sommet) d'un triangle. Le fait de déterminer quelles sont ces intersections est un problème intrinsèquement parallélisable et est donc effectué en amont sur GPU. Les coordonnées de ces intersections sont ensuite transmises au CPU, qui analyse les données associées. Au cours de cette boucle effectuée sur CPU, il faut ajouter chacune de ces données à la liste des intersections sur des arêtes/sommets pour le rayon en cours. Deux objets sont à considérer pour répertorier ces intersections :

1. un objet *ListeArêtes* (de type `std::map`), listant le nombre d'apparitions de chaque arête, chaque arête étant représentée par un couple (A, B) , où A et B sont les indices des sommets constituant l'arête (à noter que nous utilisons toujours la norme $A < B$ pour stocker l'arête, afin d'assurer que la même arête est toujours stockée de la même

manière, quel que soit le triangle d'appartenance). Pour toute nouvelle intersection trouvée avec une arête (A, B) , nous incrémentons le compteur associé $ListeArêtes(A, B)$ d'une unité.

2. un objet *ListeSommets* (de type std::map également), plus complexe : cet objet associe à chaque sommet pour lequel au moins une intersection a été trouvée une std::map *listeArêtesDuSommet*. Afin d'expliquer comment est construite cette map, prenons l'exemple d'un triangle ABC et d'une intersection du rayon traité avec le sommet A . Si la liste *ListeSommets*[A] n'existe pas encore, elle est créée ; nous la nommons donc pour cet exemple *listeArêtesDuSommetA*. Ensuite, nous incrémentons d'une unité *listeArêtesDuSommetA[B]* et *listeArêtesDuSommetA[C]* (après avoir d'abord créé ces entrées si besoin est).

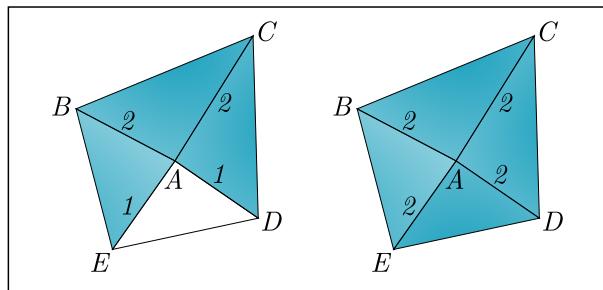


Figure 3.3 – Exemple d'un groupe de quatre triangles se rencontrant en un sommet A . Sur la figure de gauche (trois intersections avec le sommet A), l'intersection n'est pas validée, car les arêtes $[AE]$ et $[AD]$ ne sont associées qu'à une seule intersection. Sur la figure de droite, l'intersection est bien validée, car les quatre intersections identifiées avec le sommet A se transforment en deux intersections pour chaque arête arrivant en ce sommet.

Une fois toutes les intersections correspondant à un même rayon traitées, nous observons ces deux objets. Tout d'abord, pour chaque entrée dans *ListeArêtes*, donc pour chaque groupe d'intersections associées à une même arête, nous vérifions que nous comptons bien deux intersections. Nous devons ensuite vérifier la cohérence des données concernant les intersections avec des sommets de triangles. Le problème est ici plus complexe, puisque nous ne savons pas a priori combien de triangles se rencontrent en chaque sommet. Comment, alors, différencier le cas où un sommet est partagé par trois triangles, pour lequel trois intersections sont trouvées (l'intersection avec ce sommet doit alors être validée), du cas où un sommet est partagé par quatre triangles, et pour lequel trois intersections, là aussi, sont repérées (voir en partie gauche de la figure 3.3 un exemple de ce cas non valide) ? Pour différencier ces cas, nous allons exploiter les informations dont nous disposons sur la topologie du maillage, et vérifier que pour chaque arête arrivant vers le sommet, deux intersections sont trouvées de part et d'autre de l'arête. L'opération décrite plus haut, consistant à incrémenter les deux compteurs *listeArêtesDuSommetA[B]* et *listeArêtesDuSommetA[C]*, revient ainsi à transformer l'intersection avec le sommet A en une intersection avec les deux arêtes $[AB]$ et $[AC]$. Pour le cas représenté sur la gauche de la figure 3.3, trois intersections avec le sommet A (venant des triangles ABE , ABC et ACD) se transforment en deux intersections chacune pour les arêtes $[AB]$ et $[AC]$, mais une seule intersection est trouvée pour les arêtes $[AE]$ et $[AD]$, ce qui invalide l'intersection. Pour le cas représenté sur la droite de la figure, deux intersections sont bien comptées pour chacune des arêtes $[AB]$, $[AC]$, $[AD]$ et $[AE]$, ce qui valide l'intersection.

Nous pouvons donc, à partir de l'étude des deux objets *ListeArêtes* et *ListeSommets*, valider la cohérence des intersections trouvées sur des "frontières" de triangles. Une fois ces deux objets étudiés, nous pouvons les réinitialiser pour l'étude du rayon suivant.

Au final, ce traitement constitue un outil d'auto-évaluation de notre code (en principe, aucun cas "non-valide" ne doit apparaître), mais surtout, il nous permet d'éliminer les intersections en double (les intersections se produisant avec un sommet de triangles apparaissent même plus de deux fois) : nous ne comptons ainsi qu'un seul point d'entrée ou de sortie dans l'objet considéré et garantissons un calcul de débit de dose "cohérent". Notons enfin que cet algorithme permet une utilisation des informations concernant la topologie du maillage "à la volée", sans besoin d'une phase de pré-calcul permettant de générer l'intégralité de ces informations de topologie. Cette propriété est importante pour notre algorithme, car elle nous permet de traiter sans difficulté supplémentaire des scènes subissant des modifications de leur topologie au cours du temps (tant que les propriétés de "propreté" du maillage restent vérifiées).

3.3.3 Cas des rayons passant près de la frontière d'un objet

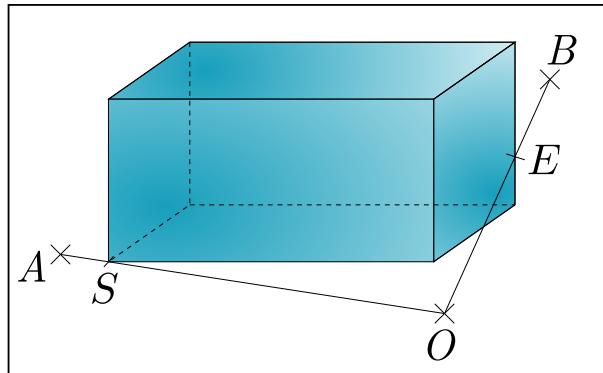


Figure 3.4 – Exemple de deux rayons $[OA]$ et $[OB]$, intersectant respectivement un sommet (point S) et une arête (point E) situés sur la frontière d'un cube triangulé.

Avant de finir cette partie, un cas particulier reste à signaler, se produisant lorsqu'une arête ou un sommet est en réalité sur la frontière d'un objet (voir figure 3.4). Dans ce cas, le rayon rentre et sort immédiatement de l'objet en question, et il faut donc considérer qu'il n'intersecte pas l'objet. Pour traiter ce cas de manière correcte, il nous faut donc également savoir, pour chaque intersection, si cette intersection correspond à une entrée ou à une sortie dans l'objet. Sur une arête, si l'on trouve une intersection entrante et l'autre sortante, il faut considérer qu'il n'y a aucune intersection. Pour un sommet, ce cas se produira s'il y a au moins une intersection de chaque sorte (entrant ou sortante) en ce sommet.

3.3.4 Transfert des informations sur la nature des intersections vers le CPU

Il nous reste un dernier point à aborder concernant le traitement des informations sur la topologie du maillage. En effet, jusque là, nous n'avons pas précisé sous quelle forme sont décrits les cas d'intersection (selon que l'intersection se produit sur une arête, sur un sommet ou à l'intérieur d'un triangle). Nous n'avons pas non plus évoqué le transfert de ces informations du GPU vers le CPU, nécessaire pour l'étape de traitement décrite en 3.3.2,

et devant forcément induire un surcoût par rapport au traitement classique de l'algorithme, sans gestion des problèmes de précision provoqués par l'arithmétique flottante.

Afin de distinguer ces cas d'intersections, au moment du test d'intersection, nous générerons une information sur la nature de l'intersection trouvée, notée $casId$ (0 pour une intersection à l'intérieur du triangle, de 1 à 3 pour les intersections avec un sommet de triangle, 1 représentant le sommet A , et de 4 à 6 avec une arête, 4 représentant l'arête $[AB]$). Une solution naturelle serait d'utiliser un caractère *char* pour modéliser cette information, puis de transférer ces données du GPU vers le CPU après les calculs d'intersections. Cependant, il existe une manière d'éviter ces transferts, via une astuce similaire à celle présentée dans le chapitre précédent (voir [2.3.9](#)). Nous avions alors souhaité représenter les deux informations $rayId$ et t , respectivement indices d'un rayon intersecté et profondeur d'intersection, par une seule valeur $val = 4 * rayId + t + 1$. Nous avions observé à cette occasion que les approximations dues à l'arithmétique flottante empêchaient une utilisation correcte de cette expression pour réaliser l'étape de tri des intersections. Pour le cas des données $casId$, le problème n'est plus le même, puisque ces données prennent des valeurs entières. Ainsi, nous utilisons cette propriété pour représenter conjointement les indices de triangles intersectés $trgId$ et les données $casId$, via la formule $val = 8 * trgId + casId$. La valeur $casId$ ne pouvant varier qu'entre 0 et 6, cette représentation implique que les données $trgId$ et $casId$ peuvent être retrouvées sans ambiguïté à partir d'une valeur de val .

Cette astuce nous permet d'économiser la consommation mémoire nécessaire au stockage des données $casId$ sur GPU, mais aussi le transfert de ces données du GPU vers le CPU. De plus, nous économisons également la réorganisation de ces données en fonction du tri effectué sur les coordonnées d'intersections. En effet, lorsque nous trions les couples $(rayId, t)$ lors de l'étape [2.3.9](#) de l'algorithme de lancer de triangles, nous modifions naturellement les positions de ces couples en mémoire. Il nous faut donc ensuite répercuter ces mouvements dans les vecteurs de données $casId$ et $trgId$, afin qu'un couple $(rayId, t)$ situé en position i en mémoire reste bien associé à des données $casId$ et $trgId$ situées en position i elles aussi. Grâce à cette astuce, nous ne devons plus réordonner les indices de ces deux vecteurs, mais seulement ceux de celui qui répertorie les données val . En outre, le “décryptage” d'une donnée val est extrêmement rapide à réaliser, même sur CPU, et le gain de temps produit par cette astuce sur le temps d'exécution de l'algorithme global est donc réel (sur une scène comme la scène NuclearCase, pour 5.7 millions d'intersections trouvées, nous économisons ainsi près de 4 millisecondes).

3.4 Problèmes posés par la projection des triangles

Nous venons d'expliquer comment utiliser les informations sur la topologie du maillage afin d'exploiter les résultats des tests d'intersection “exacts” permis par les algorithmes de Shewchuk. Nous avons néanmoins pour le moment ignoré une difficulté importante associée au travail dans un espace en perspective : la projection des triangles dans un espace à deux dimensions impose un travail sur les triangles pouvant amener à une modification de la topologie du maillage. Si ce travail n'est pas fait de manière attentive, des problèmes de cohérence peuvent apparaître entre différents triangles du maillage (par exemple, une arête n'appartenant qu'à un seul triangle), rendant inefficaces les stratégies présentées dans les paragraphes précédents. Nous détaillons donc dans cette partie comment nous gérons la projection des triangles, tout en garantissant une cohérence globale sur la topologie du maillage, nous permettant d'assurer le bon fonctionnement des traitements conjugués présentés dans les parties [3.2](#) et [3.3](#).

3.4.1 Le clipping

Pour projeter un triangle dans un espace à deux dimensions, il nous faut projeter chacun de ses sommets dans cet espace. Si nous considérons que Z est la profondeur suivant notre point de vue, la projection d'un point (X, Y, Z) en 2D est le point de coordonnées :

$$(x, y) = (X/Z, Y/Z) \quad (3.2)$$

Si deux points A et B ont des coordonnées en Z positives, alors la formule du déterminant 2D donnée par 3.1, évaluée à partir des coordonnées 2D données par la formule 3.2, est correcte. Si les deux points sont situés de part et d'autre du plan $Z = 0$, cette formule n'est plus vraie. Des solutions existent afin de calculer un déterminant 2D en n'utilisant que les coordonnées 3D des points, via l'utilisation de coordonnées 2D homogènes [OG97]. Mais elles impliquent de devoir inverser une matrice dépendant des positions des trois sommets. Le déterminant de cette matrice diminuant lorsque les points du triangle se rapprochent, des problèmes de précision peuvent apparaître pour de "petits" triangles, lors du calcul de la matrice inverse et donc des coefficients des déterminants 2D associés. Dans notre cas, nous ne pouvons ignorer ces "petits" triangles (dans le cas graphique, un petit triangle peut tout simplement ne pas être affiché). Cette difficulté nous empêche donc d'avoir recours aux coordonnées homogènes, pourtant très utiles pour de nombreuses applications de la communauté graphique.

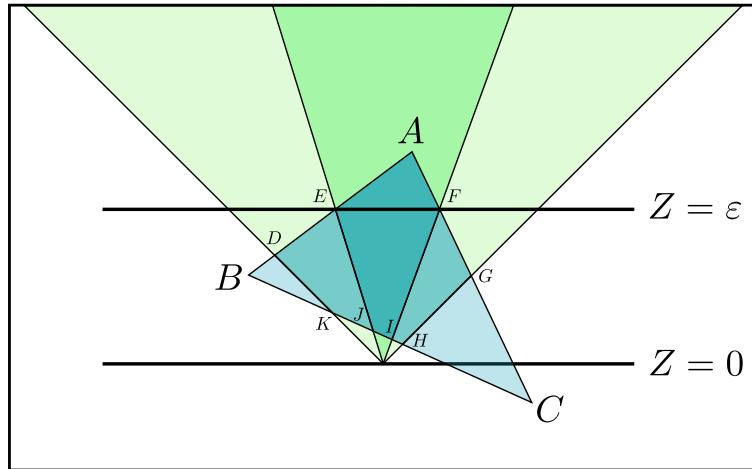


Figure 3.5 – Exemple de mauvais traitement observé lors de la découpe d'un triangle suivant le plan $Z = \epsilon$ (vue en deux dimensions). La découpe suivant ce plan va entraîner la création d'un unique triangle AEG et évincer les deux quadrilatères $DEJK$ et $FJIH$, qui intersectent pourtant le cône de vision. La projection 2D du triangle AEG ne sera donc pas suffisante pour décrire l'ensemble du triangle ABC en deux dimensions.

Comme nous ne devons travailler qu'avec des triangles ayant leurs sommets dans le demi-espace $Z > 0$, il nous faut donc procéder à une étape de "clipping" pour les triangles "à cheval" sur le plan $Z = 0$. Le "clipping" consiste ici à découper chacun de ces triangles en plusieurs morceaux, afin de ne conserver parmi ces nouveaux triangles que ceux situés intégralement dans le demi-espace $Z > 0$. Une première stratégie naturelle pourrait consister, pour cette étape de clipping, à découper les triangles au niveau d'un plan $Z = epsilon$, avec $epsilon$ "très petit". Une telle stratégie n'est cependant pas acceptable dans notre cas, car elle peut

créer des problèmes de précision au moment de la projection 2D (pour certains sommets de triangles vérifiant la propriété $Z = \text{epsilon}$, les divisions X/Z et Y/Z permettant le calcul des coordonnées 2D sont fortement sujettes à approximations). De plus, avec une telle stratégie, certains triangles passant près du centre de l'espace en perspective se retrouvent tronqués d'une partie qui doit pourtant être référencée par la grille en perspective (cf. figure 3.5 pour l'exemple d'un tel cas).

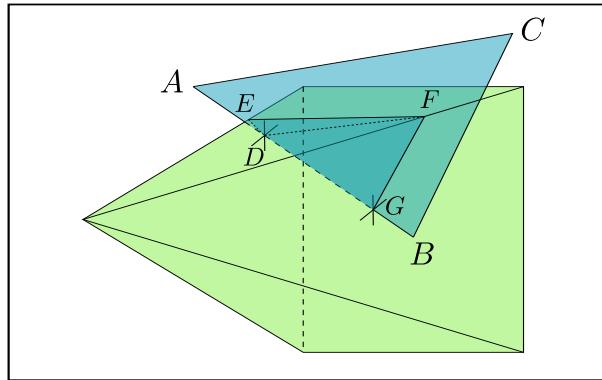


Figure 3.6 – Découpe d'un triangle suivant les quatre plans du cône de vision. Le triangle ABC , après calcul d'intersection avec le cône de vision, est remplacé par les deux triangles DEF et DFG .

De manière à éviter ces désagréments, nous procédons donc à un clipping avec les faces du cône de vision défini par la grille en perspective (cf figure 3.6). Pour procéder à ce clipping, nous avons choisi d'utiliser l'algorithme de Sutherland-Hodgman [SH74], qui consiste à calculer l'intersection du triangle avec les 4 faces du cône, les unes après les autres (voir figure 3.7 pour un exemple simple). Notons au passage qu'il est important, afin de garantir la cohérence du clipping entre les différents triangles, de considérer ces 4 faces suivant le même ordre pour l'ensemble des triangles.

3.4.2 Changement de topologie

Cette étape de clipping n'est hélas pas sans conséquences sur la topologie du maillage des triangles de la scène. En effet, la subdivision d'un triangle implique la création de nouveaux points dans le maillage. Lorsqu'une intersection avec l'arête d'un des nouveaux triangles est trouvée, les indices des points correspondants ne sont ainsi pas les mêmes que ceux du maillage original. Il faut donc, lorsque l'on ajoute un point au moment de la phase de clipping d'un triangle, garder la mémoire du sommet d'origine correspondant, c'est-à-dire le sommet dont part l'arête qui doit être coupée.

Cependant, cette précaution n'est pas suffisante, car l'étape de clipping peut aussi engendrer des problèmes de précision au cours des calculs d'intersections : en effet, en ne “clippant” que certains triangles (les triangles ayant un ou deux points dans le demi-espace $Z < 0$), on coupe certaines arêtes qui appartiennent à un triangle à découper, mais aussi à un triangle qui ne doit pas être modifié. Un tel cas peut être observé sur la figure 3.8. Le triangle ABC , après clipping, est réduit au triangle BEC . Le triangle BCD , ayant tous ses points dans le demi-espace $Z > 0$, n'a pas à être modifié. Au moment de tester de quel côté de l'arête $[BC]$ passe un rayon, on va, dans le cas du triangle ABC , considérer le segment $[BE]$, et dans le cas du triangle BCD , considérer le segment $[BC]$. Une fois de plus, à cause des approximations faites par la machine, le segment $[BE]$ n'est pas équivalent au segment $[BC]$. Pour avoir un

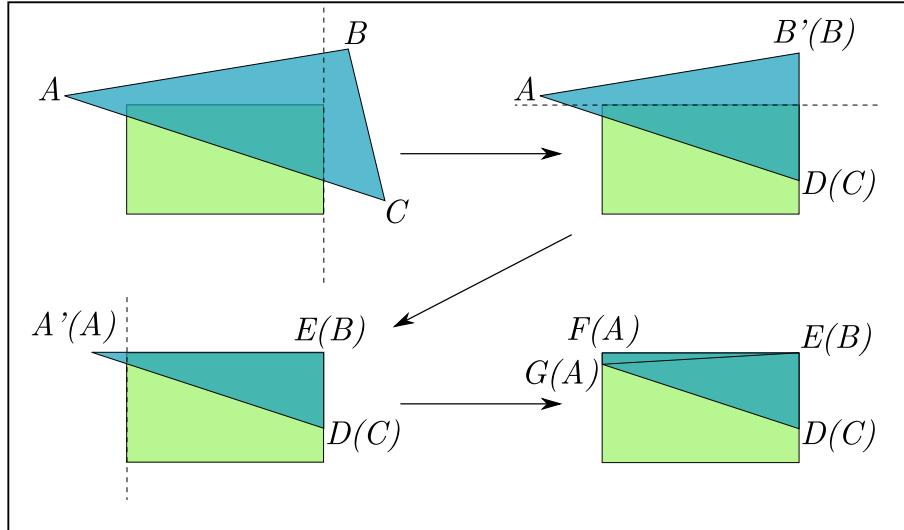


Figure 3.7 – Exemple de découpe d'un triangle suivant l'algorithme de Sutherland-Hodgman (vue du point de départ du cône de vision, centre de projection de la scène). Le triangle ABC est successivement découpé en trois polygones $AB'D$, $A'ED$ et $FEDG$. Ce dernier polygone donne naissance à deux triangles du nouveau maillage, nommés DEG et EFG . Entre parenthèses sont indiqués les pères des points nouvellement créés.

test d'intersection cohérent des deux côtés de l'arête, il faut alors être capable de garder en mémoire le fait que le point E vient du point C , qui est un point situé dans le demi-plan $Z > 0$, et faire en sorte qu'au moment du calcul d'intersection entre un rayon et le triangle ABC , ce soit en fait le segment $[BC]$ qui soit pris en compte, et non le segment $[BE]$. Il faut malgré tout absolument conserver les coordonnées du point E , car elles serviront au traitement de l'arête $[EF]$ qui, elle, n'a pas d'équivalent dans le maillage initial. De nombreux cas du même genre peuvent être générés, et de manière générale, pour assurer la cohérence des calculs, il faut pouvoir détecter si une arête d'un triangle correspond à la portion d'une arête initialement présente dans le maillage, et si cette arête initiale est intégralement présente dans le demi-espace $Z > 0$. Si tel est le cas, il faut alors utiliser les coordonnées des points "parents" (comme C , qui est le "père" de E , dans notre exemple) pour le calcul du déterminant 2D associé à une arête.

Une telle solution, couplée à l'exploitation des informations concernant la topologie du maillage, permet d'obtenir des résultats cohérents sur CPU. Cependant, un tel traitement n'est pas adapté au fonctionnement du GPU. Pour pouvoir tirer partie de la puissance des processeurs graphiques, il nous faut donc modifier cette étape de clipping.

3.4.3 Clipping sur GPU

En effet, ce type de traitement peut nécessiter des calculs assez complexes afin d'identifier si une arête d'un nouveau triangle est en réalité la portion d'une arête initialement présente. Ce genre de calculs introduit un grand nombre de branchements très inadaptés au fonctionnement du GPU. Ces calculs étant communs à un triangle quel que soit le rayon, il peut être intéressant de les réaliser dans une phase de "pré-calcul" antérieure à la phase de réalisation des tests d'intersections. Il faut alors stocker pour chaque triangle une information sur sa structure (si telle arête parmi les trois qui le composent est la portion d'une arête du

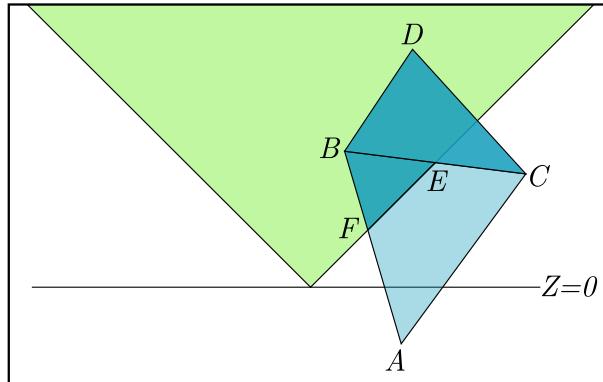


Figure 3.8 – Problèmes de précision posés par l'étape de clipping. Le triangle BCD , situé intégralement dans le demi-espace $Z > 0$, n'est pas modifié. L'arête $[BC]$ est donc transformé en deux arêtes $[BC]$ et $[BE]$, numériquement distinctes, ce qui entraîne des problèmes de cohérence lors des tests d'intersections.

maillage initial ou non) et, au moment du test, procéder à un branchement en fonction de ces informations.

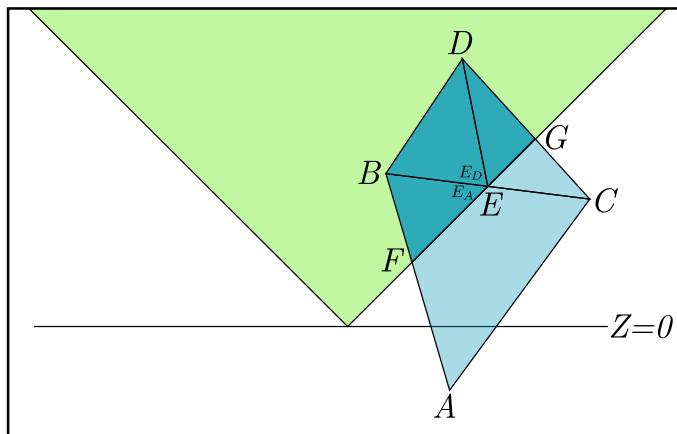


Figure 3.9 – Résolution des problèmes de cohérence posés par le clipping, via un clipping “integral” des triangles de la scène. L'arête $[BC]$ est ici transformée en l'arête $[BE]$ pour chaque triangle du maillage initial. Les points E_A et E_D sont deux instances du même point E , respectivement issues du découpage des triangles ABC et BCD .

Afin de garantir un fonctionnement bien mieux adapté à la structure hautement parallèle du GPU, nous effectuons le clipping de tous les triangles avec le cône de vision associé à la grille en perspective, même de ceux déjà initialement situés dans le demi-espace $Z > 0$. Cette solution est inefficace en termes de performance sur CPU, mais la puissance de calcul du GPU permet de ne pas perdre beaucoup de temps dans cette phase de clipping, chaque triangle pouvant être traité par un thread de manière assez efficace. Le problème présenté sur la figure 3.8 disparaît alors (voir Figure 3.9). En effet, deux triangles contigus sont tous les deux découpés, rendant les calculs identiques des deux côtés de l'arête. Il faut néanmoins veiller, là aussi, à utiliser les mêmes calculs pour la génération du point d'intersection E dans les deux triangles. Concrètement, le point O désignant le centre du repère, les coordonnées du point d'intersection E entre le segment $[BC]$ et la face de normale N sont obtenues via la formule :

$$\overrightarrow{OE} = \overrightarrow{OB} \cdot \vec{N} / (\overrightarrow{OB} \cdot \vec{N} - \overrightarrow{OC} \cdot \vec{N})$$

Afin de respecter la cohérence entre les triangles, nous avons choisi de considérer dans cette formule le point B comme celui de l'arête situé en dehors du cône de vision, le point C figurant à l'intérieur de ce même cône. Ces précautions nous permettent de garantir la cohérence des tests d'intersections 2D effectués. Cependant, le clipping induit une modification de la topologie du maillage qui influe aussi sur la phase de traitement des intersections décrite dans la partie 3.3.

3.4.4 Prise en compte de la topologie du maillage initial

En effet, si ce clipping “integral” permet de garantir, vis-à-vis des tests d’intersections, la cohérence du nouveau maillage 2D créé, il fait néanmoins perdre de vue la topologie initiale du maillage 3D de la scène. Afin de bien comprendre les raisons de cette affirmation, il nous faut nous pencher plus précisément sur l’implémentation de la phase de clipping. Revenons sur le cas de la figure 3.9 et sur la découpe des deux triangles ABC et BCD . Concentrons-nous sur le cas du point E , nouveau point du maillage, situé sur l’arête BC , commune aux deux triangles étudiés. Pour avoir un maillage “propre”, il faudrait éviter de référencer deux fois ce point E dans le nouveau maillage. Si la découpe des deux triangles est effectuée de manière indépendante, le point E est en effet créé deux fois, une pour chaque triangle. La conséquence de cette duplication est double : premièrement, elle entraîne une surconsommation de l’espace mémoire, le même point, avec les mêmes coordonnées, étant référencé deux fois dans le nouveau maillage. Deuxièmement, cette duplication entraîne une perte d’information sur la topologie du maillage, rendant inefficaces les méthodes de vérification présentées dans la partie 3.3. À titre d’exemple, nommons E_A et E_D les deux points créés à la position du point E , respectivement issus du traitement des triangles ABC et BCD . Considérons maintenant une intersection entre un rayon et l’arête $[BC]$, cette intersection ayant lieu entre les points B et E du nouveau maillage. Pour le cas du triangle ABC , une intersection va donc être trouvée entre le rayon et l’arête $[BE_A]$. Pour BCD , c’est avec l’arête $[BE_D]$ que l’intersection va être détectée. Si l’on n’y prend pas garde, ces deux intersections vont être associées à deux arêtes différentes et donc comptabilisées comme deux intersections différentes. La duplication du point E lors de l’étape de clipping n’est donc pas sans conséquence sur la robustesse de nos algorithmes.

Afin de contourner ces difficultés, la première solution, évidente, consisterait à ne pas dupliquer ce point E . Dans une implémentation CPU (sérialisée), il est assez facile d’éviter cette duplication. Il suffit de construire au fur et à mesure du clipping une liste de nouveaux points créés (et de leurs points parents) et de vérifier, pour chaque nouveau point à créer, s’il n’existe pas déjà dans la liste. Nous avons mis en place cette solution sur CPU, obtenu ainsi un maillage “propre”, et garanti des algorithmes robustes. Cependant, cette stratégie est associée à une légère perte de performance et, surtout, n’est pas du tout adaptée à une implémentation GPU. Comme nous voulons implémenter l’intégralité de notre algorithme sur GPU, pour éviter des temps de transfert extrêmement pénalisants, nous avons donc choisi de mettre en place une solution beaucoup plus grossière sur GPU, consistant à faire effectuer le traitement de chaque triangle par un thread indépendant, sans communication d’information entre les différents triangles. Nous devons donc faire face aux deux problèmes évoqués ci-dessus, à savoir la surconsommation de mémoire (qui reste cependant très raisonnable) et le changement de topologie. Cette seconde difficulté peut en réalité être contournée assez facilement, moyennant une perte de temps minime (mais, une fois de plus, une légère sur-

consommation de mémoire) : au moment de la création d'un nouveau point, nous gardons en effet en mémoire l'identité du père du point nouvellement créé (sur la figure 3.7, par exemple, le point A est le père de A'). Si ce nouveau point a d'autres fils, on choisira comme identité du père de chacun de ces descendants le père du premier point créé, pouvant être vu comme le "plus vieil ancêtre dans le maillage initial" (toujours sur la figure 3.7, le point G n'a pas comme père A' , mais A , point du maillage initial).

Comme on l'a vu, au moment des tests d'intersection, les tests douteux sont réeffectués sur le CPU (via les algorithmes de Shewchuk), puis retransférés sur le GPU pour la fin des traitements. Si, après le traitement sur CPU, une intersection sur un sommet ou une arête de triangle est détectée, nous repérons les sommets du triangle, puis nous retrouvons les identités des "pères" correspondants dans le triangle père. Ainsi, toujours dans notre exemple de la figure 3.9, les deux intersections avec les arêtes $[BE_A]$ et $[BED]$, dans les triangles $BE_A F$ et BDE_D , seront finalement vues comme deux intersections avec les triangles BCD et ABC , sur l'arête $[BC]$ pour les deux cas, C étant le père de E_A et E_D . La question du bon traitement d'une intersection sur l'arête $[DE]$ peut alors se poser. En effet, deux intersections, avec les triangles BDE_D et DGE_D , sont alors trouvées, ce qui peut paraître problématique. Cependant, la stratégie que nous venons de présenter amène là aussi une solution robuste : en effet, en appliquant la méthode du sommet "père", ces deux intersections sont au final transformées en intersections avec le triangle BCD , sur l'arête $[CD]$, C étant le père de E_D . Au moment du traitement présenté lors de la partie précédente, ces deux intersections avec la même arête vont donc être fusionnées en une seule et même intersection. Bien que ces intersections ne correspondent donc pas géométriquement à une intersection sur l'arête $[CD]$, leur traitement robuste reste garanti. La dernier danger pourrait alors être de conjuguer un tel cas avec une intersection se produisant réellement avec l'arête $[CD]$, dans le cas d'un triangle très allongé. Dans le cadre de l'arithmétique flottante, pour un triangle DEG très fin, possédant des arêtes $[DG]$ et $[DE]$ presque confondues, il serait en effet possible de cumuler des intersections avec les arêtes $[DE]$ et $[DG]$, nous amenant à un nombre d'intersections final avec l'arête $[CD]$ supérieur à 2 (ce qui ferait donc échouer les méthodes présentées dans la partie 3.9, puisqu'une intersection associée à une arête ne doit être trouvée qu'une fois). Cependant, avec les méthodes d'arithmétique exacte, une telle confusion n'est pas possible. La seule difficulté pourrait être le cas d'arêtes $[DE]$ et $[DG]$ réellement confondues, correspondant à un triangle plat dans notre repère 2D. Ce genre de triangles mérite cependant un traitement à part (que nous n'avons pas encore mis au point), que nous évoquerons en conclusion de ce manuscrit.

Nous avons donc précisé dans cette partie comment traiter correctement les difficultés créées par la projection 2D des triangles et garantir le bon fonctionnement des méthodes présentées en 3.2 et 3.3. Nous pouvons maintenant vérifier concrètement si ces méthodes garantissent des algorithmes de lancer de rayons robustes, et étudier l'impact de ces stratégies sur nos temps de calcul.

3.5 Résultats

3.5.1 Robustesse des calculs

En premier lieu, avant d'étudier la performance de nos algorithmes, il convient évidemment de vérifier que ces derniers nous permettent d'obtenir un code robuste aux problèmes de précision causés par l'arithmétique flottante. Afin de valider la robustesse de nos méthodes, nous avons repris le cas-test décrit en 3.1.1 (4466 rayons partant du centre d'une sphère

maillée, passant tous par une arête ou un sommet de triangle). Plutôt que de faire partir nos 4466 rayons du centre de la sphère, nous les faisons partir d'un point choisi aléatoirement à l'intérieur de la sphère maillée. En faisant varier la position de ce point aléatoire (un million de positions différentes testées), nous lançons donc plus de 4 milliards de rayons passant sur une arête ou un sommet de triangle. Chaque rayon partant de l'intérieur de la sphère et finissant hors de cette même sphère, nous devons garantir que pour chaque rayon est trouvée une et une seule intersection. Si un seul de ces rayons renvoie un nombre d'intersections différent de 1, nous considérons notre algorithme de gestion de la précision comme invalide (rappelons que lors du test présenté en 3.1.1, 107 rayons sur 4466 ne renvoient aucune intersection, contre 823 renvoyant plus d'une intersection). Pour chaque nouveau point de départ des rayons, nous avons généré six grilles en perspective, nous permettant ainsi de couvrir l'ensemble de l'espace autour du centre de projection.

Nous avons pu valider grâce à ce test l'algorithme CPU consistant à restreindre la phase de clipping aux triangles intersectant le plan $Z = 0$. Nous avons également pu valider l'algorithme GPU consistant à découper tous les triangles (et à mettre en place toutes les opérations de gestion des informations sur la topologie du maillage présentées en 3.4.4). Du point de vue des méthodes utilisées pour les tests d'intersections, la solution consistant à réaliser les tests en 2D sur GPU comme pour CPU a été validée. De même, la solution consistant à réaliser les tests en 3D sur GPU puis sur CPU a également été validée. En revanche, comme nous l'avions indiqué en 3.2.3, la stratégie consistant à effectuer les tests en 2D sur le GPU, puis en 3D sur le CPU pour les cas ambigus, n'a pas été validée : pour une cinquantaine de rayons sur nos 4 milliards de départ, aucune intersection n'a été trouvée. Ce résultat démontre bien le manque de cohérence pouvant exister entre les représentations 2D et 3D des triangles de la scène.

3.5.2 Mesures de performance

Scène	Classique3D	Robuste3D	Perte3D	Classique2D	Robuste2D	Perte2D
Erw6	24.9	26.6	6.8%	24.4	25.9	6.3%
Fairy	47.7	62.0	30.0%	46.3	49.3	6.3%
Conf.	58.3	101.6	74.3%	56.5	60.4	6.6%
Sully	90.5	177.6	96.2%	86.6	93.3	7.6%
Nucl.	105.0	198.0	88.4%	100.6	107.3	6.7%

Tableau 3.1 – Compte rendu de l'impact des méthodes de gestion de la précision sur les performances globales de l'algorithme de lancer de rayons (1024 * 1024 rayons primaires, exécution sur GPU). Classique3D et Classique2D représentent les temps de calcul pour les versions classiques de l'algorithme, décrites dans le chapitre 2. Robuste3D et Robuste2D représentent les performances obtenues avec les algorithmes de gestion de la précision que nous avons présentés dans ce chapitre. Les pertes de performance associées sont données dans les colonnes Perte2D et Perte3D. Tous les temps sont donnés en millisecondes.

Maintenant que la robustesse de nos calculs est garantie, il est essentiel d'étudier l'impact de ces algorithmes en termes de performance. Nous avons ici utilisé les mêmes scènes de référence que dans le chapitre 2 et la même configuration géométrique en termes de rayons. Notre test consiste donc à lancer 1024 * 1024 rayons primaires : en lançant des rayons ordinaires, ne devant a priori pas spécialement passer à proximité d'arêtes ou de sommets de triangle, nous souhaitons réaliser un test de performance en situation réelle. Le cas test décrit dans le précédent paragraphe, s'il nous a permis de valider nos algorithmes, ne représente en

effet en aucun cas une situation réelle, puisque le pourcentage de rayons générant des tests d'intersections ambigus est dans la réalité très faible.

Nous avons comparé pour nos tests de performances une exécution “classique” de l'algorithme (donc sans gestion de la précision et avec utilisation de l'arithmétique flottante pour l'intégralité du code) aux algorithmes robustes que nous avons développés. En premier lieu, on remarque que les performances affichées pour le cas *Robuste2D* sont les mêmes que celles présentées dans le chapitre 2. Nous avons en effet préféré, dans ce précédent chapitre, effectuer ces mesures en tenant déjà compte des méthodes de gestion de la précision, de manière à rendre compte du temps réel d'exécution de l'algorithme pour notre contexte de travail.

Avant de poursuivre l'étude de ce tableau, il faut encore noter que les étapes de clipping décrites dans la partie 3.4 sont également présentes dans la version “classique” de l'algorithme. Afin de mesurer la perte de performance induite par nos algorithmes de gestion de la précision, nous devons en revanche prendre en compte le transfert, du GPU vers le CPU, des données nécessaires à la réexécution des tests d'intersections sur CPU, mais aussi le temps d'exécution de ces tests d'intersections sur CPU, puis le transfert des résultats de ces tests vers le GPU. Il faut également ajouter à ces mesures le temps requis pour générer l'identité des “pères” des sommets concernés lorsqu'une intersection se produit sur une arête ou un sommet de triangle (décris dans 3.4.4). Enfin, il nous faut tenir compte de la phase d'exploitation des informations concernant la topologie du maillage, exécutée sur CPU (voir 3.3). Nous ne citons ici que les versions GPU de nos algorithmes (hormis cette phase de traitement, donc, et l'exécution des tests ambigus, tous nos calculs sont effectués sur GPU), bien plus efficaces que leurs équivalents CPU.

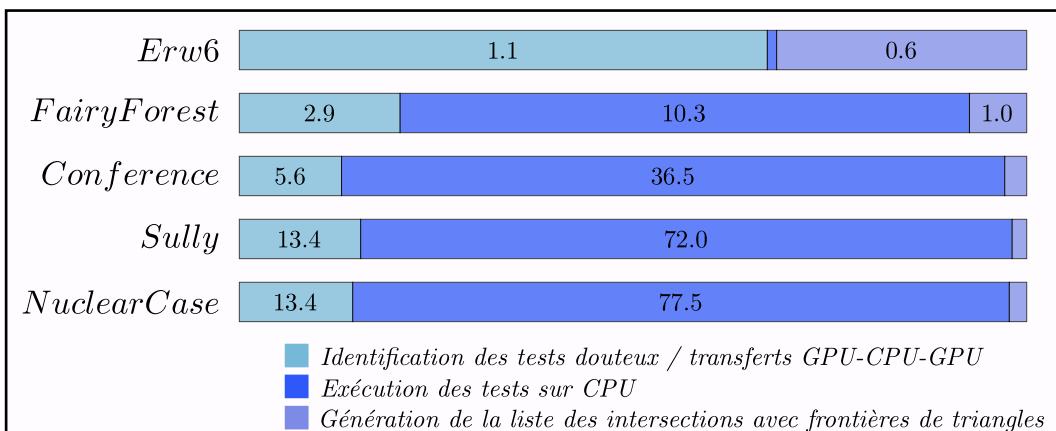


Figure 3.10 – Répartition des temps d'exécution suivant les différentes phases propres à l'algorithme de gestion de précision, lorsque la version 3D des tests d'intersection est utilisée. Les temps sont indiqués en millisecondes.

Les performances obtenues sont décrites dans le tableau 3.1. Pour la version 2D de l'algorithme (tests en 2D sur GPU comme sur CPU), la perte de performance due à la gestion de la précision est minime, toujours inférieure à 10%. Comme ce souci de robustesse est essentiel pour notre contexte de travail, un tel ralentissement est tout à fait acceptable. La version 3D, quant à elle, souffre beaucoup plus de cette stratégie de gestion de la précision. Afin de comprendre la raison d'une telle différence, nous avons mesuré, pour cette version 3D, la répartition des temps d'exécution suivant les différentes phases propres à notre algorithme de gestion de la précision. Les résultats de ces mesures peuvent être observés sur la figure 3.10. Cette figure montre clairement que les performances sont diminuées par le transfert des données du GPU vers le CPU, mais, surtout, par les tests d'intersection exécutés sur

Scène	nbTests	nbAmbiguités2D	nbAmbiguités3D
Erw6	1.3M	0	123
Fairy	3.4M	0	75697
Conference	4.3M	0	197532
Sully	9.1M	8	581269
NuclearCase	9.1M	4	620061

Tableau 3.2 – Comparaison de la précision des versions 2D et 3D des tests d’intersections rayon-triangle. Le nombre de tests pour chaque scène est indiqué dans la colonne nbTests, et les quantités de tests ambigus pour les versions 2D et 3D du test sont respectivement données dans les colonnes nbAmbiguités2D et nbAmbiguités3D.

CPU, suivant l’algorithme adaptatif de Shewchuk. Ce ralentissement est en réalité dû à une proportion de tests d’intersection “douteux” beaucoup plus importants dans la version 3D de l’algorithme que dans son équivalent 2D. Notons au passage que la phase d’exploitation des informations concernant la topologie du maillage (voir 3.3) n’est pas représentée sur cette figure, en raison d’un nombre final d’intersections sur des frontières de triangles nul pour chacune des scènes traitées (le grand nombre de tests “douteux” après la première phase de calculs sur GPU ne signifie pas que les rayons concernés passent réellement par des arêtes ou des sommets de triangles).

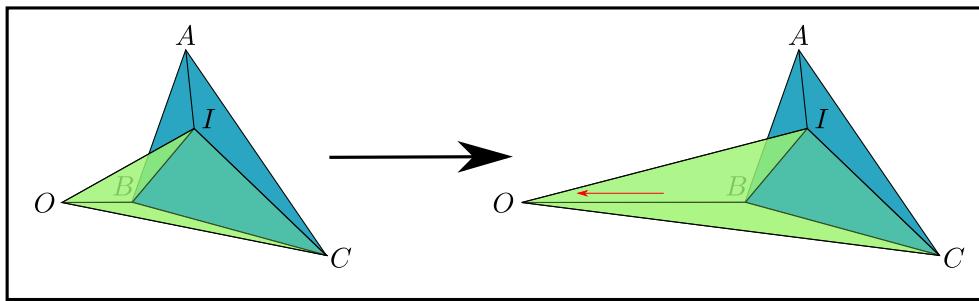


Figure 3.11 – Figure montrant le rétrécissement de la pyramide formée par le point de départ du rayon O , l’arête $[BC]$ du triangle ABC et le point d’intersection I entre le rayon et le plan du triangle (à noter que lors de l’exécution du test, c’est le vecteur directeur du rayon qui est utilisé pour les calculs et non ce point I , ce qui garantit la cohérence des calculs entre des triangles voisins). Lorsque le point O s’éloigne du triangle, le tétraèdre formé rétrécit, et l’étude du signe de son volume devient donc plus difficile.

Afin de valider cette affirmation, nous avons mesuré le nombre de tests d’intersections douteux sur GPU, donc à refaire sur le CPU, pour les versions 2D et 3D de l’algorithme (voir tableau 3.2). Le nombre de tests 3D douteux est considérablement supérieur à son équivalent 2D pour chaque scène de référence étudiée (hormis Erw6, mais pour cette scène, la majorité des tests se fait avec de grands triangles qui délimitent la pièce, pour des rayons passant loin des extrémités de ces triangles, ce qui explique qu’un faible nombre de tests est au final douteux). Deux raisons nous permettent d’expliquer cette différence de précision entre les tests 2D et 3D. Premièrement, le test d’intersection 3D met en jeu des calculs de volumes en 3D, plus complexes que ceux effectués lors de l’évaluation de déterminants 2D. Les calculs pour le test 3D sont donc plus nombreux et génèrent ainsi naturellement plus d’imprécision, les erreurs se répercutant au fur et à mesure des calculs. De plus, et c’est certainement la raison majeure de cette différence de comportement entre les deux tests, la précision du test d’intersection 3D dépend fortement de la configuration spatiale de la scène. En effet, le test d’intersection rayon/triangle que nous utilisons est basé sur le calcul de

Scène	nbTrgs après clipping	surconsommation
Erw6	810	0.003
Fairy	169K	0.64
Conference	221K	0.84
Sully	406K	1.55
NuclearCase	738K	2.82

Tableau 3.3 – Surconsommation de mémoire due aux algorithmes de gestion de précision. La surconsommation de mémoire est indiquée en Mo.

volumes formés par des tétraèdres joignant les extrémités du rayon aux arêtes du triangle. Plus le point de départ du rayon s'éloigne du triangle, plus le tétraèdre considéré s'allonge et rétrécit (voir Figure 3.11). Plus ce tétraèdre rétrécit, plus le signe de son volume (défini via une convention d'orientation du tétraèdre) devient incertain. Ainsi, nous avons pu observer que le déplacement du point de départ des rayons de la scène avait un impact important sur le nombre de tests d'intersections douteux. Lorsque le point de départ des rayons s'éloigne des objets de la scène, l'imprécision des tests 3D augmente considérablement, tandis que les tests 2D ne souffrent pas d'un tel comportement. Cela explique aussi pourquoi le pourcentage de tests 3D imprécis ne semble pas du tout constant (il est en particulier plus de deux fois plus important pour la scène Conference que pour la scène Fairy, pour un nombre de tests effectués assez équivalent), la position du point de départ des rayons par rapport à l'ensemble de la scène jouant un rôle essentiel dans la précision des calculs effectués. Pour ce qui concerne l'approche 2D, ce problème existe aussi (le triangle 2D rétrécit lorsque le centre de projection du triangle s'éloigne), mais semble donc beaucoup moins critique.

En conclusion, l'approche 2D est à préférer à son équivalent 3D et nous permet bien de garantir la robustesse des résultats, sans perte de performance notable (moins de 10% lors de tous nos tests) et sans nécessiter le réglage de certains coefficients propres à la scène.

3.5.3 Surconsommation de mémoire

Avant de conclure ce chapitre, il nous reste tout de même à mesurer la quantité de mémoire consommée par cette stratégie de gestion de la précision. En effet, si le besoin de stocker les coordonnées 2D des triangles, par exemple, n'est pas dépendant de notre souci de gestion de la précision, il n'en est pas de même pour toutes les données d'informations sur les triangles 2D. Comme nous l'avons vu, nous devons, au cours de nos algorithmes, garder en mémoire les identités des pères des nouveaux points du maillage 2D, après clipping. Nous devrions aussi, en principe, stocker les "cas" associés à chaque intersection, décrivant si l'intersection a lieu sur une arête, sur un sommet, ou à l'intérieur d'un triangle. Cependant, comme nous l'avons vu en 3.3.4, nous avons pu stocker cette information ainsi que l'indice du triangle intersecté dans une donnée unique. Nous avons donc pu éviter de conserver dans un espace à part ces informations, mais aussi effacer les coûts de transfert mémoire qui auraient été nécessaires pour rapatrier ces informations sur CPU. Le surcoût mémoire occasionné par nos algorithmes de gestion de la précision se limite donc à l'espace nécessaire pour conserver les informations sur les identités des "pères" des points 2D du maillage. Dans le tableau 3.3, nous pouvons voir la valeur de cette surconsommation de mémoire pour nos cas d'étude. Pour la scène nuclearCase (pire cas de ces tests), cette surconsommation s'élève ainsi à 2.8 Mo. Compte tenu des gains de robustesse apportés par ces algorithmes de gestion de la précision, cette surconsommation de mémoire nous semble, là aussi, tout à fait acceptable.

Conclusion

Nous avons présenté dans ce chapitre les différentes techniques nous permettant d'assurer un calcul de débit de dose “cohérent”, c'est-à-dire ne souffrant pas de la multiplication des approximations provoquées par les calculs en arithmétique flottante lors de l'exécution des tests d'intersections rayon-triangle. Afin de garantir cette cohérence, nous avons choisi d'exploiter les algorithmes adaptatifs de Jonathan Shewchuk, permettant de connaître le signe d'un déterminant 2D ou 3D de manière exacte, en exécutant les calculs en arithmétique flottante, puis en les affinant si besoin est.

Ces algorithmes ne convenant pas à une exécution sur GPU, nous avons proposé de n'effectuer que leur première passe sur GPU, avant d'exécuter le traitement des cas non résolus sur CPU. En couplant cette stratégie à une prise en compte de la topologie du maillage, nous avons pu mettre en place des algorithmes robustes moyennant une perte de temps mineure. La topologie du maillage étant modifiée au cours de l'étape de clipping permettant le calcul des projections 2D des sommets de triangles, nous avons dû garantir que tous les triangles de la scène subissaient des découpes cohérentes les uns envers les autres lors de cette étape de clipping, mais aussi que certaines informations concernant le maillage de départ, comme l'identité des points ayant donné naissance aux nouveaux sommets du maillage généré après clipping, étaient convenablement utilisées.

En conjuguant ainsi une utilisation de l'arithmétique exacte adaptée au fonctionnement du GPU et un traitement efficace des informations concernant la topologie du maillage, malgré les modifications que ce dernier subit au cours de l'étape de clipping, nous avons pu obtenir la gestion cohérente des tests d'intersections 2D rayon/triangle, sans qu'il soit nécessaire de régler la valeur de certains coefficients propres à la scène. La dégradation de performance provoquée par l'utilisation de ces méthodes de gestion de la précision (toujours inférieure à 10 % lors de nos tests) semble très acceptable au regard des gains de robustesse apportés.

Les algorithmes développés dans ce chapitre ont été présentés lors de la conférence *International Conference on Mathematics and Computational Methods applied to Nuclear Science and Engineering (MC 2011)*.

Stratégies de traitement de multiples paquets de rayons

4

Sommaire

4.1	Introduction	88
4.1.1	Configurations de test	88
4.1.2	Les différentes stratégies	89
4.2	Optimisation des calculs pour la stratégie “locale”	93
4.2.1	Les différentes étapes	93
4.2.2	Optimisation de l’étape de transformation des triangles	93
4.2.3	Réduction du nombre de triangles étudiés	96
4.2.4	Impact de la phase de tri/transfert des données	100
4.3	Résultats	101
4.3.1	Stratégie “globale”	101
4.3.2	Stratégie “locale”	103
4.3.3	Comparaison des deux stratégies	105
4.3.4	Augmentation du nombre de paquets	108
4.3.5	Performances globales de la stratégie “locale”	110
4.4	Discussion	112
4.4.1	Optimisations supplémentaires de l’algorithme actuel	112
4.4.2	Vers une structure hiérarchique ?	113

Au cours des deux chapitres précédents, nous avons présenté des méthodes permettant d’obtenir le traitement interactif d’un groupe de rayons très homogène, tout en garantissant des résultats “cohérents”, via des stratégies efficaces de gestion des imprécisions causées par les calculs en arithmétique flottante. Nous sommes cependant restés dans le cadre des hypothèses émises au début de la partie 1.2, à savoir le cas d’un seul groupe de rayons, respectant une certaine configuration géométrique et obéissant ainsi à un schéma “cohérent”. Dans ce dernier chapitre, nous souhaitons revenir vers le contexte de travail du calcul de débit de dose pour l’industrie nucléaire, dans lequel les scènes sont majoritairement composées de plusieurs sources de rayonnements. L’unique paquet que nous avons étudié jusqu’à présent sera donc désormais remplacé par plusieurs groupes de rayons homogènes, souvent moins importants en taille que ceux que nous avons étudiés jusque là (un million de rayons pour une même source de radiations 3D semble en effet être une limite haute de ce qui peut être observé dans la réalité).

Cette contrainte nous amène à développer de nouvelles méthodes afin d’optimiser nos calculs, mais aussi, parfois, à remettre en question la structure globale de notre lancer de rayons. Dans la première partie de ce chapitre (partie 4.1), nous présentons plus précisément notre problématique, en détaillant d’abord les cas-tests que nous avons étudiés avant d’examiner

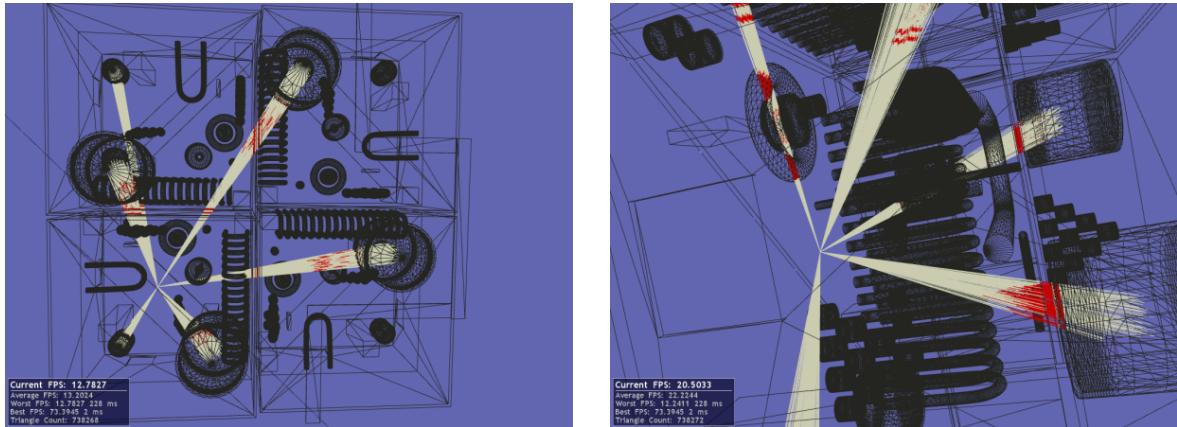


Figure 4.1 – Exemple de visualisation des interactions rayon-matière sur la scène nuclearCase (738 000 triangles). 6 cuves, contenant 100 000 sources de radiation, sont utilisées. Les rayons partant de ces sources de radiation convergent au point de mesure du débit de dose. Ils sont colorés en rouge lorsqu'ils interagissent avec un objet de la scène, et en blanc lorsqu'ils parcourent l'air ambiant.

les différentes pistes permettant de gérer ces multiples paquets de rayons. La méthode que nous privilégions consiste à définir une grille en perspective par paquet de rayons. Elle nécessite naturellement de minimiser les temps d'initialisation nécessaires au travail dans chacun des espaces en perspective définis par les différentes grilles utilisées. Nous présentons dans la partie 4.2 quelques optimisations et implémentations GPU mises en place afin d'optimiser ces phases de calculs. Les performances globales obtenues sont regroupées dans la partie 4.3, puis discutées (nous verrons entre autres les limites de cette approche) dans la partie 4.4, avant la conclusion de ce dernier chapitre.

4.1 Introduction

4.1.1 Configurations de test

En premier lieu, précisons les différentes configurations que nous souhaitons étudier dans ce chapitre. Afin de vérifier la qualité du comportement de nos algorithmes sur de multiples paquets de rayons, nous avons généré, au sein de la scène nuclearCase précédemment conçue, plusieurs groupes de points devant correspondre à des sources de radiations. Chacun de ces groupes de rayons correspond à une “cuve” de la scène (voir figure 4.1). Nous avons ainsi généré quatre cuves de grande taille, huit de taille moyenne, et douze de petite taille. Pour chacune de ces tailles de cuves, nous avons également créé des groupes de rayons de tailles différentes : les petits groupes correspondent à 2 500 rayons, les moyens à 22 500, et les grands à 100 000. En faisant varier ces quantités, nous avons donc déjà à notre disposition neuf configurations possibles, sans compter que nous pouvons également faire varier le nombre de petites, moyennes et grandes cuves. Cette variété de configurations doit nous permettre de couvrir une grande partie des cas que nous pourrons rencontrer dans la sphère nucléaire. Bien que nous ayons particulièrement adapté nos algorithmes au cas de paquets de rayons très volumineux, nous souhaitons également étudier leur comportement pour des paquets plus petits mais plus nombreux. Cela nous permettra de dégager les forces et les faiblesses de nos algorithmes actuels, et de proposer des pistes d'amélioration pour le futur.

4.1.2 Les différentes stratégies

Ces configurations de test étant présentées, nous pouvons détailler les stratégies permettant de résoudre notre problème. Nous avons principalement testé deux méthodes nous permettant de gérer de multiples paquets, aux qualités très distinctes.

4.1.2.1 Stratégie “globale”

La première stratégie, que nous qualifions de “globale”, est celle proposée par Hunt [HM08b] dans son papier introduisant la grille en perspective. Elle consiste en la création de six grilles en perspective complémentaires, permettant de couvrir l'intégralité de l'espace autour du centre de projection de la scène. Pour chaque axe du repère orthonormé de la scène (centré au point de mesure du débit de dose), deux grilles sont définies, l'une orientée dans le sens des coordonnées croissantes, l'autre dans le sens des coordonnées décroissantes. Chacune de ces grilles présente un angle d'ouverture de 90 degrés suivant les deux axes orthogonaux à leur axe directeur. Par exemple, une de ces grilles est centrée au point de mesure du débit de dose, ou point de départ des rayons (comme les cinq autres grilles), orientée suivant l'axe x , dans le sens des coordonnées croissantes, et selon un angle de 90 degrés du point de vue des coordonnées y et z . Plus intuitivement, il est possible d'imaginer un cube autour du point de mesure du débit de dose, centre de projection de la scène. Chaque grille en perspective est associée à l'une des six faces du cube, chacune de ces grilles pouvant simplement être représentée comme le résultat de la division régulière de l'image vue par une caméra placée au centre de projection de la scène et dirigée vers la face en question (voir sur la figure 4.2 un exemple de deux de ces six grilles).

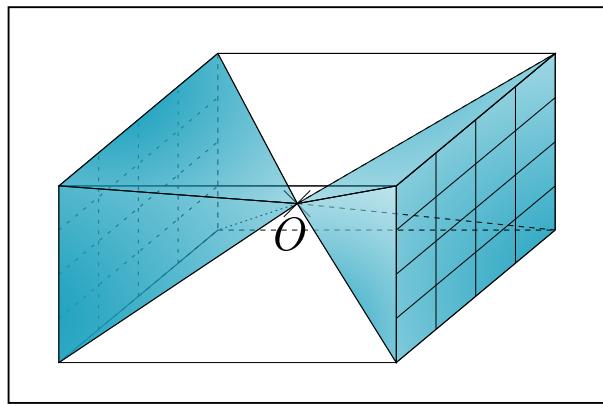


Figure 4.2 – Schéma de deux des six grilles en perspective couvrant l'intégralité de l'espace autour du point de mesure O du débit de dose. Pour faciliter la compréhension de la forme de ces grilles, nous avons modélisé un cube virtuel. Chaque grille en perspective repose sur une des faces de ce cube. Les deux grilles représentées sur ce schéma sont associées à deux faces opposées du cube, donc au même axe du repère de la scène, mais suivant des coordonnées opposées.

Afin de garantir l'absence de résultats incohérents, un soin particulier doit être apporté au traitement des rayons situés à proximité des frontières de la grille. En effet, ces zones de bordures sont partagées par deux, voire trois grilles voisines, ce qui peut provoquer deux types de problèmes différents. Tout d'abord, il est important de définir dans laquelle de ces grilles tombe un rayon passant dans ces zones limitrophes. Si aucune méthode robuste n'est

mise en place, des problèmes similaires à ceux évoqués dans le chapitre précédent peuvent apparaître : le rayon peut être vu comme appartenant à plusieurs grilles, problème que l'on peut a priori assez facilement traiter. Mais surtout, en raison, une fois de plus, des erreurs de précisions provoquées par l'arithmétique flottante, il peut n'être associé à aucune grille. Afin de garantir l'absence de ce genre de problèmes, la stratégie la plus sûre est de réaliser une première passe sur l'ensemble des rayons afin de définir la grille à laquelle chacun est associé.

Le deuxième problème pouvant alors apparaître est celui d'un triangle dont la frontière est située au niveau de cette zone de voisinage entre les grilles, mais dont seule l'arête (voire même le sommet) appartient à la grille choisie (un tel cas est représenté sur la figure 4.3). Sur cette figure, un rayon est également modélisé, passant par l'arête commune aux deux triangles. Ce rayon ayant été associé, arbitrairement, à la grille de droite, il faut garantir que le triangle ABC sera référencé dans la grille de droite (et, naturellement, dans la grille de gauche). De la même manière, ce rayon pourrait être référencé dans la grille de gauche, et le triangle ABD doit donc lui aussi être référencé dans les deux grilles. Dans le but de garantir ces comportements, nous “gonflons” chacune de nos grilles au moment de l'étape de clipping (voir 3.4), comme au niveau de l'étape de rastérisation (voir 2.3.1). Cette stratégie nous oblige à référencer plusieurs fois les mêmes triangles dans des grilles différentes, mais nous garantit des algorithmes robustes. Il faut préciser que nous parlons là uniquement d'un gonflement de la grille et non des triangles de la scène. Pour finir, notons que ce problème s'est effectivement présenté à nous lors de l'exécution des tests de robustesse présentés précédemment (voir 3.5.1). Sur nos 4 milliards de rayons testés, nous avons de nombreuses fois rencontré le cas d'un mauvais traitement de rayons intersectant une arête située à la frontière de deux grilles, et avons même dû faire face au cas de rayons intersectant un triangle en un sommet situé sur cette frontière. Les méthodes que nous venons de présenter nous ont cependant permis de régler facilement ces problèmes.

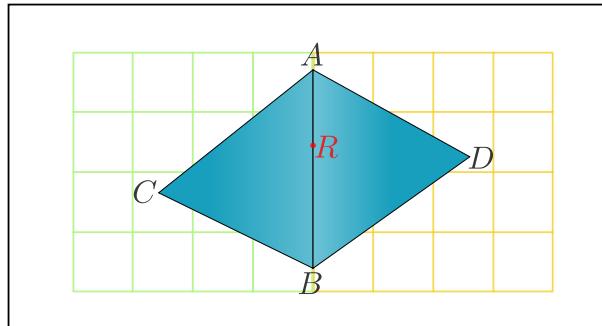


Figure 4.3 – Schéma d'une arête $[AB]$, appartenant aux deux triangles ABC et ABD , commune à deux grilles en perspective. Le point R modélise la position 2D d'un rayon de la scène, passant à proximité de cette arête. Il est ici arbitrairement identifié comme appartenant à la grille en perspective située à la droite du schéma.

Après avoir présenté la mise en place de cette stratégie, concentrons-nous sur ses qualités (et défauts) vis-à-vis du traitement de multiples paquets de rayons. L'avantage de cette stratégie est qu'elle est valable pour toutes les configurations possibles, puisque l'ensemble de l'espace est couvert par ces six grilles. Naturellement, cette stratégie présente les défauts associés à cette qualité : la subdivision de ces grilles étant régulière, il est très difficile de bien tenir compte de la nature des paquets de rayons (s'ils couvrent ou non un large angle solide autour du centre de projection, si les rayons sont nombreux ou en faible quantité...). Nous verrons dans la partie 4.3 que les performances de cette stratégie deviennent très vite médiocres en présence de paquets formant un faible angle solide, mais composés de nombreux

rayons. En effet, la résolution de chacune des six grilles devient alors beaucoup trop grossière pour que la grille élimine efficacement un grand nombre de triangles non intersectés. En raison du faible volume (du point de vue de l'angle solide) du paquet de rayons, de nombreuses cellules de la grille en perspective sont vides (de rayons), tandis que celles qui sont occupées contiennent un très grand nombre de rayons (à cause du grand nombre de rayons composant le paquet). La figure 4.4 montre un tel cas de figure.

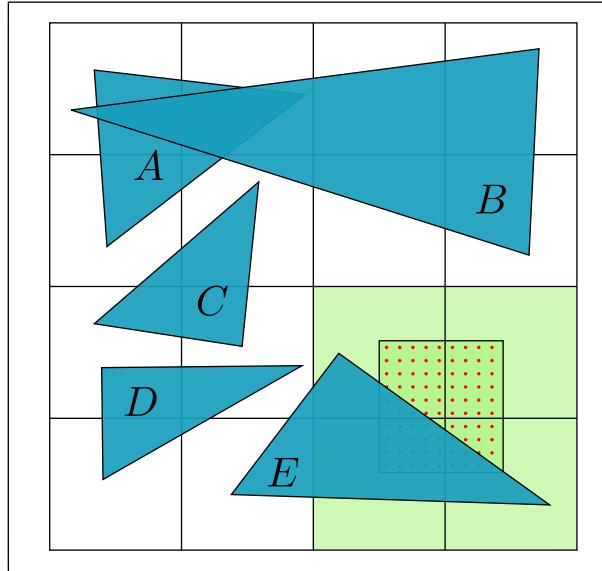


Figure 4.4 – Exemple de mauvaise gestion d'un paquet de rayons de faible volume par la stratégie “globale”. Cinq triangles sont représentés, au milieu d'une grille en perspective de grande résolution. Le paquet de rayons rouges n'intersecte que quatre cellules de la grille (colorées en vert).

Deux problèmes complémentaires apparaissent alors : tout d'abord, en raison de la résolution trop grossière de la grille, un très grand nombre de tests d'intersections vont être effectués, dont la majorité aurait été évitée avec une structure d'accélération mieux adaptée aux propriétés géométriques du paquet (sur la figure, on observe que les cellules vertes contiennent un grand nombre de rayons, qui vont générer de nombreux tests d'intersections avec le triangle *E*). Nous nous retrouvons alors face à un problème typique des grilles régulières, déjà évoqué lors de notre état de l'art (voir 1.3.2.1), de type *teapot in a stadium*. Les pertes de performances dans de telles configurations peuvent vite devenir considérables, et ce genre de situation peut aussi engendrer une très forte surconsommation de mémoire, toujours problématique sur GPU, lors de l'exécution des tests d'intersections. L'autre difficulté vient du fait que pour toutes les cellules vides de la grille, nous allons effectuer un travail de rastérisation et de préparation des tests d'intersections (étapes 2.3.1 à 2.3.5 de l'algorithme de triangle-tracing) parfaitement inutile (toujours sur la figure 4.4, la phase de rastérisation va être effectuée pour les triangles *A* à *D*, bien que ces triangles ne rencontrent que des cellules blanches, ne contenant donc aucun rayon). Afin de contourner ce problème, nous avons tenté d'identifier, dès l'étape de rastérisation, quelles cellules étaient réellement parcourues par des rayons avant de les ajouter à la liste de cellules à traiter, mais cette stratégie s'est en réalité révélée encore moins performante que la première. Elle impliquait en effet de nouveaux accès mémoire, parfois incohérents, au sein d'un kernel de rastérisation souffrant déjà de problèmes de répartition de la charge de travail, ce qui ne faisait qu'amplifier ce phénomène. Une autre solution peut alors consister à tenter d'identifier certains cônes englobant les groupes

de rayons, et de tester, avant la phase de rastérisation, si ces triangles intersectent ces cônes. Ne serions-nous pas alors en train d'effectuer une phase de clipping entre les triangles et le cône définissant le groupe de rayons ? Quitte à effectuer cette phase de clipping, pourquoi ne pas définir une grille en perspective par paquet de rayons, et ainsi profiter d'une résolution propre à chaque paquet ? Ces deux interrogations nous ont naturellement amené à définir la stratégie que nous allons maintenant présenter.

4.1.2.2 Stratégie “locale”

Comme nous venons de le suggérer, les défauts de la stratégie précédente nous ont incité à associer une grille en perspective à chaque groupe de rayons. Bien que cette méthode ne soit pas sans défauts (nous en parlerons plus en détails dans la partie 4.4), elle élimine une grande partie des difficultés associées à la stratégie dite “globale”. En effet, en définissant une grille en perspective par paquet de rayons, nous pouvons régler la résolution de la grille en fonction du nombre de rayons présents dans le paquet. Comme la répartition des sources de radiations est assez homogène au sein d'une source volumique émettrice, la structure de la grille régulière est assez adaptée à ce genre de cas (même si, pour certaines situations, certaines cellules de la grille peuvent devenir “trop grandes” par rapport à un groupe de triangles de très petite taille). Nous bénéficions ainsi assez directement des méthodes présentées lors des chapitres précédents. Il nous suffit, là encore, dans un souci de cohérence, de “gonfler” chacune des grilles de manière à ce que leur volume englobe (avec un peu de marge) le paquet de rayons associé. Avec cette méthode toutefois, un problème peut apparaître avec certains objets aux formes géométriques particulières, tels les tores. Pour un tel objet, à l'intérieur duquel des sources ponctuelles de radiation sont réparties, il peut arriver que le paquet de rayons couvre du point de vue du centre de projection un angle solide supérieur à 2π stéradians (voir figure 4.5). Nous sommes alors obligés de définir plusieurs grilles afin de couvrir l'espace occupé par le paquet de rayons. Nous ne nous sommes pas encore spécifiquement penchés sur ce problème, mais il faudrait sans doute définir quelques méthodes, a priori peu complexes, afin de définir les grilles adéquates.

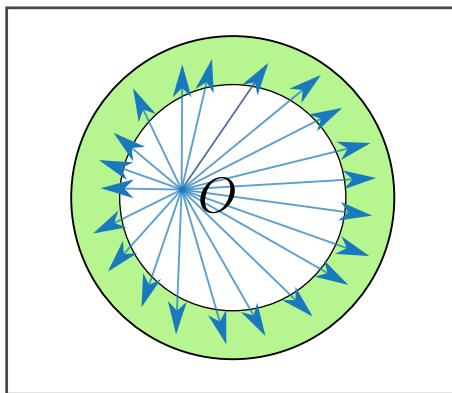


Figure 4.5 – Exemple d'un objet en forme de tore émettant des radiations, vu du dessus. Le point de mesure O , passant au centre de l'objet, se retrouve encerclé par les sources de radiations. Il n'est alors pas possible de couvrir l'ensemble de ces rayons avec une seule grille en perspective.

La difficulté posée par cette stratégie est principalement liée aux calculs redondants qu'elle implique. En dehors du fait que plusieurs groupes de rayons peuvent couvrir le même espace vis-à-vis du point de mesure du débit de dose (et donc, devraient, normalement, pouvoir bénéficier de certains calculs communs, comme par exemple les projections dans l'espace à

deux dimensions), nous allons surtout devoir veiller à minimiser la perte de temps lors de la gestion d'un grand nombre de groupes de rayons. En effet, à chaque paquet de rayons, donc à chaque grille, est associée une étape de clipping, mais aussi la projection des coordonnées des triangles en deux dimensions, et, naturellement, la construction de la grille en perspective présentée en 2.2. Dans le cas de la stratégie “globale” précédemment définie, ces calculs sont mis en commun entre les différents paquets de rayons, et l'augmentation du nombre de paquets est donc beaucoup moins pénalisante. Afin d'optimiser nos performances pour cette stratégie “locale”, nous devons donc particulièrement veiller à l'optimisation de cette phase d'initialisation que nous n'avons pas encore détaillée dans ce manuscrit.

4.2 Optimisation des calculs pour la stratégie “locale”

4.2.1 Les différentes étapes

La phase d'initialisation comporte plusieurs parties : en premier lieu, il faut procéder à l'étape de “transformation des triangles”, décrivant l'ensemble des calculs permettant de procéder au clipping des triangles puis de générer le nouveau maillage résultant. Une fois le nouveau maillage 3D créé, il nous faut d'abord projeter ces données dans le repère 3D associé à la grille en perspective (axe Z suivant l'axe directeur de la grille). En effet, nous effectuons les calculs de l'étape de clipping dans l'espace classique à trois dimensions, centré au point de projection de la grille. Dans ce repère, les normales des faces du cône constituant les limites de la grille en perspective sont plus faciles à définir, et permettent donc des calculs plus rapides. De plus, comme de nombreux triangles du maillage initial peuvent ne pas du tout intersecter le cône associé à la grille, il ne sert à rien de les projeter dans le repère 3D de la grille avant de savoir s'ils donneront naissance à des triangles exploités par la grille. Une fois ces projections dans le repère 3D local à la grille effectuées, il nous faut projeter chacun des sommets de triangles dans l'espace à deux dimensions défini par le centre de la grille en perspective et son axe directeur. La dernière étape de cette phase d'initialisation consiste en la création de la grille de rayons détaillée plus haut (partie 2.2).

Les étapes de projection des sommets dans le repère local en trois dimensions, comme dans le repère 2D, sont des phases intrinsèquement parallèles, et correspondent parfaitement aux capacités de calcul du GPU. En revanche, la création de la grille de rayons est nettement moins intuitive, mais nous avons déjà largement abordé les différentes stratégies permettant de minimiser son temps d'exécution. Il nous reste par conséquent à aborder l'optimisation de l'étape de transformation des triangles, assez complexe à paralléliser, et dont on verra dans la partie 4.3 qu'elle représente le goulot d'étranglement de cette phase d'initialisation. Afin d'obtenir des performances intéressantes lors du traitement de multiples paquets de rayons, il est donc essentiel d'optimiser son implémentation sur GPU.

4.2.2 Optimisation de l'étape de transformation des triangles

En premier lieu, il convient de détailler les différentes phases de cette étape de transformation des triangles, car celles-ci n'ont pas du tout le même impact sur les performances. Ces différentes phases sont les suivantes :

1. Tout d'abord, il faut préparer le stockage des nouveaux points générés par le clipping. Après découpe, chaque triangle peut donner naissance jusqu'à sept nouveaux sommets de triangles (voir Figure 4.6 un schéma de ce pire cas). Nous allouons donc à l'avance

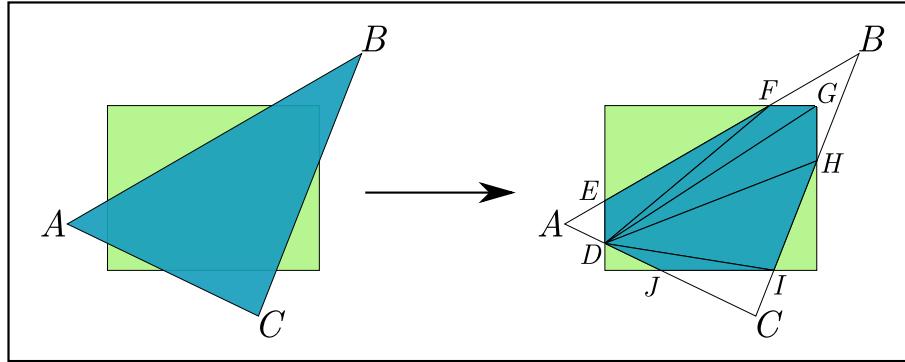


Figure 4.6 – Clipping d’un triangle ABC donnant lieu à la création de sept nouveaux sommets de triangles (D à J), répartis dans cinq nouveaux triangles, partant tous du sommet D .

l’espace nécessaire au stockage de ces sept points (trois flottants par point, le clipping ayant lieu en trois dimensions) pour chaque triangle. Là encore, nous stockons ces sommets suivant le schéma *Structure of Arrays* (voir 2.3.2) : les sept emplacements associés à un triangle ne sont pas contigus en mémoire ; en revanche, les $n^{\text{ème}}$ sommets associés à des triangles successifs occupent des emplacements contigus en mémoire. Cette précision est importante, car elle nous a permis de diviser par deux le temps d’exécution de la seconde phase de l’algorithme, assez lourde en calculs. Néanmoins, contrairement aux apparences, cette première étape reste souvent la plus lourde, car elle fait appel à des allocations mémoire conséquentes, et à des opérations de lecture /écriture en mémoire parfois très incohérentes.

2. Après avoir préparé l’espace nécessaire au stockage de ces nouveaux sommets, nous procédons à l’étape de clipping en elle-même : pour chaque triangle, nous stockons les nouveaux points générés et gardons en mémoire le nombre de ces nouveaux sommets par triangle.
3. Comme la plupart des triangles génèrent beaucoup moins de sept points (certains même, non intersectés par le cône enveloppant la grille, n’en génèrent aucun), une grande quantité de mémoire est alors consommée inutilement. Il nous faut donc maintenant éliminer cet espace superflu. Pour cela, nous commençons par compter le nombre de nouveaux triangles pour chaque triangle d’origine. Notre manière de définir les nouveaux triangles fait qu’une relation simple existe entre le nombre de nouveaux triangles et le nombre de nouveaux sommets :

$$nbNouveauxTriangles = nbNouveauxSommets - 2$$

Cette formule vient du fait que pour former les nouveaux triangles, nous partons du premier nouveau point généré, que nous lions à tous les autres via des arêtes de nouveaux triangles. Sur la figure 4.6, le premier nouveau point D appartient bien aux $7 - 2 = 5$ nouveaux triangles générés.

Nous effectuons donc un scan exclusif (somme cumulée) pour savoir où démarera le stockage des informations liées à chaque nouveau triangle du maillage (nous ne parlons pas ici des coordonnées des points, mais des indices A, B, C de chaque point dans la liste des nouveaux sommets). Au passage, nous générerons ainsi le nombre de triangles dans le nouveau maillage.

4. Comme pour les nouveaux triangles, nous comptons ensuite le nombre de “nouveaux sommets” par triangle du maillage initial, avant, là encore, d’effectuer un scan exclusif,

pour savoir où commencera le stockage des coordonnées de points associés à chaque nouveau triangle.

5. Enfin, nous effectuons un dernier kernel (un thread par triangle de l’ancien maillage), qui se charge de copier les coordonnées des nouveaux points dans la version finale du maillage. Les indices de chaque sommet de triangle dans la liste de points sont également stockés lors de l’exécution de ce kernel.

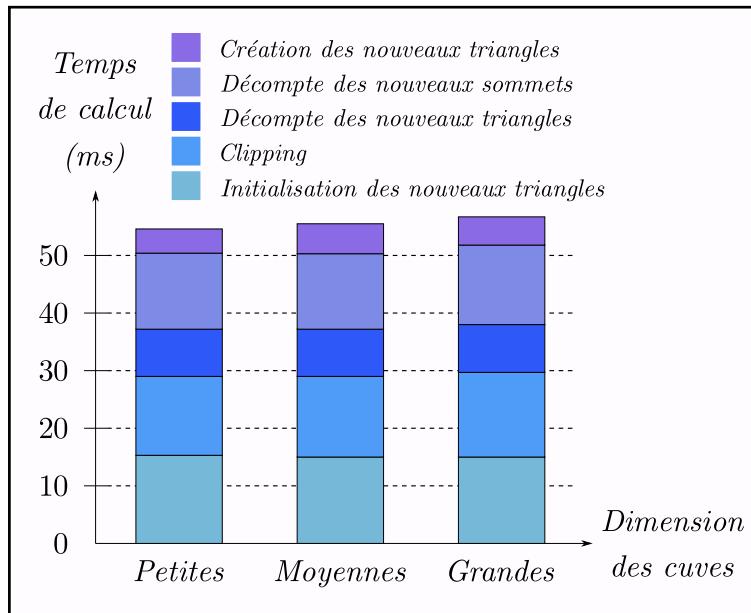


Figure 4.7 – Répartition des performances lors de la phase de transformation des triangles, pour le cas de 4 cuves de dimension variable (les tests de “rejet rapide” présentés plus loin ne sont ici pas encore utilisés).

Sur la figure 4.7, nous avons repris les performances de ces différentes étapes, lors du traitement de quatre cuves de “grande”, “moyenne” et “petite” dimension (il peut être noté au passage que les performances pour des cuves de dimensions inférieures sont peu différentes, nous verrons dans la sous-partie 4.2.3 qu’avec une meilleure stratégie de clipping, les cuves de plus faible dimension demandent un peu moins de temps de calcul). Cette figure montre que toutes les étapes de cette phase d’initialisation n’ont pas la même importance. Lors de nos premières implémentations, la deuxième étape de l’algorithme, à savoir la phase de clipping, consommait un temps très important, ce qui s’avérait extrêmement pénalisant pour le traitement de multiples paquets de rayons. Nous avons néanmoins réussi à réduire de manière importante le temps d’exécution de cette phase, en mettant en place plusieurs optimisations. Nous avons ainsi utilisé le plus possible la mémoire partagée pour stocker les données fréquemment utilisées, quitte à réduire le nombre de blocs exécutables sur chaque multiprocesseur. Nous avons également travaillé sur le nombre de registres utilisés par chaque bloc. Ces diverses optimisations nous ont permis, sur une scène comme Sully, de faire passer le temps nécessaire à l’exécution de cette phase de 11 à 4 millisecondes.

Nous espérons encore dans le futur pouvoir améliorer ces performances, peut-être en séparant la phase consistant à identifier quels points doivent être gardés et quelles arêtes doivent être coupées, de la phase où l’on effectue réellement ces copies et où l’on calcule les coordonnées de ces points issus de découpes d’arêtes. En effet, notre programme actuel souffre encore de problèmes de synchronisation entre les différents threads d’un warp, et nous espérons, via cette modification, améliorer le parallélisme du kernel. Entre autres, une fois

les copies et les découpes d’arêtes effectuées, nous pourrions faire travailler chaque thread sur une copie ou une découpe différente, tandis que dans l’implémentation actuelle, le même thread doit effectuer toutes les copies et découpes associées à un même triangle.

4.2.3 Réduction du nombre de triangles étudiés

Néanmoins, nous avons pour l’instant passé sous silence une optimisation essentielle dans notre traitement de cette phase de transformation des triangles. Lors de nos premières implémentations, nous appliquions ce traitement à chaque triangle du maillage (la figure 4.7 du paragraphe précédent a été générée avec cette stratégie naïve). Cela provoquait une surconsommation de mémoire importante et surtout une très forte perte de temps lors de la phase d’initialisation des traitements. En effet, dans le cas d’une scène comportant des faisceaux de rayons de faible volume, chaque faisceau intersecte une proportion très faible de triangles par rapport à la globalité du maillage. Il est donc parfaitement inutile, et particulièrement coûteux, d’effectuer, pour chaque triangle de la scène et pour chaque faisceau, la phase de clipping. Nous avons donc mis au point des tests de “rejet rapide”, très souvent utilisés dans les travaux de la communauté du lancer de rayons (voir par exemple [BWS06]). Dans notre cas, ces tests permettent d’identifier rapidement qu’un triangle n’intersecte pas le cône formé autour d’une grille en perspective. De manière générale, ce genre de tests ne permet pas de garantir l’éviction de tous les triangles n’intersectant pas le cône, mais doit en éliminer le plus grand nombre possible, tout en demandant un minimum de calculs.

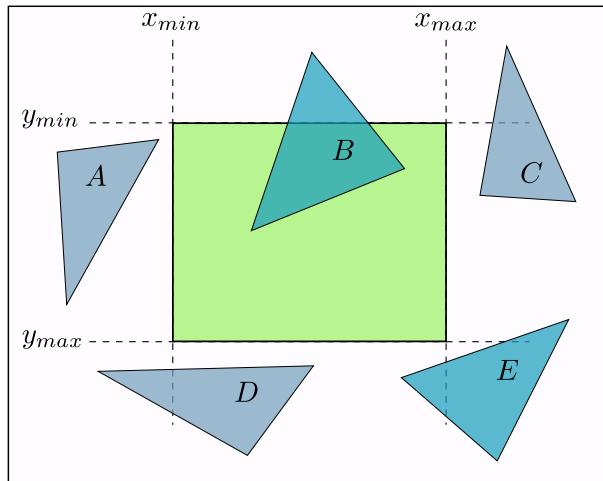


Figure 4.8 – Résultats de tests de “rejet rapide” pour cinq triangles différents (vue en deux dimensions, du centre de projection de la scène, les tests sont normalement effectués en trois dimensions). Les triangles directement rejetés par ces tests (A , C et D) sont grisés sur la figure. Le triangle A est rejeté car tous ses sommets sont situés à gauche du plan $x = x_{min}$, le triangle C car tous ses sommets sont situés à droite du plan $x = x_{max}$, et le triangle D car tous ses sommets sont situés sous le plan $y = y_{max}$. Les triangles B et E ne sont rejetés par aucun de ces plans, ils seront donc traités lors de l’étape de clipping.

Nous avons utilisé comme critère simple le fait de vérifier si chaque triangle n’est pas situé du côté “extérieur” d’un plan du cône (voir figure 4.8 pour quelques exemples de triangles facilement rejetés ou non). Pour chaque triangle, il suffit de vérifier de quel côté du plan se situe chacun de ses sommets. Pour chaque sommet, un simple calcul de produit scalaire est nécessaire (entre son vecteur de coordonnées et la normale du plan, puisque nous centrons, pour l’étape de clipping, l’espace 3D au centre de projection de la scène). Les plans que

nous avons utilisés pour évincer facilement des triangles sont naturellement les quatre plans formant le cône de vision, mais aussi le plan de coordonnée $Z = 0$ dans le repère local associé à la grille.

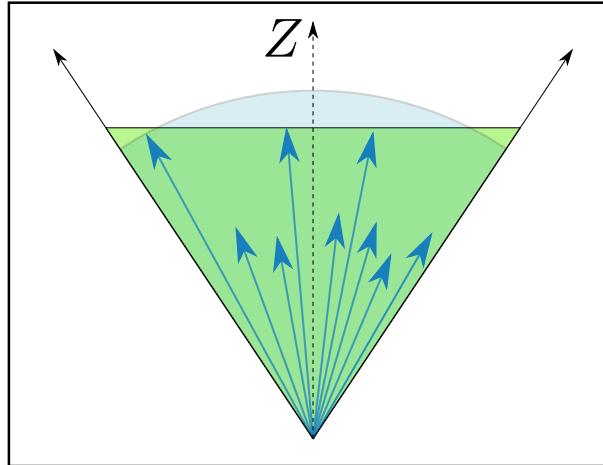


Figure 4.9 – Plan “de profondeur maximale” associé à un faisceau de rayons. Le calcul de ce plan ne se base pas en fait sur le calcul du plus long rayon (cas du rayon situé le plus à gauche sur la figure), mais sur l’identification du rayon atteignant la profondeur Z la plus grande.

Enfin, nous avons considéré un autre plan, le plan dit “de profondeur maximale”, permettant d’écarter tous les triangles situés “derrière” le faisceau de rayons (du point de vue du point de mesure du débit de dose), et ne pouvant donc intersecter aucun rayon de ce paquet. Pour trouver ce plan, il nous faut avant tout calculer, au sein du paquet étudié, la distance maximale entre le point de départ d’un rayon et son point d’arrivée (suivant l’axe Z du repère, voir figure 4.9). Une fois cette distance d calculée, nous générerons simplement le plan “de profondeur maximale” comme étant le plan perpendiculaire à l’axe Z (dans le repère local associé à la grille) situé à une distance d du centre de projection de la scène. L’utilisation de ce plan peut par exemple être très utile lorsqu’une source volumique de radiations se situe entre le point de mesure du débit de dose et un large mur de la scène. En effet, il n’est pas rare que certains triangles constituant un tel mur recouvrent alors l’intégralité des cellules de la grille en perspective. Ces triangles sont les plus durs à manipuler, puisqu’ils demandent un traitement très important en comparaison des autres (ils occupent en effet un grand nombre de cellules). S’ils ne sont pas intersectés par le faisceau de rayons, il peut donc être particulièrement intéressant de les éliminer rapidement.

Sur les tableaux 4.1, 4.2 et 4.3, nous montrons l’évolution des performances observées en fonction du nombre de plans utilisés lors de ces tests de “rejet rapide”, pour nos quatre cuves de diverses dimensions. En premier lieu, il est important de noter la faible perte de temps qu’implique la phase de tests de rejet. Il est particulièrement intéressant d’observer que le temps de calcul associé à cette phase semble quasiment rester constant lorsque l’on utilise entre 1 et 6 plans de “rejet rapide”. Cela est dû au fait que les accès mémoire nécessaires lors de cette phase sont dominants, et que l’augmentation des calculs, comme elle n’entraîne pas de surconsommation excessive de registres, est donc très peu perceptible. La deuxième observation, plus importante, concerne naturellement le gain de temps observé grâce à ces tests de rejet rapide. Pour le cas le moins avantageux, celui des cuves de “grande” taille, le temps dédié à la phase de clipping passe de 29 à 4 millisecondes, soit un facteur d’accélération supérieur à 8.

Il convient alors de remarquer que le gain de performance réalisé sur l’ensemble de la phase

nbPlans	nbTrgsGardés	tps Tests rejet	tps clipping	tps init	tps total
0	738 K	0.0	29.8 (-0.0)	78.3 (-0.0)	160.1 (-0.0)
1	614 K	2.8	25.2 (-4.6)	75.2 (-3.1)	156.7 (-3.4)
2	373 K	2.8	16.8 (-13.0)	58.4 (-19.9)	140.2 (-19.9)
3	110 K	2.8	7.2 (-22.6)	41.0 (-37.3)	122.9 (-37.2)
4	107 K	2.9	7.0 (-22.8)	40.5 (-37.8)	121.6 (-38.5)
5	44 K	2.9	4.2 (-25.6)	35.4 (-42.9)	116.3 (-43.8)
6	44 K	2.9	4.2 (-25.6)	35.5 (-42.8)	116.4 (-43.7)

Tableau 4.1 – Performances comparées des tests de “rejet rapide” sur 4 cuves de grande dimension. 6 plans de rejet sont utilisés au fur et à mesure, le premier plan correspondant au plan $Z = 0$ dans le repère local de la grille, les quatre plans suivants aux quatre faces du cône englobant le faisceau de rayons, le sixième plan étant le plan dit “de profondeur maximale”. $nbTrgsGardés$ indique le nombre moyen de triangles conservés par cuve, après les tests de rejet. La colonne *tps init* tient ici compte de l’ensemble de la phase d’initialisation, y compris la création de la grille de rayons et la projection des triangles dans le repère 2D propre à la grille. La colonne *tps total* indique le temps global d’exécution de l’algorithme, avant les phases de tri et de transfert des intersections vers le CPU. Les valeurs entre parenthèses indiquent les variations de performance par rapport au cas $nbPlans = 0$. Les temps sont indiqués en millisecondes.

nbPlans	nbTrgsGardés	tps Tests rejet	tps clipping	tps init	tps total
0	738 K	0.0	29.1 (-0.0)	78.3 (-0.0)	163.3 (-0.0)
1	506 K	2.8	20.7 (-8.4)	65.3 (-13.0)	152.1 (-11.2)
2	260 K	2.8	12.1 (-17.0)	50.0 (-28.3)	136.8 (-26.5)
3	29 K	2.9	3.4 (-25.7)	33.7 (-44.6)	120.6 (-42.7)
4	11 K	2.9	1.6 (-27.5)	30.3 (-48.0)	116.4 (-46.9)
5	10 K	2.9	1.4 (-27.7)	30.2 (-48.1)	116.2 (-47.1)
6	10 K	2.9	1.4 (-27.7)	30.0 (-48.3)	115.3 (-48.0)

Tableau 4.2 – Performances comparées des tests de “rejet rapide” sur 4 cuves de dimension “moyenne”. Le tableau suit la même organisation que le tableau 4.1.

nbPlans	nbTrgsGardés	tps Tests rejet	tps clipping	tps init	tps total
0	738 K	0.0	29.2 (-0.0)	76.5 (-0.0)	188.0 (-0.0)
1	654 K	2.8	26.0 (-3.2)	73.9 (-2.6)	184.8 (-3.2)
2	346 K	2.8	15.4 (-13.8)	55.7 (-20.8)	166.2 (-21.8)
3	34 K	2.9	3.5 (-25.7)	33.1 (-43.4)	143.9 (-44.1)
4	10 K	2.9	1.5 (-27.7)	29.4 (-47.1)	140.0 (-48.0)
5	10 K	2.9	1.4 (-27.8)	29.1 (-47.4)	139.7 (-48.3)
6	5 K	2.9	0.9 (-28.3)	28.6 (-47.9)	109.0 (-79.0)

Tableau 4.3 – Performances comparées des tests de “rejet rapide” sur 4 cuves de faible dimension. Le tableau suit la même organisation que le tableau 4.1.

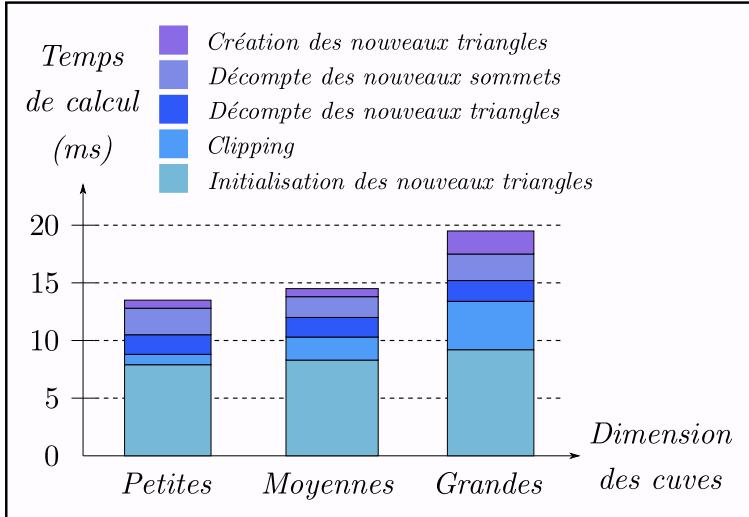


Figure 4.10 – Répartition des performances lors de la phase de transformation des triangles, pour le cas de 4 cuves de dimension variable (les tests de “rejet rapide” sont ici utilisés).

d’initialisation est supérieur au temps gagné lors de la phase de clipping, comme l’indiquent les quantités placées entre parenthèses dans les trois tableaux. En réalité, le fait d’évincer dès le début de la phase d’initialisation un grand nombre de triangles permet de gagner du temps lors de plusieurs étapes de la phase de transformation des triangles. Sur la figure 4.10, à comparer avec la figure 4.7 précédemment présentée (en notant la différence d’échelle entre les deux figures), on peut observer le temps pris pour chacune de ces étapes, lorsque les six plans séparateurs sont utilisés lors de ces tests de rejet. On observe alors en effet que de nombreuses étapes de l’algorithme bénéficient de l’utilisation de ces plans séparateurs, en particulier les étapes, comme l’étape “création de triangles”, souffrant de forts défauts de parallélisme. Bien que nous ne puissions pas directement le lire sur ces figures, un gain de temps notable est également réalisé en raison des allocations et désallocations de mémoire, beaucoup moins importantes grâce à cette nouvelle stratégie.

	nbCouples Cell-Trg	nbTests
sans plan de profondeur max.	4.1 M	11.0 M
avec plan de profondeur max.	2.6 M	7.3 M

Tableau 4.4 – Impact de l’utilisation d’un plan “de profondeur maximale”, lors du traitement de 4 cuves de faible dimension. La colonne *nbCouples Cell – Trg* indique le nombre de couples cellule-triangle obtenus après rastérisation des triangles.

Enfin, il est intéressant de noter que pour le cas des cuves de faible dimension, un gain de temps très important est réalisé sur le temps global de l’algorithme lors de l’ajout du sixième plan séparateur dit “de profondeur maximale”. Le temps global d’exécution de l’algorithme passe alors en effet de 140 à 109 millisecondes, tandis que la phase d’initialisation est presque inchangée. Comme nous l’avons expliqué, le plan “de profondeur maximale” joue alors son second rôle et permet de diminuer grandement le nombre de couples cellule-triangle mémo-risés et donc, le nombre de tests d’intersections exécutés. Sur le tableau 4.4, nous pouvons observer la diminution de ces deux quantités lors de l’ajout de ce sixième plan séparateur, qui explique les gains de temps observés. Le fait que ce plan “de profondeur maximale” ne permette pas les mêmes gains de performance pour les autres tailles de cuves s’explique par plusieurs considérations : premièrement, les cuves sont placées à des endroits différents dans

nbInters	tri	réorg Indices	transfert	total synchrone	total asynchrone
3.7M	24.5	4.2	6.8	35.5	31.5
4.3M	26.7	4.7	7.8	39.2	34.8
8.7M	52.3	9.8	15.8	77.9	68.4

Tableau 4.5 – Impact sur les performances de l'utilisation de transferts de mémoire asynchrones. Les colonnes *tri*, *réorg Indices* et *transfert* indiquent les temps séparés des trois étapes constituant la phase de tri/transfert des intersections vers le CPU. Les deux dernières colonnes indiquent le temps nécessaire à l'exécution de cette phase, en fonction de la nature des transferts de mémoire utilisés (synchrone ou asynchrones). Les temps sont donnés en millisecondes.

la scène, et elles ne souffrent pas toutes de la même manière de la présence de triangles situés derrière ce plan de profondeur maximale ; par ailleurs, leurs positions propres vis-à-vis de tels triangles ne sont pas les mêmes, et il peut arriver que le plan “de profondeur maximale” ne soit pas assez sélectif pour certaines configurations, mais soit meilleur dans d’autres cas, en fonction uniquement de la disposition des objets dans la scène. Cet effet est accentué lorsque la dimension des cuves augmente, puisque la conception du plan “de profondeur maximale” est telle qu’il devient de moins en moins efficace à mesure que la largeur du faisceau de rayons croît. Il est donc assez naturel d’observer un gain de temps supérieur lors du traitement de cuves de plus petite taille, donc de faisceaux de rayons plus fins.

4.2.4 Impact de la phase de tri/transfert des données

Afin de terminer cette étude détaillée des optimisations possibles lors du traitement de multiples paquets de rayons via la stratégie “locale”, il convient d’évoquer la dernière phase de l’algorithme sur GPU (avant l’analyse des intersections présentée en 3.3), consistant à trier et à transférer les données d’intersections vers le CPU. Comme nous l’avons signalé dans le chapitre 2, nous avons mis au point un kernel peu optimisé permettant, une fois les intersections triées par indice de rayon, de les trier, pour chaque rayon différent, par profondeur. Si cette méthode donne de très bons résultats pour des scènes au sein desquelles chaque rayon rencontre un faible nombre de surfaces, elle présente des performances bien moins bonnes en présence de rayons intersectant une, voire plusieurs dizaines de surfaces, comme c’est le cas au sein de la scène que nous avons créée. Pour ces tests, nous avons donc préféré mettre en place une stratégie consistant à trier simultanément les intersections par indice de rayon, puis par profondeur pour chaque rayon. Pour cela, nous avons concaténé les informations *rayId* (de type *int*) et *t* (de type *float*), dans un seul élément de type *double*. Comme les valeurs de profondeur *t* sont toujours positives, leur bit de signe est toujours nul, ce qui garantit le bon fonctionnement d’un tel tri (pour des valeurs de type *float* positives, le fonctionnement d’un tri par base (*radix*) est déjà garanti par la continuité de la représentation entre flottants normalisés et dénormalisés).

Une fois ce tri effectué, les positions des intersections dans le vecteur les répertoriant sont modifiées. Il convient donc alors de répercuter ces changements de position dans le vecteur des indices de triangle intersectés, de façon que toutes les valeurs situées en position *i* de ces vecteurs (*rayIdsFinal*, *trgIdsFinal* et *profondeursFinal*, voir paragraphe 2.3.8) correspondent à la même intersection. Cette opération peut prendre un temps important, puisqu’elle demande des écritures en mémoire potentiellement très incohérentes. Comme, au moment d’effectuer cette opération, les vecteurs d’intersections correspondant aux indices de rayon *rayId* et aux profondeurs *t* sont déjà correctement triés, nous procédons à leur transfert vers le CPU de manière asynchrone, en même temps que nous réordonnons les indices de

triangles intersectés. Nous pouvons observer sur le tableau 4.5 les gains de performance apportés par ces transferts asynchrones, qui nous permettent de masquer une très grande partie de cette phase de réorganisation d'indices. Ce tableau nous permet surtout de constater que l'étape de tri/transfert des données est largement dominée par la phase de tri, qui mériterait sans doute une attention particulière dans le cadre d'implémentations futures.

4.3 Résultats

Comme annoncé au début de ce chapitre, nous allons maintenant présenter les différents résultats obtenus par les deux stratégies que nous avons choisies, en faisant varier le volume géométrique des paquets de rayons considérés, mais aussi le nombre de rayons que ces paquets contiennent.

4.3.1 Stratégie “globale”

En premier lieu, nous proposons donc de mesurer les performances de la stratégie “globale”, consistant à générer six grilles en perspective pour représenter l'espace, quelle que soit la configuration géométrique de la scène. Nous étudions ici séparément le traitement de quatre cuves dont la taille varie (trois tailles, “grande”, “moyenne” et “petite”). Pour chaque taille de cuve, nous faisons varier le nombre de sources de radiations qu'elle contient, avec selon les cas 2 500 ($50 * 50$), 22 500 ($150 * 150$) et presque 100 000 (en réalité $316 * 316 = 99856$) sources. Notons que ces sources de radiations suivent une répartition assez homogène dans chaque cuve, mais qu'elles ne suivent pas de schéma régulier comme celles traitées lors des phases de tests du chapitre 2. Nous obtenons donc neuf configurations différentes à tester. Pour chacune de ces configurations, nous avons fait varier la résolution des six grilles en perspective. Il est important de noter que nous choisissons à chaque fois la même résolution pour les six grilles. De meilleurs résultats pourraient sans doute être obtenus en utilisant des résolutions différentes pour chaque grille de la scène, mais les problèmes que nous voulons mettre en perspective (de style *teapot in a stadium*, entre autres), resteraient tout de même présents. Enfin, précisons que nous ne mesurons pas ici l'ensemble du processus de lancer de rayons. En effet, nous ne comptons ici pas le temps de transfert des rayons du CPU vers le GPU, ni le tri final des intersections, et leur transfert du GPU vers le CPU. En effet, toutes ces étapes sont communes à nos deux stratégies (et même, quasiment, à toute stratégie GPU) et ne méritent donc pas d'être ici comparées. Nous étudierons cependant leur impact à la fin de cette partie (voir 4.3.5).

Les performances obtenues lors de nos tests sont reportées sur les figures 4.11, 4.12 et 4.13, correspondant respectivement aux cuves de grande, de moyenne et de petite taille. En premier lieu, il convient de préciser que, lors de nos tests, nous avons testé plus de trois résolutions différentes pour les six grilles de la scène. Cependant, pour chaque cas, la performance optimale était associée à une large plage de résolutions de grilles. Ainsi, les temps de calcul que nous affichons témoignent quasiment toujours du meilleur temps de calcul obtenu. Pour certaines configurations, nous avons pu observer des performances sensiblement meilleures avec des résolutions différentes de celles que nous présentons sur ces figures (par exemple, pour le cas des cuves de grande taille, et avec un faible nombre de rayons, nous avons observé un temps de calcul de 78 millisecondes pour une résolution de grille de $100 * 100$, au lieu des 86 millisecondes de la résolution $400 * 400$). Cependant, les écarts de performances observés dans de tels cas ne modifient en rien les conclusions que nous pouvons tirer de ces expérimentations.

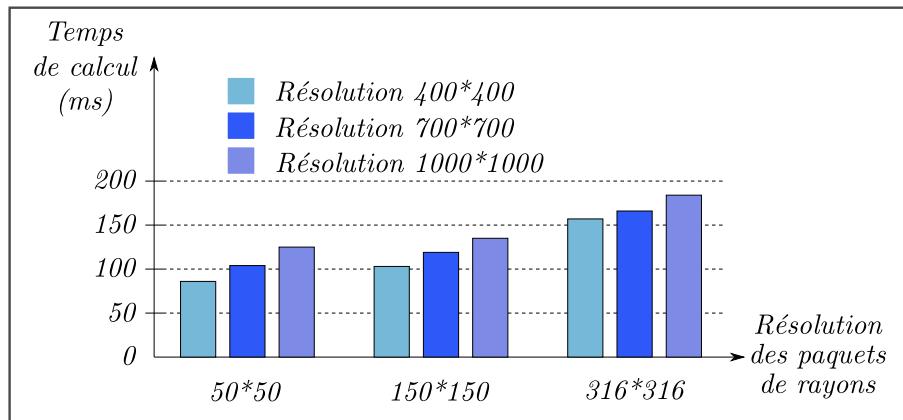


Figure 4.11 – Performances de la stratégie “globale” lors du traitement de 4 cuves de grande dimension. Pour chaque résolution de paquets de rayons, trois résolutions de grille (400 * 400, 700 * 700 et 1000 * 1000) sont testées.

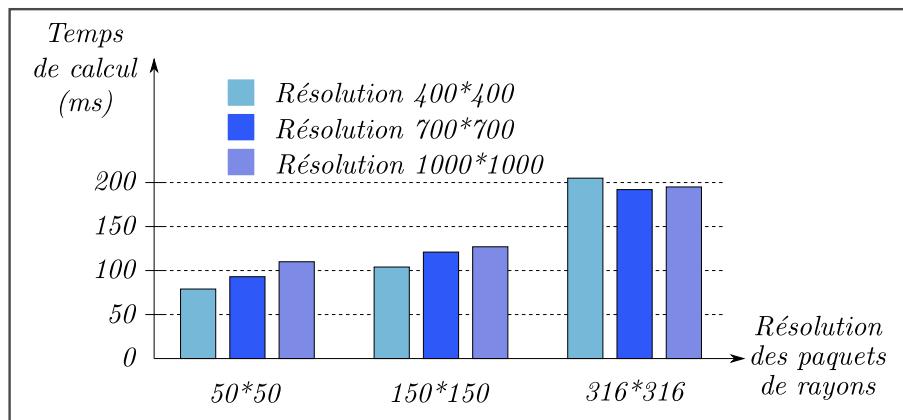


Figure 4.12 – Performances de la stratégie “globale” lors du traitement de 4 cuves de dimension “moyenne”. Pour chaque résolution de paquets de rayons, trois résolutions de grille (400 * 400, 700 * 700 et 1000 * 1000) sont testées.

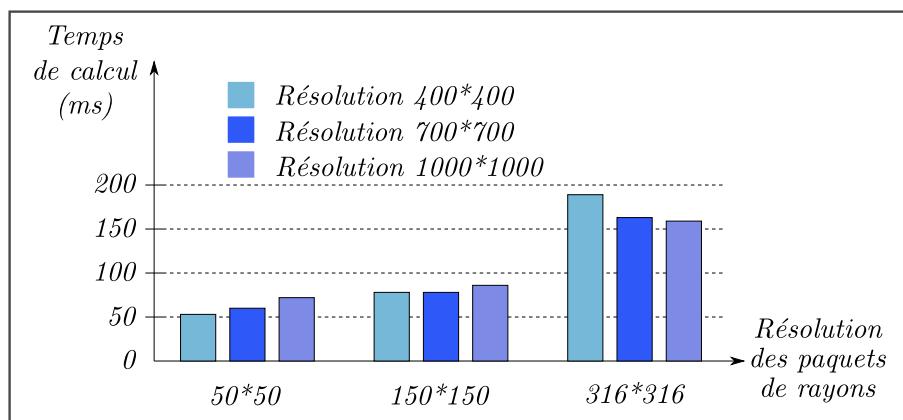


Figure 4.13 – Performances de la stratégie “globale” lors du traitement de 4 cuves de faible dimension. Pour chaque résolution de paquets de rayons, trois résolutions de grille (400 * 400, 700 * 700 et 1000 * 1000) sont testées.

Au vu de ces résultats, plusieurs constats s'imposent : tout d'abord, bien que les résolutions choisies pour les grilles en perspective influencent les résultats, les variations de performances en fonction de ces choix semblent assez légères. Cette observation est similaire à celle effectuée en 2.4.4 : lorsque la résolution de la grille est faible, l'étape de rastérisation est plus courte, mais la quantité de tests d'intersections augmente. Pour des résolutions plus élevées, les tests d'intersections sont moins nombreux, mais le temps nécessaire à la rastérisation des triangles croît en même temps. En second lieu, il est intéressant de noter que, de manière générale, les performances semblent assez peu varier en fonction de la taille des cuves. Cela est en partie dû au fait que les phases de clipping sont identiques pour toutes ces tailles de cuves, puisque la stratégie “globale” ne tient pas compte des propriétés des paquets de rayons.

Enfin, il semble surtout important de noter qu'il est difficile d'identifier une résolution optimale pour les grilles en perspective. En fonction de la taille des cuves et du nombre de rayons pour chaque cuve, le meilleur choix de résolution varie. Il ne semble même pas évident de définir une règle simple permettant de choisir la résolution des grilles en fonction des caractéristiques de la scène : ainsi, pour des cuves de grande taille et un grand nombre de rayons, le meilleur choix semble être la résolution $400 * 400$. Pour le même nombre de rayons, mais des cuves de petite taille, le meilleur choix devient la résolution $1000 * 1000$. Il est cependant impossible d'en déduire une résolution optimale pour chaque taille de cuve, puisque, lorsque le nombre de rayons diminue, la résolution optimale pour les cuves de petite taille devient là aussi $400 * 400$. Il est en fait normal que le choix de la résolution optimale semble aussi complexe : cette dernière sera principalement fonction du nombre de triangles intersectés par chaque faisceau de rayons, donc du nombre de tests d'intersections à effectuer pour chaque paquet. Hélas, déterminer à l'avance le nombre de tests d'intersections pour chaque faisceau revient en quelque sorte à effectuer une partie de l'algorithme de rastérisation des triangles. Le choix d'une résolution optimale, pouvant “s'adapter” à la scène, semble donc particulièrement difficile.

4.3.2 Stratégie “locale”

Afin de comparer nos deux stratégies de gestion de multiples paquets, nous effectuons maintenant les mêmes tests pour notre stratégie “locale”. Les performances obtenues sont rassemblées dans les figures 4.14, 4.15 et 4.16, respectivement associées aux cuves de grande, moyenne et petite taille. Nous avons ici choisi les résolutions en fonction du nombre de rayons par paquet, ce qui a semblé être une stratégie assez bonne. Là aussi, nous notons que les performances varient assez peu en fonction de la taille des cuves. Bien que les phases d'initialisation (en partie le clipping) soient en principe moins longues en présence de cuves de faible dimension, les optimisations que nous avons réalisées sur ces phases font que, pour des ordres de grandeur semblables en termes de nombres de triangles intersectés par un faisceau, les performances varient assez peu. Or, les cuves de notre maillage étant toutes maillées avec la même finesse, le nombre de triangles intersectés par chaque paquet, ainsi que le nombre d'intersections rayon-triangle observées, sont quasiment constants, quelles que soient les dimensions des cuves associées.

Par comparaison avec la stratégie “globale”, il est particulièrement intéressant de noter que le choix d'une résolution de grille optimale semble ici beaucoup plus aisé. De manière générale, une résolution de grille égale à la résolution de rayons du paquet associé a amené de bonnes performances. Cependant, il est assez évident qu'en augmentant artificiellement la finesse des maillages étudiés, nous pourrions dégrader les performances mesurées. La résolution de grille “parfaite” devrait tenir compte non seulement du nombre de rayons du paquet, mais aussi du nombre de triangles intersectés par le paquet. De telles heuristiques

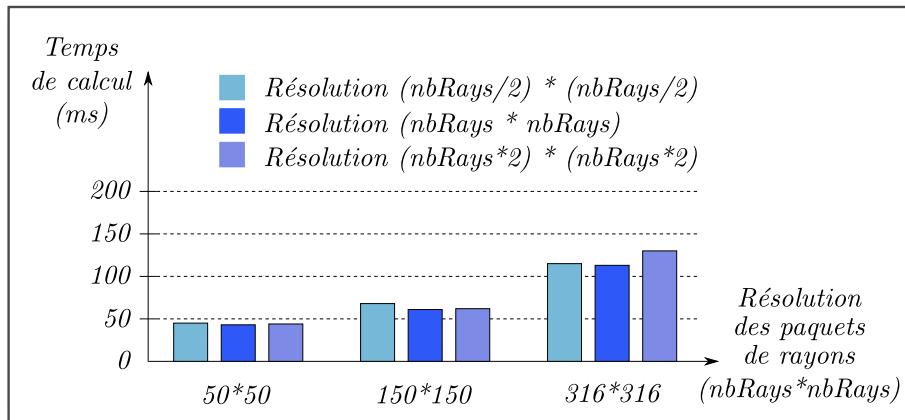


Figure 4.14 – Performances de la stratégie “locale” lors du traitement de 4 cuves de grande dimension. Pour chaque résolution de paquets de rayons, trois résolutions de grille, fonction de cette densité du paquet de rayons, sont testées.

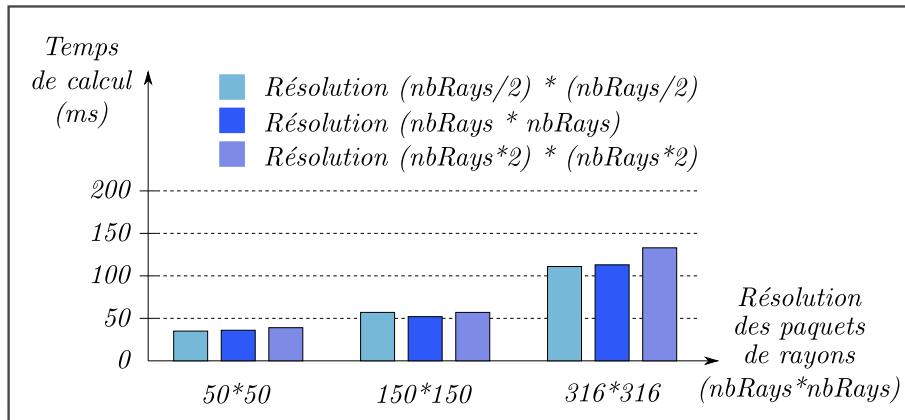


Figure 4.15 – Performances de la stratégie “locale” lors du traitement de 4 cuves de dimension “moyenne”. Pour chaque résolution de paquets de rayons, trois résolutions de grille, fonction de cette densité du paquet de rayons, sont testées.

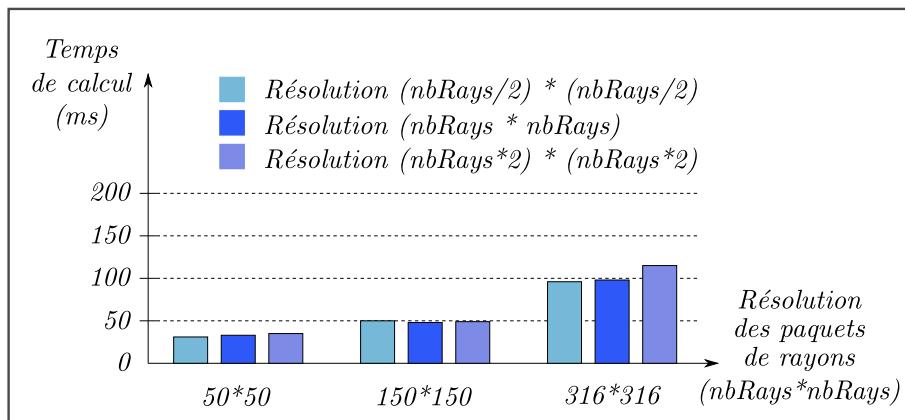


Figure 4.16 – Performances de la stratégie “locale” lors du traitement de 4 cuves de faible dimension. Pour chaque résolution de paquets de rayons, trois résolutions de grille, fonction de cette densité du paquet de rayons, sont testées.

sembleraient néanmoins aisées à mettre en défaut (avec un maillage très peu homogène, par exemple). Nous verrons à la fin du chapitre (voir partie 4.4) que le fait de réellement tenir compte des particularités du maillage ou de la répartition des rayons demande l'utilisation d'une structure d'accélération hiérarchique, ce que n'est pas notre grille régulière.

4.3.3 Comparaison des deux stratégies

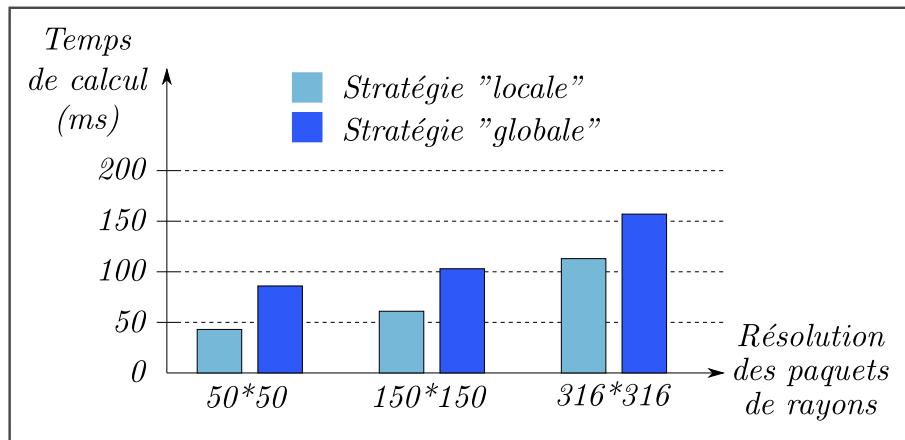


Figure 4.17 – Comparaison des stratégies “globale” et “locale” lors du traitement de 4 cuves de grande dimension. Pour chaque résolution du paquet de rayons, la meilleure performance observée pour chaque stratégie (voir figures 4.11 et 4.14) est affichée.

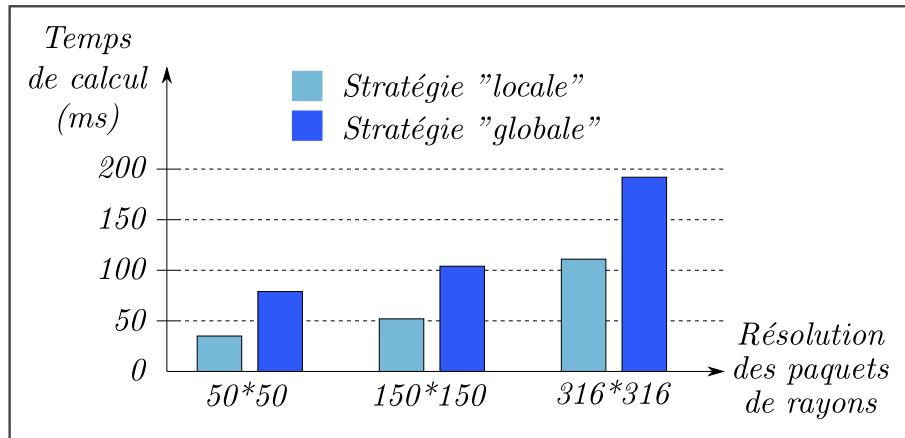


Figure 4.18 – Comparaison des stratégies “globale” et “locale” lors du traitement de 4 cuves de dimension “moyenne”. Pour chaque résolution du paquet de rayons, la meilleure performance observée pour chaque stratégie (voir figures 4.12 et 4.15) est affichée.

Après ces premières analyses succinctes, nous pouvons désormais étudier les performances comparées de ces deux stratégies. Sur les figures 4.17, 4.18 et 4.19, nous avons affiché, pour chacune de nos neuf configurations différentes, le meilleur temps obtenu par chacune des stratégies en faisant varier la résolution des grilles en perspective utilisées. Pour chacune des neuf configurations traitées, la stratégie “locale” est meilleure. Afin d'expliquer ces différences, nous avons choisi de mesurer plusieurs données supplémentaires, en nous concentrant sur

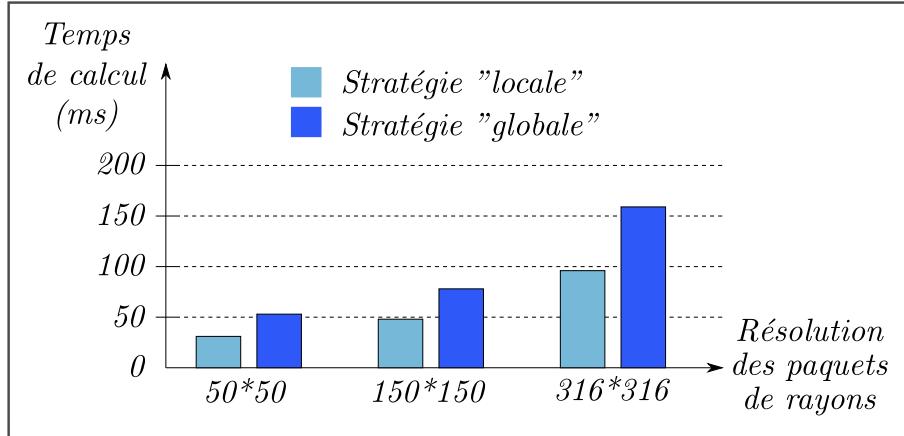


Figure 4.19 – Comparaison des stratégies “globale” et “locale” lors du traitement de 4 cuves de faible dimension. Pour chaque résolution du paquet de rayons, la meilleure performance observée pour chaque stratégie (voir figures 4.13 et 4.16) est affichée.

Dimension des cuves	nbTrgs Stratégie “globale”	nbTrgs Stratégie “locale”
Grandes	759 K	182 K
Moyennes	693 K	42 K
Petites	238 K	18 K

Tableau 4.6 – Comparaison du nombre total de triangles traités par chaque stratégie, lors du traitement de 4 cuves. Chaque ligne du tableau correspond à une dimension différente de cuves.

le cas des fortes densités de rayons (paquets de 100 000 rayons chacun), car ces analyses resteraient valables pour les autres densités.

Pour dresser le tableau 4.6, nous avons mesuré, pour chaque taille de cuve associée à ces paquets et pour la résolution optimale associée à chaque cas, le nombre total de triangles traités par chaque stratégie, après clipping (les triangles traités par plusieurs grilles étant comptés deux fois). Sans surprise, le nombre de triangles traités par la stratégie “globale” est très nettement supérieur à celui observé pour la méthode concurrente. Dans le cas de la méthode “locale”, le clipping des triangles est en effet appliqué à des portions de l'espace beaucoup plus restreintes. Afin de réduire ce phénomène, nous avons mis en place une stratégie permettant de n'effectuer aucun traitement pour une des six grilles de la stratégie “globale”, lorsque cette dernière ne rencontre aucun rayon. Néanmoins, comme nous le voyons sur ce tableau, cette stratégie n'empêche pas un très net écart entre les deux stratégies. Cette différence entre les nombres de triangles traités par chaque stratégie entraîne des temps d'initialisation bien plus importants dans le cadre de la stratégie “globale”. Nous verrons plus loin qu'en augmentant le nombre de paquets, il est naturellement possible de diminuer cet écart, voire d'inverser le rapport de force entre ces deux stratégies.

Cet écart entre le nombre de triangles traités par chaque stratégie n'a pas seulement un impact lors de la phase d'initialisation des calculs. En effet, il implique également que lors de la phase de rastérisation, une très grande quantité de triangles va être étudiée, la plupart n'étant pourtant pas associés au moindre paquet de rayons. Afin de mesurer l'impact de ce travail inutile, nous avons mesuré, pour chaque cas-test, le nombre de cellules de grilles “vides” de rayons. Ces cellules n'étant associées à aucun rayon, nous devrions normalement abandonner tous les calculs qui leur sont associés. Au cours de la rastérisation, il est très difficile de vérifier si chaque cellule intersectée par un triangle contient ou non des rayons

Cuves	nbCouples “inutiles” Strat. “globale”	nbCouples “inutiles” Strat. “locale”
Grandes	3.8M (89 %)	1.6M (56 %)
Moyennes	15.8M (97 %)	1.2M (40 %)
Petites	18.9M (98 %)	0.8M (34 %)

Tableau 4.7 – Comparaison du nombre total de couples triangle-cellule “inutiles” générés par chaque stratégie, lors du traitement de 4 cuves. Chaque ligne du tableau correspond à une dimension différente de cuves. Pour chaque colonne est indiqué, entre parenthèses, le pourcentage de ces couples triangle-cellule “inutiles” parmi l’ensemble de couples triangle-cellule générés après rastérisation des triangles.

sans fortement ralentir l’exécution du programme sur le GPU. Bien que cette stratégie semble très inadaptée, il est en réalité plus efficace d’effectuer la rastérisation sans tenir compte de la grille de rayons construite auparavant. Cependant, cette stratégie génère forcément un grand nombre de couples triangle-cellule parfaitement inutiles, car n’étant associés à aucun rayon. Sur le tableau 4.7, nous avons mesuré le nombre total de ces couples triangle-cellule “inutiles”, pour chaque stratégie, tout en indiquant le pourcentage de tels couples parmi l’ensemble des couples générés. Là encore, nous voyons que la stratégie “globale” souffre de lacunes importantes, dues, une fois de plus, à son manque d’adaptativité à la géométrie de la scène. Cependant, nous notons aussi au passage que la proportion de couples inutiles est également très importante pour la stratégie “locale”. Cette statistique indique sans doute que les grilles en perspective associées à chaque faisceau de rayons couvrent un espace trop grand autour du faisceau. Il serait intéressant de revoir la stratégie de création de ces grilles en perspective, basées sur les boîtes englobantes des faisceaux de rayons dans l’espace global 3D associé à la scène.

Dimension des cuves	nbTests Strat. “globale”	nbTests Strat. “locale”
Grandes	8.9 M	6.6 M
Moyennes	13.1 M	7.0 M
Petites	11.0 M	5.5 M

Tableau 4.8 – Comparaison du nombre de tests d’intersection rayon-triangle effectués en fonction de la stratégie choisie. Chaque ligne du tableau correspond à une dimension différente de cuves.

Enfin, la stratégie “globale” génère également un nombre de tests d’intersections beaucoup trop important, là où, idéalement, nous souhaiterions ne réaliser des tests d’intersections que pour des couples rayon-triangle réellement intersectant. Or, en présence de paquets de rayons de faible volume, mais à forte concentration de rayons, les cellules des six grilles en perspective de la grille deviennent bien trop larges pour couvrir efficacement la finesse des faisceaux de rayons. Chaque cellule contient alors un nombre de rayons beaucoup trop important et génère un nombre d’intersections bien trop élevé. Cette affirmation peut être vérifiée à la lecture du tableau 4.8, répertoriant le nombre de tests d’intersections effectués pour chaque stratégie.

Il ressort de cette première analyse que la stratégie “globale” n’est pas adaptée au traitement de paquets de faible volume, surtout lorsque ces derniers contiennent un nombre important de rayons. Néanmoins, comme nous allons le voir maintenant, la stratégie “locale” souffre également de défauts importants lorsque le nombre de paquets de rayons à l’intérieur d’une scène augmente.

4.3.4 Augmentation du nombre de paquets

Comme nous l'avons déjà dit, la stratégie “locale” présente l'avantage de pouvoir adapter les structures d'accélération utilisées à la structure de la scène. Cependant, la multiplication des paquets de rayons provoque naturellement une multiplication des grilles en perspective et une perte d'efficacité, puisqu'aucun calcul de projection ou de clipping n'est mis en commun entre les différents paquets de la scène. Afin de vérifier cette mauvaise gestion de l'augmentation du nombre de paquets au sein d'une scène, nous avons étudié l'évolution des performances lors du traitement de 8 cuves de taille moyenne et de 12 cuves de petite taille, présentes dans notre scène. Les performances obtenues sont visibles sur les figures 4.20 et 4.21, correspondant à chacun de ces cas. Nous affichons ici directement la meilleure performance réalisée en faisant varier la résolution de grille utilisée.

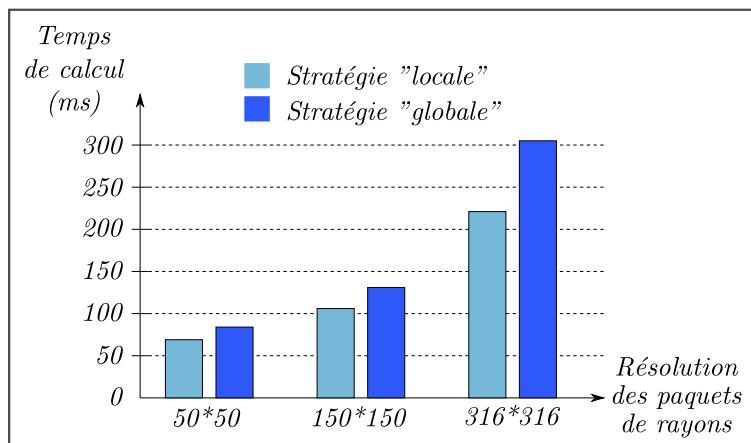


Figure 4.20 – Comparaison des stratégies “globale” et “locale” lors du traitement de 8 cuves de dimension “moyenne”. Pour chaque résolution du paquet de rayons, la meilleure performance observée pour chaque stratégie est affichée.

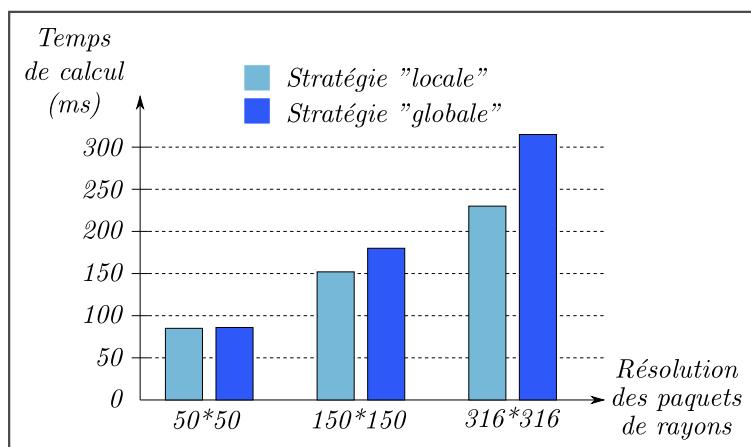


Figure 4.21 – Comparaison des stratégies “globale” et “locale” lors du traitement de 12 cuves de faible dimension. Pour chaque résolution du paquet de rayons, la meilleure performance observée pour chaque stratégie est affichée.

Avant toute chose, il est important de noter que le traitement de ces cuves a demandé

nbCuves	nbInters	Stratégie “globale”		Stratégie “locale”	
		temps	ratio	temps	ratio
4	4.0M	159	25.2	96	41.7
	911K	78	11.7	48	19.0
	101K	53	1.9	31	3.3
12	13.5M	315	42.8	230	58.7
	3053K	180	17.0	152	20.1
	339K	86	3.9	85	3.9

Tableau 4.9 – Évolution du ratio nbIntersections/seconde avec la diminution de la densité des paquets de rayons, pour des cuves de faible dimension. Pour chaque nombre de cuves, les trois lignes correspondent à une densité de rayons différente (100 000 rayons pour la première ligne, puis 22 500 et 2 500). Les colonnes *temps* indiquent le temps nécessaire à l'exécution de l'algorithme global de calcul d'intersections (excepté le temps de tri/transfert des intersections vers le CPU). Ces temps sont indiqués en millisecondes. Les colonnes *ratio* expriment, pour chaque stratégie, le nombre de millions d'intersections traités par seconde.

plusieurs passes (jusqu'à 3) pour la stratégie “globale” : en effet, avec la stratégie “globale”, il arrive qu'une des six grilles de la scène concentre un nombre important de faisceaux, donc un nombre très important de triangles traités, de couples triangle-cellule générés et de tests d'intersections exécutés (d'autant plus en regard des caractéristiques que nous avons détaillées dans la partie précédente). Cette observation met ainsi en évidence l'un des grands défauts de la stratégie “globale” et sa faible résistance à une “montée en charge”, lorsque les quantités de rayons par paquets sont importantes.

En dehors de cette affirmation, il est surtout essentiel de noter que, pour le cas de paquets de rayons à faible densité, la stratégie “locale” perd vite en efficacité devant l'augmentation du nombre de paquets, et devient même équivalente pour le cas des 12 cuves de faible dimension. Il ne fait aucun doute qu'en augmentant davantage ce nombre de cuves, la stratégie “globale” prendrait rapidement un avantage important sur la stratégie “locale”. Cependant, la stratégie “globale” semble elle aussi fortement souffrir de la diminution du nombre de paquets par cuve.

Afin de vérifier ce mauvais comportement de nos deux stratégies, nous avons mesuré, pour le traitement des 4 petites cuves, l'évolution du nombre de millions d'intersections gérées par seconde, lorsque le nombre de rayons par cuve diminue. Ces mesures sont reportées dans le haut du tableau 4.9. Comme on pouvait s'y attendre, on observe, pour chacune des deux stratégies, une diminution très importante de la performance lorsque le nombre de paquets diminue. Nous avons également mesuré cette évolution lors du traitement de 12 petites cuves (voir bas du tableau). Là encore, la dégradation des performances observée est très importante. Il est cependant intéressant de comparer ces deux parties de tableau : tout d'abord, de manière assez surprenante, on note que la stratégie “locale” progresse clairement entre le traitement de 4 et de 12 cuves, pour le cas de paquets à forte résolution : cette évolution positive ne peut être due qu'au fait que les 8 cuves ajoutées entre les deux tests sont plus “faciles” à traiter que les quatre premières (meilleure homogénéité des cellules, taux plus faible de couples triangle-cellule inutiles...), et résulte donc de configurations spatiales particulières. Cependant, toujours dans le cas de la stratégie “locale”, l'évolution des performances pour les cuves de moyenne et faible densité est quasiment imperceptible entre les deux tableaux (ce qui est plus logique, car le traitement des différents paquets se fait de manière indépendante, et la phase de tri des intersections, qui permettrait une mise en commun de certains calculs, n'est ici pas prise en compte). Dans le même temps, lors de l'augmentation du nombre de paquets, les performances semblent, comme on l'attendait, évoluer de façon nettement plus favorable

cuves	nbTests	nbIntersections	init	lancer Trgs	tri-transfert	total
4 grandes	7.9M	3.7M (47.4 %)	39.3	83.0	31.4	153.7
	2.0M	841K (42.5 %)	35.1	30.4	9.0	74.5
	449K	93.6K (20.8 %)	31.5	15.1	3.2	49.8
8 moyennes	3M	8.6M (53.1 %)	65.2	172.6	69.5	307.3
	1.0M	1.9M (48.4 %)	52.5	61.6	16.8	130.9
	799K	85K (27.1 %)	51.1	27.5	5.2	83.8
12 petites	24.9M	13.6M (54.6 %)	97.0	366.9	107.0	474.0
	6.0M	3.1M (50.8 %)	80.9	102.4	26.8	210.1
	1.1M	339K (31.7 %)	72.4	40.8	6.2	119.4

Tableau 4.10 – Performances globales de la stratégie “locale”. Pour chaque configuration de cuves, les trois lignes représentent une variation de la densité de la cuve (identique à celle du tableau 4.9). Dans la colonne *nbIntersections* est indiqué entre parenthèses le pourcentage de tests d’intersections rayon/triangle réussis (renvoyant *true*). La colonne *init* indique le temps nécessaire à la phase totale d’initialisation (import des rayons + clipping + projection des triangles + construction de la grille). La colonne *lancerTrgs* indique le temps d’exécution de l’algorithme de lancer de triangles, privé de la phase de tri/transfert des intersections vers le CPU. Tous les temps sont donnés en millisecondes.

pour la stratégie “globale”. Au fur et à mesure de cette augmentation, la stratégie “globale” bénéficie d’un amortissement du temps d’initialisation commun à tous les paquets, et atteint logiquement des performances plus intéressantes. En présence de faisceaux de rayons à forte densité, en revanche, cette stratégie souffre trop de sa structure non-adaptative pour qu’elle puisse rivaliser rapidement en termes de performance avec la seconde stratégie.

Au final, il ressort donc de ces tests que, malgré ses qualités supérieures, la solution consistant à construire une grille en perspective par paquet de rayons n’est pas sans défauts et souffre de forts problèmes de facteurs d’échelle.

4.3.5 Performances globales de la stratégie “locale”

Nous souhaitons néanmoins, pour finir cette partie, mesurer les performances de l’algorithme global de lancer de rayons exhaustif, qui consiste donc en l’identification, le tri puis le transfert vers le CPU de l’ensemble des intersections ayant lieu entre les rayons et les triangles de la scène. Nous avons pour cela mesuré le temps d’exécution global de notre algorithme, via l’utilisation de la stratégie “locale”, lors du traitement de 4 cuves de grande taille, 8 de taille moyenne et 12 de faible dimension (voir tableau 4.10). Le temps total d’exécution de l’algorithme est séparé en trois phases : la phase d’initialisation, celle de lancer de triangles présentée en 2.3 et celle de tri et de transfert des intersections trouvées vers le CPU. Pour évaluer les temps d’initialisation et de lancer de triangles, nous sommes les durées nécessaires à la réalisation de ces phases pour chaque grille en perspective traitée. Il est important de noter que le temps d’initialisation tient ici compte des différentes étapes présentées en 4.2.1, mais aussi des temps de projection des triangles dans l’espace à deux dimensions associé à chaque grille. Nous intégrons également à ce temps l’impact de la phase de transfert des rayons du CPU vers le GPU, puisque nous transférons ces rayons au fur et à mesure du traitement des différentes cuves.

Une première observation à la lecture de ce tableau concerne la proportion de tests d’intersection rayon-triangle réussis : en dehors des paquets de rayons de faible résolution, cette proportion est très bonne, de l’ordre de 50 %. Cela signifie que la moitié des couples rayon-triangle dont nous testons l’intersection s’intersectent réellement, ce qui est une statistique

bien meilleure que lors de l'utilisation d'un BVH traditionnel (à titre d'exemple, sur nos propres cas-tests, nous observons plutôt un ratio de l'ordre de 10 % pour des scènes imposantes). La valeur de ce ratio baisse cependant de manière sensible lorsque le nombre de rayons par paquet est plus réduit : dans ce cas, les résolutions de grille utilisées deviennent trop faibles par rapport aux dimensions propres du maillage de la scène, et les cases de la grille contiennent donc alors un trop grand nombre de rayons. En augmentant la résolution de nos grilles, nous pourrions faire croître ce ratio, mais nous augmenterions aussi le temps nécessaire à la rastérisation des triangles, et obtiendrions ainsi des performances de moins bonne qualité.

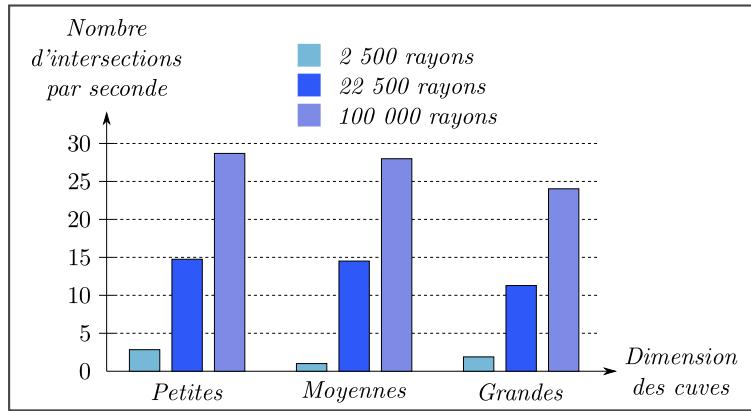


Figure 4.22 – Évolution du ratio nbIntersections/seconde en fonction de la densité des paquets de rayons, lors du traitement de 12 cuves de faible dimension, 8 cuves de dimension “moyenne”, et 4 cuves de grande taille. Pour chacune de ces configurations, nous faisons varier la densité de la cuve et mesurons le nombre d’intersections par seconde que traite l’algorithme.

L'autre observation naturelle à la lecture de ce tableau concerne, une fois encore, les différences assez faibles entre les performances obtenues pour les différentes densités de rayons, compte tenu des écarts entre les différentes résolutions des paquets de rayons lancés (n'oublions pas l'existence d'un rapport 10, par exemple, entre le nombre de rayons dans les cuves de moyenne densité et celles de faible densité, tandis que les différences entre les performances globales constatées sont bien moins importantes). Nous observons surtout, une fois de plus, un écart très faible entre les durées des différentes phases d'initialisation lorsque le nombre de rayons par paquet varie. Afin de quantifier une dernière fois cette perte d'efficacité au fur et à mesure de la raréfaction des rayons, nous avons mesuré le nombre d'intersections traitées par seconde, en tenant donc compte, cette fois, de l'intégralité de l'algorithme (voir figure 4.22). Les performances sont à nouveau très satisfaisantes pour les fortes résolutions de rayons (près de 25 millions d'intersections par seconde), mais elles deviennent moins bonnes (entre 10 et 15 millions d'intersections par seconde) pour des résolutions moyennes et très mauvaises (inférieur à 3 millions d'intersections par seconde) pour de faibles densités. Nous confirmons donc bien ici le comportement variable de la stratégie “locale” en fonction de la densité des paquets de rayons.

Avant d'étudier dans la dernière partie de ce chapitre les diverses pistes de travail qui permettraient de mieux gérer des scènes hétérogènes, comprenant un grand nombre de paquets, il convient de noter que les performances observées pour des paquets de rayons très denses constituent une avancée importante dans la perspective du calcul de débit de dose interactif. En effet, le fait de traiter très rapidement d'importants paquets de rayons pourrait permettre d'utiliser des maillages beaucoup plus fins pour échantillonner les sources volumiques de radiations. En effet, dans bien des cas, les utilisateurs de logiciels de calcul de

débit de dose limitent la finesse de ces échantillonnages afin d'obtenir des résultats plus rapides, mais perdent ainsi en précision du point de vue de l'évaluation du débit de dose. Les performances que nous obtenons pourraient donc permettre d'augmenter ce niveau de précision, tout en conservant des temps de calcul interactifs. De plus, il est assez fréquent que les scènes étudiées lors d'études de radioprotection ne présentent qu'un nombre très limité de sources de radiations, ce qui constitue le meilleur cas d'utilisation de nos algorithmes.

Néanmoins, malgré ces perspectives positives, il convient également d'envisager des solutions au traitement de paquets de rayons plus nombreux (et possiblement moins denses), représentant également certaines configurations de scènes irradiées.

4.4 Discussion

À la lueur des défauts propres à nos deux stratégies vis-à-vis du traitement de scènes hétérogènes, contenant un grand nombre de sources de radiations, nous souhaitons maintenant, dans cette dernière partie, étudier différentes pistes de travail qui pourraient nous permettre d'améliorer les performances actuellement obtenues.

4.4.1 Optimisations supplémentaires de l'algorithme actuel

En premier lieu, penchons-nous sur les différentes manières d'optimiser davantage nos méthodes actuelles. Nous avons déjà expliqué (voir [2.5.1](#)) que nous pensions pouvoir sensiblement diminuer le temps de construction de la grille en perspective indexant les rayons de la scène. Nous avons également évoqué dans la suite de la partie [2.5](#) les différentes optimisations qui pourraient nous permettre de réduire les temps de parcours de ces grilles en perspective.

Cependant, nous ne pensons pas que l'optimisation de ces deux phases puisse nous permettre de modifier fondamentalement l'ordre de grandeur de nos performances. En effet, comme nous l'avons vu, le traitement de multiples paquets de rayons est particulièrement freiné par la multiplication des phases d'initialisation associées à chaque grille en perspective. Une première manière d'améliorer nos temps de calcul serait par conséquent d'accélérer cette première phase de traitement. Comme nous l'avons vu, la majeure partie de ce temps d'initialisation est dédiée à la création des vecteurs utilisés lors de la phase de clipping, puis à cette phase de clipping. Pour la première étape, nous pensons possible une meilleure utilisation de la mémoire partagée, afin de rendre les accès mémoire effectués plus cohérents. Nous croyons également, comme nous l'avons expliqué en [4.2.2](#), que les performances de l'algorithme de clipping peuvent encore être améliorées, via une répartition plus homogène des calculs entre les différents threads.

La deuxième phase de l'algorithme qui mérirait, selon nous, le plus d'attention lors d'implémentations futures, est celle du traitement final des intersections (tri + transfert des résultats vers le CPU). Comme nous l'avons vu, cette phase de travail a des conséquences importantes sur le temps d'exécution final de l'algorithme. De plus, le temps nécessaire à son exécution dépend fortement du nombre d'intersections trouvées et nous semble, pour l'instant, trop sensible aux variations de ce facteur. Dans le cadre d'une implémentation future, nous pensons donc que les phases de tri et de transfert des données d'intersections vers le CPU devraient pouvoir être largement optimisées. Nous avons expliqué dans ce chapitre (voir [4.2.4](#)) la mise en place d'une astuce nous permettant d'effectuer le tri des indices de rayon et des coordonnées de profondeur en une seule étape. Néanmoins, nous croyons davantage au potentiel de la solution de tri en deux étapes. Tout d'abord, le premier tri effectué sur les intersections (suivant les indices de rayons) devrait pouvoir bénéficier d'optimisations

semblables à celles présentées en 2.5.1, concernant le tri réalisé lors de la construction de la grille en perspective. Dans un second temps, le tri des intersections suivant les coordonnées de profondeur devrait, lui aussi, pouvoir être accéléré, via une exploitation intelligente de la mémoire partagée et une meilleure mutualisation des calculs entre les différents threads travaillant sur le même indice de rayon.

Enfin, la phase de transfert des données d'intersections vers le CPU devrait pouvoir être quasiment réduite à néant, grâce à une meilleure gestion des transferts de mémoire asynchrones rendus possibles sur les cartes graphiques de dernière génération. En effet, nous utilisons déjà aujourd'hui ces transferts asynchrones, mais les calculs effectués dans le même temps sur le GPU (consistant à réordonner les indices associés à chaque intersection, en fonction de la modification des positions induite par les étapes de tri) sont nettement moins coûteux et ne peuvent donc pas complètement cacher les coûts de transfert vers le CPU. Il serait certainement possible d'effectuer le transfert de données vers le CPU tandis que le traitement pour un nouveau pas de temps commence. Cela entraînerait une légère latence dans le calcul du débit de dose, mais permettrait une fréquence d'évaluation du débit de dose plus élevée.

La somme de ces optimisations permettrait donc sans doute de réduire nos temps de calcul actuels. Cependant, des gains de temps plus importants pourraient peut-être être réalisés via l'utilisation de structures d'accélération mieux adaptées à la structure géométrique non régulière des scènes 3D.

4.4.2 Vers une structure hiérarchique ?

En effet, même en améliorant les performances de nos algorithmes actuels, nous risquons de continuer à nous heurter à des difficultés intrinsèques aux méthodes que nous utilisons, en particulier à l'usage de grilles régulières comme structures d'accélération. Comme nous l'avons vu, la solution “locale”, si elle apporte des résultats satisfaisants, ne semble pas adaptée à une augmentation importante du nombre de paquets de rayons. De plus, cette solution ne permet pas de mettre en commun les calculs associés à plusieurs faisceaux de rayons se “chevauchant” (voir figure 4.23). On pourrait dès lors imaginer certaines stratégies permettant de n'exploiter qu'une seule grille pour deux tels faisceaux. Cependant, dans le cas où les deux faisceaux présenteraient des densités très différentes, quelle résolution devrait être choisie ? De plus, en conservant une forme de grille “carrée” dans l'espace en perspective, on risquerait à nouveau de générer de nombreuses cellules vides de rayons, comme on peut déjà l'observer sur la figure. Faudrait-il alors autoriser des formes moins régulières dans l'espace en perspective ? Pour le cas de la figure 4.23, la meilleure solution ne serait-elle pas plutôt de générer plusieurs grilles, celle associée à l'espace de recouvrement des deux faisceaux, par exemple, présentant la résolution la plus fine des deux ? Enfin, la stratégie “locale” réalise un traitement peu satisfaisant dans le cas des objets recouvrant un grand espace autour du centre de perspective de la scène (revoir figure 4.5 pour un tel objet).

Toutes ces réflexions suggèrent que la stratégie “locale”, si elle permet de facilement mettre en valeur nos algorithmes de construction et de parcours d'une grille en perspective, n'est sans doute pas la meilleure solution pour la gestion de scènes complexes et hétérogènes, avec de très nombreux paquets de rayons (de faible dimension) au sein de la scène. La solution “globale”, consistant à utiliser six grilles couvrant l'ensemble de l'espace autour du point de mesure du débit de dose, présente l'avantage de mieux supporter une “montée en charge”, mais souffre de défauts inhérents à la structure régulière des grilles choisies.

Dans le cadre d'implémentations futures, il nous semblerait possible de mettre en commun les qualités de ces deux stratégies, tout en réduisant leurs défauts. La solution nous paraissant

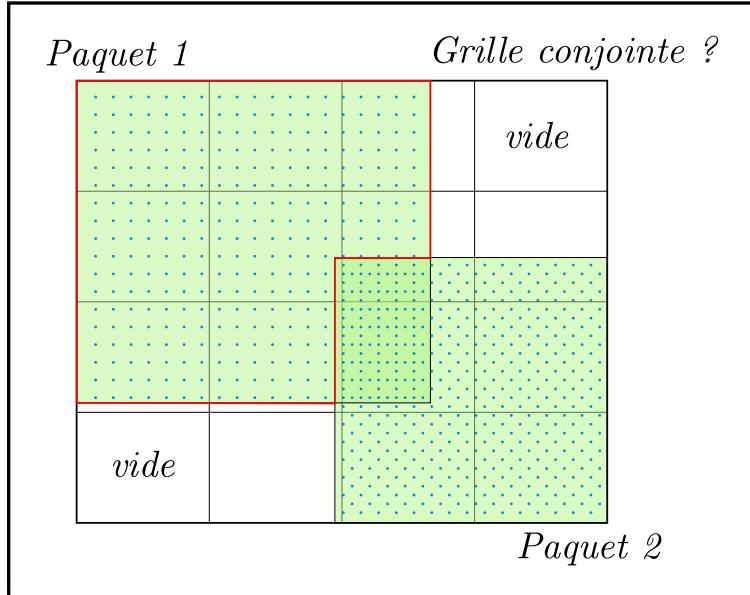


Figure 4.23 – Exemple de deux paquets de rayons se “chevauchant”, vus du point de mesure du débit de dose. Une grille conjointe, encadrant les deux paquets, est proposée. Deux cellules de cette grille conjointes sont vides, comme indiqué sur le schéma. Une grille de forme “non carrée”, encadrant uniquement les rayons du paquet 1 ne rencontrant pas le paquet 2, est également proposée (contour en rouge sur le schéma).

aujourd’hui la plus adaptée consisterait à exploiter la stratégie “globale”, mais en utilisant, pour couvrir tout l’espace autour du point de mesure du débit de dose, six structures d’accélération plus adaptatives que les grilles régulières. La grille à deux niveaux de détails, très récemment proposée par Kalojanov [KBS11], pourrait sembler une solution intéressante, car elle présente des temps de construction très faibles sur GPU et pourrait facilement bénéficier des algorithmes de construction et de parcours de grille que nous avons déjà développés. Des structures plus adaptatives, telles le kd-tree ou le BVH, pourraient même être envisagées. En effet, le fait d’exploiter ces structures dans un espace 2D pourrait simplifier grandement les étapes de construction et de parcours associées à ces structures.

Comme pour nos grilles régulières, il serait sans doute, là aussi, intéressant de construire des BVHs ou des kd-trees indexant les rayons, et non les triangles de la scène. En effet, dans notre contexte d’utilisation, nous avons la chance de connaître, dès le début du calcul de débit de dose associé à un point de mesure, la position des rayons de la scène. Il nous est donc possible d’identifier dès le début des calculs un grand nombre de triangles ne devant pas être traités, car n’intersectant aucune enveloppe associée à un faisceau de rayons. Cet argument nous a déjà en partie amené à privilégier la stratégie “locale” par rapport à la stratégie “globale”. De la même manière, le fait, par exemple, de construire, dans l’espace en perspective, un BVH indexant les rayons, pourrait nous permettre d’éliminer, dès les premières étapes de parcours de la structure d’accélération, un grand nombre de triangles de la scène n’intersectant aucun des noeuds fils de la racine de l’arbre BVH. Dans sa proposition d’algorithme de rastérisation irrégulière, Aila [AL04] proposait d’ailleurs déjà l’utilisation d’un kd-tree répertoriant les rayons de la scène. Notons au passage que l’utilisation de structures d’accélération hiérarchiques indexant les rayons de la scène pourrait également nous permettre de mieux traiter des paquets de rayons présentant une distribution de rayons plus hétérogène que celles que nous avons manipulées dans ce manuscrit. En effet, comme nous l’avons vu en 1.1.3, le calcul de débit de dose lors de l’émission de radiations par une source

volumique se fait via plusieurs itérations successives. Pour chaque nouvelle itération, la répartition des sources ponctuelles générées aléatoirement varie par rapport à l'étape précédente, et peut devenir assez irrégulière au fil des itérations si certaines parties d'une source de radiations participent beaucoup plus à l'émission de dose que d'autres. Le fait d'utiliser des structures permettant de manipuler des distributions hétérogènes de rayons pourrait donc être particulièrement intéressant dans de tels cas de figure.

De plus, dans l'espace en perspective, l'expression d'un rayon se réduit à un simple point en deux dimensions, ce qui devrait permettre une construction de BVH ou de kd-tree bien plus rapide que les habituelles structures dans l'espace classique à trois dimensions. En particulier, pour le cas du kd-tree, cette propriété nous permettrait d'avoir l'assurance qu'un rayon ne pourrait pas appartenir à deux fils différents d'un même noeud. Or, comme nous l'avons vu, le fait qu'habituellement, un même triangle puisse être référencé plusieurs fois dans un kd-tree constitue un défaut majeur de ces structures et une des raisons de la difficulté à construire des kd-trees sur GPU (en comparaison des BVHs). De telles structures, "orientées rayons", demanderaient néanmoins un parcours de la structure d'accélération par les triangles de la scène a priori plus complexe, puisqu'il nécessiterait la répétition de tests d'intersections triangle-boîte dans l'espace en perspective.

On peut dès lors se demander si les temps de construction et de parcours cumulés de ces six structures d'accélération ne seraient pas en réalité trop importants, nous menant à des performances moins bonnes que celles que nous connaissons aujourd'hui. Certes, il resterait possible de revenir à la construction et au parcours d'une seule structure d'accélération en trois dimensions, mais nous risquerions alors de perdre de nombreuses optimisations rendues possibles par nos calculs en deux dimensions (les temps de traversée pouvant redevenir au passage bien plus longs), ainsi que la gestion optimisée des problèmes de précision que nous avons présentée dans le chapitre 3.

Conclusion

Nous avons présenté dans ce chapitre deux stratégies différentes de traitement de multiples paquets de rayons, reposant sur les algorithmes de gestion de grilles en perspective présentés dans les deux chapitres précédents. À la stratégie dite "globale", consistant en la construction de six grilles différentes afin de couvrir l'intégralité de l'espace autour du point de mesure du débit de dose, nous avons préféré la construction d'une grille par paquet de rayons, qui nous permet de mieux tenir compte du volume et de la densité de chaque paquet de rayons au sein de la scène. Grâce à cette dernière stratégie, nous avons pu garantir le traitement de multiples paquets de rayons volumineux, au sein d'un modèle 3D complexe, de manière interactive.

Nous avons cependant observé que cette méthode était mal adaptée au traitement de nombreux paquets de rayons, en raison d'une phase d'initialisation propre à chaque grille assez lourde en calculs. Afin d'obtenir un algorithme résistant mieux aux différents facteurs d'échelle au sein d'une scène, nous avons donc présenté plusieurs pistes de travail pour des implémentations futures, en particulier le remplacement des six grilles régulières permettant de couvrir l'espace par six structures d'accélération hiérarchiques, comme le kd-tree ou le BVH. Néanmoins, nous avons établi que l'utilisation de telles techniques soulevait d'autres problèmes, la construction et le parcours de telles structures demandant a priori des algorithmes plus complexes que dans le cas d'une grille régulière. Bien que cette solution nous paraisse présenter le meilleur potentiel, le problème du traitement optimal de multiples paquets de

rayons reste donc ouvert et mériterait qu'on s'y attarde davantage lors d'expérimentations futures.

Conclusion et Perspectives

NOUS avons présenté dans ce manuscrit plusieurs algorithmes permettant d'accélérer le calcul de débit de dose via la méthode d'atténuation en ligne droite avec facteurs d'accumulation. Dans ce but, nous nous sommes concentrés sur l'optimisation des calculs géométriques nécessaires à l'utilisation de cette méthode.

Nous avons d'abord réalisé une synthèse de l'état de l'art qui nous a permis de dégager notre problématique et d'introduire les outils utilisés dans le reste du manuscrit. En premier lieu, nous avons présenté le domaine de l'ingénierie de la radioprotection et la méthode d'atténuation en ligne droite avec facteurs d'accumulation, couramment utilisée dans le domaine afin de concevoir certaines installations ou pour préparer des interventions sur des sites existants. Cette introduction nous a permis de dégager les caractéristiques du problème de lancer de rayons que nous avons tenté de résoudre dans ce manuscrit : l'objectif de nos travaux était donc de permettre le calcul d'intersections interactif entre de multiples paquets de rayons (représentant les radiations) et une scène 3D de taille importante (plusieurs centaines de milliers de triangles) possiblement en mouvement et pouvant même subir des modifications de sa topologie au cours du temps. Nous avons alors noté l'existence de deux caractéristiques propres à notre problème de lancer de rayons, imposées par la méthode de calcul de débit de dose employée : en premier lieu, pour chaque rayon, l'ensemble des intersections le long de ce rayon devaient être trouvées, puis triées et mémorisées, de manière à simuler convenablement la propagation des rayonnements γ dans la matière. En outre, nous avons observé que, lors du calcul de débit de dose en un seul point de la scène, l'ensemble des rayons considérés convergeaient en ce point de calcul, propriété laissant la place à de nombreuses optimisations potentielles.

Après cette présentation, nous avons donc cherché à étudier les pistes de travail proposées par la littérature pour résoudre notre problème. En raison de la nature intrinsèquement parallèle des requêtes d'intersections effectuées, nous avons d'abord souhaité étudier l'architecture hautement parallèle des processeurs de cartes graphiques (GPUs), actuellement en plein essor. Dans le but de tirer profit de cette architecture performante, nous avons d'abord étudié les différentes possibilités permettant de contourner le comportement traditionnel du pipeline graphique afin de répondre à nos besoins. Nous avons cependant déduit de cette étude que l'irrégularité de nos paquets de rayons, ainsi que le besoin de stocker l'intégralité des intersections pour chaque rayon de la scène, nous empêchaient de bénéficier des implantations matérielles de l'algorithme de rastérisation sur carte graphique. Nous avons donc choisi de développer nos propres algorithmes pour le GPU, via l'interface de programmation CUDA, dédiée aux cartes de la marque NVIDIA.

Une fois ce choix réalisé, nous avons effectué une synthèse approfondie de l'état de l'art dans le domaine du lancer de rayons. Cette étude nous a permis de présenter les nombreuses avancées réalisées au cours des dix dernières années dans le domaine, via la création d'algorithmes innovants, et une exploitation optimale du matériel utilisé. Nous avons ainsi présenté le BVH et le kd-tree, structures d'accélération abondamment utilisées dans le domaine, ayant récemment permis d'obtenir des performances interactives lors du traitement de scènes statiques ou dynamiques. Nous avons cependant noté que ces deux structures n'étaient pas

adaptées au parcours des rayons que nous utilisons, convergeant tous au point de mesure du débit de dose et semblables en ce sens aux rayons “primaires” utilisés dans le lancer de rayons. Nous avons donc introduit la grille en perspective (*perspective grid*), parfaitement adaptée à notre problématique, que nous avons choisi d’utiliser pour la suite de nos travaux.

Nous avons ensuite présenté deux algorithmes, optimisés pour le GPU, de construction et de parcours de la grille en perspective. En raison de la nature particulière du lancer de rayons effectué (les rayons sont imposés par le code de calcul de débit de dose et sont donc connus à l’avance), nous avons pu inverser l’ordre d’exécution traditionnel de l’algorithme et construire une grille indexant les rayons de la scène (et non les triangles). Cette inversion nous a permis d’améliorer les performances de l’algorithme global, mais aussi d’offrir une plus grande souplesse lors de l’exécution de l’algorithme. Les différents triangles de la scène peuvent ainsi être traités par “passes” successives, ce qui rend possible le traitement de scènes plus complexes, demandant normalement un espace mémoire trop important. Au final, les deux algorithmes proposés nous ont permis d’assurer le traitement (calcul, tri et transfert vers le CPU des intersections trouvées) d’un paquet de rayons très dense (un million de rayons), au sein d’une scène 3D volumineuse (plus de 700 000 triangles), en une centaine de millisecondes.

Nous avons ensuite proposé un algorithme de gestion de la précision permettant de contrôler les erreurs dues à l’arithmétique flottante lors des tests d’intersections rayon-triangle. En effet, les différentes approximations effectuées au cours de ces tests peuvent provoquer un mauvais traitement des rayons passant à proximité des “frontières” d’un triangle (arête ou sommet). Ainsi, pour un tel rayon, la cohérence des résultats des tests d’intersections avec les différents triangles de l’espace n’est pas garantie, ce qui peut provoquer l’“oubli” d’une interaction entre un rayon et un matériau de la scène. Afin de contourner ces difficultés, nous avons choisi d’utiliser des algorithmes adaptatifs d’arithmétique exacte, permettant d’identifier avec certitude le signe d’un déterminant 2D ou 3D (base de nos tests d’intersections), en commençant par une évaluation du déterminant via l’arithmétique flottante, puis en affinant les calculs si besoin est. Ces méthodes n’étant pas adaptées à une architecture hautement parallèle comme celle des GPUs, nous avons proposé de ne réaliser sur GPU que la première passe d’itérations de l’algorithme adaptatif. Lorsque des calculs plus fins sont nécessaires, l’intégralité de l’algorithme adaptatif pour le test concerné est exécuté sur CPU. En raison de la sélectivité du seuil de précision utilisé à la fin de la première étape de l’algorithme adaptatif, ces transferts vers le CPU restent cependant limités. Afin de garantir l’absence de ces problèmes de précision, nous avons mis en place une stratégie conjointe de gestion des informations sur la topologie du maillage, prenant en compte l’étape de clipping, nécessaire à la projection des triangles en deux dimensions, et pouvant modifier la topologie du maillage initial. Toutes ces méthodes, ajoutées aux algorithmes présentés dans le chapitre précédent, nous ont permis de garantir le bon traitement des rayons passant à proximité des “frontières” d’un triangle, sans qu’il soit besoin de régler la valeur de certains coefficients dépendant de la scène, moyennant une perte de performance très acceptable (moins de 10 % de dégradation sur les scènes étudiées).

Enfin, dans le dernier chapitre de ce manuscrit, nous avons mis en place les algorithmes des deux chapitres précédents dans le cadre de la gestion de plusieurs paquets de rayons, représentant plusieurs sources de radiations. Nous avons ainsi comparé les performances de deux stratégies différentes : la première, dite “globale”, consiste en la création de six grilles en perspective permettant de représenter l’ensemble de l’espace autour du point de mesure du débit de dose. Il s’est avéré que cette première méthode, ne tenant pas compte des caractéristiques des différents paquets de rayons au cœur de la scène, était bien moins efficace que la stratégie “locale”, consistant à construire une grille en perspective pour chacun de ces pa-

quets. Cette dernière stratégie nous a ainsi permis d'atteindre des performances interactives lors du traitement de paquets de rayons très denses, ce qui ouvre la voie à une estimation plus fine du débit de dose reçu par un opérateur. Cependant, lorsque la densité de ces paquets de rayons diminue, la performance des algorithmes utilisés est comparativement bien moins satisfaisante. Surtout, lors du traitement de multiples paquets, cette stratégie “locale” empêche une mutualisation des différents calculs nécessaires à la projection en deux dimensions des triangles de la scène. D’autres méthodes doivent donc être trouvées dans l’optique du traitement de paquets très nombreux, probablement moins denses.

Ainsi, nous avons déjà présenté à la fin du chapitre 4 de nombreuses perspectives de travail dans le but de générer des structures d'accélération plus adaptatives que les grilles régulières que nous utilisons aujourd’hui. Nous avons envisagé l'utilisation de 6 BVHs dans l'espace projectif afin de couvrir l'ensemble de l'espace, tout en conservant une structure tenant compte de la géométrie de la scène. Néanmoins, nous avons également observé dans ce dernier chapitre que pour des paquets de rayons de haute résolution, l'utilisation de la grille régulière restait très adaptée, en particulier en raison de la répartition spatiale assez homogène des sources de radiation associées à un même volume. Une perspective de travail pourrait donc consister à envisager un traitement différent des rayons en fonction de la nature des paquets auxquels ils appartiennent. Les BVHs projectifs pourraient par exemple être utilisés pour le traitement des paquets de rayons de faible résolution, tandis que des grilles en perspective plus adaptées resteraient employées pour les paquets plus importants. Une structure d'accélération à plusieurs niveaux de détails pourrait également être envisagée : 6 BVHs projectifs assez grossiers pourraient servir à orienter les calculs vers une grille en perspective ou un BVH plus fin en fonction de la nature des paquets de rayons, tout en permettant la mutualisation des calculs nécessaires aux différentes opérations de clipping et de projection des triangles de la scène.

Néanmoins, la manipulation de 6 BVHs projectifs dans le même temps reste difficile et demanderait donc toujours une gestion attentive des zones de séparation des différents espaces projectifs. La question du clipping ne pourrait toujours pas être évacuée et les triangles appartenant à plusieurs BVHs projectifs à la fois devraient être dupliqués dans les différentes structures d'accélération de la scène. Afin de contourner ces difficultés et d'éliminer définitivement les nombreux calculs liés à la projection dans un espace en deux dimensions, il serait intéressant d'envisager la création d'une structure d'accélération sphérique. Au lieu de projeter les triangles et les rayons de la scène sur les six faces d'un cube entourant le point de mesure du débit de dose comme nous le faisons aujourd’hui, les différents éléments de la scène seraient ici projetés sur une sphère autour de ce point de mesure. Via l'utilisation, par exemple, des coordonnées sphériques, les coordonnées des objets de la scène après projection sur la sphère pourraient a priori être assez rapidement obtenues. Il faudrait alors définir une première subdivision régulière de la sphère (par exemple un icosaèdre), puis subdiviser les cellules ainsi créées (ici, donc, les faces de l'icosaèdre), de manière à construire une structure hiérarchique. Il resterait alors à étudier la définition d'un BVH, par exemple, dans un tel espace, et à définir des opérations efficaces de rastérisation, entre autres. Bien que cette perspective de travail semble particulièrement intéressante, elle demanderait donc de nombreuses réflexions avant de pouvoir être mise en place.

Une autre piste de travail concerne un point que nous avons brièvement évoqué lors du chapitre 3, mais que nous n'avons hélas pas eu le temps de résoudre. En effet, nous avons dans ce chapitre présenté des techniques nous permettant d'assurer le traitement correct de rayons passant par un sommet ou une arête de triangle. Nous avons néanmoins ignoré un cas moins fréquent, mais devant être résolu, et demandant un traitement plus complexe : celui des rayons tangents à certains triangles, intersectant donc ces triangles en une infinité de points. Afin de

résoudre ce genre de cas, il nous faudrait là aussi mettre en place des méthodes permettant d'éviter les erreurs dues à l'arithmétique flottante : en effet, pour les mêmes raisons que celles évoquées dans le chapitre 3, le calcul du produit scalaire entre le rayon et la normale du triangle, permettant d'identifier ces rayons tangents, est là aussi sujet à approximations. Nous devrions donc certainement mettre en place des méthodes d'arithmétique exacte afin de garantir la cohérence des calculs entre les différents triangles. En liant, là encore, ces calculs à une gestion adaptée des informations sur la topologie du maillage, il devrait alors être possible de garantir le bon traitement de ces rayons tangents. Néanmoins, cette étape de gestion de la topologie du maillage serait nettement plus complexe que celle que nous avons présentée lors du chapitre 3 et demanderait donc un soin particulier.

Enfin, nous souhaiterions surtout dans le futur essayer de mettre à profit ces différents algorithmes pour des applications plus larges, comme la prise en compte de méthodes de calculs de rayonnements plus complexes ou la simulation de propagation de rayons lumineux. Certes, les algorithmes développés ici tirent fortement partie, pour la plupart, des spécificités de notre problème, en particulier du fait que les rayons convergent tous en un même point. Néanmoins, ces méthodes pourraient peut-être servir dans le cadre du traitement de paquets de rayons cohérents, comme les rayons primaires, ou les rayons d'ombre, dans le ray-tracing. Il serait surtout particulièrement intéressant d'essayer d'appliquer les techniques d'arithmétique exacte à ces algorithmes plus génériques, qui ont parfois un grand besoin de précision. Nous avons vu dans le chapitre 3 que la technique consistant à n'effectuer que la première passe des algorithmes de Shewchuk sur GPU n'était pas adaptée au cas des tests d'intersections 3D. Néanmoins, il serait peut-être possible d'envisager de porter une, voire deux étapes supplémentaires de ces algorithmes adaptatifs, de manière à limiter au maximum les post-traitements sur CPU et à observer une perte de performances plus proche de celle que nous avons obtenue pour le cas 2D.

Ces applications potentielles constituent donc de nombreuses pistes de développements futurs et montrent bien la richesse de ce sujet, malgré les contraintes très particulières qui le caractérisent.

Appendix

A

Sommaire

A.1 L'équation de transport de Boltzmann	121
A.1.1 Sections efficaces	121
A.1.2 Formulation de l'équation	122
A.2 Configuration matérielle	125
A.3 Publications associées à la thèse	127

A.1 L'équation de transport de Boltzmann

Nous souhaitons dans cette annexe présenter la formulation exacte de l'équation de Boltzmann, tout en essayant d'apporter une compréhension intuitive de ses différents termes.

A.1.1 Sections efficaces

En premier lieu, avant de détailler cette équation de transport, il nous faut définir une première notion essentielle pour la compréhension de l'équation. Chaque type de particule (α , β , neutron) ou de rayonnement (γ) interagit avec la matière qu'il rencontre (les *écrans* de la scène) selon différents événements (diffusion élastique ou inélastique, capture avec réémission d'autres "particules"...). Chacun de ces événements se produit selon des lois de probabilité, dépendant :

- de la nature de la matière cible (sa *composition atomique*)
- des propriétés de la particule ou du rayonnement incident, c'est-à-dire sa direction de propagation et son niveau d'énergie : pour une particule (α , β , neutron), l'énergie est fonction de la vitesse de la particule; pour un rayonnement γ , elle dépend de la fréquence du rayonnement.

Ces lois de probabilité d'apparition de chaque événement, fruits de mesures et de calculs sur des modèles de la physique, sont définies par des données appelées *sections efficaces*. Ces sections efficaces sont ensuite utilisées pour modéliser le parcours des particules ou rayonnements dans la matière de la manière suivante :

Soit un flux de particules de type P , d'intensité I_0 , arrivant sur un matériau, ce matériau comptant N atomes de type A par unité de volume. Sur une épaisseur dx , le nombre ΔI de particules interagissant avec la matière selon l'événement EVT est naturellement proportionnel à :

- l'intensité I_0 du flux
- la densité d'atomes rencontrés (N)
- l'épaisseur dx

Ainsi, on obtient la formule :

$$\Delta I = I_0 \cdot \sigma \cdot N \cdot dx$$

Cette valeur σ (qui traduit la probabilité de l'événement *EVT*) est la section efficace microscopique de cet événement, pour le type de particules P et la matière A . Une analyse dimensionnelle montre que σ a la dimension d'une surface. On l'exprime en *barn*, correspondant aux échelles de longueur atomiques ($1 \text{ barn} = 10^{-24} \text{ cm}^2$).

La section efficace macroscopique $\Sigma_t(\vec{r}, E)$, traduisant les mêmes phénomènes, que nous utiliserons dans le paragraphe suivante, représente la probabilité d'interaction par unité de longueur (densité de probabilité d'interaction) d'une particule d'énergie E , au point \vec{r} , tous phénomènes confondus.

A.1.2 Formulation de l'équation

La définition de ces sections efficaces va désormais nous permettre de présenter plus en détails la formulation exacte de l'équation de transport de Boltzmann. Les descriptions qui suivent sont très largement inspirées du chapitre 10 de l'ouvrage [Mét06].

Nous introduisons la densité angulaire $\Phi(\vec{r}, E, \vec{\Omega})$ des rayonnements (ou particules) étudiés dans l'espace des phases (position \vec{r} , énergie E , direction ou angle solide $\vec{\Omega}$). Nous nous plaçons ici en régime stationnaire, qui est l'hypothèse la plus fréquente dans les études de radioprotection. La quantité $(\Phi(\vec{r}, E, \vec{\Omega}) \cdot dr \cdot dv)$ représente le nombre de particules situées dans le volume élémentaire dr autour du point M à l'instant t , dont l'extrémité du vecteur vitesse est dans le volume élémentaire de l'espace des vitesses $d\vec{v}$.

L'équation intégro-différentielle de transport de Boltzmann, qui traduit l'évolution locale du flux de particules dans l'espace $(d\vec{r}, d\vec{v})$, est une équation bilan dans un volume élémentaire de l'espace des phases centré autour du point $P(\vec{r}, E, \vec{\Omega})$ de coordonnées $(\vec{r}, E, \vec{\Omega})$. Elle s'écrit de la manière suivante :

$$\vec{\Omega} \cdot \overrightarrow{\text{grad}}\Phi(\vec{r}, E, \vec{\Omega}) + \Sigma_t(\vec{r}, E)\Phi(\vec{r}, E, \vec{\Omega}) = \\ \int_{4\pi} d\vec{\Omega}' \cdot \int dE' \cdot \Sigma_s(\vec{r}, E' \rightarrow E, \vec{\Omega}' \rightarrow \vec{\Omega})\Phi(\vec{r}, E', \vec{\Omega}') + S(\vec{r}, E, \vec{\Omega})$$

Les termes de cette équation se décomposent de la manière suivante :

- Le premier terme traduit la variation particulière du flux de particules (quittant le volume élémentaire de l'espace des phases centré autour du point $P(\vec{r}, E, \vec{\Omega})$ à travers les surfaces physiques du volume)
- Le second terme correspond aux particules qui vont disparaître du volume élémentaire de l'espace des phases centré autour du point $P(\vec{r}, E, \vec{\Omega})$ au cours d'une interaction avec la matière (diffusion, captures...)
- Le terme intégral rend compte des particules qui “apparaissent”, c'est-à-dire qui, présentes dans le volume élémentaire de l'espace des phases centré autour du point $P(\vec{r}, E, \vec{\Omega})$, subissent une diffusion leur donnant une énergie E et une direction $\vec{\Omega}$. La section efficace différentielle en angle et en énergie, notée $\Sigma_s(\vec{r}, E' \rightarrow E, \vec{\Omega}' \rightarrow \vec{\Omega})$ permet de rendre compte de ce phénomène.
- Enfin, le dernier terme représente d'autres particules qui “apparaissent”, issues de la source et qui naissent par unité de temps et de volume à l'énergie E , dans la direction $\vec{\Omega}$ dans le volume élémentaire de l'espace des phases centré autour du point $P(\vec{r}, E, \vec{\Omega})$.

Cette équation traduit donc une équation de bilan dans le volume élémentaire de l'espace des phases considéré : en régime stationnaire, on compte autant de particules disparaissant de la cellule de l'espace des phases considérée, que de particules apparaissant dans cette cellule.

A.2 Configuration matérielle

La configuration matérielle suivante a été utilisée pour l'ensemble des expérimentations présentées dans la thèse :

- Mémoire RAM : 4 Go
- CPU : quadcore hyperthreadé Intel Xeon, 2.67 GHz
- Carte graphique NVIDIA GTX 470 :
 - Génération 2.0
 - Mémoire globale : 1 280 Mo
 - Mémoire partagée par multiprocesseur : 48 ko
 - Nombre de registres par bloc : 32768
 - Nombre total de processeurs : 448
 - Nombre de multiprocesseurs : 14
 - Nombre de processeurs par multiprocesseur : 32
 - Taille de warp : 32

A.3 Publications associées à la thèse

Hybrid GPU-CPU Adaptive Precision Ray-Triangle Intersection Tests for Robust High-Performance GPU Dosimetry Computations. *Lancelot Perrotte, Bruno Bodin, Laurent Chodorge.* Publication acceptée à International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (MC2011), Brésil, 08-12 mai 2011.

Fast GPU perspective grid construction and triangle tracing for Exhaustive Ray Tracing of Highly Coherent Rays. (version étendue) *Lancelot Perrotte, Guillaume Saupin.* Article accepté pour publication dans la revue International Journal of High Performance Computing Applications (IJHPCA).

Fast GPU perspective grid construction and triangle tracing for Exhaustive Ray Tracing of Highly Coherent Rays. *Lancelot Perrotte, Guillaume Saupin.* Proceedings of GPUScA 2010, Workshop, Autriche, 11 Septembre 2010.

High performance dosimetry calculations using adapted ray-tracing. *Lancelot Perrotte, Guillaume Saupin.* Journal of Physics: Conference Series, Volume 250, Issue 1, pp. 012068 (2010).

Bibliographie

- [AK89] James Arvo and David Kirk. *A survey of ray tracing acceleration techniques*, pages 201–262. Academic Press Ltd., London, UK, UK, 1989. [20](#)
- [AL04] Timo Aila and Samuli Laine. Alias-Free Shadow Maps. In *Proceedings of Eurographics Symposium on Rendering 2004*, pages 161–166. Eurographics Association, 2004. [12](#), [114](#)
- [AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG ’09, pages 145–149, New York, NY, USA, 2009. ACM. [26](#)
- [App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS ’68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM. [17](#)
- [AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics ’87*, pages 3–10, 1987. [19](#)
- [Bad90] Didier Badouel. Graphics gems. chapter An efficient ray-polygon intersection, pages 390–393. Academic Press Professional, Inc., San Diego, CA, USA, 1990. [64](#)
- [BBP98] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proceedings of the fourteenth annual symposium on Computational geometry*, SCG ’98, pages 165–174, New York, NY, USA, 1998. ACM. [68](#)
- [BBZ08] Robert G. Bellerman, Jeroen Bédorf, and Simon F. Portegies Zwart. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy*, 13(2):103 – 112, 2008. [12](#)
- [BCL⁺07] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, ao L. D. Comba, Jo and Cláudio T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *I3D ’07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 97–104, New York, NY, USA, 2007. ACM. [12](#)
- [Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975. [20](#)
- [BFH⁺04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH ’04, pages 777–786, New York, NY, USA, 2004. ACM. [26](#)

- [BK82] Richard P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31:260–264, 1982. [60](#)
- [Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990. [16](#), [60](#)
- [BM69] Paul Bonet-Maury. *La radioprotection*. Presses Universitaires de France, Paris, France, 1969. [4](#)
- [BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *HPG ’09: Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, New York, NY, USA, 2009. ACM. [16](#)
- [Bre65] Jack E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25 –30, 1965. [10](#)
- [Bre98] J. E. Bresenham. *Algorithm for computer control of a digital plotter*, pages 1–6. ACM, New York, NY, USA, 1998. [19](#)
- [BW09] Carsten Benthin and Ingo Wald. Efficient ray traced soft shadows using multi-frusta tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG ’09, pages 135–144, New York, NY, USA, 2009. ACM. [25](#)
- [BWS06] Solomon Boulos, Ingo Wald, and Peter Shirley. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Technical Report UUCS-06-010, 2006. [96](#)
- [BSWF06] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. *Technical Report, inTrace Realtime Ray Tracing GmbH, No inTrace-2006-001 (submitted for publication)*, 2006. [25](#)
- [Car84] Loren Carpenter. The A-buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, 1984. [11](#)
- [Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, 1974. [10](#)
- [cga] CGAL, Computational Geometry Algorithms Library. <http://www.cgal.org>. [68](#)
- [CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS ’02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. [25](#), [37](#)
- [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. [28](#)
- [CL07] Gilles Cadet and Bernard Lecussan. Coupled Use of BSP and BVH Trees in Order to Exploit Ray Bundle Performance. In *RT ’07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 63–71, Washington, DC, USA, 2007. IEEE Computer Society. [25](#)

- [CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18:137–145, January 1984. [17](#)
- [DK08] Holger Dammertz and Alexander Keller. The edge volume heuristic - robust triangle subdivision for improved bvh performance. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 155–158, 2008. [49](#)
- [DPD⁺06] C. M. Diop, O. Petit, E. Dumonteil, F. X. Hugot, Y. K. Lee, A. Mazzolo, and J. C. Trama. Tripoli-4 : A 3D continuous-energy Monte Carlo transport code. *Transactions of the American Nuclear Society*, 95:661, 2006. [5](#)
- [DSD⁺09] Carsten Dachsbaecher, Philipp Slusallek, Tomas Davidovic, Thomas Engelhardt, Mike Phillips, and Iliyan Georgiev. 3D Rasterization – Unifying Rasterization and Ray Casting. Technical report, VISUS/University Stuttgart and Saarland University, 2009. [66](#), [70](#)
- [EG07] Manfred Ernst and Gunther Greiner. Early split clipping for bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 73–78, Washington, DC, USA, 2007. IEEE Computer Society. [48](#)
- [EL10] Christian Eisenacher and Charles Loop. Data-parallel micropolygon rasterization. In Stefan Seipel and Hendrik Lensch, editors, *Eurographics 2010 Annex: Short Papers*, May 2010. [57](#)
- [Eve01] Cass Everitt. Interactive Order-Independent Transparency, NVIDIA. Technical report, 2001. [11](#)
- [FG85] R. Forster and T. Godfrey. MCNP - a general Monte Carlo code for neutron and photon transport. In Raymond Alcouffe, Robert Dautray, Arthur Forster, Guy Ledanois, and B. Mercier, editors, *Monte-Carlo Methods and Applications in Neutronics, Photonics and Statistical Physics*, volume 240 of *Lecture Notes in Physics*, pages 33–55. Springer Berlin / Heidelberg, 1985. 10.1007/BFb0049033. [5](#)
- [FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph.*, 14:124–133, July 1980. [20](#)
- [FLB⁺09] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG ’09, pages 59–68, New York, NY, USA, 2009. ACM. [57](#)
- [FS88] Donald S. Fussell and K. R. Subramanian. Fast ray tracing using k-d trees. Technical report, Austin, TX, USA, 1988. [20](#)
- [FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu ray-tracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS ’05, pages 15–22, New York, NY, USA, 2005. ACM. [26](#)
- [FTI86] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. ARTS: accelerated ray-tracing system. *IEEE Comput. Graph. Appl.*, 6:16–26, April 1986. [19](#)

- [Gar08] Kirill Garanzha. Efficient clustered bvh update algorithm for highly-dynamic models. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 123–130, 2008. [27](#)
- [GFW⁺06] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3):517–525, September 2006. (Proceedings of Eurographics). [27](#)
- [GL10] K. Garanzha and C. Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum*, 29:289–298, 2010. [26](#), [50](#), [51](#)
- [Gla88] Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Comput. Graph. Appl.*, 8:60–70, March 1988. [20](#)
- [Gla89] Andrew S. Glassner, editor. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989. [64](#)
- [GPSS07] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Real-time ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 113–118, Washington, DC, USA, 2007. IEEE Computer Society. [22](#), [26](#)
- [GR08] Christiaan P. Gribble and Karthik Ramani. Coherent ray tracing via stream filtering. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 59–66, 2008. [25](#), [26](#)
- [GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7:14–20, May 1987. [23](#)
- [Han86] P Hanrahan. Using caching and breadth-first search to speed up ray-tracing. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 56–61, Toronto, Ont., Canada, Canada, 1986. Canadian Information Processing Society. [26](#)
- [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. [20](#), [23](#), [24](#)
- [Hay03] Brian Hayes. A lucid interval. *American Scientist*, 91(6):484–488, November–December 2003. [68](#)
- [HB09] Jared Hoberock and Nathan Bell. Thrust: A Parallel Template Library, 2009. Version 1.2. [14](#), [38](#), [39](#)
- [HLB01] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ARITH '01, pages 155–, Washington, DC, USA, 2001. IEEE Computer Society. [67](#)
- [HM08a] Warren Hunt and William R. Mark. Adaptive acceleration structures in perspective space. In *IEEE/EG Symposium on Interactive Ray Tracing 2008*, pages 11–17. IEEE/EG, Aug 2008. [30](#)

- [HM08b] Warren Hunt and William R. Mark. Ray-Specialized Acceleration Structures for Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing 2008*, pages 3–10. IEEE/EG, Aug 2008. [29](#), [89](#)
- [HMS06] W. Hunt, W.R. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. *Symposium on Interactive Ray Tracing*, 0:81–88, 2006. [28](#)
- [HPS] Mike Houston, Arcot J. Preetham, and Mark Segal. Graphics Hardware (2005), pp. 1-6 M. Meissner, B.-O. Schneider (Editors) A Hardware F-Buffer Implementation. [11](#)
- [HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986. [16](#)
- [HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D ’07, pages 167–174, New York, NY, USA, 2007. ACM. [26](#)
- [IWP] Thiago Ize, Ingo Wald, and Steven G Parker. Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*. [27](#)
- [Jae68] Robert Gottfried Jaeger. *Engineering Compendium On Radiation Shielding, Volume 1: Shielding Fundamentals And Methods*. Springer-Verlag Berlin Heidelberg, New York, 1968. [4](#)
- [JC99] Norman P. Jouppi and Chun-Fa Chang. Z3: an economical hardware technique for high-quality antialiasing and transparency. In *HWWS ’99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 85–93, New York, NY, USA, 1999. ACM. [11](#)
- [JLBM05] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular Z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.*, 24(4):1462–1482, 2005. [12](#)
- [KBS11] Javor Kalojanov, Markus Billeter, and Philipp Slusallek. Two-level Grids for Ray Tracing on GPUs. In *Eurographics 2011, à paraître*, 2011. [29](#), [114](#)
- [KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20:269–278, August 1986. [21](#)
- [KS97] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Comput. Graph. Appl.*, 17:42–51, January 1997. [20](#)
- [KS06] Andrew Kensler and Peter Shirley. Optimizing Ray-Triangle Intersection via Automated Search. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 33–38, Sep 2006. [66](#)
- [KS09] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *HPG ’09: Proceedings of the Conference on High Performance Graphics 2009*, pages 23–28, New York, NY, USA, 2009. ACM. [29](#), [35](#), [36](#), [37](#)

- [LD08] Ares Lagae and Philip Dutré. Compact, Fast and Robust Grids for Ray Tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(4):1235–1244, June 2008. [34](#)
- [LGS⁺09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009. [29](#)
- [LHLW09] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Efficient depth peeling via bucket sort. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 51–57, New York, NY, USA, 2009. ACM. [12](#)
- [LLAm01] Jonas Lext, , Jonas Lext, and Tomas Akenine-möller. Towards rapid reconstruction for animated ray tracing. In *Eurographics Short Presentations*, pages 311–318, 2001. [27](#)
- [LYM06] Christian Lauterbach, Sung-Eui Yoon, and Dinesh Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45, 2006. [25](#), [27](#)
- [MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6:153–166, May 1990. [23](#)
- [Mét06] H. Métivier. *Radioprotection et ingénierie nucléaire*. Génie atomique. EDP Sciences, 2006. [4](#), [122](#)
- [MG09] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. Technical Report CS2009-14, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2009. [16](#)
- [MG10] Duane G. Merrill and Andrew S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT ’10, pages 545–546, New York, NY, USA, 2010. ACM. [16](#), [38](#)
- [Mol97] Moller, Tomas and Trumbore, Ben. Fast, minimum storage ray-triangle intersection. *journal of graphics, gpu, and game tools*, 2(1):21–28, 1997. [64](#)
- [MP01] William R. Mark and Kekoa Proudfoot. The F-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 57–64, New York, NY, USA, 2001. ACM. [11](#)
- [MW04] Jeffrey Mahovsky and Brian Wyvill. Memory conserving bounding volume hierarchies with coherent ray tracing. *IEEE Transactions on Visualization and Computer Graphics (submitted)*, 2004. [26](#)
- [NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008. [14](#)
- [NVI09] NVIDIA. Cuda Zone, The resource for CUDA Developers. http://www.nvidia.com/object/cuda_home.html, 2009. [14](#), [68](#)

- [OG97] Marc Olano and Trey Greer. Triangle scan conversion using 2d homogeneous coordinates. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '97, pages 89–95, New York, NY, USA, 1997. ACM. [75](#)
- [OOR09] Katsuhsia Ozaki, Takeshi Ogita, Siegfried M. Rump, and Shin'ishi Oishi. Adaptive and efficient algorithm for 2D orientation problem. *Japan journal of industrial and applied mathematics*, 26(2-3):215–231, 2009. [68](#)
- [Ope09] OpenCL. <http://www.khronos.org/opencl/>, 2009. [14](#)
- [PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21:703–712, July 2002. [25](#)
- [PGDS09] Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. Object partitioning considered harmful: space subdivision for bvhs. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 15–22, New York, NY, USA, 2009. ACM. [23](#)
- [PGSS06] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of sah kd-trees. *Symposium on Interactive Ray Tracing*, 0:89–94, 2006. [28](#)
- [PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics). [26](#)
- [Pin88] Juan Pineda. A parallel algorithm for polygon rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 17–20, New York, NY, USA, 1988. ACM. [66](#)
- [PL10] J. Pantaleoni and D. Luebke. Hlbvh: hierarchical lvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 87–95, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association. [29](#)
- [PPL⁺99] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5:238–250, July 1999. [20](#)
- [Roc56] Theodore. Rockwell. *Reactor shielding design manual / ed. by T.Rockwell*. Macmillan, London :, 1956. [4](#)
- [RSF01] Comparative Study Rafael, Rafael J. Segura, and Francisco R. Feito. Algorithms to test ray-triangle intersection. In *Journal of WSCG*, pages 200–1, 2001. [64, 66](#)
- [RSH00] Erik Reinhard, Brian E. Smits, and Chuck Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 299–306, London, UK, 2000. Springer-Verlag. [20, 27](#)

- [RSH05] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM. [25](#)
- [RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph.*, 14:110–116, July 1980. [21](#)
- [SAA07] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, pages 395–404, 2007. [28](#)
- [SC05] C Suteau and M Chiron. An iterative method for calculating gamma-ray build-up factors in multi-layer shields. *Radiation Protection Dosimetry*, 116(1-4 Pt 2):489–492, 2005. [7](#)
- [SCS⁺08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Gochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008. [56](#)
- [SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17:32–42, January 1974. [76](#)
- [She96] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1996. [68](#)
- [SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society. [16](#), [38](#)
- [SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, number 3, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. [16](#)
- [SM03] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2 edition, 2003. [17](#)
- [Spa89] John Neil Spackman. *Scene decompositions for accelerated ray tracing*. PhD thesis, 1989. AAIDX89112. [19](#)
- [SPF⁺07] John E Stone, James C Phillips, Peter L Freddolino, David J Hardy, Leonardo G Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007. [12](#)
- [Sze03] László Szecsi. *An effective implementation of the k-D tree*, pages 315–326. Charles River Media, Inc., Rockland, MA, USA, 2003. [27](#)

- [TMLT11] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. Collision-streams: Fast GPU-based collision detection for deformable models. In *I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2011. 12
- [TSP07] Kresimir Trontl, Tomislav Smuc, and Dubravko Pevec. Support vector regression model for the estimation of [gamma]-ray buildup factors for multi-layer shields. *Annals of Nuclear Energy*, 34(12):939–952, 2007. 7
- [Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 64, 65
- [Wal07] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society. 28, 29
- [WBS03] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 11–, Washington, DC, USA, 2003. IEEE Computer Society. 27
- [WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007. 25, 27
- [WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001. 18, 20, 24
- [WGBK07] Ingo Wald, Christiaan P Gribble, Solomon Boulos, and Andrew Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012, 2007. 25
- [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 the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006. 23, 27, 28
- [Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23:343–349, June 1980. 17
- [WIK⁺06] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3):485–493, 2006. 25
- [Wit01] Craig M. Wittenbrink. R-buffer: a pointerless A-buffer hardware architecture. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 73–80, New York, NY, USA, 2001. ACM. 11
- [WK06] Carsten Wächter and Alexander Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 - Proceedings of the 17th Eurographics Symposium on Rendering*, pages 139–149, 2006. 22

- [ZHR⁺09] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. Renderants: interactive reyes rendering on gpus. *ACM Trans. Graph.*, 28:155:1–155:11, December 2009. [42](#)
- [ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008. [29](#)
- [ZM10] Wei Zhang and Ivan Majdandzic. Fast triangle rasterization using irregular z-buffer on cuda. Master’s thesis, Chalmers University of Technology, Göteborg, Sweden, 2010. [55](#)

Algorithmes robustes de lancer de rayons pour calcul de dose interactif

Dans le contexte actuel, il est plus que jamais nécessaire de disposer d'outils de simulation permettant d'évaluer rapidement la dose reçue par des opérateurs travaillant sur des sites irradiés. Afin d'étudier facilement de nombreux scénarios d'intervention, il nous faut diminuer les temps de calcul des simulateurs actuels, ce qui passe principalement par une accélération des calculs géométriques associés au calcul de dose. Ces calculs consistent à identifier et trier l'ensemble des intersections entre plusieurs groupes de rayons "radiatifs", convergeant tous au point de mesure de la dose, et une scène 3D volumineuse.

Afin d'effectuer l'ensemble de ces calculs en une fraction de seconde, nous proposons d'abord un algorithme GPU complet permettant le traitement efficace d'un paquet de rayons cohérents. Ensuite, nous présentons une modification de cet algorithme garantissant la robustesse des tests d'intersection rayon-triangle et l'absence de problèmes de précision dus aux calculs en arithmétique flottante, sans utilisation de coefficients dépendants de la scène et sans perte de performance notable (moins de 10% de dégradation). Enfin, nous proposons une stratégie efficace de traitement de multiples paquets (nécessaire lors de l'étude de multiples sources de radiations) exploitant ces premiers résultats.

Ces méthodes nous permettent d'obtenir un calcul de dose interactif et robuste sur des scènes de taille importante (une scène de plus de 700 000 triangles, et 12 paquets de 100 000 rayons chacun, générant plus de treize millions d'intersections, stockées, triées le long de chaque rayon et transférées vers le CPU en 470 millisecondes).

Mots clés : Lancer de rayons - GPU - calcul parallèle - algorithmes géométriques - arithmétique flottante - arithmétique exacte - radioprotection - dosimétrie

Robust Ray-tracing Algorithms for Interactive Dose Rate Evaluation

More than ever, it is essential today to develop simulation tools to rapidly evaluate the dose rate received by operators working on nuclear sites. In order to easily study numerous different scenarios of intervention, computation times of available softwares have to be lowered. This mainly implies to accelerate the geometrical computations needed for the dose rate evaluation. These computations consist in finding and sorting the whole list of intersections between a big 3D scene and multiple groups of « radiative » rays meeting at the point where the dose has to be measured.

In order to perform all these computations in less than a second, we first propose a GPU algorithm that enables the efficient management of one big group of coherent rays. Then we present a modification of this algorithm that guarantees the robustness of the ray-triangle intersection tests through the elimination of the precision issues due to floating-point arithmetic. This modification does not require the definition of scene-dependent coefficients (« epsilon » style) and only implies a small loss of performance (less than 10%). Finally we propose an efficient strategy to handle multiple ray groups (corresponding to multiple radiative objects) which use the previous results.

Thanks to these improvements, we are able to perform an interactive and robust dose rate evaluation on big 3D scenes: all of the intersections (more than 13 million) between 700 000 triangles and 12 groups of 100 000 rays each are found, sorted along each ray and transferred to the CPU in 470 milliseconds.

Key-words : ray-tracing – GPU - parallel computing - geometric algorithms - floating-point arithmetic - exact arithmetic - radiation protection - radiation dosimetry