Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohamed Khider Biskra

Faculté des Sciences Exactes, des sciences de la Nature et de la Vie

Département d'Informatique

N° d'ordre :…………
Série :………………

Mémoire

Présenté en vue de l'obtention du diplôme de Magister en Informatique

Option**: Intelligence Artificielle et systèmes distribués**

Titre :

Accélération par l'utilisation du matériel graphique pour la représentation et le rendu volumique.

Par :

BERIMA Salima

Soutenu le : 20/ 09/2011

<u>Devant le jury :</u>

Pr. A . BELAMI	**PR**	**Université de Batna**	**Président**
Dr. M. C. BABAHENINI	**MCA**	**Université de Biskra**	**Rapporteur**
Dr. A. ZIDANI	**MCA**	**Université de Batna**	**Examinateur**
Dr. O. KAZAR	**MCA**	**Université de Biskra**	**Examinateur**

Résumé

De nos jours, les images virtuelles sont omniprésentes dans plusieurs domaine, citons par exemple : les jeux vidéos, la publicité et l'imagerie médicale. Le défit principale dans la synthèse d'images et le temps de rendu.

Durant ces dernières années, de très nombreuses contributions en rendu volumique ont été apportées afin de satisfaire les besoins d'exploration de l'utilité des GPUs et d'apporter plus d'interactivité pour le processus de rendu. L'avancé considérable de l'architecture des cartes graphiques permet depuis quelques années la visualisation de données numériques sur PC standard, donnant ainsi lieu à une nouvelle famille d'algorithmes tirant profit des possibilités offertes par ce matériel.

L'objectif de ce mémoire est l'accélération du rendu volumique en introduisant une approche basée sur le traitement des goulets d'étranglement rencontré au niveau du pipeline graphique. Ces goulets servent à générer des points de ralentissement du processus de rendu et donc détruisent la notion de *temps réel.*

L'approche proposée dans ce mémoire est une combinaison de deux structures de données : le briking et l'octree afin de traiter les goulets d'étranglement et par conséquent accélérer le rendu volumique.

L'algorithme de rendu Raycasting, est l'algorithme le plus naturel qui permet d'atteindre un niveau haut de qualité de rendu volumique ainsi qu'il est bien adaptée pour être implémenter sur les GPUs modernes.

Mots clé : *rendu volumique, accélération, GPU, temps réel, goulets d'étranglement, brique, ray casting.*

Abstract

Nowadays, virtual images are ubiquitous in many fields, for instance: video games, advertising and medical imaging. The main challenge in computer graphics is rendering time. In recent years, numerous contributions in volume rendering have been made to satisfy the needs of exploration of the utility of GPUs and provide more interactivity for the rendering process. The advanced architecture of large graphics cards in recent years, allows the visualization of digital data on standard PC, giving rise to a new family of algorithms taking advantage of the opportunities offered by this material.

The objective of this thesis is the acceleration of volume rendering by introducing an approach based on the processing bottlenecks encountered in the graphics pipeline. These bottlenecks generate points of slowing the rendering process and thus destroy the concept of real time.

The approach proposed here, is a combination of two data structures: the briking and the octree to address bottlenecks and therefore accelerate volume rendering.

The ray casting rendering algorithm is the most natural algorithm that achieves a high level of quality of volume rendering, and it well suited to be implemented on modern GPUs.

الصور الافتراضية في العديد من على سبيل
: الفيديو، و التصوير الطبي يتمثل التحدي الرئيسي في رسومات
.

ظهرت في السنوات الأخيرة، مساهمات عديدة
وتوفير مزيد عملية .

التطور المذهل في السنوات الأخيرة البيانات
الرقمية على جهاز الكمبيوتر ظهور جديدة من الخوارزميات
يتيحها هذ .

يتمثل الهدف من هذه المذكرة تسريع عن طريق إدخال نهج قائم
.

هذه عملية إزالة مفهوم
الوقت الحقيقي.

يتكون النهج المقترح في هذه المذكرة من مزيج من هياكل البيانات: التقسيم خلايا
متماثلة والشجرة الثمانية تسريع .

يعتبر خوارزمية العرض قذف حيث يسمح
كما أنه لتنفيذ على وحدات معالجة الرسومات الحديثة.

Remerciements

Je souhaite, dans un premier temps, exprimer toute ma gratitude envers mon Encadreur Mr BABAHENINI Mohamed Chaouki, Maitre de conférences à l'université de Biskra pour toute la confiance et les encouragements qu'ils m'ont prodigués pendant ce mémoire Ces travaux n'auraient jamais été menés à bien sans lui et sans ses précieux conseils. Qu'il soit ici assuré de toute mon estime et de mon profond respect.

Je remercie aussi tout particulièrement Mr BELAMI Azzedine, Professeur à l'université de Batna, pour l'honneur qu'il me fait en acceptant la présidence de ce jury.

Je voudrai aussi adresser mes plus profonds remerciements à Mr ZIDANI Abdelmadjid, Maitre de conférences à l'université de Batna et Mr KAZAR Okba, Maitre de conférences à l'université de Biskra pour avoir eu l'amabilité de participer à mon jury de soutenance.

Ces remerciements ne pourraient se terminer sans que j'exprime mes plus profonds remerciements à ma famille, mes parents et tous ceux qui m'ont supporté et soutenu durant ce mémoire.

Table des matières

Résumé……………………………………………………………………………………...……i

Remerciements……………………………………………………………………………….…iv

Table de matière…………………………………………………………………………………v

Liste des figures………………………………………………………………………………..…viii

Liste des tableaux……………………………………………………………………………………..x

Introduction générale ……………………………………………………………………………...1

Chapitre1 *Introduction au rendu volumique*

1. Introduction……………………………………………………………………………………...3

2. Le rendu volumique direct : principe...3

2. Les algorithmes de rendu volumique direct…………………………………………………..3

3. Procédure de rendu volumique ………………………………………………………………4

4. Les défis du rendu volumique…………………………………………………………………...6

5. Applications du rendu volumique ……………………………………...………………6

6. Les algorithmes de rendu volumique direct…………………………………………………..8

6.1 Lancer de rayons……………………………………………………………………………....8

6.2 Splatting ………………………………………………………………………………...9

6.3 Algorithme Shear-Warp…..10

7. Les techniques de rendu volumiques……………………………………………………………12

7. 1. Rendu à base de points…………………………………………………………....12

7.2. Rendu à base d'images……………………………………………………………......13

7. Illumination…………………………………………………………………………………….18

7.1. Modèles d'illumination…………………………………………………………......20

7.1.1. Spécularité et diffusivité………………………………………………………….21

7.1.2 BRDF : Bidirectional Reflectance Distribution Function……………………….....22

8. Conclusion………………………………………………………………………………….......23

Chapitre2 : *Utilisation des GPUs dans le rendu volumique*

1. Introduction et objectifs attendus…………………………………………………….......24

3. Introduction aux GPUs…..24

3.1 Définition de GPU…………………………………………………………………….24

3.2 Apports du GPU……………………………………………………………………………25

3.3 Evolution des cartes graphiques ……………………………………………………….25

4. Programmation GPU……………………………………………………………………......26
4.1 GPUs Programmables ……………………………………………………………….27
4.2 Exemple de programmation GPU : Phong shading………………….……………….…28
5. Les défis traités par les GPUs…..30
5.1 Gestion de gros volumes de données …………………………………………...30
5.2 Mémoires embarquées……………………………………………………………30
5.3. Interconnections avec les autres composants matériels……………………………….31
6. Matériel graphique pour le rendu interactif………………………………………………31
6.1 Description du pipeline graphique……………………………………………….....32
7. Les Shaders………………………………………………………………………….....35
7.1 Les générations de shaders…..35
8. Les architectures de dernière génération…………………………………………….....36
8.1 Architecture matérielle…………………………………………………………….....36
9. Exemple d'étude: Lancer de rayon sur GPUs…………………………………………….38
10. Conclusion…..41

Chapitre 3 : Les structures de données accélératrices

1. Introduction……………………………………………………………………….42
2. Les structures accélératrices ……………………………..……………………………42
2.1 Volumes englobants………………………………………………………………….42
2.2. Subdivisions spatiales……………………………………..…………………………43
2.2.1 La grille régulière (brique régulière)………………………………………….44
2.2.2 Le kd tree (kd arbre) ……………………..…………………..……......50
2.2.3. La hiérarchie des volumes englobants (the Bounding Volume Hierarchy ou BVH)…...56
2.2.4 Les structures hybrides…………………………………………………………59
5. Classification des différentes structures existantes……………………………………….60
6. Conclusion………………………………………………………………………….....60

Chapitre 4 : Traitement des goulets d'étranglement

1. Introduction……………………………………………………………………….61
2. Les goulets d'étranglement au niveau de pipeline graphique……………………………….62
2.1 Définition des goulets d'étranglement…...63
2.2 Les causes et les motivations……………………………………………………....63
3. Description de la méthode……………………………………………………………..64
3.1 La structure de données choisie………………………………………………...64
3.1.1 Subdivision en briques régulières (Bricking)…...64

3.1.2 L'octree…..66
3.2 La méthode (algorithme) accélératrice de rendu…..67
4. La contribution……………………………………………………………………....69
4.1 Les techniques introduites dans la méthode……………………………………………….69
4.1.1 Early ray termination…...69
4.1.2 Le raycasting…………………………………………………………………….70
4.1.3 La technique "neighbor skipping" optimisée…...73
5. Conclusion……………………………………………………………………………76

Chapitre 5 : *Implémentation du système : Accélération de rendu volumique en utilisant Briking+octree*

1. Introduction………………………………………………………………………...78
2. Études de cas des scènes à traiter………………………………………………….....78
3. Na ture de données à visualiser………………………………………………………79
4. Le choix de la structure d'accélération……………………………………………..79
4.1 La structure de brique régulière…..79
4.2 La structure Octree …...80
4.2.1 Construction de l'octree…..80
5. Aperçu global du système……………………………………………………………81
6. Description des principaux algorithmes utilisés……………………………………… 83
6.1. Algorithme de création des briques régulières…..84
6.2. Arbre octal: algorithme de découpage ……………………………………………....84
6.3. Sauvegarde et lecture d'un octree ……………………………………………………....85
6.4. Algorithme de rendu : le ray casting………………………………………………...86
7. Résultats et discussions………………………………………………………………...89
7.1 Choix de l'environnement de travail……………………………………………….89
7.2 Les résultats expérimentaux……………………………………………………...90
7.3 Influence des paramètres de la structure sur le rendu ……………………………….90
8. Conclusion…………………………………………………………………………….....94
Annexe………………………………………………………………………………………95
Conclusion générale ……………………………………………………………………102
Bibliographie 104

Table des figures

Figure 1.1 : Rendu volumique direct……………………………………………………………..4
Figure 1.2: Le processus de rendu volumique……………………………………………......4
Figure 1.3 : Plans de découpage…………………………………………………………….5
Figure1.4 : Le lancer de rayon : concept de base………………………………………...8
Figure1.5 : Interpolation trilinéaire……………………………………………………....9
Figure 1.6 : L'algorithme du shear-warp avec une projection parallèle…………………......11
Figure 1.7 : L'algorithme du shear-warp avec une projection perspective…………..………11
Figure 1.8 : Objet 3D représenté sous forme d'un nuage de points……………..……………12
Figure 1.9: Illustration d'un effet de parallaxe………………………………………………14
Figure 1.10 : Les deux principaux types d'images de profondeur……………………………15
Figure 1.11: Les étapes d'un rendu volumique typique à base de texture…...………………16
Figure 1.12 : Visualisation de la série de polygones parallèles au plan de vue support de rendu des données volumiques………………………………………………………………18
Figure 1.13 : Illustration du rendu volumique à base d'une texture 3D……………………...19
Figure 1.14 : Modèle d'illumination : a) ambiante, b) diffuse, c) spéculaire……………….20
Figure 1.15 : Réflexion spéculaire…………………………………………………………...21
Figure 1.16 : réflexion diffuse……………………………………………………………......21
Figure 1.17 : Les angles polaires comme paramètres d'une BRDF…………………………22
Figure 2.1: Les premières cartes graphique……………………………………………….....26
Figure 2.2: Deux sphères rendues avec l'ombrage de Phong………………………………...29
Figure 2.3 : Hiérarchie de la mémoire dans le GPU……………………………………….....31
Figure 2.4: GPU NVidia GeForce 8800 GTS……………………………………………......32
Figure 2.5: Vue générale du pipeline graphique………………………………………….......32
Figure 2.6: Schéma de principe du pipeline de rendu temps réel……………………………34
Figure 2.7 : Diagramme de bloc de l'architecture du G80…………………………………...36
Figure 2.8 : Schéma de principe d'un multiprocesseur du G80……………………………...37
Figure 2.9: Division du lancer de rayon en noyaux……………………………………….....39
Figure 3.1:Uun objet délimité par sa boîte englobante………………………………………43
Figure 3.2 : Trois différentes méthodes de découpage de l'espace……………………….......44
Figure 3.3 : Illustration de la subdivision en grille régulière………………………………...45
Figure 3.4 : L'arbre octal…………………………………………………………………….....46

Figure 3.5: Schéma de la mémoire de la Grille…...48
Figure 3.6: Disposition de la mémoire pour les triangles et les listes de triangle……….......49
Figure 3.7 : Le parcours de la grille régulière…………………………………………….....49
Figure 3.8:Evaluation de la fonction de coût…………………………………………...51
Figure 3. 9: Le test d'intersection…………………………………………………….....53
Figure 3.10 : L'organisation de la mémoire pour le KD-tree………………………………...55
Figure 3.11 : Illustration de la structure BVH……………………………………………......56
Figure 4.1 : Le pipeline graphique et ses goulets d'étranglement……………………………62
Figure 4.2 : Partition du volume en 4^3 briques……………………………………………65
Figure 4.3 : Une division octree, et son arbre…………………………………………….....66
Figure 4.4: Le même fragment de volume, rendu avec (a) Bricking et l'octree………….......68
Figure 4.5: L'arrêt précoce de rayon……………………………………………………….....70
Figure 4.6: Le Ray casting……………………………………………………………………...71
Figure 4.7 : Résultats des deux premières passes de la méthode de Ray-Casting…………...72
Figure 4.8 : near neighbor skipping……………………………………………………………74
Figure 4.9 : Rendu volumique basé tranche de texture………………………………………75
Figure 4.10 : volumes de test : (a) 512x512x512 voxels, (b) vasculaires……………………75
Figure 5.1: La structure de données brique……………..…………………………………80
Figure 5.2 : (a) objet à mailler, (b) construction de l'octree…………………………………81
Figure 5.3 : vue globale du système…………………………………………………………82
Figure 5.4: La taille de brique détermine la finesse du maillage…….………………………84
Figure 5.6 : Digramme fps en fonction de taille de brique…………………………………90
Figure 5.5 : Scènes rendues avec taille brique (TB)………………………………………....91
Figure 5.8: Digramme fps en fonction de la profondeur de l'octree………………………...92
Figure 5.7 : Scènes rendues avec des profondeurs d'octree variées……………………….....93
Figure 5.7 : Scènes rendues avec des profondeurs d'octree variées……………………….....94

Liste des tableaux

Tableau 3.1 : Classification des différentes structures de données………………………..…60
Tableau 4.1 : Comparaison entre lancer de rayon et le ray casting……..…………………...71
Tableau 5.1 : Variation de temps de rendu suivant la taille de brique……………………….92
Tableau 5.2 : Variation de temps de rendu suivant la profondeur de l'octree……………....94

Introduction Générale

Introduction Générale

À l'heure actuelle, le domaine de la visualisation de données volumiques est un des axes majeurs de recherche. Ce domaine très vaste permet des applications variées et offre des solutions de représentations visuelles de données parfois complexes, de grandes tailles et provenant de sources diverses, comme par exemple, de simulations ou encore de numérisations (images IRM, scanners 3D, . . .).

Le rendu volumique est un domaine de recherche en constante évolution, car la puissance grandissante des machines remet toujours en question le compromis entre réalisme et le coût des calculs nécessaires à l'obtenir. Des algorithmes trop coûteux deviennent abordables et leur utilisation plus courante.
D'une manière globale, le rendu volumique est utilisé dans des applications où la navigation interactive est essentielle, c'est pourquoi la communauté scientifique développe des méthodes et outils pour optimiser le rendu volumique, malgré que chaque domaine a ses spécificités, en particulier la visualisation scientifique où de gros volumes de données, de champs scalaires, vectoriels, etc... sont manipulés.

L'un des défis est donc, de pouvoir trouver des méthodes permettant de rendre de façon temps réel ces données avec une qualité visuelle proche de la réalité.
Bien que les performances du matériel augmentent approximativement tous les ans, la demande de réalisme et par conséquent la complexité des scènes ne cesse d'augmenter et dépasse toujours les capacités de calcul existantes. A partir de cette remarque, des approches pour contourner ce problème sont récemment apparues : le *rendu temps réel* et l'accélération des algorithmes du rendu en utilisant les cartes graphiques. Ces deux approches sont maintenant un domaine de recherche très actif afin de donner une visualisation proche le plus possible de la réalité. Pour cela, nous focalisons notre étude sur les techniques accélératrices à l'aide des GPUs ("Graphic Processing Unit", ou processeur de la carte graphique).

L'avènement des GPU modernes offrent un degré de programmabilité qui ouvre un large champ d'applications pour le traitement des millions de triangles.

Le grand avantage d'utiliser le GPU par rapport au CPU afin d'accélérer le processus de rendu réside dans les points suivants :

- Son architecture massivement parallèle.
- La séparation en deux unités (vertex shader et fragment shader).
- Des instructions dédiées à des tâches graphiques.

Notre objectif est de proposer une méthode permettant d'accélérer le rendu volumique à l'aide de GPU. Pour cela, nous allons étudier dans ce mémoire quelques structures de données pour décider quelle est la meilleure structure en terme de consommation mémoire et de temps de rendu et principalement son efficacité de traiter les goulets d'étranglement au niveau du pipeline graphique, car ce traitement décharge beaucoup le processeur graphique et accélère en plus le rendu.

Ce mémoire est composé en cinq parties principales, **le premier chapitre** présente une introduction au rendu volumique en s'appuyant sur les principaux algorithmes utilisés dans ce domaine ainsi que quelques techniques existantes.

Le deuxième chapitre est consacré à fournir une description du GPU et son pipeline graphique, nous nous sommes concentré sur l'utilisation des GPUs afin d'accélérer le processus de rendu.

Le troisième chapitre est dédié à la description de quelques structures de données adaptées à l'accélération de la visualisation.

Le quatrième chapitre apporte la proposition d'une approche permettant l'adaptation d'une structure de données à être implémenter sur la carte graphique et à traiter les goulets d'étranglement du pipeline graphique

Finalement dans **le chapitre cinq**, nous devons choisir l'algorithme de rendu proprement dit et qui s'adapte au GPU. Le Raycasting est le meilleur moyen pour rendre les volumes et il peut profiter des fonctionnalités intégrées dans les cartes graphiques d'aujourd'hui. Le raycasting amélioré par la technique de ‘‘ plus proche voisin’’ utilise les points forts de l'architecture de GPU pour créer plus de réalisme aux images en des taux d'affichage interactive.

Chapitre 01 :

Introduction au rendu volumique

Chapitre 1 : Introduction au rendu volumique

1. Introduction

De nos jours, nombreuses applications comme l'imagerie médicale, la sismologie ou l'industrie des films, utilisent et manipulent de gigantesques tableaux tridimensionnels de données. Ces volumes peuvent contenir plusieurs Méga-octets de données. Il est possible de visualiser directement des données 3D grâce à l'outil informatique. Cette action s'appelle le rendu volumique.

Dans ce chapitre, nous allons fournir une introduction au rendu volumique en présentant premièrement les algorithmes de rendu volumique les plus populaires puis les différentes techniques utilisées pour la visualisation 3D.
Nous terminerons ce chapitre par la présentation de quelques modèles d'illumination pour le rendu volumique.

2. Le rendu volumique direct : principe

Le terme « rendu volumique » consiste en un ensemble de techniques qui permettent la visualisation de données 3D sous la forme d'images 2D (projection de données volumiques). Les données 3D sont associées à une grille de points 3D, qui va définir leur position, leur voisinage et l'algorithme de visualisation à utiliser.

Les algorithmes de rendu volumique direct utilisent les données originales du volume. Chaque pixel de l'image est considéré comme un rayon qui traverse un volume (voir figure 1.1). La complexité de l'algorithme réside alors dans la manière dont le rayon va interagir avec les échantillons (ou voxels) du volume. Ainsi, tous les algorithmes de rendu volumique direct fonctionnent en échantillonnant des valeurs le long du rayon et en calculant la contribution de ces valeurs, suivant un modèle de rendu, à la couleur finale du pixel.

Il existe en fait deux manières possibles de calculer l'image finale. La première consiste à calculer la contribution de chaque élément du volume sur les pixels de l'image. Cette façon de calculer l'image est appelée object-order car les voxels sont passés en revue un par un dans un ordre de visibilité donné. A l'inverse, la seconde manière consiste à calculer chacun des pixels de l'image un par un. Il faut alors pour chaque rayon traverser le volume afin de trouver les voxels qui contribuent à la couleur finale. On peut ainsi éviter de parcourir les parties cachées du volume. Cette méthode est appelée image-order **[CC07]**.

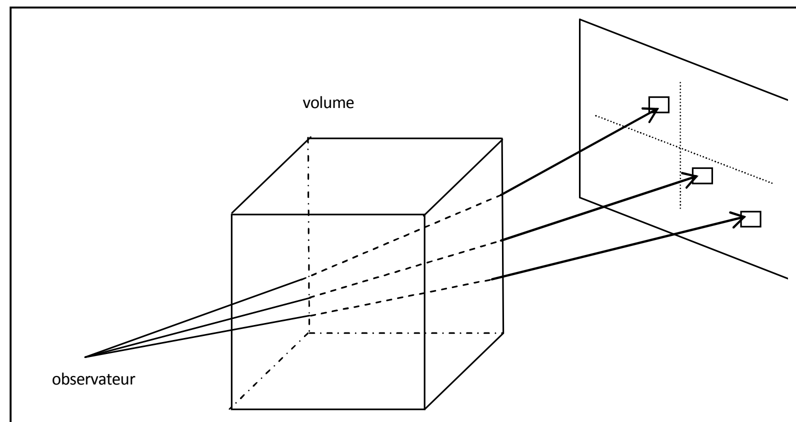


Figure 1.1 *: Rendu volumique direct*

3. Procédure de rendu volumique :

Les méthodes de rendu volumique génèrent des images d'un ensemble de données 3D volumiques sans l'extraction explicite des surfaces géométriques à partir des données. Ces techniques utilisent un modèle optique pour mapper les valeurs de données aux propriétés optiques, tels que la couleur et l'opacité. Durant le rendu, les propriétés optiques sont accumulées le long de chaque rayon pour former une image des données (voir Figure 1.2).

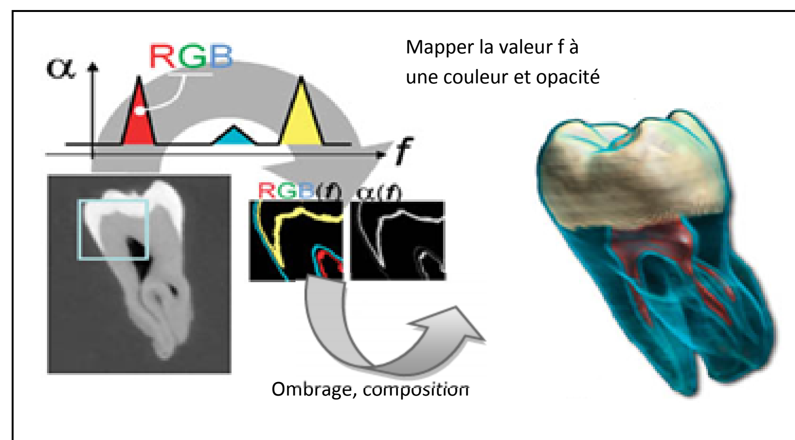


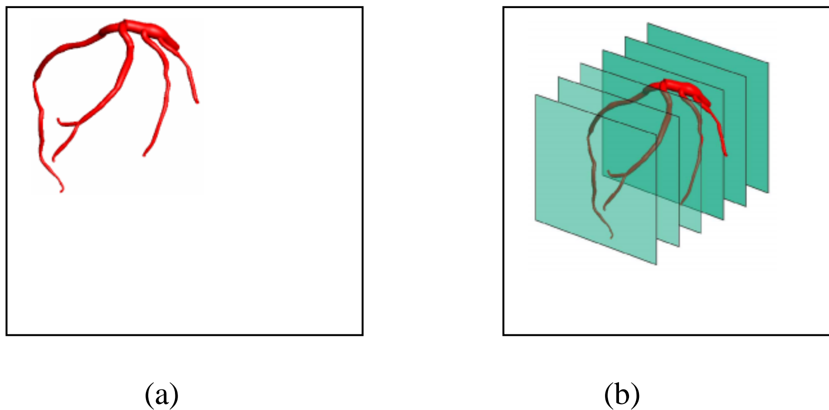
Figure 1.2*: le processus de rendu volumique*

Le rendu d'un volume est un processus complexe, il est composé des étapes suivantes :

3.1 Classification : la classification est une étape de pré-traitement qui attribut l'opacité à chaque voxel. Une valeur entre 0 et 1 décrivant la quantité de la lumière absorbée par le voxel. Cette étape de classification permet à l'utilisateur de mettre l'accent sur les structures du volume en rendrant visible les voxels d'une grande opacité. Les voxels éloignés seront rendues transparents **[GE00]**.

3.2 Plans de découpage (clipping planes):

Une autre façon de voir à l'intérieur du volume est l'utilisation des plans de découpage (voir figure 1.3). Un plan de découpage, comme son nom l'indique, supprime la section du volume situé dans l'un des demi- espaces dans lequel la scène est divisée par le plan. En faisant déplacer et tourner le plan de découpage, des tranches arbitraires du volume peuvent être affichées **[KE05]**.



(a) (b)

***Figure 1.3** : Plans de découpage*

3.3 Composition :

Vu la présence de plusieurs voxels projetés sur le même pixel, certains pourraient être transparents, ces voxels doivent être combinées pour former la couleur du pixel final. L'étape de combinaison est appelée composition et est en général une opération complexe et non linéaire, il n'est pas facile de prévoir la valeur finale d'un pixel après avoir combiner les valeurs des couleurs obtenues à partir d'un ensemble de valeurs de voxel. Même un changement léger dans la fonction de transfert, qui prend une valeur de voxel en entrée et donne une valeur de couleur comme sortie, peut produire une couleur finale totalement

différente. Il existe plusieurs opérateurs de composition disponibles. La composition peut être faite soit en avant à l'arrière ou d'arrière à l'avant **[GF01]**.

3.4 Composition de rayon partiel :

Une propriété importante de la composition d'avant à l'arrière et d'arrière à l'avant est que les rayons partiels peuvent être composés. Cela signifie que le rayon peut être divisé en de petits rayons qui peuvent ensuite être traitées de façon indépendante et les segments de rayon seront combinés par la suite pour former la couleur finale du pixel **[KE05]**.

Les images sont ainsi créées en échantillonnant le volume le long de tous les rayons de visualisation et en accumulant les propriétés optiques résultantes **[GE00].**

4. Les défis du rendu volumique:

Le rendu volumique est dans plusieurs égards plus difficile que le rendu polygonal traditionnel. La raison la plus importante est la grande taille des ensembles de données volumiques, qui peuvent atteindre plusieurs centaines de méga-octets, voire des dizaines de giga-octets pour les études géologiques. Les exigences pour la qualité du rendu et la taille des ensembles de données qui ne cesse d'augmenter avec un matériel plus puissant et disponible, et des solutions intelligentes doivent être prises pour donner des résultats satisfaisants.

Un autre défi significatif concerne la correction de la visualisation. Lorsque le rendu volumique est utilisé pour inspecter des détails dans l'ensemble de données, il est important que même des petits détails soient visible et aucun artefact ne soit introduit.

5. Applications du rendu volumique :

Les utilisateurs du rendu volumique les plus connus sont l'industrie de l'imagerie médicale et les compagnies pétrolières qui font des études géologiques dans leur recherche de pétrole et de nouveaux réservoirs de gaz. Il existe de nombreux autres usages, et leurs utilisations augmentent, du fait que la technologie d'échantillonnage améliore la visualisation et le matériel devient plus abordable et plus facile à utiliser. Nous présentons dans ce qui suit un aperçu de certaines utilisations du rendu volumique :

- **L'imagerie médicale** est l'une des premières applications du rendu volumique et peut-être le plus important. L'IRM [(1)] et les scanners produisissent des données d'imagerie 3D, permettant aux médecins d'être en mesure de faire pivoter, de zoomer, ainsi que decolorer l'ensemble de données, pour examiner les détails. En plus de ces propriétés le rendu volumique permet d'introduire des parties transparentes du volume pour mieux se concentrer sur les détails pertinentes. Un développement plus récent est la planification chirurgicale, où le chirurgien peut examiner et virtuellement travailler sur un volume de numérisation du patient avant l'opération proprement dite, afin de mieux préparer et découvrir les problèmes à l'avance.
  Le chirurgien peut aussi avoir la visualisation au cours de l'opération, et même voir des données numérisées en temps réel. Le chirurgien peut opérer un patient à distance avec un robot qui simule ses mouvements **[DR08]**.
- **Modélisation des phénomènes physiques**, tels que la turbulence des océans, des tempêtes solaires magnétiques, la couche d'ozone, les ouragans et les typhons. L'utilisation du rendu volumique, peut fournir des indications sur la façon dont ces phénomènes se développent.
  La dynamique des fluides est souvent utilisée pour modéliser l'écoulement de l'air sur les carrosseries de voitures ou le fuselage des avions pour minimiser les résistances, et lorsqu'on utilise le rendu volumique pour afficher les données en 3D des zones de tourbillon élevé (la composante de rotation de l'écoulement) peuvent être rapidement identifiés et l'utilisateur peut également obtenir une sensation globale de l'écoulement dans le système **[KE05]**.
- **Le contrôle du désassemblage des objets** peut être utilisé pour examiner l'intérieur d'un objet sans le démonter. Comme le désassemblage peut perturber l'objet et qui conduit à invalider l'examen, ou il ya un désir de préserver l'objet intact, ce qui est souvent le seul moyen de recueillir les données d'intérêt **[KE05]**.

(1)IRM : Imagerie par résonance magnétique (IRM), utilisant l'effet d'un champ magnétique intense sur le spin des protons. C'est un procédé tomographique, permettant d'obtenir des "coupes virtuelles" du corps suivant trois plans de l'espace

6. Les algorithmes de rendu volumique direct

6.1 Lancer de rayons

L'algorithme de lancer de rayons est l'algorithme le plus célèbre dans la catégorie "image-order". Il s'agit d'émettre des rayons à partir du point de vue jusqu'au plan de l'image pour chaque pixel en traversant le volume. Lorsqu'un rayon traverse le volume, le volume est échantillonné selon un certain nombre d'intervalles, en général, un ou deux par voxel. La valeur à chaque point d'échantillonnage est obtenue par une interpolation trilinéaire (voir ci-après) des valeurs voisines de ce point. Les contributions de ces échantillons sont accumulées jusqu'à ce que le rayon ait quitté le volume **[AM04]**.

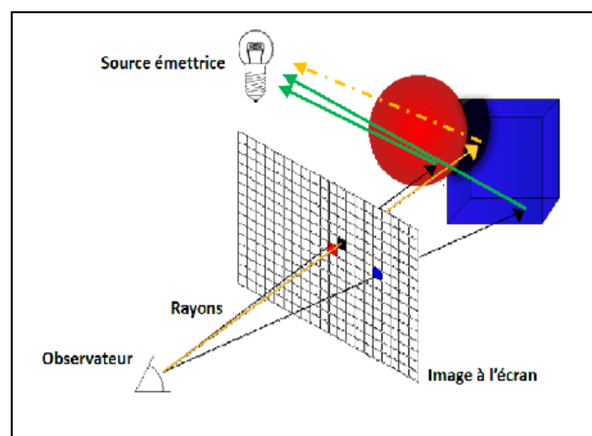


***Figure1.4** : le lancer de rayon : concept de base*

L'avantage majeur du lancer de rayons est sa qualité du rendu. Cependant, cet algorithme étant un algorithme de type image-order, il n'accède pas aux données du volume dans l'ordre naturel du stockage, car il le traverse dans des directions arbitraires.

En fait, il passe un temps considérable à calculer les positions des points d'échantillonnage et à effectuer les calculs d'adresses au lieu de calculer le rendu proprement dit. De plus, les rayons peuvent passer plusieurs fois par le même voxel, et par conséquent il est obligatoire de recharger des données déjà chargées précédemment. Enfin, l'élimination des régions vides du volume, bien que faisable, est moins facile qu'avec des méthodes de type object-order, et donc plus coûteuse **[CC07]**.

L'interpolation trilinéaire :

L'interpolation trilinéaire affecte au point (x, y, z) ϵR^3l'intensité pondérée des 8 points voisins selon la formule :

Interpolation(t,u,v)= (1-t) (1-u)(1-v).V_{000}
+t(1-u)(1-v).V_{100}

$+ (1\text{-}t)u(1\text{-}v).V_{010}$
$+ (1\text{-}t)(1\text{-}u)v.V_{001}$
$+ t(1\text{-}u)v.V_{101}$
$+ (1\text{-}t)uv.V_{011}$
$+tu\ (1\text{-}v).V_{110}$
$+tuv.V_{110}$

Où $\begin{cases} t = x - E(x) \\ u = y - E(y) \\ v = z - E(z) \end{cases}$

Les V_{ijk} , $k \epsilon \{0.1\}$ sont définies sur la figure suivante:

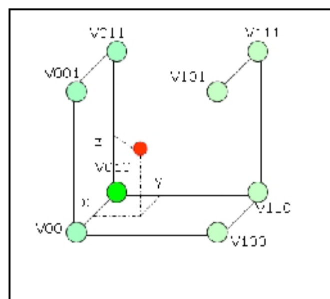


***Figure1.5** : Interpolation trilinéaire : on associe au point (x,y, z) ici en rouge l'intensité pondérée des 8 points Vi jk, k ϵ {0;1}: plus le point est proche d'un sommet plus ce sommet compte dans la pondération*[MA05].

6.2 Splatting (éclaboussures) :

Cet algorithme sert à projeter le volume voxel par voxel sur le plan image. En projection parallèle, l'empreinte d'un voxel sur ce plan de vue est constante pour tous les voxels. Comme un voxel ne se projettera pas exactement sur un pixel, il est nécéssaire d'utiliser un filtre pour calculer la contribution en couleur et en transparence aux pixels voisins. Comme cet algorithme itère sur le volume, il peut parcourir celui-ci selon son ordre naturel de stockage. En revanche, le calcul du filtre de rééchantillonnage est très coûteux. En théorie, cet algorithme peut fournir les mêmes images qu'avec l'algorithme du lancer de rayon. En pratique, vu le calcul des paramètres des filtres est difficile à réalisées, des approximations sont utilisées, celles-ci permettent soit d'obtenir une exécution efficace, soit des images de qualité, mais jamais les deux simultanément **[FL03]**.

6.3 Algorithme Shear-Warp (projection orthogonale-déformation)

L'algorithme shear-warp s'appuye sur la factorisation de la transformation liée au point de vue pour simplifier la projection du volume vers l'image. Dans cet algorithme, on n'itère pas sur les pixels de l'image, mais sur les voxels de la scène. Cependant, on évite la complexité des algorithmes de splatting en décomposant la projection des voxels sur le plan image en deux étapes :

1. Au début, on effectue une projection des voxels sur un plan objet parallèle à l'une des faces du volume à visualiser.
2. ensuite, le plan objet est projeté sur le plan image, le coût de projection étant très bas car il ne s'agit plus de projections d'informations volumiques mais bidimensionnelles.

La factorisation shear-warp consiste à considérer un système de coordonnées intermédiaires. Dans cet espace objet transformé, les rayons liés au point de vue sont parallèles au troisième axe de coordonnées (voir figures 1.6 et 1.7). Un algorithme basé sur cette transformation dans l'espace objet intermédiaire se fait en deux étapes :

1. une première étape de composition appelée “shear”, composée de la transformation des données du volume dans l'espace objet transformé et de l'accumulation de ces données sur le plan image. Cette étape crée une image intermédiaire “distordue” correspondant à l'espace objet transformé.
2. une deuxième étape appelée “warp”, transformant l'image intermédiaire en image finale.

Passer par une image intermédiaire permet aux lignes de balayage des données du volume d'être alignées avec les lignes de balayage de l'image intermédiaire. On balaye alors l'objet et l'image simultanément dans leur ordre de stockage. Cet algorithme combine les avantages des algorithmes d'ordre objet et des algorithmes d'ordre image. Les conditions d'utilisation des techniques d'optimisation séquentielles telles que l'exploitation de la cohérence de données et la terminaison anticipée de rayon sont réunies grâce à ce parcours simultané des deux structures de données (objet et image) **[KE06] [FL03]**.

L'implantation de l'algorithme du shear-warp peut être décomposée en trois unités fonctionnelles : calcul d'une table d'opacité, l'étape de composition et l'étape de warp de l'image intermédiaire.

Le calcul de la table d'opacité est un pré-calcul d'opacité. Le calcul des opacités est dépendant du point de vue, il dépend de l'intervalle d'échantillonnage. Le volume de données contient des opacités associées à un intervalle x_0. On calcule à chaque nouveau point de vue

la correction nécessaire à chacune de ces valeurs pour le nouvel intervalle x, d'où le calcul d'une table d'opacité.

Pour la phase de composition qui est la combinaison des étapes de transformation du volume et d'accumulation, on doit échantillonner chaque coupe de voxels lorsqu'elle est translatée dans l'espace objet déformé. L'interpolation utilisée est l'interpolation bilinéaire (contrairement à la plupart des algorithmes de rendu volumique qui utilisent l'interpolation trilinéaire) utilisant deux lignes de balayage d'une coupe pour produire une ligne résultat dans l'image intermédiaire **[DP04]**.

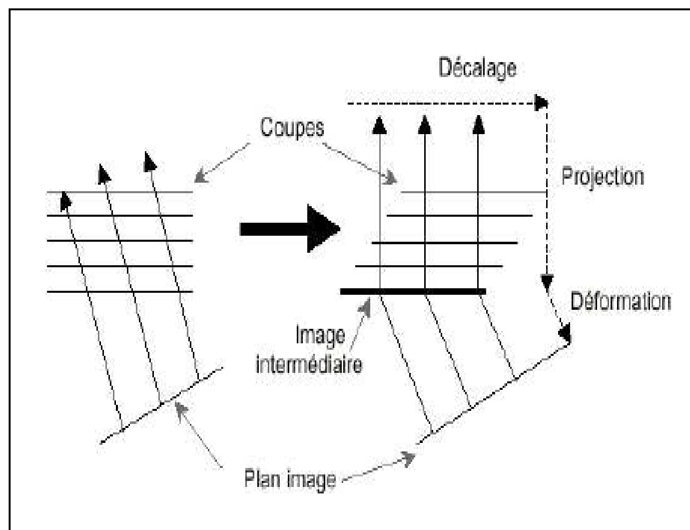


Figure 1.6 *: L'algorithme du shear-warp avec une projection parallèle*

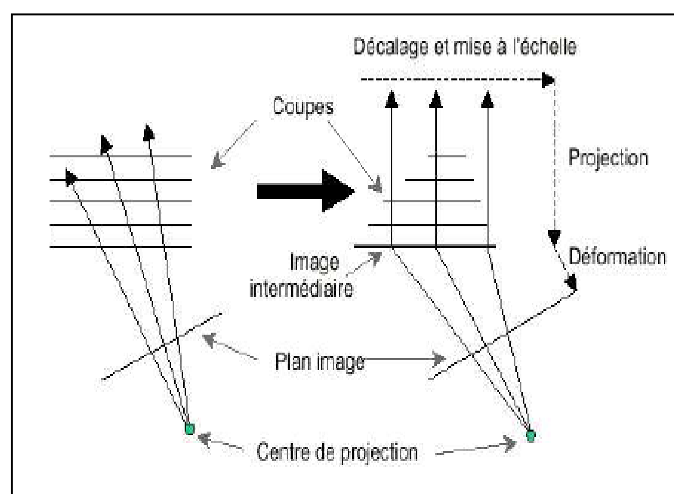


Figure 1.7 *: L'algorithme du shear-warp avec une projection perspective*

L'étape "warp" consiste en la projection de l'image intermédiaire distordue sur le plan image, pour obtenir l'image finale.
Il est clair que le principal avantage de cette méthode est sa rapidité de rendu. Elle cumule ainsi beaucoup d'avantages qui vont dans ce sens. La mémoire est parcourue de manière optimale que ce soit pour l'accès au volume ou pour l'accès au plan image. La projection est aussi très simplifiée (donc rapide), lorsqu'il y a correspondance entre un voxel et un pixel. Cependant cette méthode comporte beaucoup d'inconvénients liés, essentiellement à la qualité de l'image résultante **[DP04]**.

7. Les techniques de rendu volumiques :

7. 1. Rendu à base de points:

Vu l'augmentation croissante de la complexité des scènes, l'utilisation de maillages de primitives géométriques pour réaliser le rendu est apparue de moins intéressant comparativement à la manipulation directe de points. Le rendu à base de points est un autre axe pour la visualisation. Il s'agit de ne considérer qu'un nuage volumique de points. L'avantage de ces méthodes de rendu est l'utilisation d'une simple primitive graphique : le point. Un autre avantage est le fait que ces méthodes n'ont pas besoin d'effectuer une phase de pré calcul (génération de texture, création de maillage, . . .) à priori **[DP04]**.

Le point 3D comme primitive de rendu :

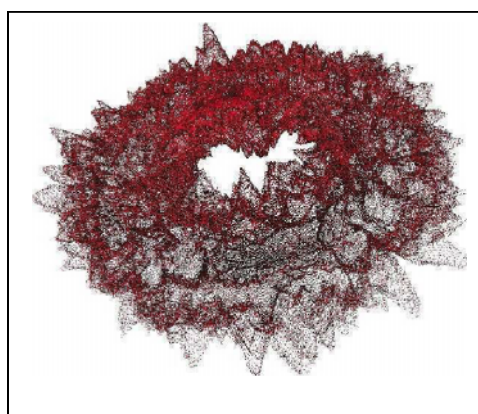


Figure 1.8 *: Objet 3D représenté sous forme d'un nuage de points*

Tout d'abord la scène définie par un maillage de polygones est transformée en un ensemble de points.

Cet ensemble de points est généré par discrétisation régulière de la surface à l'aide des coordonnées de texture : il est donc défini comme une grille 2D de points de coordonnées (u; v) dans l'espace de la texture. Chaque point de cette grille stocke la position 3D de la surface (x; y; z) ainsi que la couleur (r; v; b) du point.

Cet ensemble de points déplacés est ensuite transformé et projeté à l'écran. La contribution de chaque point projeté à la couleur d'un pixel donné de l'écran est fonction de sa distance avec le centre de ce pixel, maximum au centre et décroissante en fonction de l'éloignement. Pour calculer cette contribution, la distance point/pixel est pondérée par une fonction d'atténuation, représentée par un filtre Gaussien **[DF04]**.

7.2 Rendu à base d'images :

Les techniques de rendu à base d'images (*IBR*, ou Image-Based Rendering.) utilisent principalement des images comme primitives de rendu, c'est à dire que les informations contenues dans une image sont utilisées pour synthétiser de nouvelles images.

Le concept d'image a évolué graduellement. La texture, la plus simple des images (un tableau 2D de pixels, un pixel stockant une couleur) ne permet pas, quand elle est déformée, de prendre en compte les effets de parallaxe. Ces effets peuvent être pris en compte en ajoutant une profondeur à chaque pixel. Une telle image contenant couleurs et profondeurs est appelée *image de profondeurs* (*depth image*). Néanmoins, une seule image de profondeur ne contient pas assez d'informations pour créer un nouveau point de vue : les effets de parallaxe génèrent, par exemple, l'apparition de certaines parties de la scène qui ne sont pas dans l'image de base (**effets d'occlusion** ou **de masquage**, voir la figure 1.9). Pour prendre en compte ce problème, les *images à plans de profondeurs* (*LDI*, Layered Depth Images) stockent par pixel, non plus simplement une couleur et une profondeur, mais une liste de couleurs et de profondeurs, c'est à dire un échantillonnage des parties cachées **[DP04]**.

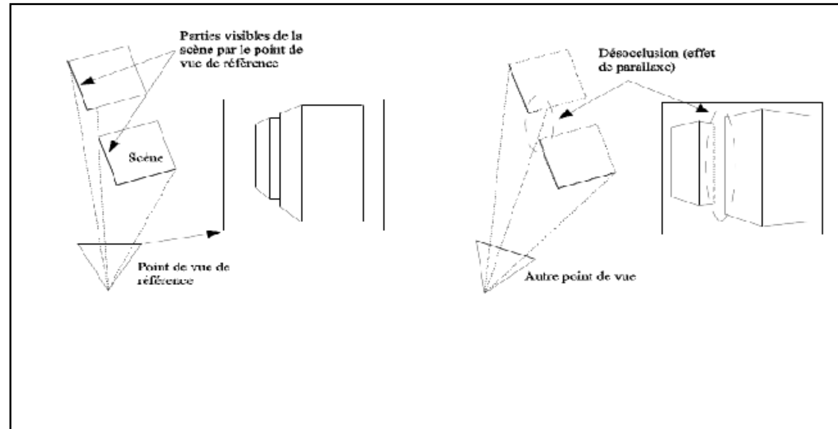


Figure 1.9*: Illustration d'un effet de parallaxe : la désocclusion. L'image de référence utilisée pour afficher la scène selon un autre point de vue ne contient pas assez d'informations et des trous apparaissent dans l'image à l'endroit ou on devrait voir une partie de la scène.* **[DP04]**

Ces trois types d'images : *texture*, *image de profondeurs* et *image à plans de profondeurs* constituent les primitives de base des techniques de rendu à base d'image.

Par la suite, nous utiliserons le terme de *texel* (*Texture Element*) pour désigner les données stockées dans une texture.

Les images de profondeurs peuvent être de deux types en fonction de ce que représente cette information de profondeur. Le premier type correspond au *champ de hauteurs (Height Field)* ou *carte d'élévation,* utilisé par exemple dans le *placage de bosselures* (*Bump-mapping*) et représentant un déplacement orthogonal à la surface, par pixel. Le second type d'image de profondeurs est le *tampon de profondeurs* (*Z-Buffer*), dont la profondeur représente la distance caméra/surface **[DP04]**.

Dans le premier cas (figure 1.10, image de gauche), la hauteur représente la distance entre une surface de référence et la surface réelle, sous forme d'une élévation perpendiculaire à la normale au plan de référence. Par la suite nous dénommerons une telle image de profondeurs par le terme *carte de hauteurs.*

Dans le second cas (image de droite), la profondeur d'un pixel correspond à la distance entre le centre de projection de la caméra et la surface. Cette distance est la composante z de la coordonnée 3D d'un point dans l'espace de la caméra. L'image de profondeurs correspond au *z-buffer* de la scène et nous utiliserons le terme *carte de profondeurs* pour dénommer une telle image.

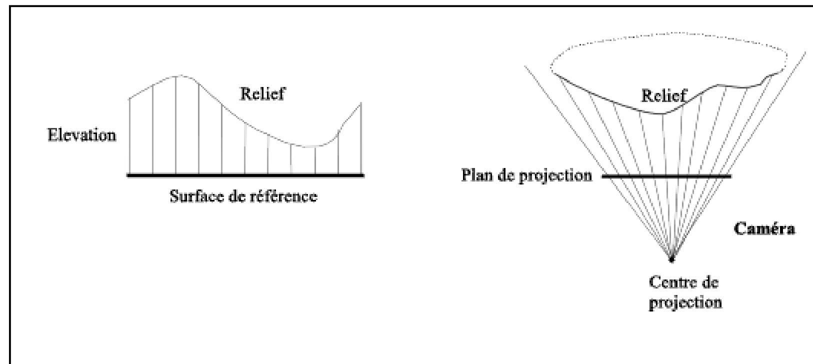


***Figure 1.10** : Les deux principaux types d'images de profondeur. A gauche : carte d'élévations. A droite: carte de profondeurs issue de la projection de la scène sur le plan de projection d'une caméra.*

La façon dont sont utilisées ces informations permet de distinguer les algorithmes à base d'images.

En effet, ils utilisent, déforment et combinent une ou plusieurs images selon différentes techniques qui vont du simple placage de texture au transfert épipolaire en passant par la déformation 3D ainsi que différentes méthodes d'interpolation.

Les méthodes IBR (image based rendering) sont également caractérisées et différenciées par la façon dont il est fait usage des images :

1. La scène synthétisée à la fin peut simplement être composée en partie d'éléments obtenus à l'aide de techniques basées images, ou être intégralement obtenue à l'aide d'images. Dans le premier cas, il s'agit par exemple de techniques telles que les *imposteurs* ou les *sprites.*
2. Les techniques de rendu basé image (IBR) peuvent être *pures*, c'est à dire n'utiliser que des images, ou être combinées avec les techniques de rendu standard, soit pour accélérer certaines techniques de rendu existantes, soit pour tirer parti des avantages des deux approches. Nous parlerons alors de techniques *hybrides* basées images / basées géométrie.
3. Finalement, les techniques d'IBR peuvent être distinguées en fonction de la façon dont sont obtenues les images utilisées pour les calculs.
   a) Elles peuvent être pré-acquises (comme dans le cas d'images réelles, par exemple des photographies) ou pré calculées (dans le cas d'images virtuelles).
   b) Les images peuvent également être calculées, dynamiquement, lors du rendu, en fonction des besoins **[BB03]**.

7.2.1. Placage de texture

Le placage de texture (*texture mapping*) est la plus simple et la plus ancienne des techniques à base d'images. Le principe est de remplacer un objet géométrique complexe par un objet plus simple sur lequel on applique une image de la surface de l'objet complexe d'origine. C'est une technique puissante, actuellement intégrée dans toutes les cartes graphiques qui obtiennent dans ce domaine des performances impressionnantes et toujours croissantes **[DP04]**.

a. Les étapes de rendu volumique basé texture :

En général, comme le montre la Figure 1.11, les algorithmes de rendu volumique à base de texture peuvent être divisés en trois étapes: (1) Initialisation, (2) Mise à jour, et (3) dessin. L'étape d'initialisation est généralement effectuée une seule fois. Les étapes mise à jour et de dessin sont effectuées autant de fois que l'application reçoit les entrées de l'utilisateur, par exemple, lorsque la visualisation ou les paramètres du rendu changent **[GE00].**

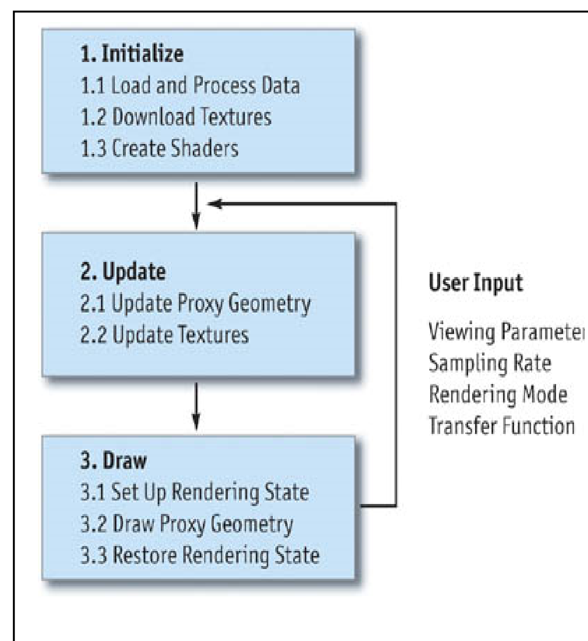


***Figure 1.11**: Les étapes d'un rendu volumique typique à base de texture*

- Au début de l'application, des volumes de données sont chargés en mémoire centrale. Dans certains cas, les ensembles de données doivent également être traitées avant de les collecter et de les télécharger dans la mémoire de texture.

- Après l'initialisation et à chaque fois les paramètres de visualisation changent, la géométrie est calculée et stockée dans les tables de vertex. Lorsque l'ensemble des données est stocké comme un objet de texture 3D, la géométrie se compose d'un ensemble de polygones, découpant le volume en tranches perpendiculaire à la direction de vision.
- Les tranches de polygones sont tout d'abord calculés en intersèctant les plans à découper avec les bords de la boîte englobante de volume et en classant les sommets résultants. Pour chaque sommet, les coordonnées de texture 3D correspondante sont calculées sur le CPU, dans un programme vertex.
- Quand un ensemble de données est stocké comme un ensemble de tranches de texture 2D, les polygones sont tout simplement des rectangles alignés avec les tranches. Bien qu'elle soit plus rapide, cette approche présente plusieurs inconvénients.

Premièrement, elle nécessite trois fois plus de mémoire, parce que les tranches de données doivent être dupliquées le long de chaque direction principale. La duplication de données peut être évitée mais au prix de performance, en reconstruisant les tranches à la volée.

Deuxièmement, le taux d'échantillonnage dépend de la résolution du volume. Ce problème peut être résolu en ajoutant des tranches intermédiaires et en effectuant une interpolation trilinéaire avec un pixel shader **[GE00]**.

b. Principe de la méthode de rendu volumique basé texture :

Le principe de la méthode consiste à échantillonner le volume de données à l'aide d'une série de polygones de support planaires. Ces polygones sont texturés avec une texture 3D (généralisation à un tableau 3D des textures 2D classiques) contenant soit les données qui seront transformées à la volée à l'aide d'une fonction de transfert (post-classification), soit directement les propriétés d'émission et d'absorption du matériau (pré-classification). Ces polygones sont ainsi dessinés d'arrière vers l'avant ou d'avant vers l'arrière par rapport au point de vue et combinés via la fonction de composition du matériel (blending). Les polygones de support peuvent être alignés selon un axe du volume ou parallèles au point de vue **[DP04]**.

Ce principe est illustré dans la figure 1.12 avec des polygones alignés sur le plan de vue.

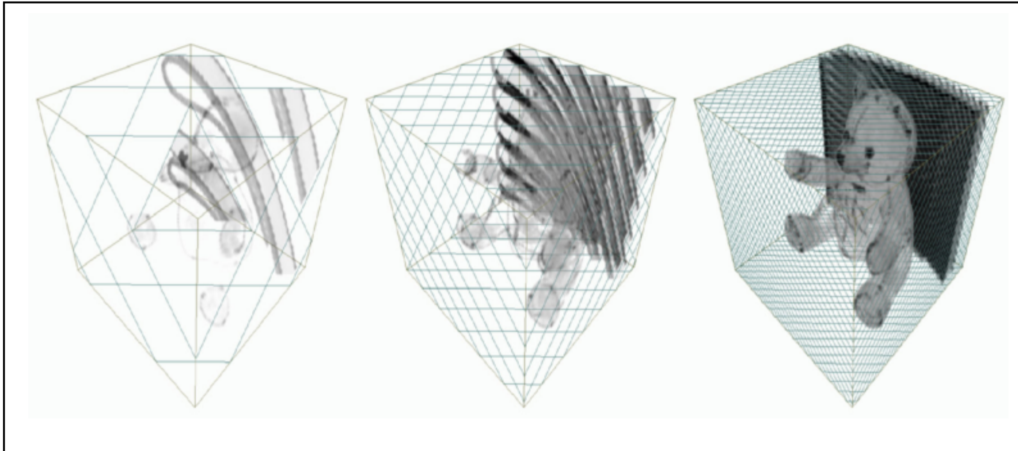


***Figure 1.12** : Visualisation de la série de polygones parallèles au plan de vue support de rendu des données volumiques.*

7.2.2. Rendu Volumique utilisant des « tranches » de textures 3D :

C'est ce qui s'appelle en anglais *3D texture sliced-based volume rendering*. Cette technique consiste dans un premier temps à associer la texture 3D des données volumiques à leur volume englobant (*Bounding Volume* ou *Bounding Box*) de sorte que les 8 sommets de la *BBox* correspondent aux 8 coins de la texture 3D (*mapping*). Ensuite des plans perpendiculaires à la direction d'observation sont générés et ajustés sur la *BBox*. Une carte graphique 3D qui supporte les textures 3D, échantillonne la texture 3D avec une interpolation trilinéaire par fragment correspondant pour le rendu de chaque pixel de l'image finale associée à la texture 3D (en général un fragment par tranche est associe à un pixel). Une valeur de texture (une couleur) peut être associée en tout point d'intérêt de la surface d'une tranche. On obtient donc des tranches de textures et la qualité du rendu volumique peut être ajustée en adaptant la distance entre les tranches avec le re-échantillonnage adéquate. Ensuite en rendant les plans en ordre arrière vers devant (*back-to-front)*, l'opération de composition (*compositing)* ou de mélange (*blending*) appropriée est réalisée, pour afficher la bonne couleur dans chaque pixel (encore géré par le GPU) **[VV01]**.

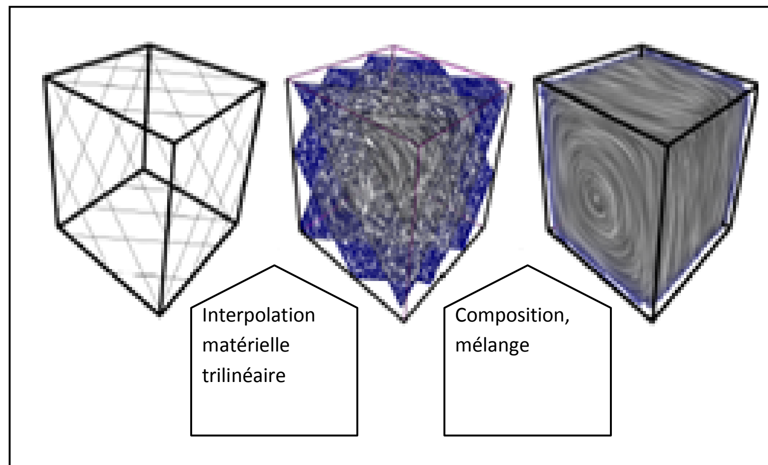


***Figure 1.13** : Illustration du rendu volumique à base d'une texture 3D échantillonnée (interpolation trilineaire selon des plans paralleles au plan d'observation)* **[VV01]**

Une texture 3D peut être construite à partir des données volumiques, ensuite elle peut être réutilisée avec des plans parallèles textures semi-transparents (avec les bonnes spécifications de coordonnées de texture). Et c'est la carte graphique qui fait le reste (conversion de coordonnées, extraction de texture, choix du niveau de *MIP-map*, calculs d'interpolation). Ces plans peuvent être alignés (*axis-aligned*) uniquement, selon l'un des trois axes si la carte graphique ne supporte pas les textures 3D ou alignes selon l'axe de vue (*viewing axis*). Signalons encore que le rendu utilisant les textures 3D permet de choisir le type de projection utilisé pour l'échantillonnage et le mélange des couleurs.

8. Illumination :

Une scène est également composée de sources lumineuses qui éclairent les objets dans la scène. Une phase de calcul d'illumination permet de calculer les informations pertinentes au niveau des objets pour le calcul effectif des couleurs des futurs pixels de l'écran. Encore une fois, cette phase de calcul ne s'effectue qu'au niveau des sommets des objets de la scène.

Le modèle d'illumination couramment employé se décompose en trois composantes de complexité croissante **[Ph00]** :

– une composante ambiante qui correspond à l'illumination globale de la scène.

L'intensité est alors $I = k_a I_a$

Où k_a correspond au coefficient de réflexion ambiante spécifique à l'objet et

I_a à l'intensité de la lumière ambiante.

- une composante diffuse qui correspond à l'application de la loi de Lambert sur l'illumination d'objets mats. L'intensité d'une source lumineuse réfléchie ne dépend que de son angle avec la normale à la courbe et du coefficient *Kd* de diffusion du matériau. L'intensité pour la source j est alors $I^j = k_d I_d^j \cos$
- une composante spéculaire qui modélise l'effet de la rugosité d'un objet. Cette composante dépend de l'angle α entre la direction de vision et celle de la source lumineuse. Phong a modélisé cette composante par $I^j = k_s I_s^j \cos^n$

  Où n est un coefficient d'atténuation spéculaire **[CC04]**.

Ces trois composantes sont illustrées par la figure 1.14

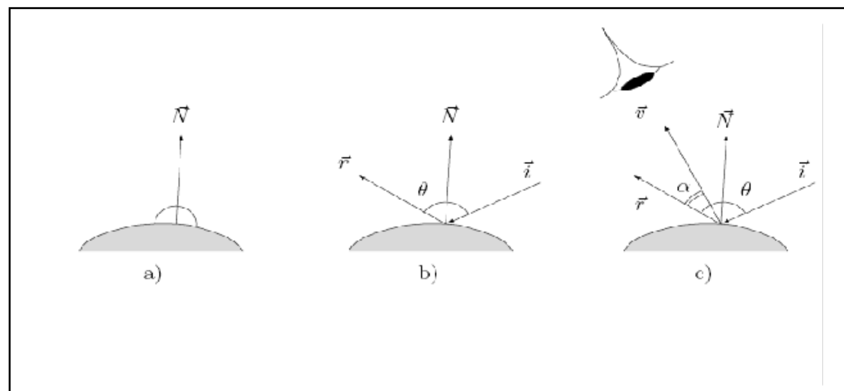


***Figure 1.14** : Modèle d'illumination : a) ambiante, b) diffuse, c) spéculaire.*

8.1. Modèles d'illumination :

Les modèles d'illumination définissent la manière dont on modélise la lumière à l'intérieur de la scène, et celle-ci interagit avec les objets. Ils définissent ainsi la qualité de rendu de l'image à l'écran : plus un modèle est complexe et plus il permet de prendre en compte de phénomènes physiques liées à la lumière, et donc plus le réalisme des images sera grand **[CC04]**.

Les calculs d'illumination représentent à l'heure actuelle la majorité du coût de calcul d'une image, et n'ont cessé de croître du fait de l'introduction de modèles d'illumination de plus en

plus sophistiquées. De plus, ces nouveaux modèles nécessitent des volumes de données extrêmement importants, qui dépassent les capacités des machines séquentielles classiques. Ce domaine est donc propice à l'utilisation du parallélisme.

8.1.1. Spécularité et diffusivité :

L'interaction entre la lumière et la matière au niveau macroscopique a été très étudiée par les physiciens. Ceux-ci ont dégagé deux types de comportement lorsqu'un faisceau lumineux est réfléchit sur une surface :

- réflexion spéculaire : il y'a réemission unidirectionnelle de la lumière avec un angle égale à l'angle d'incidence du rayon lumineux, et conservation de l'énergie. Ce type de réflexion est observable sur les objets finement polis, tels les miroirs

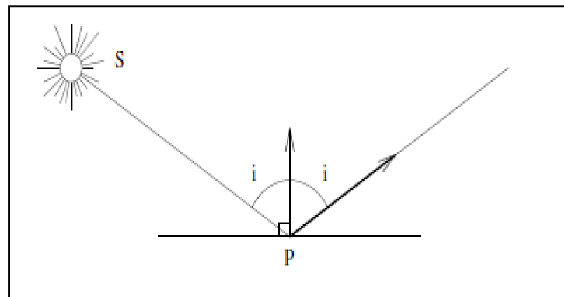


Figure 1.15 *: réflexion spéculaire*

- réflexion diffuse : il y'a réémission omnidirectionnelle isotrope de la lumière incidente, avec perte d'énergie. Les surfaces provoquant des réflexions diffuses, dites ''lambertiennes ', sont en général mates et rugueuses **[CC04]**.

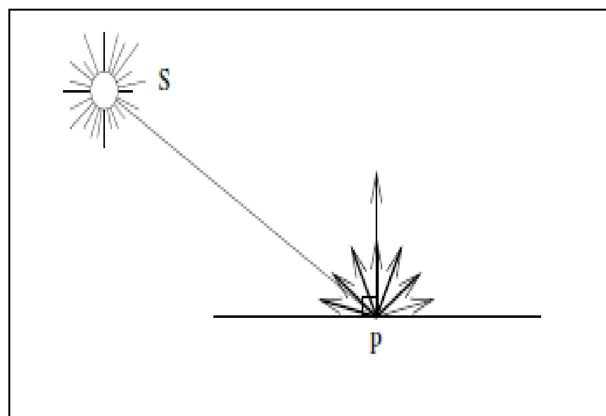


Figure 1.16 *: réflexion diffuse*

8.1.2 BRDF : Bidirectional Reflectance Distribution Function :

Le modèle diffus et le modèle spéculaire présentés dans les sections précédentes ne sont finalement que des cas particuliers du comportement de la lumière. De façon plus générale, le matériau qui caractérise une surface réagit différemment en fonction des angles d'incidence, et l'émission dépend des angles de sortie. En utilisant les angles polaires θ et pour caractériser une direction autour d'un point, on peut ainsi définir une fonction de quatre paramètres : θ_{in}, $_{in}$,θ_{out}, $_{out}$, permettant de quantifier le flux d'énergie sortant dans une direction donnée en (θout, out) par rapport à un flux incident dans une direction donnée en

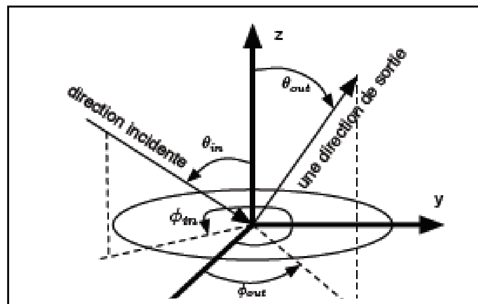


***Figure 1.17** : Les angles polaires comme paramètres d'une BRDF*

(θin, in) (Figure 17). Cette fonction se nomme BRDF, pour *Bidirectional Reflectance Distribution Function* **[PC06]**.

La fonction BRDF est présentée par l'équation suivante :

$$\rho\left(x, \lambda, \underset{\omega_{in}}{\longrightarrow}, \underset{\omega_{out}}{\longrightarrow}\right) = r(x, \lambda, \underset{\omega_{in}}{\longrightarrow}, \underset{\omega_{out}}{\longrightarrow})/cos\theta_{out}$$

où la fonction $r\left(x, \lambda, \underset{\omega_{in}}{\longrightarrow}, \underset{\omega_{out}}{\longrightarrow}\right).d\omega_{out}$ est la probabilité qu'un photon d'énergie $h.c/\lambda$

arrivant au point *x* depuis la direction ω_{in} soit renvoyé autour de la direction ω_{out}

la fonction BRDF dépend donc de :

- la matière qui constitue la surface.
- la longueur d'onde du rayon incident.
- des directions incidente ω_{in}et réfléchie ω_{out}
- la géométrie locale de la surface

Comme pour la limite entre géométrie représentable et micro-géométrie modélisée, on peut s'interroger sur l'échelle à laquelle appliquer le principe de la BRDF. Si l'on représente un arbre, on peut distinguer les BRDFs des feuilles, et de l'écorce. Si en revanche on veut représenter une forêt dans un paysage, il est sans doute judicieux de proposer une BRDF comme réaction globale de la forêt au rayonnement solaire.

Pour associer une BRDF à la représentation d'un objet réel, il peut être utile de la mesurer pour en connaître les caractéristiques, et ainsi la modéliser elle aussi. L'acquisition d'une BRDF pour un objet de taille raisonnable peut se faire avec un dispositif d'éclairage et de mesure permettant de relever les flux d'énergie dans des positions arbitraires ; pour un macro-objet, il faut déployer d'autres techniques d'évaluation. Dans tous les cas, il est probable que l'ensemble des mesures ne soit pas utilisable directement par un algorithme, et qu'il faille passer par une étape de modélisation de la BRDF. L'utilisation d'une BRDF, au prix d'un modèle d'éclairement coûteux, permet d'obtenir un comportement plus réaliste de la lumière dans les images de synthèse. Cependant, cela ne suffit pas encore à représenter tous les comportements de la lumière couramment observables à notre échelle **[PC06]**.

9. Conclusion

Dans ce chapitre, nous avons essayé de présenter une introduction au rendu volumique; tout d'abord, nous avons exposé un état de l'art rassemblant quelques algorithmes de rendu volumique, puis nous avons projeté la lumière sur une étude permettant de montrer les techniques qui sont utilisées dans le champ du rendu volumique. Finalement, il est judicieux d'inclure une partie discutant sur l'illumination et les modèles existants vu son influence directe sur la qualité de rendu.

Quand on dit rendu volumique, l'esprit tourne rapidement vers les cartes graphiques, vis-à-vis son utilité directe sur la visualisation 3D. Pour cela, le sujet de notre chapitre suivant est l'utilisation des GPUs dans le rendu volumique.

Chapitre 02 :

Utilisation des GPUs dans le rendu volumique

Chapitre 2: Utilisation des GPUs dans le rendu volumique

1. Introduction et objectifs attendus :

Durant les dernières années, la technique du rendu volumique basé GPU a atteint un niveau haut de maturité, en produisant des résultats de haute qualité et avec un taux d'affichage interactif. De nos jours, certaines techniques ont été implémentées directement dans le hardware des cartes graphiques, et profitent la puissance de calcul croissante des GPU. L'orientation actuelle dans le domaine du rendu temps-réel exige donc d'effectuer le maximum de calculs graphiques avec les GPUs, vu que les performances de ces dernières sont plus élevées.

Les cartes graphiques modernes incluent un GPU qui est capable d'exécuter des simples vertex et fragment shader directement sur la carte elle-même. L'objectif principal est de permettre d'obtenir un rendu en temps réel, cependant, les jeux d'instructions disponibles ont été suffisamment généraux pour exécuter d'autres types de calcul. Ce chapitre met l'accent sur l'existence des possibilités de faire usage du GPU dans le domaine du rendu volumique, en cherchant toujours l'accélération du processus de rendu, l'augmentation de la qualité de visualisation, ou les deux.

Dans ce chapitre, nous allons aborder les caractéristiques matérielles des GPUs, leurs évolutions et enfin une description détaillée du pipeline graphique ainsi que les shaders.

3. Introduction aux GPUs :

3.1. Définition de GPU:

Un GPU est un processeur spécialisé dans les calculs graphiques, il est donc moins flexible qu'un CPU (*Central Processing Unit* ou Unité centrale de Traitement), mais permet en contre-partie des exécutions en parallèle et l'obtention de meilleures performances d'affichage spécialisé pour le traitement des sommets et des pixels **[VV07]**.

La carte graphique permet de traiter les informations concernant l'affichage afin de les envoyer au moniteur. Initialement, ces cartes servaient uniquement à traiter l'affichage. Avec la généralisation de la 3D, elles soulagent désormais le processeur pour le traitement des calculs 3D, grâce à une architecture entièrement dédiée **[AP05]**.

3.2 Apports du GPU :

Pour aboutir à un rendu temps réel, nous devons utiliser des méthodes permettant un taux d'affichage d'au moins 24 images par seconde (*FPS*, Frame per second). Les cartes graphiques déchargent un peu le CPU en permettant d'effectuer matériellement une partie du processus de visualisation. C'est à dire que certains des algorithmes de base utilisés en rendu sont implantés dans un ou plusieurs processeurs auxiliaires (Graphics Processing Unit.), indépendants du processeur central, et dédiés exclusivement à cette tâche. Ces cartes graphiques ouvrent de nouvelles perspectives. Il est désormais possible d'afficher interactivement des scènes composées de millions de primitives géométriques. De plus, des algorithmes comportant des calculs lourds peuvent maintenant être envisagés pour la synthèse d'images en temps réel et l'agrandissement de la flexibilité de ces cartes permet d'implanter des algorithmes de complexité croissante **[DP04]**.

3.3 Evolution des cartes graphiques :

Les anciennes cartes graphiques ont été des simples frames buffers (tampons d'images) dont la seule tâche était d'afficher le contenu de la mémoire tampon à l'écran d'ordinateur. L'ordinateur hôte devrait faire tout le travail en ce qui concerne la détermination de la couleur de chaque pixel avant de les remettre à la carte graphique. Comme le coût et la taille des transistors a été réduit, il existe une tendance dans l'industrie à implémenter sur le matériel les fonctionnalités précédemment effectués au niveau du logiciel, étant donné le matériel spécialisé exige toujours, beaucoup de transistors et est généralement plus rapide que le matériel généralisé. Pour l'infographie cela sert à une transition d'un nombre élevé d'étapes dans le pipeline graphique du CPU hôte vers la carte graphique.

- Les premières cartes accélératrices 3D ont eu un pipeline à fonction fixe, ce qui signifie que la carte est considérée comme une boîte noire dans laquelle les sommets et les textures sont introduites à l'entrée du pipeline et les pixels sont produits à la sortie (figure 2.1). Diverses options peuvent être définies pour modifier un certain nombre de constantes utilisées dans le pipeline **[JK03]**.

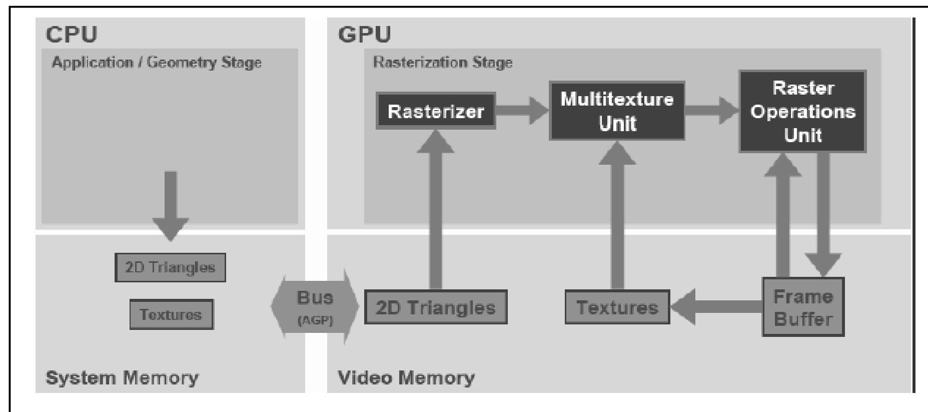


Figure 2.1*:les premières cartes graphique*

- Cela a changé avec l'introduction de GPU, qui permet au programme d'être insérer à deux points dans le pipeline: la transformation de vertex et la rastérisation. Ces programmes sont connus respectivement par des vertex shaders et fragment shaders. Ces programmes peuvent effectuer des calculs mathématiques et des opérations logiques et ont la possibilité de transformer leur entrée en n'importe quelle manière, ou même les rejetés, après d'écrire le résultat en sortie. Des données supplémentaires peuvent être fournies par le programme d'application en tant que variables. L'utilisation des GPU offre une plus grande flexibilité par rapport au pipeline fixe, et les algorithmes exigeants peuvent être implémentés dans le logiciel, tant que le matériel reste accélérateur et capable d'exécuter de manière interactive.

Pendant leurs introductions, les GPUs avaient un inconvénient significatif, qui concerne l'implémentation de ses programmes qui devait être écrits dans un langage assembleur. Cela provoque que le GPU doit consommer beaucoup de temps pour expérimenter avec des algorithmes différents, par conséquent l'utilité est limité quelque peu. Une solution à ce problème est apparue avec l'introduction de Cg, un langage de programmation de haut niveau pour les GPUs **[CC01]**.

4. Programmation GPU:

Cg, ou C graphique, est semblable au langage de programmation C dans la syntaxe et fournit au développeur des structures de haut niveau telles que les vecteurs et les matrices, les déclarations si, les boucles 'pour' et les appels de fonction. Les fonctions de bibliothèques sont fournit pour des fonctionnalités communes, telles que le calcul du produit scalaire ou de multiplication de matrice, qui s'exécutent souvent plus vite que la mise en œuvre directe, car

ils peuvent profiter d'instructions spéciales au niveau du matériel. Le programme Cg est compilé et optimisé, généralement au moment de l'exécution, pour le matériel spécifique GPU sur laquelle elle doit être exécutée.

Cg est un langage de haut niveau pour le développement de vertex shader et les fragments shader. Deux autres langages de ce type existent également, High Level Shading Language (HLSL), de Microsoft, qui est identique à Cg, mais implémenté pour l'API Direct3D plutôt que OpenGL et OpenGL Shading Language, OGSL.

Bien que Cg est comme le langage C dans la syntaxe, il est très différent du C normal car il est adapté spécialement pour le traitement des graphiques. L'une des différences les plus importantes est l'introduction d'un certain nombre de types de données additionnels supportés directement dans le langage **[RS00]**.

Chaque programme Cg doit générer une sortie, et également prendre une ou plusieurs entrées. Dans le cas des vertex shaders, l'entrée c'est les coordonnées des sommets et de données supplémentaires comme la couleur de vertex, les coordonnées de la texture, et toutes les données fournies par l'application hôte.

La sortie est formée par des nouvelles coordonnées de sommets transformés depuis l'espace local à l'espace d'image et une nouvelle couleur et des coordonnées de texture qui peuvent également être incluses **[CC07]**.

Les entrées du fragment shader sont la sortie générée à partir des vertex shaders ainsi que toutes les variables et les textures fournies par l'application hôte. Le fragment shader peut alors accéder aux textures et effectuer les calculs nécessaires pour déterminer la couleur du pixel final.

L'accès à la mémoire peut être simulé par des recherches de texture dépendant (*dependent texture lookups*,) i.e.; la valeur de la texture à partir d'une texture est utilisée comme une coordonnée pour rechercher une valeur dans une seconde texture. La sortie du fragment shader est une valeur de couleur écrite dans le frame buffer.

La puissance du calcul en virgule flottante se trouve dans le moteur de fragment, et les applications qui souhaitent profiter de cette capacité GPU, ne font pas souvent appel à un vertex shader; un simple passe à travers le fragment shader est utilisé **[SL06]**.

4.1 GPUs Programmables :

Les GPUs modernes sont dirigés d'opérer sur un flot de données large et continu des sommets et des fragments. Le terme fragment est utilisé pour décrire les données avant qu'elles soient rendues comme des pixels à l'écran. Dans les anciens matériels graphiques, la géométrie envoyée au GPU est statique, et les opérations qu'on peut effectuer sur les

fragments sont limitées. Les fonctions de GPU sont restreintes à certaines étapes de pipeline comme le texturage et l'illumination. Dans les nouvelles générations des cartes graphiques, les unités de vertex et de texture sont maintenant programmables par l'utilisateur. Vertex shaders permet un contrôle sur les transformations de sommets sur le GPU elle-même, les fragments shaders sont introduits pour fournir un contrôle sur la façon avec la quelle les textures sont appliquées aux fragments. Une autre amélioration dans les nouvelles GPU concerne l'augmentation de la taille du pixel de 32 bit par pixel à 128 bit par pixel, ce qui veut dire que chaque composants : rouge, vert, bleu et alpha peut avoir 32 bit de la précision de virgule flottante à travers le pipeline graphique. L'augmentation de la précision de données combinées avec l'augmentation de programmabilité du GPUs courantes implique qu'un algorithme de rendu 3D peut maintenant être implémenté directement sur GPUs avec une efficacité plus grande et des effets de visualisation réaliste **[QZ02]**.

4.2 Exemple de programmation GPU : Phong shading

L'un des aspects les plus importants pour produire des rendus réalistes est le modèle d'éclairage utilisé. Un des modèles les plus populaires est le modèle d'illumination de Phong. Quand l'équation de Phong est évalué et les valeurs du pixel intermédiaires interpolées à partir des sommets, la technique est connue sous le nom ombrage de Gouraud, l'évaluation de l'équation pour chaque fragment est connu par l'ombrage de Phong, les deux utilisent le modèle d'illumination de Phong.

L'équation d'illumination de Phong est relativement simple et son implémentation est facile. Bien que l'explication détaillée de l'ombrage de Phong ne soit pas le sujet de cette partie, une brève description est donnée. L'équation est une somme de quatre composantes de la lumière:

$$\mathrm{I} = Ke + \mathrm{Ka} + Kd(\bar{N} \,.\, \bar{L}) + \ K_s(\bar{R}.\bar{V})^n$$

Tel que *Ke* est la composante émissive, décrivant la lumière produite par l'objet ; *Ka* est la lumière ambiante, qui est indépendante de la direction. *Kd* est la composante diffuse; *N* est le vecteur normal; *L* est le vecteur de la lumière; *Ks* la composante spéculaire; *R* le vecteur du rayon reflectant, *V* le vecteur de vue et *n* l'exposant de la réflexion spéculaire **[SA98]**.

L'exemple suivant est « Phong shading » implémenté en Cg:

```
// la position p et la  normale N
float3 P = position.xyz;
float3 N = normal;
// La composante emissive
float3 emissive = Ke;
// Calcul de la composante ambiante
float3 ambient = Ka * globalAmbient;
// Calcul de la composante diffuse
float3 L = normalize(lightPosition – P);
float diffuseLight = max(dot(N, L), 0);
float3 diffuse = Kd * lightColor * diffuseLight;
// Calcul de la composante spéculaire
// H est utilisé comme approximation à R
float3 V = normalize(eyePosition – P);
float3 H = normalize(L + V);
float specularLight = pow(max(dot(N, H), 0), shininess);
// si la composante diffuse est 0,  la composante spéculaire est 0 aussi
if (diffuseLight <= 0) specularLight = 0;
float3 specular = Ks * lightColor * specularLight;
// Calculer la somme des composants pour former la couleur finael du pixel
color.xyz = emissive + ambient + diffuse + specular;
```

le résultat du programme ci-dessus est illustré par la figure 2.2 suivante:

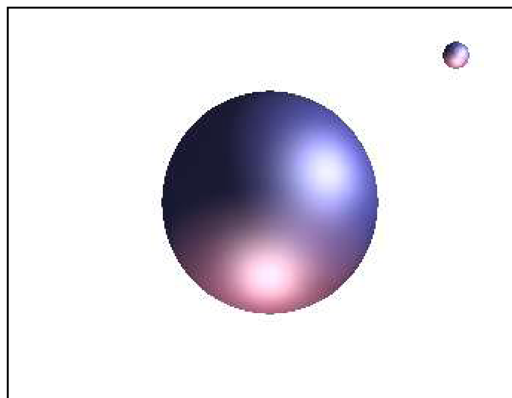


Figure 2.2*: Deux sphères rendues avec l'ombrage de Phong. Il existe deux sources de lumière dans la scène, un bleu et un rouge, donnant les points saillants spéculaire highlight*

Bien que le code fragment shader soit relativement court, il implémente un modèle d'éclairage complet et réaliste. En fait, les fragments shaders sont habituellement courts, leur puissance vient de la capacité du GPU pour les exécuter très rapidement. Des algorithmes simples, mais coûteux en calculs, qui précédemment étaient trop lents pour être considéré pour le rendu interactif mais peut être faisable quand ils sont implémentés sur le GPU **[KE05]**.

5. Les défis traités par les GPUs :

Pour répondre au besoin de flexibilité dans le traitement des diverses opérations du pipeline, et en particulier celles de transformations de sommets et d'opérations par fragments (texturage, éclairage etc.), le matériel doit être traité au défis suivants:

5.1 Gestion de gros volumes de données :

Les scènes que nous devront traiter sont constituées d'une quantité très importante de données volumiques arrivant peut être à plusieurs milliards de voxels. De tels volumes de données posent déjà des problèmes de gestion mémoire dans le cadre du rendu haute qualité, ces problèmes deviennent encore plus gênant pour le rendu temps réel. Des structures de données permettant un stockage compact ainsi qu'un accès rapide et efficace aux données doivent être mises en place (pour plus d'information voir chapitre 3) **[KE06]**.

5.2 Mémoires embarquées :

Des stratégies de gestion de la mémoire doivent également être adoptées afin de permettre une manipulation simple de ces données qui ne peuvent être tenu entièrement en mémoire. Dans le cadre du rendu temps réel, les zones mémoires employées (sur le matériel graphique en particulier) sont très proches des unités de calcul et spécifiquement conçues pour un accès rapide et performant. La contrepartie de cette efficacité est qu'elles ont une capacité très limitée et qu'elles sont encore plus éloignées des mémoires de masses dans lesquelles sont stockées ou générées les données. De ce fait, leur approvisionnement en données est particulièrement lent. En plus d'être coûteuses à transférer, ces données sont également coûteuses à générer et cet aspect devra également être pris en compte.

En plus de ces capacités de calcul, les GPU disposent de leur propre mémoire embarquée (figure 2.3) qui permet de stocker des géométries, textures ou tampons d'images (utilisées comme cible de rendu) .

Les textures sont des zones mémoires dotées d'opérations spéciales comme des accès interpolés, des mécanismes de mip-mapping ou des caches permettant un accès accéléré aux données utilisées récemment par une ressource de calcul du GPU **[KE05]**.

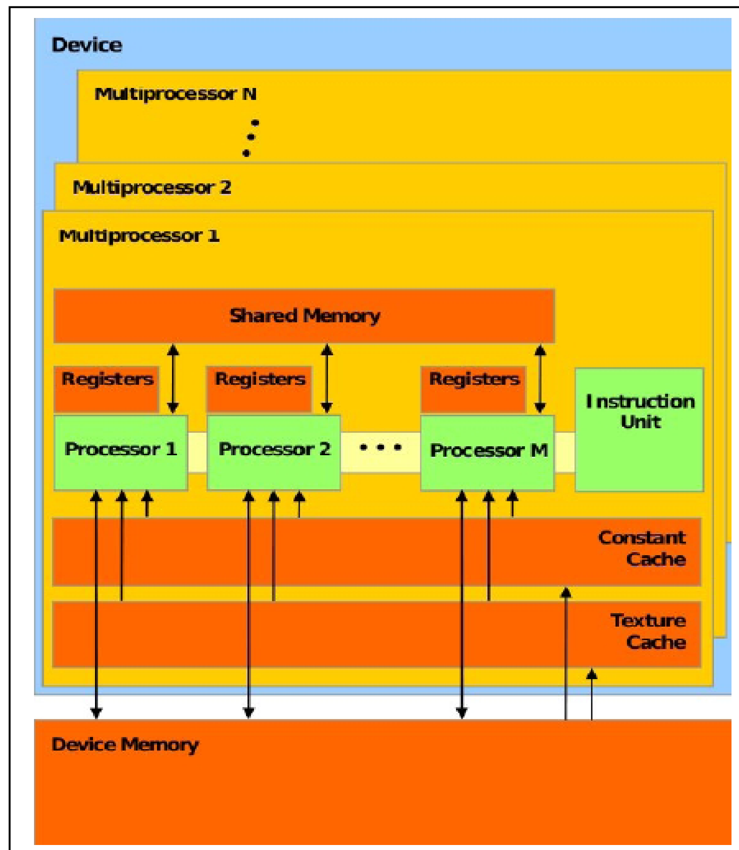


***Figure 2.3** : hiérarchie de la mémoire dans le GPU*

Ces caches sont optimisées pour accélérer les opérations de lecture spatialement proches, ce qui est généralement le cas lors du traitement de primitives graphiques **[CC07]**.

5.3 Interconnections avec les autres composants matériels :

Le GPU est relié au reste de la machine via un bus graphique spécial. Ce bus offre des débits théoriques très importants (d'environ 8Go/s) mais ne permet en pratique, (à cause des limitations du matériel) qu'un débit varie entre 250 et 500Mo/s pour le transfert de textures. Il s'agit d'un débit très faible par rapport à la vitesse d'accès du processeur à la mémoire centrale par exemple (de l'ordre de 6Go/s).

6. Matériel graphique pour le rendu interactif :

Le domaine de la synthèse d'image se compose d'une étape de modélisation et rendu haute qualité, avec un coût élevé en calcul. Pour un rendu interactif ou temps réel, demandé

dans des différents champs comme l'industrie des effets spéciaux, jeux vidéo et outils de visualisation. Pour cet usage, tous les compromis doivent être faits en termes de qualité pour fournir une fréquence d'affichage de 10 à 60 images par seconde (FPS ou Frames Per Second).

Le calcul du rendu des scènes de synthèse à l'usage des applications interactives est accéléré par du matériel spécialisé, il s'agit de la carte graphique et de ses circuits dédiés au rendu.

***Figure 2.4**: GPU NVidia GeForce 8800 GTS*

6.1 Description du pipeline graphique :

Les GPU est conçue de traiter principalement des objets géométriques et des pixels. Les images sont créées en réalisant des transformations géométriques sur les sommets et en divisant les objets en fragments ou pixels. Les calculs sont effectués par différents étapes, c'est ce que l'on appelle le pipeline graphique, comme présenté sur la figure 2.5. La machine hôte envoie des sommets afin de positionner dans l'espace des objets géométriques (polygones, lignes, points) **[FD09]**.

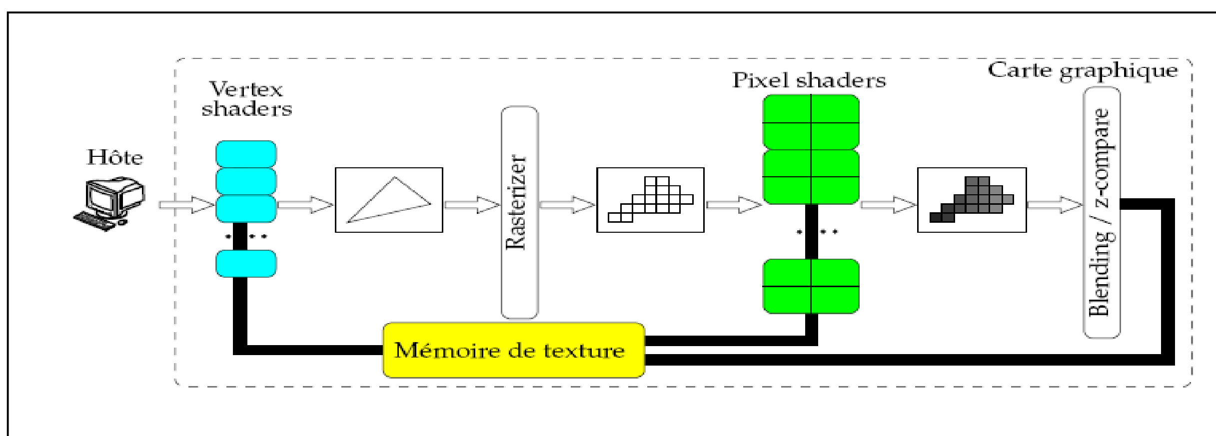


***Figure 2.5**:Vue générale du pipeline graphique*

Ces objets primitifs subissent à leurs tours, des transformations (rotations, translation, illumination. . .) avant d'être assemblés pour créer finalement, un objet plus complexe. Ces opérations sont appliquées au niveau de l'unité de traitement des sommets (vertex shader).

Après avoir accompli tout les attributs de l'objet qui sont: sa position, sa forme et son éclairage final, l'objet est découpé en fragments ou pixels. Une interpolation est effectuée pour obtenir les propriétés de chaque pixel. Ensuite, un traitement effectué sur les pixels par l'unité de pixel shader qui effectue les tâches d'affichage comme par exemple appliquer une texture ou calculer la couleur d'un pixel **[MD06]**.

a. Description :

Le pipeline de rendu graphique représente l'enchaînement de l'ensemble des opérations nécessaires au passage d'une scène définie spatialement à l'aide de primitives graphiques à une image plane visualisable à l'écran.

Le rôle des processeurs graphiques tels que nous les connaissons aujourd'hui est d'effectuer le traitement de l'ensemble des étapes qui forment ce pipeline de rendu graphique temps réel. Le modèle de rendu utilisé est basé sur l'algorithme du Z-Buffer.

C'est grâce à ce pipeline que les bibliothèques graphiques OpenGL et Direct 3D permettent de contrôler, son schéma de principe général correspondant au dernier modèle exposé par ces API est présenté dans la figure 2.6

b. Les étapes du pipeline graphique :

Pour bien comprendre l'importance derrière l'utilisation du matériel graphique, il est important d'en connaître l'usage et le mode de fonctionnement particulier. Dans cette section, nous présenterons tout d'abord le pipeline graphique implémenté par ce matériel :

Entrées : On trouve en entrée du pipeline l'ensemble des sommets composant les objets à visualiser, ces sommets sont regroupés en primitives (triangles, quadrilatères, liste de triangles adjacents etc.).

Opérations par sommets : ce sont des transformations effectuées sur les sommets afin d'effectuer l'ensemble des changements de repère nécessaires à leur position spatial. Ils sont également projetés en espace image via une projection perspective ou parallèle orthographique.

Assemblage : Après l'étape de transformation et projection, les sommets sont regroupés et assemblées en primitives reconnues par le système, il s'agit généralement de triangles.

Rastérisation : L'étape de rastérisation s'occupe à discrétiser les triangles projetés en créant une série de pixels, appelés fragments, couvrant leur surface. Ces fragments sont ensuite traités par l'étape d'opérations par fragments en charge de leur appliquer une couleur ainsi qu'un modèle d'éclairage.

Opérations en espace image : L'écriture des fragments dans le frame-buffer (tampon image) se fait après une étape finale comportant un certain nombre d'opérations sur les fragments (Frame-buffer Operations). Il s'agit des tests d'opacité (Alpha test), de stencil (Stencil test) et de profondeur (Depth test), ainsi que du mélange en fonction de l'opacité (Alpha blending) qui permet entre autre de simuler la transparence **[CC07]**.

Les shaders programmables : Les shaders sont des étapes de traitement du pipeline totalement programmables, elles apparaissent en pointillé dans la figure 2.6. Ces étapes sont conçues pour l'application de petits programmes appelés shaders, de manière identique et indépendante sur l'ensemble des primitives à traiter (les sommets pour le Vertex Shader, les fragments pour le Fragment Shader et les primitives pour le Geometry Shader).

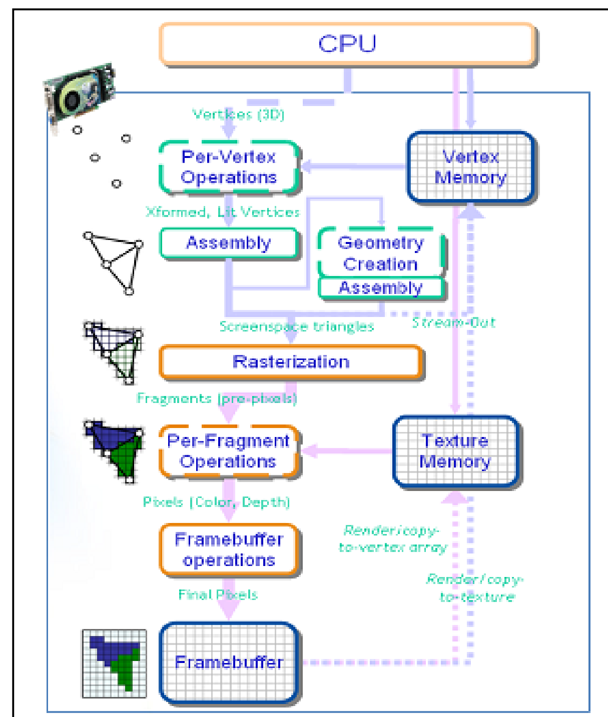


Figure 2.6*: Schéma de principe du pipeline de rendu temps réel.*

7. Les Shaders :

Les *shaders* sont des programmes informatiques conçus pour être implémentés sur les cartes graphiques. On distingue les *vertex shaders*, qui sont exécutés pour chaque sommet d'un maillage 3D à afficher, et les *pixesl shaders*, ou *fragment shaders*, qui sont exécutés pour chaque pixel affiché.

Ce sont les deux parties principales du pipeline graphique spécialisés pour le traitement des sommets (vertex), et le traitement des pixels. Un vertex contient les informations de position, de couleur, et de normale. Il suit des transformations dans le pipeline graphique afin d'être finalement projeté dans le repère de l'écran. Puis les sommets sont assemblés en primitives (triangle, carré ...) qui sont par la suite texturées et colorées. Les pixels se trouvant à l'intérieur de ces primitives sont appelés "fragments". Ces opérations sont appelées opération de shading (ombrage).

Les premières et deuxièmes générations des cartes graphiques ont un pipeline graphique complètement automatisé. Les opérations de texturage et de colorisation se font par interpolation.

Avec l'arrivée des cartes de troisième et quatrième génération (Geforce 4, Geforce FX), il est devenu possible de programmer ces deux parties du pipeline graphique.

Le faite de pouvoir implémenter ses propres opérations d'ombrage directement pour le GPU offre une flexibilité en plus, et permet de profiter de l'architecture entièrement optimisée pour les traitements 3D. Les gains de calculs sont considérables, comparés à une implémentation des mêmes techniques d'ombrage pour un CPU **[AP05]**.

7.1 Les générations de shaders :

Ces dernières années, le domaine du matériel graphique a connu des pas géants en effectuant une série de révolutions successives.

L'évolution des GPU a été marquée par plusieurs étapes d'évolution des capacités des shaders programmables.

- Les GPU de premières générations (c'est le moment où les cartes 3D sont devenues programmables et ont été appelées GPU) implémentaient le premier modèle de shaders appelé shaders model 1.0. Cette première génération ne fournissait qu'une étape de vertex shader réellement programmables, l'ancêtre des fragments shaders alors appelé 'register combiner ' était simplement dédié à la combinaison des couleurs lors de l'emploi de plusieurs textures.
- Les shaders model 2.0 ont été les premiers à fournir la programmabilité sur les deux étapes, cette programmabilité était par contre limitée sur de nombreux points (types de textures, manipulation de données flottantes, pas de structures conditionnelles dynamiques etc…).

- Les shaders model 3.0 ont ensuite apporté, entre autre, la possibilité d'utiliser des structures conditionnelles, ainsi que des boucles dynamiques. La dernière génération de shaders introduite lors de la sortie du G80 unifie totalement les possibilités des différentes étapes de shaders et introduit un grand nombre de possibilités nouvelles comme l'indexation dynamique de données ou la manipulation de nombre entiers.

8. Les architectures de dernière génération :

Le G80 (année 2007) **[CC07],** dernière génération de GPU introduite par le constructeur NVidia effectue une nouvelle étape dans cette évolution. Ce GPU est en effet doté d'une architecture dite unifiée, totalement centrée autour des unités de calcul programmables (shaders) et rompant avec les architectures calquées sur le pipeline graphique utilisée jusqu'alors. Ces unités de calcul sont en effet devenues totalement génériques et utilisées indifféremment pour mettre en oeuvre les différentes étapes programmables du modèle de pipeline graphique exposé par les API (OpenGL et Direct3D). **[CC07]**
Une vue globale de cette architecture est présentée dans la figure 2.7.

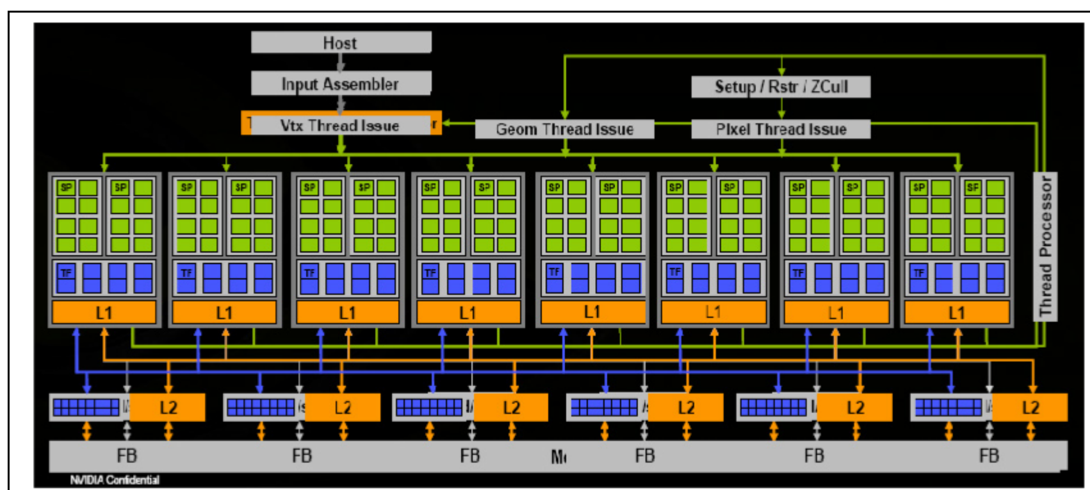


***Figure 2.7** : Diagramme de bloc de l'architecture du G80, dernière génération de GPU du constructeur NVidia constitué de 16 multi-processeurs formés chacun de 8 unités de calcul et regroupés par 2 dans 8 clusters de texture.*

8.1 Architecture matérielle

Pendant longtemps, l'architecture interne du matériel graphique grand public à évolué en intégrant progressivement, sous la forme d'unités de calcul dédiées, l'ensemble des étapes du pipeline graphique. Ces architectures proposaient ainsi un pipeline matériel très figé et totalement dédié à l'accélération du modèle de rendu temps réel classique. Pour répondre au besoin de flexibilité dans le traitement des diverses opérations du pipeline, et en particulier

celles de transformations de sommets et d'opérations par fragments (texturage, éclairage etc.), le matériel à évoluer en intégrant des unités de traitement programmables pour ces opérations. Aux fonctionnalités tout d'abord limitées et dédiées à leur rôle dans le pipeline, ces unités ont pris une place de plus en plus d'importance au sein du matériel et se sont vues dotées de capacités de calcul de plus en plus importantes et génériques.

Les multi-processeurs

Les unités de calcul appelées Stream Processors (SP) (voir figure 2.8) sont regroupées par grappes (appelées Multi- Processeurs) de 8 unités. Chaque multi-processeur fonctionne en SIMD (Single Instruction Multiple Data) en exécutant la même instruction en parallèle mais sur des données différentes à chaque cycle d'horloge. Les "processus" de calcul appelés threads sont ainsi exécutés en SIMD sur ces multi-processeurs par groupes appelés warps. Sur le G80, la taille d'un warp peut être de 16 ou 32 threads, ce qui signifie que chaque multiprocesseur est optimisé pour exécuter la même instruction pendant 2 ou 4 cycles d'horloge.

La taille d'un warp correspond donc à la granularité d'efficacité des branchements dynamiques. **[KE05]**

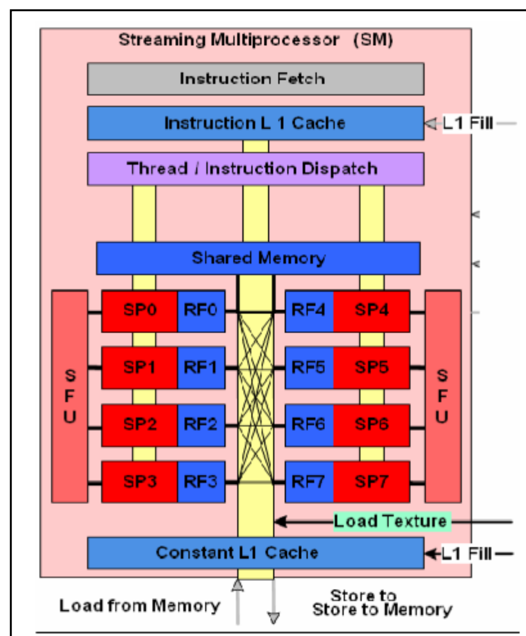


Figure 2.8 *: Schéma de principe d'un multiprocesseur du G80.*

Si un minimum de 16 ou 32 threads batchés en warp ne suit pas le même chemin pour un branchement dynamique, les threads divergents sont "masqués" par le multi-processeur ce qui

signifie que les SP correspondant ne sont pas utilisés et que l'architecture n'est pas utilisée au maximum de ses possibilités.

Les stream-processeurs

Les Stream Processors sont dédiés aux opérations d'addition et de multiplication flottantes ou entières opérant sur des scalaires 32 bits. Les multi-processeurs sont en plus dotés de deux SFU (Super Function Unit) correspond aux fonctions spéciales (inverse, racine, fonctions trigonométriques etc.) dont l'exécution prend 4 cycles (4 fois moins d'unités de ce type que de SP étant présentes, un warp prend 4 fois plus de cycles à exécuter une instruction).

Actuellement des algorithmes performants de *Ray-Casting* sur GPU existent et sont simples à implémenter **[VV07]**.

9. Exemple d'étude: Lancer de rayon sur GPUs :

Pour mieux comprendre l'utilité du GPU, nous présentons ci-dessous la mise en œuvre d'un lancer de rayon sur un GPU. Nous commençons par présenter les approches précédentes d'utilisation des GPUs pour accélérer le lancer de rayons.

9.1 Travaux antérieurs :

Il existe deux types d'approches d'utilisation des GPUs pour accélérer le lancer de rayons. La première de Carr et al **[CA02]** utilise le GPU pour le test intersection rayon /triangle sans aucune génération des rayons ou opération d'ombrage, cela veut dire qu'une grande partie de rendu reste à la charge du CPU. Carr et al arrivent à un test de 120 millions d'intersections rayon/ triangle par seconde sur ATI Radeon 8500. Il est à noter que la précision de la virgule flottante limitée (24 bits) de cette GPU laisse place à l'apparition d'artefacts.

La seconde approche a été faite par Purcell **[PU04]** qui a implémenté un lancer de rayon complet sur GPU. L'intégralité des étapes de la génération des rayons à travers une structure d'accélération jusqu'à l'étape d'ombrage sont réalisées sur GPU.

9.2 Mappage du GPU (A GPU mapping) :

Il est claire d'après le paragraphe ci-dessus qu'il y'a plus qu'une manière pour implémenter un lancer de rayon classique sur GPU. Dans cette section nous allons détailler l'implémentation utilisée par Purcell. Une vue générale de l'approche est montrée dans la figure 2.9.

Purcell **[PU04]** a divisé le noyau intersection/ traversal en deux noyaux séparés, une d'intersection et l'autre de traverse.

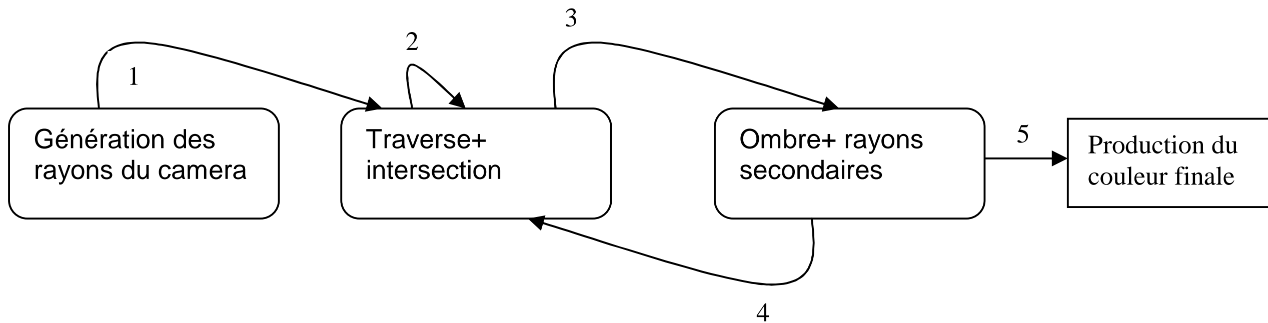


***Figure 2.9**: division du lancer de rayon en noyaux*

9.3 Génération des rayons :

Avant de pouvoir lancer des rayons, nous devons générer des rayons primaires démarrant de l'œil jusqu'à la scène. Avec la GPU, on peut profiter des capacités du moteur de rastérisation (the rasterizer) afin de générer tous les rayons primaires dans un noyau d'invocation unique.
En donnant les 04 coins d'un rectangle et le point de l'œil, on calcule d'abord 04 rayons pour les coins. Si on permet au moteur de rastérisation d'interpoler la direction de ces 04 rayons à travers une région 512x512 pixel, cela implique que toutes les directions des rayons primaires doivent être génèré une image de 512x512 pixel. On peut stocker ces directions dans une texture et puis les utiliser comme une entrée pour le noyau d'intersection. Tout les rayons primaire ont la même origine, mais on les stock toujours dans une texture de la même dimension que la texture de direction. Cela est nécessaire car l'origine des rayons change après l'opération d'ombrage ou la réflexion des rayons **[NL05]**.

9.4 Test d'intersection :

Le noyau de traverse/intersection prend comme entrée les origines des rayons et leurs directions.
On produit pour chaque pixel un enregistrement pour le point d'intersection.

- Si un rayon provoque une intersection, on reporte les 03 coordonnées barycentrique de l'intersection, relative du triangle intersecté.
- En outre, on va montrer à quelle distance le long du rayon, l'intersection sera effectuée et une référence à la matière du triangle.
- En effet, on a besoin deux coordonnées des trois coordonnées barycentriques tant qu'ils sont toujours égale à 1 avec le triangle.
- De cette manière, il est possible de stocker les enregistrements d'intersection dans une texture RGBA unique de la même dimension que les deux textures de rayons.

Moller et Trumbore **[MO97]** ont proposé un algorithme d'intersection rayon/ triangle comme suit :

```
/**Used for triangle intersection testing.
le triangle est définit par les sommets a, b et c
@param o est l'origine du rayon
@param d est la direction du rayon
@param lasthit est l'enregistrement de la précédente d'intersection
where .x= u, .y= v, .z= t, .w= material index for best hit
@return une nouvelle enregistrement d'intersection avec le meme format
*/
float4 Intersects(float3 a, float3 b, float3 c,
float3 o, float3 d, float minT,
float mat_index, float4 lasthit)
{
float3 e1 = b - a;
float3 e2 = c - a;
float3 p = cross(d, e2);
float det = dot(p, e1);
bool isHit = det > 0.00001f;
float invdet = 1.0f / det;
float3 tvec = o - a;
float u = dot(p, tvec) * invdet;
float3 q = cross(tvec, e1);
float v = dot(q, d) * invdet;
float t = dot(q, e2) * invdet;
isHit = (u >= 0.0f) && (v >= 0.0f)
&& (u + v <= 1.0f)
&& (t < lasthit.z)
&& (t > minT);
return isHit ? float4(u, v, t, mat_index) : lasthit;
}
```

9.5 Ombrage (Shading) :

Apres avoir calculé les états d'intersection de tous les rayons primaires, on peut traiter les normales du surface, et les propriétés de la matière pour l'opération d'ombrage. Chaque enregistrement d'intersection stocke un indice de la matière de la texture qui contient les normales du triangle et la matière du couleur. Les trois sommets du normales sont interpolés vis-à-vis les coordonnées barycentrique. La couleur finale du pixel peut être maintenant calculée par : CF= (N · L) C, tel que CF est la couleur finale, N est le normal, L est la direction à la lumière et C est la couleur du triangle.

Vis à vis du type de la matière, on est besoin de générer des rayons secondaires pour estimer les ombres et les réflexions **[RS00]**.

- si le rayon quitte la scène, le pixel prend la couleur de l'arrière plan.
- si le rayon frappe une matière diffuse, on génère un rayon d'ombre. L'origine de ces rayons est le point d'intersection.
- si le rayon frappe une matière spéculaire, on génère un rayon de réflexion. L'origine des rayons de réflexion aussi est le point d'intersection mais la direction est une réflexion parfaite des rayons par rapport au normal interpolé.
- Si le rayon d'ombre ne frappe rien, cela veut dire que la géométrie trouvée entre l'origine du rayon d'ombre et la source de la lumière et le pixel en question doit être noire. Quand on lance un rayon d'ombre, on ne s'intéresse pas de l'intersection la plus proche mais n'importe quelle intersection. Pour cela, il est possible d'accélérer le lancer de rayons d'ombre par le faite d'arrêter la traverse dès qu'une frappe est trouvée **[NL05]**.

Après la génération des rayons secondaire pour chaque pixel, il est possible d'effectuer une autre passe du traverse/ intersection s'il est nécessaire. Chaque entrée au noyau d'ombre retourne la couleur et un nouveau rayon pour chaque pixel.
Le noyau d'ombre prend comme entrée le buffer de la couleur résultante du passe d'ombre précédente. Cela permet de combiner les couleurs des surfaces spéculaires successives quand on lance des rayons successifs.

10. Conclusion :

Dans cette partie, nous avons présenté le matériel graphique (GPUs) dédié pour la visualisation, en se focalisant sur la nécessité d'introduire ce matériel afin de toucher de taux d'affichage proche du temps réel. Nous avons également montré l'évolution des cartes graphiques avec une présentation détaillée du pipeline.

Cependant, il est important de noter que l'utilisation de GPUs pour effectuer les calculs d'intersection n'est pas une fin en soit et qu'elle ne doit être introduite que comme un outil parmi divers autres. A son tour, cet aspect matériel représente un outil qu'il conviendra d'associer à d'autres techniques d'accélération logicielles, en vue d'obtenir des applications beaucoup plus rapides que celles disponibles à ce jour.

Pour cela, nous allons présenter dans le chapitre suivant une classification des structures de données permettant l'accélération du processus de rendu volumique.

Chapitre 03 :

Les structures de données accélératrices

Chapitre 3: Les structures de données accélératrices

1. Introduction :

Dans ce chapitre, nous allons décrire quelques structures de données utilisées pour l'accélération du rendu volumique. Les structures accélératrices servent à réduire le nombre de tests d'intersection rayon/triangles et donc accélérer le processus de rendu.

Au début, nous détaillons les schémas de subdivision spatiale, ensuite nous projetons la lumière sur les grilles régulières ainsi que les KD-tree. Etant donné que notre principal objectif consiste à trouver des structures d'accélération pour un rendu sur GPU, nous allons introduire une discussion concernant l'application du GPU pour chaque structure.

2. Les structures accélératrices:

La principale importance de ces méthodes est de réduire le nombre de polygones à traiter afin de diminuer le temps de calcul. Une autre approche est donc en général utilisée pour diminuer le nombre d'objets à manipuler : réduire le nombre de tests d'intersection dans un lancer de rayons, ou le nombre de projections dans un tampon de profondeur **[DF04]**.

Nous présentons ici des structures accélératrices fréquemment utilisées en rendu volumique.

2.1 Volumes englobants :

Afin d'accélérer les calculs d'intersection entre un rayon et la scène, une méthode consiste à regrouper les faces entre elles et à créer des volumes autour de ces groupes. L'intérêt est toujours de réduire le nombre de polygones testés. Les intersections sont alors calculées avec les groupes (moins nombreux) puis avec les faces du groupe au premier plan.

La construction de ces volumes se réalise à partir d'un ensemble de paires de plans parallèles. La figure 3.1 montre à droite un exemple de volume englobant créé à partir de paires de plans

parallèles. Lorsque le volume englobant de chaque objet a été créé, une hiérarchie est construite à l'aide d'une technique de découpage médian (*median cut*). Chaque liste de faces est coupée en deux sous-listes, elles-mêmes subdivisées ensuite de manière récursive. Un arbre binaire est en même temps construit et décrit la scène de manière hiérarchique. Cette technique permet d'accélérer les tests d'intersections entre un rayon et les groupes de la scène. Si un rayon ne rencontre pas le volume englobant d'un nœud de l'arbre, alors il ne rencontre pas non plus l'ensemble de ses fils. Ainsi, seul un petit nombre des faces de la scène est pris en compte dans le calcul d'intersection **[QZ02]**.

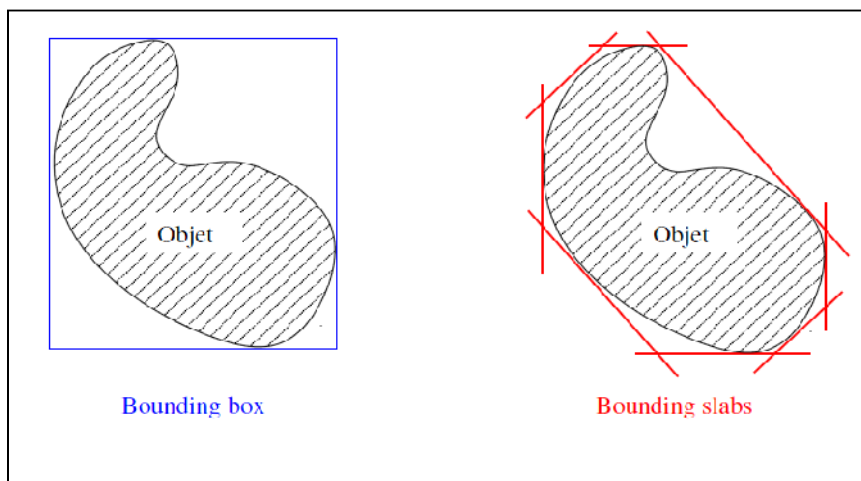


Figure 3.1*: À gauche, un objet délimité par sa boîte englobante. - À droite, le même objet est enfermé dans des bounding slabs : technique de découpage de scène consistant à définir des paires de plans parallèles autour de l'objet pour obtenir des limites plus précises qu'avec les boîtes englobantes.*

2.2. Subdivisions spatiales :

Ces techniques de subdivision spatiale consistent à appliquer un schéma de découpage pouvant s'adapter à la scène. Elles sont fréquemment utilisées pour accélérer les algorithmes à base de lancer de rayons. L'idée de base est de ne pas tester pour un rayon donné l'ensemble des faces lors de la recherche de l'intersection la plus proche. Pour cela, la scène est subdivisée en sous-parties et seuls les polygones appartenant aux parties traversées par le rayon sont testés. Le nombre de polygones contenus dans une partie étant en général très petit par rapport au nombre total, et donc l'accélération du calcul peut être très utile. Nous présentons ici 3 techniques de subdivision : la grille régulière, la multi-grille, et l'arbre octal. Voir la figure 3.2.

2.2.1 La grille régulière (brique régulière):

La grille régulière est peut être la plus simple de toutes les structures accélératrices, elle se nomme aussi brique régulière dans certaines littératures **[CC07] [DR08]**. Les méthodes basées sur les grilles régulières consistent à subdiviser la scène en des grilles 3D uniforme de voxels (figure 3.3). Chaque voxel possède une liste des triangles **[MC05]**.

Les volumes sont tous identiques et subdivisent l'espace de la scène sans prendre en compte les objets eux-mêmes. Pour construire ces grilles, des algorithmes de discrétisation sont utilisés pour déterminer quels sont les voxels touchés par une face. Grace à cette rapidité de parcours, la grille régulière est considéré comme une structure très utilisée en lancer de rayons **[DF04]**.

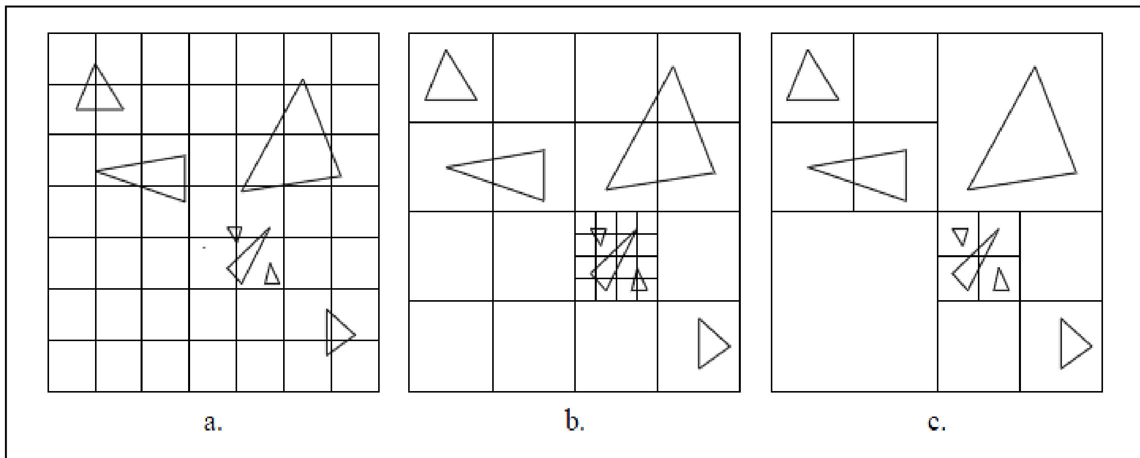


Figure 3.2 *: Trois différentes méthodes de découpage de l'espace : a. la grille régulière. - b. la multigrille. - c. le quadtree*

Pour calculer l'intersection d'un rayon avec la grille, on traverse les voxels qui sont intersectés par le rayon dans l'ordre d'intersection. Pour chaque voxel visité, on teste la liste des triangles associés pour une intersection avec le rayon et on indique le plus proche, le cas échéant.

Une fois que le voxel fait une intersection, on peut arrêter le parcours et on signale l'intersection comme étant le plus proche. On remarque qu'aucune autre intersection ne peut être plus proche lorsque nous visitons les voxels dans l'ordre croissant de la distance de l'origine du rayon. Nous avons également terminé le parcours lorsque nous atteignons l'extrémité de la grille **[MC05] [NL05].**

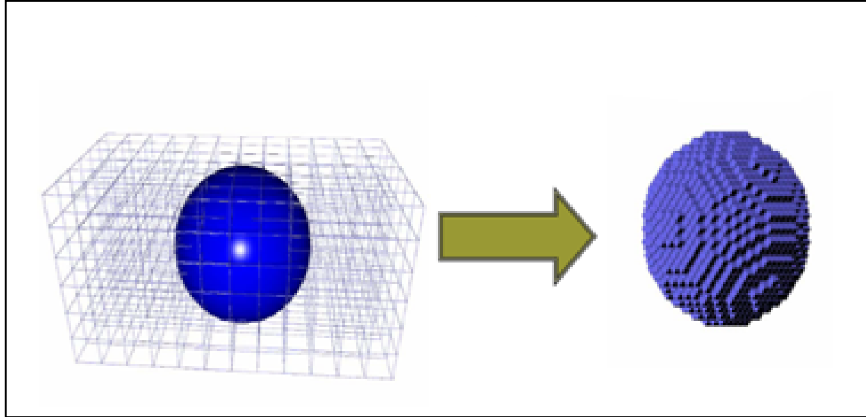
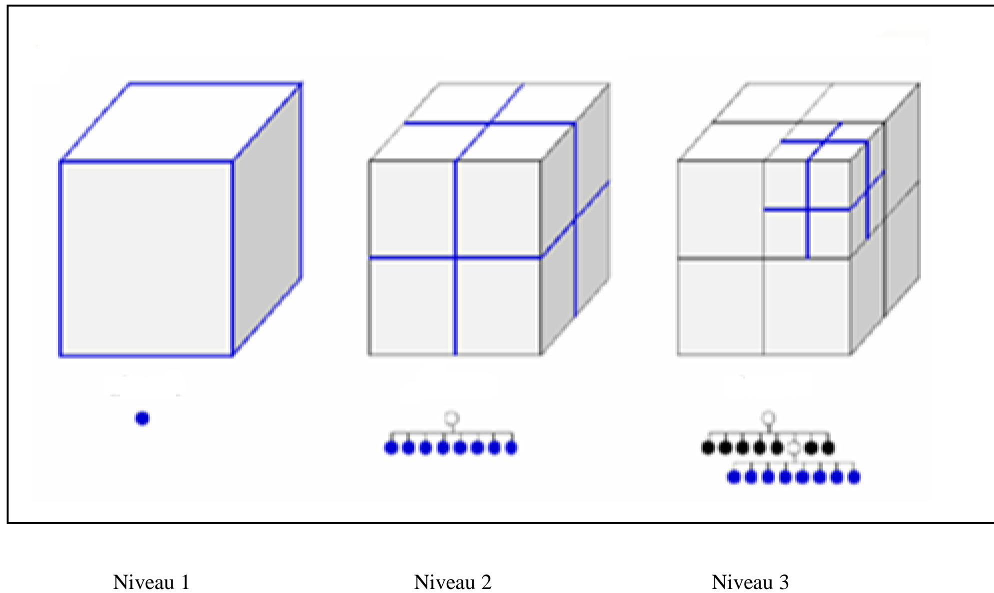


***Figure 3.3** : Illustration de la subdivision en grille régulière*

L'inconvénient majeur de la grille régulière est de ne pas tenir compte du déséquilibre de la scène en termes de disposition des faces dans l'espace. Le cas où la scène manque de régularité, de nombreux voxels parcourus peuvent être vides et certains peuvent contenir trop de faces. Afin de mieux s'adapter à la scène, il est plus utile de construire une *multigrille la* (figure 3.2 b), ou une hiérarchie de grilles uniformes. Si un voxel de la grille possède un trop grand nombre de faces par rapport aux autres, il est à nouveau subdivisé par une autre grille. Ceci permet, lorsque des scènes sont irrégulières, de conserver une rapidité de parcours des voxels comme pour une grille régulière **[MC05]**.

Pour l'octree, la subdivision de la scène se fait en régions cubiques de tailles variables. La construction de l'octree se fait de manière descendante. Au départ l'arbre ne possède qu'un nœud qui représente le cube englobant de la scène (figure 3.2). Puis de manière récursive, si un nœud de l'arbre possède beaucoup de faces, il est subdivisé en 8 régions égales. Nous obtenons alors une scène découpée en fonction de la répartition des polygones dans l'espace. La figure 3.2.c montre un exemple d'octree. Les scènes irrégulières sont donc beaucoup mieux représentées à l'aide de cette structure d'arbre (figure 3.4), mais le coût de parcours d'un octree est plus important que celui d'une grille.



Niveau 1 Niveau 2 Niveau 3

Figure 3.4 *: l'arbre octal*

2.2.1.1 Construction de la grille régulière:

Etant donné une boîte englobante de la scène et une liste de triangles, on peut construire une grille uniforme. La première chose à décider est la résolution de la grille le long des trois axes. Vu que la grille possède une structure fixe, c'est le seul paramètre que nous pouvons contrôler dans la phase de construction.

Une résolution élevée donnera lieu à plusieurs voxels qui sont de taille plus petite. Cela suppose aussi de moins triangles par voxel, et donc de moins tests d'intersection du triangle par voxel. D'autre part, plus de voxels sert à perdre plus de temps à traverser la grille. Par conséquent, le meilleur choix de résolution pour une scène donnée n'est pas évident. **[NL05]**

Quelques approches utilisent un paramètre $3\sqrt[3]{N}$ de voxels sur l'axe le plus long, où N est le nombre de triangles. Nous avons utilisé une méthode similaire pour le calcul de la résolution de la grille sur l'axe le plus court. Pour certaines scènes, il est encore nécessaire d'ajuster la résolution à la main pour obtenir des résultats optimaux. Une fois qu'une résolution a été fixée, on peut allouer un tableau 3D, des listes de triangles. Pour chaque triangle dans la scène, nous trouvons maintenant les voxels qu'il intersectent et on ajoute une référence au triangle dans chaque voxel **[NL05].**

2.2.1.2 L'applicabilité sur les GPUs :

La grille uniforme était la première structure d'accélération implémentée sur GPU, et cela pour plusieurs raisons:

1. D'après les études **[NL05] [MC05]**, aucune structure de données accélératrice ne semble être la plus efficace.
2. Les grilles uniformes sont particulièrement simples pour les implémentations matérielles. L'accès aux structures de données de la grille nécessite un temps constant; les structures de données hiérarchiques, en revanche, ont besoin de temps variable pour l'accès ce qui implique un pointeur de suivie. Une boucle *'for'* pour le parcours de la grille est également très simple et peut être hautement optimisé au niveau matériel.

L'algorithme suivant décrit le parcours de la grille régulière.

***Algorithm 1** Incremental part of the $DDA^{(1)}$traversal.*

Tant que** X et Y dans la grille **faire
tester les triangles pour des intersections
***Si** une intersection est trouvée dans ce voxel alors*
stopper le parcours et retourner intersection
fin si
Si** tmaxx < tmaxy **alors
X := X + stepx
Tmaxx := tmaxx + deltax
Sinon
Y := Y + stepy
Tmaxy := tmaxy + deltay
Fin si
Fin tant que
***retourner** non_ intersection*

Quand on examine certaines caractéristiques clés de la grille régulière, il devient plus clair la raison pour laquelle la grille régulière soit une bonne structure d'accélération GPU:

- Chaque voxel du parcours peut être localisée et accessible en un temps constant, en utilisant des opérations arithmétiques simples. Ceci élimine la nécessité du parcours d'un arbre et élimine donc un grand nombre de recherches répétées et couteuses dans la texture.

(1) :3D digital differential analyzer (DDA) : est un dispositif pour calculer directement la solution d'équations différentielles.

- Le parcours du voxel se fait progressivement en utilisant des opérations arithmétiques. Ceci élimine la nécessité d'une pile et il est possible de parcourir les voxels par ordre de distance croissante à partir du rayon d'origine.
- Nous pouvons exploiter l'ordre de visite pour arrêter le parcours dès qu'une intersection est signalée dans un voxel visité.

La limite principale des grilles uniformes réside dans le fait qu'elles sont un cas particulier d'une structure de subdivision spatiale et peuvent donc référencer le même triangle à partir de plusieurs voxels.

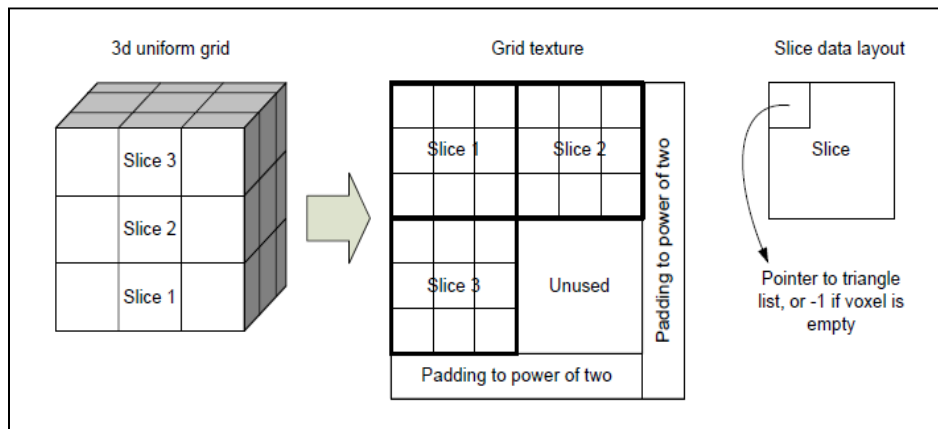


Figure 3.5*: Schéma de la mémoire de la Grille. La grille 3D des pointeurs de liste est mappé en texture 2D.*

2.2.1.3 Implémentation GPU :

Etant donné une grille uniforme déjà construite, on peut mapper les structures de données à un schéma de texture que nous pouvons utiliser comme entrée sur le GPU. La grille de voxel peut être stockée directement comme une texture 3D, où nous pouvons rassembler les tranches 2D de la grille en une seule texture 2D. L'organisation de la mémoire choisie est une collection de tranches 2D stockées dans une texture unique, voir la figure 3.5.

Il existe d'autres moyens de collecter les données dans une texture mais nous avons choisi celui-ci car le débogage sera plus facile.

Comme chaque voxel contient uniquement un seul pointeur vers une liste de triangles, nous avons uniquement besoin d'une texture avec un canal de couleur de 32 bits, ce qui économise de la mémoire et la bande passante.

Les listes du triangle sont stockées dans une texture en virgule flottante séparée avec un seul pointeur vers un triangle par élément de texture. Chaque liste est terminée par une valeur (-1) voir la figure 3.6. Dans une texture RGBA, il est possible de stocker quatre pointeurs par élément de texture. Cela permettrait de réduire le nombre de lectures de texture dans le fragment programme mais cela va aussi compliquer le code.

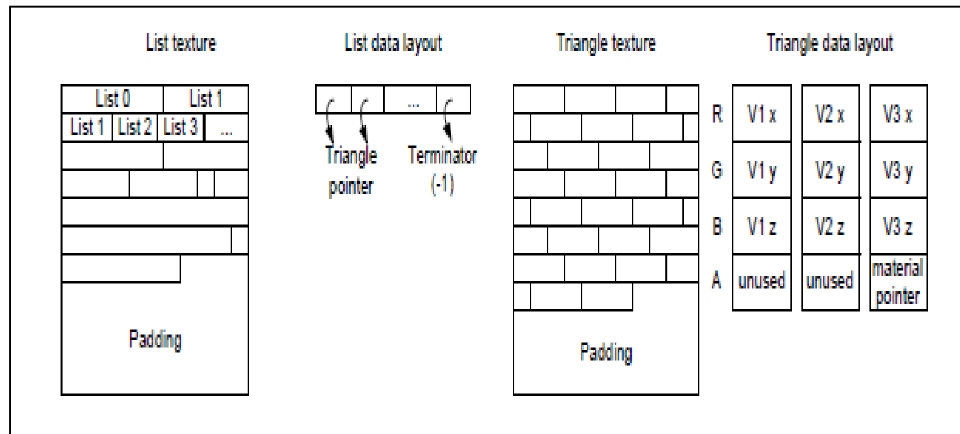


Figure 3.6: *Disposition de la mémoire pour les triangles et les listes de triangle* **[NL05]**.

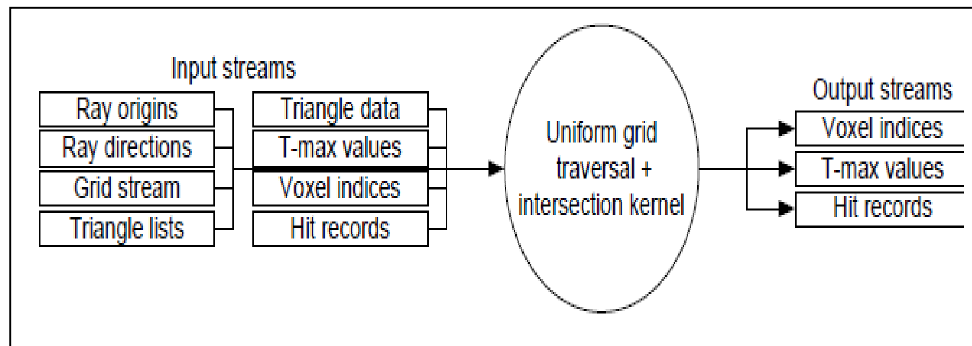


Figure 3.7 *: le parcours de la grille régulière*

Les triangles eux-mêmes sont stockés dans une autre texture avec chaque triangle occupant trois valeurs de vecteur RGBA en virgule flottante, un pour chaque sommet du triangle, voir la figure 3.6. Cela implique également que nous n'avons pas partagé de données des vertex entre triangles **[NL05]**.

Pour résumer: La grille uniforme est une structure simple et a été la première implémenté sur un GPU. La phase de construction divise la scène en une grille régulière de voxels qui référence les triangles qu'ils intersectent. La phase de parcours utilise le dessin de ligne 3D

comme base pour localiser les voxels qui sont touché par un rayon. Après avoir repéré le voxel correct, on peut donc tester chaque triangle dans ce voxel.

2.2.2 Le kd tree (kd arbre):

Comme la grille régulière, le KD-tree est une instance d'une structure de subdivision spatiale. Contrairement à la grille régulière, le KD-tree représente la scène comme une structure hiérarchique sous une forme d'arbre binaire **[CC07].**

Nous distinguons entre les nœuds internes et les feuilles dans les arbres. Les feuilles correspondent aux voxels et ont des références aux triangles qui intersectent les voxels respectifs. Un nœud interne correspond à une partition d'une région de l'espace. Par conséquent, les nœuds internes contiennent un plan de coupe et des références à deux sous-arbres. Les feuilles contiennent seulement une liste de triangles.

2.2.2.1 La construction : La construction d'un kd-tree se fait de haut en bas de façon récursive, elle est faite de la façon suivante :

- ✓ Etant donné une boite englobante et une liste de triangles qu'elle contient, nous choisissons un plan de coupe perpendiculaire à l'axe divisant la zone en deux.
- ✓ Chaque primitive de la boîte originale est affectée au nœud enfant qui le contient.
- ✓ Si une primitive se situe sur le plan de coupe, alors les deux volumes enfants obtenaient une référence.
- ✓ Cette procédure se poursuite jusqu'à ce que nous arrivions à une profondeur maximale choisie ou jusqu'à ce que le nombre de triangles dans un nœud atteint un seuil donné.
- ✓ Des approches proposent une profondeur maximale de 16 et un nombre minimal de triangles égal à 2 est proposé pour une performance optimale.
- ✓ Mais il semble qu'une profondeur égale à 16 n'est pas un bon choix pour une meilleure profondeur de toutes les scènes réalistes. Comme la scène est divisée en deux à chaque niveau de l'arbre, une profondeur maximale serait une fonction logarithmique du nombre de triangles de la scène.

La partie la plus difficile de la construction réside dans le choix de l'emplacement du plan de coupe, étant donné un ensemble de géométrie. Une fonction de coût est dérivées, elle est basée sur le fait que la probabilité d'un rayon intersectant un nœud enfant est proportionnelle au rapport de la surface du nœud enfant à la surface du parent **[NB09]**.

D'après la figure 3.6, la fonction de coût est évalué à des positions a, b,. . . , j. Ces positions correspondent aux extrémités des intervalles qui liaient les triangles individuels le long de

l'axe de Split. Notez également que la position de split indiqué est seulement notre estimation où la fonction de coût aurait le plus bas coût.

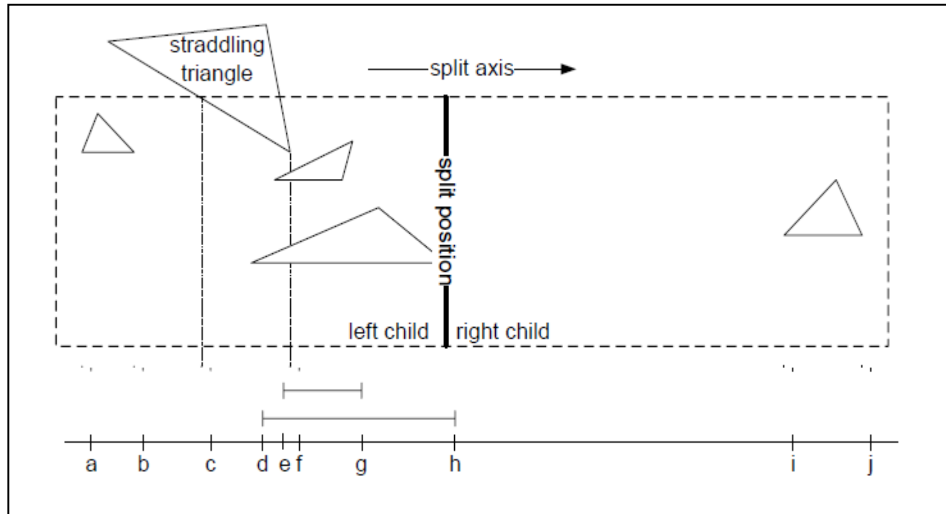


Figure 3.8*: évaluation de la fonction de coût*

Cette approche a ensuite été raffinée en dérivant une fonction de coût différente qui prend également en compte le fait qu'une primitive se situe sur un plan de coupe. Ceci est important car les triangles situant sur le plan sont propagées aux deux volumes enfants, et provoque ainsi un coût plus élevé de rendu.

La base du choix d'une position du plan de division est l'évaluation de la fonction de coût à tous les bords des boîtes englobantes des triangles voir la figure 3.8. Il se peut qu'un triangle ne tombe pas entièrement dans le voxel il faut alors le diviser. Dans ce cas, nous ne devrons pas utiliser la boîte englobante complète du triangle, mais plutôt une partie sur le triangle pour le voxel, puis générer un nouveau volume englobant. Ceci est également représenté dans (la figure 3.8) où le triangle le plus élevé n'est pas entièrement contenue dans le voxel.

Malgré le fait que nous découpons le triangle pour le voxel afin d'obtenir une nouvelle boîte englobante, le triangle actuel propagé aux enfants de ce nœud est le triangle d'origine. Après avoir employé cette technique, nous avons observé une amélioration des performances du parcours du kd-tree comme prévu **[NL05]**.

L'algorithme suivant décrit décrit la construction d'un Kd tree :

***Algorithme 2** : le code de construction KD-tree*.

```
KDtreeNode structure(voxel)
  Si num_triangles(voxel) ≤ Min_triangle alors
      return nouvelle feuille avec une liste de triangles
  fin Si
  Si  profondeur de l'arbre ≥Max profondeur alors
      nouvelle feuille avec une liste de triangles
 fin Si
      bon_ diviseur :=vide
      bon_cout :=
Pour chaque axe dans {x, y, z} faire
          positions  := liste vide
   pour chaque triangle dans voxel faire
          diviser le triangle correspondent au voxel
          calculer la boite englobante du triangle divisé
          trouver les points a et b de la boite englobante le long de l'axe
          ajouter a et b à la liste de positions
   Fin Pour
   Pour chaque point p dans positions faire
          Si coùt(p) < bon_cout alors
             bon_cout:= cout (p)
            bon_diviseur:=  (p, axis)
        Fin si
  Fin pour
Fin pour
(voxel gauche, voxel droit) := diviseur voxel correspondant à bon_diviseur
Pour chaque triangle t faire
         Si t intersecte voxel gauche alors
             ajouter t au voxel gauche
        Fin Si
        Si    t intersecte voxel droit alors
              ajouter t  au voxel droit
       Fin Si
Fin Pour
     Enfant gauche := structure (voxel gauche)
     Enfant droit := structure (voxel droit)
    Retourner nouveau nœud interne (Enfant gauche,  Enfant droit, bon_diviseur)
```

[NL05]

2.2.2.2 L'applicabilité sur les GPUs :

Le premier problème d'une implémentation de kd-tree sur un GPU est le fait que l'algorithme standard de parcours est récursif. L'absence de la structure de pile sur le GPU devient un problème gênant. Une solution serait d'afficher le kd-tree comme un cas particulier d'un volume englobant hiérarchique (BVH) (voir paragraphe 2.3), et puis utiliser son algorithme de

parcours. Cela signifie de traverser les enfants dans un ordre fixe mais empêche l'arrêt du parcours le plus rapidement possible quand une intersection est trouvé.

En outre, comme le kd-tree est habituellement beaucoup plus grand que la BVH correspondant pour la même scène, nous allons effectivement créés un BVH inefficace. La solution est donc d'employer une stratégie de parcours différente. Avant le parcours descente récursive a été inventé, le kd-tree et l'octree étaient traversés d'une manière séquentielle. La technique consiste à déplacer un point le long du rayon et descendre de la racine de l'arbre jusqu'à localiser la feuille contenant le point. Une autre technique similaire est apparue, sert à déplacer un intervalle $[\boldsymbol{t_{min}}, \boldsymbol{t_{max}}]$ au lieu d'un point le long du rayon et utiliser cet intervalle pour déterminer le voxel intersecté **[CA02]**.

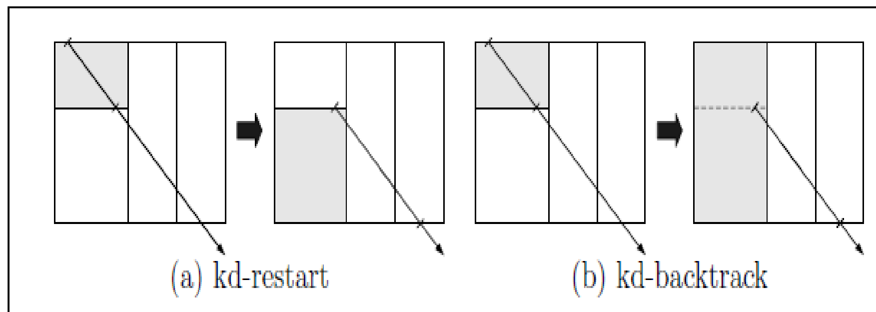


Figure 3. 9*: le test d'intersection*

Comme nous le remarquons dans la figure 3.9 : (a) Après l'échec du test d'intersection pour un voxel feuille le t-Intervalle doit débuter à la fin de la feuille. De cette façon, nous pouvons placer la feuille suivante lors du redémarrage du parcours. (b) L'intervalle est mis à jour de la même manière que pour un kd-restart, mais au lieu de redémarrer le parcours de la racine on traverse l'arbre jusqu'à ce que nous trouvons un ancêtre qui intersecte l'intervalle mis à jour (marqués en gris sur le côté droit).

La procédure de parcours d'un Kd tree se fait de la manière suivante :

- Initialement, l'intervalle traverse les valeurs t où le rayon est au niveau haut de la boite de la scène.
- Pour chaque niveau, on descend de l'arbre, $\boldsymbol{t_{max}}$ est mis comme le min $(\boldsymbol{t_{max}}, \boldsymbol{t_{split}})$ où $\boldsymbol{t_{split}}$ est la distance le long du rayon au plan de coupe au sein du nœud courant.
- Si la feuille n'a donné aucune intersection, nous mettons à jour l'intervalle. L'intervalle que nous utilisons pour le parcours continue commence à la fin du voxel en cours et se termine à la fin de la boîte de la scène voir la figure 3.9 (a).

- Deux variantes d'algorithme sont introduites, *restart* et *backtrack.*
- *L'algorithme restart* commence de la racine de l'arbre pour chaque étape de parcours du voxel. En visitant uniquement les voxels qui intersectent le t-intervalle, on peut garantir que les voxels sont visités dans l'ordre. Malheureusement, l'algorithme *restart* a une complexité en termes de temps plus grande qu'un algorithme basé sur la pile.
- Pour remédier à cela, l'algorithme de *backtrack* modifie l'algorithme de *restart* en permettant de progresser sur l'arbre au lieu de commencer de la racine à chaque fois. Quand une feuille voxel échoue son test d'intersection, on monte dans l'arbre jusqu'à ce que nous trouvons un ancêtre qui intersecte le nouveau t-intervalle **[EK05]**.

Le choix du bon nœud enfant à visiter lorsqu'on passe d'un parcours ascendant à un parcours descendant se fait de la même manière que pour l'algorithme *restart*. Cela provoque une complexité en temps similaire à celle d'un parcours en utilisant la pile.

Pour remédier cela, l'algorithme backtrack modifie restart en permettant de monter l'arbre au lieu d'être à la racine à chaque fois. Quand une feuille échoue dans ses tests d'intersection, on monte l'arbre jusqu'on trouve un ancêtre qui intersècte le t-intervalle.

L'utilisation d'un intervalle est une solution plus simple et plus élégant que la solution du Point de localisation **[MC05]**.

2.2.2.3 Implémentation sur GPU :

Le fait de mettre un kd-tree à une représentation texture est assez simple. Dans l'algorithme restart, chaque nœud occupe un seul vecteur RGBA en virgule flottante. Le parcours backtrack exige deux vecteurs RGBA en plus pour stocker une boîte englobant, cependant, il est identique en ce qui concerne l'organisation de la mémoire. Notez que cela signifie que la version *backtrack* utilise trois fois plus de mémoire que la version restart. Le stockage par nœud consiste à:

- **L'axe Split**: Lequel des trois axes est le plan perpendiculaire de découpe.
- **Position Split**: Où situe le plan de découpe.
- **Pointeur enfant**: Si c'est un nœud interne, nous avons besoin de savoir comment passer à ses deux enfants. En stockant toujours l'enfant gauche comme le nœud parcouru le premier, nous avons uniquement besoin de stocker un pointeur vers l'enfant droit.
- **Pointeur Parent**: Pour le *backtrack*, nous avons besoin de savoir comment parcourir de nouveau l'arbre.

- **Liste pointeur:** S'il s'agit d'un nœud feuille, alors on stocke l'indice à la liste des triangles référencés. Si la feuille est vide, nous stockons -1 pour indiquer que nous pouvons ignorer la recherche.
- **Type de nœud:** Indique s'il s'agit d'un nœud interne ou d'une feuille.

Il est possible de stocker les six valeurs dans un vecteur RGBA en observant qu'elles ne sont pas toutes utilisées à la fois. Par exemple, nous stockons l'axe Split, le type de nœud, et le pointeur liste dans un nombre à virgule flottante. Pour cela, nous utilisons ce mappage:

- Si la valeur est -1 alors nous avons une feuille vide **[EK05]**.

	internal node (restart)	internal node (backtrack)			leaf node
R	right child index	right child index	box min x	box max x	unused
G	split value	split value	box min y	box max y	unused
B	parent index	parent index	box min z	box max z	unused
A	split axis*	split axis*	unused	unused	list index

***Figure 3.10** : l'organisation de la mémoire pour le KD-tree*

- Si la valeur est soit -2, -3, ou -4, nous avons un nœud interne et l'axe split peut être déduit en mappant -2 à X, -3 à Y et -4 à Z.
- Si la valeur est 0 ou plus, nous avons une feuille avec une liste triangle non-vide.

La disposition exacte de la mémoire est affichée dans la figure 3.10.

En plus de l'arbre actuel, nous avons aussi besoin de représenter les listes de triangles qui pointent les nœuds feuille. Cela se fait de la même manière que les grilles uniformes:

Le nœud feuille pointe au premier élément de la liste et nous terminons la liste en stockant un élément avec une valeur -1. Comme c'était également le cas pour la grille uniforme, les éléments de la liste ne sont que des pointeurs vers les triangles et pas eux-mêmes triangles. Bien que c'est un compromis temps / espace pour la grille uniforme, nous avons constaté qu'il est plus qu'une nécessité pour le KD-tree, car le nombre de référence triangle peut être très élevé.

Pour résumer : Nous avons étudié les principes de base du kd-tree et expliqué les techniques de construction et d'implémentation. La construction démarre de haut en bas et divise récursivement la scène en deux voxels en positionnant un split selon une fonction de coût. Nous pouvons traverser le kD-tree d'une manière récursive, mais ce n'est pas possible sur les GPU, car nous ne disposons pas d'une pile. Cependant, on peut traverser l'arbre dans les deux sens en précisant quel point est déjà testé le long du rayon. Ceci élimine la nécessité d'une pile et rend possible le parcours GPU.

Lorsqu'on utilise un parcours GPU, le KD-tree est représenté comme une collection de textures de la même manière comme ce fut le cas dans la grille uniforme. Cela signifie que nous avons une seule texture pour l'arbre, une pour les listes de triangle et une pour les triangles actuels.

2.2.3 La hiérarchie des volumes englobants (the Bounding Volume Hierarchy ou BVH):

Le BVH est une structure de partitionnement hiérarchique de la scène. Elle diffère des techniques de la subdivision spatiales par le fait de partitionner les objets par opposition à l'espace.

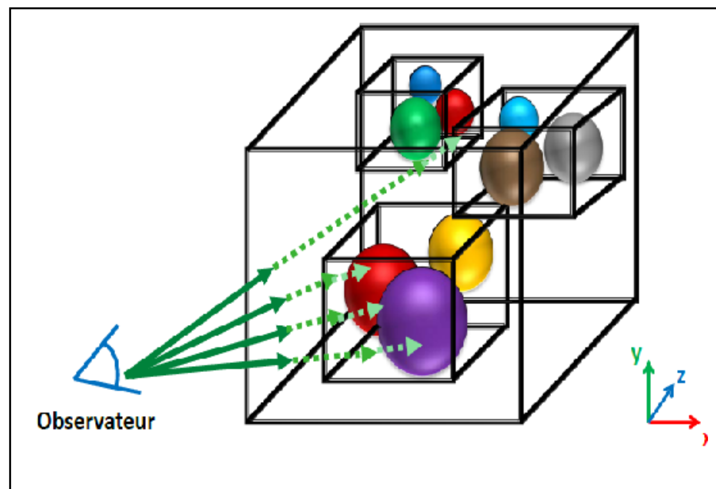


***Figure 3.11** : illustration de la structure BVH*

Le volume englobant d'un objet est une forme géométrique simple qui entoure la géométrie. De toute évidence, un rayon qui échoue à un test d'intersection pour un volume englobant ne

peut toucher aucune géométrie contenue à l'intérieur. La motivation derrière les volumes englobants est qu'un volume englobant simple possède généralement un test d'intersection moins cher que la géométrie qu'il contient. Cela est plus ou moins dépendant de la complexité des primitives géométriques ainsi que la complexité du volume englobant.

Une hiérarchie des volumes englobants consiste à un nœud racine avec un volume englobant contenant tous les autres volumes englobants et donc, toute la géométrie de la scène. Chaque nœud interne dans l'arbre a un certain nombre d'enfants qui sont soit des nœuds internes avec des volumes englobants associées ou des feuilles avec un certain nombre de triangles.

Le parcours d'un BVH se fait en utilisant une descente récursive simple et intuitive **[FL03]**.

2.2.3.1 Construction

Une mesure raisonnable de la qualité d'un BVH est le coût moyen de le parcourir, étant donné quelque rayon arbitraire. Il n'y a aucun algorithme connu pour la construction optimale d'un BVHs, car il n'est pas évident de savoir comment évaluer le coût moyen d'un parcours BVH.

Une fonction de coût, communément appelée la surface heuristique a été proposée. Elle est formalisée en utilisant les zones de la surface du parent et des enfants comme dans la relation suivante: $\boldsymbol{p}(\boldsymbol{hit}(\boldsymbol{c})|\boldsymbol{hit}(\boldsymbol{p})) \propto \frac{\boldsymbol{S_c}}{\boldsymbol{S_p}}$

Tel que :

- hit(n) est l'événement que le rayon touche le nœud n.
- $\boldsymbol{S_n}$ est la zone surface du nœud n.
- c et p sont respectivement le nœud enfant et le nœud parent.

La fonction de coût nous donne une estimation du coût de la hiérarchie quand elle intersecte un rayon arbitraire **[GF01]**.

Comme aucun algorithme n'est disponible pour construire efficacement une BVH optimale, plusieurs constructions heuristiques ont été proposées. Nous présentons ici quelques variables communes.

En pratique, le volume englobant le plus utilisé pour BVHs est the axis aligned bounding box (AABB). La construction est décrite dans l'algorithme 3.

Algorithm 3 Kay/Kajiya BVH construction

BV_Noued construire_arbre(triangles)

Si un seul triangle a été passé **Alors**

Retourner la feuille tenant le triangle

Si non

Calculer le meilleur axe de division

BV_Noued_résultat

résultat.enfant gauche:= construire_arbre (les triangles gauche du diviseur)

résultat.enfant droit := construire_arbre (les triangles droit du diviseur)

résultat.Boite_Englobante := Boite_Englobante de tous les triangles donnés

retourner résultat

Fin Si

- L'algorithme commence en assignant le premier triangle comme la racine de l'arbre.
- Pour chaque objet supplémentaire, la meilleure position dans l'arbre se trouve en évaluant une fonction de coût dans une descente récursive, en suivant le chemin le moins cher.
- Enfin, l'objet est inséré comme étant une nouvelle feuille, ou en remplaçant une feuille existante avec un nœud interne contenant l'ancienne feuille et le nouvel objet comme son enfant.
- En conséquence de cette approche, un nœud interne peut avoir un nombre arbitraire d'enfants.

2.2.3.2 Le parcours :

La méthode habituelle de parcourir d'un BVH est une méthode descente récursive :

- ❖ Pour les nœuds internes, nous testons le rayon pour intersection avec le volume englobant associé.
  - Si une intersection est trouvée, nous testons le rayon pour les nœuds enfants récursivement.

    Notez que contrairement au KD-tree, il faut visiter tous les enfants.
  - Si le rayon n'intersecte pas le volume englobant d'un nœud, on ignore ses enfants.
- ❖ Pour les feuilles, le rayon est testé pour une intersection avec le triangle détenu par la feuille, et le parcours reprend en fonction du contenu de la pile de la récursivité.

Le problème principal dans le parcours d'un BVH est l'ordre dans lequel les enfants d'un nœud sont parcourus.

Kay et Kajiya **[KJ86]** ont proposé une méthode, dont laquelle ils ont tenté de choisir l'enfant le plus proche de l'origine de rayons le long de la direction du rayon. Leur technique est assez complexe car elle exige une file d'attente prioritaire qui doit être maintenue, afin d'extraire rapidement le nœud suivant à parcourir.

Pour résumer : Dans cette section, nous avons introduit le BVH. Nous avons proposé un moyen de transformer l'arbre pour permettre le parcours sur le GPU. Notre méthode de parcours a été décrite en détail et les avantages et les inconvénients ont été discutés. Le plus grand avantage est la simplicité du code du parcours, tandis que le plus grand inconvénient est le fait que le parcours est susceptible d'être d'ordre fixe.

2.2.4 Les structures hybrides :

Ces dernières années, plusieurs structures d'accélération ont été proposées pour le rendu volumique qui sont essentiellement une combinaison de deux des structures mentionnées ci-dessus. Une structure hybride de BVH /grille uniforme fournit de bons résultats, en laissant les deux structures ne participe que dans la partie qui la maitrise, tant que les grilles uniformes sont placés dans les feuilles du BVH.

Une autre structure hybride est la grille uniforme hiérarchique (HUG). Elle est similaire à BVH/grille uniforme, dans des régions locales de la géométrie dense qui dispose de leurs propres grilles uniformes. La différence réside dans la façon dont ces grilles sont arrangées. Avec HUG les petites grilles sont placées dans des voxels de grilles plus grandes. Cela permet de multiples niveaux de grilles dans la hiérarchie.

3. Classification des différentes structures existantes :

Nous donnons dans le tableau ci-dessous une synthèse des structures de données utilisées pour accélérer le rendu volumique ; dans laquelle nous avons essayé de résumer les principaux avantages et les principales limites :

	Avantages	**Inconvénients**
La grille régulière	• Simples pour les implémentations matérielles • L'accès aux structures de données de la grille nécessite un temps constant	• Ne pas tenir compte du déséquilibre de la scène. • Référencer le même triangle à partir de plusieurs voxels.
Le KD tree	• Meilleure représentation des scènes irrégulières	• Difficulté du choix de l'emplacement du plan de coupe. • L'algorithme standard de parcours est récursif. • Le coût de parcours est élevé
BVH	• Un volume englobant simple possède généralement un test d'intersection moins cher que la géométrie qu'il contient	• Le parcours est d'ordre fixe

Tableau 3.1 : Classification des différentes structures de données

4. Conclusion :

Il existe plusieurs structures permettant de stocker des données volumiques. Ces structures sont plus ou moins adaptées aux diverses utilisations de ces données.

Pour cela, nous avons introduit dans ce chapitre une étude couvrant quelques structures de données qui servent à l'accélération du rendu volumique. Nous avons présenté des arguments pour inclure la grille uniforme, le KD-tree, et le BVH dans notre classification. Nous avons également montré l'efficacité de chacune et le degré de son applicabilité sur le GPU, d'après la classification présentée précédemment, nous pouvant conclure que la grille uniforme et le Kd tree (qui possède le même principe que l'octree) sont des structures convenables pour le rendu volumique.

Chapitre 04 :

Traitement des goulets d'étranglement

Chapitre 4: Traitement des goulets d'étranglement

1. Introduction :

Les dernières générations de cartes accélératrices 3D comportent une unité de traitement graphique (GPU), qui peut être vu comme un co-processeur spécialisé pour effectuer des tâches graphiques. Cette spécialisation lui donne moins de souplesse par rapport au CPU, mais lui permet aussi d'exécuter les instructions en parallèle à un taux beaucoup plus élevé que les CPUs. Elle permet également d'améliorer la vitesse par rapport aux CPUs, ce qui rend les cartes graphiques un moyen particulièrement intéressant pour développer des calculs intensifs.

Ce chapitre explore les moyens pour tirer parti des GPUs dans le domaine du rendu volumique; son objectif principal est de permettre d'adapter et d'équilibrer la charge sur les goulets d'étranglement (figure 4.1) en étudiant la façon avec laquelle ces goulets seront traités au niveau du pipeline graphique, puis en introduisant les deux techniques ''Briking et l'octree'' afin d'atteindre une exploitation optimale de la puissance du matériel graphique. En outre, nous avons également étudié quelques techniques de rendu interactif, en projetant la lumière beaucoup plus sur la technique du ray casting et les optimisations qu'elle a ajouté.

Une démarche pour améliorer le ray casting, qui est le meilleur moyen de rendre les volumes **[EK05] [KE06]** a été introduite dans ce cadre, en tirant profit du GPU. Nous avons améliorer cette approche en intégrant la technique near neighbor skipping, une technique bien connue dans les algorithmes à base de ray casting, en profitant des avantages du parallélisme disponibles dans le pipeline GPU, qui peut être accéléré de manière significative en prenant compte des différents goulets d'étranglement rencontrés dans le matériel graphique.

2. Les goulets d'étranglement au niveau de pipeline graphique

Le pipeline de rendu est le processus général qui est utilisé pour décrire des scènes virtuelles en trois dimensions. Une telle scène se compose de primitives géométriques plates, comme des points, des lignes, des triangles et des polygones. Le pipeline de rendu peut être implémenté dans le matériel à des degrés différents **[DR08]**.

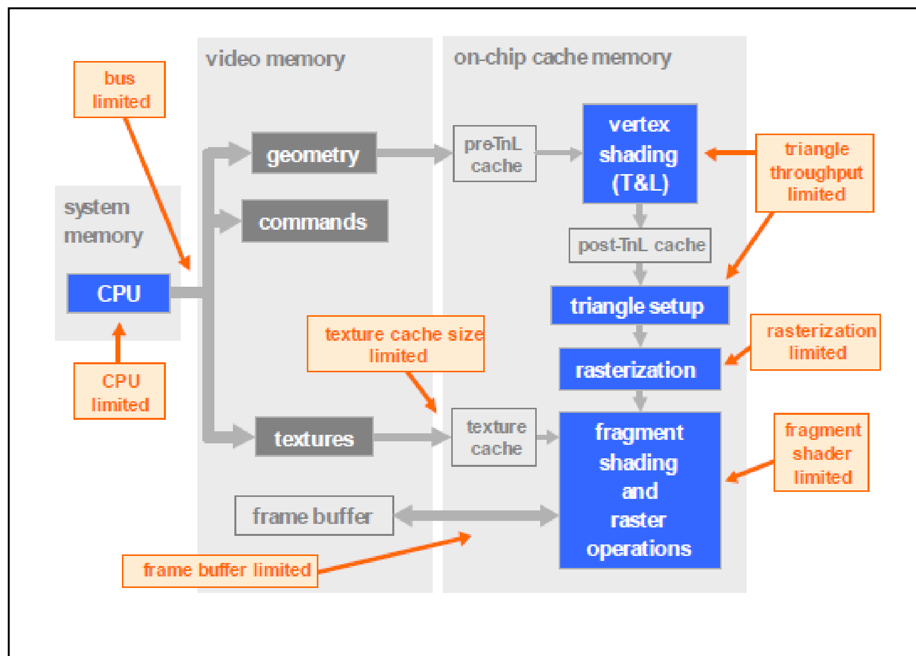


***Figure 4.1** : Le pipeline graphique et ses goulets d'étranglement, le gris clair: les unités de mémoire, le gris foncé: structures de données, bleu: les unités de traitement, l'orange: les goulets d'étranglement* **[DR08]**.

Le processus permettant de rendre une image consiste à traverser toutes les primitives dans l'ordre afin de transformer leurs coordonnées de scène en coordonnées de la caméra. Un modèle d'éclairage est réalisé sur les primitives, et les résultats sont stockés sous forme de couleur par vertex. L'étape suivante est la rastérisation des primitives, qui consiste à convertir les primitives mappées en fragments, qui correspondent aux endroits de pixel dans le frame buffer, et contiennent certaines propriétés comme la couleur, les coordonnées de texture, et la profondeur (Z buffer).

Avant d'être placés dans le tampon d'image (frame buffer), chaque fragment peut être soumis à une série de tests et de modifications, comme le test de stencil (stencil test), test de

profondeur, et le mélange (blending). Finalement, l'image 2D qui a été rendue dans le frame buffer, est affichée sur l'écran **[JK03]**.

La figure 4.1 illustre le pipeline graphique, employé pour le rendu volumique basé sur GPU. Nous discutons les points les plus importants dans le pipeline qui engendrent un goulet d'étranglement :

2.1 Définition des goulets d'étranglement :

Dans un processus physique un **goulet d'étranglement** est considéré comme un point étroit qui provoque un engorgement. Dans le domaine de l'informatique, un goulet d'étranglement est un point d'un système limitant les performances globales, et pouvant influencer négativement les temps de traitement et de réponse. Les goulets d'étranglement peuvent être matériels et/ou logiciels **[DR08]**.

2.2 Les causes et les motivations :

Dans ce qui suit, nous discutons les points les plus importants dans le pipeline qui engendrent un goulet d'étranglement :

1. **Le bus :** Les données volumiques doivent être transférées sur le bus de la mémoire centrale à la mémoire de la carte graphique. Puisque c'est la partie la plus lente du pipeline entier, ces transferts doivent être fait le moins possible.
2. **Le débit du Triangle** (Triangle throughput) **:** Le débit du triangle est principalement limité par le vertex shader et la phase de configuration des triangles. Une implémentation du rendu volumique à base de texture-mapping ne s'applique que sur quelques triangles, mais les techniques de space-skipping peuvent augmenter considérablement la quantité de triangles. Si le nombre de triangle devient trop élevé, cela deviendra un facteur limitant pour le taux d'affichage.
3. **Rastérisation :** Lorsque nous appliquons un rendu volumique de la scène basé sur des tranches de texture, la majorité des pixels sur l'écran sont accessibles à plusieurs reprises. Les techniques d'élimination des parties cachées peuvent être utilisées pour réduire la quantité de pixels accessibles, mais cela augmente aussi le nombre de triangle **[JF05]**.
4. **La taille du cache texture** : Lors de l'étape de rastérisation, la recherche de texture est l'une des opérations les plus gourmandes en consommation de temps. Lorsque la texture est chargée dans le cache, l'opération de recherche sera plus rapide.

5. **Le fragment shader** : Le fragment shader influe sur la durée de l'étape rastérisation. Des fragments programmes simples, comme l'application d'une table de recherche, généralement ne limite pas la vitesse d'affichage, mais des opérations plus complexes, tels que l'éclairage spéculaire, les fonctions de transfert multi-dimensionnelle ou un rendu pré-intégré, peuvent générer un goulet d'étranglement. Ceci est surtout remarquable lorsque le fragment shader effectue des recherches multiples de texture (par exemple, le calcul du gradient pour l'éclairage spéculaire) **[PU04]**.

3. Description de la méthode :

Le problème abordé pendant ce travail se décompose en plusieurs aspects.

- ✓ Le premier aspect est le choix d'une structure de données permettant un stockage dynamique efficace des scènes volumiques que nous souhaitons traiter. Pour éviter un gaspillage de la mémoire, cette structure doit permettre à la fois un stockage compact des données et un rendu volumique efficace sur le matériel graphique. Vu que les données même condensées ne peuvent tenir intégralement en mémoire, cette structure doit inclure notamment des mécanismes de chargement progressif et dynamique, et permettre une mise à jour dynamique rapide en fonction du point de vue et de la résolution nécessaire des données.
- ✓ Le deuxième aspect abordé consiste en l'algorithme de rendu de données volumiques qui doit être compatible avec cette structure de données, en tirant le meilleur profit possible du matériel graphique.

3.1 La structure de données choisie :

Comme nous avons vu dans le chapitre précédent, il existe plusieurs structures de données utilisées pour accélérer la procédure de rendu.

Dans cette section, nous allons détailler la structure de donnée choisie ainsi que la motivation de ce choix **[CC07]**.

3.1.1 Subdivision en briques régulières (Bricking) : La subdivision en briques régulières consiste à :

- diviser le volume en morceaux, appelé briques (figure 4.2), afin de faire face à des tailles d'ensembles de données supérieures à la taille de la mémoire texture du matériel graphique.
- Notons que ces briques doivent contenir les valeurs originales des voxels, donc les valeurs avant d'appliquer la fonction de transfert.

- Les briques sont chargées dans la mémoire vidéo comme des textures 3D.
- Lorsque la quantité de données dans la texture dépasse la taille de mémoire de texture disponible, la texture est échangée entre la mémoire principale et la mémoire de texture.
- Si une brique demandée ne réside pas dans la mémoire de texture, elle est chargée à partir de la mémoire principale, en remplaçant une texture résidente.
- La taille optimale de la brique doit être défini en fonction de la mémoire texture disponible, la taille de la texture optimale, la nature de l'ensemble de données, la surcharge provoquée par les chevauchements, et les contraintes imposées par le matériel graphique.

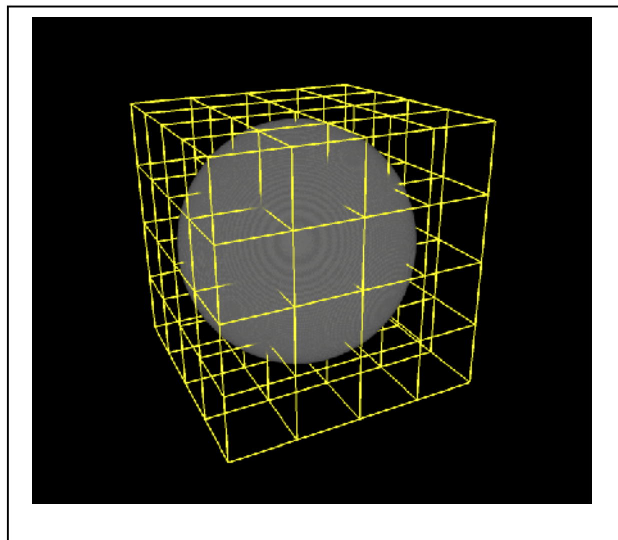


***Figure 4.2** : partition du volume en* 4^3 *briques.*

Nécessité de briques régulières

Plusieurs raisons font qu'il est très intéressant de ne pas stocker les voxels indépendamment dans la structure hiérarchique mais de conserver des blocs 3D -ou briques- de données régulières :

- Tout d'abord, le coût de la structure en termes d'occupation mémoire doit être raisonnable par rapport aux données.
- Ensuite, le parcours des briques régulières est bien plus simple et efficace que celui d'une structure plus complexe.
- Et enfin, la génération de briques régulieres de données, ainsi que leur transfert vers le GPU est toujours plus rapide et plus efficace qu'une génération indépendante voxel par voxel **[CC07]**.

3.1.2 L'octree :

Le faite d'ignorer le rendu des briques vides, implique que la charge sur le goulet d'étranglement de rastérisation est réduite. La charge est réduite en plus par l'application de l'octree. Chaque brique possède son propre octree. Chaque nœud de l'octree correspond à une partie cubique du voxel, qui peut être divisé en huit parties, correspondant aux nœuds enfants (voir figure 4.3). L'octree est stocké dans la mémoire principale. Il ne décrit que la géométrie des données visibles. Les données actuelles du voxel se trouvent dans les textures de briques **[KE05]**.

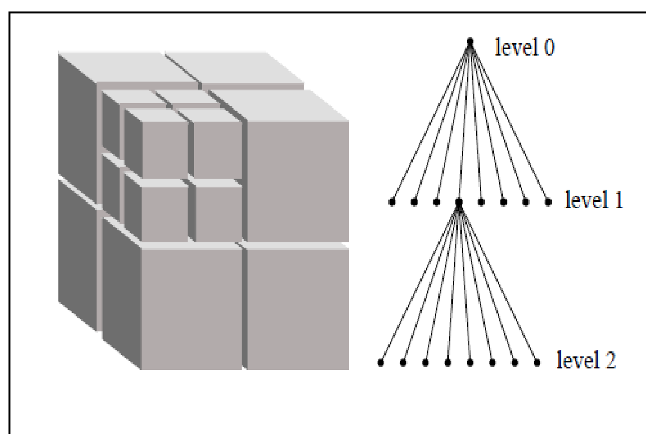


Figure 4.3 *: Une division octree, et son arbre*

Pour l'interpolation tri-linéaire, une cellule est définie comme un cube avec les valeurs de voxels adjacents attribuées à ses huit coins. Pour chaque position au sein de la cellule, une valeur d'intensité est définie comme une interpolation tri-linéaire des valeurs des coins. Par conséquent, une cellule ne peut être complètement annulée ou vide que si ses huit valeurs des coins sont complètement transparentes (= 0) après avoir appliqué la fonction de transfert **[JK03]**.

Chaque nœud de l'octree porte une variable décrivant le rapport r des données visibles à l'ensemble des données au sein de son cube. Au niveau final de l'octree, chaque nœud représente une cellule unique, et est considérée rempli complètement (r = 1) ou vide (r = 0). A chaque niveau de nœuds supérieur dans l'octree, le rapport peut être calculé par la moyenne des rapports de ses enfants. Ce calcul ne doit être effectué que lorsque la fonction de transfert est changée.

Le rendu d'une image signifie que les briques doivent être traitées dans un ordre de l'arrière ver l'avant. Pour chaque brique, l'octree correspondant est traversé, en commençant par son nœud parent.

Selon son rapport r, il ya trois façons de traiter un nœud:

- r = 0: Le nœud est complètement vide. Il ne doit être ni dessiné, ni traversé.
- 0 <r <seuil: Les nœuds enfants seront traversés, et à chaque nœud enfant ce processus sera appliquée de manière récursive.
- r seuil: Le nœud sera dessiné complètement, il ne sera pas parcouru **[DR08]**.

Cette stratégie est appliquée pour éviter la surcharge de traverser l'octree de la racine jusqu'aux feuilles. En plus, cette approche nous permet de mettre à jour l'octree à la volé, quand la fonction de transfert change. Les nœuds de l'octree au niveau L contiennent la valeur minimale et maximale du voxel qui est représentée par les échantillons du voxel dans leurs cubes correspondants. Ce sont des données constantes, indépendamment de la fonction de transfert.

Pour réduire la charge sur le pipeline graphique, et donc le GPU, L'octree est généré et parcourue sur le CPU. L'octree réduit le temps que le GPU passe dans le traitement des données qui ne contribuent pas à l'image finale **[DR08]**.

L'algorithme de rendu volumique actuel, ainsi que l'interpolation, la fonction de transfert, et éventuellement, de l'éclairage spéculaire, sont exécuté par le GPU.

3.2 La méthode (algorithme) accélératrice de rendu :

En réalité, lorsqu'on effectue un rendu volumique de la scène, une fraction de tous les voxels contribue effectivement à l'image finale, car une quantité relativement faible de voxels sont d'un intérêt considérable. Pour les données Médicales 3D (obtenues par exemple, de l'échographie, scanner, ou IRM), les structures anatomiques d'intérêts encapsulés dans les ensembles de données occupent uniquement une partie de l'ensemble entier des données : typiquement 5% à 40% de tous les voxels contiennent des données visibles **[CA02]**.

Notre objectif vise à atteindre le maximum d'avantages dans l'exploitation d'éliminer des parties vide du volume (space-skipping). La nouveauté qui a été introduite réside dans la division du space-skipping en deux étapes, une division épaisse à l'aide du Bricking (figure 4.4 a) et une division plus fine en utilisant l'octrees (figure 4.4 b). Ces étapes sont basées sur une analyse de l'étranglement rencontré dans le pipeline graphique lors de l'exécution du rendu volumique.

- ✓ La première étape, le Bricking, sert à découper le volume en des briques de texture. Les briques sont chargées dans la mémoire vidéo, pour servir comme des données pour l'algorithme de rendu, qui est exécuté par le GPU. Les briques traitent le goulet d'étranglement résultant du bus et de la taille du cache de la texture.

Pour alléger encore la charge sur le fragment shaders, l'algorithme d'arrêt précoce de rayon ou the early ray termination (voir section 3.1.1) est appliqué à chaque brique en plus. Cela est particulièrement bénéfique pour des ensembles de données fortement chargés.

✓ La deuxième étape est l'utilisation d'un octree au sein de chaque brique. L'octree traite le goulet d'étranglement de rastérisation. Comme il sera démontré, les deux étapes doivent être équilibrées, car le fait de se débarrasser d'un goulet d'étranglement peut surcharger un autre goulet d'étranglement (par exemple, le goulet d'étranglement de rastérisation et le goulet d'étranglement de débit du triangle).

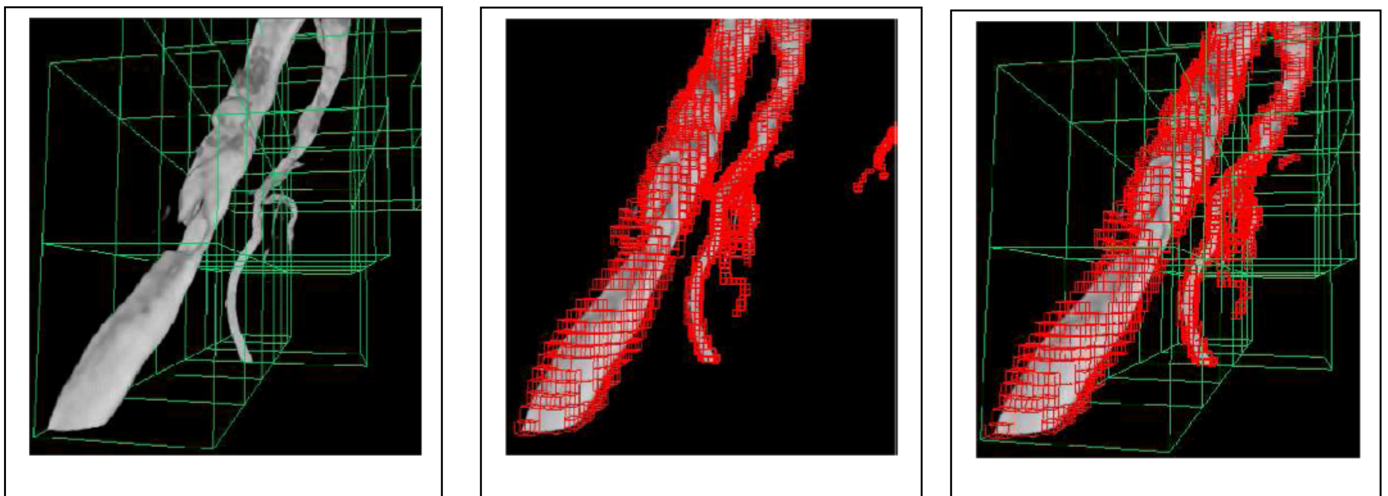


Figure 4.4: *Le même fragment de volume, rendu avec (a) Bricking des cubes visibles, (b) octree visibles (notez les tailles des cubes différentes) et (c) les deux : Bricking et l'octree visible* **[DR08]**.

Le rôle de la fonction de transfert dans le rendu volumique est de transformer l'information scalaire du voxel à des propriétés optiques (par exemple, la couleur et l'opacité). L'approche décrite ci-dessus est implémentée de telle sorte que la flexibilité de changer la fonction de transfert au temps d'exécution est préservée. Pour cela, les valeurs du scalaire non modifiés sont stockées dans les textures de briques, et un fragment shader est utilisé pour rechercher les valeurs RGB après avoir effectué l'interpolation des valeurs du voxel.

Lorsque l'octree dépendra de la visibilité des données, et donc sur la fonction de transfert, ils doivent être recalculés en cas de changement de la fonction de transfert. Cela peut être fait à la volée, comme on le verra ci-dessous **[CA02]**.

Après avoir traité les goulets d'étranglement, l'algorithme de rendu est accéléré beaucoup plus en effectuant un ray casting optimisé par l'intégration de la technique du near neighbor skipping.

4. La contribution :

Dans le rendu volumique à base de CPU, l'approche la plus naturelle est celle du raycasting, qui peut être enfin implémenté sur la carte accélératrice 3D **[JK03]**.

Une première tentative d'une telle implémentation a déjà été publié avant le début de cette étude, donc on a utilisé cela comme un point de départ et cherché des moyens de l'améliorer.

Deux axes majeurs sont mis en place dans ce travail: une combinaison de briking et de la technique de l'octree pour réduire les goulets d'étranglement rencontrés au niveau du pipeline ainsi qu'une méthode d'intégration de la technique 'the near neighbor skipping ou le plus proche voisin' (voir section 3.1.3) pour le raycasting à base de GPU **[KE05]**.

3.1 Les techniques introduites dans la méthode :

3.1.1 Early ray termination (l'arrêt précoce de rayon):

Cette technique est utilisée, lorsque le volume est rendu dans un ordre d'avant vers l'arrière. Pour chaque rayon lancé vers pixel, une fois qu'une matière dense a été rencontrée, les autres échantillons n'ont pas une contribution significative au pixel et peuvent être donc ignorés. Le rôle de cette technique est de réduire le calcul en éliminant les voxels invisible dans le volume à visualiser.

Avant qu'une brique soit rendue, le early ray termination est appliqué à ses pixels. Cela a été testé par l'exécution d'un fragment shader, tout en dessinant une boîte englobante autour de la brique. Le fragment shader écrit la valeur maximale dans le tampon de profondeur pour les rayons saturé **[RS00]**.

Lorsqu'on découpe la texture de la brique en tranches, le test early z (Technique consistant à effectuer le test de profondeur avant d'appliquer un shader à un pixel. Ainsi si le pixel est caché par un autre, cette technique permet d'économiser le coût d'application du shader) permettra d'éviter l'exécution de toute opération de fragment pour ces rayons, réduisant ainsi la charge sur les goulets d'étranglement au niveau de la rastérisation et du fragment shader. Le early ray termination est effectuée uniquement une fois pour chaque brique (comme par exemple, pour chaque nœud de l'octree ou chaque échantillon).

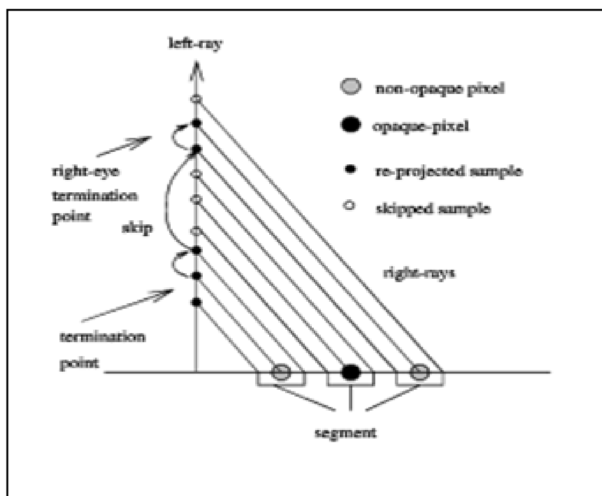


Figure 4.5*: l'arrêt précoce de rayon*

4.1.2 Le raycasting (lancer de rayon volumique):

4.1.2.1 le principe :

Le Ray casting est l'approche la plus habituelle de rendu volumique. Son principe consiste à lancer un rayon est à partir de l'emplacement des caméras vers chaque pixel visible sur la surface du volume de la boîte englobante et les valeurs sont échantillonnées à des intervalles réguliers le long du segment de chaque rayon à l'intérieur du volume.

le ray casting est similaire au lancer de rayons, une approche commune, mais une technique de calcul très intensive utilisé pour le rendu des images réalistes, voir tableau 4.1, le raycasting repose sur le lancer de rayon pour chacun des pixels de l'image, mais là où il diffère c'est que dés qu'une intersection a été trouvé l'algorithme s'arrête là et ne lance pas de rayons secondaires. Le coût élevé est provoqué du fait que l'objet le plus proche avec lequel chaque intersection de rayons doit être déterminé, et chaque rayon peut aussi engendrer deux nouveaux rayons - une réfléchit, l'autre réfracté. Aucune de ces propriétés ne s'appliquent au ray casting, mais il provoque toujours un calcul intense vu le grand nombre d'échantillons qui doivent être prises le long de chaque rayon pour obtenir une qualité acceptable **[FC04]**.

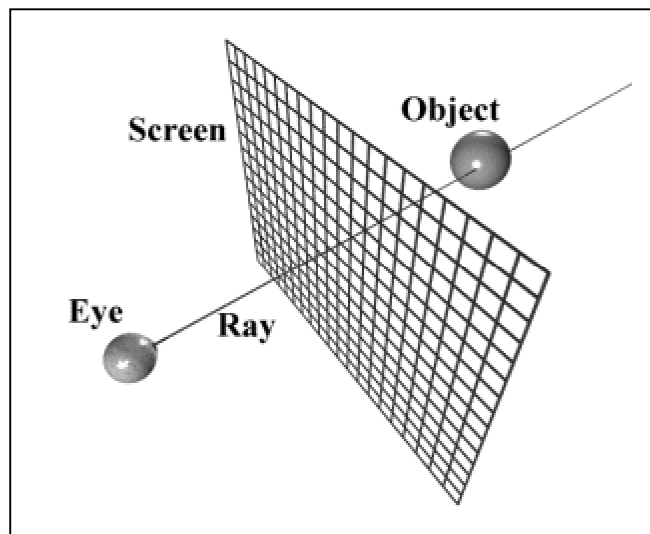


Figure 4.6: *Le Ray casting*

Ray-Casting	Lancer de rayon
Principe: les rayons sont collectés et lancer en groupe. Par exemple: avec une résolution 320x200, un ray-casting lance seulement 320 rayons (le nombre 320 vient du fait que l'affichage à résolution horizontal 320 pixel, donc 320 colonne vertical).	**Principe**: chaque rayon est lancé séparément, donc chaque pixel est lancé par un rayon. Par exemple: avec une résolution 320x200, le lancer de rayon doit lancer 320x200 (64,000) rayons. (à peu prés 200 fois plus lent que le ray-casting.)
Vitesse: très rapide comparativement au lancer de rayon; convenable pour les processus temps réel.	**vitesse**: lent; non convenable pour les processus temps réel.
Qualité: les images produites ne sont pas très réalistes.	**Qualité:** les images produites sont réalistes, des fois très réalistes.

Tableau 4.1 *: comparaison entre lancer de rayon et le ray casting*

3.1.2.2 les différentes passes de l'algorithme :

Grâce à l'évolution du matériel graphique vers plus de programmabilité, de nouvelles méthodes implémentant directement un lancer de rayons sur GPU commencent à être proposées. La méthode consistant à ne lancer qu'un rayon primaire par pixel de l'image, en opposition au lancer de rayon utilisé plutôt pour du rendu de surfaces et dans lequel des rayons secondaires sont réémis de manière récursive (ombres, reflets).

Le but principal de la méthode est de permettre l'arrêt du calcul d'un rayon lorsque celui ci a atteint une opacité totale. Pour cela, une fonctionnalité des GPU appelée test précoce de profondeur (Early Depth Test) est utilisée **[JK03]**. Cette fonctionnalité permet au matériel de détecter précocement les pixels qui seront rejetés par le test de profondeur (depth-test) et évite ainsi leur traitement coûteux par le fragment shader. Le ray casting est une méthode multi-passe dans laquelle les données sont stockées dans une texture 3D et qui se décompose de la façon suivante :

- ***Passe 1*** : Rendu vers une texture des faces avant de la boite englobante du volume de données et inscription pour chaque fragment des coordonnées de texture au point d'entrée du rayon dans le volume.
- ***Passe 2*** : Rendu vers une autre texture des faces arrière et calcul par fragment (fragment shader) du vecteur direction du rayon correspondant via les coordonnées récupérées dans la texture précédente. Le résultat de ces deux premières passes est présenté figure 4.8.
- ***Passes 3 à N principales*** : N rendus successifs des faces avant du cube. Pour chaque rendu, le fragment shader accumule M échantillons de la texture 3D le long du rayon correspondant au fragment, la géométrie de ce dernier étant récupéré via les textures calculées par les deux premières passes. Les résultats de l'accumulation est combiné à chaque passe avec le résultat de la passe précédente et écrit dans une texture.
- ***Passes 3 à N secondaires*** : Ces passes intermédiaires permettent d'inscrire une profondeur dans le tampon de profondeur pour l'utilisation du rejet précoce de fragment. Une valeur de profondeur forçant le rejet du fragment aux passes suivantes est ainsi inscrite pour chaque fragment ayant atteint l'opacité maximum **[KW03]**.

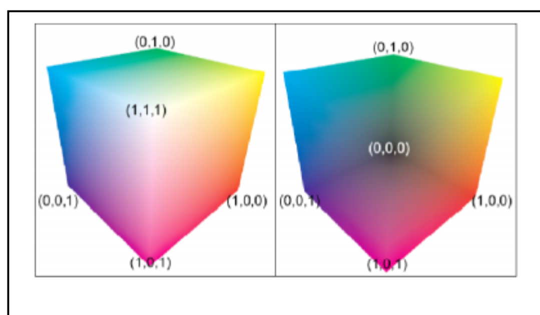


Figure 4.7 *: Résultats des deux premières passes de la méthode de Ray-Casting* **[JK03]**

L'inconvénient majeur du ray casting est le fait qu'il est très cher en termes de calcul et il est directement proportionnelle à la taille du volume de données et la taille de l'image finale qui

sera projetée à l'écran. Comme les données deviennent volumineuses, le processeur ne peut plus gérer une quantité énorme de calculs nécessaires pour le ray casting à des taux d'affichage interactive. A cet effet, nous nous sommes orientés vers l'intégration de la technique de plus proche voisin détaillée ci-dessous.

3.1.3 La technique de plus proche voisin " near neighbor skipping":

Plusieurs techniques d'optimisation sont couramment utilisées dans le rendu volumique à base de CPU, telle que la « early ray termination », et « near neighbor skipping ». Vu que la plupart des volumes de données contiennent de grandes régions d'espace vide ou des voxels rendu totalement transparent par la fonction de transfert en cours d'utilisation, des économies importantes peuvent être réalisées en utilisant des méthodes pour détecter et ignorer plus efficacement ces régions.

L'approche de plus proche voisin (near neighbor) stocke pour chaque voxel dans le volume, la distance la plus proche de voxel visible. Ainsi, après l'échantillonnage d'un voxel, la distance la plus proche du voxel visible est connue et il est alors sûr de passer directement à cette distance le long du rayon avant d'échantillonner le voxel prochain.

Bien qu'il soit simple de mettre en œuvre cette technique au niveau du logiciel une fois que toutes les distances soient calculées, il est nettement plus efficace que l'on implémente sur un GPU pour réduire le calcul étant donné son architecture parallèle.

La façon la plus commune d'éviter l'exécution d'un fragment shader donnée est de faire échouer le test de la profondeur. Cependant, les fragments de programmes qui modifient la valeur de profondeur ne peuvent normalement pas être ignorés, mais en faisant usage d'une extension d'OpenGL, *la GL_EXT_depth_bounds_test*, même les fragments programmes décrivant une valeur de profondeur peuvent être ignorés sans risque, car cette extension ne concerne pas la valeur de profondeur de fragments entrants. En utilisant de cette extension, il est possible de contrôler précisément les fragments qui sont traitées à un moment donné, ce qui est exactement la chose nécessaire pour l'implémentation du ''near neighbor'' optimisée **[KE05]**.

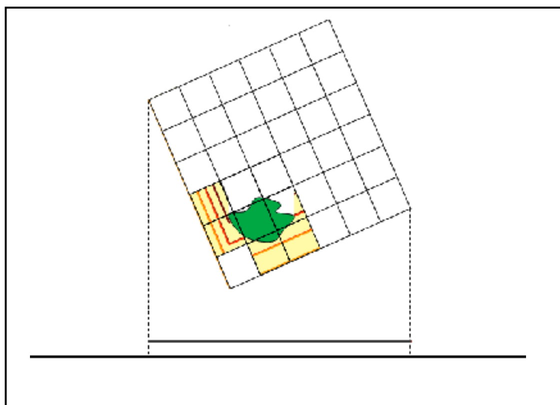


***Figure 4.8** : near neighbor skipping*

En faisant usage d'une structure de données de ''near neighbor'' optimale, où la distance au plus proche voxel est donnée uniquement dans une direction spécifique (la direction du vue), une réduction dans la quantité des fragments traités peut être réalisée.

4. Comparaison de notre approche contre le rendu volumique basé tranches de texture :

Pour valoriser l'approche que nous proposons et motiver notre choix de la technique du ray casting comme algorithme de rendu, nous allons la comparer par les résultats de Daniel Ruijters et Anna Vilanova **[DR08].**

Dans leurs travaux, ils ont adopté l'algorithme de rendu volumique basé sur les tranches de texture qui est une approche classique.

4.1 Principe

La méthode de rendu volumique basé tranches de texture est l'une des premières méthodes utilisant fortement la capacité du matériel 3D à avoir été proposée. Le principe de la méthode consiste à échantillonner le volume de données à l'aide d'une série de polygones de support planaires. Ces polygones sont texturés avec une texture 3D (généralisation à un tableau 3D des textures 2D classiques) contenant soit les données qui seront transformées à la volée à l'aide d'une fonction de transfert soit directement les propriétés d'émission et d'absorption du matériau (pré-classification). Ces polygones sont ainsi dessinés d'arrière vers l'avant ou d'avant vers l'arrière par rapport au point de vue. Les polygones de support peuvent être alignés selon un axe du volume ou parallèles au point de vue **[JF05]**.

4.2 L'algorithme de rendu volumique basé tranche de texture:

1. Echantillonner un certain nombre de tranches de données perpendiculairement à la direction d'observation.

2. Rendu chaque tranche comme une texture 2D.
3. Mettre à jour les couleurs et les valeurs alpha-blending selon une fonction de transfert sélectionnée. **[NB09]**

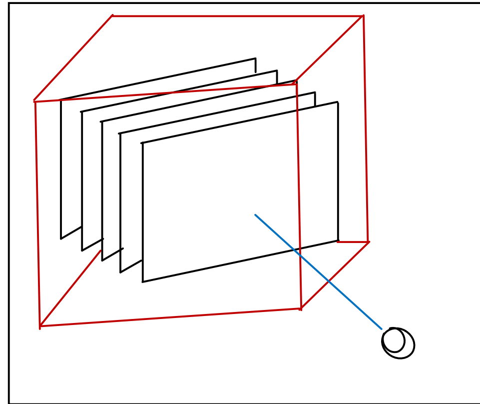


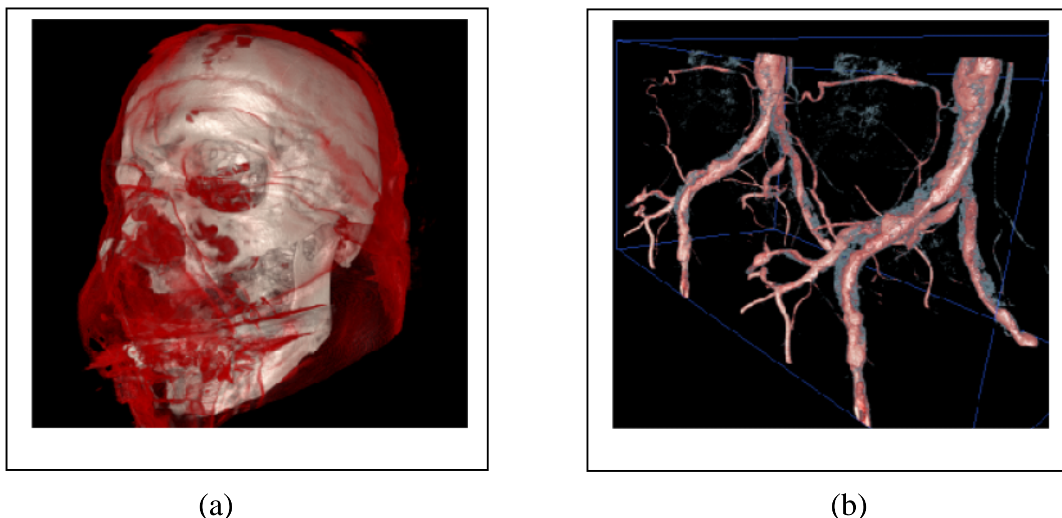
Figure 4.9 *: Rendu volumique basé tranche de texture*

Malgré que cette technique est considéré comme l'une des plus courantes techniques de rendu volumique, en utilisant les profits du matériel graphique, elle devenue plus rapide, mais elle souffre de plusieurs limitates.

4.3 Inconvénients de rendu volumique basé sur des tranches de texture :

- Les textures doivent être recalculées en cas de déplacement de la caméra
- Les artefacts dùs à la rastérisation
- Baisse de flexibilité (difficile à optimiser la technique ou à l'étendre)
- La réfraction de la lumière du volume est très difficile à l'implémenter. **[NB09]**

Nous présentons ci-dessous, quelques résultats pris de **[DR08]**:



(a) (b)

Figure 4.10 *: volumes de test : (a) 512x512x512 voxels, (b) vasculaires 642 x 642 x 1284 voxels,*

Malgré que l'algorithme du ray casting semble lent par rapport à l'algorithme basé sur les tranches de texture, il génère des images de haute qualité et calcule quelques effets optiques additionnels (par exemple la réfraction). Un effet qui est impossible par l'algorithme basé sur des tranches de texture, cela se manifeste très clairement dans les résultats de la figure 4.10.
La comparaison de nos résultats avec celles-ci illustrés par la figure 4.10 présentée dans le chapitre 5, section 7.2
Le ray casting optimisé par l'introduction de l'algorithme du plus proche voisin permet de générer des images de bonne qualité et avec un taux d'affichage interactif grâce à la possibilité d'ignorer les fragments vides qui est offerte par le near neighbor skipping

5. Conclusion

Les recherches effectuées au cours de cette étude ont abouti à deux grandes nouvelles techniques implémentées sur le GPU: La première approche consiste en deux étapes de la technique space-skipping et le early ray termination qui a été adapté pour lever les goulets d'étranglement rencontrées dans le pipeline graphique. Puis nous avons introduit la technique Ray casting/ near neighbor skipping comme une amélioration des limites rencontrées dans la première approche.
Les deux techniques ont une utilité immédiate dans le rendu volumique et peuvent être facilement mis en œuvre dans des systèmes de rendu volumique existants.
Dans la première approche, la totalité du volume est coupé en briques, et à partir de ces briques les textures 3D sont créées. Les briques vides ne sont jamais dessiné, ni stockés dans la mémoire vidéo, et donc le goulet d'étranglement de bus est éliminé. La taille optimale de brique dépend de la nature des données, la mémoire de texture disponible, et la taille du cache de texture. Puisque la texture de brique ne dépend pas de la fonction de transfert, elle doit être créée une seule fois pour les données statiques.
L'octree, qui constitue la deuxième approche, focalise sur les données à éliminer qui ne sont pas visible après l'application de la fonction de transfert. De cette façon, le goulet d'étranglement de rastérisation est réglé.
Comme l'octree dépend de la fonction de transfert, il doit être recalculé en cas de changement de cette fonction.
Dans ce chapitre, il a été également montré la façon avec laquelle les goulets d'étranglements ont été éliminés par une approche en deux parties. D'abord, le goulet d'étranglement de bus et de la taille du cache texture a été réglé par le Bricking, et le goulet d'étranglement de

rastérisation a été résolu par l'octree. Le goulet d'étranglement de rastérisation et de fragment shader ont également été levés en employant le early ray termination. Vu que la fonction de transfert ne contribue qu'à recalculer les octrees, et non pas le rechargement des briques, elle peut également être changé rapidement et de manière interactive.

En ce qui concerne la deuxième approche : le ray casting; l'objectif principal de la méthode est de permettre l'arrêt du calcul d'un rayon lorsque celui ci a atteint une opacité totale. L'amélioration ajoutée ici est l'intervention de 'near neighbor skipping'. Cette technique permet de détecter précocement les espaces vides qui seront rejetés et évite ainsi leur traitement coûteux par le fragment shader.

L'influence des paramètres de la structure sur le rendu définit par le choix de la taille optimale de la brique ainsi que la profondeur de l'octree seront traités en détail dans le chapitre suivant.

Chapitre 05 :

Accélération de rendu volumique en utilisant Briking+octree

Chapitre 5: Accélération de rendu volumique en utilisant Briking+octree

1. Introduction :

Avec l'importante croissance des simulations numériques dans certains domaines scientifiques, les communautés de la visualisation scientifiques et plus particulièrement du rendu volumique se trouvent confrontées à des masses de données (champs scalaires, champs vectoriel…) de plus en plus importantes. Contrairement aux méthodes de rendu de surfaces (par exemple, avec un maillage sous la forme de polygones), le rendu volumique consiste à visualiser des champs de données par transparence, permettant ainsi de visualiser ce qui se trouve à l'intérieur essentiellement par la manipulation d'une fonction de transfert.

Durant ces dernières années, de très nombreuses contributions en rendu volumique ont été apportées afin de satisfaire les besoins d'exploration de l'utilité des GPUs et d'apporter plus d'interactivité pour le processus de rendu. Les approches étant au départ purement logicielles puis réservées au matériel haut de gamme. L'avancé considérable de l'architecture des cartes graphiques permet depuis quelques années la visualisation de données numériques sur PC standard, donnant ainsi lieu à une nouvelle famille d'algorithmes tirant profit des possibilités offertes par ce matériel.

Dans ce chapitre nous avons proposé une structure de données permettant un stockage dynamique et compact des scènes volumiques, cette structure doit confrontée les goulets d'étranglement crées au niveau du pipeline et bien entendu devrait permettre un rendu efficace sur le matériel graphique.

Nous allons également appliquer l'algorithme de rendu Raycasting, c'est l'algorithme le plus naturel qui permet d'atteindre un niveau haut de qualité de rendu volumique ainsi qu'il est bien adaptée pour être implémenter sur les GPUs modernes.

Pour finir nous allons présenter quelques résultats de l'application que nous avons développée, en indiquant l'utilité de notre approche avec une discussion de l'influence du paramètre de structure sur le rendu temps réel.

2. Étude de cas des scènes à traiter :

Les scènes étudiées dans le cadre de ce mémoire sont de nature divers, détaillée et intrinsèquement multi-échelle. Nous avons testé notre programme en introduisant des objets spécifiques par exemple un robot, un avion, ou une statue. On peut également penser à une représentation intégralement volumique de paysages forestiers. En outre, nous pouvons également traiter des données volumiques réelles (des données médicales d'os par exp)

Mais comme l'objectif visé dans ce mémoire consiste à réaliser un rendu temps réel d'une scène complexe alors les objets qui nous intéresse doivent avoir un grand nombre de facettes, nous avons pris comme exemple:

- La statue comporte 100000 facettes
- Le robot comporte 235000 facettes
- L'avion comporte 206000 facettes

3. Nature de données à visualiser :

En raison de traitement informatique, les valeurs de données sont généralement converties en entier ou flottants codé sur 1, 2, 4, voire 8 octets. En raison des limitations mémoires des cartes graphiques ainsi que de la taille des données souvent importante, des entiers codés sur 1, voire 2 octets sont couramment utilisées en visualisation.

Afin de bénéficier des performances des cartes graphiques, il est nécessaire de stocker les données sous la forme d'une texture 3D (tableau tridimensionnel de valeurs discrètes), qui constitue un maillage structuré de l'espace. On affecte ainsi à tout voxel une valeur calculée en fonction des données initiales, ce qui peut engendrer dans certains cas un rééchantillonnage.

4. Choix de la structure d'accélération

4.1 La structure de brique régulière :

La structure de brique régulière est la structure la plus facile à l'implémenter sur le GPU, car quand on la parcours, elle nécessite un nombre minimal d'accès aux données en outre le parcours est linéaire. Dans cette structure régulière, la scène est divisé uniformément en voxels et ces derniers contiennent des triangles ou partie d'un triangle.

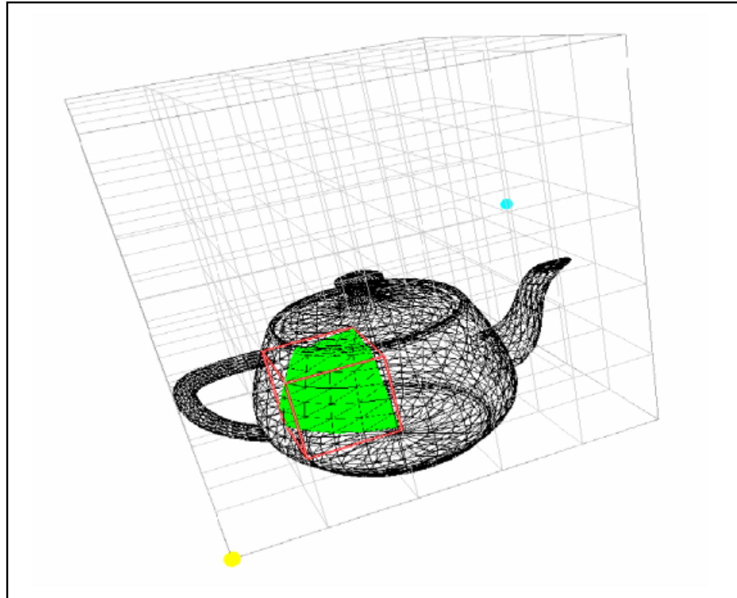


***Figure 5.1**: La structure de données brique*

4.2 La structure Octree :

Un octree est un arbre qui possède 8 fils, C'est une méthode de subdivision de l'espace, plutôt simple à mettre en œuvre (surtout si on la compare à d'autres structures).

On utilise ce genre de structure pour réduire les temps de calcul, pour cela on représente une scène avec des objets contenus à l'intérieur de celle-ci, par un octree. Le découpage de celui-ci se fait de façon régulière, en ne subdivisant que là où c'est nécessaire (on a un niveau de profondeur élevé que là où il y a beaucoup d'objets). Il sera facile de déterminer quelle est la partie visible de la scène, qui nécessite des calculs pour son affichage, et quelle partie est invisible et donc ne nécessite aucun calcul.

4.2.1. Construction de l'octree

Pour calculer le rendu d'une scène représentée de cette façon, l'arbre octree est initialisé avec l'octant racine, puis récursivement subdivisé en respectant un ensemble de critères. Tout octant est subdivisé en huit "fils" si l'un des critères le permet. Ces critères sont de nature géométrique (courbure de la surface ou épaisseur locale de l'objet), physique (résultat d'un estimateur d'erreur à posteriori issu d'un calcul) ou bien directement indiqués par l'utilisateur qui peut spécifier une certaine taille minimale ou maximale de maille pour l'ensemble de l'objet.

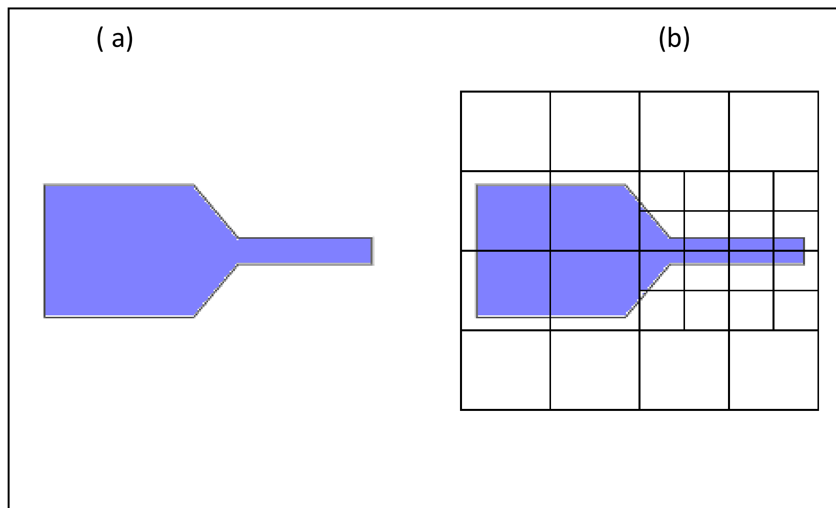


***Figure 5.2** : (a) objet à mailler, (b) construction de l'octree.*

5 Aperçu global du système

L'objectif du système que nous proposons est l'accélération du rendu volumique en introduisant une approche basée sur le traitement des goulets d'étranglement rencontré au niveau du pipeline graphique. Comme nous savons, ces goulets servent à générer des points de ralentissement du processus de rendu et donc détruisent la notion de *temps réel*. Pour parvenir à ce but, nous allons tout d'abord préciser les parties du pipeline où se produisent souvent les goulets d'étranglement puis déterminer la structure de données adaptée pour les traiter. Les briques traitent le goulet d'étranglement résultant du bus et de la taille du cache de la texture. La deuxième structure de données utilisée est l'octree au sein de chaque brique. L'octree traite le goulet d'étranglement de rastérisation.

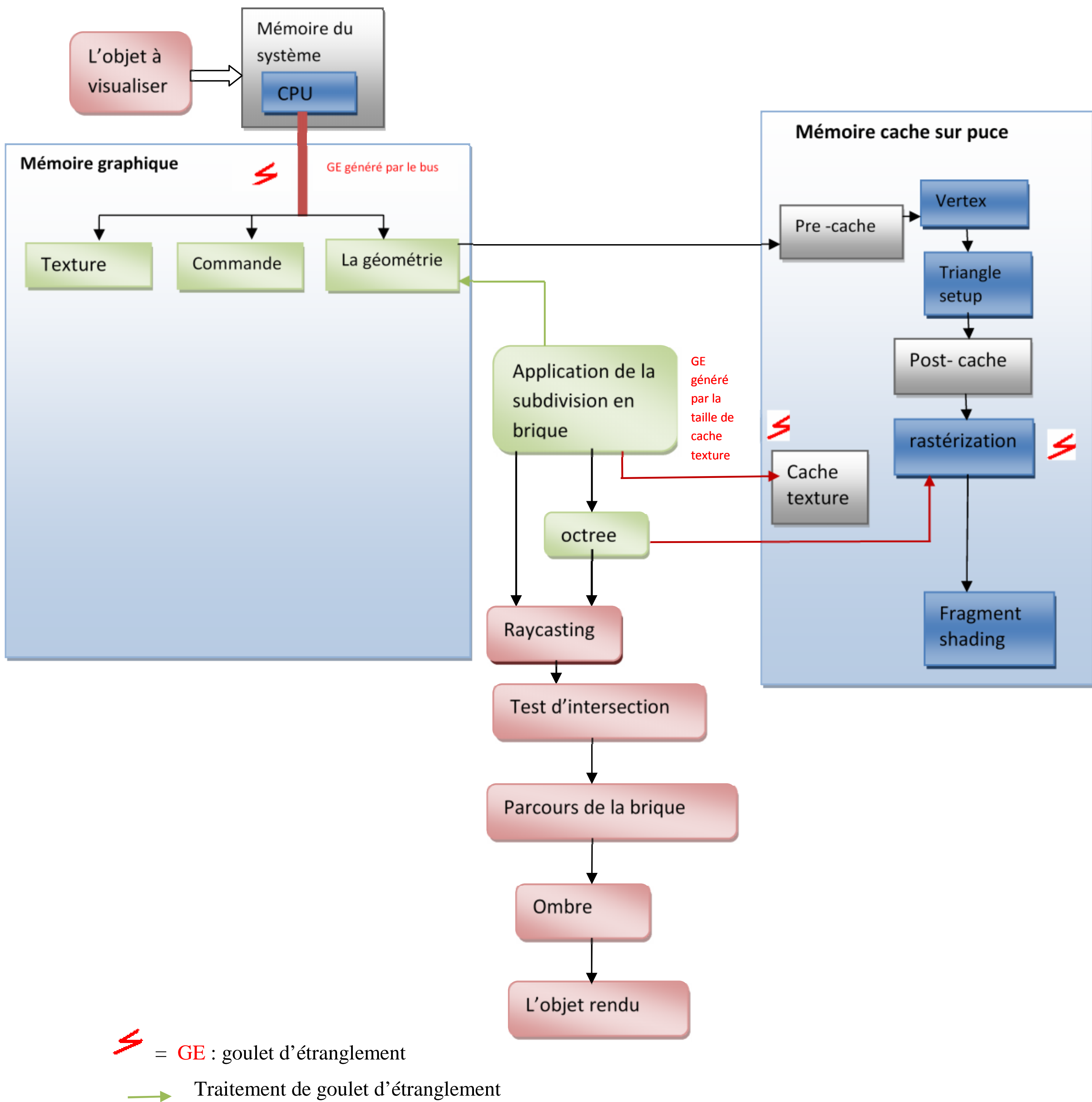


Figure 5.3 *: vue globale du système*

Le premier problème à résoudre est de déterminer la taille optimale de brique vis-à-vis la profondeur optimale de l'octree afin de bénéficier de la meilleure façon possible des profits offerts par le GPU.

Le deuxième problème est de décider quel algorithme de rendu nous allons effectuer, tout en garantissant un temps de rendu acceptable et une qualité d'image comparable à la réalité. Pour atteindre ces contraintes, le ray casting est la technique la plus populaire pour la visualisation 3-D de données volumiques.

La figure 5.3 expose un aperçu global de notre programme. Les boites bleues illustrent les unités de traitement, les grises : les unités de mémoire, les boites en vert indique la structure de données.

Après avoir introduit la géométrie de l'objet à visualisé dans la mémoire, nous appliquons une subdivision en brique régulière, cette dernière permettant de traiter le goulet d'étranglement résultant du bus et de la taille du cache de la texture. Puis nous agissons vers une subdivision plus fine au sein de chaque brique ; il s'agit de l'octree qui s'occupe de traiter le goulet d'étranglement produit par la rastérisation. Pour visualiser l'objet nous avons effectué un ray casting amélioré par la technique du near neighbor comme algorithme de rendu, vu son efficacité à atteindre un taux d'interactivité acceptable et aussi le fait qu'il peut être implémenté sur GPU pour aboutir à l'accélération du rendu qui est le but de notre travail.

6. Description des principaux algorithmes utilisés.

Maintenant nous allons présenter les principaux algorithmes développés au niveau de notre programme.

- Le principe de maillage en briques est de diviser l'espace en un nombre fini de blocs réguliers bien choisis, en ne gardant parmi elles que celles qui rencontrent la surface de l'objet. On fait l'hypothèse qu'à l'échelle de chaque brique, la surface de l'objet est à peu près plane, ce qui permet à partir de quelques points échantillonnés de créer un polygone approximant la surface dans la brique considérée.

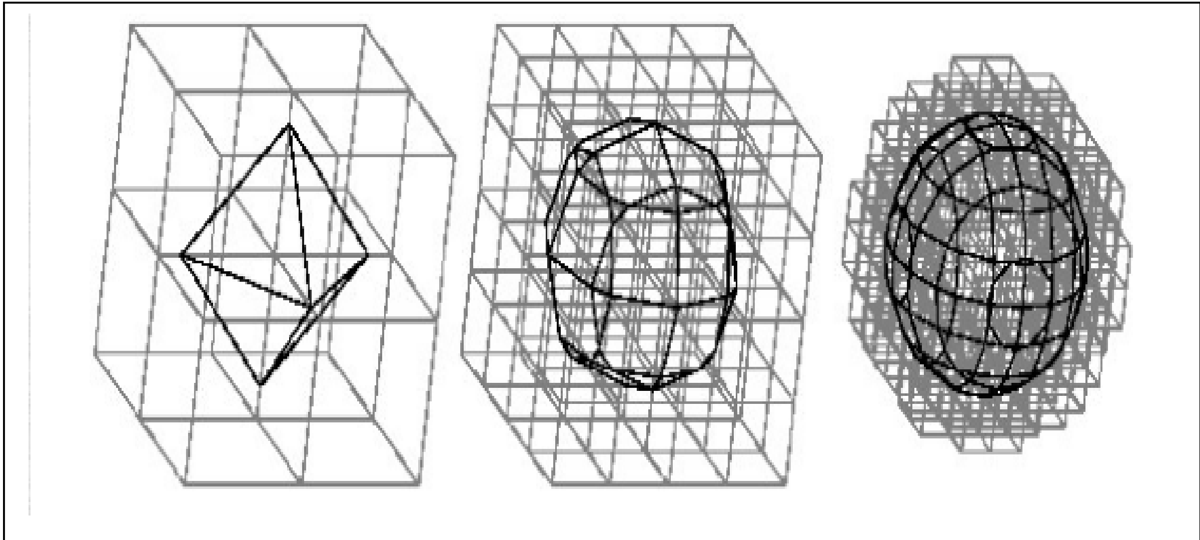


***Figure 5.4**: La taille de brique détermine la finesse du maillage.*

- Ensuite L'ensemble de ces blocs peut être structuré en un arbre hiérarchique tridimensionnel : l'**octree**. Cette structure permet de définir une subdivision adaptive de l'espace, i.e une subdivision plus ou moins fine selon la zone dans laquelle on se trouve, on obtient alors un maillage raffiné des zones comportant des détails, plus grossier des zones planes.

6.1 Algorithme de création des briques régulières :

Une approche simple peut être utilisée afin de créer les briques régulières.

Pour tous les triangles Ti dans la scène :

1. Calculer les cellules frontières (f1, f2) du triangle Ti
2. Tester l'intersection_boite du triangle : Ti avec tout cellule Cj ϵ(f1, f2)
3. Si l'intersection_boite du triangle retourne vrai, ajouter une référence de Ti à Cj.

6.2. Arbre octal: algorithme de découpage

```
Insérer (point, octree)
point : point à insérer
      octree : Case d'insertion
             {
               Si le point est dans la cellule
               {
                 Si la cellule est une feuille de l'arbre (pas de fils)
                    Insérer le point dans la case :
                 sinon
                    pour tous les fils Insérer (point, fils)
                 }
              }
```

L'appel initial insère l'objet au niveau de la racine de l'arbre.

6.3 Sauvegarde et lecture d'un octree

structure d'un noeud :

```
structure  octree
        {
          int nbFaceContenu;//nombre de face contenues dans le noeud
           int *indiceFace;      //indice des faces contenues dans le noeud
         octree *fils[8];     //fils potentiels . Si fils[0] == NULL alors pas de fils
        }
```

sauvegarde d'un noeud et de ses sous noeud :

```
procedure  sauveNoeud( octree *noeud,FILE *out)
    {
      //sauvegarde des infos du noeud : nombre de face et liste des indices
      fwrite(&noeud->nbFaceContenu,sizeof(int),1,out);
      fwrite(noeud->indiceFace,sizeof(int),noeud->nbFaceContenu,out);

     //on regarde si il y a un fils
      unsigned char aFils = 0;
      if (noeud->fils[0] != NULL)
      aFils = 1;
         fwrite(&aFils,sizeof(unsigned char),1,out);
      if (aFils==1)
       {
          for (int i=0;i<8;i++)
             sauveNoeud(noeud->fils[i],out);
        }
     }
   procédure litNoeud( octree *noeud,FILE *in)
    {
      //lecture du nombre de face / indice de face contenues dans le noeud
        fread(&noeud->nbFaceContenu,sizeof(int),1,in);
        noeud->indiceFace = new int[noeud->nbFaceContenu];
        fwrite(noeud->indiceFace,sizeof(int),noeud->nbFaceContenu,in);
      //on regarde si il y a un fils
       unsigned char aFils
        fread(&aFils,sizeof(unsigned char),1,in);
           if (aFils==1)
           {
               for (int i=0;i<8;i++)
                {
                    noeud->fils = new octree;
                        litNoeud(noeud->fils[i],in);
                }
           }
       }
```

L'Octree permet de compresser les zones totalement vides ou constantes, tout en subdivisant les zones détaillées. De plus, cette structure est totalement dynamique et gérée par un mécanisme de cache qui permet de manipuler des volumes de voxels de taille virtuellement infinie. Ce cache charge les données à la demande sur le GPU, et recycle les emplacements mémoires les plus anciennement utilisés. Les requètes de chargement/subdivision sont émises directement par les rayons lors du rendu par ray-casting, ce qui permet de ne charger que le minimum de données réellement visibles et non occultées.
Tous ces mécanismes sont implémentés et exécutés entièrement sur GPU (en CUDA), laissant ainsi le CPU totalement libre pour tout autre calcul.

6.4 Algorithme de rendu : le ray casting :

Nous allons essayer d'expliquer comment implémenter un raycasting sur GPU, en utilisant Open GL et Cg Nvidia.
Tout d'abord pourquoi nous avons besoin de cet algorithme? Parce que c'est une manière efficace d'atteindre un niveau élevé de qualité de rendu volumique et l'algorithme raycasting est bien adaptée pour le GPU modernes.
Le noyau principal de l'algorithme est d'envoyer un rayon par pixel et lancer ce rayon à travers le volume. Cela est possible à mettre en œuvre dans un fragment shader et le rendu peut être effectué en temps réel. La technique est assez souple pour par exemple un effet comme l'ombre et peut être mis en œuvre par quelques lignes de code.
Afin de générer les rayons nécessaires, nous utilisons les capacités d'OpenGl pour rendre la géométrie. Nous définissons d'abord un rayon:

- Un rayon est simplement un point d'origine *o* et un vecteur de direction *dir*
- Un rayon décrit une ligne dans l'espace 3D en utilisant la formule $P(t) = o + t * dir$

Donc, pour générer un rayon, nous devons trouver le point d'origine et le vecteur de direction.

Vu qu'il n'est pas possible de mettre en œuvre les fonctions récursives sur les cartes graphiques actuelles, nous avons été contraints de mettre au point un algorithme de ray casting itérative. Notre algorithme est un algorithme multi-passe qui exécute chaque programme shader à plusieurs reprises. Comme le nombre d'instructions qui peuvent être exécutées dans un programme shader est limité, nous avons divisé l'algorithme en trois programmes distincts, les calculs d'intersection, le parcours de la brique et l'ombrage.

6.4.1 Algorithme de Tests d'intersection :

Pour tout rayon lancé
Si intersection avec les triangles dans le voxel courant **alors**
Stocker le pointeur de triangle et les informations sur le point d'intersection dans une texture
Si non
Suivre le rayon jusqu'à intersection
Fin pour

6.4.2 Algorithme du parcours de la grille :

début
Pour un rayon courant
Si intersection avec voxel courant retourne faux alors
Déplacer au prochain voxel
Si intersection retourne vrai **alors**
Arrêter le parcours et créer un nouveau rayon avec le voxel courant comme point de départ
Répéter la procédure
fin

6.4.3 Algorithme Ombres :

initialisation
couleur_texture, La direction_ rayons et position_départ
Si une intersection a été trouvée **Alors**
Ombrer le pixel en fonction du point d'impact, la position de la lumière et la matière.
La couleur calculée est ensuite mélangé avec la couleur calculée dans les étapes précédentes
Ecrire la couleur du pixel dans une mémoire tampon
Remplacer couleur_texture précédente par la couleur du pixel
Si le triangle touché n'est pas diffus et la de réflexion maximale n'a pas été atteinte **Alors**
un nouveau rayon est calculé
écraser direction_ rayons antérieur et position_départ de rayon.
fin

Ce qui suit est le ray casting shader :

```
struct app_vertex
```

Ce qui suit est le ray casting shader:

```
Struct app_vertex
{
   float4 Position     : POSITION;
   float4 TexCoord     : TEXCOORD1;
   float4 Color       : COLOR0;

};

// Définir l'interface entre le vertex shader et le fragment shader

struct vertex_fragment
 {
   float4 Position       : POSITION; // For the rasterizer
   float4 TexCoord       : TEXCOORD0;
   float4 Color           : TEXCOORD1;
   float4 Pos             : TEXCOORD2;
};

struct fragment_out
{
  float4 Color     : COLOR0;
};

// Implémentation du vertex shader du Ray casting

vertex_fragment vertex_main( app_vertex IN )
{
  vertex_fragment OUT;

  // Acquérir les états des matrices OpenGL

  float4x4 ModelView = glstate.matrix.modelview[0];
  float4x4 ModelViewProj = glstate.matrix.mvp;

  // Transformation des vertex
  OUT.Position = mul( ModelViewProj, IN.Position );
  OUT.Pos =  mul( ModelViewProj, IN.Position );
  OUT.TexCoord = IN.TexCoord;
  OUT.Color = IN.Color;
  return OUT;
}

// implementation du fragment shader du Raycasting
fragment_out fragment_main( vertex_fragment IN,
                           uniform sampler2D tex,
                            uniform sampler3D volume_tex,
                           uniform float stepsize
                          )

{
  fragment_out OUT;
  float2 texc=((IN.Pos.xy/IN.Pos.w)+1)/2;//trouver la position correcte
                                        // pour chercher dans le tampon
  float4 start = IN.TexCoord; // la position début de rayon est sauvgaredé
                            //dans les coordonnées texture
```

```
  float4 back_position  = tex2D(tex, texc);
  float3 dir = float3(0,0,0);
  dir.x = back_position.x - start.x;
  dir.y = back_position.y - start.y;
  dir.z = back_position.z - start.z;
  float len = length(dir.xyz); // la longueur d'avant à l'arrière est
                               //calculé est utilize pour terminer le rayon
  float3 norm_dir = normalize(dir);
  float delta = stepsize;
  float3 delta_dir = norm_dir * delta;
  float delta_dir_len = length(delta_dir);
  float3 vec = start;
  float4 col_acc = float4(0,0,0,0);
  float alpha_acc = 0;
  float length_acc = 0;
  float4 color_sample;
  float alpha_sample;

  for(int i = 0; i < 450; i++)
    {
      color_sample = tex3D(volume_tex,vec);
      alpha_sample = color_sample.a * stepsize;
      col_acc   += (1.0 - alpha_acc) * color_sample * alpha_sample * 3;
      alpha_acc += alpha_sample;
      vec += delta_dir;
      length_acc += delta_dir_len;
      if(length_acc >= len || alpha_acc > 1.0)
      break; // terminer si l'opacité > 1 ou le volume est dehors le volume
    }
   OUT.Color =  col_acc;
  return OUT;
}
```

7. Résultats et discussions

Dans cette section, nous commencerons par présenter les différents résultats en termes de performances obtenus au cours de ce mémoire et nous terminerons par une discussion sur ces travaux.

7.1. Choix de l'environnement de travail

C# est un environnement de programmation simple complet, Il est fourni avec l'environnement de développement intégré Visual Studio. Ce langage permet d'intégrer une bibliothèque complète de classes ainsi qu'un moteur d'exécution appelé la Common Language Infrastructure (CLI).

C# est un langage de programmation conçu pour la création d'une vaste gamme d'applications qui s'exécutent sur le .NET Framework.

C# est simple, puissant, de type sécurisé et orienté objet. Avec ses nombreuses innovations, C# permet le développement rapide d'applications tout en conservant la simplicité et l'élégance des langages de style C.

7.2 Les résultats expérimentaux :

Pour évaluer la performance de notre modèle de briking+octree, nous avons utilisé des différentes scènes complexes et le moteur de rendu utilisé dans ce cas est le ray casting. Nous avons utilisé NVIDIA GeForce 7300 SE/7200 GS avec un pilote CUDA et un processeur Intel Core 2 duo.

7.3 Influence des paramètres de la structure sur le rendu

La scène utilisée ici est une statue (figure 5.5 c) de 100000 facettes occupant 40% du volume total de la scène. Les paramètres sont notés ***TB***: taille de brique, ***PO*** : profondeur de l'octree.

a. Trouver la taille optimale des briques

Comme nous avons dit ultérieurement, la taille de brique est principalement en fonction des propriétés de la mémoire de texture (chapitre 4) et la profondeur optimale de l'octree est principalement déterminée pour équilibrer la charge de rastérisation et le débit de triangle, nous allons étudier l'influence de chaque paramètre séparément.

Pour analyser l'impact de la taille des briques sur le rendu et trouver la taille optimale, nous devons garder la profondeur de l'octree constante. Dans notre cas elle est fixé à 8^3 . Les figures ci-dessous (figure 5.5) montrent des résultats de notre programme, nous faisons varier à chaque fois la taille des briques, pour déterminer quelle est la taille optimale à choisir.

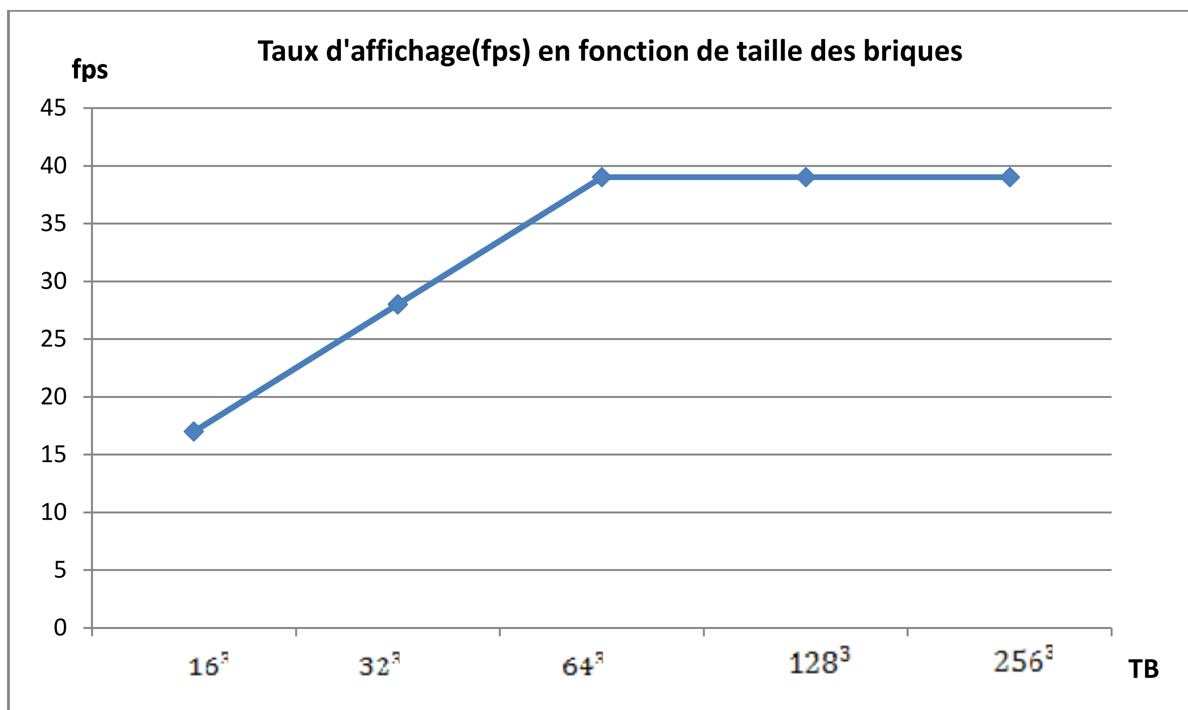
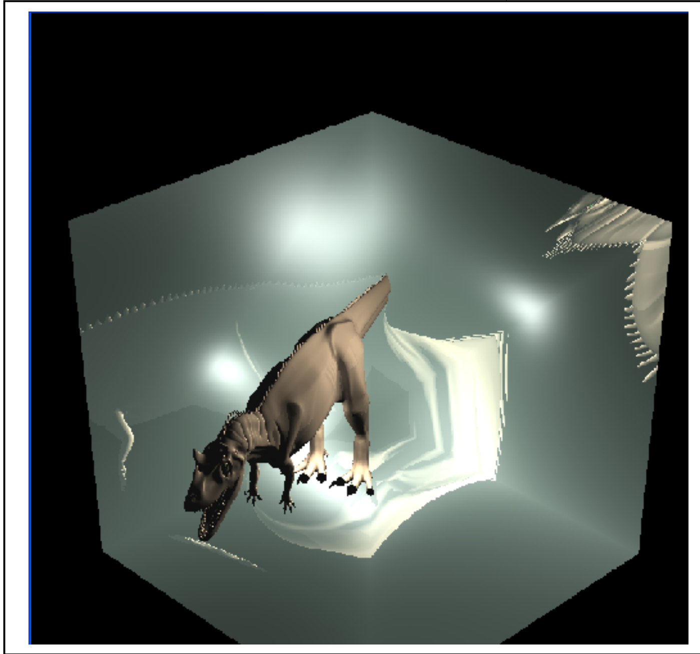


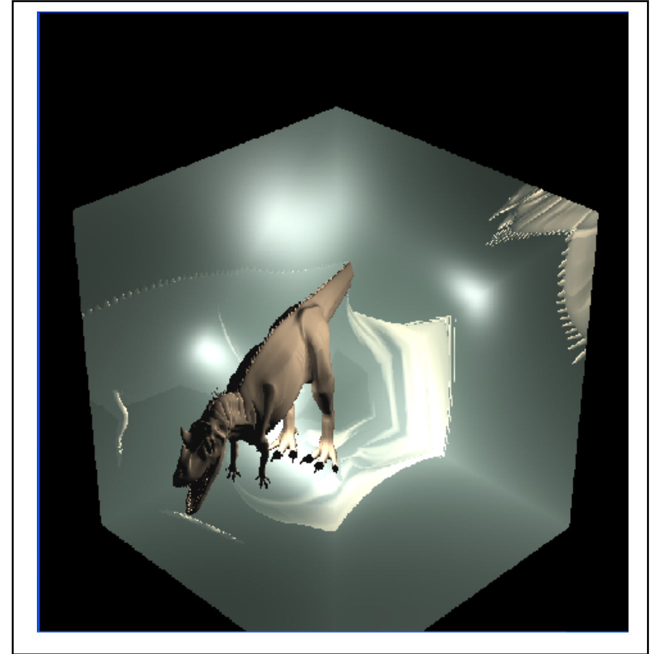
Figure 5.6 : *Digramme fps en fonction de taille de brique*

Le diagramme illustré dans la figure 5.6 montre clairement que le taux d'affichage arrive à son maximum et se stabilise en 64 et que si on augmente la taille d'une brique à 128 ou 256, il reste pratiquement constant.

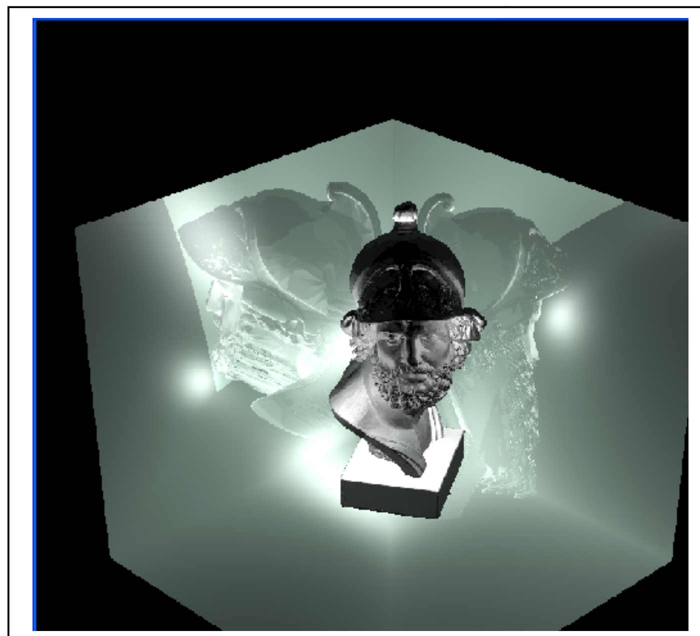
Cela confirme qu'une taille optimale de brique est 64 avec un taux d'affichage moyen : 45fps.



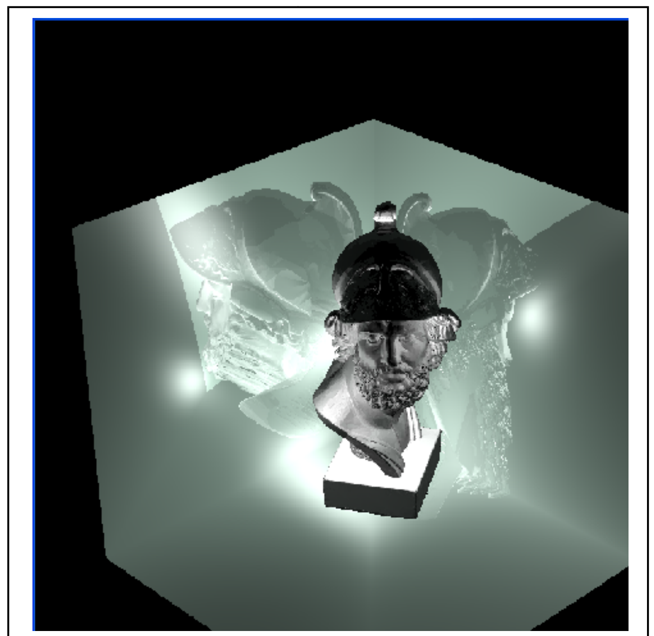
(a) : TB=16 , fps= 17



(b) : TB=32, fps=28



(c) : TB=64 , fps= 45



(d): TB=256, fps=45

Figure 5.5 *: scènes rendues avec taille brique (TB)*

Le tableau suivant résume les informations liées aux images précédentes :

Image	Fichier source	Nbr facettes	TB	fps	Temps de rendu
(a)	Dinosaur.3ds	40000	16	17	120s
(b)			32	28	100s
(c)	Tête.3ds	100000	64	45	180s
(d)			256	45	120s

Tableau 5.1 *: Variation de temps de rendu suivant la taille de brique*

b. Trouver la profondeur optimale de l'octree :

Pour étudier l'influence de la profondeur de l'octree, le volume a été rendu avec une taille fixe de briques de 64^3 voxels et une profondeur variable d'octree. Une profondeur minimale surcharge le CPU et de nombre de triangles, et un octree plus large conduit à surcharger la rastérisation.

Les figures ci-après présentent quelque scène rendu en utilisant des profondeurs variés d'octree.

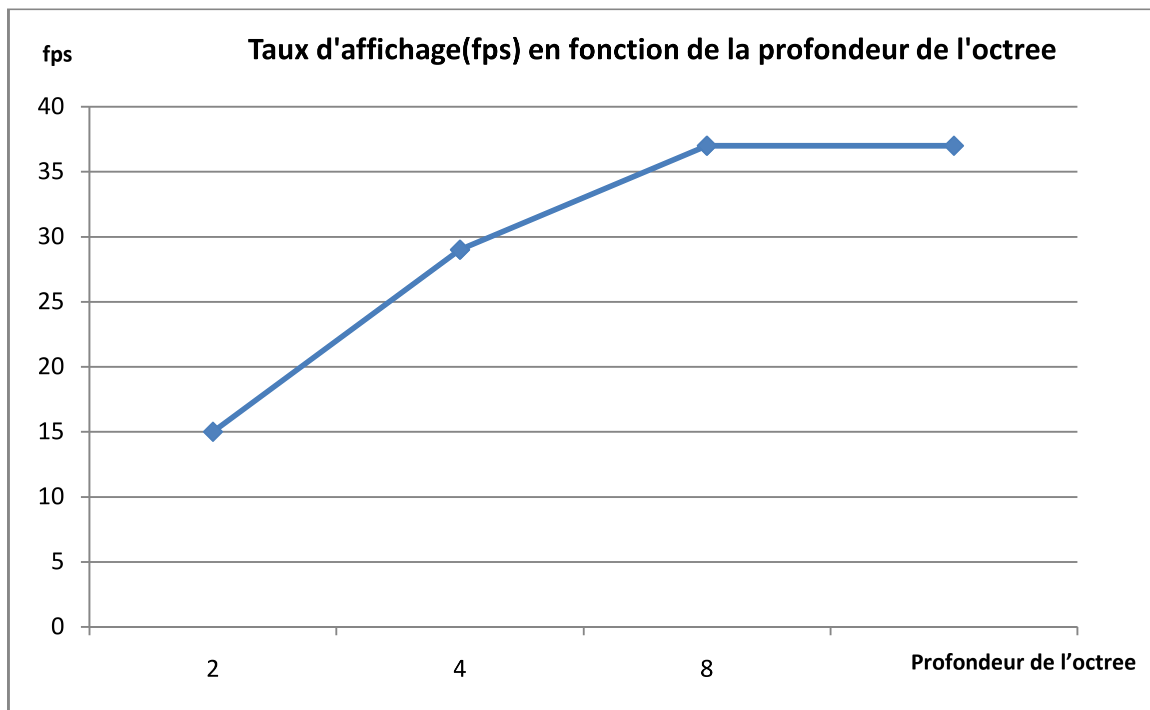
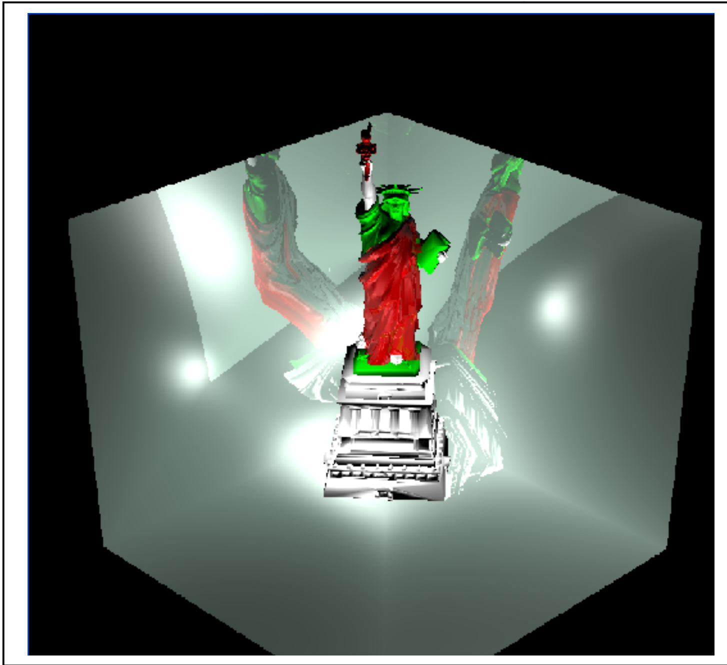
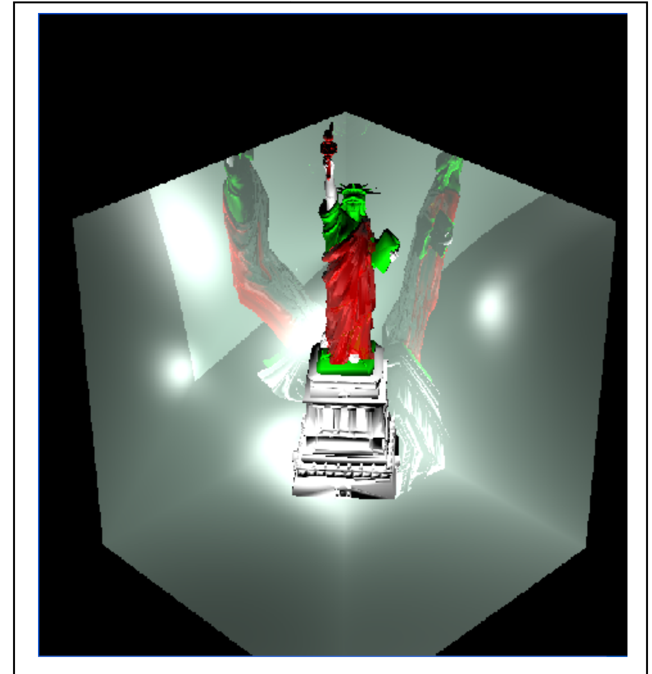


Figure 5.8: *Digramme fps en fonction de la profondeur de l'octree*

Le diagramme de la figure montre l'influence de la profondeur sur le taux d'affichage, nous allons adopter à une même scène (un statue voir la figure 5.8); en fixant la taille de brique à 64 et en variant à chaque fois la profondeur, nous avons remarqué que la profondeur a un impact sur le taux d'affichage en proportion c.-à-d. quand nous augmentons la profondeur le taux croît, jusqu'à arriver une profondeur de 8. A ce moment, le taux d'affichage reste constant.

Pour cela nous avons choisir la profondeur 8 comme une profondeur optimale de rendu.





(e) : PO=2, fps=15 **(f) : PO= 4, fps= 39**

(g) : PO=8, fps=40 **(h) : PO=16, fps= 40**

***Figure 5.7 :** scènes rendues avec des profondeurs d'octree variées*

Le tableau suivant résume les informations liées aux images précédentes :

image	Fichier source	Nbr facettes	PO	fps	Temps de rendu
(e)	USA.3ds	25000	2	15	10s
(f)			4	39	40s
(g)	Tyrannosaurus.3ds	60000	8	40	30s
(h)			16	40	100s

***Tableau 5.2 :** Variation de temps de rendu suivant la profondeur de l'octree*

8. Conclusion :

Le rendu 3D temps réel vit une période particulièrement faste. Avec l'avènement des GPU de plus en plus programmables, de nombreuses approches jusqu'ici inenvisageables ressurgissent pour tenter de résoudre les problèmes auxquels nous sommes confrontés.

Dans ce chapitre, nous avons décrit notre approche basée sur une subdivision de la scène en brique, puis utilisation l'octree au niveau de chaque brique. La motivation d'agir vers ces deux structure réside dans leurs capacités à traiter les goulets d'étranglement qui se produisent fréquemment au niveau du pipeline. Nous avons également introduit les différents algorithmes utilisés au sein de notre application.

Pour effectuer le rendu volumique, le raycasting est une technique particulièrement prometteuse pour augmenter de façon drastique la complexité géométrique des scènes à visualiser.

Enfin, nous avons montré quelques résultats illustrant l'efficacité de notre système.

Ces résultats étudient l'influence des paramètres de structure (taille de brique ***TB***, profondeur de l'octree *PO*) sur la vitesse de rendu exprimé en fps, afin d'aboutir à une combinaison optimale ***TB/PO***.

Annexe

L'environnement CUDA

1. Introduction et définition:

CUDA est un acronyme de Compute Unified Device Architecture, c'est une architecture parallèle de calcul développée par NVIDIA.
Dans l'industrie des jeux sur ordinateur, en plus de rendu volumique, des cartes graphiques sont utilisés dans les calculs de la physique des jeux (effets physiques tels que des débris, de fumée, fer, fluides). CUDA est utilisé en plus pour accélérer des applications non graphiques dans la biologie, la cryptographie et d'autres domaines. **[FD09]**
CUDA est l'engin de calcul dans les unités de processeur graphique (GPU) NVIDIA, qui est accessible au programmeur via un langage de programmation standard. Les programmeurs utilisent C pour CUDA (C avec les extensions NVIDIA). L'architecture CUDA supporte quelques API comme OpenGL et DirectX Compute. Les derniers pilotes contiennent tous les éléments nécessaires CUDA.
Les derniers GPU NVIDIA deviennent effectivement des architectures ouvertes comme les processeurs. Cependant, à l'inverse des CPU, les GPU ont une architecture parallèle " multi core", chaque cœur est capable d'exécuter des milliers de threads simultanément.
Si une application est adaptée à ce genre d'architecture, le GPU peut offrir une performance importante.

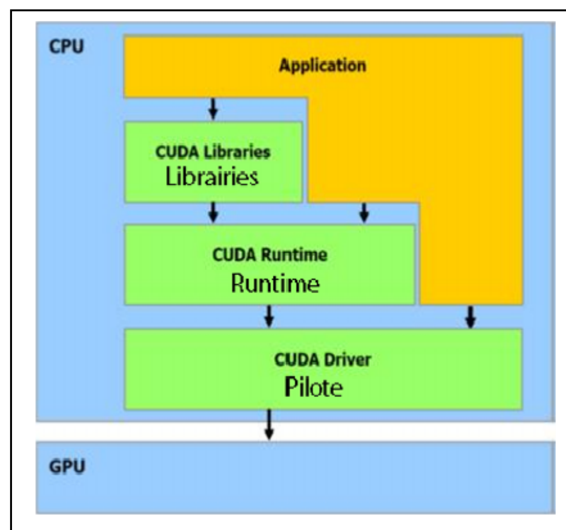
CUDA est la réponse de NVIDIA aux demandes sans cesse croissantes de puissance de calcul. Cette librairie permet d'employer la puissance de calcul des GPU. Elle n'est que la partie logicielle du tout : il faut encore une carte graphique compatible.
CUDA supporte plusieurs langages : le C, le C++ et le Fortran. On peut donc utiliser conjointement ces trois langages dans les fonctions et les kernels.
CUDA est constitué d'un pilote, déjà intégré aux matériels les plus récents ; d'un runtime ; et de quelques librairies. **[TC09]**

CUDA est aussi un langage, dérivé du C (mais n'apportant que peu de modifications : 9 nouveaux mots clés, 24 nouveaux types et 62 nouvelles fonctions). Ces extensions nécessitent leur compilateur, lui aussi fourni.

CUDA est prévu pour s'exécuter sur un GPU, mais il est aussi disponible sur CPU, en émulation.

L'API CUDA est de haut niveau : on ne s'occupe donc pas du GPU directement. CUDA en est une couche d'abstraction. **[TC09]**

Voici, graphiquement représenté, toutes les composantes de CUDA et de son utilisation.



1. **Pilote** :

• Rôle : transmettre les calculs de l'application au GPU ;

• Distribution avec les ForceWare les plus récents ;

• Inconvénient : pas d'automatismes.

2. **Runtime**

• Rôle : interface entre le GPU et l'application, en fournissant quelques automatismes ;

• Distribution : en même temps que le pilote ;

• Inconvénient : impossibilité d'optimiser à partir d'un certain point. **[TC09]**

3. **Bibliothèques**

Pour le moment, CUDA est livré avec CuBLAS (Basic Linear Algebra Subprograms) et CuFFt (Fast Fourier Transform library), respectivement les implémentations de BLAS (une bibliothèque d'algèbre) et de la transformation rapide de Fourier (utilisée en analyse de Fourier et en traitement du signal). La dernière n'est pas inspirée d'une bibliothèque préexistante.

Ces implémentations reprennent le fonctionnement des bibliothèques originelles (CuBLAS), ou bien des algorithmes les plus performants (CuFFT) et les optimisent au maximum pour CUDA.

2. CUDA sur CPU et GPU :

2.1 Quelques différences :

La puissance des GPUs n'a de cesse d'augmenter depuis quelques années. À un point qu'il est désormais possible de les utiliser pour réaliser des calculs autres que pour des jeux.
Cependant, ces différences énormes s'expliquent très facilement dans le tableau suivant :

	CPU (hors SIMD)	GPU
Nombre de tâches	Une seule et unique	Le plus grand nombre
Variété des tâches	Toutes possibles	Restreinte
Subdivision de la tâche	Aucune : tout en un coup	Maximale, pour mieux la répartir sur les différentes unités de calcul

Il ne faut pas oublier de préciser que les GPU préfèrent travailler avec des vecteurs. Dans le cas contraire, les gains sont réellement minimes.
Les deux types de processeur travaillent de façon radicalement différente. L'emploi de GPU à la place de CPU ne se fait donc pas en un tour de main : il faut repenser le calcul pour l'adapter au type de processeurs désiré.

2.2. Précision des calculs :

Les GPU actuels, avec CUDA, n'ont qu'une précision FP32, sur 32 bits. Il faut se tourner vers les solutions d'ATI/AMD pour une précision double sur 64 bits, ou bien vers des GPU plus chers, ou récents.
Tous les processeurs ne fonctionnent pas à la même précision : sur les premières GeForce compatibles CUDA, tous sont FP32. Le peu d'unités dédiées au calcul à double précision sur les Tesla et autres explique leur faible puissance à ce niveau de précision, en comparaison de la simple précision ou bien des solutions d'AMD. Ainsi, pour du calcul en haute précision, les solutions NVIDIA tous publics ne sont pas encore au point.

2.3 Shaders :

Les calculs demandés à CUDA sont, pour le moment, effectués sur les unités de shaders, les processeurs les plus rapides sur les GPU. Par exemple, les GeForce 8800 GTX ont des unités cadencées à 1,2 GHz.

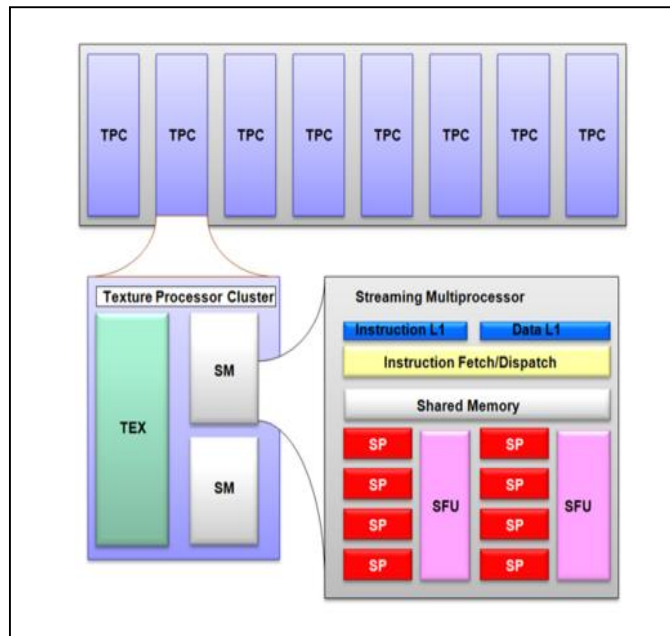


***Figure**: l'unité de traitement des shaders*

Chaque unité de traitement des shaders est, comme montré ci-dessus, constituée de Texture Processor Clusters (TPC).

Chacun de ces clusters est fait d'une unité de traitement des textures (TEX) et de deux unités de traitement des flux (SM, Streaming Multiprocessor).

Vous n'avez pas vraiment besoin d'en savoir beaucoup plus pour pouvoir aborder CUDA.

3. Le modèle de programmation :

Comme vu précédemment, le GPU est un processeur extrairement multithreadé avec sa propre mémoire. L'API C pour CUDA est une extension du C qui permet de s'adapter à cette architecture pour tirer profit de la puissance de calcul disponible.

Dans un programme, chaque phase de calcul massivement parallèle sur le GPU suit le schéma global suivant :

1. Copie des données nécessaires au calcul dans la mémoire du GPU
2. Exécution du même code N fois par N threads GPU en parallèle
3. Recopie des résultats depuis la mémoire du GPU dans la mémoire centrale.

3.1 Parallélisme des données :

Les applications actuelles qui doivent traiter de grandes quantités de données prennent beaucoup de temps à l'exécution. Ce temps pourrait être réduit en parallélisant les opérations : des phénomènes physiques peuvent être calculés indépendamment les uns des autres, des images à analyser peuvent être découpées en portions, et un flux vidéo peut être découpé image par image.

La parallélisation des données sert à la capacité du programme de gérer parallèlement et indépendamment ces instructions arithmétiques.

Par exemple, pour des multiplications de matrices de taille 1000 x 1000, il s'agit de 1.000.000 de multiplications, sans rapport les unes avec les autres, qui peuvent donc être parallélisées sans problème. Un GPU peut fortement améliorer les performances en exécutant toutes ces opérations simultanément.

3.2 Structure du programme :

Un programme CUDA est constitué d'une partie qui s'exécute sur l'hôte et d'une partie qui s'exécute sur le périphérique.

Les phases peu ou pas parallèles sont exécutées sur l'hôte.

Les phases massivement parallèles sont exécutées sur le périphérique.

Le programme peut tenir en un seul fichier, comprenant ces deux phases et les environnements. Le compilateur se charge de les séparer : le code pour l'hôte est du standard C ANSI/ISO, il est compilé par le compilateur principal du système, et sera lancé comme un simple processus. Le code pour le périphérique est aussi écrit en C ANSI/ISO, avec quelques extensions CUDA, mais il est compilé par NVCC, et sera exécuté sur le périphérique.

Les kernels génèrent généralement beaucoup de threads pour exploiter au mieux le parallélisme des données.

Dans notre exemple de produit matriciel, il y a autant de threads que de cellules dans la matrice résultante. Chacun de ces threads prend, généralement, très peu de cycles, vu le peu de tâches qui leur sont demandées.

L'exécution commence avec le CPU, qui prépare l'appel au kernel. Le GPU prend le relais pour le kernel, qui sera, lui, massivement multithread. Quand le kernel a fini sa tâche, il renvoie le résultat au CPU, et son exécution continue.

3.3 Exemple : la multiplication de matrices carrées :

Pour commencer par clarifier la situation, voici le fonctionnement que décrira notre programme.

1. **CPU** : Initialisation des matrices M, N et P, toutes carrées ;
2. **CPU** : Remplissage des matrices d'entrée M et N ;
3. **GPU** : Calcul du produit matriciel de M et de N, dont le résultat est stocké sur P ;
4. **CPU** : Écriture de la matrice P ;

5. **CPU** : Nettoyage de la mémoire et fin de l'exécution du programme.

Le but étant de montrer le fonctionnement d'un programme CUDA et non d'optimiser au maximum une application.

Ces fonctions se trouvent, heureusement, dans l'API CUDA. Leurs noms sont très recherchés : cudaMalloc() et cudaFree().

Voici un bref exemple d'utilisation de ces deux fonctions, très proche des fonctions malloc() et free() du C. On considère que *Width* est le nombre de ligne et de colonne de la matrice pour laquelle on crée l'espace mémoire.

```
float *Md;
float *Nd;
float *Pd;
const int size = Width * Width * sizeof(float);
cudaMalloc( (void**) & Md, size);
cudaMalloc( (void**) & Nd, size);
cudaMalloc( (void**) & Pd, size);
cudaFree            ( Md );
cudaFree            ( Nd );
cudaFree            ( Pd );
```

cudaMalloc() prend deux paramètres, pour définir la mémoire à allouer en mémoire globale.

1. L'adresse d'un pointeur vers la mémoire allouée,
2. La taille de la mémoire à allouer.

cudaFree() ne prend qu'un paramètre, pour désallouer cette mémoire en mémoire globale.

Un pointeur vers la mémoire à désallouer.

Une fois que le programme a alloué sa mémoire, il peut demander les données des matrices à stocker en mémoire.

Ceci s'obtient avec une fonction de copie de mémoire : cudaMemcpy(). Cette fonction requiert 4 paramètres.

1. Un pointeur vers les données source à copier,
2. La destination des données,
3. Le nombre d'octets à copier,
4. Le type de mémoire vers laquelle copier.

Concernant le quatrième paramètre, il peut prendre une de ces valeurs.

- cudaMemcpyHostToDevice : copie de l'hôte vers le périphérique
- cudaMemcpyHostToHost : copie de l'hôte vers l'hôte
- cudaMemcpyDeviceToHost : copie du périphérique vers l'hôte
- cudaMemcpyDeviceToDevice : copie du périphérique vers le périphérique.
-

Voici les appels réalisés pour copier les matrices sur lesquelles nous allons travailler, et pour envoyer le résultat à l'endroit souhaité. *M*, *N*, *P*, *Md*, *Nd*, *Pd* et *size* gardent leurs valeurs précédentes.

```
cudaMemcpy(Md, M, size,
cudaMemcpyHostToDevice);
cudaMemcpy(Nd, N, size,
cudaMemcpyHostToDevice);
cudaMemcpy(P, Pd, size,
cudaMemcpyDeviceToHost);
```

Conclusion Générale

Conclusion Générale

Dans ce mémoire, nous avons essayé de présenter une méthode de rendu volumique permettant de réduire le temps de rendu afin d'atteindre un taux d'affichage interactif, en utilisant les fonctionnalités offertes par les GPUs.

D'après l'étude réalisée au coursde la réalisation ce travail et qui concerne l'architecture du GPU et les différentes unités du pipeline graphique, nous allons localiser les points où se produise les goulets d'étranglement: le premier responsable du gaspillage du temps de calcul.

Le traitement de ces goulets conduit à accélérer le processus de rendu qui est notre objectif principal. Pour cela, nous avons introduit des structures de données adaptées à la fois au rendu volumique interactif et à la gestion de grosses masses de données volumiques. Ces structures disposent en effet de bonnes qualités pour permettre un temps de rendu peu dépendant de la quantité de données visualisées.

Les structures de données choisies doivent avoir des effets directs sur le soulagement des points de goulets d'étranglement. Ces structures sont principalement l'hybridation de la subdivision en brique et la structure d'octree.

Plusieurs raisons nous ont pousse à adopter le choix de ces deux structures :

- Les briques vides ne sont jamais dessiné, ni stockés dans la mémoire vidéo, et donc le goulet d'étranglement de bus est éliminé.
- L'octree réduit le temps que le GPU passe dans le traitement des données qui ne contribuent pas à l'image finale.
- L'octree traite le goulet d'étranglement de rastérisation.

Grâce à son adéquation d'être implémenter sur le GPU, le ray casting est considéré commele meilleur algorithme de rendu volumique vis-à-vis de la qualité d'image générée en plus de sa possibilité d'atteindre du temps d'affichage interactif.

Cependant, il est également important de noter que l'utilisation de GPUs pour effectuer les calculs d'intersection n'est pas une fin en soit et qu'elle ne doit être

envisagée que comme un outil parmi divers autres. Nos travaux futurs concernant plus particulièrement l'accélération matérielle des calculs de l'illumination globale, impliquant ainsi l'étude d'une collaboration plus poussée entre CPU et GPU, l'utilisation de plusieurs cartes 3D par processeur, la distribution des calculs sur une architecture à mémoire partagée, disposant éventuellement plusieurs GPUs. A son tour, cet aspect matériel représente un outil qu'il conviendra d'associer à d'autres techniques d'accélération logicielles, en vue d'obtenir des applications beaucoup plus rapides que celles disponibles à ce jour.

De nombreux problèmes restent ouverts comme la visualisation de volumes beaucoup plus profonds et larges, la gestion et le rendu de données dynamiques ou la prise en compte de l'illumination globale (ombres, inter-réflections etc.) est également une piste qu'il reste à explorer.

Bibliographie

[AM04]: *Alexandre MEYER,* Représentations d'arbres réalistes et efficaces pour la synthèse d'images de paysages. Thèse Phd de l'Université Joseph Fourier - décembre 2001

[AP05]: *Antoine Pericchi,* Simulation réaliste de ruisseaux en temps réel. DEA, Institut National Polytechnique de Grenoble, juillet 2004.

[BB03]: *BRADAI Benazouz,* Rendu volumique d'images 3D adapté à la génération de radiographies projectives, DEA de l'École doctorale de l'université de S avoie - Département SPI et école doctorale électronique, électrotechnique, et automatique de Lyon, 2003.

[CA02]: *Nathan A. Carr, Jesse D. Hall, John C. Hart,* The Ray Engine, Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS conference on Graphics hardware, 2002, p.37-46. ISBN 1-58113-580-7

[MD06]: *Marc DAUMAS, Guillaume Da Graça et David Defour,* Caractéristiques arithmétiques des processeurs graphiques. In Proceedings of CoRR 2006

[CC01]: *François ROUSSELLE, Christophe RENAUD,* Tracé de chemins accéléré par l'utilisation de GPUs, *Christophe Cassagnabere.* Laboratoire d'Informatique du Littoral, 2001

[CC04]: *Christophe CUNAT*, Accélération matérielle pour le rendu de scènes multimédia vidéo et 3D, Doctorat Electronique et Communications, ENST - COMELEC Communication et Electronique, Paristech > ENST.2004.

[CC07]: *Cyril CRASSIN*, Représentation et Algorithmes pour l'Exploration Interactive de Volumes Proceduraux Etendus et Détaillés, DEA de l'INPG/UJF M2R - IVR - 2007

[DF04]: *David FRADIN*, Modélisation et simulation d'éclairage à base topologique: application aux environnements architecturaux complexes, -TEL - CCSd - CNRS- 2004.

[DP04]: *Damien PORQUET,* Rendu en temps réel de scène complexe, Thèse de doctorat : Informatique. Limoges : Université de Limoges, 2004

[DR08]: *Daniel RUIJTERS, Anna Vilanova*, Optimizing GPU Volume Rendering, Journal of WSCG, Volume 14, Number 1-3, January 2006, Pilzen (Czech Republic), pp. 9-16, 2008

[EK05]: *Kristian EIDE*, Near-Neighbor Approach to GPU-based Volume Rendering, I.3.8 [Computer Graphics]: Applications 2005

[FC04]: *Florent COHEN,* Ombrage et illumination pour le rendu 3D de forêts en temps réel, Juin, Master, Institut National Polytechnique de Grenoble. Juin, 2004.

[FD09]: *Forent DAHM*, Etude pour l'accélération du code de calcul parallèle eslsA à l'aide de processeur graphique GPU, Master de mathématiques et applications, 2009

[FL03]: *Fabien LAVIGNOTTE. Mathias PAULI,* Scalable Photon Splatting for Global Illumination Proceedings of the Eurographics Workshop on Rendering Techniques 2000.

[GE00]: http://http.developer.nvidia.com/GPUGems

[GF01]: *Gabriel Fournier et Bernard Peroche*, Vers un rendu réaliste interactif, LIRIS, Laboratoire d'InfoRmatique en Image et Systèmes d'information, 2001

[JF05]: *Jean-François Dufort*, Rendu interactif de détails de surface par textures 3D semi-transparentes, Mémoire de maîtrise, 2005.

[JK03]: *J. Kruger end R.Westermann*, Acceleration Techniques for GPU-based Volume Rendering, Computer Graphics and Visualization Group, Technical University Munich 2003.

[KE05]: *Kristian Eide*, Use of GPU Functionality in Volume Rendering, Institutt for datateknikk og informasjonsvitenskap, 2005.

[KE06]: *Kristian Eide*, GPU-based Transparent Polygonal Geometry in Volume Rendering, ; I.3.8 [Computer Graphics]: Applications 2005

[KJ86]: *James T. Kajiya*, The Rendering Equation, Proceedings of the 13th annual conference on Computer graphics and interactive techniques, p. 143-150, 1986

[KP01]: *Ken PERLIN and Fabrice NEYRET,* Flow noise. In Siggraph Technical Sketches and Applications, 2001.

[MA05]: *Mathieu TOBIE*, "Construction de modèles 3D osseux à partir d'images TDM et simulation radiogaphique", École Nationale Supérieure de Physique de Strasbourg, 2005.

[MC05]: *Martin CHRISTEN,* Ray Tracing on GPU, Diploma Thesis, University of Applied Sciences Basel,2005

[MO97]: *Tomas MOLLER, Ben TRUMBORE*, Fast minimum storage ray-triangle intersection, Journal of Graphics Tools, Volume 2, Issue 1, 1997, p. 21 – 28

[MP03]: *Matt PHARR et Greg HUMPHEREYS,* phisiqually based image synthesis from theory to implementation, SIGGRAPH Pages: 848 - 855, 2003.

[MP04]: *Mathias PAULIN* , Le Rendu en Synthèse d'Images : du Réalisme au Temps Réel, HDR UPS, 11/2004.

[NB09]: *Nadir BENMOUNAH*, Rendu temps-réel d'objets translucides, Thèse de doctorat :

Informatique. Limoges : Université de Limoges, 2009

[NL05]: *Niels THRANE Lars Ole SIMONSEN*, A Comparison of Acceleration Structures for GPU Assisted Ray Tracing, Master's thesis, University of Aarhus, 2005.

[PC06]: *Pierre Y. CHATELIER*, Une approche de la radiosité par voxels, application à la synthèse d'images, Doctorat de l'Université d'Auvergne, 2006.

[PU04]: *Timothy PURCELL*, Ray tracing on a stream processor, Phd. thesis, 2004

[QZ02]: *Qi ZHANG, Roy Eaglesona, and Terry M. Petersa*, GPU-Based Real-Time Beating Heart Volume Rendering Using Dynamic, Imaging Research Laboratories, Robarts Research Institute, London, Canada N6A 5K8, 2002

[RS00]: *C. Rezk-SALAMA, K. Engel, M. Bauer, G. Greiner, and T. Ertl;* Interactive volume on tandardpc graphics hardware using multi-textures and multi-stage rasterization, GH 1999.

[SA98]: *S. ARYA, D. M. Mount, N. S. Netanyahu, R. Silverman,* An optimal algorithm for approximate nearest neighbor. Journal of the ACM 45, 891–923. WU1998

[SD06]: *S. DUMAZ et V. Biri,* Photon mapping et radiosité à base de fonctions, Université de Marne-la-Vallée Équipe SISAR, 2006

[SL06]: *Sylvain LEFEBVRE,* Modèles d'habillage de surface pour la synthèse d'images, Thèse de doctorat, Université Joseph Fourier. avril 2005.

[TC09]: *Thibaut CUVELIER*, Une introduction à CUDA, 2009.

[VV01]: *M. Vincent VIDAL*, Étude et développement d'un système multi-échelle pour la visualisation réaliste et interactive de végétaux (rendu volumique), Laboratoires : INRIA Rhône-Alpes. Équipe EVASION, 2007