REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohamed Khider - BISKRA

Faculté des Sciences Exactes et 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: Data Mining et Multimédia

Titre:

Utilisation des GPUs pour la reconstruction 3D en imagerie médicale

Présenté par: Djaafar Lakhdar

Devant le jury:

NourEddine DJEDI	Université de Biskra	PR	**Président**
Med Chaouki BABAHENINI	Université de Biskra	MCA	**Rapporteur**
Foudil CHERIF	Université de Biskra	MCA	**Examinateur**
KamalEddine MELKMI	Université de Biskra	MCA	**Examinateur**

Session: 2015

Remerciements

AVANT TOUT, *merci mon DIEU de m'avoir donné le courage, la volonté et la patience de mener ce travail à terme.*

Je tiens à adresser des remerciements tout particuliers à Dr. Mohamed Chaouki BABA-HENINI, Maître de conférences, pour l'investissement très important qu'il a mis dans l'encadrement de mon mémoire. Son encouragement et ses remarques pertinentes m'ont permis de mieux structurer ce travail et de mieux le décrire. L'aboutissement de celui-ci doit beaucoup à son contribution. Son perfectionnisme et ses vastes connaissance sont permis de baliser l'avancée de ce travail jusqu'à son aboutissement.

Je remercie également les membres du jury de me faire l'honneur d'examiner et de juger cette thèse :

Mr NourEddine DJEDI Professeur à l'université de Biskra qui m'a fait l'honneur de présider le jury de cette thèse.

Mr Foudil CHERI Maître de conférences à l'université de Biskra, Mr KamalEddine MELKMI Maître de conférences à l'université de Biskra, qui m'ont fait l'honneur, de prendre ce travail en considération en tant qu'examinateurs de cette thèse.

خلاصة : إن إعادة إعمار الصور الطبية ثلاثية الأبعاد 3D بطيء جدا بسبب الخوارزميات الحالية. الحل التقليدي بإستخدام مجموعة حواسيب لحل مشكلة الحوسبة عالية الأداء غير ممكن في بيئة روتينية إكلينيكية. معالجات البطاقات الرسومية « وحدات معالجة الرسومات » (GPU) تمثل الآن حل اقتصادي جدا من أجل الحصول على قدرة حاسوبيه عالية في التصوير الطبي. ومع ذلك، عدد قليل من الطرق المخصصة لـ GPU متوفرة ضمن سياق إعادة الإعمار.

أهداف هذه المذكرة هي :

١- استعراض حالة طرق اعادة الإعمار ثلاثي الأبعاد في مجال التصوير الطبي،

٢- استعراض الدراسات السابقة حول خوارزميات التصور بشكل فوري باستخدام الموازاة واسعة النطاق،

٣- اقتراح لنهج إعادة الإعمار و تطبيقه على الصور الطبية عن طريق إطلاق مجموعة من الأشعة بإستخدام بطاقة الشاشة GPU ،

٤- تنفيذ النهج المقترح،

٥- اختبار وتقييم النهج المقترح ومقارنة النتائج المتحصل عليها مع الأعمال القائمة في أرض الواقع.

كلمات دليلية : إعادة الإعمار ثلاثي الأبعاد، بطاقة الشاشة، إطلاق مجموعة من الأشعة

Résumé : La reconstruction 3D des images médicales est très lente notamment à cause des algorithmes actuels utilisés. La solution classique qui consiste à utiliser un cluster d'ordinateur pour résoudre le problème du calcul intensif n'est pas réaliste à l'intérieur d'un environnement de routine clinique. Les processeurs des cartes graphiques « Graphics processing units » (GPU) représentent aujourd'hui une solution très économique pour obtenir une puissance de calcul importante en imagerie médicale. Toutefois, peu de méthodes dédiées au GPU sont proposées dans un contexte de reconstruction.
La description des objectifs du mémoire est :

1. État de l'art sur la reconstruction 3D dans le domaine de l'imagerie médicale,
2. Étude bibliographique de l'État de l'art sur les GPU et sur les algorithmes de visualisation en temps réel en utilisant un parallélisme massif,
3. Proposition d'une approche de reconstruction 3D appliquée aux images médicales et basée sur un lancer de rayon optimisé par les GPU,
4. Implémentation de l'approche proposée,
5. Teste et bilan du système proposé et comparaison des résultats obtenus avec les travaux existants dans la littérature.

Mots clés : reconstruction 3D, GPU, lancer de rayon optimisé

Abstract : The rendering 3D of the medical images is very slow in particular because of the current algorithms used. The traditional solution which consists in using a cluster computer to solve the problem of intensive calculation is not realistic inside an environment of clinical routine. The processors of the graphics cards « Graphics processing units » (GPU) represent a very economical solution today to obtain an important computing power in medical imagery. However, few methods dedicated to the GPU are proposed in a context rendering. The description of the objectives of the report is:

1. State of the art on the rendering 3D in the field of the medical imagery.
2. Bibliographical study of the state of the art on the GPU and the algorithms of visualization in real-time by using a massive parallelism.
3. Proposal for an approach of rendering 3D applied to the images medical and based on a raycasting by the GPU.
4. Implementation of the approach suggested.
5. Test and assessment of the system suggested and comparison between the results obtained and existing work in the literature.

Keywords : The rendering 3D, GPU ,raycasting

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

INTRODUCTION

0.1 Contexte

De nos jours, l'informatique est un outil de plus en plus utilisé par les scientifiques, car elle offre un moyen rapide et efficace pour manipuler et extraire de l'information des données scientifiques. Parmi les moyens d'extraction d'informations de données, on trouve notamment la visualisation, qui consiste à offrir à un utilisateur une représentation d'un ensemble de données numériques. Cette représentation peut prendre différentes formes, dont un texte ou une image, par exemple. Son principal but est de proposer un moyen d'interpréter correctement des données en offrant à la fois la représentation la plus intuitive possible et en se fiant et s'adaptant à la perception humaine, la vue notamment.

En matière d'imagerie et alors que la visualisation d'informations s'était cantonnée aux modèles surfaciques jusque là, les années 90 ont vu apparaître un nouveau type de données : les données dites *"volumiques"*. Par opposition aux données surfaciques, qui permettent la description d'un objet par ses bords, les données volumiques constituent une discrétisation complète de l'objet, et plus précisément, de ce qu'il contient. Ce nouveau thème dans la visualisation a notamment pu être développé avec la forte demande émanant du domaine médical : l'imagerie par résonance magnétique, datant du début des années 80, est un exemple de procédé produisant des données volumiques. L'appareil IRM permet de produire un ensemble d'images représentant des coupes virtuelles d'un objet.

Ce domaine d'imagerie médicale ,permet de construire et d'utiliser de grands tableaux tridimensionnels de données. Ces volumes peuvent contenir plusieurs Méga-octets de données. Le scanner, comme une multitude d'autres sources (IRM, ultrasons, microscopes, simulations, outils de modélisation, ...), peut fournir des données numériques sous la forme d'une grille à trois dimensions. Ces données, généralement appelées *volumes*, ne sont évidemment pas directement utilisables par une personne. Ainsi les volumes issus d'un scanner sont visualisés généralement par un médecin sous forme d'une série de coupes 2D. Les formes 3D du volume sont alors difficilement interprétables. Il est possible de visualiser directement des données 3D grâce à l'outil informatique. Cette action s'appelle *"le rendu volumique"*.

Les algorithmes de rendu volumique sont souvent classées en deux catégories, la première catégorie est constituée d'algorithmes dits de rendu volumique direct et la seconde est constituée d'autres algorithmes dits de rendu volumique indirect. Nous nous sommes **intéressés** ici uniquement aux **algorithmes de rendu volumique direct** .

Donc, plusieurs techniques de rendu de volume ont été développés pour visualiser l'ensemble des données 3D en une image 2D unique. Les techniques de rendu de volume permettent de transmettre plus d'informations que les méthodes de rendu de surface, mais au profit du coût de l'augmentation de la complexité de l'algorithme et par conséquent une augmentation du temps de calcul [Bruckner 2008].

Raycasting [Kajiya & Von Herzen 1984] est l'une de ces techniques. Il évalue la couleur de chaque pixel de l'image finale par le tracé d'un rayon à travers la scène à partir de la position de l'observateur. Si le rayon frappe le volume, la couleur du pixel est calculée en échantillonnant les valeurs de données le long du rayon en un nombre fini de positions dans le volume et en les combinant ensemble. Cette technique possède, cependant, une limitation, lorsqu'elle est exécutée sur des **CPUs** : pour les grands ensembles de données de volume qui maximisent le nombre de rayons qui doit être tracé, le temps de rendre une seule image est trop élevée pour permettre une visualisation en temps réel.

Guidés par la demande de l'industrie du jeu, les performances des **GPUs** modernes a dépassé la puissance de calcul des **CPUs** en chiffres bruts et en taux de croissance [Stegmaier *et al.* 2005, Weiskopf 2006]. Par conséquent, les **GPUs** apparaissent comme une

opportunité intéressante pour exécuter des algorithmes lourds, pour lesquels les **CPUs** ne peuvent pas donner une réponse en temps réel. Les cartes graphiques modernes sont caractérisés par les caractéristiques suivantes [Kruger 2003, Scharsach 2005] :

- Une architecture massivement parallèle,
- Une séparation en deux unités distinctes (vertex et fragment shaders) qui peuvent doubler les performances si la charge de travail peut être divisé,
- Mémoire rapide et interface de la mémoire,
- Instructions spécifiques pour les tâches graphiques,
- Opérations vectorielles sur quatre flottants qui sont aussi rapides que les opérations scalaires,
- Interpolation trilinéaire qui est automatiquement (et extrêmement rapide) mis en œuvre dans la texture 3D.

L'algorithme de *raycasting* s'adapte à toutes les cartes graphiques modernes. Il présente un parallélisme intrinsèque, sous la forme de rayons lumineux totalement indépendants, ce qui permet de profiter de l'architecture massivement parallèle du **GPU**.

0.2 Problématique

Les méthodes en temps réel pour transmettre directement le contenu d'information de grands champs scalaires volumiques ont toujours été un défi à la communauté d'infographies. Partant du constat que la capacité d'un seul CPU à usage général n'est pas suffisante pour atteindre le temps réel ou même l'interactivité pour les grands ensembles de données en général.

En général, **la vitesse et la qualité** atteints en proposant un algorithme qui peut atteindre le plus haut niveau de qualité avec une interaction maximale [Jonsson 2005, WikipediaTrilInterp 2015]. Dans la littérature, un grand nombre de techniques de rendu de volume ont été présentés et utilisés soit pour augmenter la vitesse de rendu des grands ensembles

de données ou pour améliorer la qualité de rendu des images reconstruites. Par exemple, les algorithmes de *raycasting* offrent une excellente qualité de création d'image, au prix du manque d'interactivité souhaitée par l'utilisateur final. L'algorithme nécessite beaucoup de stratégies d'optimisations de vitesse pour réduire le temps nécessaire pour parcourir la totalité du volume.

L'inconvénient majeur du *raycasting* est donc le fait que ce calcul est très coûteux et il est directement proportionnel à la taille des données de volume et de la taille de l'image finale qui doit être projetée sur l'écran.

Considérant la technologie de l'informatique d'aujourd'hui, quelle technique peut accélérer l'étape de *raycasting* dans le rendu d'imagerie médicale 3D ?

0.3 Objectif du mémoire

L'avènement des GPUs, combiné avec leur haut degré de programmabilité ouvre un large champ de nouvelles applications pour les cartes graphiques. *Raycasting* est parmi ces applications, présentant un parallélisme intrinsèque, sous la forme des rayons lumineux totalement indépendants, ce qui permet de profiter de l'architecture massivement parallèle des GPUs.

L'objectif de ce mémoire de magister est d'explorer les possibilités d'utilisation du calcul parallèle sur le GPU pour le rendu en temps réel de l'imagerie médicale. Après des années de recherche et des améliorations, un tel rendu est maintenant en mesure de fonctionner en temps réel. Les capacités en temps réel des mises en œuvre de ces dernières s'appuient principalement sur l'utilisation du calcul parallèle.

Le calcul parallèle sur les CPUs multi-core est un domaine établi depuis longtemps. Le calcul parallèle sur les GPUs, cependant, est assez récent, et a été rendu célèbre par l'API **CUDA** de *NVidia* en 2007. Par rapport à un CPU, le GPU possède beaucoup plus d'unités de calcul (noyaux) et il est donc capable de gérer beaucoup plus de calculs simultanés.

Quelques années plus tard, *Khronos Group* a publié sa première mise en œuvre d'**OpenCL**. Même si les deux APIs sont destinés à effectuées les mêmes tache, l'API CUDA reste plus populaire, parce qu'il a été publié avant, il est donc considéré comme plus mature

qu'OpenCL. L'API OpenCL, cependant, a le grand avantage d'être multi-plateforme, ce qui signifie que le même programme est capable de fonctionner sur différents types de matériel.

En conclusion notre but est le rendu interactif d'imagerie médicale 3D sur le GPU en recourant à des techniques de *raycasting* mise en œuvre à l'aide d'un ensemble des noyaux d'OpenCL.

0.4 Plan du mémoire

Le présent mémoire est structuré en quatre chapitres de la manière suivante :

Chapitre 1 : État de l'art sur les méthodes de visualisation 3D en Imagerie Médicale

Dans ce premier chapitre, nous présentons un aperçu sur les méthodes de rendu volumique et quelque définitions sur les notions qui lui son lié, ensuite nous comparons entre ces méthodes afin de motiver nos choix sur la proposition de la technique que nous avons faite, et nous terminerons le chapitre par une conclusion dans laquelle nous allons faire un bilan sur les techniques sus citées.

Chapitre 2 : Proposition D'une Approche de Rendu volumique Basée sur GPU

Dans le deuxième chapitre, nous nous intéressons à notre approche de rendu volumique d'imagerie médicale par la méthode de **GPU raycasting**, nous exposons les étapes de l'approche et nous expliquons chaque étape séparément à partir de l'étape d'acquisition des données de volume 3D jusqu'à l'affichage d'une image 3D projeté sur un plan 2D.

Chapitre 3 : Algorithme Et Mise En Œuvre

Dans la troisième chapitre, nous présentons la définition des étapes de notre algorithme générale qui est comprend deux parties, l'une est la partie CPU (hôte) ; l'autre est la partie GPU (thread), puis la forme globale du programme et de ses deux parties : code du hôte qui

s'exécute sur le CPU, et le code de thread qui s'exécute sur le GPU par un seul dispositif (device) qui définit l'algorithme de **GPU raycasting**.

Chapitre 4 : Analyse et interprétation des résultats

Dans ce chapitre, nous présentons les résultats de l'approche proposée de raycasting basé sur le GPU, que nous allons testés sur un ensembles de données d'imagerie médicale 3D et sur une plateforme opencl/opengl installé sur le système windows.

Conclusion et perspectives

Ce mémoire s'achève par une conclusion générale récapitulant et traçant les perspectives potentielles que nous envisageons d'entreprendre dans des travaux futurs.

CHAPITRE 1

ÉTAT DE L'ART SUR LES MÉTHODES DE VISUALISATION 3D EN IMAGERIE MÉDICALE

1.1 Introduction

L'interprétation des champs scalaires 3D pose de beaucoup de problèmes en raison de leur complexité. Dans le contexte de ce mémoire le rendu de volume se rapporte à la visualisation de champs statiques scalaires 3D en imagerie médical 3D. Bien qu'une variété d'approches différentes ont été développées au cours des dernières décennies, il reste encore quelques problèmes à résoudre, surtout avec l'apparition des nouvelles techniques en infographie et le développement du matériel graphique.

Les recherches récentes pour le rendu de volume sont principalement axées sur l'amélioration de la qualité de l'image et l'affichage des informations plus détaillées sur les données de volume. L'interactivité est également un sujet de recherche actif pour le rendu de volume.

Initialement, nous pouvons résumé les problématiques liées au rendu de volume dans : la génération de la fonction de transfert, le rendu de volume non-photo-réaliste, l'accélération matérielle, la combinaison de la segmentation en rendu de volume et d'autres approches pour améliorer la qualité des images et les performances de rendu, ... etc.

Ce chapitre tente d'évaluer les différents algorithmes de rendu de volume et de tirer une

vue d'ensemble des concepts et des idées importantes. Il examine tout d'abord les algorithmes de rendu de volume existants et compare les avantages et les inconvénients de chacune.

1.2 Notions de base pour le rendu de volume en médecine

1.2.1 Les données de volume

Traditionnellement, le rendu de volume représente un modèle comme étant un ensemble de vecteurs qui ont été affichées sur écrans graphiques vectoriels. Avec l'introduction de l'affichage de données raster, les polygones sont devenus les éléments de base du rendu, pour cela les polygones d'un modèle sont transformés en pixels, dans les tampons de le mémoire vidéo. Par rapport aux données de surface qui déterminent uniquement l'enveloppe extérieure d'un objet, les données de volume sont utilisées pour décrire les structures internes d'un objet solide.

Un volume est une grille 3D régulière des valeurs de données scalaires appelés *voxels*. La grille à trois dimensions peut également être considérée comme un empilement des tableaux de valeurs de données et chacun de ces tableaux à deux dimensions comme une image (ou tranche), où chacune des valeurs de données représente un pixel (Figure 1.1). Cette vision alternative est motivée par la tranche orientée, la façon traditionnelle des médecins d'observer un volume de données, elle est notée par une matrice $V = \Gamma^{X \times Y \times Z}$ avec des lignes X, des colonnes Y et des tranches Z, ce qui représente une grille discrète d'éléments de volume (ou voxels) $v \in 1, ..., X \times 1, ..., Y \times 1, ..., Z$. pour chaque voxel on désigne par $I(v) : N^3 \rightarrow \Gamma$ sa valeur de gris, qui, par exemple, reflète l'intensité des rayons X dans les données de volume CT. La valeur de voxel peut être un vecteur représentant les propriétés de l'objet dans certains domaines spécifiques (par exemple, Computational Fluid Dynamics). Chaque voxel est caractérisée par sa position dans la grille 3D. Les données de volume médicale obtenues à partir de l'IRM (Imagerie par Résonance Magnétique) et les scanners CT (Computed-Tomography) sont généralement de diverses propriétés avec une densité d'échantillonnage égale en direction x et y mais cette densité augmente le long de la direction z. La taille est habituellement d'environ plus de 100 tranches avec 512×512 de voxels chacun, ce qui nécessite un soin particulier concernant l'efficacité des algorithmes développés. Ces ensembles

des données V sont la base de notre évaluation d'algorithmes de rendu de volume.

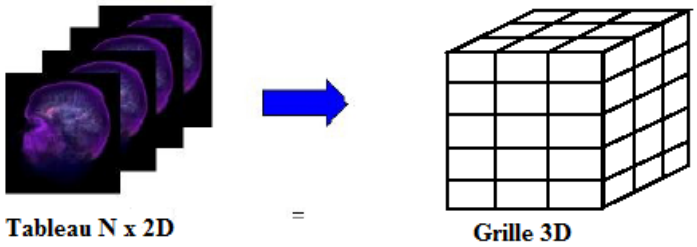


FIGURE 1.1 – Représentation des données de volume 3D

1.2.2 Équation du rendu volumique

L'objectif de base du rendu de volume est de trouver une bonne approximation du modèle optique qui exprime la relation entre l'intensité du volume et la fonction de l'opacité et l'intensité dans le plan d'image. Dans cette section, nous décrivons un modèle physique typique basé sur les algorithmes de rendu de volume. Notre intention est d'expliquer les fondements théoriques sur les algorithmes de rendu de volume. Les modèles optiques physiques bénéficient en théorie du transport radiatif qui tente d'afficher un volume comme un nuage peuplé des particules. Le transport de la lumière est étudiée en tenant compte des différents phénomènes à l'œuvre. La lumière d'une source peut être soit dispersée ou absorbée par les particules. Il pourrait y avoir une augmentation nette lorsque les particules émettent eux-mêmes la lumière. Les modèles qui tiennent compte de tous les phénomènes ont tendance à être très compliqués.

Dans la pratique, les modèles locaux beaucoup plus simples sont utilisés par la plupart des algorithmes de rendu de volume standards utilisent une équation intégrale de rendu de volume [Max 1995, Meissner *et al.* 2000] définit par :

$$I_\lambda(x,y) = \int_0^L C_\lambda(S)\mu(s)e^{-\int_0^s \mu(t)dt}ds \tag{1.1}$$

I_λ est la quantité de lumière de la longueur d'onde λ venant de la direction de rayon r qui est reçu à l'emplacement x sur le plan d'image, L est la longueur du rayon r, μ est la densité des particules en volume qui reçoivent la lumière provenant de toutes les sources lumineuses environnantes et qui reflètent cette lumière vers l'observateur en fonction de leur spécularité

et les propriétés des matériaux diffus, C_λ est la lumière de longueur d'onde λ réfléchie et/ou émise au lieu de s dans la direction de r.

Dans le cas général, l'équation 1.1 ne peut pas être calculé analytiquement. Par conséquent, dans des applications pratiques, la plupart des algorithmes de rendu de volume doivent obtenir une solution numérique de l'équation 1.1 grâce à l'emploi d'une quadrature d'ordre zéro de l'intégrale intérieure avec une approximation de premier ordre de l'exponentielle. L'intégrale extérieure est également résolu par une somme finie des échantillons uniformes. Alors nous obtenons l'équation suivante [Levoy 1990] :

$$I_\lambda(x,y) = \sum_{k=1}^{M} C_\lambda(s_k)\alpha(s_k) \prod_{i=1}^{k-1}(1-\alpha(s_i)) \tag{1.2}$$

où $\alpha(s_k)$ sont les échantillons d'opacité le long du rayon et $C_\lambda(s_k)$ sont les valeurs de couleurs locales dérivées du modèle d'illumination. Cette expression est désigné comme discrétisé, où l'opacité $\alpha = 1.0 - transparence$. L'équation 1.2 représente un cadre théorique commun à tous les algorithmes de rendu de volume. Qui permettent d'obtenir des couleurs et des opacités dans des intervalles discrets le long d'un chemin linéaire et les associes en avant vers l'arrière. Cependant, les algorithmes peuvent se distingués par le procédé dans lequel les couleurs $C_\lambda(s_k)$ et les opacités $\alpha(s_k)$ sont calculées à chaque intervalle k, et la largeur de l'intervalle Δs est choisi [Meissner *et al.* 2000]. C et α sont maintenant des fonctions de transfert, généralement mise en œuvre comme des tables de consultation. Les densités de volume brutes sont utilisés pour indexer les fonctions de transfert pour la couleur et l'opacité, donc les petits détails des données de volume peuvent être exprimés dans l'image finale par l'utilisation de différentes fonctions de transfert.

1.3 Rendu volumique

1.3.1 Introduction

De nombreuses applications, telle l'imagerie médicale, la sismologie ou la microscopie électronique, construisent et utilisent de grands tableaux tridimensionnels de données. Ces volumes peuvent contenir plusieurs Méga-octets de données. Le scanner, comme une multitude d'autres

sources (IRM, ultrasons, microscopes, simulations, outils de modélisation, ...), peut fournir des données numériques sous la forme d'une grille à trois dimensions, ces données, généralement appelées volumes, ne sont évidemment pas directement utilisables par une personne, ainsi les volumes issus d'un scanner sont visualisés généralement par un médecin sous forme d'une série de coupes 2D, les formes 3D du volume sont alors difficilement interprétables. 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 décrire les algorithmes de rendu volumique les plus populaires et les plus récents. Ces méthodes sont souvent classées en deux catégories, la première catégorie est constituée d'algorithmes dits de rendu volumique direct et la seconde est constituée d'autres algorithmes dits de rendu volumique indirect. Nous nous sommes intéressé ici uniquement aux algorithmes de **rendu volumique direct**

1.3.2 Raycasting

Principe

La technique de décomposition dans l'espace de l'image la plus courante est le raycasting [Levoy 1988]. Dans ce cas, la discrétisation de l'intégrale de rendu est explicite dans la mesure où, comme le montre la Figure 1.2, un rayon virtuel est lancé explicitement dans le volume pour chaque pixel de l'espace de l'image et l'intégration est calculée le long de ce rayon par échantillonnage à pas réguliers. Comme le montre l'agrandissement sur cette même figure, les points d'échantillonnage du rayon ne coïncident pas forcément avec ceux de la grille. Ainsi, une étape d'*interpolation* applique un des filtres de reconstruction, ensuite à l'étape de classification qui applique la fonction de transfert, puis la composition le long du rayon peut avoir lieu.

Les types de classification

Il existe deux types de classification selon qu'elle intervient avant ou après interpolation : La *post-classification*, et la *pré-classification* (Figure 1.3). La 1$^{\text{ère}}$ applique la fonction de transfert sur une valeur interpolée du champ scalaire. Inversement, la 2$^{\text{ème}}$ applique la fonction de transfert aux voxels de la grille, l'interpolation ayant lieu dans l'espace *(RVB,A)*.

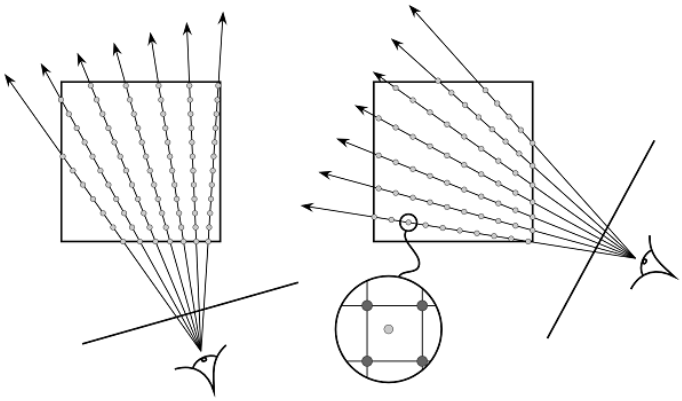


FIGURE 1.2 – Discrétisation explicite de l'intégrale de rendu volumique par raycasting. L'agrandissement montre que les points d'échantillonnage du rayon (gris clair) ne coïncident pas forcément avec les voxels de la grille (gris foncé).

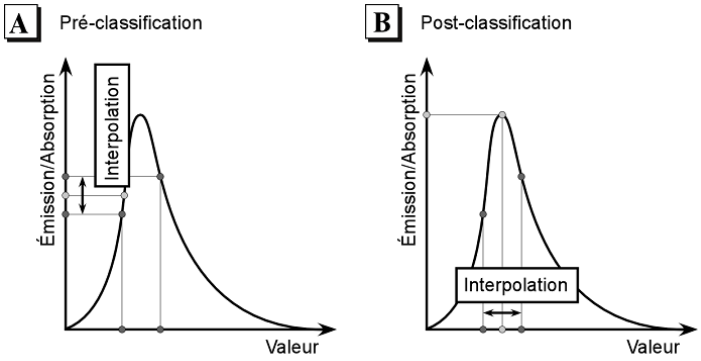


FIGURE 1.3 – Différence entre pré-classification (A) et post-classification (B). Selon que l'étape de classification intervient avant (pré-classification) ou après (post-classification) l'étape d'interpolation, les propriétés retournées par la fonction de transfert peuvent être très différentes (d'après [Rezk-Salama 2001]).

Cette dernière méthode conduit dans certains cas à des incorrections sur l'image finale. Une des raisons est que chaque composant de l'espace *(RVB,A)* est interpolé indépendamment [Wittenbrink *et al.* 1998]. Pour éviter ces effets il faut multiplier l'émission propre par l'absorption préalablement à l'interpolation [Gasparakis 1999, Wittenbrink *et al.* 1998]. Les couleurs primaires ainsi utilisées sont dites *couleurs associées (associated colors)* [Blinn 1994]. Cependant, une autre source d'erreurs réside dans la non linéarité de la fonction de transfert [Engel *et al.* 2001]. La Figure 1.3 montre qu'une seule fonction de transfert constante ou linéaire peut conduire à un résultat correct, ce qui dans la pratique est peu probable. Donc, la post-classification est la seule qui produit des résultats correct au sens où il s'agit d'appliquer une fonction de transfert à un champ scalaire continu défini sur un maillage [Engel *et al.* 2001, Lichtenbelt *et al.* 1998b].

Méthodes d'accélération

Quel que soit l'ordre d'application des opérations d'interpolation et de classification, le raycasting permet l'intégration de méthodes d'accélération [Levoy 1990] telles que *"early ray termination" (la terminaison précoce de rayon)* et *"empty space sleeping" (l'accélération dans les espaces vides)*. La 1ère consiste à parcourir le rayon de l'avant vers l'arrière et à stopper la progression dès qu'un degré suffisamment élevé d'opacité a été atteint, la limite pouvant être fixée par l'utilisateur. La 2ème utilise des structures de données hiérarchiques qui permettent d'accélérer la progression le long du rayon dans les zones transparentes du volume.

Avantages et inconvénients

- Ce type d'algorithme est aisément parallélisable sur des architectures SIMD,
- Facilement accélérée matériellement par une carte graphique dédiée,
- Contrairement au lancer de rayons, le raycasting est nettement plus rapide.

1.3.3 Algorithme d'« Aplatissement » (« Splatting »)

Principe

Les techniques de décomposition dans l'espace de l'objet sont appelées techniques de projection. Elles ont pour philosophie l'approche originale d' "Aplatissement" [Westover 1989]. Cette approche, illustrée à la Figure 1.4, calcule la contribution de chaque voxel de la grille indépendamment par projection sur le plan de l'image. Ainsi, les voxels sont successivement composés au niveau des pixels qu'ils affectent, les pixels affectés sont déterminés par l'extension sur l'image de l'*empreinte ("footprint"* en anglais) du voxel projeté. Elle dépend d'un noyau de convolution 3D appliqué au voxel avant projection. Il s'agit généralement d'un noyau Gaussien, équivalent à une petite sphère diffuse autour du voxel. La symétrie du noyau permet une pré-intégration de son empreinte 2D une seule fois pour tous les voxels indépendamment du point de vue. Il suffit ensuite de multiplier cette empreinte générique par la valeur du voxel projeté, ce qui fait l'efficacité de la méthode.

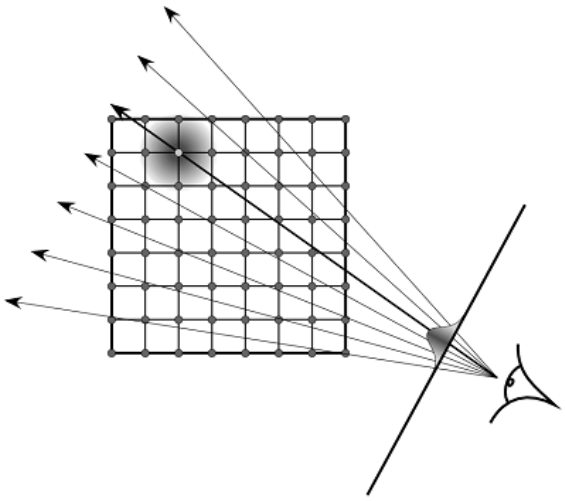


FIGURE 1.4 – Discrétisation implicite de l'intégrale de rendu volumique par la technique d' *"Aplatissement"*. La contribution de chaque voxel à l'image finale est calculée par projection. L'empreinte (*"footprint"*) du voxel projeté (gris clair) sur l'image dépend d'un noyau de convolution (gaussien dans notre cas) appliqué avant projection.

Du fait de la non commutativité de la composition, l'ordre de projection est important. Les voxels sont classés par tranches alignées selon l'axe du volume le plus perpendiculaire au plan de l'image et, dans chaque tranche, par ordre croissant de leur distance à l'œil de l'observateur.

Cette approche appelée *"axis-aligned sheet buffered splatting"* [Westover 1990], permet de projeter chaque tranche indépendamment sur le plan de l'image, puis de les composer. Pour aller plus loin, les voxels peuvent être classés par tranches, non pas alignées sur un axe du volume, mais parallèlement à l'axe de l'image. Cette méthode appelée *"image-aligned sheet buffered splatting"* [Mueller & Crawfis 1998] élimine certains artefacts liés à l'approche précédente. Nous verrons comment les techniques de rendu volumique à base de textures s'apparentent à ces méthodes.

La distinction entre pré-classification et post-classification soulignée dans la section précédente pour les méthodes explicites intervient également pour l' "aplatissement". L'approche originale [Westover 1989, Westover 1990] applique la fonction de transfert avant projection, ce qui s'apparente à une pré-classification. Une autre approche, plus récente, permet d'effectuer une post-classification dans l'espace de l'image après projection [Mueller *et al.* 1999].

Splatting vs Raycasting

À la différence des algorithmes basés sur le lancer de rayons, les algorithmes d'aplatissement itèrent sur les voxels.

Comme les algorithmes d'aplatissement itèrent sur le volume, ils peuvent parcourir celui-ci selon son ordre naturel de stockage. En revanche, le calcul du filtre de ré-échantillonnage est très coûteux, car celui-ci dépend de la position de l'observateur : il peut être mis à l'échelle, tourné, et transformé de façon arbitraire.

En théorie, les algorithmes d'aplatissement peuvent fournir les même images qu'avec les algorithmes de lancer de rayons. En pratique, du fait que le calcul des paramètres des filtres est difficile, des approximations sont utilisées, qui permettent soit d'obtenir une exécution efficace, soit des images de qualité, mais jamais les deux simultanément.

1.3.4 Shear-Warp

L'algorithme de rendu volumique *shear-warp* [1] permet de gagner plus d'un ordre de grandeur en exploitant conjointement les cohérences spatiales du volume de données et du plan image.

[1] proposé par `Lacroute` [Lacroute & Levoy 1994]

Principe

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 d'aplatissement en décomposant la projection des voxels sur le plan image en deux étapes :

1. Tout d'abord, on effectue une projection des voxels sur un plan objet parallèle à l'une des faces du volume à visualiser. Comme le plan objet utilise le même système de coordonnées que le volume, cette projection peut se faire de façon efficace en suivant l'ordre naturel de stockage du volume,

2. Ensuite, le plan objet est projeté sur le plan image, à un coût très bas car il ne s'agit plus de projections d'informations volumiques mais bidimensionnelles.

Afin d'optimiser la construction de l'image intermédiaire, il est possible de réorganiser les plans du volume de façon à ce que la projection des voxels sur le plan objet soit toujours parallèle à l'axe de profondeur. Pour cela, on décale (« *shear* ») les plans du volumes en fonction de la direction initiale des rayons de vision, comme on le voit ci-dessous dans le cas d'une perspective isométrique (Figure 1.5).

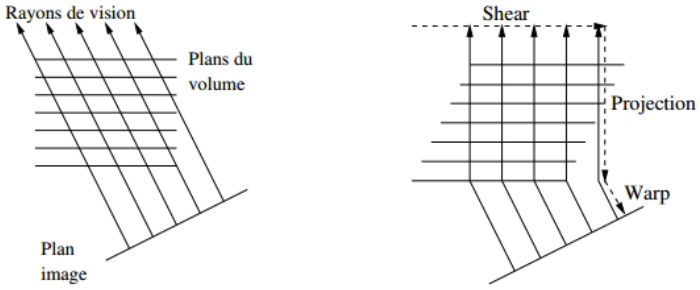


FIGURE 1.5 – Algorithme du *shear-warp* avec une projection parallèle [Neumann 2001].

Cette technique est également valable pour les perspectives classiques, au prix d'un changement d'échelle des plans en fonction de leur profondeur (Figure 1.6).

On obtient donc un algorithme en trois phases (voir [Mora 1986] pour la projection isométrique et [Lacroute & Levoy 1994] pour la perspective normale) :

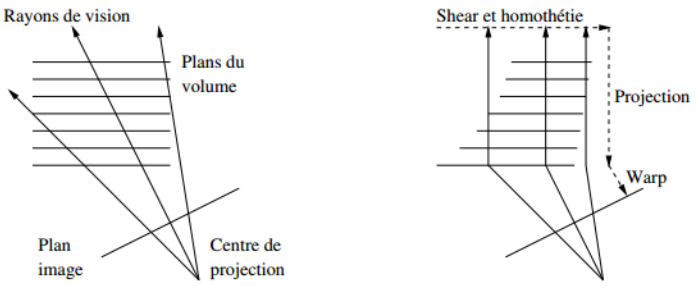


FIGURE 1.6 – Algorithme du *shear-warp* avec une projection perspective [Neumann 2001].

1. Transformation des données du volume dans l'espace objet déformé, par translation et remise à l'échelle de chaque plan. La direction des plans est choisie comme celle étant la plus perpendiculaire à la direction de vision. Comme les coefficients de translation et d'homothétie ne sont pas nécessairement entiers, chaque plan objet doit être correctement ré-échantillonné,

2. Composition des plans objet déformés, du plus proche de l'observateur vers le plus lointain, en utilisant l'opérateur de recouvrement : les contributions ajoutées derrière un pixel translucide sont atténuées, et celles ajoutées derrière un pixel opaque sont ignorées. On obtient ainsi une image intermédiaire distordue, car non perpendiculaire à la direction réelle de vision et non alignée avec les axes de l'image finale,

3. Projection de l'image intermédiaire distordue sur le plan image, pour obtenir l'image finale.

Afin de ne pas manipuler de gros volumes de données, les mises à l'échelle peuvent se faire de façon logique, en calculant à la volée ces transformations lors du parcours des plans du volume selon la direction choisie.

Structures de données

Des études statistiques sur des cas réels ont montrée qu'entre 65% et 90% des voxels sont considérés comme transparents. Pour éviter de les stocker, on utilise une structure de type « *run-length* ».

Comme le parcours de la structure compactée ne peut se faire que dans la direction de compactage, on pré-calcule trois structures compactées à partir des données volumique, une par direction principale, de sorte qu'un changement de direction passe inaperçu lors de la phase de visualisation.

Parallélisation

l'étape *warp* représente, pour des volumes de données de taille moyenne, moins de 20% du temps total de rendu, l'essentiel du temps étant consommé par la phase de composition des voxels sur l'image intermédiaire, sur laquelle doit se porter l'effort de parallélisation.

Le parallélisme de l'algorithme de *shear-warp* se réalise assez naturellement sur une machine multi-processeurs. En revanche, la parallélisation sur machine distribuée pose de nombreux problèmes, en particulier la redistribution des données entre processeurs lorsque la direction de vision change brutalement.

Chaussumier [Chaussumier *et al.* 1999] a implémenté une version parallèle de l'algorithme de *shear-warp* sur une grappe de PC reliés par un réseau d'interconnexion *Myrinet*. La distribution des calculs se fait par lignes de l'image intermédiaire, afin de conserver une certaine localité des données, ainsi que les algorithmes de terminaison anticipée. Les données des lignes de voxels correspondant aux lignes de l'image intermédiaire sont distribuées sur les processeurs, de façon à minimiser la réplication. Ceci n'est pas simple, car cette distribution est dépendante du point de vue et morcelle chaque coupe de données sur chaque processeur.

Les communications engendrées par cette distribution de données ont lieu chaque fois que le volume est déformé, c'est-à-dire à chaque changement de point de vue. les données n'étant pas répliquées, chaque processeur demande à tous les autres les lignes qu'il lui manque dans chacune des coupes du volume qu'il doit traiter. Il faut alors réaliser une multi-distribution de morceaux de structures creuses, puisque chaque processeur envoie des données différentes à tous les autres.

L'utilisation de structures creuses fait qu'on ne peut prédire simplement la quantité de travail à partir de la quantité totale de données et de l'angle de vue. Un équilibrage de charge simple basé sur l'entrelacement de blocs de lignes de l'image intermédiaire n'étant pas efficace,

il faut avoir recours à des techniques d'équilibrage dynamique de la charge. Cependant, un équilibrage adaptatif se basant sur le rendu précédent n'est pas suffisant lorsque l'on change de point de vue de manière arbitraire.

Pour remédier à cela, chaque processeur calcule, sur les portions de données qu'il possède, un coût partiel pour chacune des lignes du nouveau rendu. Les tableaux des coûts partiels de lignes sont alors sommés par l'ensemble des processeurs (réduction de type « *all-reduce*) », de telle sorte que chaque processeur puisse calculer le coût total du rendu (en sommant les cases du tableau réduit), et en déduise localement une distribution des lignes, il ne reste plus alors à chacun qu'à effectuer la redistribution des données en conséquence.

Cependant le redistribution est coûteuse, et empêcher toute scalabilité si les communications ne sont pas recouvertes par le calcul. Pour recouvrir les communications, il faut identifier, à l'intérieur de l'algorithme, des communications et du calculs indépendants pouvant s'effectuer simultanément. Dans l'algorithme de *shear-warp*, la plus petite granularité de composition efficace est au niveau des lignes (pour bénéficier du codage *run-length*). La granularité choisie pour le recouvrement a été la coupe (ensemble de lignes), car elle permet un recouvrement efficace sans multiplier le nombre de messages.

En pratique, sur une grappe de 4 PC inter-connectés par un réseau *Myrinet* [Chaussumier *et al.* 1999], une image de 512^3 voxels est calculée en 1.5 secondes, avec un déséquilibre de charge n'excédant pas 10%.

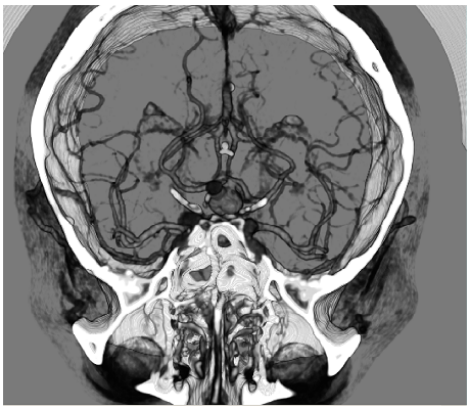
1.3.5 Projection d'intensité maximum (« MIP »)

Principe

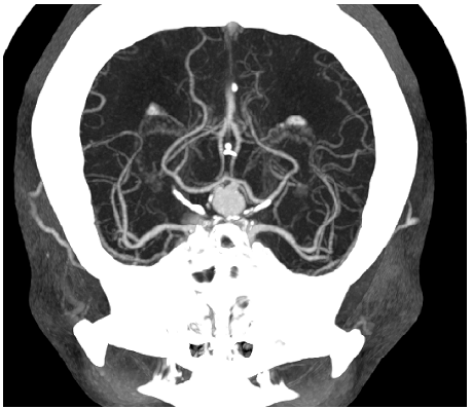
La projection de l'intensité maximale, ou MIP "Maximal Intensity Projection" [Wallis *et al.* 1989, Engel *et al.* 2004], se base sur l'affichage du voxel d'intensité la plus élevée selon une ligne virtuelle traversant le volume d'intérêt depuis l'œil de l'observateur. En fait cette dernière ne requière aucune intégration numérique. Elle ne conserve que l'intensité maximale émise le long de chaque rayon. Avec cette technique, le rapport information par pixel est dans ce cas 1, mais l'intensité maximale projetée ne correspond pas forcément aux structures que l'on souhaite analyser. De plus, on perd le contexte spatial de l'information capturée,

notamment en matière de profondeur, ce qui peut mener à des visualisations prêtant à confusion comme l'image de gauche de la figure 1.7. Les travaux récents de [Díaz & Vázquez 2010] améliorent ce point en proposant de considérer les échantillons dans un intervalle autour de l'intensité maximale rencontrée sur chaque rayon. Ils pondèrent chaque échantillon par leur profondeur réincorporant ainsi leur position relative. On peut aussi utiliser le rendu additif, qui consiste à additionner les couleurs pré-multipliées des échantillons sur les rayons lancés.

Le rendu MIP est fréquemment utilisé en imagerie médicale pour sa simplicité et son efficacité pour visualiser des données tomographiques issues d'angiographies. La figure 1.7 compare le rendu volumique par intégration complète 1.7a et par intensité maximum MIP 1.7b. tandis qu'en (a) une fonction de transfert doit être correctement définie afin de faire ressortir les vaisseaux sanguins, ceux-ci apparaissent naturellement en (b).



(a) rendu volumique par intégration complète



(b) rendu volumique par intensité maximum MIP

FIGURE 1.7 – Comparaison de la tomographie d'une angiographie sur des vaisseaux sanguins à l'intérieur d'un crâne humain calculée par le modèle d'émission-absorption (a) et par projection de l'intensité maximale MIP (b) (d'après [Engel *et al.* 2004]).

Avantages

1. Permet de mettre en évidence les os en scanner X,

2. Très utilisé en imagerie vasculaire pour rehausser le contraste des vaisseaux par rapport aux tissus environnants (IRM). Des produits de contraste sont souvent injectés au

patient pour accentuer le contraste.

Inconvénients

1. Perte d'information de profondeur :
   - Élimination des petits vaisseaux,
   - Pas de discrimination contraste/thrombose marginale,
   - Cela implique un fort risque de mauvaise interprétation de la cohérence spatiale des différentes structures comme là montré [Hastreiter 1999] de manière saisissante.
2. Structures de haute densité (calcifications vite gênantes),
3. La reconstruction MIP des images peuvent masquer des calculs, même ceux de taille importante. On doit évaluée toujours les coupes originales d'acquisition.

1.3.6 Visualisation volumique à base de Placage de texture (« Texture Mapping »)

L'un des éléments déterminants dans l'adoption du matériel graphique standard par les techniques de rendu volumique est l'introduction du support du plaquage de texture. En effet, l'utilisation de textures permet d'accélérer considérablement les opérations d'interpolation bilinéaire [2] et trilinéaire [3] [Cabral *et al.* 1994] particulièrement coûteuses en rendu volumique. La façon dont est conçu le pipeline graphique nous conduit à aborder dans un premier temps l'accélération matérielle des approches implicites [4], plus intuitive que pour les approches explicites [5].

Approches implicites

Textures 2D Les approches implicites à base de textures 2D [Cabral *et al.* 1994] procèdent de manière analogue à l'algorithme du "shear-warp". Les interpolations trilinéaires dans le vo-

[2] textures 2D
[3] textures 3D
[4] approches objet
[5] approches image

lume sont remplacées par des interpolations bilinéaires dans ses plans. Ces plans sont stockés sous forme de textures 2D en mémoire graphique. Comme le montre la Figure 1.8, ces textures sont plaquées sur une série de polygones perpendiculaires à l'axe du volume le plus proche de la direction d'observation. Le GPU se charge des transformations géométriques dans l'unité de traitements de sommets, de la décomposition en fragments à l'étape de rasterisation et l'interpolation bilinéaire native du matériel graphique est utilisée dans l'unité de traitement de fragments. Cette dernière étant optimisée pour effectuer un très grand nombre d'interpolations par seconde, il est possible de générer les plans de fragments texturés en temps réel. Finalement, la composition est calculée par pixel grâce aux opérations d'"alpha-blending".

Cependant, comme le montre la Figure 1.8, lors d'un changement de point de vue l'axe du volume le plus proche de la direction de visualisation peut changer. Dans ce cas, la série de plans à dessiner et donc à stocker sous forme de textures est modifiée. Pour réagir interactivement à une telle éventualité, il est nécessaire de stocker le volume trois fois, sous la forme de trois séries de plans alignés perpendiculairement à chaque axe. D'autre part, au moment du changement de plans, des artefacts apparaissent, proches de ceux observés dans l'"axis-aligned sheet buffered splatting" [Westover 1990]. Ceux-ci peuvent être réduits en générant des plans intermédiaires à la volée sur le GPU [Rezk-Salama *et al.* 2000a]. Malheureusement, cette approche ne résout pas le problème de manière satisfaisante et surtout ne permet pas de s'affranchir des trois copies du volume. Pour cela, il est nécessaire d'utiliser des textures 3D.

Textures 3D Les approches à base de textures 3D [Cabral *et al.* 1994] conservent l'idée de dessiner des plans texturés de l'arrière vers l'avant qui a conduit à l'utilisation de textures 2D. Cependant, l'interpolation bilinéaire dans les textures 2D, qui nécessitait l'utilisation de plans alignés selon les axes du volume, est remplacée par une interpolation trilinéaire dans les textures 3D. Ceci permet donc de dessiner des plans alignés sur l'image indépendamment de l'orientation du volume qui est stocké une seule fois sous forme de texture 3D. L'interpolation trilinéaire native du matériel graphique permet de générer de nouveaux plans en temps réel lorsque le point de vue est modifié. D'autre part, cette approche résout définitivement le pro-

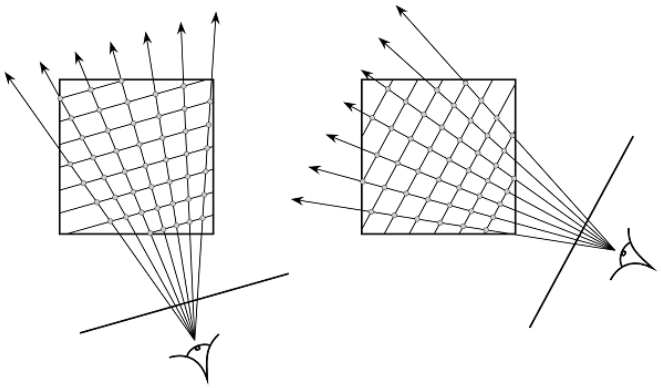


FIGURE 1.8 – Accélération matérielle du rendu volumique à base de textures 3D. Le volume est stocké en mémoire graphique sous forme de texture 3D. L'interpolation trilinéaire native du matériel graphique permet de générer en temps réel des plans alignés sur l'image.

blème des artefacts liés au changement de série de plans dans l'utilisation de textures 2D, de la même manière que l' "image-aligned sheet buffered splatting" [Mueller & Crawfis 1998] apportait une solution aux artefacts de l' "axis-aligned sheet buffered splatting" [Westover 1990].

L'une des principales limitations de l'utilisation des textures 3D par rapport aux textures 2D est le schéma d'accès aux textures qui réduit l'efficacité du cache optimisé pour les textures 2D [Engel *et al.* 2004], beaucoup plus utilisées dans les applications "grand public" telles que les jeux vidéo. Cette distinction se justifie cependant de moins en moins avec les nouvelles cartes graphiques ,il est possible d'optimiser l'utilisation du cache en jouant sur la taille des textures 3D utilisées.

Approches explicites

Historiquement, l'approche explicite par lancer de rayons a toujours été considérée comme la référence en termes de qualité de rendu. En contrepartie, cette qualité s'est longtemps payée au prix de performances bien inférieures à celles des approches implicites à base de textures. Pour cette raison, un effort particulier a été consenti pour faire bénéficier le lancer de rayons des avancées du matériel graphique. L'accélération matérielle dans ce cas est beaucoup plus complexe et moins intuitive à mettre en œuvre. Outre le recours aux textures 2D et 3D, ces techniques tirent également partie du caractère programmable des cartes graphiques, ce qui explique leur apparition plus tardive [Roettger *et al.* 2003, Kruger & Westermann 2003].

La technique de lancer de rayons présente un parallélisme intrinsèque, chaque rayon pouvant être calculé indépendamment. Ainsi, le parallélisme de l'unité de traitement de fragments [6] est directement exploité en associant un rayon à chaque pixel. Il suffit de dessiner un polygone qui va générer un fragment pour chaque pixel de l'écran d'où un rayon doit être lancé, un programme de fragment se chargeant de calculer l'intégrale de l'équation de l'intégrale de rendu volumique. La boucle est gérée au niveau du CPU par l'application qui génère autant de passes de rendu qu'il y a d'itérations. Comme pour les techniques implicites, les données volumiques sont stockées sous forme de textures 3D en mémoire graphique. D'autre part, l'information nécessaire à chaque rayon au cours de la propagation (couleur/opacité accumulée, ...) est stockée dans des textures 2D avec une correspondance directe entre texels et pixels de l'écran, ce qui permet au programme de fragment d'y accéder lors du rendu. À noter que le programme accède à ces textures en écriture, ce qui en pratique est mis en œuvre à l'aide de l'extension OpenGL `EXT_framebuffer_object` [Kilgard 2003] ou de la fonctionnalité *render-to-texture* de Direct3D [Corporation 2002]. D'autre part, le matériel graphique ne supporte pas l'accès au textures simultanément en lecture et en écriture dans une même passe de rendu. Pour cette raison, une technique dite de *ping-pong* utilise deux copies de chaque texture 2D, l'une accédée en lecture, l'autre en écriture, leur rôle étant inversé à chaque nouvelle passe. Les optimisations classiques du lancer de rayons telles que la terminaison précoce de rayon et l'accélération dans les espaces vides peuvent également être mises au point dans ce contexte. La première repose sur l'utilisation du z-buffer tandis que l'équivalent des structures de données hiérarchiques de la seconde peut être stocké dans des textures 3D.

1.4 Comparaison des algorithmes de rendu de volume directe

Chaque approche de rendu de volume possède ses propres avantages et inconvénients. Généralement, lorsque l'approche de rendu vise à réaliser des images de haute qualité, elle doit perdre un peu de performance, et vice versa. *Shear warp* et *placage de texture 3D* sont conçus pour maximiser les fréquences d'images sur la détriment de la qualité de l'image, tandis que *splatting* et *raycasting* sont conçus pour obtenir une haute qualité d'image sur détriment de

[6] pixel pipeline

la performance.

Habituellement, la qualité d'image obtenue avec un placage de texture montre de graves artefacts de couleur-hémorragie dus à la non-opacité des couleurs pondérées, ainsi que l'escalier. Celui-ci est dû à la précision de bits limitée de la tampon d'image, et peut être réduite en augmentant le nombre de tranches. Certaines approches ont été proposées pour améliorer la qualité de rendu de volume basée sur la texture d'image [Rezk-Salama *et al.* 2000b, Engel *et al.* 2001]. L'image créée à l'aide de *shear-warp* montre un petit aliasing de flou significatif sous la forme d'escalier. Cela est dû à la fréquence d'échantillonnage de rayons étant inférieur à 1,0 et peut être gênant dans la visualisation d'animation. *Splatting* aligné à l'image offre une qualité de rendu similaire à celle du *raycasting*. Cependant, il produit des images plus lisses due au noyau de z-moyenne et l'effet anti-aliasing du la plus grand filtre gaussien. Les deux méthodes : *placage de texture 3D* et *shear-warp* sont toujours sensiblement plus rapides que ceux de *raycasting* et *Splatting*, dans la plupart des cas par un ordre de un ou deux amplitudes [Meissner *et al.* 2000]. *Splatting* peut générer une image de haute qualité plus rapidement que *raycasting* lors de la visualisation des données de grand volume, et ne provoque pas le flou extensif de *shear-warp*. Contrairement à *raycasting*, *splatting* considère chaque voxel une seule fois (pour une interpolation 2D sur l'écran) et non plusieurs fois (pour une interpolation 3D). En outre, comme une approche basée objet-ordre, seuls les voxels concernés doivent être pris en considération, qui, dans de nombreux cas, ne représentent que 10 % des voxels de volume [Wilhelms & Van Gelder 1991].

Le tableau 1.1 établit une comparaison des caractéristiques distinctives et les différences conceptuelles des algorithmes typiques de rendu de volume.

	Raycasting	**Splatting**	**Shear-Warp**	**placage de texture 3D**
Taux d'échantillonnage	sélection libre	sélection libre	fixé $[1.0, 0.58]$	sélection libre
évaluation d'échantillon	post-classification	moyenne à travers de Δs	point échantillonné	point échantillonné
Noyau d'interpolation	trilinéaire	gaussienne	bilinéaire	trilinéaire
Pipeline de rendu	post-classification	post-classification	Pré-classification, couleurs d'opacité pondérés	Pré/Post classification, pas de couleurs d'opacité pondérée
Accélération	la terminaison précoce de rayon	la terminaison précoce de rayon	encodage RLE d'opacité	Matériel graphique
Précision/Canal	Virgule flottante	Virgule flottante	Virgule flottante	8–12 bits
voxels considérés	Tout	Pertinent	Pertinent	Tout
Vitesse	$+^a$	+	++	++
Qualité	+++	+++	++	++
Projection Perspective	Y^b	Y	Y	Y
Irréguliers-grilles	Y	Y	N	Y
Accélération matériel	spécial	Y	N	Y
combinaison avec polygones	spécial	N	N	facilement
Informations richesse	++	++	++	++

$+^a$: général, ++ : bon, +++ : très bon ;
Y^b : Oui, N : Non ;

TABLE 1.1 – La comparaison des différents algorithmes de rendu de volume [Meissner *et al.* 2000]

1.5 Conclusion et Motivation

L'algorithme *raycasting* est avantageux par rapport à d'autres techniques de visualisation de volume interactif en raison de sa qualité d'image élevée, la flexibilité inhérente, et simple mise en œuvre sur les GPUs programmables. Les implémentations appliquent habituellement des techniques de programmation de GPU à usage général (GPGPU), qui ignorent la plupart des fonctionnalités de la géométrie du matériel et utilisent des fragment shaders pour effectuer le *raycasting* à travers l'ensemble de données de volume. Les GPUs Modernes supportent le traitement de flux comme un modèle de programmation alternative aux API graphiques classiques telles que OpenGL. Ces modèles de traitement de flux, e. g., OpenCL ou CUDA de NVIDIA, donnent un accès plus général au matériel et également prennent en charge certaines caractéristiques matérielles qui ne sont pas disponibles via les API graphiques, telles que la mémoire sur puce commune. Les cartes graphiques modernes sont devenues programmables avec ces langages haut niveau ce qui leur permet d'exécuter de petits programmes pour chaque pixel de l'image finale. Leur architecture est très parallèle à 16 ou même 24 pipelines de pixel de travail en même temps. Vu que la vitesse de *raycasting* est directement influencée par quantité de pixels que nous pouvons traiter simultanément, il serait donc logique d'utiliser le GPU à cet effet.

CHAPITRE 2

PROPOSITION D'UNE APPROCHE DE RENDU VOLUMIQUE BASÉE SUR GPU

2.1 Introduction

Raycasting est la méthode d'ordre image le plus populaire pour le rendu de volume. L'idée de base est d'évaluer directement l'intégrale de rendu de volume le long de rayons qui sont traversés par la caméra. Pour chaque pixel de l'image, un seul rayon est lancé à travers le volume. Ensuite, les données de volume sont ré-échantillonnée à des positions discrètes le long du rayon. La figure 2.13 illustre le raycasting. L'ordre de parcours naturel est avant-vers-arrière parce que les rayons sont conceptuellement à partir la caméra. Raycasting est le procédé le plus important pour le rendu de volume de l'unité centrale ; elle a été utilisée depuis un certain temps, et plusieurs procédés d'accélération ont été développés. GPU raycasting est assez nouveau, car les GPU antérieures ne prennent pas en charge la fonctionnalité requise pour lancer de rayons. GPU raycasting présente les avantages liés au fait qu'il peut être facilement étendue pour bénéficier de techniques d'accélération et qu'il prend en charge les grilles uniformes et des grilles tétraédriques. Par conséquent, GPU raycasting est déjà devenu très populaire dans une courte période de temps, s'il est logique de supposer que le raycasting jouera un rôle plus important que les GPUs encore évoluer.

Dans cette section, nous donnons un aperçu structurel de l'algorithme de rendu de volume raycasting. Cette technique est généralement utilisé par une approximation discrète (section 2.2.2.) de l'intégrale de rendu de volume (section 2.2.1).

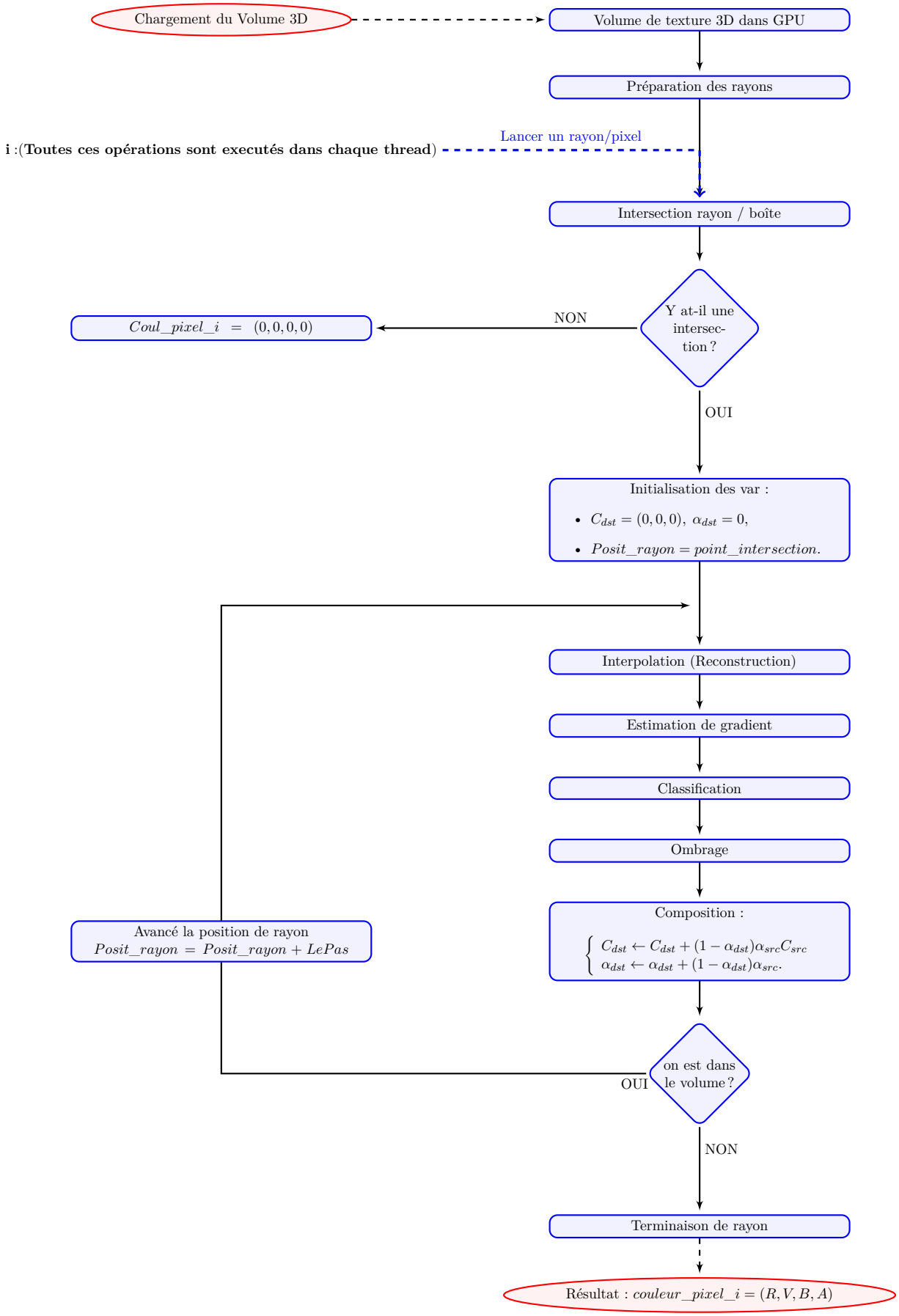


FIGURE 2.1 – Pipeline de rendu de volume GPU raycasting.

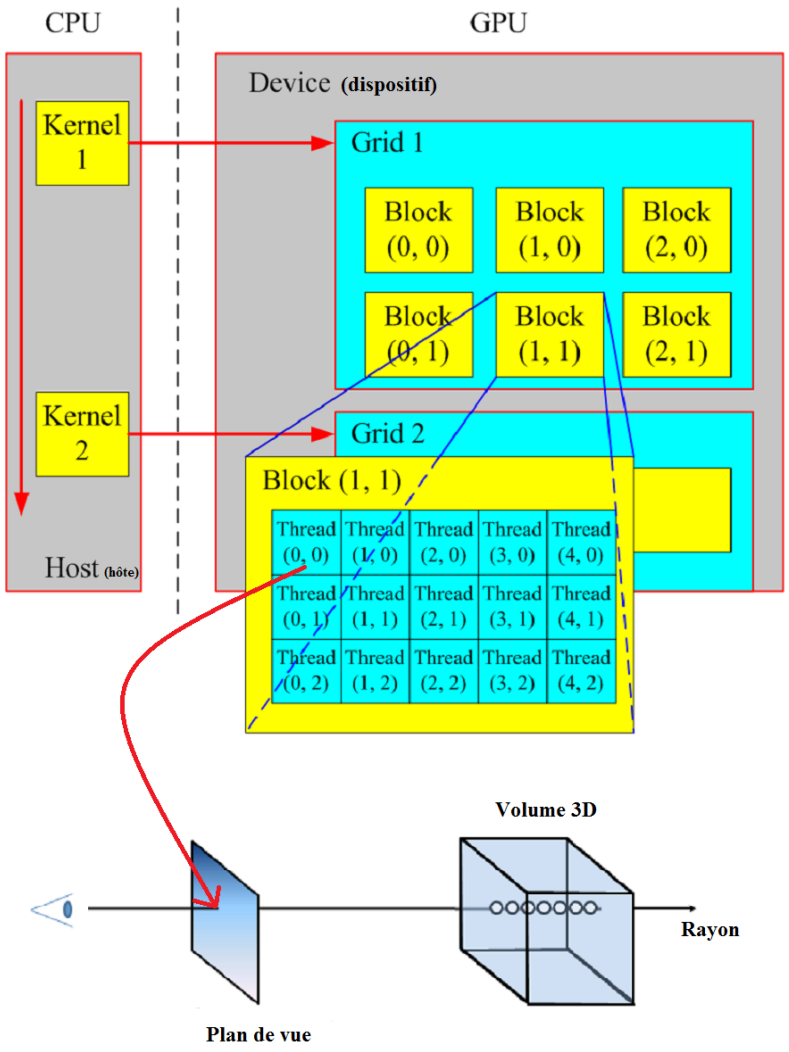


FIGURE 2.2 – Pipeline de rendu de volume GPU raycasting sur l'API d'opencl.

2.2 Équation de rendu de volume

2.2.1 L'intégrale de rendu de volume

L'équation de rendu de volume dans sa forme différentielle peut être résolue en intégrant le long de la direction du flux lumineux à partir de $s = s_0$ du point de départ au point de terminaison $s = D$, conduisant à l'intégrale de rendu de volume.

L'expression décrivant la progression de l'intensité lumineuse portée par un rayon est appelée l'intégrale de rendu volumique. Pour l'obtenir, on définit tout d'abord la variation d'intensité lumineuse le long d'un rayon :

$$dI = -I_{absorbee} + I_{emise} \tag{2.1}$$

De là, on peut en déduire une équation différentielle, qui développée s'écrit alors sous la forme suivante (l'obtention et le développement est décrit par Max [Max 1995]) :

$$I(D) = I_0 e^{-\int_{s_0}^{D} k(t)dt} + \int_{s_0}^{D} q(s) e^{-\int_{s_0}^{D} k(t)dt} ds \tag{2.2}$$

- $I(D)$: est l'intensité de la lumière à la distance D, ce qui représente la quantité de lumière venant de rayon de direction r qui est reçu à l'emplacement s_i sur le plan d'image,
- s_0 : le point où le faisceau pénètre dans le volume,
- $q(s) = c(s)k(s)$,
- $k(s)$ et $q(s)$ et $c(s)$: sont respectivement l'absorption ,l'émission et la couleur émise au point s,
- $e^{-\int_{s_0}^{D} k(t)dt}$: la transparence du milieu entre s_0 et D,
- $I_0 e^{-\int_{s_0}^{D} k(t)dt}$: décrit l'intensité initiale. Cette partie modélise donc l'absorption de la lumière (ou plutôt la lumière non absorbée),
- $\int_{s_0}^{D} q(s) e^{-\int_{s_0}^{D} k(t)dt} ds$: l'ensemble de la lumière émise en chaque point sur le rayon de lumière en tenant compte du facteur d'atténuation.

Le terme I_0 représente la lumière entrant dans le volume de l'arrière-plan dans la position de $s = s_0$

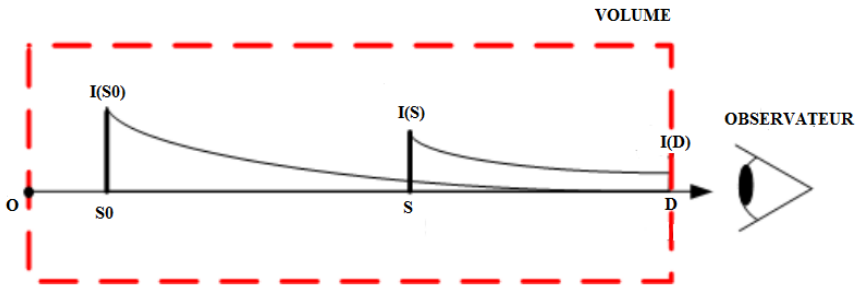


FIGURE 2.3 – L'intégral de rendu de volume.

, $I(D)$ est le rayonnement quittant le volume à $s = D$ et atteignant la caméra.
Le premier terme de l'équation 2.2 décrit la lumière provenant de l'arrière-plan atténuée par le volume. Le second terme représente la contribution intégrale des termes sources atténuées par le milieu participant le long des distances restant à la caméra.

Le terme :

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} k(t)dt \tag{2.3}$$

est défini comme étant la profondeur optique entre des positions s_1 et s_2. La profondeur optique a une interprétation physique sous la forme d'une mesure de combien de temps la lumière peut voyager avant qu'il ne soit absorbé ; c'est à dire, la profondeur optique indique la longueur typique de propagation de la lumière avant la diffusion se produit. Les petites valeurs de la profondeur optique signifient que le milieu est plutôt transparent, et des valeurs élevées pour la profondeur optique sont associés à un matériau plus opaque. La transparence correspondant (pour un matériau entre s_1 et s_2) est :

$$T(s_1, s_2) = e^{-\tau(s_1,s_2)} = e^{-\int_{s_1}^{s_2} k(t)dt} \tag{2.4}$$

Avec cette définition de la transparence, on obtient une version légèrement différente de l'intégrale de rendu de volume :

$$I(D) = I_0 T(S_0, D) + \int_{s_0}^{D} q(s)T(s, D)ds \tag{2.5}$$

L'intégrale de rendu de volume est la description la plus commune de rendu de volume.

2.2.2 Discrétisation

L'objectif principal de rendu de volume est de calculer l'intégrale de rendu de volume de l'équation 2.2. Typiquement, l'intégrale ne peut être évaluée de manière analytique. Au lieu de cela, les méthodes numériques sont appliquées pour trouver une approximation aussi proche que possible de la solution.

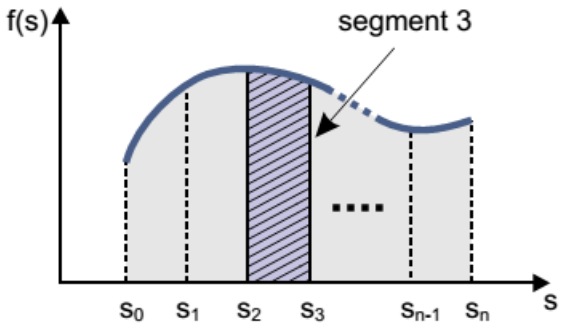


FIGURE 2.4 – Partitionnement du domaine d'intégration en plusieurs intervalles. Les intervalles sont décrits par les positions $s_0 < s_1 < \ldots < s_{n-1} < s_n$. L'intervalle ou un ième segment est $[s_{i-1}, s_i]$. La zone hachurée indique le résultat d'intégration pour le troisième segment.

Fractionnement en plusieurs intervalles d'intégration

Une approche de calcul divise le domaine de l'intégration en n intervalles. Les intervalles sont décrits par les positions $s_0 < s_1 < \ldots < s_{n-1} < s_n$, où s_0 est le point du domaine d'intégration de départ et $s_n = D$ est le point final. Il vous fait noter que les intervalles n'ont pas nécessairement la même longueur. La figure 2.4 illustre le partitionnement du domaine d'intégration en plusieurs intervalles ou des segments.

Considérant le transport de la lumière dans l'i-ème intervalle $[s_{i-1}, s_i]$ (avec $0 < i \leq n$), nous pouvons obtenir l'intensité à l'emplacement s_i selon :

$$I(s_i) = I(s_{i-1})T(s_{i-1}, s_i) + \int_{s_{i-1}}^{s_i} q(s)T(s, s_i)ds \tag{2.6}$$

Nous introduisons une nouvelle notation pour la transparence et la couleur de l'i-ème intervalle :

$$T_i = T(s_{i-1}, s_i),\ c_i = \int_{s_{i-1}}^{s_i} q(s)T(s, s_i)ds \tag{2.7}$$

La zone hachurée sur la figure 2.4 illustre le résultat de l'intégration sur un intervalle. La radiance à l'endroit du volume de sortie est alors donnée par :

$$I(D) = I(s_n) = I(s_{n-1})T_n + c_n = (I(s_{n-2})T_{n-1} + c_{n-1})T_n + c_n = \ldots, \tag{2.8}$$

Qui peut être écrite comme :

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j, \text{ avec } c_0 = I(s_0). \tag{2.9}$$

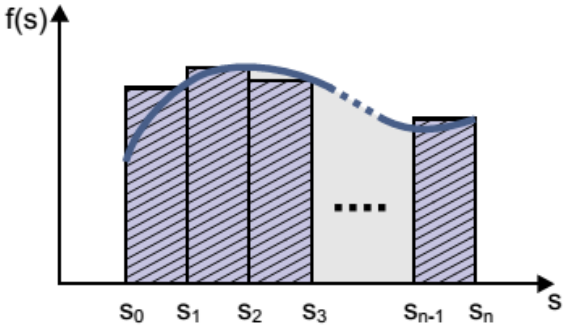


FIGURE 2.5 – Approximation d'une intégrale par une somme de Riemann.

En général, la notation $\sum_{i=k}^{l} a_i$ signifie que tous les termes a_i (pour $k \leq i \leq l$) sont additionnés. De même, $\prod_{i=k}^{l} a_i$ est une notation pour la multiplication de tous les termes a_i (pour $k \leq i \leq l$). Souvent, la transparence T_i est remplacé par l'opacité $\alpha_i = 1 - T_i$.

À ce point, nous avons résolu le problème du la moitié de calcul du rendu de volume : le domaine d'intégration est segmenté en n intervalles discrets, et les sommes ou multiplications dans l'équation 2.9 peuvent être calculée. Le point manquant est l'évaluation de la transparence et la couleur des contributions des intervalles.

Une approche plus courante rapproche l'intégrale de rendu de volume à une somme de *Riemann* sur n segments équidistants de longueur $\Delta x = (D - s_0)/n$. Ici, la fonction à intégrer est approximée par une fonction constante par morceaux, comme illustré sur la figure 2.5. L'intégrale sur un seul intervalle correspond à l'aire du rectangle défini par la valeur de fonction en un point d'échantillonnage et par la largeur de l'échantillonnage (voir les cases hachurées sur la figure 2.5).

Dans cette approximation, la transparence du l'i-ème segment est :

$$T_i = e^{-k(s_i)\Delta x} \tag{2.10}$$

Et la contribution de la couleur pour l'i-ème segment est :

$$C_i = q(s_i)\Delta x. \tag{2.11}$$

Systèmes de composition

La composition est la base pour le calcul itératif de l'intégrale de rendu de volume discrétisé (équation 2.9). L'idée est de diviser les sommes et des multiplications contenus dans l'équation 2.9 en plusieurs, encore plus simples qui sont exécutées séquentiellement. Deux schémas de composition de base différents existent : la composition avant-vers-arrière et la composition arrière-vers-avant.

Le schéma de composition d'avant vers arrière est appliqué lorsque les rayons de visée sont traversés à partir du point de l'œil dans le volume. Ici, nous utilisons des noms légèrement différents des variables : C représente une couleur RVB, généralement donné avec trois valeurs (rouge, vert, bleu). Alors que la radiance C nouvellement contribué et la radiance I cumulé à partir des sections précédentes sont maintenant associés à une telle description de la couleur.

Ensuite, les équations d'itération avant vers l'arrière sont :

$$\widehat{C}_i = \widehat{C}_{i+1} + \widehat{T}_{i+1} C_i \ ,$$
$$\widehat{T}_i = \widehat{T}_{i+1}(1 - \alpha_i) \ ,$$

Avec l'initialisation :

$$\widehat{C}_n = C_n \ ,$$
$$\widehat{T}_n = 1 - \alpha_n \ .$$

Les résultats de l'étape d'itération actuelle sont $\widehat{C}_i$ et $\widehat{T}_i$; $\widehat{C}_{i+1}$ et $\widehat{T}_{i+1}$ sont les résultats accumulés des calculs précédents. Le terme source C_i et l'opacité α_i sont donnés par la fonction de transfert. L'itération commence à la première position d'échantillonnage $i = n$ (proche de la caméra) et se termine à $i = 0$ (à l'arrière du volume).

En renommant les variables selon $C_{dst} = \widehat{C}_j$ (avec $j = i, i+1$), $C_{src} = C_i$, $\alpha_{dst} = 1 - \widehat{T}_j$ (avec $j = i, i+1$), et $\alpha_{src} = \alpha_i$, la composition avant-vers-arrière peut être écrite dans sa forme la plus commune :

$$\boxed{\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \ , \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \ . \end{aligned}} \tag{2.12}$$

Cet ensemble d'équations d'affectation montre explicitement la nature itérative de composition. Variables avec indice *src* (comme pour "source") décrivent les quantités introduites comme entrées des propriétés optiques de l'ensemble de données (par exemple, grâce à une fonction de transfert), alors que les variables avec indice *dst* (comme pour "destination") décrivent les quantités de sortie qui détiennent couleurs et opacités accumulés. L'équation 2.12 est appliquée à plusieurs reprises tout en marchant le long d'un rayon, la mise à jour de couleur C_{dst} et l'opacité α_{dst} long de son chemin.

En inversant le sens de parcours, on obtient le schéma de la composition arrière vers l'avant :

$$\begin{aligned} \widehat{C}_i &= \widehat{C}_{i-1}(1 - \alpha_i) + C_i \ , \\ \widehat{T}_i &= \widehat{T}_{i-1}(1 - \alpha_i) \ , \end{aligned}$$

Avec l'initialisation :

$$\begin{aligned} \widehat{C}_0 &= C_0 \ , \\ \widehat{T}_0 &= 1 - \alpha_0 \ . \end{aligned}$$

L'itération commence à $i = 0$ et se termine à $i = n$. A noter que la transparence accumulé $\widehat{T}_i$ n'est pas nécessaire de calculer la contribution $\widehat{C}_i$ de couleur et peut donc être omise.

De manière analogue au schéma avant-vers-arrière, nous pouvons également réécrire la composition arrière vers l'avant dans un mode itératif explicitement :

$$\boxed{C_{dst} \leftarrow (1 - \alpha_{src})C_{dst} + C_{src} \ .} \tag{2.13}$$

Couleurs associées

Jusqu'à présent, nous avons supposé les couleurs associées, tel que présenté par **Blinn** [Blinn 1994]. Les couleurs associées sont constituées de composantes de couleurs qui sont déjà pondérées par leur opacité correspondante. Une autre description utilise des composants de couleurs qui n'ont pas été pré multipliées avec opacité. Les équations précédentes doivent être modifiées pour permettre des couleurs

non associées : termes de couleurs originales doivent être remplacés par des termes de couleurs qui sont explicitement pondérés par l'opacité.

Par exemple, C_{src} doit être substituée par $\alpha_{src}C_{src}$ dans les équations de composition itératives. Avec des couleurs non associées, l'équation de composition avant vers l'arrière 2.12 est remplacé par :

$$C_{dst} \leftarrow C_{dst} + (1 - \alpha_{dst})\alpha_{src}C_{src}$$
$$\alpha_{dst} \leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} .$$

De même, le schéma de composition d'arrière vers l'avant est modifié à partir de l'équation 2.13 :

$$C_{dst} \leftarrow (1 - \alpha_{src})C_{dst} + \alpha_{src}C_{src} .$$

2.3 Algorithme de RayCasting sur les CPUs

Le but de raycasting est le calcul (approximative) de l'intégrale de rendu de volume. Pour tous les rayons de visualisation au niveau du plan d'image et passant par le volume. Ces évaluations sont réalisées afin de faire avancer les rayons à travers le volume avec des incréments Δs réguliers (c'est à dire la distance de l'échantillon de l'objet) et effectuer les étapes suivantes à chaque emplacement qu'on peut le voir sur la figure 2.6 :

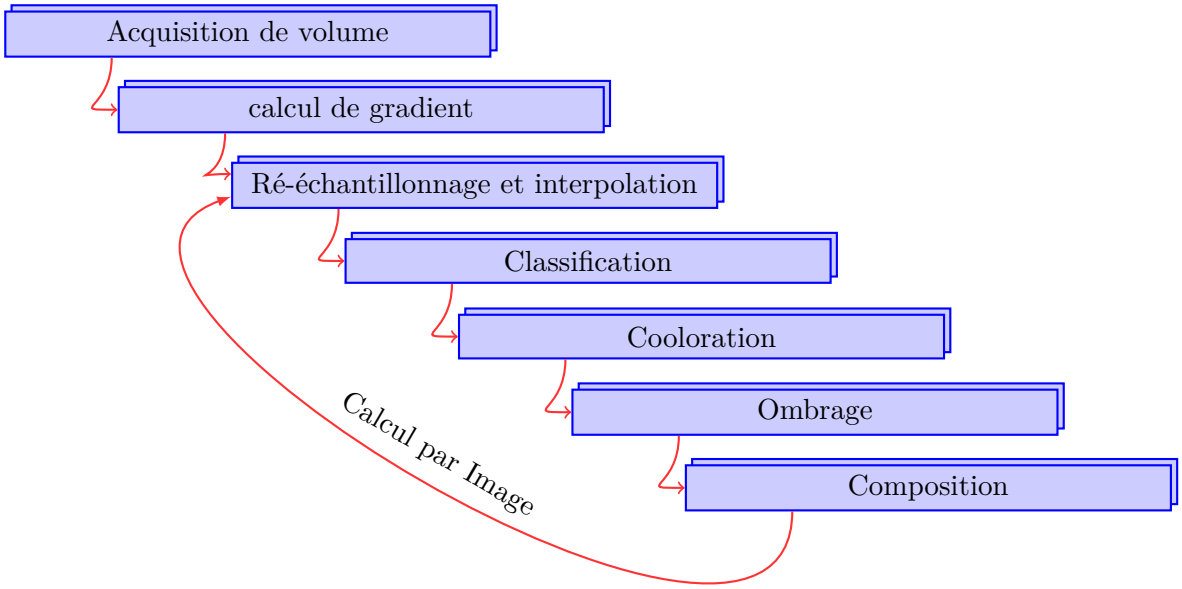


FIGURE 2.6 – Le pipeline de raycasting

2.3.1 Sources de données et acquisition de Volume

Les données pour le rendu de volume peuvent provenir d'une variété de différents domaines d'application. Un type d'application important est la visualisation de données scalaires scientifique. Plus précisément, l'imagerie médicale a été l'un des premiers domaines qui ont adopté le rendu de volume. En imagerie médicale, les données 3D est généralement acquis par une sorte de périphérique de numérisation (CT (computerized tomography) ,MRI (magnetic resonance imaging)).

2.3.2 Le calcul de gradient

La prochaine étape du pipeline est le calcul de gradient. Cette étape est responsable de trouver les bords ou les frontières entre les différents matériaux dans l'ensemble de données. Le gradient est un vecteur 3D contenant l'orientation et l'amplitude révèle que la quantité de variation entre un voxel et ses voxels voisins. Les gradients pour tous les voxels peuvent être calculées à l'aide de nombreuses méthodes différentes. Certaines méthodes de gradient couramment utilisés sont l'estimateur de gradient au Central-Différence, l'opérateur de différence intermédiaire est l'opérateur *Sobel* [Levoy 1988, Möller *et al.* 1996]. La différence centrale et intermédiaire n'utilise que six des voxels voisins pour le calcul du gradient et elle est relativement facile à mettre en œuvre. Cela permet aux deux méthodes d'être exécutées rapidement pour l'évaluation de gradient continu pour chaque image rendue. Toutefois, aucune de ces méthodes n'est pas un estimateur de gradient très précis. L'opérateur de *Sobel* est beaucoup plus précis, car il utilise les 26 voxels voisins pour calculer le gradient au détriment de l'efficacité de calcul. Il est préférable d'utiliser cet opérateur dans les cas où le gradient pour chaque voxel est calculé seulement une fois et stocké dans la mémoire au lieu d'être calculée pour chaque image. Une fois que les gradients sont calculés pour chaque voxel dans l'ensemble de données, l'information peut être réutilisé à l'étape de classification et d'ombrage.

2.3.3 La reconstruction (L'interpolation)

À la fin de l'étape de l'acquisition de volume et le calcul de gradient, le rendu peut commencer. La première étape de rendu est le ré-échantillonnage ou l'interpolation. Dans cette étape, les rayons sont émis à partir de chaque pixel des coordonnées d'écran dans la direction de vue à travers la scène 3D. Les rayons qui n'intersecte pas avec le volume devront simplement rendue la couleur de fond. Les autres rayons vont commencer l'échantillonnage à la première intersection avec le volume, ou au f_i comme le montre la Figure 2.7. Des échantillons supplémentaires seront ensuite prises et accumulés à des intervalles déterminés le long du rayon jusqu'à ce qu'il quitte le volume à I_i.

Malheureusement, l'emplacement de l'échantillon est corrélée rarement à un emplacement exact de voxel. Pour cette raison, des méthodes d'interpolation sont utilisées pour générer des valeurs approximatives pour les échantillons qui se situent entre un groupe de voxels. Certaines méthodes couramment utilisées sont : plus proche voisin, l'interpolation trilinéaire, B-splines et l'interpolation tricubic [He & Kaufman 1993, Keys 1981]. La complexité de calcul de chacun de ces procédés en trois dimensions peut être vu dans le tableau 2.1. Plus proche voisin est la méthode la plus rapide, mais produit également les plus mauvais résultats car elle n'effectue aucune interpolation. La convolution tricubic

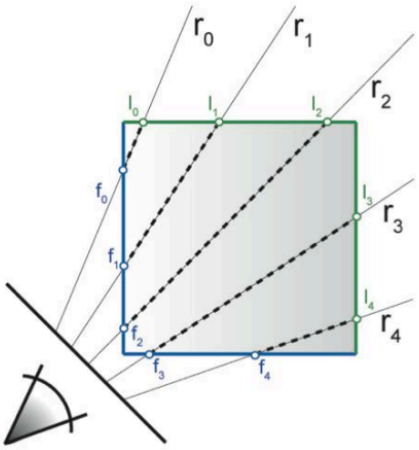


FIGURE 2.7 – Diagramme de raycasting en 2D où chaque rayon est lancé à partir du point de vue dans une projection en perspective [Hadwiger *et al.* 2008].

et les méthodes B-splines produisent des résultats de qualité supérieure, mais ont un coût élevé de calcul [Lichtenbelt *et al.* 1998a].

	Plus proche voisin	Trilinear	Convolution Tricubic	B-spline
Multiplier	0	2	5	12
Ajouter / Soustraire	0	3	9	9

TABLE 2.1 – Nombre total de multiplications, additions et soustractions nécessaires pour chaque méthode d'interpolation en trois dimensions [Lichtenbelt *et al.* 1998a].

Interpolation trilinéaire

Pour les applications en temps réel, l'interpolation trilinéaire [WikipediaTrilInterp 2015] est souvent la méthode la plus raisonnable en raison de sa capacité à réduire les problèmes d'aliasing avec très peu de temps de calcul. L'interpolation trilinéaire suppose une relation linéaire entre un point d'interpolation et ses points voisins et peut être effectuée dans un ordre particulier, par exemple long de x, puis le long de y, et enfin long de z. Pour le démontrer, un voxel C peut être vu dans la figure 2.8 entre 8 voxels voisins. Tout d'abord, les quatre valeurs sur l'axe des x ont été calculées, C_{00}, C_{01}, C_{10} et C_{11}. Ensuite, les valeurs ont été interpolées sur l'axe des z produisant C_0 et C_1. Enfin, C_0 et C_1 ont été interpolées sur l'axe des y pour produire la valeur résultante de C. Encore, ces opérations peuvent être calculées dans un ordre quelconque, et seront toujours produire le même résultat.

L'interpolation trilinéaire est l'extension de l'interpolation linéaire, qui fonctionne dans des espaces de dimension $D = 1$, et l'interpolation bilinéaire, qui fonctionne avec la dimension $D = 2$, à dimension

$D = 3$. La précision est de l'ordre 1 pour toutes ces techniques d'interpolation, et il exige $(1+n)^D = 8$ valeurs adjacents prédéfinies qui entourent la point d'interpolation. Il y a plusieurs façons d'arriver au interpolation trilinéaire, il est équivalent à l'interpolation de trois dimensions B-spline d'ordre 1, et l'opérateur d'interpolation trilinéaire est aussi un produit de tenseur des opérateurs d'interpolation trilinéaires.

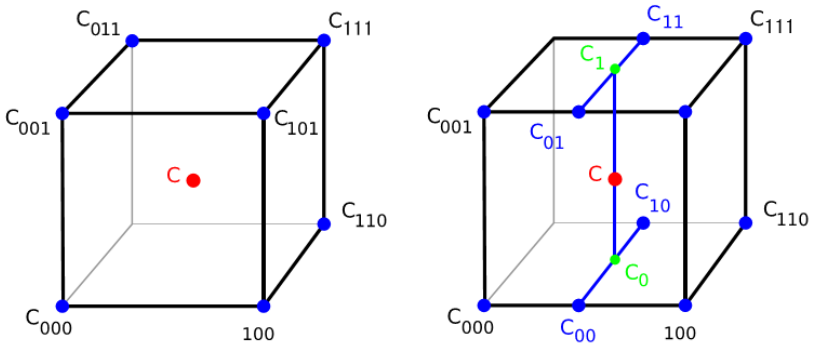


FIGURE 2.8 – Schéma d'un seul voxel C entouré par huit voxels voisins (gauche). Comment l'interpolation trilinéaire peut être utilisé pour calculer la valeur de C (à droite)

La méthode

Sur un réseau périodique et cubique, laisser x_d, y_d, et z_d être les différences entre chacun de x, y, z et la plus petite coordonnée associée, qui est :

$$\begin{cases} x_d = (x - x_0)/(x_1 - x_0) \\ y_d = (y - y_0)/(y_1 - y_0) \\ z_d = (z - z_0)/(z_1 - z_0) \end{cases} \tag{2.14}$$

où x_0 indique le point de treillis ci-dessous x, x_1 et indique le point de treillis ci-dessus x, et de même pour y_0, y_1, z_0 et z_1.

Nous avons d'abord interpolons le long de x (imaginons que nous poussons de la face avant du cube vers l'arrière), ce qui donne :

$$\begin{cases} c_{00} = V[x_0, y_0, z_0](1 - x_d) + V[x_1, y_0, z_0]x_d \\ c_{10} = V[x_0, y_1, z_0](1 - x_d) + V[x_1, y_1, z_0]x_d \\ c_{01} = V[x_0, y_0, z_1](1 - x_d) + V[x_1, y_0, z_1]x_d \\ c_{11} = V[x_0, y_1, z_1](1 - x_d) + V[x_1, y_1, z_1]x_d \end{cases} \tag{2.15}$$

Où $V[x_0, y_0, z_0]$ signifie la valeur de fonction de (x_0, y_0, z_0). Ensuite, nous interpolons ces valeurs (selon y, que nous poussions le bord haut vers le bas), ce qui donne :

$$\begin{cases} c_0 = c_{00}(1 - y_d) + c_{10}y_d \\ c_1 = c_{01}(1 - y_d) + c_{11}y_d \end{cases} \tag{2.16}$$

Enfin, nous interpolons ces valeurs selon z (marche à travers une ligne) :

$$c = c_0(1 - z_d) + c_1 z_d \tag{2.17}$$

Cela nous donne une valeur prévue pour le point.

Le résultat de l'interpolation trilinéaire est indépendant de l'ordre des étapes d'interpolation le long des trois axes : toute autre ordonnance, par exemple le long x, puis le long de y, et enfin le long z, produit la même valeur.

Les opérations ci-dessus peuvent être visualisés comme suit : d'abord, nous trouvons les huit coins d'un cube qui entourent notre point d'intérêt. Ces coins sont les valeurs C_{000}, C_{100}, C_{010}, C_{110}, C_{001}, C_{101}, C_{011}, C_{111}.

Ensuite, nous effectuons l'interpolation linéaire entre C_{000} et C_{100} pour trouver C_{00}, C_{001} et C_{101} pour trouver C_{01}, C_{011} et C_{111} pour trouver C_{11}, C_{010} et C_{110} pour trouver C_{10}.

Maintenant, nous faire l'interpolation entre $C00$ et $C10$ pour trouver $C0$, $C01$ et $C11$ pour trouver $C1$. Enfin, nous calculons la valeur C par l'interpolation linéaire entre $C0$ et $C1$.

Dans la pratique, une interpolation trilinéaire est identique à trois interpolations linéaires successives, ou une interpolation bilinéaire associée à une interpolation linéaire :

$$C \approx L(b(C_{000}, C_{010}, C_{100}, C_{110}), b(C_{001}, C_{011}, C_{101}, C_{111})) \tag{2.18}$$

2.3.4 Classification

Après le calcul de l'intensité en utilisant l'interpolation de voxel échantillonné, l'étape suivante consiste à déterminer si ce voxel va faire partie du voxel à rayons accumulée. Cette étape du processus de rendu est appelée classification. La classification est l'un des outils les plus puissants dans le rendu de volume, car il permet à certaines structures pour être visualisées, même s'ils peuvent être obstrués par d'autres objets. Ceci est réalisé par la création d'une correspondance entre l'intervalle d'intensités de voxel et des valeurs d'opacité entre zéro et un. L'opacité est une mesure de la façon dont un objet est translucide. En attribuant une valeur d'opacité à chaque voxel échantillonné, certaines structures dans l'ensemble de données peuvent être ignorées si la valeur d'opacité est zéro. D'autre part, si l'opacité du voxel n'est pas zéro, le voxel passe à être coloré dans la phase de coloration du pipeline de rendu de volume.

La mise en correspondance entre l'intensité de voxel et l'opacité est généré par une fonction de transfert d'opacité [Levoy 1988, Drebin *et al.* 1988]. Concevoir la fonction de transfert d'opacité peut être assez complexe, en fonction de ce type de structures qui doivent être visualisées. Les histogrammes sont un outil utile dans la conception de fonctions de transfert qu'ils révèlent où les intensités de haute fréquence dans le mensonge de données. Par conséquent, la fonction de transfert d'opacité peut être conçu en conséquence à exposer certaines parties de données.

2.3.5 Coloration

Pour attribuer une couleur à la voxel, rouge, vert et bleu (RVB) des fonctions de transfert (dénommés collectivement la fonction de transfert de couleur) sont utilisés pour mapper l'intensité de voxel à une valeur de couleur RVB. Autres propriétés de voxels peuvent être mappés à la couleur, comme la direction de gradient ou l'amplitude, mais le plus couramment utilisé est intensité de voxel. Il est important de noter que l'objectif de la fonction de transfert de couleur est d'améliorer la qualité visuelle pour interpréter les données volumétriques, pas à atteindre le photo-réalisme. Par conséquent, chacune des couleurs peut avoir sa propre fonction de transfert de couleur pour définir la manière dont chaque valeur d'intensité est rouge, vert et bleu. Ces valeurs RVB sont ensuite combinées pour produire la couleur finale pour le voxel. Généralement, chaque couleur utilise une fonction de transfert unique. Sinon, si elles sont toutes les mêmes, une image en niveaux de gris est produite.

Un exemple de rendu de volume en utilisant deux fonctions différentes de transfert de couleur peut être vu sur la figure 2.9. Il est également possible d'utiliser une approche plus automatisée pour la création de fonctions de transfert de couleur. [He *et al.* 1996] Ont utilisé des techniques de recherche

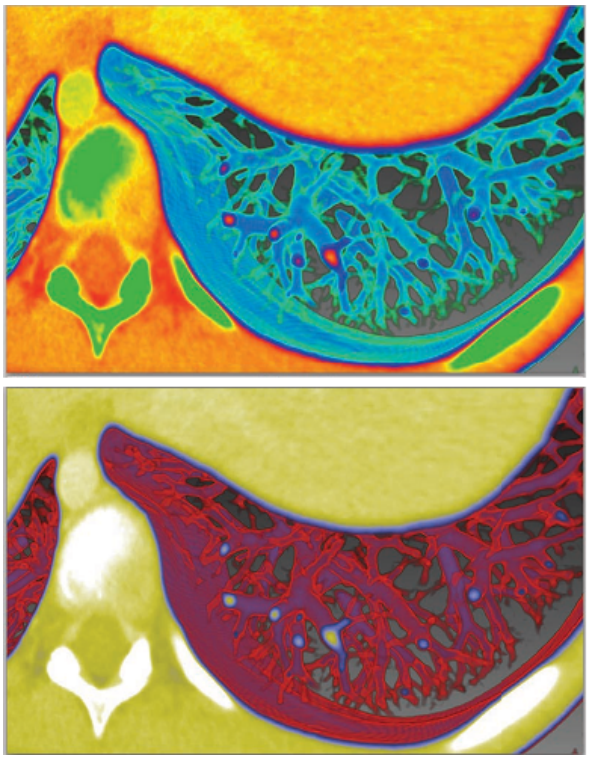


FIGURE 2.9 – La même image de rendu de volume en utilisant deux fonctions de transfert de couleur différentes à l'aide de la même fonction de transfert d'opacité.

stochastiques pour aider les utilisateurs à générer des fonctions de transfert automatisés. L'avantage de ceci est qu'une large plage de couleurs peut être appliquée sur une petite plage de valeurs d'intensité des voxels pour une meilleure distinction. Cela peut se faire manuellement, ce qui peut être difficile et prend du temps, ou automatiquement ce qui est plus facile et parfois plus efficace.

2.3.6 L'ombrage

Après que la couleur est affectée, il faut appliquer le modèle d'éclairage et d'ombrage à la couleur RVB de voxel. Le but du modèle d'illumination est de simuler la réflexion de la lumière d'une surface, et l'effet qu'elle a sur l'observateur lorsqu'il regarde cette surface. Par exemple, le cas d'une boule de billard noir, illuminer par un source blanche (lampe de proche), on remarque une tâche blanche dû à la lumière. Cet effet peut être observé sur la figure 2.10. L'interaction de la lumière à la surface de la boule de billard affecte la perception de la balle elle-même. Il nous permet de voir la forme exacte et le contour de la surface plus clairement.

FIGURE 2.10 – Une boule noir de billard dans une pièce sombre (gauche). Une boule noir de billard dans une pièce sombre éclairer avec une petite lampe de poche apparaît sur (à droite).

Afin d'appliquer un modèle d'illumination de l'ombrage, la première étape consiste à calculer le gradient du voxel échantillonnée avec l'une des méthodes d'interpolation utilisés dans la section 2.3.3. La méthode la plus couramment utilisée est interpolation trilinéaire. Le calcul de la normale est alors appliqué en combinaison avec le vecteur de lumière et la direction de vue à un modèle d'illumination de l'ombrage pour calculer la couleur RVB finale du voxel échantillonné. Les techniques d'ombrage les plus populaires dans le rendu de volume sont les modèles d'ombrage Phong [Phong 1975] et Gouraud [Gouraud 1971]. Les deux méthodes utilisent la lumière ambiante, la réflexion diffuse et la réflexion spéculaire en combinaison avec le vecteur de lumière incidente, vecteur normal et la direction de vue pour calculer la couleur RVB ombragée du voxel échantillonné quand il interagit avec la lumière. Enfin,

la couleur et l'opacité de voxel sont ensuite cumulés et l'échantillonnage continue.

2.3.7 La composition

Comme chaque rayon qui est lancé ne peut que représenter un seul pixel, chaque voxel échantillonné doit être accumulée en une couleur RVBA unique (le A dans RVBA signifie l'opacité de la couleur). C'est l'étape finale du pipeline de rendu de volume est appelé la composition. Pour combiner toutes les valeurs de voxel, la fonction d'accumulation avant-vers-arrière ou arrière-vers-avant est utilisée. La composition avant-vers-arrière est le plus couramment utilisée car elle offre des améliorations de performances. La fonction de composition avant-vers-arrière est représentée dans l'équation 2.19 :

$$I(x,y) = \sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} (1 - \alpha_j) \tag{2.19}$$

Dans l'équation 2.19, l'intensité totale pour le voxel (x, y) est la somme de I_i multiplié par toutes les transparences $(1 - a_j)$ rencontrés précédemment le long du rayon. L'intensité I_i est généralement représentée par l'équation 2.20 :

$$I_i = C_i \times \alpha_i \tag{2.20}$$

L'équation 2.20 montre que l'intensité I_i est une fonction de la couleur et l'opacité de point d'échantillonnage. La fonction de composition d'avant vers l'arrière évalue continuellement l'intensité du voxel courant échantillonné, puis se mélange avec le voxel accumulé, et ce processus se poursuit tandis que le rayon est encore contenu dans le volume. Un avantage majeur à la fonction de composition avant-vers-arrière est qu'on peut appliquer la méthode d'accélération de "la terminaison précoce de rayon". Où le ré-échantillonnage est arrêtée lorsque l'opacité de voxel accumulée atteint 1.0, ou une valeur suffisamment proche. La raison pour laquelle le ré-échantillonnage peut être terminée, c'est que le voxel ensuite échantillonné n'aura plus aucun effet sur la couleur de voxel accumulée. C'est une optimisation facile du rendu de volume qui peut être fait directement dans la fonction de composition. Pour plus de détails sur les fonctions de composition, on peut se référer à [Blinn 1982, Upson & Keeler 1988].

2.4 Architecture basée sur OpenCL

Systématiquement Le flux de traitement sur OpenCL, comme le montre la figure 2.11, comporte les étapes suivantes :

1. Copier les données de la mémoire principale à la mémoire du GPU,
2. CPU charge le processus pour le GPU,
3. Exécution parallèle dans chaque cœur du GPU,
4. Copier le résultat du mémoire GPU vers la mémoire principale.

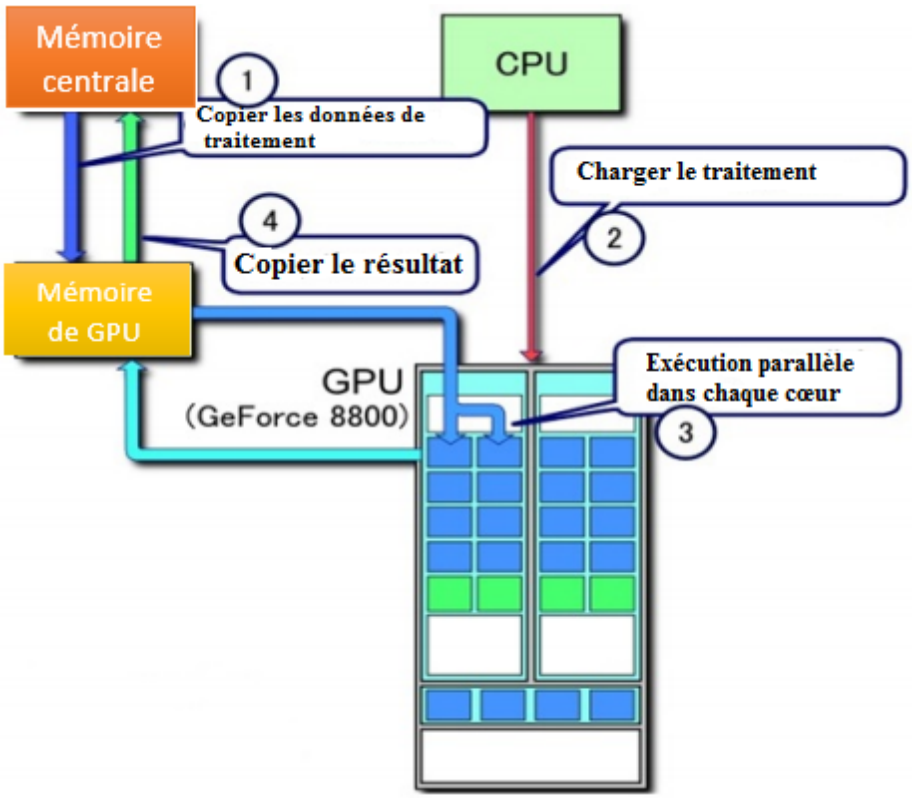


FIGURE 2.11 – Le flux de traitement sur OpenCL

2.5 Le modèle graphique

Dans le modèle de processus graphique le rendu GPU est un mode pipeline fixe. L'approche serait d'abord prendre un ensemble de données de volume et d'évaluer les sommets (*Vertex*). Puis une rastérisation est appliquée pour la segmentation des données. La rastérisation est utilisée pour définir

les données sous une forme de grille séquentielle. Après la segmentation puis cochez la cache de trame pour les voxels segmentées. Ceci est représenté sur la figure 2.12, pour plus de détails voir l'annexe B.

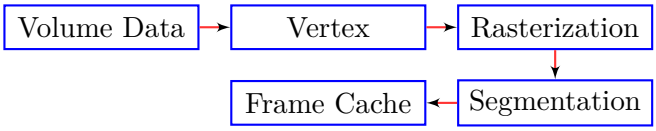


FIGURE 2.12 – Schéma du pipeline de rendu.

2.6 Pipeline de rendu de volume par GPU Raycasting

Raycasting est une méthode bien connue pour le rendu de volume à base de CPU qui a été utilisée des années 1980. D'autre part, le GPU raycasting est assez nouvelle, avec ses premières mises en œuvre, publié en 2003. La raison de la fin du développement de GPU raycasting est la demande pour la fonctionnalité de fragment shader de pointe qui n'était pas disponible auparavant. GPU raycasting s'appuie souvent sur des méthodes de CPU précédentes, essentiellement l'adoption de ces derniers pour le matériel graphique. L'objectif principal est de décrire comment le raycasting peut être réalisé sur des architectures GPU. L'idée de raycasting est d'évaluer directement l'intégrale de rendu de volume (voir le section 2.2.1 pour une discussion sur l'intégral de rendu de volume) le long de rayons qui sont traversés par la caméra. Pour chaque pixel de l'image, un seul rayon est tracé dans le volume. Ensuite, les données de volume sont ré-échantillonnées à des positions discrètes le long du rayon (la figure 2.13 illustre le raycasting). Au moyen de la fonction de transfert, les valeurs de données scalaires sont plaquées à des propriétés optiques qui sont à la base de l'information d'accumulation de lumière le long du rayon. Typiquement, la composition peut être effectuée dans le même ordre que le parcours des rayons. Par conséquent, la composition avant vers l'arrière (voir équation 2.12) est appliquée.

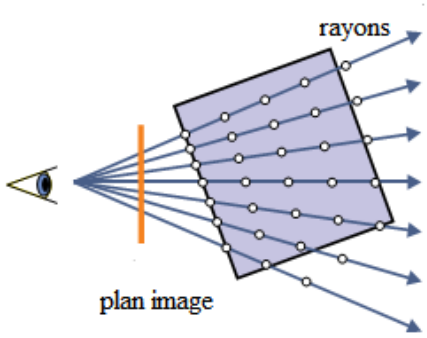


FIGURE 2.13 – Principe de Raycasting. Pour chaque pixel, un rayon de visualisation est tracé. Le rayon est échantillonné à des positions discrètes pour évaluer l'intégrale de rendu de volume.

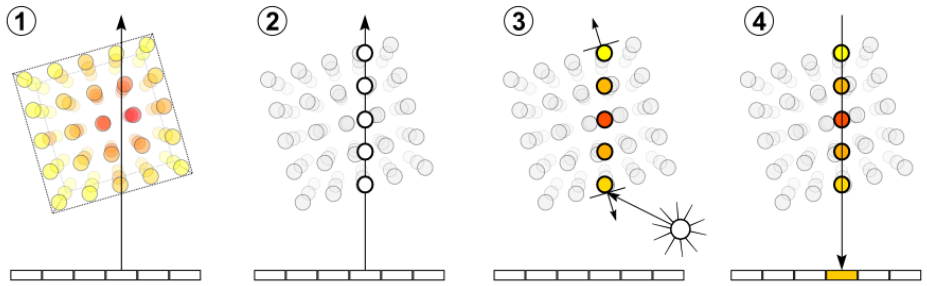


FIGURE 2.14 – Les étapes de raycasting : lancé un rayon par pixel, interpolation, classification, composition

$$\begin{aligned} C_{dst} &= C_{dst} + (1 - \alpha_{dst})C_{src} \ , \\ \alpha_{dst} &= \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \ . \end{aligned} \tag{2.21}$$

En conséquence, GPU raycasting peut être subdivisé en plusieurs composants majeurs suivants :

2.6.1 Préparation des rayons

Tout d'abord, un rayon de visualisation doit être mis en place en fonction des paramètres de la caméra et la position du pixel respective. Ce composant calcule la position d'entrée de volume et les coordonnées de l'intersection entre le premier rayon et l'ensemble de données de volume. Ce composant détermine également la direction du rayon.

Accès aux données

L'ensemble de données est accessible à la position du rayon courant, ce qui peut impliquer un filtre de reconstruction (c'est à dire l'interpolation). La couleur et l'opacité correspondant sont calculées en appliquant la fonction de transfert. Chacune des classification point par point ou classification pré-intégrée sont possibles.

Traversée de données

Les rayons sont tracés à partir du point de vue (aussi appelé centre de projection) par pixels à l'écran dans le volume. Les directions des rayons peuvent être calculées à partir des transformations de modèles et d'affichage utilisant des techniques standard d'infographie [Francis & Hill. 2000]]. L'espace vide entre le plan d'observation et le volume peut être ignorée en rendant un cadre de sélection polygonale du volume dans un tampon de profondeur [Avila *et al.* 1992]. Les points de départ des rayons et leurs directions normalisées sont ensuite utilisés pour générer des points de ré-échantillonnage espacés de façon égale le long de chaque rayon. Le volume est stocké dans la mémoire de texture 3D et échantillonné pendant le rendu et Les coordonnées de texture sont calculés par rapport aux paramètres de texture 3D dans l'intervalle $[0, 1]$.

2.6.2 Intersection rayon / boîte

Les intersections entre le rayon et les plans parallèles de la boîte englobante qui englobe le volume 3D : *Kay et Kayjia* [Kay] ont développé une méthode basée sur des « plaques » où une plaque est l'espace entre deux plans parallèles.

On recherche les intersections entre le rayon et chacune des plaques ($t_{i,min}$ et $t_{i,max}$).
On calcule T_{min} la plus grande des $t_{i,min}$ et T_{max} la plus petite des $t_{i,max}$.

- Si $T_{min} > T_{max}$ l'intersection est vide,
- Sinon l'intersection est le segment $[T_{min} T_{max}]$.

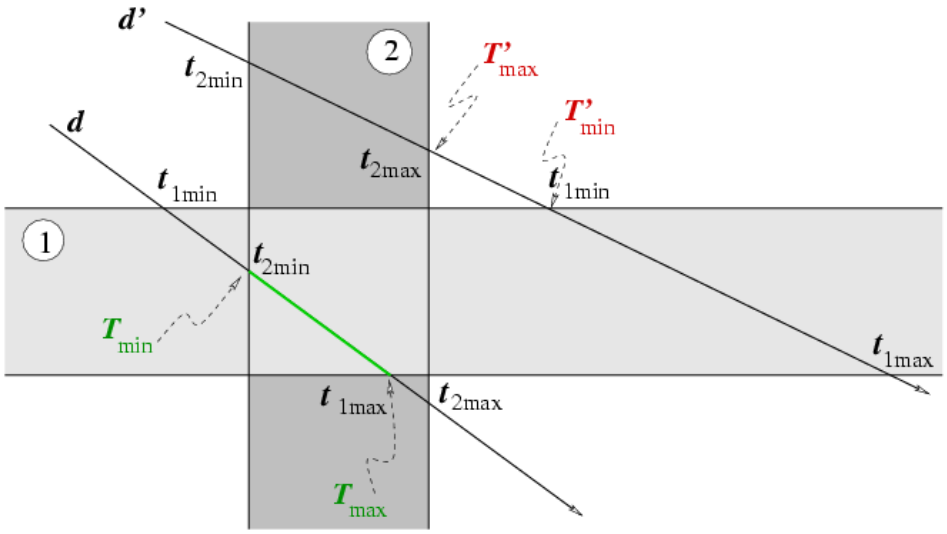


FIGURE 2.15 – Intersection rayon / boîte [Kay]

2.6.3 Évaluation de l'intégrale

Cette composante principale traversant le long du rayon pour l'évaluation de l'intégral de rendu de volume à travers plusieurs itérations. Le rayon est échantillonné à des positions discrètes et effectuer les opérations suivantes dans chaque point d'intersection avec le volume 3D :

Interpolation (Reconstruction)

A chaque emplacement de ré-échantillonnage, les données sont interpolées, généralement par interpolation trilinéaire dans les cellules de voxels $2 \times 2 \times 2$. Les données de volume sont automatiquement ré-échantillonnées par le matériel de texture 3D en utilisant une interpolation trilinéaire. A noter que l'interpolation trilinéaire peut être efficacement évaluée en utilisant la mise en cache des données de cellule entre les rayons voisins [Pfister *et al.* 1999] et que le taux d'échantillonnage le long des rayons de visualisation est constant.

Le calcul du gradient

Les vecteurs gradients sont généralement pré-calculées à des emplacements de voxels et stockées avec les données de volume. Sinon, ils sont calculés lors du rendu en utilisant le filtre de gradient de différence central. Les vecteurs gradients d'une cellule sont interpolées à l'emplacement de ré-échantillonnage les plus proches. En option, l'amplitude du gradient est calculé pour la modulation d'amplitude de gradient lors de la post-classification. Les GPUs actuels ne prennent pas en charge le calcul des gradients 3D dans le matériel. Les gradients sont, mise à l'échelle et biaisé dans l'intervalle $[0, 1]$, et stockée dans la texture 3D. Typiquement, le canal RVB est utilisée pour les gradients de volume, tandis que les valeurs scalaires sont stockés dans le canal alpha [Westermann & Ertl 1998]. Depuis les gradients sont interpolées de la même façon que les valeurs scalaires, les opérations d'ombrage suivants sont calculés par pixel dans l'espace de l'écran.

La Classification

La classification nous permet de distinguer les différents matières dans un volume. Il est généralement basé sur les fonctions de transfert. La fonction de transfert attribue généralement les propriétés optiques discrétisées sous la forme de la couleur C et l'opacité α. Si les valeurs scalaires ont été interpolées, la couleur et l'opacité sont attribuées à chaque échantillon en utilisant une recherche post-classification (post-classification lookup). Éventuellement, l'opacité de l'échantillon est modulé par l'amplitude du gradient. Si les données ont été pré-classifiés, aucune autre classification des couleurs interpolées associé est nécessaire. La plupart des méthodes utilisent la post-classification en stockant une fonction de transfert 1D ou plus comme une texture [Meissner *et al.* 1999]. La valeur scalaire interpolée est stocké en tant que texture, qui est ensuite utilisé comme une recherche de coordonnées (lookup coordinate) dans la fonction de transfert de texture. Il est également appelé dépendant de recherche de texture (texture lookup) car les coordonnées de texture pour la deuxième texture sont obtenues à partir de la première texture. Si pré-classification est utilisé, une de texture 3D avec des couleurs associées sont stockées en plus d'un gradient de texture 3D.

Ombrage et illumination

Utilisation du gradient comme une approximation de la normale de surface et la couleur classifiée comme la couleur émise (ou primaire), un modèle d'ombrage locale est appliquée à chaque échantillon. Par exemple, le modèle d'éclairage Phong pour les lumières directionnels peut être efficacement mis en œuvre à l'aide de pré-calcul et la table de consultation (table lookup) dans que l'on appelle

les cartes de réflectance [Voorhies & Foran 1994]. La mise en œuvre de réflexion de carte prend en charge un nombre illimité de sources de lumière directionnelles, mais pas de lumière de position. Avant l'introduction de matériel graphique programmable, l'ombrage des volumes stockés en tant que textures 3D a été ignoré ou effectué dans une étape de pré-traitement [Van Gelder & Kim 1996]. Cependant, les volumes pré-ombragée doivent être recalculée chaque fois que la position de la lumière et le changement de la direction d'observation. Un problème plus fondamental est que la classification et l'ombrage sont en général des opérations non linéaires. L'interpolation des valeurs pré-calculées dégrade la qualité de l'image par rapport à l'évaluation des fonctions non linéaires dans un champ scalaire interpolée linéairement [Klaus Engel 2003, Neumann 1993].

La Composition

La composition est la base pour le calcul itératif de l'intégrale de rendu de volume discrétisé. L'équation de composition dépend de l'ordre de traversée. Les équations d'itération d'avant vers l'arrière sont utilisées lorsque les rayons de visée sont tracés à partir du point de vue dans le volume. Tous les échantillons le long du rayon sont groupés en valeurs de pixels, typiquement en ordre d'avant vers l'arrière pour produire l'image finale. Pour une meilleure qualité d'image, plusieurs rayons par pixel sont jetés et combinés en utilisant des filtres d'image de qualité studio [Francis & Hill. 2000]. Raycasting offre une haute qualité d'image et il est conceptuellement facile à mettre en œuvre. Malheureusement, ces avantages sont obtenues au prix de conditions de calcul élevées. Les données de volume n'a pas été activé dans l'ordre de stockage en raison de la direction de traversée arbitraire des rayons d'observation. Cela conduit à une mauvaise localisation spatiale des références de données, en particulier pour l'échantillon et interpolation de gradient. La couleur et l'opacité accumulée précédemment sont mis à jour en fonction de la composition d'équation (Équation 2.21) avant-vers-arrière.

Avancer la position du rayon

La position des rayons courant est avancé à la prochaine station d'échantillonnage le long du rayon.

Terminaison du rayon

La boucle de parcours se termine lorsque le rayon quitte du volume d'ensemble de données. Cette sous-composante vérifie si la position des rayons actuelle est à l'intérieur du volume et il n'entre la prochaine itération de la boucle lorsque le rayon est toujours à l'intérieur.

2.7 Conclusion

Dans cet chapitre ont abouti à une nouvelle technique de raycasting implémentée sur le GPU via textures 3D se positionne comme un outil efficace pour l'affichage et l'analyse visuelle des champs scalaires volumétriques. Il est couramment admis que, pour définir les données de taille fixe de qualité approprié à des taux interactifs peut être réalisé au moyen de cette technique. Cette approche est beaucoup plus souple et peut être étendue dans un certain nombre de façons, ce qui est la partie principale de mémoire, et cela nous permet d'utiliser les avantages spécifiques d'un GPU a plus d'un CPU de la meilleure façon possible, à savoir :

- Une séparation en deux unités distinctes, le CPU (acquisition des données et la création d'ensemble des threads et l'affecter aux GPU), et le GPU (stockage des données dans la mémoire de texture 3D et l'exécution parallèle de ces threads de l'algorithme de raycasting pour doubler les performances),
- Le volume est stocké dans la mémoire graphique de texture 3D et échantillonné pendant le rendu,
- Une architecture massivement parallèle, donc ce qui est approprié d'utiliser la méthode de raycasting,
- Interpolation trilinéaire est automatiquement (et extrêmement rapide par le matériel du GPU) mis en œuvre dans le texture 3D,
- Instructions spécifiques pour les tâches graphiques,
- Opérations vectorielles sur quatre flottants *RVBA* qui sont aussi rapides que les opérations scalaires.

CHAPITRE 3

ALGORITHME ET MISE EN ŒUVRE

3.1 Introduction

Dans ce chapitre, nous implémentons l'algorithme GPU raycasting (évoqué dans la section 2.6) avec l'utilisation de la formule d'accumulation donnée dans l'équation 2.2

L'augmentation incessante de la puissance de calcul parallèle brute et de la bande-passante mémoire des GPUs, rend ceux-ci de plus en plus intéressantes pour implanter des algorithmes hautement parallèles. Ceci est d'autant plus vrai depuis l'apparition des API dédiées au calcul générique sur GPU telles que celle d'AMD-ATI appelée CTM *(Close-To-Metal)* ou bien celle de Khronos Group appelée OpenCl *(Open Computing Language)*. Ces nouvelles API ouvrent un accès direct de bas niveau aux nombreux cœurs de calcul parallèles ainsi qu'à leur mémoire (l'annexe C donne plus de détails sur l'API Opencl).

Dans ce chapitre, nous allons décrire la mise en œuvre des différentes étapes précédemment décrites. L'essentiel de cette mémoire est lié à l'utilisation d'OpenCL, donc nous allons d'abord concentrer sur la mise en œuvre des programmes des noyaux. En d'autres termes, nous allons décrire les différents programmes des noyaux et comment mettre en place la plate-forme OpenCL afin de calculer un tel type de rendu.

OpenCL considère un système informatique comme étant constitué d'un certain nombre de dispositifs de calcul, ce qui pourrait être des unités centrales de traitement (CPU) ou "accélérateurs" tels que les unités de traitement graphique (GPU), attachés à un processeur hôte (une CPU). Il définit un langage de type C pour écrire des programmes, appelés noyaux, qui s'exécutent sur les dispositifs de

calcul. Un dispositif de calcul unique se compose généralement de nombreux éléments individuels de traitement (PE) et l'exécution d'un noyau unique peut fonctionner sur tout ou la plupart des PEs en parallèle. En outre, OpenCL définit une interface de programmation d'application (API) qui permet aux programmes en cours d'exécution sur l'hôte pour lancer les noyaux sur les dispositifs de calcul et de gérer la mémoire de dispositif, qui est (au moins conceptuellement) séparée de la mémoire hôte.

OpenCL distingue donc d'un coté d'application tournant sur le processeur hôte (et qui va appeler l'API OpenCL), et de l'autre coté les noyaux qui sont programmés en OpenCL-C. voir la figure 3.1.

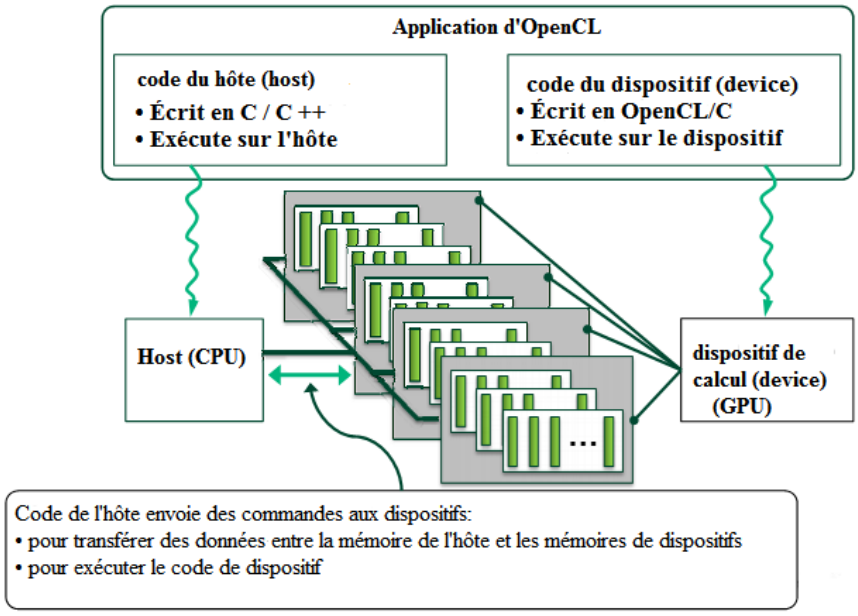


FIGURE 3.1 – Architecture de notre Application

Juste un rappel, nous allons appliquer les formules suivantes mentionnées précédemment :

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \ , \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \ . \end{aligned} \tag{3.1}$$

$$C_{dst} \leftarrow (1 - \alpha_{src})C_{dst} + C_{src} \ . \tag{3.2}$$

Dans l'algorithme de raycasting, le processus de calcul sur les rayons le long d'une direction fixe de la lumière émise est indépendante de l'autre, la méthode de calcul est la même et a les avantages d'une énorme parallélisme. Selon cette étude, l'écran est considéré comme une grille, chaque pixel considéré comme un thread, selon les caractéristiques de l'image, l'écran est divisé en plusieurs régions, ce qui est considéré comme un bloc, puis l'algorithme de raycasting peut être mis en œuvre basé sur

l'architecture opencl à trois niveaux. L'algorithme comprend deux parties, l'une est partie CPU (hôte) ; une autre est partie GPU (Device). Partie CPU lire les données de fichiers et stocke les données dans la mémoire 3D, initialiser l'environnement OpenGL, et établir des liens entre la mémoire vidéo. reçoit les résultats finaux du GPU et afficher le résultat de la visualisation. Partie GPU lancer les rayons et trouver l'intersection de rayons et de l'ensemble de données, accumuler la couleur et l'opacité, renvoie le résultat final à la mémoire.

Les étapes spécifiques de l'algorithme sont les suivantes :

- **Partie CPU :**

  1. Lire l'ensemble de données du format RAW, stocker dans la mémoire de dispositif comme la forme d'un tableau de type unsigned char (caractère non signé),
  2. Initialiser l'environnement OpenGL, établir PBO (Opengl pixel buffer objects) et créer un lien vers la mémoire GPU ; définissez la modèle de la matrice de transformation, définissez la direction d'observation,
  3. Selon la taille de l'ensemble de données, calculer la taille requise de bloc et de thread.

- **Partie GPU :**

  1. Initialiser l'environnement opencl, établir le tableau 3D et la texture 3D d'opencl, l'ensemble de données de tableau de type unsigned char de mémoire du dispositif lu dans un tableau 3D, définir les paramètres de texture et de lier la texture 3D et le tableau 3D,
  2. Paramètres pré établies, calculer la position de pixel de l'écran correspondant, les coordonnées de point de vue selon l'indice de thread,
  3. Calculer la distance vers la caméra, la coordonnée de l'intersection de la rayon entrer dans l'ensemble de données et sur le rayon quitter l'ensemble de données avec la méthode d'intersection Rayon - Box, en fonction de la direction du rayon,
  4. Échantillon le long de la direction de rayon de l'avant vers l'arrière selon un intervalle d'échantillonnage prédéterminé, obtenir la couleur et de l'opacité des points d'échantillonnage et les accumuler. Définissez la seuil d'opacité est égale à 1, si la valeur de la transparence proche de 1.0, arrêter de calcul, sinon continuer à échantillonner le point suivant, jusqu'à ce que le point de gauche est atteint,
  5. Les valeurs finalement accumulés de couleur et d'opacité sont stockés dans le PBO,
  6. Ainsi, le travail de GPU est terminée, les données du PBO sont directement affichés sur l'écran via l'API de l'environnement OpenGL.

3.2 Première partie : Code du hôte

Dans le développement d'une projet OpenCL, la première étape consiste à coder l'application hôte. Cela s'exécute sur l'ordinateur d'un utilisateur (l'hôte) et distribue des noyaux (kernels) aux périphériques (devices) connectés. L'application hôte peut être codé en C ou C ++, et chaque application hôte nécessite cinq structures de données : `cl_device_id`, `cl_kernel`, `cl_program`, `cl_command_queue`, et `cl_context`.

3.2.1 Les principales étapes de notre application

Les étapes pour écrire notre programme sont :

- Écrire un noyau dans un fichier .cl,
- Initialisation les librairies OpenCL,
- Trouvez et initialisation de périphériques,
- Créer un contexte pour le dispositif,
- Créer une file d'attente,
- Lire et compiler le code du noyau,
- Allouer et déplacer la mémoire,
- Spécifier les paramètres pour le noyau,
- Lancer le noyau,
- Lire les résultats.

3.2.2 Résumé sur les principales fonctions du programme

La figure 3.2 et l'algorithme 1 représentent les fonctions principales de code de l'hôte qui exécuté sur le CPU principal.

1. **main() :** La fonction principal, elle est Le point d'entrée du programme
2. **Init GL() :** Initialiser, enregistrer les rappels GLUT et initialiser les extensions nécessaires de GL.
3. **create CLContext() :**

- Vérifier si le dispositif demandé supporte le partage de contexte avec OpenGL,
- Utilisation de l'interopérabilité CL-GL,
- Definie OS et creer le contexte d'OpenCL.

4. **sLoadRawFile() :** Chargement du fichier Raw,
5. **init CLVolume() :**
   - Créer une matrice 3D a partir de volume et le copier vers device,
   - Créer une fonction de transfert,
   - Créer une échantillonneur pour la fonction de transfert (interpolation linéaire),
   - Initialiser la matrice de vue.
6. **init Pixel Buffer() :**
   - Créer un tampon pour l'affichage de pixel,
   - Créer un tampon d'interopérabilité OpenCL/GL.
7. **Calcul Grad() :** Le calcul du gradient,
8. **Reshape() :** Gestionnaire de redimensionnement de la fenêtre,
9. **mouvement() :** Gestionnaire du mouvement de la souris,
10. **Souris() :** Gestionnaire des événements de la souris,
11. **Keyboard Gl() :** Gestionnaire des événements de clavier.
12. **Display GL() :**
    - Boucle principale de GLUT,
    - Utiliser OpenGL pour construire la matrice de vue,
    - Dessiner les images,
    - Incrémenter le compteur des frames.
13. **rendre() :**
    - Rendre les images par OpenCL,
    - Transférer de tampon de GL au tampon de CL,
    - Exécution de kernel(thread) d'OpenCL,

- Transférer de tampon de CL au tampon de GL,
- Copier les donnes a partir de CL_buffer à l'hôte.

Algorithme 1 : Les étapes de notre programme

(Chaque bloc exécute ce qui suit en parallèle sur GPU) ;

1. Rendu image en utilisant OpenCL
   - Obtenir le pointeur de dispositif OpenCL,
   - Appeler le kernel OpenCL, écrire les résultats au buffer, *render_kernel(gridSize, blockSize, d_output, width, height, density, brightness, transferOffset, transferScale) ;*,
   - Afficher les résultats en utilisant OpenGL (appelé par GLUT),
   - Maintenant encoder pour effectuer les opérations, *gridSize=dim3((width, blockSize.x), (height, blockSize.y)) ;*.
2. Charger les données brutes (*raw data*) à partir du disque,
   - void *sLoadRawFile(),
   - Premièrement initialiser le contexte OpenGL, afin que nous pouvons définir correctement le GL pour Opencl puis utilisez la ligne de commande, spécifiée le dispositif Opencl),
   - Appeler le noyau opencl, écrit les résultats au tampon *copyInvViewMatrix(invViewMatrix, sizeof(float4)*3)* ,
   - Lancer la minuterie à 0 et procédé plan des z boucles sur le GPU.
3. Puis libérez la mémoire tampon d'OpenCL *freeCLBuffers()*.

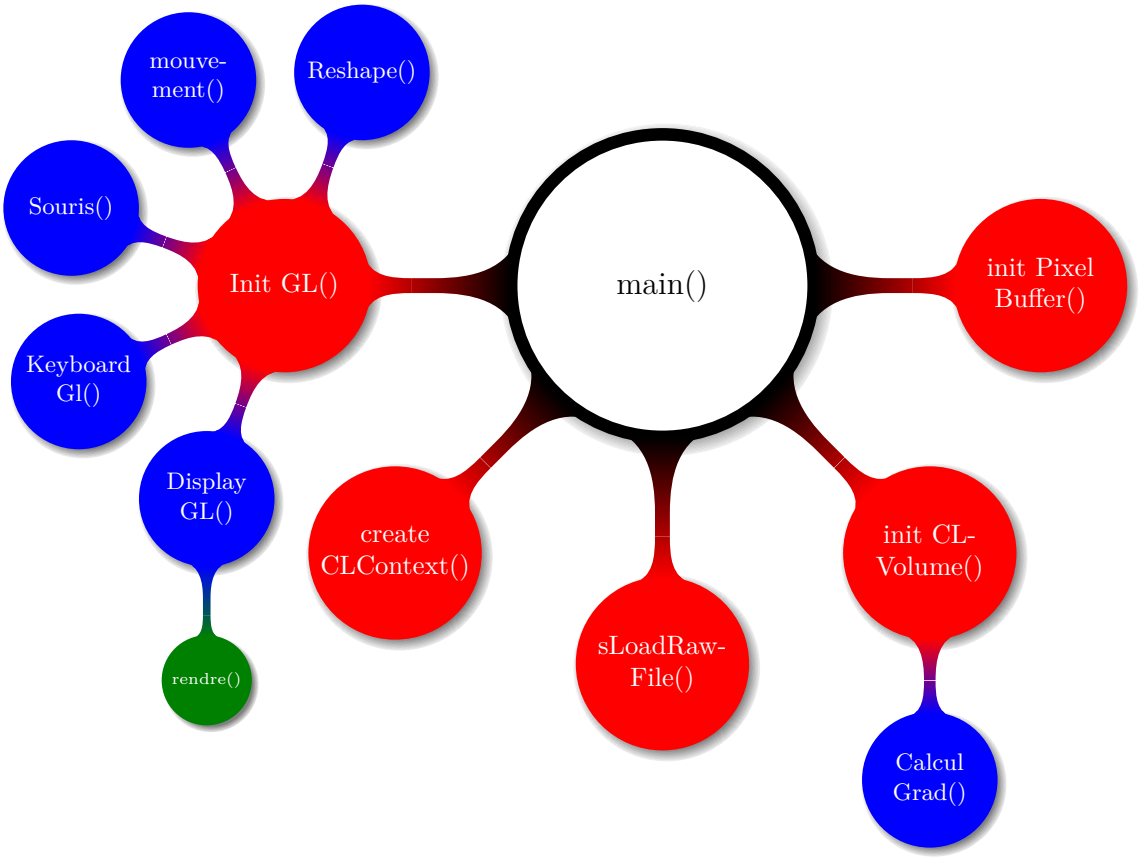


FIGURE 3.2 – Les fonctions principales du programme

3.3 Deuxième partie : Le Calcul sur les dispositifs

Par défaut, chaque noyau OpenCL tente d'utiliser toutes les ressources de calcul sur un dispositif conforme à un modèle de calcul parallèle de données. En d'autres termes, le même noyau est utilisé pour traiter des données sur l'ensemble des ressources de calcul dans un dispositif. En revanche, un modèle de tâche parallèle utilise les ressources de calcul disponibles pour exécuter un ou plusieurs noyaux indépendants sur le même dispositif. Les deux : les tâches et le parallélisme de données sont des moyens valables pour structurer le code pour accélérer les performances des applications étant donné que certains problèmes sont mieux résolus par la parallélisation des tâches tandis que d'autres se prêtent mieux à la solution par le parallélisme de données. En mission générale le parallélisme est plus compliqué à mettre en œuvre de manière efficace à la fois de point de vue matériel et logiciel. Le comportement par défaut d'OpenCL à utiliser toutes les ressources disponibles dans le modèle de données de calcul parallèle est bonne parce qu'elle fournira plus de dispositifs d'accélération (devices) individuels.

Nous allons voir ici le thread (Il s'agit d'un ensemble d'instructions de l'algorithme de GPU raycasting) qui est exécuté sur les noyaux de GPU en parallèle.

3.3.1 les étapes de l'algorithme

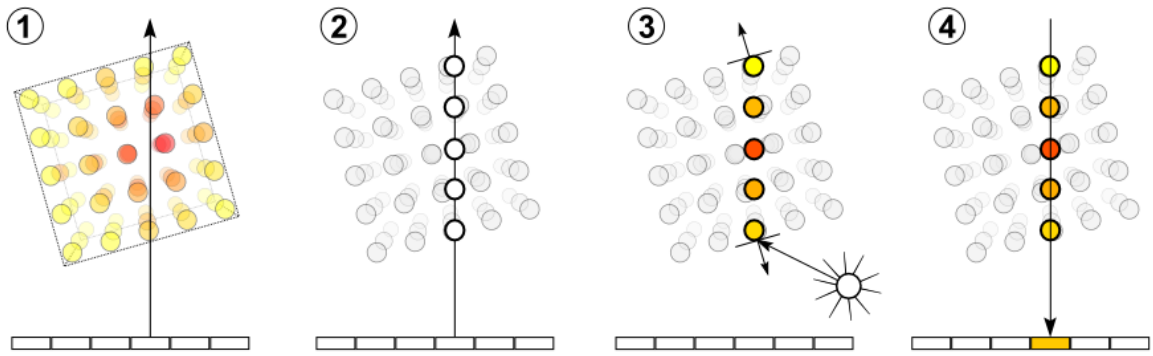


FIGURE 3.3 – Les étapes de raycasting [Baert 2012]

1. Lancé un rayon par pixel : pour chaque pixel de l'espace d'écran :
   - ⋆ Lancé un rayon,
   - ⋆ Direction d'observation.
2. Échantillonnage :

⋆ Échantillonner les données le long du rayon avec des intervalles uniformes,

⋆ Calculer la valeur scalaire du point d'intersection avec le volume par l'interpolation trilinéaire.

3. Classification : appliquer la fonction de transfert sur la valeur scalaire obtenu pour affecter la couleur et l'opacité correspondante,

4. Accumulation : intègrent les propriétés optiques couleur et opacité le long de rayon pour calculer la couleur et l'opacité finale du pixel par la formule 3.1 ou 3.2.

Ces étapes peut êtres résumer dans l'algorithme 2.

Algorithme 2 : Algorithme de GPU raycasting [Stegmaier *et al.* 2005]

```
Calculer le point d'entrée dans le volume de données ;
Calculer le vecteur de direction du rayon ;
while (l'on est dans le volume) do
    Récupérer la valeur scalaire à la position courante du rayon ;
    Accumuler couleur et opacité ;
    Avancer le long du rayon ;
end
```

3.3.2 Relation entre les threads et la lumière et de la conception de pré-paramètres

La relation entre les threads et la lumière est la création d'un lien entre tout rayon avec le thread d'opencl correspondant, qui est la clé de la parallélisation à propos de l'algorithme de raycasting. Deux variables intégrées d'opencl get_num_groups(), get_local_size() signifient le nombre de groupes de travail et la taille de bloc. Deux variables intégrées d'opencl get_group_id(), get_local_id() signifient la valeur globale unique ID pour un élément de travail, la valeur locale unique ID pour un élément de travail. Grâce à la combinaison des deux signifient l'unité de thread (élément de travail) parallèle. get_group_id(), get_local_id() contient deux dimensions, à savoir x et y. Créez l'index de chaque rayon à travers l'indice de thread selon la formule 3.3 ci-dessous.

$$\begin{aligned} thread\_indice &= get\_global\_id(0) \\ get\_global\_id(0) &= get\_group\_id(0) \times get\_local\_size(0) + get\_local\_id(0) \end{aligned} \tag{3.3}$$

Pour faciliter le calcul, la valeur des dimensions de get_loca_id() est normalisé pour l'intervalle $[-1, 1]$ selon la formule 3.4 ci-dessous.

$$u = (thread\_indice.x/imageW) * 2 - 1$$
$$v = (thread\_indice.y/imageH) * 2 - 1 \tag{3.4}$$

imageW signifier la largeur du plan de vue, imageH signifie la hauteur du plan de vue.

La conception de pré-paramètre, première taille de plan de vue et et la position de l'œil, comme le montre la figure 3.4.

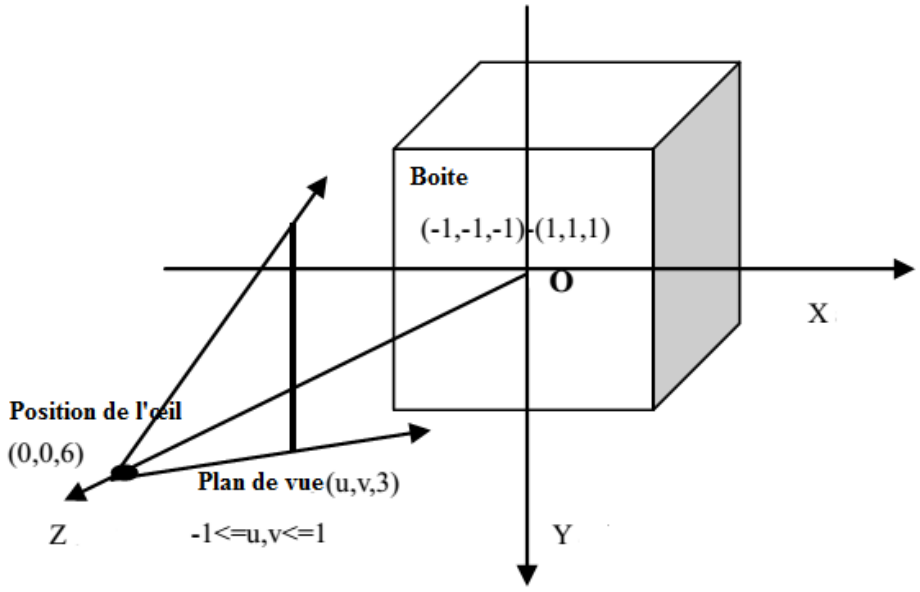


FIGURE 3.4 – Les relations spatiales

La position de l'œil est sur l'axe Z avec les coordonnées $(0, 0, 6)$, plan de vue est sur l'axe z $(0, 0, 3)$ et la taille est de $3*3$. Deuxièmement, définir l'unité Rayon-Boite, dans lequel, les coordonnées inférieures de coin gauche de la boîte sont $(-1 - 1, -1)$, les coordonnées du coin supérieur droit de la boîte sont $(1, 1, 1)$.

3.3.3 Intersection rayon - boîte et l'accumulation de couleur et d'opacité

La clé de l'algorithme de raycasting est intersection rayon - boîte, l'accumulation des valeurs de couleur et d'opacité le long de la direction du rayon, et l'image composite. La première étape est l'opération d'intersection des rayons, pour toute la lumière, en utilisant la notation de vecteur pour le décrire, fixé deux paramètres de base, coordonnées d'origine et la direction de vue, puis tout rayon peut être exprimée comme la formule 3.5 ci-dessous, t de formule 3.5 représente le déplacement de rayon dans la direction de vue.

$$Ray_i = eyeRay.o + eyeRay.d \times t \tag{3.5}$$

Pour la recherche de l'intersection entre les rayons et la boite selon les méthodes proposées par G. Scott Owen, calculer les deux intersections entre le rayon et la boite, nommé t_near et t_far, puis en prenant t_near comme l'origine, se déplaçant le long de la ligne de vue, à chaque fois, la distance de pas en avant. Chaque pas en avant, calculer les coordonnées d'un point d'échantillonnage sur le rayon dans l'espace selon la formule 3.6 ci-dessous.

$$pos = eyeRay.o + eyeRay.d \times t\_near + eyeRay.d \times step \tag{3.6}$$

($eyeRay.o$ est les coordonnées d'origine de rayon, $eyeRay.d \times t\_near$ est les coordonnées de l'endroit où le rayon entrant, $eyeRay.d \times step$ est la distance vers l'avant du rayon à chaque fois). Depuis la pos est dans la boîte, ses valeurs de coordonnées sont situés entre $[-1, -1, -1]$ et $[1, 1, 1]$. Information de la couleur et d'opacité d'image est stockée en tableau d'opencl en 3D sous la forme d'une texture 3D, les coordonnées de texture est situé entre $[0, 1]$. Convertir les coordonnées de pos aux coordonnées de texture de $pos\_texture$ selon la formule 3.7 ci-dessous.

$$pos\_texture = (pos + 1)/2 \tag{3.7}$$

Puis, selon les coordonnées du point, calculer les coordonnées de texture correspondant, obtenir la valeur de texture appropriée sur la base de coordonnées de texture qui est la valeur de gris du point. Créer une fonction de transfert à une dimension pour indiquer les différent contenu de l'image. La fonction comprend quatre composantes qui est R, V, B, α, les trois premiers paramètres est la conversion entre la couleur et le gris, le quatrième est de contrôler la transparence, grâce à la fonction que nous pouvons contrôler la conversion entre le gris et la couleur.

Enfin est l'accumulation d'opacité et de couleur. Ici, nous calculons la couleur et l'opacité selon la formule 3.8 ci-dessous la formule avec la méthode de l'avant vers l'arrière.

$$\begin{aligned} C_{out}\alpha_{out} &= C_{in}\alpha_{in} + C_{now}\alpha_{now}(1 - \alpha_{in}) \\ \alpha_{out} &= \alpha_{in} + \alpha_{now}(1 - \alpha_{in}) \end{aligned} \tag{3.8}$$

Dans la formule 3.8, C représente une valeur de couleur, α représente l'opacité. Dans l'expérience, la terminaison précoce d'opacité est adopté. Le seuil est fixé à 1. Si l'opacité a été atteint le seuil, les voxels derrière seront ignorées, ce qui peut faire gagner du temps. Enfin, la valeur accumulée est

directement stocké au PBO.

3.3.4 Thread principal : GPU raycasting

Le volume ou la grille 3D peut être identifiée par une texture 3D. L'algorithme actuel de rendu (voir le pseudo-code dans l'algorithme 2) peut être presque directement transformé à un programme de kernel (un seul thread qui exécute sur un seul noyau). Ici, un kernel pour un seul pixel sur le plan d'image est identifié avec son rayon correspondant.

Nous allons utiliser les grilles uniformes, qui ont une structure géométrique et topologique simples. En particulier, une grille uniforme 3D peut être identifiée avec une texture 3D, de la même manière que pour les coupes de texture 3D. Par conséquent, les grilles uniformes sont largement utilisées pour les algorithmes GPU raycasting. L'algorithme de rendu réel peut être directement associé à un programme de fragment. Ici, un fragment ou un pixel sur le plan d'image est identifié avec son rayon correspondant.

Les points d'entrée sont fixés comme des coordonnées de texture *TEXCOORD0*. Toutes les positions et les vecteurs de direction sont décrits par rapport au système de coordonnées locales de l'ensemble de données de volume. La boîte englobante du volume peut être orientée selon les axes de coordonnées principales. Par conséquent, la boîte englobante peut être décrite par deux points 3D *volExtentMin* et *volExtentMax*, qui fournissent les coordonnées minimales et maximales de la boîte englobante, respectivement. La position de la caméra et la taille du pas sont également donnés par rapport au système de coordonnées locales du volume. La direction du rayon est calculée comme la différence normalisée entre la position d'entrée et la position de la caméra.

Le parcours des rayons est mis en œuvre par un ensemble d'opérations pour chaque itération. Ici, l'ensemble de données de volume mémorisées dans la texture 3D *SamplerDataVolume*, sont accessibles. L'interpolation trilinéaire est une fonctionnalité intégrée automatiquement au matériel graphique et fournit un filtre de reconstruction, lorsque la position actuelle du rayon est différente d'un point de la grille. Les valeurs RVBA correspondantes sont calculées en appliquant la fonction de transfert, qui est maintenu dans la texture *SamplerTransferFunction 1D*. Ensuite, la composition avant-en-arrière est appliquée et la position des rayons est avancée avec une taille de pas fixe. La dernière partie de shader implémente la terminaison de rayons. Le parcours des rayons est poursuivi uniquement lorsque tous les trois coordonnées x, y et z de la position actuelle du rayon est supérieur au nombre de coordonnées respectives *volExtentMin* et plus petite que les coordonnées respectives des *volExtentMax*.

Les fragments pour raycasting sont générés en rendant les faces avant de la boîte englobante du volume. Les points d'entrées (en ce qui concerne le système de coordonnées local du volume) sont fixés comme coordonnées de texture des sommets de la boîte englobante, les valeurs intermédiaires sont

remplis automatiquement par l'interpolation trilinéaire.

L'algorithme 3 permet d'afficher une source d'un kernel (thread) opencl pour l'algorithme GPU raycasting. Il décrite tous les étapes mentionnées dans la section 3.3.1 : le calcul de la position d'entrée et la direction de rayon, le parcours de rayons, l'accès aux données, l'accumulation, et l'arrêt de rayon. La boite englobante peut être décrite par deux points 3D *plan_Min, plan_Max*, qui déterminent les coordonnées minimales et maximales de boite, respectivement, le calcul du point d'intersection et retourne les deux points d'intersection *(t_near, t_far)* .

Le parcours des rayons est mis en application par une boucle dans le thread (kernel). Ici, l'ensemble de données de volume, enregistré dans la texture 3D *volume_tex*, est consulté. L'interpolation trilinéaire est une fonctionnalité intégrée au matériel graphique et fournit automatiquement un filtre de reconstruction quand la position actuelle de rayon est différente d'un point de la grille. Les couleurs correspondants aux valeurs RVBA sont calculées en appliquant la fonction de transfert, qui est tenue dans la texture 2D *transfer_Fonc*. Puis, l'accumulation avant-en-arrière est exécutée par la formule 3.1 et le rayon est avancé dans le volume pour la prochaine point d'échantillonnage. Une taille d'étape fixe est utilisée par *step_t*. La dernière partie du kernel met en application l'arrêt de rayon. Le parcours de rayon est seulement arrêter quand le rayon va sortir de la boite $t > t\_far$ ou bien l'opacité courante est supérieur à l'opacité de seuil $interm.w > seuil\_opacité$. Enfin, nous obtenons la couleur du pixel i *pix_output[i]* par le thread (kernel) i.

Algorithme 3 : Algorithme plus détaillée de GPU raycasting

```
Calculer le point d'entrée dans le volume de données ;
Calculer le vecteur de direction du rayon ;
coord_cartesiennes = point_d'entree ;
direction = normaliser(point_d'entree + position_observateur) × longueur_pas ;
couleur_opacite = ((0.0, 0.0, 0.0), 0.0) ;
position = normaliser(transformer(coord_cartesiennes)) ;
while (position < 1.0) do
    scalaire = AccesTexture(donnees_Texture_3d, position) ;
    couleur_opacite_temp = AccesTexture(fonction_de_transfert, scalaire) ;
    couleur_opacite = composition(couleur_opacite, couleur_opacite_temp) ;
    if (couleur_opacite.opacite = 1.0) then
        sortir_de_la_boucle ;
    end
    coord_cartesiennes = coord_cartesiennes + longueur_pas ;
    position = normaliser(transformer(coord_cartesiennes)) ;
end
couleur_finale = couleur_opacite ;
```

3.4 Méthodes d'accélération : La terminaison précoce de rayon

Le raycasting peut facilement incorporer un certain nombre de techniques d'accélération pour surmonter les différents problèmes du rendu de volume basé sur le raycasting. Il existe plusieurs techniques d'accélération qui sont immédiatement liées au raycasting : La terminaison précoce de rayon, l'accélération dans les espaces vides. Dans notre projet nous avons utilisé la méthode de "la terminaison précoce de rayon" (early ray termination).

La terminaison précoce de rayon nous permet d'arrêter les rayons dès que nous saurons que des éléments de volume plus loin de caméra sont occluses. Le parcours de rayon peut être arrêté quand l'opacité accumulée *seuil_opacite* atteint une certaine limite spécifiée par l'utilisateur (qui est en général très proche de 1). Le critère d'arrêt est ajouté aux critères d'arrêt de rayon qui pourraient déjà faire partie du raycasting. L'algorithme 4 représente cette méthode d'accélération. Ici, la valeur choisie de seuil est $seuil\_opacite = 0.95$.

Algorithme 4 : Algorithme pour la terminaison précoce de rayon

```
#define seuil_opacite 0.95f
⋮
while (position < 1.0) do
    ...
    if (couleur_opacite.opacite > seuil_opacite) then
        sortir_de_la_boucle ;
    end
    ...
end
⋮
```

Cette technique est utilisé seulement dans l'état de l'accumulation avant-vers-arrière.

3.5 Conclusion

Les nouvelles API de calcul parallèle a ouvert de nouvelles possibilités dans le domaine de la visualisation. Le plus notable dans ce contexte est l'OpenCL du groupe de Khronos. Le standard OpenCL propose une interface de programmation adaptable à différents types d'accélérateurs (GPU, CPU . . .). L'intérêt pour cette API est motivé par la puissance de calcul qu'ils permettent. En visualisation médicale on est dans le besoin croissant pour un accès rapide à l'imagerie médicale avant, après et pendant les procédures de calcul qui nécessitent des mises à jour en temps réel, et interaction en temps réel. Dans de nombreux cas, même les petits retards sont inacceptables et rendront une méthode inutile d'un point de vue pratique.

Le standard OpenCL se divise en deux parties. La première définit un langage, proche du C, qui peut être compilé par chaque implémentation pour être exécuté sur différents accélérateurs. Les codes écrits avec ce langage sont appelés kernels (contient la source de l'algorithme de raycasting). La seconde partie décrit une API pouvant être utilisée par des programmes classiques pour allouer de la mémoire sur les accélérateurs ; pour transférer des données entre la mémoire d'un accélérateur et la mémoire hôte ; pour compiler des kernels ; pour exécuter ces kernels sur les différents accélérateurs.

CHAPITRE 4

ANALYSE ET INTERPRÉTATION DES RÉSULTATS

4.1 introduction

Dans ce chapitre, nous allons étudier les performances de notre mise en œuvre, en termes de résultats visuels, mais aussi en termes de vitesse de calcul. Nous comparons la qualité des filtres de reconstruction disponible actuellement dans notre mise en œuvre. Enfin, nous présentons les résultats de visualisation obtenus avec des données médicales dans le monde réel.

4.2 Le matériel utilisé

Nous allons présenter les résultats de notre algorithme qui a été implémenté sous le système d'exploitation Windows en utilisant le langage opencl/C avec une librairie graphique puissante : OpenGL. OpenCL (Open Computing Language) est un standard ouvert et libre dédié à la programmation parallèle de processeurs standards (CPU), graphiques (GPU) ou mixtes. OpenCL permet notamment aux programmeurs d'utiliser un langage C familier pour développer du code sur des plates-formes hétérogènes multicœurs mêlant CPU, GPU et, potentiellement (c'est en tout cas le but d'Altera) FPGA.

Nous avons utilisé deux ordinateurs avec des propriétés différentes : Le premier ordinateur est caractérisé par un CPU Intel Core 2 Duo de fréquence d'horloge de 2.93 GHz et de RAM de 3 GB et GPU NVIDIA GeForce 8400 GS , et la deuxième ordinateur est caractérisé par un CPU Intel(R) Core(TM) i5-2450M de fréquence d'horloge de 2.50GHZ 2.50GHZ et de RAM de 6 GB et GPU AMD

Raedon HD 7400M series.

Les caractéristiques des deux GPU utilisées

Device info :	GPU NVIDIA :	GPU AMD :
CL_DEVICE_NAME:	GeForce 8400 GS	Caicos
CL_DEVICE_VENDOR:	NVIDIA Corporation	Advanced Micro Devices, Inc.
CL_DRIVER_VERSION:	310.90	1445.5 (VM)
CL_DEVICE_VERSION:	OpenCL 1.0 CUDA	OpenCL 1.2 AMD-APP (1445.5)
CL_DEVICE_TYPE:	CL_DEVICE_TYPE_GPU	CL_DEVICE_TYPE_GPU
CL_DEVICE_MAX_COMPUTE_UNITS:	1	2
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:	3	3
CL_DEVICE_MAX_WORK_ITEM_SIZES:	512 / 512 / 64	256 / 256 / 256
CL_DEVICE_MAX_WORK_GROUP_SIZE:	512	256
CL_DEVICE_MAX_CLOCK_FREQUENCY:	1400 MHz	700 MHz
CL_DEVICE_ADDRESS_BITS:	32	32
CL_DEVICE_MAX_MEM_ALLOC_SIZE:	128 MByte	512 MByte
CL_DEVICE_GLOBAL_MEM_SIZE:	512 MByte	1024 MByte
CL_DEVICE_ERROR_CORRECTION_SUPPORT:	no	no
CL_DEVICE_LOCAL_MEM_TYPE:	local	local
CL_DEVICE_LOCAL_MEM_SIZE:	16 KByte	32 KByte
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE:	64 KByte	64 KByte
CL_DEVICE_IMAGE_SUPPORT:	1	1
CL_DEVICE_MAX_READ_IMAGE_ARGS:	128	128
CL_DEVICE_MAX_WRITE_IMAGE_ARGS:	8	8
CL_DEVICE_IMAGE <dim>:		
2D_MAX_WIDTH	4096	16384
2D_MAX_HEIGHT	16383	16384
3D_MAX_WIDTH	2048	2048
3D_MAX_HEIGHT	2048	2048
3D_MAX_DEPTH	2048	2048

4.3 Images de départ

Avant de présenter les images générées par notre algorithme, nous présentons les volumes scanners (ensemble de données 3D de type raw (raw DataSet extrait a partir des données CT,IRM,... par des outils de conversion puisque le type raw reconnu avec les APIs d'OpenGL)) pour lesquels nous allons l'appliquer.

Chaque échantillon est représenté par un volume $(x \times y \times z)$ résultant d'un ensemble de coupes en z, chaque coupe $2D$ possède une taille $x \times y$. On rappelle que notre objectif est de reconstruire et de visualiser le volume $3D$.

Tous les Dataset 3D utilisées dans les tests sont téléchargeables à partir les sites [rawDatasets , rawDatasetsNew]

1. Un volume scanner du pied qui a $256 \times 256 \times 256$ voxels échantillonnés sur 8 bits avec la taille d'un voxel qui est égale à $1 \times 1 \times 1\ mm$. La taille de ce volume est alors : 16 Moctets.
2. Un volume scanner de jambe qui a $341 \times 341 \times 93$ voxels échantillonnés sur 8 bits avec la taille d'un voxel qui est égale à $1 \times 1 \times 4\ mm$. La taille de ce volume est alors : 10.3 Moctets.
3. Un volume scanner du crâne qui a $256 \times 256 \times 256$ voxels échantillonnés sur 8 bits avec la taille d'un voxel qui est égale à $1 \times 1 \times 1\ mm$. La taille de ce volume est alors : 16 Moctets.
4. Un volume scanner de ventricule qui a $256 \times 256 \times 124$ voxels échantillonnés sur 8 bits avec la taille d'un voxel qui est égale à $0.9 \times 0.9 \times 0.9\ mm$. La taille de ce volume est alors : 7.75 Moctets.
5. Un volume scanner de pied qui a $256 \times 256 \times 256$ voxels échantillonnés sur 8 bits avec la taille d'un voxel qui est égale à $1 \times 1 \times 1\ mm$. La taille de ce volume est alors : 16 Moctets.

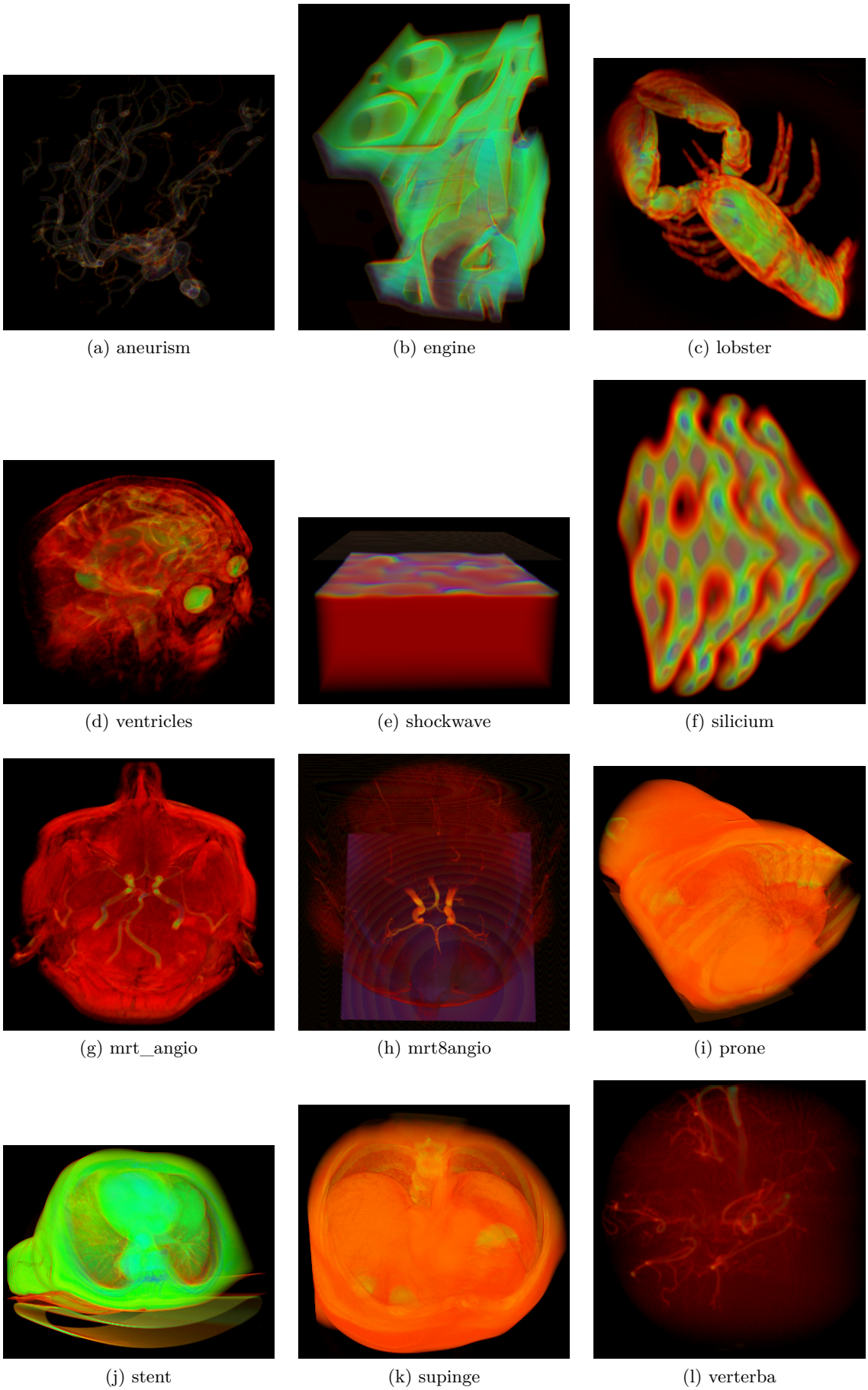
Le tableau 4.1 fournit des informations sur chaque volume :

Data Set	Taille	Nom
foot	$256 \times 256 \times 256$	DS1
statueLeg	$341 \times 341 \times 93$	DS2
skull	$256 \times 256 \times 256$	DS3
mri_ventricles	$256 \times 256 \times 124$	DS4
aneurism	$256 \times 256 \times 256$	DS5

TABLE 4.1 – Ensemble des données 3D utilisés dans les tests

4.4 Exemples d'images générées avec notre algorithme

Les images suivantes représentent quelques résultats que nous avons obtenus grâce à notre approche 4.1 :



(a) aneurism (b) engine (c) lobster

(d) ventricles (e) shockwave (f) silicium

(g) mrt_angio (h) mrt8angio (i) prone

(j) stent (k) supinge (l) verterba

FIGURE 4.1 – Exemples de rendu des différents DataSet

4.5 Tests

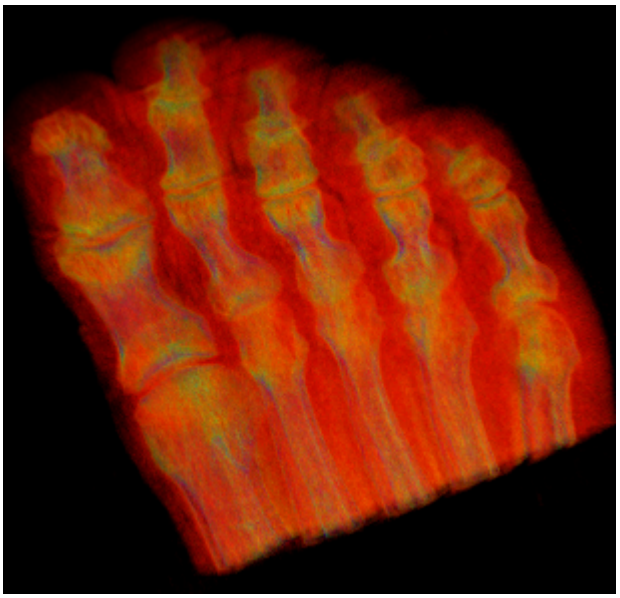
Nous avons aussi testé l'algorithme avec plusieurs facteurs, par exemple selon le type de filtrage, la taille de bloc de thread, la taille d'étape, la valeur de la seuil d'opacité, tous ces éléments sont discuté dans les sections suivantes.

4.5.1 Test selon le type de filtre (Interpolation)

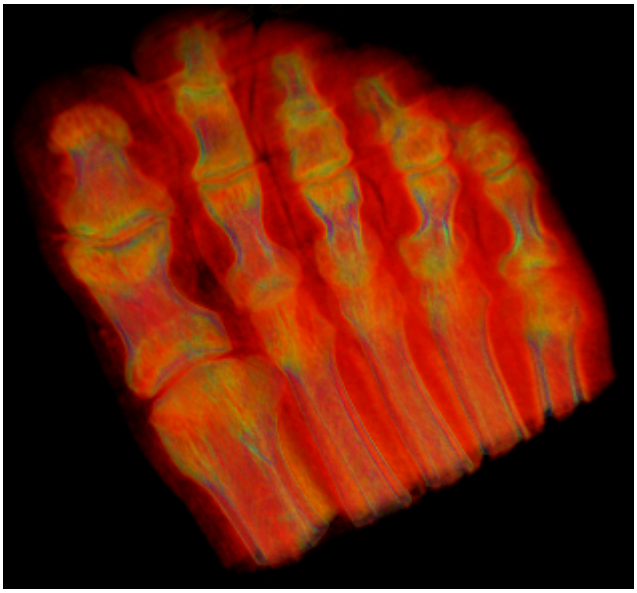
Nous avons généré des images pour “le pied” en utilisant l’interpolation linéaire et l’interpolation de plus proche voisin “*CL_FILTER_LINEAR,CL_FILTER_NEAREST*” et les résultats sont montrés dans la figure 4.2 et la table 4.2. La distinction entre les deux types d’image est difficile. Pour nos données, l’interpolation par plus proche voisin conduit à des résultats quasiment identiques à l’interpolation bilinéaire.

DataSet	Filtre	FPS(NVIDIA)	FPS(AMD)
DS1	CL_FILTER_NEAREST	11	25
	CL_FILTER_LINEAR	10	16
DS2	CL_FILTER_NEAREST	11	36
	CL_FILTER_LINEAR	08	18

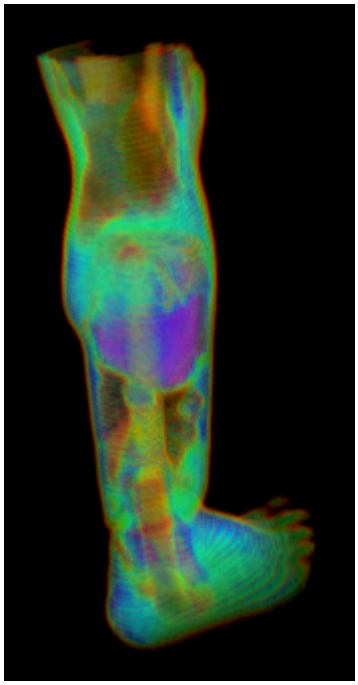
TABLE 4.2 – Taux d’affichage en frames par seconde (FPS) selon le type de filtre



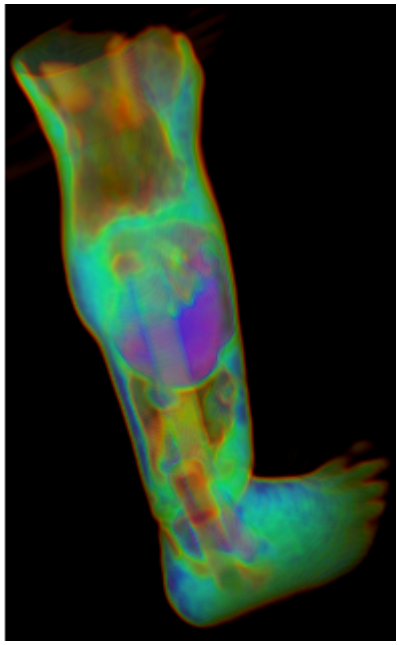
(a) rendu DS1 par CL_FILTER_NEAREST



(b) rendu DS1 par CL_FILTER_LINEAR



(c) rendu DS2 par CL_FILTER_NEAREST



(d) rendu DS2 par CL_FILTER_LINEAR

FIGURE 4.2 – Rendu des différents DataSet selon le type de filtrage

4.5.2 Temps de calcul

Nous avons généré des images pour des volumes scanners qui sont différents en taille en fixant la taille de la fenêtre voulue en pixel et nous avons calculé le temps d'exécution de notre algorithme pour chaque volume. Les résultats sont montrés dans la courbe de la figure 4.3.

Nous avons fait la même chose pour le calcul du temps d'exécution de notre algorithme en variant la taille de la fenêtre en pixel pour un seul volume scanner et les résultats sont montrés par la courbe de la figure 4.4.

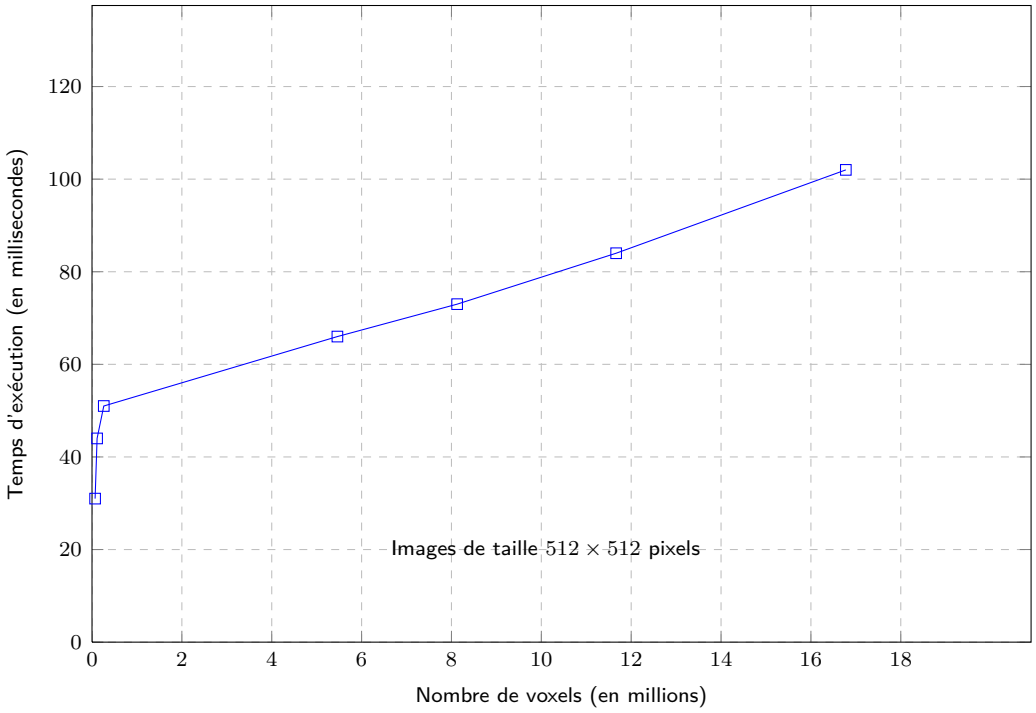


FIGURE 4.3 – Temps de calcul en fonction de nombre de voxels

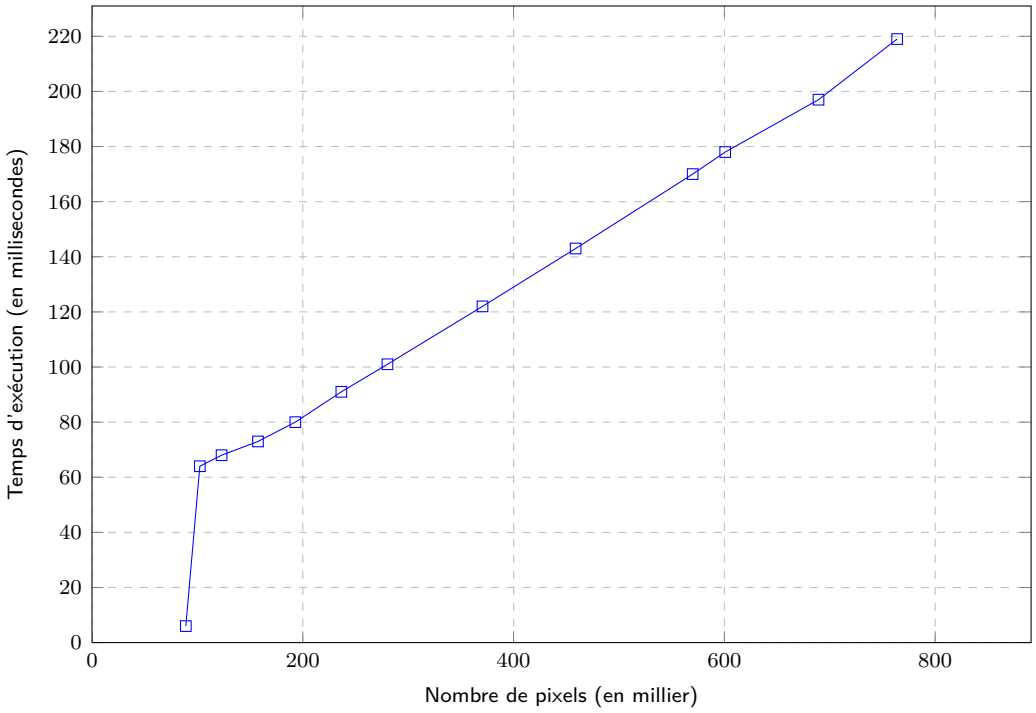
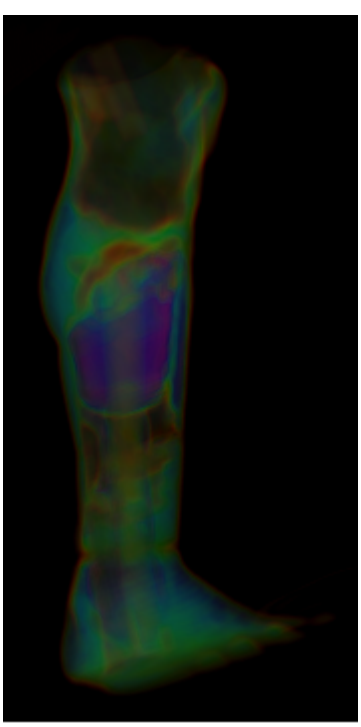


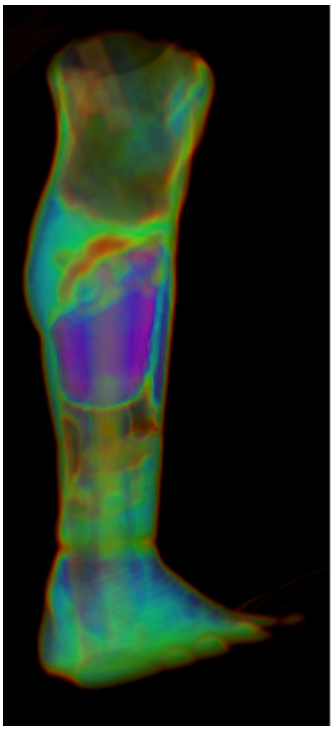
FIGURE 4.4 – Temps de calcul en fonction de nombre de pixels des images voulues.

4.5.3 Qualité d'image

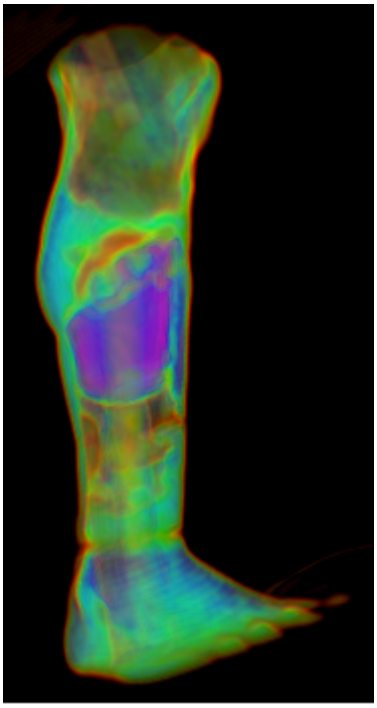
La qualité (ou la précision) de l'image générée dépend de la valeur de densité. Pour cela, nous avons pris quatre valeurs pour cette densité en génération des images pour le pied. Ces quatre valeurs sont : $0.01, 0.03, 0.05$ et 1 respectivement. Les résultats sont montrés dans la figure 4.5.



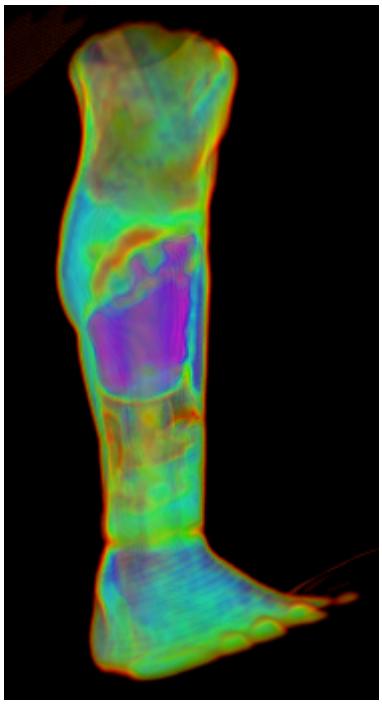
(a) rendu de pied avec *densité* = 0.01



(b) rendu de pied avec *densité* = 0.03



(c) rendu de pied avec *densité* = 0.05



(d) rendu de pied avec *densité* = 1

FIGURE 4.5 – Rendu des différents DataSet selon la valeur de la densité

4.5.4 Test selon la taille de bloc

La sélection de la taille de bloc (le nombre des thread qui s'exécutent en parallèle) peut avoir une influence énorme sur l'exécution d'un kernel d'OpenCL, nous avons testé l'algorithme avec plusieurs tailles de bloc pour trouver la longueur optimal de bloc. Le tableau 4.3 et le diagramme 4.6 affichent l'effet de la longueur de bloc sur le débit d'images.

DataSet	Taille de block	FPS(NVIDIA)	FPS(AMD)
DS1	4×4	04	7
	16×16	08	13
DS2	4×4	04	9
	16×16	08	20
DS3	4×4	04	8
	16×16	09	14
DS4	4×4	04	9
	16×16	09	23
DS5	4×4	04	8
	16×16	09	15

TABLE 4.3 – Taux d'affichage en frames par seconde (FPS) par taille de bloc

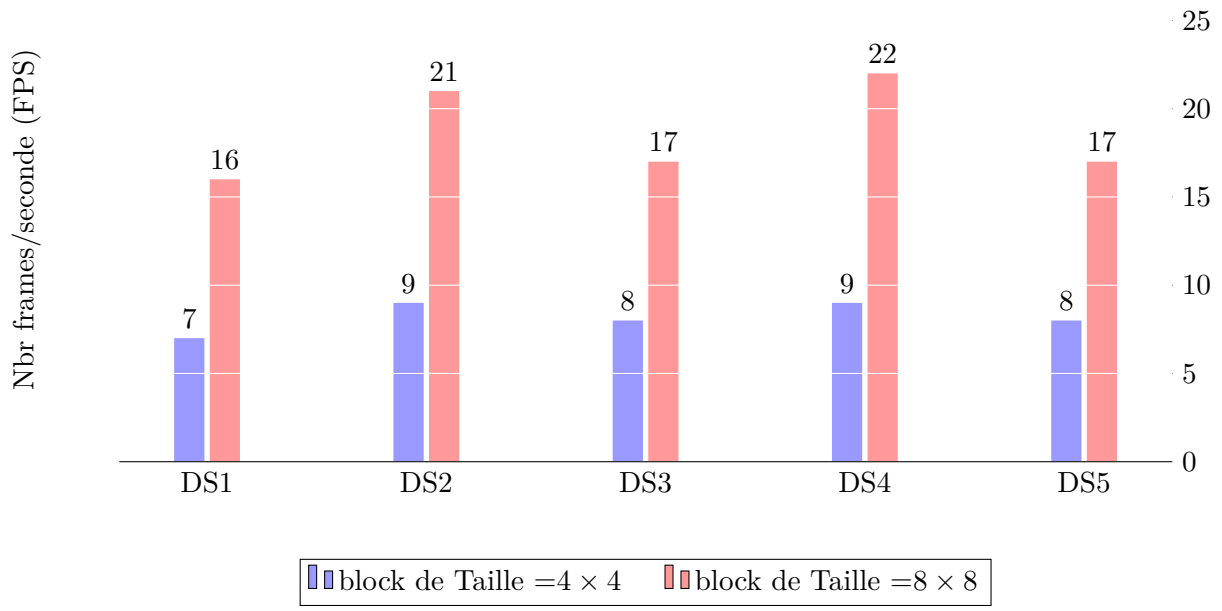


FIGURE 4.6 – Nombre de frames par seconde (FPS) en fonction de la taille de bloc de calcul de la carte AMD avec les déférents DataSet.

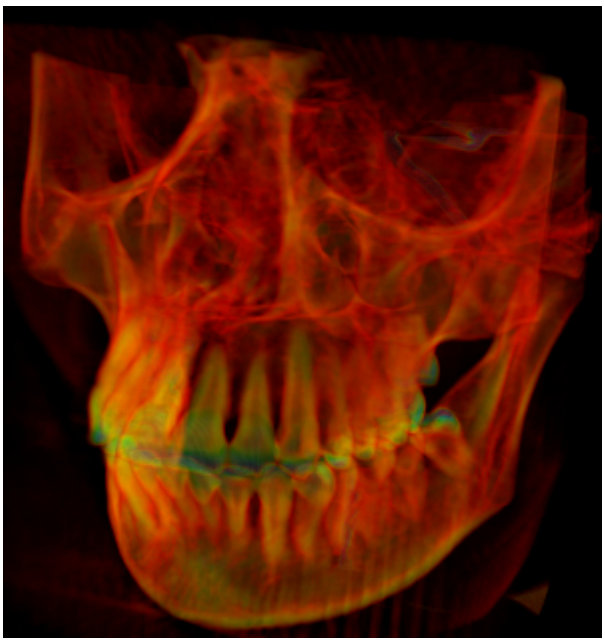
4.5.5 Test selon le pas d'échantillonnage

Un autre élément important est le pas d'échantillonnage, Les figures 4.7 et 4.8 affichent la variation de la qualité et le débit d'image finale selon la taille d'étape (*step_t*). Il peut observer que la qualité d'image finale dépend de la taille d'étape. pendant que la taille d'étape est diminuée, et par conséquent, le nombre d'échantillons par rayon est augmenté (minimiser les nombres de calcules d'intersection par thread). donc l'image rendue explique un plus de haute qualité, et augmente le nombre de frames par seconde.

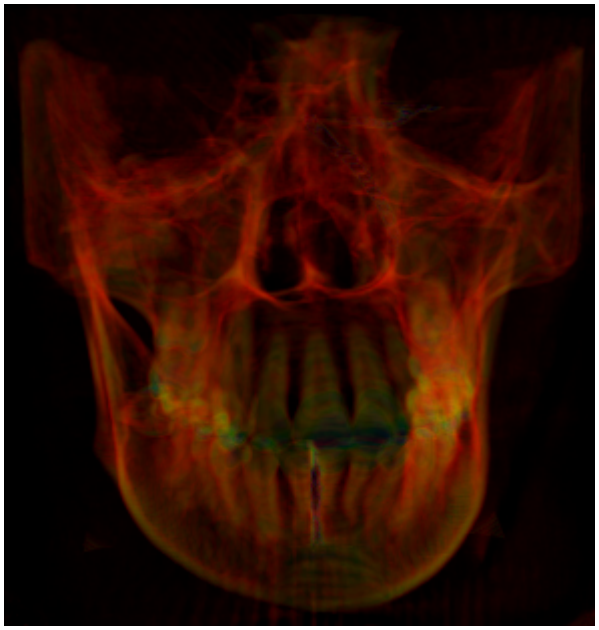
Le tableau 4.4 affiche le nombre de frames par seconde (en fps) réalisée avec 4 tailles d'étape différentes.

DataSet	Taille de l'étape	FPS(NVIDIA)	FPS(AMD)
DS1	0.01	08	16
	0.03	16	38
	0.05	18	50
	0.07	26	56
DS3	0.01	09	21
	0.03	18	43
	0.05	21	56
	0.07	27	62

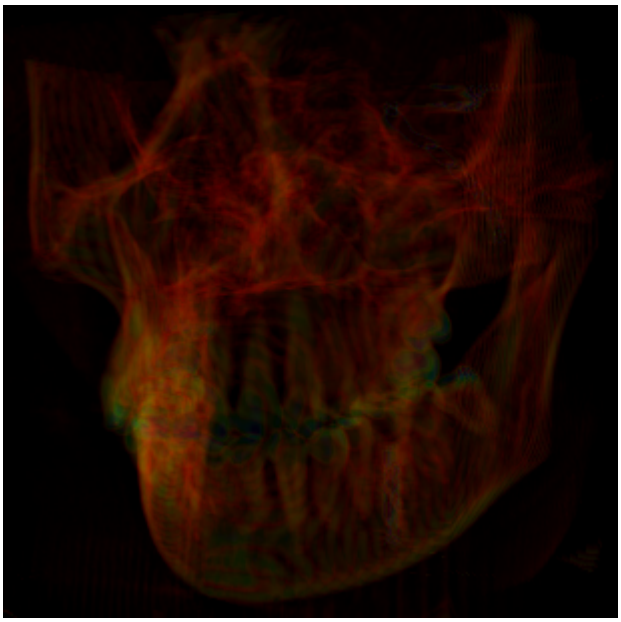
TABLE 4.4 – Nombre de frames par seconde par la taille de l'étape



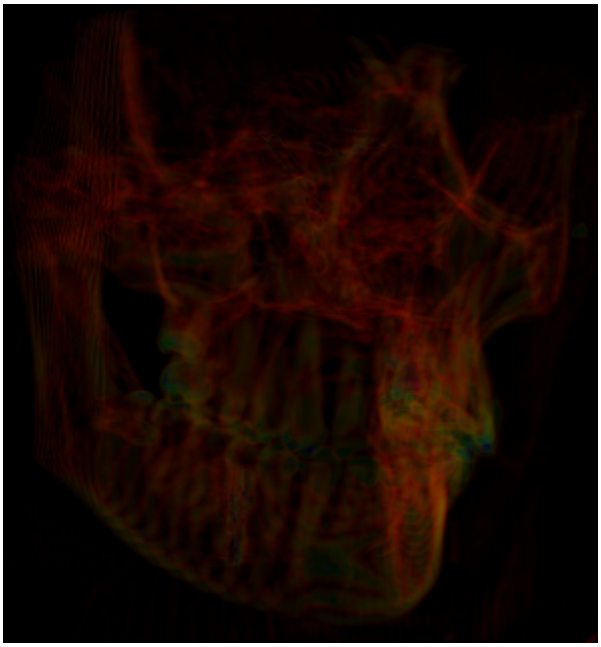
(a) rendu DS3 avec $step\_t = 0.01$



(b) rendu DS3 avec $step\_t = 0.03$



(c) rendu DS3 avec $step\_t = 0.05$



(d) rendu DS3 avec $step\_t = 0.07$

FIGURE 4.7 – Rendu des différents DataSet selon la taille d'étape

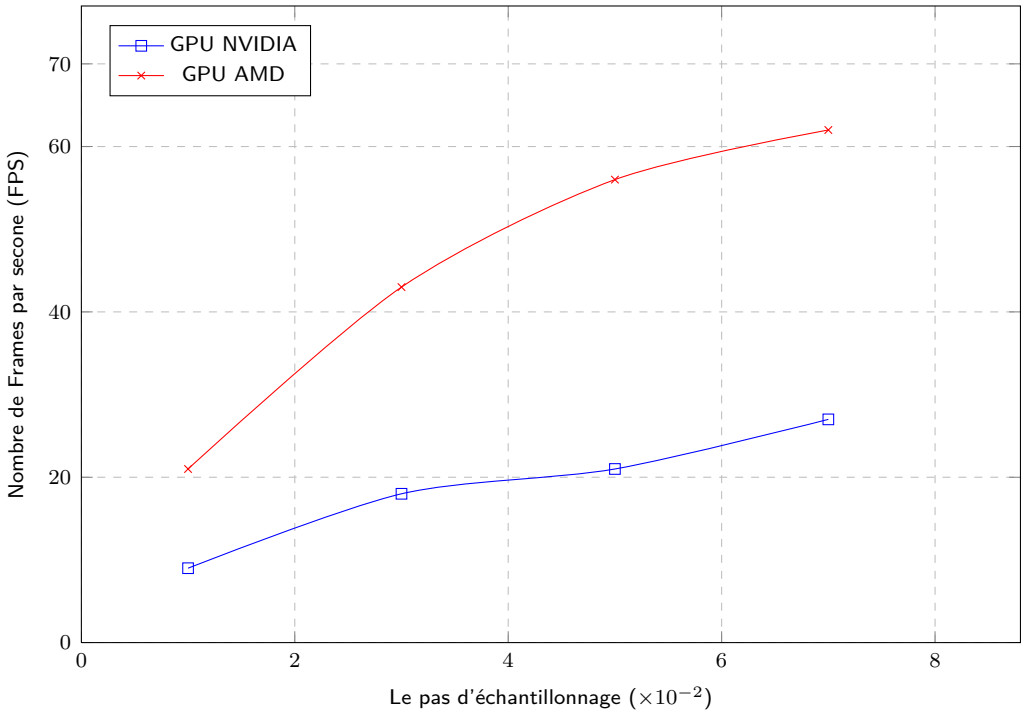


FIGURE 4.8 – Nombres de frames par seconde (FPS) en fonction de pas d'échantillonnage sur deux types de cartes graphiques.

4.5.6 Test selon la valeur de la seuil d'opacité

La technique d'accélération utilisé c'est la terminaison précoce de rayon "early ray termination" pour cela nous avons testé l'algorithme avec plusieurs valeurs d'opacités qui sont proche de 1. le tableau 4.5 et la figure 4.9 affichent l'effet d'application de cette technique sur le débit d'images.

DataSet	Seuil_Opacité	FPS(NVIDIA)	FPS(AMD)
DS1	0.9	11	19
	0.8	11	21
	0.7	13	22
	0.6	13	23
	0.5	14	24
DS3	0.9	08	18
	0.8	09	20
	0.7	10	21
	0.6	11	22
	0.5	13	24
DS4	0.9	11	23
	0.8	13	24
	0.7	14	25
	0.6	14	26
	0.5	15	29

TABLE 4.5 – Nombres de frames par seconde par la technique de “la terminaison précoce de rayon”

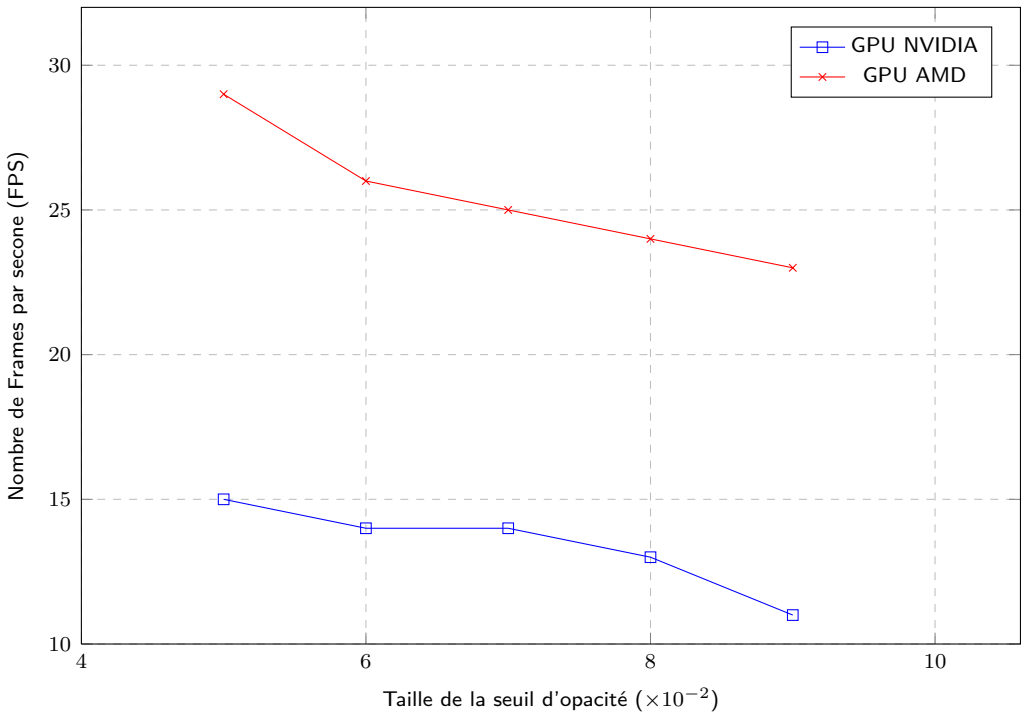


FIGURE 4.9 – Nombres de frames par seconde (FPS) en fonction de la taille de seuil d’opacité sur deux types de carte graphique.

4.6 Conclusion

Nous avons implémenté cet algorithme et nous avons présenté une série d'expérimentations sur des images 3D réelles. L'algorithme est assez rapide et répond à nos attentes (par exemple un temps d'exécution de 0.45 secondes pour générer une image de 512×512 pixels pour le volume d'un poumon de 14 millions de voxels). Même si l'algorithme raycasting est celui qui génère les images de meilleure qualité, il sert aussi comme un algorithme de base pour adapter notre approche aux besoins .

CONCLUSION ET PERSPECTIVES

Cette mémoire démontre que le matériel basé sur le raycasting est beaucoup plus que la conversion des algorithmes bien connus de la CPU vers le GPU. Les processeurs graphiques ont leurs propres forces et faiblesses, et d'exploiter ces forces, tout en évitant les faiblesses conduit à des techniques complètement différentes que dans les approches à base de CPU. Heureusement, l'évolution continue des cartes graphiques permettront des algorithmes encore plus efficaces dans un avenir proche, et avec la vitesse des GPU de plus en plus à un rythme beaucoup plus rapide que celle des processeurs, nous sommes à la recherche d'un avenir prometteur pour les approches à base de GPU .

Nous avons démontré que le modèle de programmation Opencl est adapté pour le volume raycasting et qu'un raycasting-tout basé sur Opencl pas une "solution rapide" peut être plus efficace que la mise en œuvre d'une base de shaders. Les facteurs influant sur l'accélération sont le type de GPU, la taille des blocs de threads, et la taille de l'ensemble de données. Nous avons également montré que l'utilisation de la mémoire partagée peut apporter une augmentation substantielle des performances lorsque les mêmes données de volume sont accessible à plusieurs reprises. Toutefois, les restrictions matérielles doivent être prises en compte, comme la gestion de la mémoire partagée et surtout la manipulation des voxels aux frontières peuvent introduire une surcharge importante. Des autres facteurs que les performances de rendu doivent être pris en compte aussi bien quand le choix d'un modèle de programmation pour une application raycasting. Une mise en œuvre de shader prend en charge un large éventail de matériel graphique, sans dépendre d'un seul fournisseur. Également l'intégration dans les cadres de rendu de volume existantes est plus facile, e. g., en étant capable d'utiliser directement des textures 2D et 3D et cibles de rendu d'OpenGL. Beaucoup de ces questions, nous l'espérons être éliminés par mise en oeuvre de la norme OpenCL, qui est indépendant du fournisseur et prend en charge le couplage étroit avec OpenGL.

Les travaux futurs en ce qui concerne les classes d'accélération GPU eux-mêmes peuvent être guidés par les questions en suspens mentionnées dans les exigences de la boîte à outils qui ne sont pas encore mises en œuvre, dans le futur nous avons l'intention d'utiliser le système de synchronisation de threads d'OpenCL pour des maillages non-convexes. Un autre travail futur consiste à utiliser la technique de pré-intégration partielle, afin d'améliorer la qualité du rendu. Aussi l'utilisation des techniques d'accélération comme Octree ,empty space skipping ...,etc afin d'améliorer la vitesse du rendu ,Chacune de ces méthodes mentionnés précédemment peuvent être combinés avec notre méthode GPU raycasting. Aussi l'utilisation de fonctions de transfert multidimensionnels était une nouvelle exigence. Aussi on peut pris en charge non seulement les ensembles de données de 8 bits mais aussi les ensembles les plus courantes de données de 16 bits devrait être l'une des premières choses à ajouter à la boîte à outils.

BIBLIOGRAPHIE

[Avila *et al.* 1992] Ricardo S. Avila, Lisa M. Sobierajski and Arie E. Kaufman. *Towards a Comprehensive Volume Visualization System.* In Proceedings of the 3rd Conference on Visualization '92, VIS '92, pages 13–20, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Baert 2012] Jeroen Baert. *Rendering Large Volumetric Datasets.* page 20. January 2012.

[Blinn 1982] James F. Blinn. *Light reflection functions for simulation of clouds and dusty surfaces.* SIGGRAPH Comput. Graph., vol. 16, no. 3, pages 21–29, July 1982.

[Blinn 1994] James F. Blinn. *Jim Blinn's Corner : Compositing. 1. Theory.* vol. 14, no. 5, pages 83–87, September 1994.

[Bruckner 2008] Stefan Bruckner. Efficient volume visualization of large medical datasets : Concepts and algorithms. VDM Verlag Dr. Muller Aktiengesellschaft & Co. KG, 2008.

[Cabral *et al.* 1994] Brian Cabral, Nancy Cam and Jim Foran. *Accelerated volume rendering and tomographic reconstruction using texture mapping hardware.* pages 91–98, 1994.

[Chaussumier *et al.* 1999] Frédérique Chaussumier, Frederic Desprez and Michel Loi. *Efficient Load-Balancing and Communication Overlap in Parallel Shear-Warp Algorithm on a Cluster of PCs.* In Proceedings of the 5th International Euro-Par Conference on Parallel Processing, Euro-Par '99, pages 570–577, London, UK, UK, 1999. Springer-Verlag.

[Cliff Woolley 2010] NVIDIA Developer Technology Group Cliff Woolley. *Introduction to OpenCL.* page 18. Khronos Group, 2010.

[Corporation 2002] Microsoft Corporation. *High-Level Shader Language. DirectX 9.0 Graphics, http://msdn.microsoft.com/directx*, 2002.

[Dehos 2012] Julien Dehos. *Séminaire :Introduction au calcul parallèle avec OpenCL*. page 29. January 2012.

[Díaz & Vázquez 2010] J. Díaz and P. Vázquez. *Depth-enhanced Maximum Intensity Projection*. In Proceedings of the 8th IEEE/EG International Conference on Volume Graphics, VG'10, pages 93–100, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[DirectX] Microsoft DirectX. *Microsoft. Site officiel de microsoft DirectX. , (Online Page), http://www.microsoft.com/france/msdn/technos/directx.mspx*.

[Drebin *et al.* 1988] Robert A. Drebin, Loren Carpenter and Pat Hanrahan. *Volume Rendering*. In Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '88, pages 65–74, New York, NY, USA, 1988. ACM.

[Engel *et al.* 2001] Klaus Engel, Martin Kraus and Thomas Ertl. *High-quality pre-integrated volume rendering using hardware-accelerated pixel shading*. pages 9–16, 2001.

[Engel *et al.* 2004] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Aaron E. Lefohn, Christof Rezk Salama and Daniel Weiskopf. *Real-time Volume Graphics*. In ACM SIGGRAPH 2004 Course Notes, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[Francis & Hill. 2000] S. Francis and J. Hill. *Computer Graphics Using OpenGL, 2nd Ed.* Prentice Hall, 2000.

[Gasparakis 1999] C. Gasparakis. *Multi-resolution Multi-field Ray Tracing : A Mathematical Overview*. In Proceedings of the Conference on Visualization '99 : Celebrating Ten Years, VIS '99, pages 199–206, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[Gouraud 1971] H. Gouraud. *Continuous Shading of Curved Surfaces*. IEEE Trans. Comput., vol. 20, no. 6, pages 623–629, June 1971.

[GPGPU] GPGPU. *Site pour la programmation GPGPU. 2007. , (Online Page), http://www.gpgpu.org*.

[Group] ARB/Khronos Group. *ARB/Khronos Group. Site officiel d'OpenGL., (Online Page), http://www.opengl.org*.

[Hadwiger *et al.* 2008] Markus Hadwiger, Patric Ljung, Christof Rezk Salama and Timo Ropinski. *Advanced Illumination Techniques for GPU Volume Raycasting.* In ACM SIGGRAPH ASIA 2008 Courses, SIGGRAPH Asia '08, pages 1 :1–1 :166, New York, NY, USA, 2008. ACM.

[Hastreiter 1999] P. Hastreiter. Registrierung und visualisierung medizinischer bilddaten unterschiedlicher modalitäten. Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung. Inst. für Mathematische Maschinen und Datenverarbeitung, 1999.

[He & Kaufman 1993] Taosong He and Arie E. Kaufman. *Virtual Input Devices for 3D Systems.* In Proceedings of the 4th Conference on Visualization '93, VIS '93, pages 142–148, Washington, DC, USA, 1993. IEEE Computer Society.

[He *et al.* 1996] Taosong He, Lichan Hong, Arie E. Kaufman and Hanspeter Pfister. *Generation of Transfer Functions with Stochastic Search Techniques.* In IEEE Visualization, pages 227–234, 1996.

[Jonsson 2005] M. Jonsson. *Volume rendering.* PhD thesis, M.Sc. Thesis, Umea University, Department of Computing Science, Sweden, October 2005.

[Kajiya & Von Herzen 1984] James T. Kajiya and Brian P Von Herzen. *Ray Tracing Volume Densities.* SIGGRAPH Comput. Graph., vol. 18, no. 3, pages 165–174, January 1984.

[Kay] *Kay and Kayjia : Algorithme de calcule l'intersection rayon avec une boite* http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm.

[Keys 1981] R. Keys. *Cubic convolution interpolation for digital image processing.* Acoustics, Speech and Signal Processing, IEEE Transactions on, vol. 29, no. 6, pages 1153–1160, 1981.

[Kilgard 2003] M. J Kilgard. *NVIDIA OpenGL Extension Specifications, (Online Page), NVIDIA Corporation.* http://www.nvidia.com/developer, 2003.

[Kirk] David Kirk. *The CUDA hardware model. , (Online Page),* http://courses.ece.uiuc.edu/ece498/al/lectures/lecture8-9-hardware.ppt.

[Klaus Engel 2003] Joe M. Kniss Aaron Lefohn et Daniel Weiskopf Klaus Engel Markus Hadwiger. *Interactive visualization of volumetric data on consumer pc hardware.* IEEE Visualization 2003 Full-Day Tutorial, 2003.

[Kruger & Westermann 2003] J. Kruger and R. Westermann. *Acceleration Techniques for GPU-based Volume Rendering.* In Proceedings of the 14th IEEE Visualization 2003 (VIS'03), VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society.

[Kruger 2003] R Kruger J. et Westermann. *Acceleration techniques for gpu-based volume rendering.* In IEEE Visualization Conference, page 287—292, October 2003.

[Lacroute & Levoy 1994] Philippe Lacroute and Marc Levoy. *Fast Volume Rendering Using a Shear-warp Factorization of the Viewing Transformation.* In Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '94, pages 451–458, New York, NY, USA, 1994. ACM.

[Levoy 1988] Marc Levoy. *Display of Surfaces from Volume Data.* IEEE Comput. Graph. Appl., vol. 8, no. 3, pages 29–37, May 1988.

[Levoy 1990] Marc Levoy. *Efficient Ray Tracing of Volume Data.* ACM Trans. Graph., vol. 9, no. 3, pages 245–261, July 1990.

[Lichtenbelt *et al.* 1998a] Barthold Lichtenbelt, Randy Crane and Shaz Naqvi. Introduction to volume rendering. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[Lichtenbelt *et al.* 1998b] Barthold Lichtenbelt, Randy Crane and Shaz Naqvi. Introduction to volume rendering. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[Max 1995] Nelson Max. *Optical Models for Direct Volume Rendering.* IEEE Transactions on Visualization and Computer Graphics, vol. 1, no. 2, pages 99–108, June 1995.

[Meissner *et al.* 1999] Michael Meissner, Ulrich Hoffmann and Wolfgang Strasser. *Enabling Classification and Shading for 3D Texture Mapping Based Volume Rendering Using OpenGL and Extensions.* In Proceedings of the Conference on Visualization '99 : Celebrating Ten Years, VIS '99, pages 207–214, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[Meissner *et al.* 2000] Michael Meissner, Jian Huang, Dirk Bartz, Klaus Mueller and Roger Crawfis. *A Practical Evaluation of Popular Volume Rendering Algorithms.* In Proceedings of the 2000 IEEE Symposium on Volume Visualization, VVS '00, pages 81–90, New York, NY, USA, 2000. ACM.

[Möller *et al.* 1996] Torsten Möller, Raghu Machiraju, Klaus Mueller and Roni Yagel. *Classification and Local Error Estimation of Interpolation and Derivative Filters for Volume Rendering.* In

Proceedings of the 1996 Symposium on Volume Visualization, VVS '96, pages 71–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[Mora 1986] Benjamin Mora. *Nouveaux algorithmes interactifs pour la visualisation de données volumiques*, 1986.

[Mueller & Crawfis 1998] Klaus Mueller and Roger Crawfis. *Eliminating Popping Artifacts in Sheet Buffer-based Splatting.* In Proceedings of the Conference on Visualization '98, VIS '98, pages 239–245, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Mueller *et al.* 1999] Klaus Mueller, Torsten Möller and Roger Crawfis. *Splatting Without the Blur.* In Proceedings of the Conference on Visualization '99 : Celebrating Ten Years, VIS '99, pages 363–370, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[Neumann 1993] Ulrich Neumann. *Volume Reconstruction and Parallel Rendering Algorithms : A Comparative Analysis.* PhD thesis, Chapel Hill, NC, USA, 1993. UMI Order No. GAX93-24080.

[Neumann 2001] Ulrich Neumann. *Volume Reconstruction and Parallel Rendering Algorithms : A Comparative Analysis.* PhD thesis, Th. : informatique : Toulouse 3 : 2001.

[nVIDIA] nVIDIA. *Nvidia Compute Unified Device Architecture programming guide. , (Online Page),* http://developer.download.nvidia.com/compute/cuda/0_81/NVIDIA_CUDA_Programming_Guide_0.8.2.pdf.

[Opencl & Munshi 2008] Khronos Opencl and Aaftab Munshi. The opencl specification version : 1.0 document revision : 48, page 22. 2008.

[Pfister *et al.* 1999] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer and L. Seiler. *The VolumePro Real-Time Ray-Casting System.* pages 251–260, 08/1999 1999.

[Phong 1975] Bui Tuong Phong. *Illumination for Computer Generated Pictures.* Commun. ACM, vol. 18, no. 6, pages 311–317, June 1975.

[rawDatasets] rawDatasets. *The following raw datasets from medical 3D scanners for download. , (Online Page),* http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/datasets.html.

[rawDatasetsNew] rawDatasetsNew. *The following new raw datasets from medical 3D scanners for download. , (Online Page),* http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/new.html.

[Rezk-Salama *et al.* 2000a] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner and T. Ertl. *Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization.* pages 109–118, 2000.

[Rezk-Salama *et al.* 2000b] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner and T. Ertl. *Interactive Volume on Standard PC Graphics Hardware Using Multi-textures and Multi-stage Rasterization.* In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, HWWS '00, pages 109–118, New York, NY, USA, 2000. ACM.

[Rezk-Salama 2001] Christof Rezk-Salama. *Volume Rendering Techniques for General Purpose Graphics Hardware.* PhD thesis, Technischen Fakultät, Universität Erlangen-Nürnberg,Erlangen, Germany, 2001.

[Roettger *et al.* 2003] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl and Wolfgang Strasser. *Smart Hardware-accelerated Volume Rendering.* In Proceedings of the Symposium on Data Visualisation 2003, VISSYM '03, pages 231–238, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[Scarpino 2011] M. Scarpino. Opencl in action : How to accelerate graphics and computation, page 323. Manning Publications Company, 2011.

[Scharsach 2005] Henning Scharsach. *Advanced GPU Raycasting.* In In Proceedings of CESCG 2005, pages 69–76, 2005.

[Stegmaier *et al.* 2005] Simon Stegmaier, Magnus Strengert, Thomas Klein and Thomas Ertl. *A simple and flexible volume rendering framework for graphics-hardware-based raycasting.* pages 187–195, 2005.

[Upson & Keeler 1988] Craig Upson and Michael Keeler. *V-buffer : Visible Volume Rendering.* SIGGRAPH Comput. Graph., vol. 22, no. 4, pages 59–64, June 1988.

[Van Gelder & Kim 1996] Allen Van Gelder and Kwansik Kim. *Direct Volume Rendering with Shading via Three-dimensional Textures.* In Proceedings of the 1996 Symposium on Volume Visualization, VVS '96, pages 23–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[Voorhies & Foran 1994] Douglas Voorhies and Jim Foran. *Reflection Vector Shading Hardware.* In Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '94, pages 163–166, New York, NY, USA, 1994. ACM.

[Wallis *et al.* 1989] J.W. Wallis, Tom R. Miller, C.A. Lerner and E.C. Kleerup. *Three-dimensional display in nuclear medicine.* Medical Imaging, IEEE Transactions on, vol. 8, no. 4, pages 297–230, Dec 1989.

[Weiskopf 2006] Daniel Weiskopf. Gpu-based interactive visualization techniques (mathematics and visualization). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Westermann & Ertl 1998] Rüdiger Westermann and Thomas Ertl. *Efficiently using graphics hardware in volume rendering applications.* pages 169–178, 1998.

[Westover 1989] Lee Westover. *Interactive Volume Rendering.* In Proceedings of the 1989 Chapel Hill Workshop on Volume Visualization, VVS '89, pages 9–16, New York, NY, USA, 1989. ACM.

[Westover 1990] Lee Westover. *Footprint Evaluation for Volume Rendering.* In Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '90, pages 367–376, New York, NY, USA, 1990. ACM.

[WikipediaTrilInterp 2015] WikipediaTrilInterp. *Trilinear interpolation, (Online Page), Wikipedia.* http://en.wikipedia.org/wiki/Trilinear_interpolation, January 2015.

[WikipediaZbuffering 2015] WikipediaZbuffering. *Z-buffering algorithm description, (Online Page), Wikipedia.* http://en.wikipedia.org/wiki/Z-buffering, May 2015.

[Wilhelms & Van Gelder 1991] Jane Wilhelms and Allen Van Gelder. *A Coherent Projection Approach for Direct Volume Rendering.* In Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '91, pages 275–284, New York, NY, USA, 1991. ACM.

[Wittenbrink *et al.* 1998] Craig M. Wittenbrink, Thomas Malzbender and Michael E. Goss. *Opacity-weighted Color Interpolation, for Volume Sampling.* In Proceedings of the 1998 IEEE Symposium on Volume Visualization, VVS '98, pages 135–142, New York, NY, USA, 1998. ACM.

Annexes

ANNEXE A

L'IMAGERIE MÉDICALE

A.1 Introduction

L'imagerie médicale est non seulement un outil de diagnostic puisqu'elle apporte des informations topographiques (angiographie, échographie, tomodensitométrie) et fonctionnelles mais elle est aussi un outil de suivi thérapeutique ainsi qu'une thérapeutique dans le cadre de l'imagerie interventionnelle.

A.2 Les méthodes avec le principe physique associé

Les méthodes d'imageries médicales sont nombreuses et utilisent plusieurs types de procédés physiques que sont :

- Les rayons X,
- Les ultrasons,
- L'émission de rayonnement par des particules radioactives,
- Le magnétisme du noyau des atomes.

Le tableau ci-dessous indique quelques unes des méthodes avec le principe physique associé, le type d'images obtenues, les parties du corps explorées et le type de récepteur.

	Méthodes	Procédé	Type d'image obtenue	Partie du corps explorée	Type de récepteur
Radiologie	Radiographie	Rayon X	Image statique, projection du volume du corps.	poumons, abdomen, squelette, seins.	Film radiographique.
	Radioscopie		Image dynamique, projection du volume du corps.	poumons, abdomen, squelette.	Intensificateur d'image radiologique ou amplificateur de luminance
	Radiographie numérique		Image statique, projection du volume du corps.	poumons, abdomen, squelette, seins.	Ecrans radioluminescents à mémoire (ERLM), Détecteurs plans numériques (DPM)
Ultrasonographie	Echographie	Ultrasons	Image dynamique , coupes.	Abdomen, coeur, seins, muscles et tendons.	Sonde
	Doppler			Vaisseaux sanguins.	
Scanner ou tomodensitomètrie	Scanner	Rayon X	Image statique, coupes.	Toutes	Détecteurs solides ou gazeux
	Scanner hélicoïdal				
Imagerie par résonance magnétique	IRM	RMN	Image statique, coupes.	Presque toutes	Antennes
	Angio-RM		Image 3D	Vaisseaux sanguins	
	Imagerie de diffusion		Coupes	Cerveau, foie	
	Imagerie fonctionnelle		Image dynamique, coupes.	Cerveau	
Imagerie vasculaire	Artériographie	Rayon X	Image dynamique, projection du volume du corps	Vaisseaux sanguins	Intensificateur d'image
	Angiographie				
	Angiographie numérisée				
Médecine nucléaire	Scintigraphie	Emission de rayonnements	Image statique, projection du volume du corps.	Toutes	Gamma caméra
	Tomographie par émission de positons	Positons	Coupes		
	SPECT		Coupes	Cerveau	

Table A.1 – Les méthodes avec le principe physique associé

ANNEXE B

UTILISATION DES CARTES GRAPHIQUES POUR LA VISION PAR ORDINATEUR

B.1 Introduction

L'utilisation de l'informatique dans le monde qui nous entoure est devenue banale. L'homme essaie d'inculquer à des machines une vision humaine et même une vision plus performante pour permettre à une machine de réaliser ce que l'humain ne peut pas faire. Dans le monde de la robotique, de la médecine, de la domotique ou même du ludique, la vision est un domaine de réalisation important. Étant donné que notre vision se fait en « temps réel », c'est-à-dire de façon quasi-instantanée, il est alors nécessaire d'essayer d'obtenir un traitement des images le plus rapide possible. Les organes des machines qui nous permettent de traiter ces images sont un ou plusieurs processeurs et une ou plusieurs cartes graphiques.

B.2 Vision par ordinateur

L'objectif de la vision par ordinateur, appelée aussi vision artificielle, est de reproduire la vision humaine à l'aide d'un ordinateur. Le dispositif général pour la vision par ordinateur peut être illustré par la figure B.1.

Les éléments importants sont donc :

- **la scène :** elle correspond à ce qui est vu, c'est l'environnement qui est perçu,

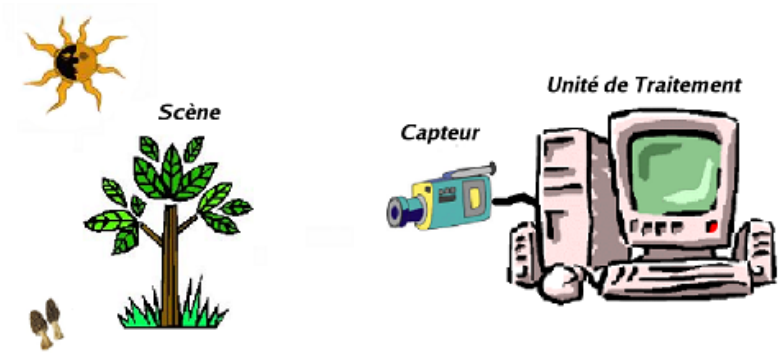


FIGURE B.1 – Dispositif pour la vision par ordinateur

- **le capteur :** c'est le dispositif matériel qui permet d'obtenir les images. Par analogie, c'est ce qui correspond à l'œil humain. Le capteur peut être une ou plusieurs caméras, un ou plusieurs appareils photographiques, etc,
- **l'unité de traitement :** c'est là où est réalisé le traitement des données. C'est dans cette unité de traitement que se trouve la carte graphique.

B.3 Carte graphique

FIGURE B.2 – Une carte graphique.

La carte graphique est un composant d'une unité de traitement qui permet de convertir des données numériques en données graphiques pouvant être affichées sur un périphérique de sortie (écran, rétroprojecteur, etc.). La carte graphique possède les composants suivants :

- **le processeur graphique :** il est aussi appelé *GPU (Graphical Processing Unit)*, c'est le cœur de la carte graphique. C'est là où sera réalisé tout le traitement d'une image. Il comporte le

pipeline graphique mais aussi des unités de calcul arithmétiques et logiques,

- **la mémoire vidéo (*frame buffer*) :** c'est la mémoire qui permet de stocker l'image avant qu'elle ne soit affichée,
- **le RAMDAC :** il correspond aux initiales de *Random Access Memory Digital-Analog Converter*. Ce composant permet de transformer les données numériques d'une image, stockée dans la mémoire vidéo, en signal analogique pour être affiché,
- **le bios vidéo :** cette mémoire contient des informations propres à la carte graphique,
- **le bus :** c'est le connecteur qui permet de relier la carte graphique à une carte mère. Le bus AGP était un bus créé à cet effet, mais il est aujourd'hui remplacé par le bus PCI Express,
- **les connectiques :** elles permettent de relier la carte graphique à un écran, à un téléviseur, etc.

B.4 Contraintes et usages du matériel graphique

Pour bien comprendre les enjeux derrières 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 en section B.4.1, puis nous aborderons l'architecture interne de ce matériel en section B.4.2

B.4.1 Description du pipeline graphique

Présentation

Le rôle des processeurs graphiques tels que nous les connaissons aujourd'hui est d'effectuer le traitement de l'ensemble des étapes qui forment le pipeline de rendu graphique temps réel. Le modèle de rendu utilisé est basé sur l'algorithme du *Z-Buffer* proposé par `Edwin Catmull` en 1974 [WikipediaZbuffering 2015]. 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.

C'est ce pipeline que les librairies graphiques OpenGL [Group] et Direct 3D [DirectX] permettent de contrôler, son schéma de principe général correspondant au dernier modèle exposé par ces API est présenté figure B.3.

Description rapide

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

Opérations par sommets Ces sommets sont ensuite transformés afin d'effectuer l'ensemble des changements de repère nécessaires à leur placement spatial. Ils sont également projetés en espace image via une projection perspective ou parallèle orthographique.

Assemblage Une fois transformés et projetés, les sommets sont regroupés et assemblées en primitives reconnues par le système, il s'agit généralement de triangles.

Rasterisation L'étape de rasterisation est ensuite chargée de 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.

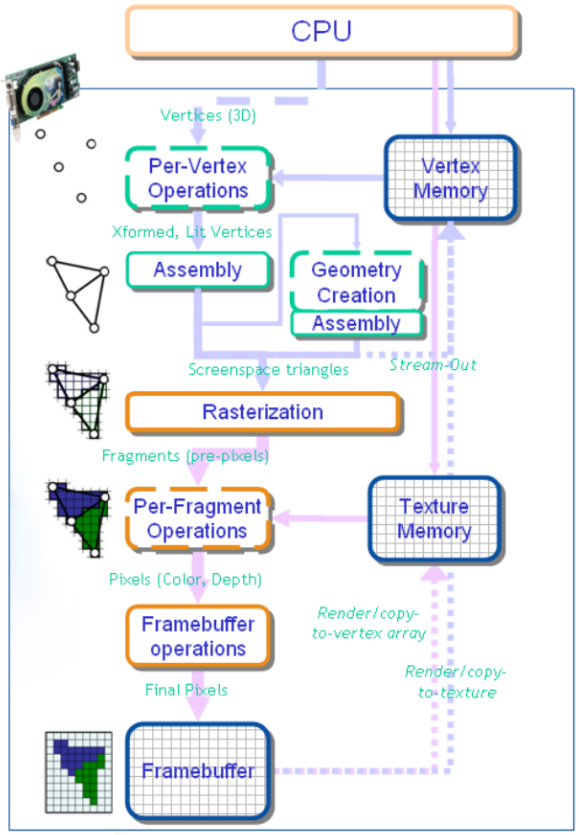


FIGURE B.3 – Schéma de principe du pipeline de rendu temps réel.

Opérations en espace image L'écriture des fragments dans le framebuffer (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.

Les shaders programmables Les shaders sont des étapes de traitement du pipeline totalement programmables, elles apparaissent en pointillé dans le schéma B.3. 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).

B.4.2 Architecture matérielle

Évolution

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 présenté section B.4.1. 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 générations de shaders

L'évolution des GPU a été marquée par plusieures étapes d'évolution des capacités des shaders programmables. Les GPU de premières génération (c'est le moment ou les cartes 3D sont devenues programmables et ont été appelées GPU) implémentaient le premier modèle de shaders appelé shaders model 1.0. Cette premier génération ne fournissait qu'une étape de vertex shader réellement programmables, l'ancêtre des fragment shaders alors appelé register combiner était simplement dédié au paramétrage de 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 et ainsi que des boucles dynamiques. La dernière génération de shaders introduite lors de la sortie du G80 présenté section B.4.3 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.

Mémoires embarquées

En plus de ces capacités de calcul, les GPU disposent de leur propre mémoire embarquée. Celle-ci a généralement une taille de l'ordre de 500Mo et permet de stocker 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. Ces caches sont optimisées pour accélérer les opérations de lecture spatialement proches ce qui est généralement le cas lors du traitement de primitives graphiques.

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 (de l'ordre de 8Go/s) mais ne permet en pratique, du fait des limitations du matériel, qu'un débit situé 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 théoriques)

B.4.3 Les architectures de dernière génération

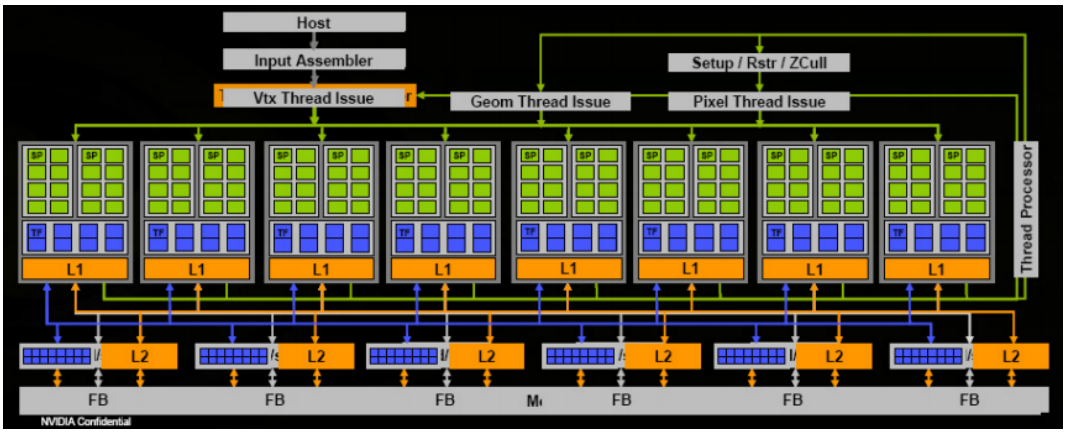


FIGURE B.4 – Diagramme de bloc de l'architecture du G80, dernière génération de GPU du constructeur NVidia constitué de 16 multi-processeurs formés chacun de 8 unités de calcul et regroupés par 2 dans 8 clusters de texture.

Le G80, dernière génération de GPU introduite en novembre dernier par le constructeur NVidia marque 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 œuvre les différentes étapes programmable du modèle de pipeline graphique exposé par les API (OpenGL et Direct3D).

Une vue globale de cette architecture est présentée figure B.4.

Les multi-processeurs

Les unités de calcul appelées Stream Processors (SP) sont regroupées par grappes (appelées MultiProcesseurs) de 8 unités. Chaque multi-processeur fonctionne en SIMD (Single Instruction Multiple Data 1) 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 multi-processeur 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. 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.

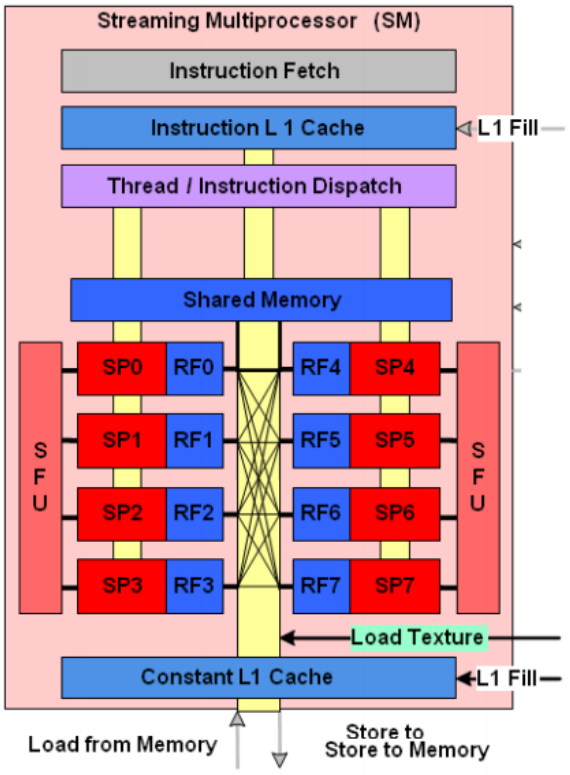


FIGURE B.5 – Schéma de principe d'un multi-processeur du G80. (Source : [Kirk])

Les stream-processeurs

Les Stream Processors sont dédiés aux opérations d'addition et de multiplication flottantes ou entières opérant sur des scalaires 32bits. Les multi-processeurs sont en plus dotés de deux SFU (Super Function

Unit) dédiés 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).

Les clusters de texture

Les clusters de texture regroupent les multi-processeurs pour l'accès aux ressources de placage de textures du matériel.

L'interface CUDA

L'ensemble de cette architecture de calcul peut être accédé directement via l'API CUDA fournie par NVidia. En plus de permettre l'accès aux ressources de calcul et de stockage du GPU, cette API expose un modèle de calcul de flux qui permet de tirer partie du parallélisme de ce genre d'architecture [nVIDIA].

B.5 Programmation GPGPU

La programmation GPGPU (*General-Purpose Computation Using Graphics Hardware*) [GPGPU] consiste à utiliser les différents éléments du GPU pour pouvoir arriver à un résultat recherché. Du point de vue du développeur, la programmation avec la carte graphique revient à traiter un flux de données qui passe par le pipeline graphique. Il peut programmer le vertex, le fragment et le geometry processor. Pour cela, plusieurs méthodes peuvent être mises en œuvre. La première façon de procéder est la méthode du Read Back (voir figure B.6) : le CPU envoie les données au pipeline qui sont traitées par les différentes unités programmées. Le CPU peut alors récupérer l'image stockée dans le frame buffer.

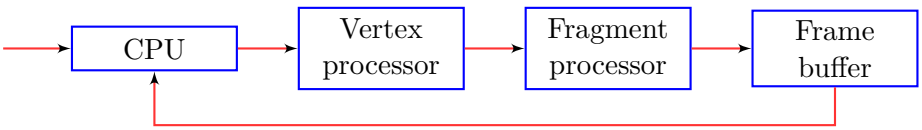


FIGURE B.6 – Read Back.

La deuxième façon de faire est d'utiliser les textures (*Copy to Texture*) ce qui permet d'éviter de repasser par le CPU lors de la réalisation de plusieurs passes (voir figure B.7). De plus, le débit des données, a‘ la sortie du frame buffer, est supérieur au débit que le bus peut assurer. Le bus est alors un « goulot d'étranglement ». L'image contenue dans le frame buffer est copiée dans l'unité de texture.

Cette possibilité existe depuis les cartes graphiques de type NV40 (NVIDIA GeForce 6) et R520 (ATI X1300), donc depuis les années 2004-2005. Notons que l'utilisation du vertex processor est alors « courtcircuitée » lors de l'utilisation en plusieurs passes du fragment processor.

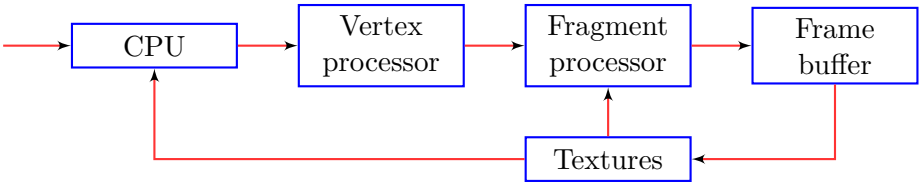


FIGURE B.7 – Copy to Texture.

On peut aussi éviter de passer par le frame buffer en copiant l'image, restituée par le fragment processor, directement dans une texture : c'est la méthode du *Render to Texture* (voir figure B.8). Il devient inutile de passer par le frame buffer si on ne veut pas afficher l'image à l'écran.

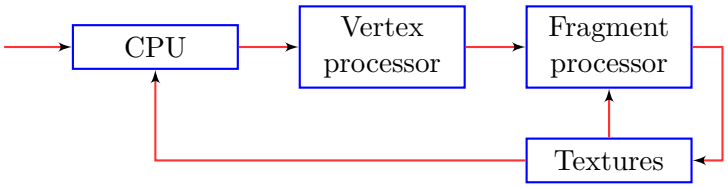


FIGURE B.8 – Render to Texture.

Une autre méthode est d'utiliser le VBO (*vertex buffer object*) (voir figure B.9). Le VBO est très similaire au vertex array mais avec plus de modes de transfert des données. Un vertex array est le tableau des sommets qui sera traité par le vertex processor. Cette méthode permet de réutiliser le vertex processor.

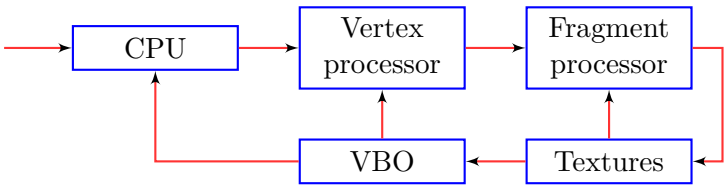


FIGURE B.9 – Copy to VBO.

On peut copier directement l'image issue du fragment processor dans le VBO : c'est la méthode du *Render to VBO* (voir figure B.10). Cette méthode permet d'éviter le transfert des donné processor vers textures, puis textures vers VBO.

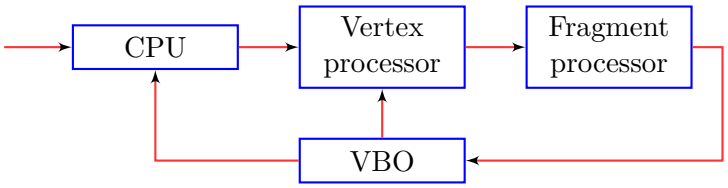


FIGURE B.10 – Render to VBO.

Avec l'arrivée de la carte graphique GeForce 8 de NVIDIA, on a donc le geometry shader en plus. On peut récupérer les données traitées juste après le geometry processor.

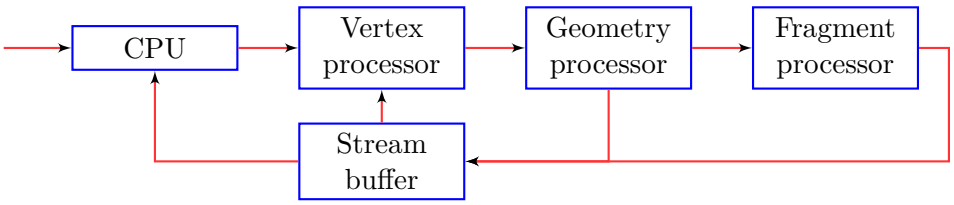


FIGURE B.11 – G80 : render to Stream Buffer.

Le tableau B.1 donne l'utilisation des vertex shader, fragment shader, textures, VBO et du frame buffer selon les méthodes qui ont été présentées.

Méthodes	Plusieurs passes	Vertex processor	Geometry processor	Fragment processor	Textures	VBO	Frame buffer	Stream buffer
Read Back	non	x		x			x	
Copy to Texture	oui			x	x		x	
Render to Texture	oui			x	x			
Copy to VBO	oui	x		x	x	x		
Render to Render	oui	x		x		x		
Render to Stream Buffer	oui	x	x	x				x

TABLE B.1 – Méthodes de programmation GPGPU.

La programmation GPGPU offre le concept de flux (stream) et de noyau (kernel) : le *Stream Computing Model*. Le flux correspond à l'ensemble des données qui vont être traitées. Les données du flux sont indépendantes entre elles. Le noyau correspond au traitement à appliquer à chaque donnée. Pour parvenir au résultat recherché, il est souvent nécessaire d'utiliser plusieurs noyaux (voir figure B.12). Les données vont être chargées souvent dans une texture ou dans le VBO. Le noyau sera exécuté par le fragment processor et le vertex processor.

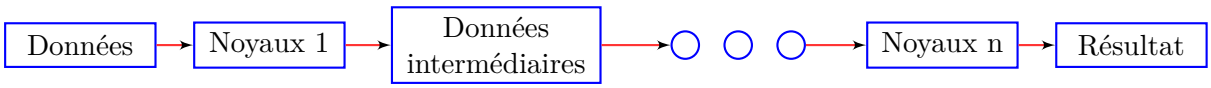


FIGURE B.12 – GPGPU : utilisation de plusieurs noyaux.

ANNEXE C

L'API OPENCL

Le noyau de ce projet se fonde sur la mise en place d'une rendu volumique 3D utilisé OpenCL. Pour la mise en place, une bonne compréhension d'api est exigée. Le section suivant est une introduction courte à OpenCL qui présente les aspects clé d'api.

OpenCL est une api conçu pour la programmation parallèle d'usage universel. L'api abrège les nuances entre les différents genres de matériel et permet au même programme d'être exécuté sur les machines multiples. OpenCL est un outil intuitif pour l'amélioration de calcul. Il convient aux infographies mais également au calcul non graphique, tel que les programmes scientifiques qui exigent l'informatique à haute performance. L'api influence la puissance de GPU et de CPU multi-core et permet à des threads(kernels) de fonctionner en parallèle sur les noyaux (cores) de multiples des dispositifs (devices).

C.1 Terminologie d'OpenCL

L'api OpenCL comporte une terminologie qui nous aidera à comprendre son exécution. Parmi les divers concepts le plus important est : le dispositif (device), le hôte (host), le thread (kernel), les objets de mémoire et les éléments-fonctionner (work-items).

C.1.1 Dispositif (Device)

Le matériel sur lequel OpenCL exécute les instructions en parallèle s'appelle "dispositif" (device). Chaque dispositif contient plusieurs unités de calcul (compute-units ou cores) : une unité de matériel

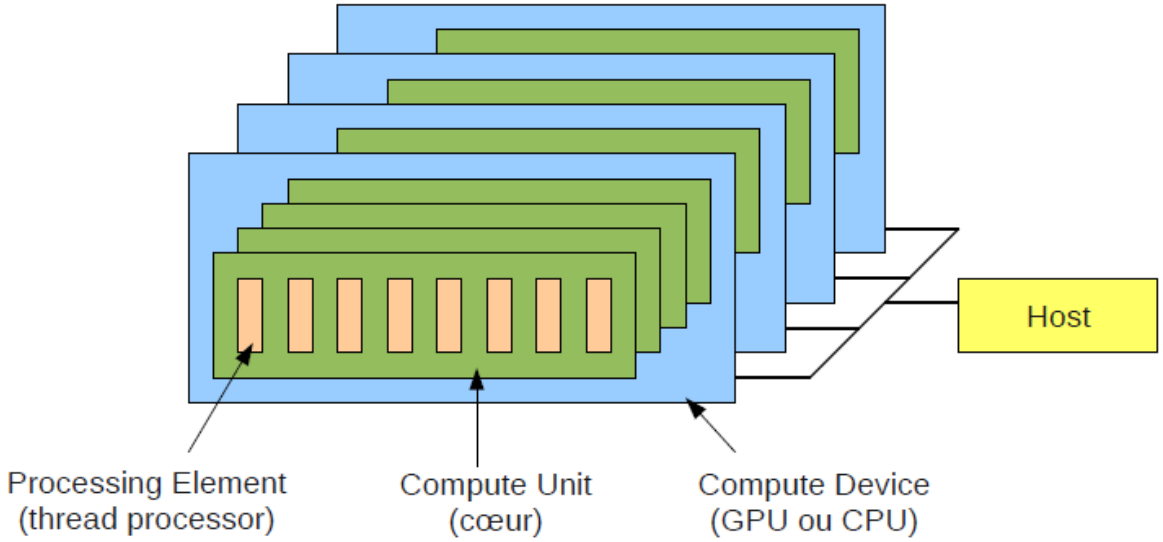


FIGURE C.1 – Le modèle de la plate-forme d'OpenCL [Dehos 2012]

capable des calculs indépendants. La capacité d'un dispositif pour effectuer des calculs efficaces avec OpenCL dépend du nombre d'unités de calcul. Plus que nous avons des unités de calcule (compute-units), plus des calculs simultanément que nous obtenons. Une CPU contient habituellement 2 à 8 unités différentes de calcul (noyaux), alors qu'un GPU moderne peut contenir des dix aux centaines d'unités de calcul. C'est pourquoi les GPUs sont plus appropriés aux calculs d'OpenCL, même si un programme OpenCL fonctionner de la même manière sur un CPU, seul le temps de calcul serait différent.

C.1.2 Application hôte (Host application)

Le rôle de l'application hôte est d'appeler les fonctions externes d'OpenCL pour effectuer des opérations sur le dispositif. Ces fonctions peuvent être utilisées, par exemple, pour installer le contexte d'OpenCL, ou pour déclencher l'exécution des programmes d'OpenCL sur le dispositif. Le dispositif sur lequel l'application hôte fonctionne s'appelle le dispositif de hôte (the host device). Dans la plupart des cas l'application hôte fonctionne sur la CPU tandis que les programmes d'OpenCL fonctionnent sur le GPU. Dans certains cas, le dispositif hôte peut être le même dispositif que celui d'exécuter le calcul OpenCL, lorsque le dispositif OpenCL et le dispositif d'hôte sont à la fois le CPU par exemple.

C.1.3 Les Threads (Kernels)

Nous nous référons aux programmes qui sont exécutés sur le dispositif comme "Kernels". Leur utilisation est en quelque sorte similaire à des programmes de shader GLSL (une fonction exécutée en parallèle pour chaque élément d'un ensemble de données). La principale différence est que les données de sortie ne peuvent pas être liés à l'infographie (autre qu'un sommet ou un fragment). Le langage d'OpenCL est une variation du langage C juste comme GLSL. Afin d'être personnalisé pour chaque genre de dispositif, les programmes de "Kernels" doivent être compilés sur l'application hôte avant d'être exécuté sur le dispositif.

Quoiqu'on exécute le même programme en parallèle il ne signifie pas qu'on exécute les mêmes instructions en parallèle. Par exemple, lorsque vous utilisez "if", les actions qui correspondent aux déclarations seront effectuées uniquement par les threads satisfaisant à ces conditions. Ceci permet beaucoup de flexibilité pour le programme, car chaque threads peut avoir son propre comportement. Dans certains cas, lorsque le programme du noyau devient plus complexe, une attention particulière doit être prise lors de l'écriture du programme pour éviter les artefacts de traitement parallèle. Dans OpenCL, ces objets seraient causés, par exemple, lorsque plusieurs threads tentent d'accéder à la même partie de la mémoire en même temps. Dans ce cas, nous pourrions risquer d'effacer une valeur importante ou en lisant une valeur hors de date.

C.1.4 Work-items

Le but d'OpenCL est d'exécuter des programmes de noyau en parallèle. Par conséquent, le même programme sera exécuté plusieurs fois en même temps. Chaque instance du programme est appelé "Work-items". Un élément de travail d'un programme "Work-items" de noyau est exécuté une seule fois et sur une unité de calcul du dispositif. Il est très important d'obtenir le sens de l'élément de travail parce que la logique d'OpenCL réside dans l'organisation de ces éléments. Les Work-items sont similaire à la notion des threads dans le calcul parallèle. Dans les autres api similaires, tels que les shaders ou CUDA, les Work-items sont appelés "threads". Les Work-items peuvent être synchronisés ou partager des informations avec d'autres Work-items. Ces opérations, cependant, ne sont possibles que entre les Work-items du même groupe. C'est pourquoi OpenCL nous permet d'organiser les Work-items dans les groupes de travail "work-groups". Dans la section suivante sur le modèle d'exécution OpenCL, nous allons voir comment une telle organisation peut être structuré.

C.1.5 Objets de mémoire

Les objets de mémoire sont un aspect très important du calcul d'OpenCL, ils permettent aux programmes du noyau pour lire et écrire des valeurs à la mémoire du dispositif. Les objets de mémoire qui doivent être partagées entre l'hôte et l'application de dispositif sont déclarés de l'application hôte, mais stockées sur la mémoire du dispositif. OpenCL fournit quelques fonctions d'API pour lire et écrire aux objets de mémoire de dispositif à partir de l'application hôte. Ils peuvent être utilisés, par exemple, lorsque l'on souhaite transférer le résultat d'un calcul à l'hôte pour afficher les résultats. Les objets de mémoire peuvent soit conserver des données non typées (valeur ou un tableau de valeurs) ou des données d'image (images 2D ou 3D).

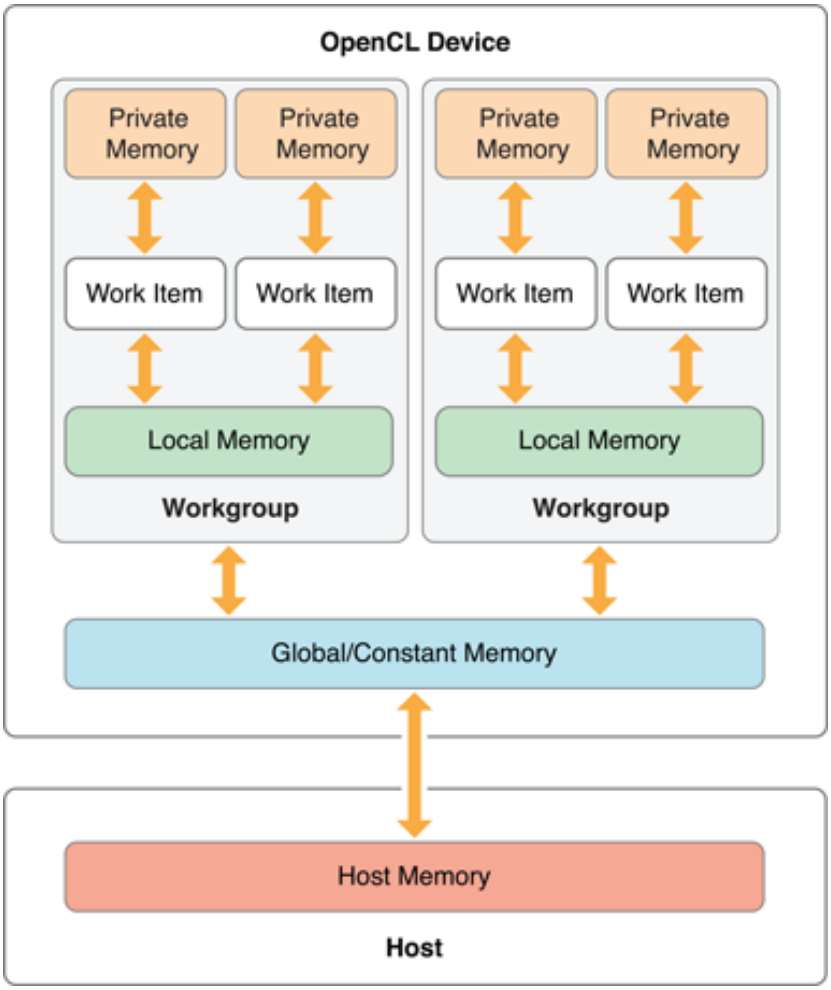


FIGURE C.2 – Schéma représente l'api OpenCL [Cliff Woolley 2010]

C.2 Le modèle de la mémoire d'OpenCL

Le modèle objet de mémoire décrit les différents domaines des objets de mémoire. Chaque domaine a ses propres propriétés et définit le comportement des objets de mémoire. Dans OpenCL, un objet de mémoire peut être global, local, constante ou privé (global, local, constant or private).

C.2.1 Global

Un objet de mémoire globale peut être consulté (lire et écrire) par n'importe quel work-item dans n'importe quelle work-group, juste comme la mémoire globale d'un programme C. En comparaison avec les objets de mémoire non globale, la mémoire globale a haute latence d'accès. C'est pourquoi le développeur doit être prudent lors de l'écriture d'un noyau, de ne pas surcharger la bande passante en accédant à la mémoire globale trop fréquemment. La meilleure pratique serait d'utiliser les données en cache à la place.

C.2.2 Constant

Les objets constants de mémoire peuvent être consultés par tous les work-items, comme l'objet global de mémoire, mais il permet juste l'accès en lecture seule.

C.2.3 Local

Un objet de mémoire défini comme local est accessible seulement aux work-items dans la même work-group (chaque work-group aurait son propre tampon locale). Le temps d'accès à la mémoire locale est beaucoup plus petit que la mémoire globale. Par conséquent, il est plus sage d'utiliser une mémoire locale de partager des valeurs entre les threads plutôt que d'utiliser la mémoire globale.

C.2.4 Privé

L'objet de mémoire qui est déclaré comme privé est propre à chaque work-item et ne peut être consulté par tous les autres work-items. Par défaut, si aucun type est spécifié pour un objet de mémoire, celui-ci est déclaré comme privé.

C.3 Le modèle d'exécution d'OpenCL

Dans cette section nous nous expliquerons brièvement le modèle d'exécution dans OpenCL. Le modèle d'exécution définit comment les work-items sont organisés. Une organisation logique est une tâche

importante ; elle définit comment sera exécuté le programme du noyau sur le dispositif.

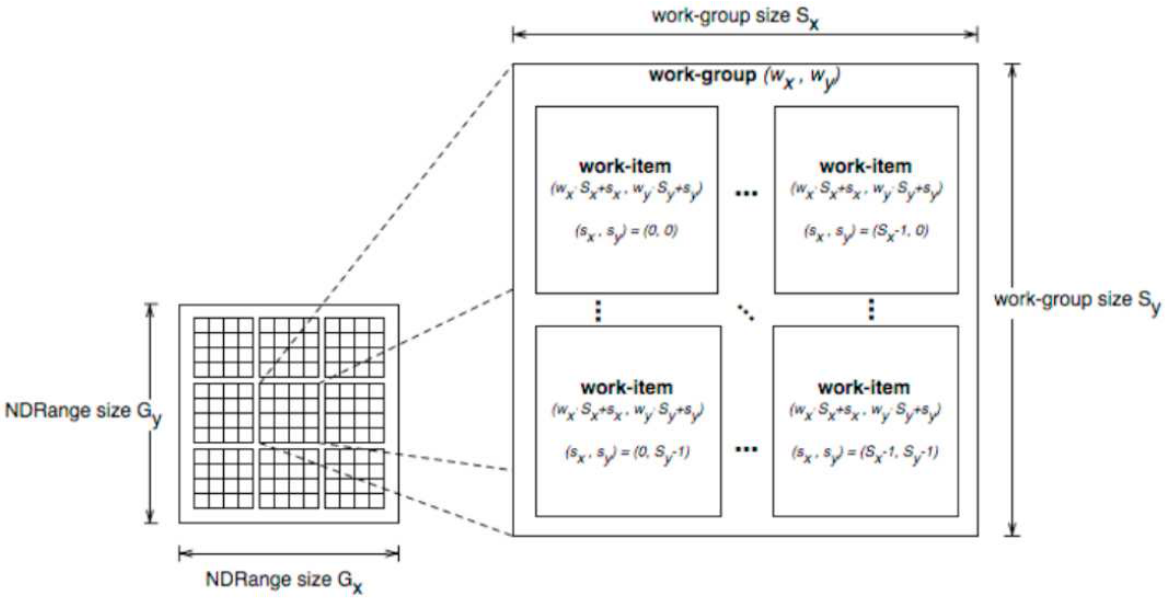


FIGURE C.3 – Le modèle d'exécution parallèle d'OpenCL [Opencl & Munshi 2008]

Le but d'OpenCL est d'exécuter des instructions en parallèle. OpenCL exploite chaque unité de calcul d'un dispositif pour lancer les programmes de kernel. Lors de l'exécution d'un programme de noyau, le dispositif hôte envoie autant d'instances du programme que nécessaire pour le dispositif. Les work-items sont structurés en groupes, car il est généralement peu probable que le dispositif peut gérer le calcul de tous les work-items en même temps. Le groupement des work-items permet également de partager la mémoire et de synchroniser les work-items dans la même work-group. Le partage est rendu possible avec l'utilisation des objets de mémoire locale et la synchronisation avec la barrière et les fonctions de clôture de mémoire. Chaque work-item est défini par un indice global et local, et l'espace d'indice peut comprendre en 1,2 ou 3 dimensions, selon les besoins de l'algorithme, L'identifiant globale d'un work-item représente l'indice parmi tous les autres work-items, tandis que l'identification locale se réfère à l'indice dans le work-group seulement. Cette information peut être consultée dans un programme de noyau à travers des fonctions intégrées. Par défaut, l'organisation des work-groups est géré par OpenCL. On peut aussi définir le nombre et la taille des work-groups manuellement, cependant, ces propriétés ne doivent pas être supérieure à la taille maximale autorisée par le dispositif.

C.4 L'interopérabilité OpenCL/OpenGL

Cette section explore comment réaliser l'interopérabilité entre OpenCL et OpenGL. L'interopérabilité OpenCL/OpenGL est une puissante de fonctionnalité qui permet aux programmes pour partager des données entre OpenGL et OpenCL.

C.4.1 Partage des données entre OpenGL et OpenCL

Le module de Données d'OpenGL utilisé trois structures de données : les objets de mémoire tampon (VBOs), des objets de texture, et des objets de renderbuffer. De même, les applications d'OpenCL accèdent à des données utilisé deux structures : objets de mémoire tampon et objets d'image. Le concept fondamental étant à la base de l'interopérabilité OpenGL-OpenCL est que les objets de mémoire d'OpenCL peuvent partager des données avec des structures de données d'OpenGL. Ceci permet à des noyaux de traiter des structures d'OpenGL comme s'elles étaient les objets réguliers de mémoire tampon et les objets d'image. La figure C.4 illustre ceci graphiquement. Comme représenté dans la figure,

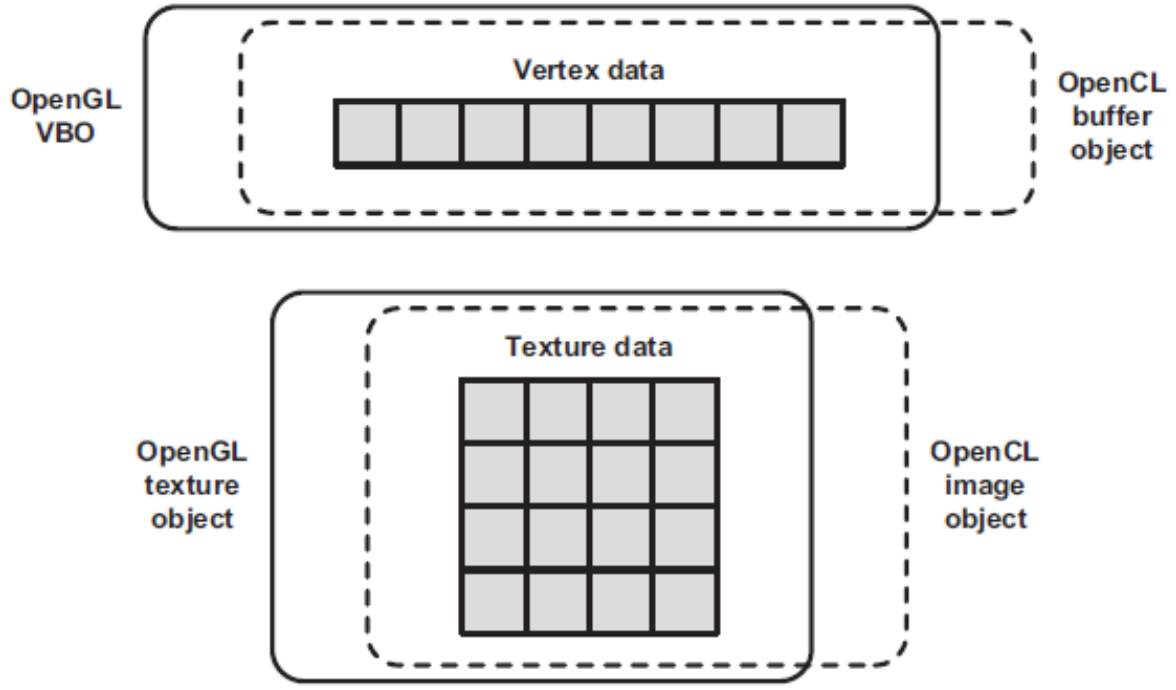


FIGURE C.4 – Les objets de mémoire d'OpenCL (des objets de mémoire tampon et des objets d'image) partagent des données avec des objets d'OpenGL (VBOs et objets de texture) [Scarpino 2011]

l'application ne transfère pas les données entre les structures de données d'OpenCL et d'OpenGL. Ils

accèdent aux mêmes données à l'aide de différents types de structures. Après le noyau OpenCL traite les données, le processus de rendu OpenGL peut continuer normalement.

Pour configurer l'interopérabilité OpenGL-OpenCL dans le code, trois étapes doivent être effectuées dans l'ordre :

- Créer un contexte d'OpenCL *(cl_context)* qui références le contexte courant d'OpenGL ou groupe partagé,
- Construire les objets de mémoire d'OpenCL (des objets de mémoire tampon et des objets d'image) a partir des éléments de données d'OpenGL (VBOs, objets de texture, et objets de renderbuffer),
- Saisir l'accès exclusif aux données partagées pour le noyau. Après que le noyau exécute, libérer cet accès de sorte que le rendu peut procéder.