HAL
open science

Représentation et algorithmes pour l'exploration interactive de volumes procéduraux étendus et détaillés

Cyril Crassin

To cite this version:

Cyril Crassin. Représentation et algorithmes pour l'exploration interactive de volumes procéduraux étendus et détaillés. Synthèse d'image et réalité virtuelle [cs.GR]. 2007. inria-00598406

HAL Id: inria-00598406

https://inria.hal.science/inria-00598406v1

Submitted on 6 Jun 2011

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

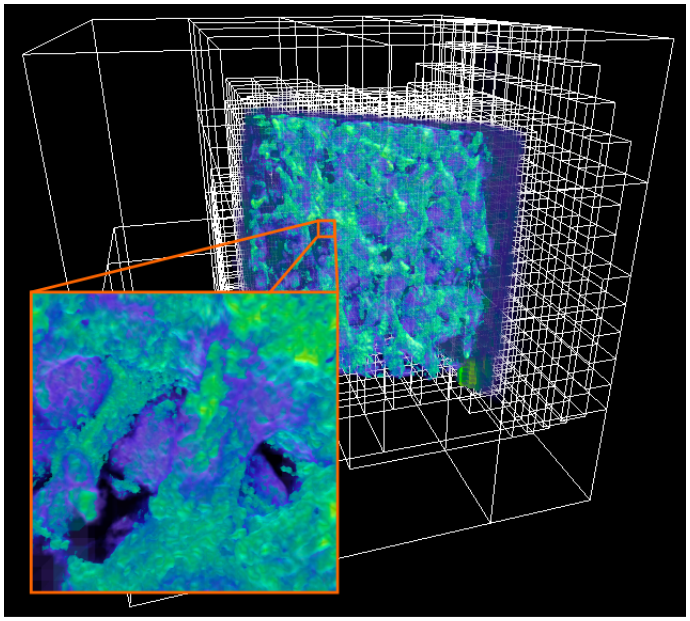
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cyril Crassin Master 2 Recherche - IVR 2007

Encadrant : Fabrice Neyret (Chercheur CNRS, équipe EVASION-INRIA)
Co-Encadrant : Sylvain Lefebvre (Chercheur INRIA, équipe REVES)

Projet de Master :

Représentation et Algorithmes pour l'Exploration Interactive de Volumes Proceduraux Etendus et Détaillés



INP Grenoble
UNIVERSITE JOSEPH FOURIER SCIENCES. TECHNOLOGIE. MEDECINE
Evasion
CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
INRIA RHÔNE-ALPES

Remerciements

Je tiens tout d'abord à remercier mon tuteur, Fabrice Neyret, pour son encadrement, son guidage scientifique, ses conseils toujours pertinents et surtout pour m'avoir fait confiance en me proposant ce stage. Ce fut un plaisir de travailler avec lui et j'ai particulièrement apprécié la liberté qu'il a pu m'accorder. Un grand merci également à Sylvain Lefebvre qui a co-encadré ce stage et qui m'a apporté de très bons conseils et idées scientifiques et techniques. Je remercie ensuite Bruno Raffin d'avoir accepter de faire partie du jury devant lequel je présente ce travail.

Je remercie également toutes les autres personnes présentes dans l'équipe Evasion et avec qui j'ai pu travailler que sont Francois Faure, Franck Hétroy, Lionel Revéret, Georges-Pierre Bonneau, Paul Kry et Eric Bruneton, Antoine Bouthors, Cédric Manzoni, Grégoire Aujay et Michaël Adam.

Un grand salut pour finir à mes compagnons de DEA présents dans l'équipe que sont Adeline Pihuit et Maxime Tournier.

Table des matières

1 Problématique **7**
1.1 Contexte général . . . 7
1.1.1 Scènes naturelles et complexité . . . 7
1.1.2 Scènes complexes et représentation volumique . . . 8
1.1.3 Les scènes volumiques dans l'industrie cinématographique . . . 8
1.2 Accélération par utilisation du matériel graphique . . . 10
1.2.1 Matériel graphique pour le rendu interactif . . . 10
1.2.2 Évolution des cartes graphiques grand public . . . 10
1.2.3 Convergence entre rendu réaliste et rendu interactif . . . 10
1.3 Gestion de gros volumes de données . . . 11
1.3.1 Chargement adaptatif et progressif . . . 11
1.3.2 Création procédurale à la volée . . . 11

2 Contexte et État de l'art **13**
2.1 Contraintes et usages du matériel graphique . . . 14
2.1.1 Description du pipeline graphique . . . 14
2.1.2 Architecture matérielle . . . 15
2.1.3 Les architectures de dernière génération . . . 16
2.2 Rendu de milieux volumiques . . . 18
2.2.1 Les modèles physiques . . . 18
2.2.2 Le modèle émission/absorption . . . 18
2.2.3 Modèles d'illumination globale . . . 21
2.2.4 Méthodes de rendu interactives . . . 23
2.2.5 Méthodes volumiques appliquées au *shading* de surfaces . . . 26
2.3 Représentation des volumes . . . 28
2.3.1 Structures adaptées au lancer de rayons surfacique sur GPU . . . 28

3 Contributions **31**
3.1 Exposé de la problématique visée . . . 31
3.1.1 Présentation . . . 31
3.1.2 Cahier des charges . . . 32
3.2 Études de cas des scènes à traiter . . . 33
3.3 Notre structure de stockage . . . 36
3.3.1 Nécessité de briques régulières . . . 36
3.3.2 Notre structure N^3-tree à briques régulières mip-mappées . . . 37
3.3.3 Gestion du mip-mapping 3D . . . 40
3.3.4 Construction de la structure . . . 41
3.4 Rendu à l'intérieur de la structure . . . 42
3.4.1 Principe général . . . 42
3.4.2 Calcul des points d'entrée et de sortie du volume de donnée . . . 42
3.4.3 Algorithme de parcours de l'arbre . . . 43
3.4.4 Pas d'échantillonnage volumique et arrêt précoce d'un rayon . . . 46
3.4.5 Pré-intégration . . . 46

3.4.6 Performances des deux algorithmes 47
3.5 Chargement dynamique et mise à jour progressive de la structure 48
3.5.1 Gestion des niveaux de détails 48
3.5.2 Le problème de la gestion mémoire 49
3.5.3 Gestion du cache de briques 49
3.5.4 Stratégies pour le temps réel 50
3.5.5 Le modèle de producteur de données 51
3.6 Prise en compte de la visibilité 52
3.6.1 Description de notre méthode d'*occlusion culling* volumique 52
3.6.2 Génération d'une liste ordonnée 54
3.6.3 Stratégie pour le temps réel 56
3.6.4 Extension pour l'exploitation de la structure hiérarchique 56
3.7 Résultats et discussions 57
3.7.1 Performances comparées 57
3.7.2 Influence des paramètres de la structure sur le rendu 58
3.7.3 Discussion 59

Conclusion et travaux futurs **61**

Bibliographie **63**

Chapitre 1

Problématique

Les scènes naturelles sont souvent a la fois très riches en détails et spatialement vastes. Dans ce projet, on s'intéresse notamment a des données volumiques de type nuage, avalanche, écume. L'industrie des effets spéciaux s'appuie sur des solutions logicielles de rendu de gros volumes de voxels, qui ont permis de très beau résultats, mais a très fort coût en temps de calcul et en mémoire. Réciproquement, la puissance des cartes graphiques programmable (GPU) entraine une convergence entre le domaine du temps réel et du rendu réaliste, cependant la mémoire limitée des cartes fait que les données volumiques représentables en temps réel restent faibles (512^3 est un maximum). Proposer des structures et algorithmes adaptées au GPU permettant un réel passage a l'échelle, et ainsi, de traiter interactivement ce qui demande actuellement des heures de calcul, est le défi que ce projet cherche a relever.

1.1 Contexte général

1.1.1 Scènes naturelles et complexité

En synthèse d'images, on représente souvent les objets par un maillage de leur surface (représentation *BRep*). La simulation visuelle de scènes naturelles, et plus particulièrement les phénomènes naturels spectaculaires tels que les avalanches, les nuages, les explosions, les phénomènes météorologiques (cyclones, tempêtes, etc.), représentent un véritable défit en informatique graphique. Ce sont en effet des phénomènes occupant un espace très vaste (pouvant atteindre plusieurs dizaines de kilomètres) et composés de très nombreux détails organisés de manière hiérarchique et particulièrement visibles aux premiers plans de la scène. Ils sont également extrêmement complexes aussi bien au niveau de leurs formes que de leur aspect visuel mais également de leur mouvement.

FIG. 1.1 – Exemples d'images réelles de phénomènes naturel tels que ceux qui nous intéressent.

Simuler de façon réaliste ces phénomènes dans toute leur complexité est ainsi extrêmement difficile. L'industrie du cinéma et des effets spéciaux y parvient à l'heure actuelle avec énormément de travail

manuel et artistique. Réaliser ce genre de simulation en temps réel est actuellement impensable, c'est pourtant l'objectif global dans lequel s'inscrit ce projet.

Les difficultés à surmonter pour atteindre cet objectif couvrent l'ensemble de la chaîne de visualisation. Elles concernent la spécification des données, leur génération, leur stockage, leur animation éventuelle ainsi que leur rendu réaliste et en temps réel. Au sein de cet objectif et à moyen terme, cette étude vise à proposer des structures de données et algorithmes pour permettre la représentation et le rendu en temps interactif de telles données afin de permettre l'exploration d'un vaste volume détaillé.

1.1.2 Scènes complexes et représentation volumique

En informatique graphique, les objets sont généralement représentés par leur surface au moyen de maillages triangulés (on parle de BRep ou *Boundary Representation*). Dans la plupart des cas, il s'agit d'une représentation simple et compacte pour des objets dont on ne s'intéresse généralement qu'à la surface extérieure visible. Mais dans le cas d'objets dont l'occupation spatiale est très complexe, voir fractale, les détails de la surface peuvent excéder la résolution visuelle (*e.g.,* feuillage d'une forêt). Paradoxalement dans ce cas, une représentation volumique, c'est à dire incluant des informations d'occupation de l'espace sur l'intégralité du volume des objets , peut s'avérer plus économique qu'une représentation surfacique. L'intérêt de ce genre de représentation alternative a été étudiée et montrée par Neyret [Ney06].

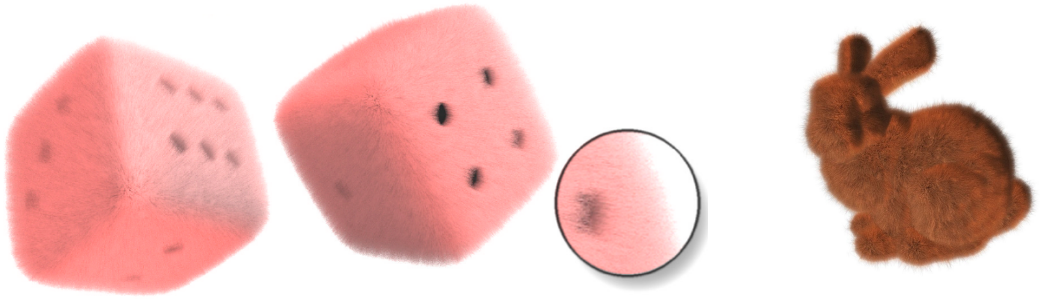


FIG. 1.2 – Exemples de rendus de fourrures réalisés à l'aide d'une technique volumique (source : Jed Legyel, Microsoft Research. *Real-Time Fur over Arbitrary Surfaces*).

Au delà de cette complexité géométrique, certains phénomènes ou objets sont intrinsèquement volumique du point de vue du comportement optique de leurs matériaux. Il s'agit des objets translucides dans lesquels la lumière pénètre puis ressort après avoir interagit avec le matériau à l'intérieur du volume de l'objet. Nous verrons plus en détail les équations physiques et modèles de simulation du rendu section 2.2.

Ces deux aspects de la représentation volumique nous intéressent dans le cadre du rendu de phénomènes naturels. Cette représentation doit en effet permettre à la fois plus de compacité et d'efficacité mais également une meilleur richesse et un meilleur réalisme des effets visuels et donc de l'interaction de la lumière avec les matériaux.

1.1.3 Les scènes volumiques dans l'industrie cinématographique

Le cinéma est de plus en plus demandeur de scènes naturelles spectaculaires de synthèse qu'il est extrêmement coûteux voir impossible de créer réellement pour un tournage. De plus, le réalisateur souhaite pouvoir modeler précisément sa vision d'une scène, ce qui suppose de disposer d'un modèle à la fois réaliste et paramétrable. Pour répondre à cette demande, l'industrie des effets spéciaux a développé des solutions logicielles de rendu de gros volumes de voxels. Mais ces solutions sont extrêmement

lourdes, à la fois en temps de calcul et en occupation mémoire, et elles ne permettent pas une exploration interactive sur des machines grand public.

L'approche de Digital Domain

A titre d'exemple, on peut citer les effets spéciaux de la scène d'avalanche du film *"XXX"*, la tornade du film *"The Day After Tomorrow"* (*"Le jour d'après"*) ou encore la vague géante et son écume se transformant en chevaux du film *"Lord Of The Ring"* (*"Le seigneur des anneaux"*) [Car03, Dun04].

Ces trois scènes ont été réalisé par la société d'effets spéciaux *Digital Domain* [Dom] spécialisée dans le rendu de ce genre de phénomènes. Ils ont pour cela développé leur propre solution de rendu de champs de *voxels* appelé originairement *Voxel-B* puis devenu par la suite *Storm*.

Afin de permettre un rendu réaliste de ce genre de phénomène, il n'est pas possible d'employer des méthodes à base de particules utilisées couramment en rendu temps réel. Des structures volumiques fournissant un accès global aux données doivent être utilisées afin de permettre des calcul d'éclairage complexes prenant en compte de multiples phénomènes lumineux [KH05, Kap02].

1.2 Accélération par utilisation du matériel graphique

1.2.1 Matériel graphique pour le rendu interactif

Depuis longtemps, le champ de la synthèse d'image se sépare entre rendu et simulation haute qualité, très cher en calcul et à destination par exemple de l'industrie des effets spéciaux, et le rendu interactif voire temps réel, à l'usage notamment des jeux vidéos et outils de visualisation ou de spécification. Pour cet usage, tous les compromis doivent être fait en terme de qualité pour garantir une fréquence d'affichage de 10 à 60 images par seconde (on parle de FPS ou *Frames Per Second*) et l'écart en temps de calcul pour une image par rapport à un rendu haute qualité peut être d'un facteur 10^5 à 10^7.

FIG. 1.3 – Stations de rendu graphique SGI.

Depuis plus de 20 ans pour les machines spécialisées et 10 ans pour les machines grand public, le calcul du rendu des scènes de synthèse à l'usage des applications interactives est accéléré par du matériel spécialisé. Il s'agit de la carte graphique et de ses circuits dédiés au rendu, aujourd'hui appelé GPU (*Graphics Processing Unit*, Unité de Traitement Graphique). L'algorithme de rendu de prédilection est alors le Z-Buffer couplé à la *rasterisation* (*i.e.,* méthode projective), tandis que l'algorithme classique pour le rendu haute qualité est la lancer de rayons (simulation inverse du parcours de la lumière depuis l'oeil).

1.2.2 Évolution des cartes graphiques grand public

Mais ces dernières années, le domaine du matériel graphique a connu une série de révolutions successives. Les fonctions matérielles chargé de positionner en 3D les triangles, puis de les projeter à l'écran pour enfin les transformer en *pixels* (opération appelée *rasterisation*) et y appliquer un modèle d'illumination (*shading*), sont devenues progressivement non seulement extrêmement puissantes en terme de puissance brute de calcul (plusieurs centaines de millions de triangles et dizaines de milliards de pixels par seconde), mais également progressivement programmables, et ce de façon de plus en plus générique. Les cartes d'accélération 3D sont maintenant appelés GPU (*Graphics Processing Unit*) et cette programmabilité atteint un tel degré de généricité que ces architectures commencent à être utilisées pour implémenter et accélérer des algorithmes sans aucun rapport avec le graphisme (c'est le domaine du GPGPU, *General Purpose computing on GPU* ou calcul générique sur matériel graphique (*cf.,* section 2.1).

FIG. 1.4 – GPU NVidia GeForce 8800 GTS, toute dernière génération de processeurs graphiques mise sur le marché.

1.2.3 Convergence entre rendu réaliste et rendu interactif

De part cette évolution du matériel, on assiste à une certaine convergence entre effets spéciaux et jeux vidéos, le rendu temps réel étant de plus en plus complexe et réaliste, et les logiciels d'effets spéciaux cherchant à tirer parti de cette puissance de calcul brute. De plus, généricité naissante du matériel ouvre de nouvelles voies dans la palette d'algorithmes qu'il devient possible d'accélérer grâce au matériel. Depuis quelques années déjà, des techniques de rendu volumiques tirant partit du matériel graphique ont été proposés (*cf.,* section 2.2), mais l'efficacité de ces techniques restait largement lié à leur lien avec le modèle de rendu temps réel pour lequel est conçu le matériel.

Les toutes dernières générations de GPU permettent maintenant d'envisager de nouvelles approches du rendu volumique plus proches des techniques utilisées pour le rendu réaliste mais à une vitesse d'affichage temps réel.

1.3 Gestion de gros volumes de données

Les scènes que nous souhaitons traiter sont constituées d'une quantité très importante de données volumiques pouvant atteindre plusieurs milliards de voxels. De telles quantités de données posent déjà des problème de gestion mémoire dans le cadre du rendu haute qualité, ces problèmes deviennent encore plus capitaux pour le rendu temps réel. Des structures de données permettant un stockage compact ainsi qu'un accès rapide et efficace aux données doivent être mises en place. Des stratégies de gestion de la mémoire doivent également être employées afin de permettre la manipulation de ces données qui ne peuvent toutes tenir en mémoire. Dans le cadre du rendu temps réel, les zones mémoires employées (sur le matériel graphique en particulier) sont très proches des unités de calcul et spécifiquement conçues pour un accès rapide et performant. La contrepartie de cette efficacité est qu'elle ont une capacité très limitée et qu'elles sont encore plus éloignées des mémoires de masses dans lesquelles sont stockées ou générées les données. De ce fait, leur approvisionnement en données est particulièrement lent. En plus d'être coûteuses à transférer, ces données sont également coûteuses à générer et cet aspect devra également être pris en compte.

1.3.1 Chargement adaptatif et progressif

Pour permettre l'approvisionnement en données de ces mémoires, des stratégies de chargement adaptatif et progressif doivent être mises en place afin de traiter uniquement les données nécessaires pour un point de vue donnée, tout en fournissant une mise à jour douce et progressive qui n'impacte pas la fluidité de l'animation.

1.3.2 Création procédurale à la volée

Un moyen de contourner le problème du transfert des données consiste à générer ces données à la volée de manière procédurale directement dans les zones de mémoire proches de l'unité en charge de leur rendu à partir de paramètres plus simples et légers.

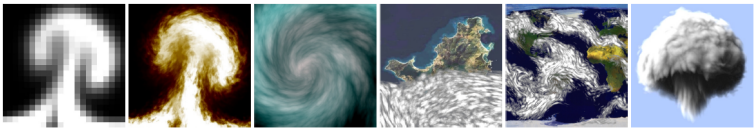


FIG. 1.5 – Illustration de diverses utilisation et extension des Flow Noise pour amplifier des données basse résolution [PN01, Ney03].

Ces méthodes sont généralement employées pour l'ajout de détails très fins sur des données issues d'une simulation grossière. De telles méthodes ont été employées pour l'ajout de bruit sur une simulation de fluide par exemple, comme présenté par [Ney03] et étendu ensuite par [AN06] qui a montré qu'il était possible d'en faire une implémentation directement sur processeur graphique. Ces méthodes sont basées notamment sur le Flow noise [PN01] qui propose un modèle de bruit animé procédural aléatoire continue dérivable fractal basé sur les textures de Perlin [Per85].

Chapitre 2

Contexte et État de l'art

Dans ce chapitre, nous détaillerons les différents domaines nécessaires au projet et que nous avons introduits au chapitre précédent. Nous commencerons tout d'abord par présenter en 2.1 les spécificités des algorithmes et structures adaptées au matériel graphique moderne, en lien avec leurs modèles d'architectures. En 2.2, nous verrons ensuite la physique liée au rendu de milieux volumiques et nous passerons en revue les algorithmes de rendu volumique existants. Les différentes représentations et structures de données existantes pour stocker et permettre le rendu efficace des volumes de *voxels* seront détaillés section 2.3.

2.1 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 2.1.1, puis nous aborderons l'architecture interne de ce matériel en section 2.1.2.

2.1.1 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 [Wik]. 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 [Grob] et Direct 3D [Mic] permettent de contrôler, son schéma de principe général correspondant au dernier modèle exposé par ces API est présenté figure 2.1.

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.

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

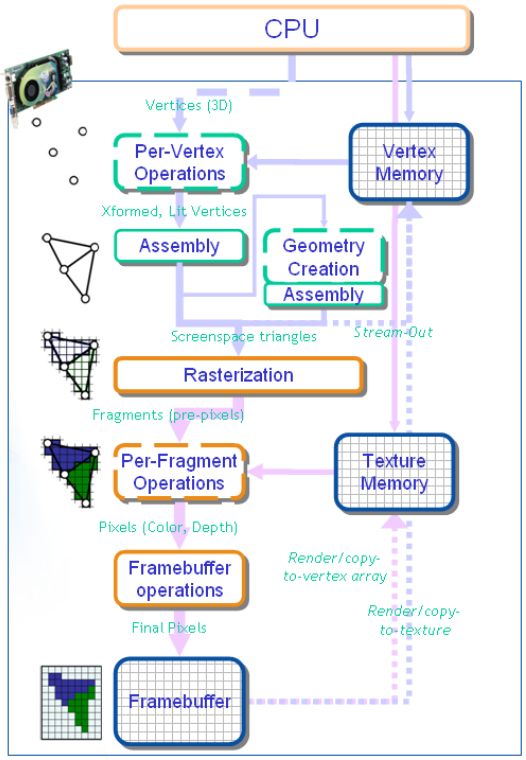


FIG. 2.1 – Schéma de principe du *pipeline* de rendu temps réel.

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 2.1. 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*).

2.1.2 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 2.1.1. 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 plusieures 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 2.1.3 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).

2.1.3 Les architectures de dernière génération

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 oeuvre 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 2.2.

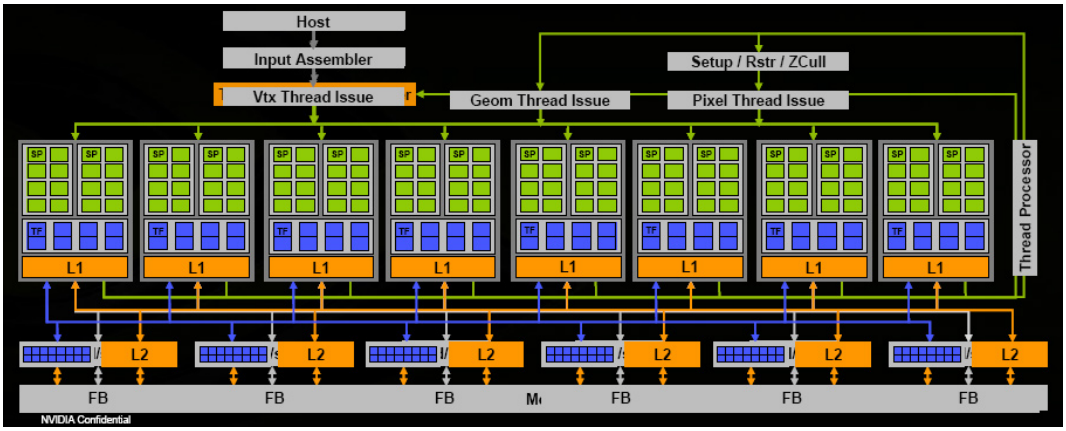


FIG. 2.2 – Diagramme de bloc de l'architecture du G80, dernière génération de GPU du constructeur NVidia constitué de 16 multi-processeurs formés chacun de 8 unités de calcul et regroupés par 2 dans 8 clusters de texture.

Les multi-processeurs

Les unités de calcul appelées *Stream Processors* (SP) sont regroupées par grappes (appelées Multi-Processeurs) 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.

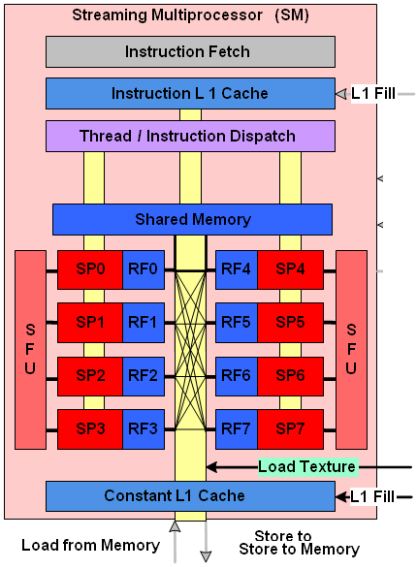


FIG. 2.3 – Schéma de principe d'un multi-processeur du G80. (Source : [Kir])

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 [nVIa].

[1] Modèle de traitement parallèle dans lequel chaque instruction est appliquée en parallèle et de manière identique sur un ensemble de données

2.2 Rendu de milieux volumiques

La visualisation de données volumiques a longtemps été un domaine réservé à la visualisation scientifique ou médicale et l'on trouve ainsi un grand nombre de publications en lien avec ces domaines. Elle permet en effet dans ces domaines de visualiser directement des données produites sous cette forme, par des simulations numériques ou des scanners tridimensionnels. Le *rendu volumique direct* (*Direct Volume Rendering*) regroupe l'ensemble des techniques produisant une image projetée directement à partir du volume de donnée, sans construction intermédiaire telle qu'une iso-surface[2].

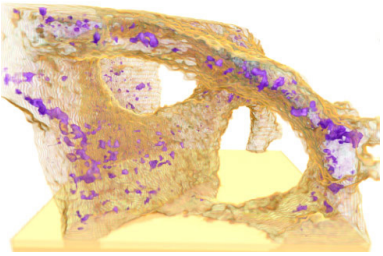


FIG. 2.4 – Exemple de rendu volumique.

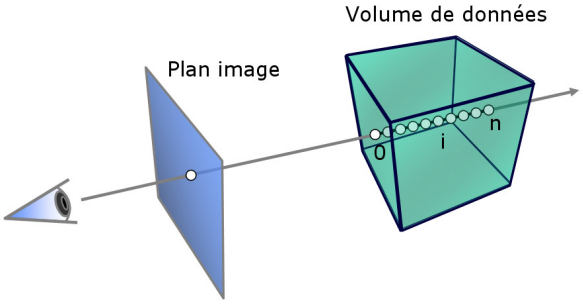
Ces techniques nécessitent la définition d'un modèle optique décrivant la façon dont la lumière interagit avec le volume de données correspondant à un *milieu participatif* (*participating medium*). Pour cela, des propriétés optiques d'émission, d'absorption ou de dispersion (*scattering*) correspondant aux propriétés physiques du matériau sont ainsi associées en tout point du volume.

2.2.1 Les modèles physiques

Selon la nature du matériau à visualiser et le réalisme de la visualisation souhaité, différents modèles optiques intégrant différent niveau de précision dans la prise en compte de l'interaction lumineuse ont été proposés. Le modèle le plus simple consiste à prendre uniquement en compte la contribution lumineuse du milieu le long d'un rayon partant du point de vue et traversant chacun des pixels de l'image (modèle émission/absorption décrit section 2.2.2).

2.2.2 Le modèle émission/absorption

Le modèle le plus simple est le modèle décrit par *Williams et Max* [WM92] (voir également [Max95]) et dans lequel seules les propriétés optiques d'émission et d'absorption du milieu sont prises en compte. Ce modèle revient alors à calculer la contribution lumineuse du milieu le long de rayons partant de chacun des pixels de l'image finale et traversant les données ce qui revient ainsi à une simple projection des données sur l'image. Il est traditionnellement utilisé en visualisation médicale et scientifique. Son principe est illustré figure 2.2.2 et il sera détaillé plus précisément section 2.2.2.



Nous décrirons dans un premier lieu le modèle physique théorique dans un milieu continue puis nous montrerons comment ce modèle peut être discrétisé.

Absorption seule

Le comportement optique le plus simple d'un matériau est celui d'un média parfaitement noir qui ne ferait qu'absorber la lumière le traversant.

[2]Surface polygonale générée à partir d'une iso-valeur dans le volume de données

Dans ce cas, en considérant un rayon donné et un point s_0 émetteur (ou réémetteur) de lumière ayant une caractéristique d'émission isotrope $I(s_0)$ sur ce rayon et dans la direction de l'oeil, l'intensité lumineuse $I(s)$ résultante en un point s du rayon après passage au travers d'une distance $|s - s_0|$ absorbante est donnée par ([MHR02], [WM92]) :

$$I(s) = I(s_0)e^{-\int_{s_0}^{s} \tau(t)\,dt} \tag{2.1}$$

On appel $\tau(t)$ le coefficient d'absorption en un point t du rayon et la quantité $T(s_0, s) = e^{-\int_{s_0}^{s} \tau(t)\,dt}$ est la transparence (ou *Transmittance*) du matériau entre $s0$ et s.

Absorption et émission

En plus de l'absorption, la matériau peut également ajouter de la lumière au rayon par émission propre ou réflection. Si l'on considère maintenant tout point du rayon $\tilde{s}$ situé entre s_0 et s comme émetteur et absorbant de lumière, l'intensité lumineuse résultante au point s est donnée par :

$$I(s) = I(s_0)e^{-\int_{s_0}^{s} \tau(t)\,dt} + \int_{s_0}^{s} g(\tilde{s})e^{-\int_{\tilde{s}}^{s} \tau(t)\,dt}\,d\tilde{s} \tag{2.2}$$

Cette équation s'illustre de manière intuitive par le schéma 2.5 montrant l'évolution de l'intensité lumineuse le long du rayon. La première partie de l'équation représente l'atténuation de la lumière venant de l'arrière plan (avant s_0), la seconde partie est l'intégration de la contribution d'un terme d'émission $g(\tilde{s})$ pour tout point $\tilde{s}$, multiplié par la transparence $T(\tilde{s}, s)$ du matériau entre les points $\tilde{s}$ et s.

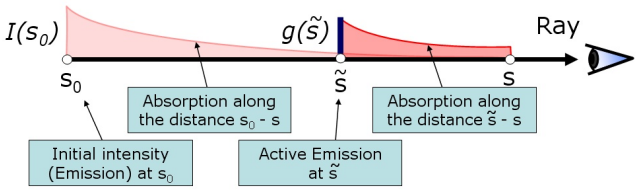


FIG. 2.5 – Illustration du modèle émission/absorption pour tout point du volume émetteur de lumière. Source : [MHR02]

Discrétisation du modèle

Dans le cas général, l'intégrale de l'équation 2.2 ne peut pas être calculée de manière analytique. Pour calculer l'image finale on effectue donc une évaluation numérique de l'équation pour chaque point de l'image afin de déterminer l'intensité résultante et donc la couleur du pixel à la sortie du rayon. Pour cela, on approxime l'intégrale à partir d'un nombre fini de points prélevés dans le volume de données et répartis uniformément le long du rayon (Fig. 2.2.2).

L'approximation numérique la plus simple pour une intégrale de la forme $\int_0^D h(x)dx$ est la somme de Riemann $\sum_{i=1}^{n} h(x_i)\Delta x$. L'intervalle de 0 à D est alors subdivisé en n segments égaux de longueur $\Delta x = D/n$ et un échantillon x_i est choisit pour chacun des segments. La transparence $T(s_0, s) = e^{-\int_{s_0}^{s} \tau(t)\,dt}$ s'exprime alors :

$$T \simeq e^{-\sum_{i=1}^{n} \tau(x_i)\Delta x} = \prod_{i=1}^{n} e^{-\tau(x_i)\Delta x} = \prod_{i=1}^{n} T_i \tag{2.3}$$

La valeur $T_i = e^{-\tau(x_i)\Delta x}$ pouvant être vue comme la transparence du i^{eme} segment sur le rayon. On peut définir de manière identique l'émission $C_i = g(i\Delta x)$ et l'on obtient ainsi l'approximation de l'équation 2.2 (en considérant $I(s_0) = 0$, c'est à dire en prenant en compte uniquement la contribution lumineuse transmise par le matériau) :

$$C \approx \sum_{i=1}^{n} C_i \prod_{j=i+1}^{n} (T_j) \tag{2.4}$$

Dans la pratique, les propriétés optiques d'émission et d'absorption sont généralement représentées par une couleur (RGB) et une opacité (Alpha, A), avec $A_i = (1 - T_i)$. Pour des raison d'efficacité, on évite l'évaluation de l'exponentielle de T_i en simplifiant A_i par $A_i = \tau(x_i)\Delta x$ ce qui reste une approximation relativement bonne lorsque $\tau(x_i)\Delta x \ll 1.0$ [WM92]. Ceci permet ainsi d'évaluer la couleur finale d'un pixel par une composition d'arrière ($i = n$) vers l'avant ($i = 1$) avec l'opérateur OVER (cf. [Pho75]) :

$$C'_i = C_i + (1 - A_i)C'_{i+1} \text{ , avec } C'_{n+1} = 0 \tag{2.5}$$

Pour $n+1$ échantillons, i varie de n à 0 et une nouvelle couleur C'_i est calculée à partir de la couleur C_i et de l'opacité A_i de l'échantillon courant (à la position i) ainsi que la couleur C'_{i+1} calculée à la position précédente $i + 1$. Il faut noter que dans toutes les opérations de composition, les couleurs C_i utilisée sont des couleurs pré-multipliées par leur opacité (*opacity-weighted colors*).

Cette évaluation itérative peut également être effectuée d'avant vers l'arrière, avec i variant de 1 à $n + 1$. Cette fois l'opacité accumulée A'_i doit également être évaluée et stockée :

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \text{ , avec } C'_0 = 0 \tag{2.6}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \text{ , avec } A'_0 = 0 \tag{2.7}$$

Des données aux propriétés optiques

A la base, les données volumiques sont généralement décrites dans des grilles régulières de scalaires (densités, températures etc.). Si l'on souhaite échantillonner entre les sommets d'une grille, la valeur des échantillons doit être interpolée à partir de la valeur des 8 éléments de la grille.

Les propriétés optiques sont associées aux échantillons via l'utilisation d'une *fonction de transfert*. Cette fonction est généralement implémentée sous la forme d'une table de correspondance. Lors d'un rendu volumique de données issues de simulation ou médicales, elle permet de sélectionner les zones des données à visualiser (par exemple en rendant transparentes les densités inférieures à un seuil) ou de mettre en valeur des détails (en colorant ou opacifiant une plage de valeurs). On parle de pré-classification si la fonction de transfert est appliquée aux données avant leur interpolation, de post-classification dans le cas contraire, cette dernière méthode donnant de bien meilleurs résultats.

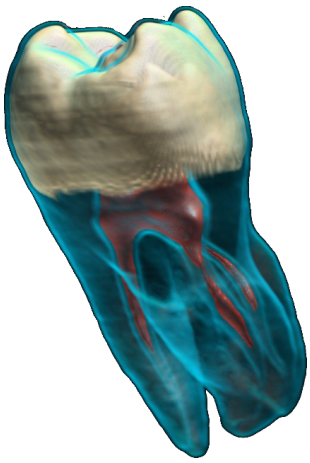


FIG. 2.6 – Illustration de l'application d'une fonction de transfert sur des données médicales. (Source : [Had02])

De nombreux travaux ont été effectués sur ces fonctions, les plus intéressantes concernent l'introduction de fonctions de transfert *pré-intégrées* et leur implémentation sur GPU [EKE01, RE02]. Il s'agit de fonctions 3D pré-calculées et fournissant les propriétés optiques pour des tranches de volumes dans lesquels l'évolution des données est considérée comme linéaire. L'entrée de ces fonction sont la densité d'entrée et de sortie du rayon dans la tranche ainsi que l'épaisseur de celle-ci.

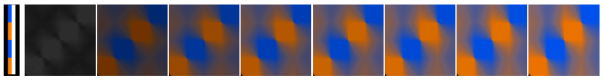


FIG. 2.7 – Exemple de fonction de transfert pré-intégrée. Chaque vignette représente une tranche de la texture 3D (épaisseur), les axes horizontaux et verticaux les densité du point d'entrée et de sortie. (Source [RE02])

2.2.3 Modèles d'illumination globale

Pour permettre un rendu réaliste de la plupart des phénomènes naturels, le modèle émission/absorption doit être enrichi afin de prendre en compte l'ensemble de la lumière pénétrant dans le milieu et son interaction avec la matière.

Réflections simples

Les premiers travaux proposant la prise en compte de la réflexion et de la diffusion dans un *milieu participatif* ont été publiés par Blinn en 1982 [Bli82] pour la simulation de nuages. Ce modèle fonctionne dans le cas de milieux à l'*albedo* faible [3] en ne considérant ainsi qu'un seul rayon secondaire traversant la matière à partir de la particule et en direction de la source lumineuse. Cette approche est appelée couramment *single scattering*. La quantité de lumière réfléchie dans une direction est décrite par une *fonction de phase* fournissant la proportion de l'intensité lumineuse réfléchie en fonction de l'angle d'incidence du rayon sur une particule (la direction). A titre d'illustration, une représentation de la fonction de phase est présentée figure 2.8.

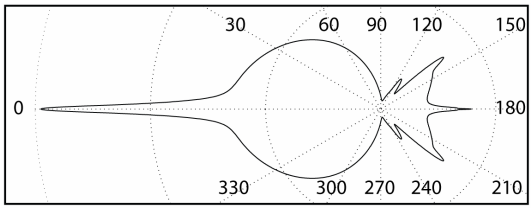


Fig. 2.8 – Illustration de la fonction de phase (Mie) dans un nuage (en log). Le rayon incident vient de la droite et les pics vers 120 et 135 degrés génèrent l'arc en ciel.

Réflections multiples

Alors que le modèle de Blinn est limité à des milieux sous forme de couche, Max a étendu ce modèle à des formes quelconques [Max86] via des fonction analytiques et a introduit la prise en compte de réflexions multiples (*multiple-scattering*). Le premier modèle de rendu permettant la prise en compte du *scattering* a été présenté par Kajiya [KH84]. C'est une méthode en deux passes, la première pré-calculant couche par couche pour chaque voxel du volume la proportion de lumière réfléchie en prenant en compte l'atténuation, la seconde effectuant un rendu de l'image par lancer de rayons sur le modèle émission/absorption à partir des information d'intensité pré-calculées dans le volume. Ce modèle a été étendu par Max [Max94] qui ajoute la prise en compte de la dispersion anisotrope de la lumière à l'intérieur du milieu, ce qui permet d'obtenir par exemple l'effet de corolle illuminée de l'illustration 2.9 lorsque la source de lumière se trouve derrière le volume.

[3] L'albedo correspond à la proportion de lumière reflétée par une particule au sein du milieu par rapport à la lumière absorbée.

FIG. 2.9 – Illustration de l'effet de dispersion et de réflexion multiple à l'intérieur d'un nuage. (Source : Antoine Bouthor [BNL06])

2.2.4 Méthodes de rendu interactives

Il existe 3 grandes familles de méthodes permettant la mise en oeuvre de ce modèle théorique :

- Les méthodes de *splatting* : Celles ci consistent à simuler la projection des voxels individuellement sur l'image via l'utilisation d'une *"empreinte"* approximant leur contribution sur différents pixels (un noyau gaussien est utilise).

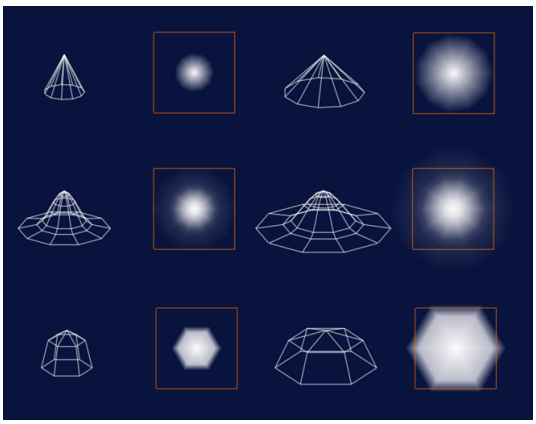


FIG. 2.10 – Illustration du noyau gaussien utiliser pour le *splatting*. (source : [*Lars 1991*])

- Les méthodes de rendu par *slicing* : Celles-ci reposent sur la projection simultanée de tranches à l'intérieur du volume. Des tranches alignées sur les données ont tout d'abord été utilisée comme avec la méthodes Shear Warp : [LL94]. Elles ont ensuite été adaptées afin d'utiliser les capacités du matériel graphique en reposant sur la rasterisation de polygones texturés. Des textures 2D ont d'abord été utilisées avec des tranches alignées sur les données, puis des textures 3D avec des tranches parallèles à l'écran. Cette technique sera présentée dans la section suivante.
- Les méthodes par lancer de rayons : Celles-ci réalisent directement un lancer de rayons au travers des données en espace image.

Textures 3D et *Slicing*

La méthode par slicing et texture 3D est l'une des premières méthodes utilisant fortement les capacité du matériel 3D à avoir été proposée. Le principe de la méthode a été introduit en 1993 par Cullip et Neumann dans [CN94] puis a améliorée par Cabral et al. en 1994 [CCF94] pour exploiter les capacités de texturage 3D du matériel et en particulier la fonctionnalité d'échantillonnage interpolé à l'intérieur des textures 3D (interpolation *tri-linéaire*).

La méthode a ensuite étés revisitée de nombreuses fois dans la littérature, en particulier pour suivre l'évolution du matériel graphique grand public [RSEB+00], [GK96], ou proposer des amélioration dans l'utilisation des fonctions de transfert [EKE01]. Joe Kniss et al. [KPHE02] ont également proposé en 2002 une méthode en deux passes permettant de simuler de manière interactive l'atténuation volumique de la lumière diffusée au sein du volume *(single scattering, cf.*, section 2.2.3).

FIG. 2.11 – Exemple de données médicales visualisées par texture 3D. (Source : Joe Kniss [KPHE02])

Principe de la méthode Le principe de la méthode consiste à échantillonner le volume de données à l'aide d'une série de polygones de support planaires. Ces polygones sont texturés avec une texture 3D (généralisation à un tableau 3D des textures 2D classiques) contenant soit les données qui seront transformées à la volée à l'aide d'une fonction de transfert (post-classification *cf.*, section 2.2.2), soit

directement les propriétés d'émission et d'absorption du matériau (pré-classification). Ces polygones sont ainsi dessinés d'arrière vers l'avant ou d'avant vers l'arrière par rapport au point de vue et combinés selon l'équation 2.5 ou 2.6 via la fonction de composition du matériel (*blending*). Les polygones de support peuvent être alignés selon un axe du volume [RSEB+00] ou parallèles au point de vue [WE98]. Ce principe est illustré figure 2.12 avec des polygones alignés sur le plan de vue.

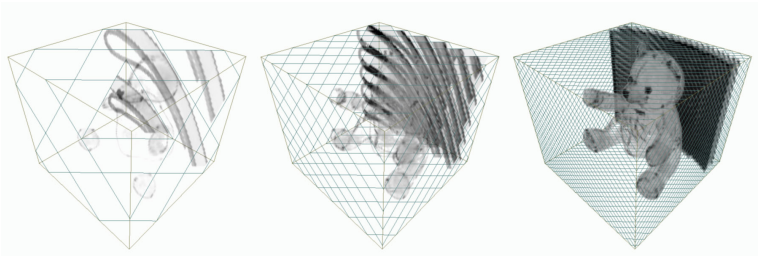


FIG. 2.12 – Visualisation de la série de polygones parallèles au plan de vue support de rendu des données volumiques.

Cette technique exploitant le matériel graphique d'une façon très similaire à la façon dont l'exploitent les jeux vidéos (placage de texture, rasterisation, blending), cela lui a permis de profiter de l'évolution du matériel graphique grand public et en particulier des capacités de *rasterisation* et de traitement par pixel (*Fragment Shaders* et *Blending*, cf. section 2.1). Pour des tailles de données raisonnables (environ 256^3), cette méthode permet ainsi d'obtenir des qualités de rendu acceptables à des fréquences interactives. Elle a de plus l'avantage d'être compatible avec la technique du Z-buffer pour la gestion de la visibilité et permet ainsi une bonne intégration dans des scènes surfaciques.

Mais cette méthode souffre d'inconvénients importants. Un premier est la difficulté voir l'impossibilité d'obtenir un échantillonnage régulier le long des rayons dans le cas d'une projection perspective (cf. figure 2.15). Cela nécessiterait en effet l'utilisation de géométries de support sphériques ce qui s'avère beaucoup trop coûteux. Ce problème provoque alors des artefacts d'échantillonnage relativement acceptables pour une visualisation extérieure du volume, mais beaucoup trop visibles pour une navigation à l'intérieur de celui ci.

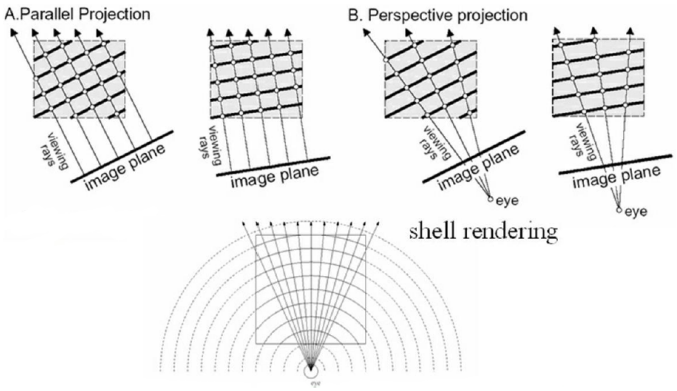


FIG. 2.13 – Illustration du problème d'échantillonnage dans le cas d'une projection perspective : la distance d'échantillonnage n'est pas constante en fonction de l'angle du rayon.

Je penses qu'n autre inconvénient qui devient de plus en plus handicapant sur le matériel graphique est le coût géométrique de cette méthode (coût en primitives graphiques générées, transformées et rasterisées). Sa précision dépend en effet directement du nombre de primitives géométrique rasterisées, alors que le matériel graphique a tendance à fournir de plus en plus de puissance pour les opérations

de traitement par fragments, la rasterisation de ces nombreuses primitives devient l'élément limitant de cette méthode.

Ray-casting volumique

Pour remédier à ces problèmes et grâce à l'évolution du matériel graphique vers plus de programmabilité (*cf.*, section 2.1), de nouvelles méthodes implémentant directement un lancer de rayons sur GPU commencent à être proposées. La méthode consistant à ne lancer qu'un rayon primaire par pixel de l'image on parle dans ce cas de *ray-casting volumique* ou de *ray-marching* (pour souligner le principe d'accumulation le long du rayon), en opposition au *ray-tracing* utilisé plutôt pour du rendu de surfaces et dans lequel des rayons secondaires sont réémis de manière récursive (ombres, reflets).

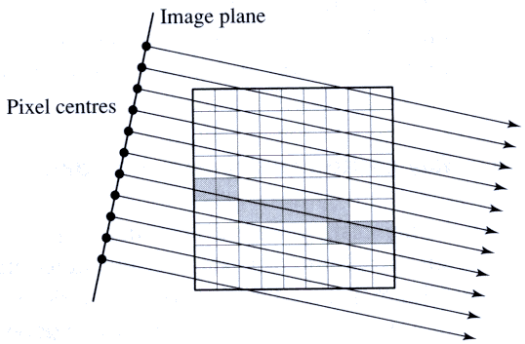


FIG. 2.14 – Illustration du ray-casting volumique dans le cas d'une projection orthographique.

La première approche de ce genre a été proposée par Krüger [KW03] et a été développée pour les GPU de deuxième génération (*Shaders Model 2.0*). Le but principal de la méthode est de permettre l'arrêt du calcul d'un rayon lorsque celui ci a atteint une opacité totale. Pour cela, une fonctionnalité des GPU appelée *test de profondeur précoce* (*Early Depth Test*) est utilisée. Cette fonctionnalité permet au matériel de détecter précocement les pixels qui seront rejetés par le test de profondeur (*depth-test*) et évite ainsi leur traitement coûteux par le *fragment shader*. C'est une méthode multi-passe dans laquelle les données sont stockées dans une texture 3D et qui se décompose de la façon suivante :

- **Passe 1 :** Rendu vers une texture des faces avant de la boite englobante du volume de données et inscription pour chaque fragment de la coordonnées de texture au point d'entrée du rayon dans le volume.
- **Passe 2 :** Rendu vers une autre texture des faces arrière et calcul par fragment (*fragment shader*) du vecteur direction du rayon correspondant via les coordonnées récupérées dans la texture précédente. Le résultat de ces deux premières passes est présenté figure 2.15.
- **Passes 3 à N principales :** N rendus successifs des faces avant du cube. Pour chaque rendu, le *fragment shader* accumule M échantillons (*cf.*, section 2.2.2) de la texture 3D le long du rayon correspondant au fragment, la géométrie de ce dernier étant récupéré via les textures calculées par les deux premières passes. Le résultats de l'accumulation est combiné à chaque passe avec le résultat de la passe précédente et écrit dans une texture.
- **Passes 3 à N secondaires :** Ces passes intermédiaires permettent d'inscrire une profondeur dans le tampon de profondeur pour l'utilisation du rejet précoce de fragment. Une valeur de profondeur forçant le rejet du fragment aux passes suivantes est ainsi inscrite pour chaque fragment ayant atteint l'opacité maximum.

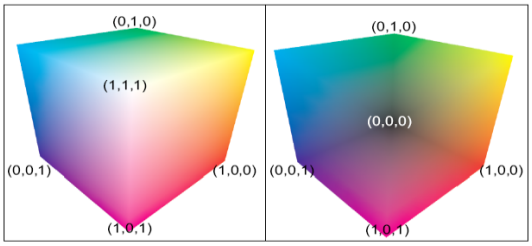


FIG. 2.15 – Résultats des deux premières passes de la méthode de Ray-Casting de Krüger [KW03].

La méthode de Krüger a ensuite été étendue par Scharsach [Sch05] grâce à l'utilisation de nouvelles possibilités offertes par les GPU de troisième génération (*Shaders Model 3.0*). Cette méthode exploite en particulier les fonctionnalités de branchement dynamiques de ces nouvelles architectures pour effectuer l'accumulation le long des rayons dans une passe unique, la boucle itérative étant effectuée à l'intérieur du *fragment shader*. La terminaison précosse des rayons est également obtenue par test dynamique sur l'opacité à l'intérieur du *shader*.

La méthode complète est réalisée en 4 passes et permet entre autre la navigation à l'intérieur du volume de données ainsi que la combinaison correcte avec un rendu surfacique classique. Scharsach introduit également un schéma de découpage en blocs du volume de données permettant un chargement sélectif en mémoire vidéo. Le problème de cette méthode est qu'elle ne tient pas bien le passage à l'échelle (en volume des données), ses performances sont en effet grandement impactées par la quantité de données, du fait de l'utilisation de nombreuses primitives de support devant être rasterisées. Des exemples de rendus obtenus avec cette méthodes sont présentés figure 3.4.

Toutes ces méthodes sont dédiées au rendu de volumes réguliers (grilles de voxels) et ne permettent donc pas une utilisation à grand échelle ou sur des scènes complexes qui nécessiterait une structure de données plus adaptée.

FIG. 2.16 – Rendu de données médicales obtenues par la méthode de Scharsach [Sch05].

2.2.5 Méthodes volumiques appliquées au *shading* de surfaces

En dehors des méthodes dédiées purement au rendu de données volumiques, des méthodes de rendu volumique basées sur un *ray-casting* purement local ont été utilisées pour l'ajout de relief ou de détails à des surfaces sans sur-cout en terme de géométries traitées. Cette idée a notemment inspiré les hypertextures proposées par Perlin [PH89] et permettant la modélisation de phénomènes intermédiaires, entre surface et texture. L'idée est basée sur une synthèse de texture procédurale mais évaluée à l'intérieur d'une région volumique autour des objet plutôt que uniquement sur la surface.

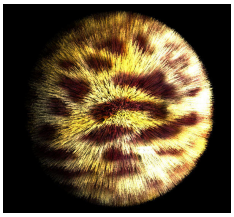


FIG. 2.17 – Exemple d'hypertexture.

Dans la même optique, le *parallax mapping* proposé par [OBM00] puis améliorée par [S.01] permet d'ajouter du relief à une surface en prenant en compte l'effet de parallaxe (effet de déplacement relatif de parties d'un même objet vue en perspective) à faible coût de rendu. Cette méthode a été étendue par [BT04] pour l'utilisation efficace des GPU. Le principe de la méthode

consiste pour chaque pixel de la surface en cours de rendu à calculer l'intersection d'un rayon issue de l'oeil avec une carte de hauteur appliquée sur la surface.

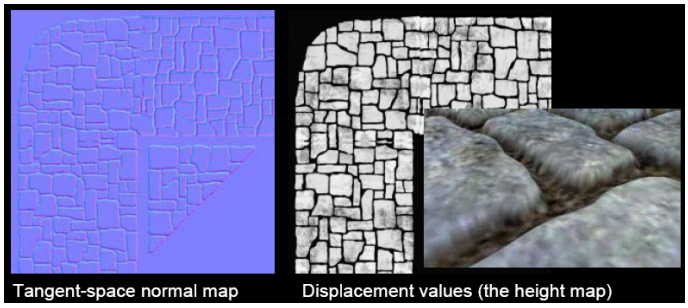


FIG. 2.18 – Illustration des cartes de normales et de déplacement utilisées pour le *parallax mapping*. [BT04]

Une méthode de ray-casting similaire basée sur un champs de hauteur (grille 2D contenant en chaque cellule une élévation) a également été utilisée par [BD06] pour le rendu de volumes liquides en temps réel. Cette méthode permet la prise en compte de réflections, réfractions, et d'effets complexes comme l'absorption lumineuse ou les ombres réfractées et les *caustics*.

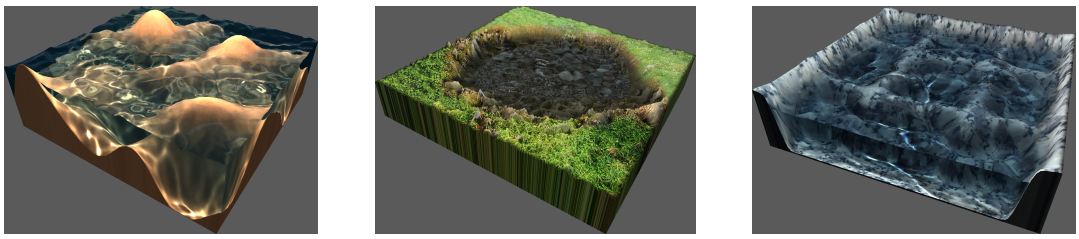


FIG. 2.19 – Rendu de volumes liquides en temps réel [BD06]

Cette idée d'utiliser des textures volumiques pour l'ajout de détails fins ou le rendu de phénomènes ou de matériaux très détaillés a également été largement exploré par Neyret [Ney98, Ney06].

Toutes ces méthodes reposent sur une paramétrisation d'un volume de données généralement 2.5D (2D+hauteur) sur une surface ce qui pose les même problèmes que pour la paramétrisation d'une texture 2D (problèmes de topologies, distorsions etc.). De plus, elles ne permettent qu'une vision locale des données ce qui limite la mise en oeuvre de modèles d'éclairage complexes.

2.3 Représentation des volumes

Il existe différentes structures permettant de stocker des données volumiques. Ces structures sont plus ou moins adaptées aux diverses utilisations de ces données. Dans notre cas, la structure de données doit permettre à la fois un stockage compact et un rendu efficace sur GPU. Le type de rendu visé étant un rendu par lancer de rayons, c'est l'opération de traversée de la structure le long de rayons qui sera l'opération le plus coûteuse, cette traversée doit donc pouvoir être réalisée efficacement. La structure doit également être suffisamment structurante pour permettre un accès rapide aux données tout comme un parcours direct (sans tri), mais elle doit également être suffisamment flexible pour permettre une mise à jour progressive en fonction de l'évolution de données dynamiques. Ce critère nous a fait éliminer d'emblée de notre étude les structures basées sur une hiérarchie de boites englobantes.

2.3.1 Structures adaptées au lancer de rayons surfacique sur GPU

Les caractéristiques des structures de données influant sur les performances de rendu des méthodes de lancer de rayons surfaciques sur GPU sont très proches de celles nécessaires à l'optimalité des structures de rendu volumique par *ray-marching*. La différence réside dans le fait que les rayons ne s'arrêtent pas à la première intersection avec les données mais avancent jusqu'au calcul d'une opacité totale. Les opérations nécessaires à cette avancée sont théoriquement les mêmes que celles nécessaires à la découverte de la première intersection ce qui permet raisonnablement d'extrapoler au *ray-marching* l'optimalité des structures dédiées au *ray-tracing* sur ce genre d'architectures.

Grilles régulières

Les grilles régulières ont été privilégiées depuis les premières expérimentations de lancer de rayons sur GPU ([Pur04, PBMH02]). Elles offrent en effet un accès en temps constant à chaque élément ainsi qu'a leur voisinage ce qui permet un parcours simple le long d'un rayon. Elles fournissent de plus une bonne localité dans l'accès aux données (les éléments accédé successivement sont spatialement proches en mémoire) ce qui est très intéressant pour le parcours sur GPU. Les données accédées successivement ont en effet une forte probabilité d'être déjà présents dans le cache de texture du matériel (optimisé pour la localité spatiale) et donc accédé bien plus rapidement (de l'ordre de 100x !). Une bonne utilisation de ce cache est encore plus critique dans le cas de *Ray-Marching* du fait qu'un rayon accède généralement un grand nombre d'éléments proches avant de s'arrêter. Cette structure est considérée comme la structure optimale sur GPU dans le cas de scènes avec une distribution uniforme des données.

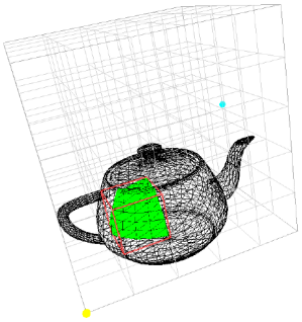


FIG. 2.20 – Scène surfacique dans une grille régulière. (*source : Martin Christen*)

Une méthode intéressante appelée *nuages de proximités* ("*Proximity Clouds*") et permettant d'accélérer la traversée de rayons dans une grille régulière remplie de zones vides a été présentée par Cohen et Sheffer [CS94]. L'idée consiste à pré-calculer dans la grille des informations de distances à l'élément plein de plus proche. Cette méthode est efficace dans le cas de scènes statiques mais trop coûteuse pour des scènes dynamiques.

Kd-Tree

Le Kd-Tree (abréviation de *k-dimentional tree*) est une structure de partitionnement spatial hiérarchique dans laquelle chaque noeud est divisé en deux sous-noeuds par un plan orthogonal à un axe et positionné arbitrairement le long de cet axe à l'intérieur du noeud. Il s'agit en fait d'un cas particulier

du BSP-Tree (dont les plans peuvent être quelconques). Un exemple de Kd-Tree est présenté figure 2.21.

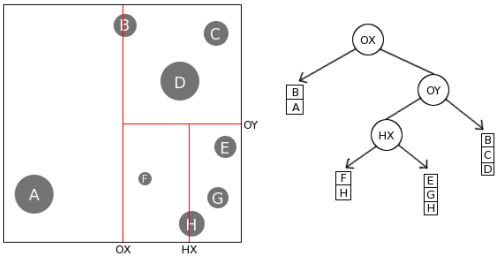


FIG. 2.21 – Exemple de Kd-Tree. (*Source : Eduardo Gastal*)

Havran [Hav00] a comparé un grand nombre de structures d'accélération pour la lancer de rayon (sur CPU) et a montré que cette structure était la plus flexible et fournissait en moyenne les meilleurs résultats pour des scènes quelconques. La première utilisation de cette structure pour le lancer de rayons sur GPU a été proposée par Ernst et al. [EVG04] via l'implémentation d'une pile et d'un algorithme multi-passes lourd ne fournissant pas des performances à la hauteurs des implémentations CPU.

Cette méthode a ensuite été améliorée par Foley et Sugerman [FS05] qui proposent deux méthodes de traversée sans pile de l'arbre qui s'adaptent mieux à l'architecture de traitement de flux des GPU. L'architecture visée par l'algorithme est celles des GPU de seconde génération (*Shaders Model 2.0*) ne permettant pas l'utilisation de branchements dynamiques. Un mode de programmation en *streaming* pur donc utilisé et un noyau de traitement est dédié à chaque étape du parcours, un algorithme multi-passes est employé pour enchaîner les différents noyaux. Les auteurs reportent malgré tout des performances encore largement en dessous de celles atteignables sur CPU (bien que tout de même meilleures que celles de Ernst). Ce manque de performance est expliquée justement par le mode de fonctionnement en streaming pur qui ne permet pas une bonne réutilisation des données calculée lors d'une étape précédents ni un équilibrage efficace de la charge entre les étapes.

La première méthode efficace sur GPU et compétitive avec les performances des meilleures implémentations CPU a été proposée cette année par Horn et al. [HSHH07]. Il s'agit d'une amélioration de l'une des deux méthodes de Foley et Sugerman qui se base sur la troisième génération de GPU (*Shaders Model 3.0*) et leurs capacités de branchements dynamiques (structures conditionnelles et boucles) dans les *fragment shaders* pour fournir un algorithme mono-passe. Cette méthode ne dépasse pourtant pas largement les performances des méthodes CPU, comme le laissait espérer la puissance théorique de l'architecture matérielle. Ce manque de performance s'explique largement par la pénalité liée aux branchements dynamiques sur les architectures de cette génération de GPU. En effet, bien qu'elles proposent des branchements dynamiques, leur mode de fonctionnement interne reste largement SIMD et dédiées au traitement en flux de centaines de fragments[4]. Les parties divergentes (c'est à dire ne suivant pas la même branche d'une structure conditionnelle) du programme exécuté pour plusieurs fragments traités en *parallèle* doivent alors être sérialisés ce qui entraîne une forte pénalité lors de l'utilisation de ces structures de branchements dynamiques dans ce cas.

Cette littérature sur l'utilisation des Kd-tree pour le lancer de rayons surfacique sur GPU est bien représentative de l'efficacité de l'emploi des structures hiérarchiques dans ce cadre. Bien que ces méthodes aient actuellement du mal a outrepasser les meilleures méthodes CPU sur les GPU de troisième génération disponibles jusqu'alors, les nouvelles architectures de quatrième génération (*Shaders Model 4.0*) -et en particulier celle introduite par le G80 de NVidia (*cf.*, section 2.1.3)- laissent supposer un gain de performances très important sur ce genre d'algorithme grâce à une granularité des branchements dynamiques très largement meilleure que sur la génération précédente.

[4]L'efficacité de cette génération d'architectures est en effet fortement liée à un *pipelining* profond des instructions exécutées par les unités de *shaders* qui force ainsi un traitement identique d'un grand nombre de fragments

Octree/N^3-tree

Les octree, et leur généralisation les N^3-tree (N subdivisions de chaque noeuds), ont été peu étudiés dans le cadre du lancer de rayons sur GPU. Cela est lié au fait qu'ils sont considérés comme un cas particulier des Kd-tree (tout octree pouvant être représenté par un kd-Tree) et que la littérature disponible sur le lancer de rayons ne propose pas d'algorithme permettant de profiter réellement de la structure spécifique des octree pour accélérer la traversée d'un rayon.
L'octree offre deux avantages majeurs dans notre cadre de *ray-marching* volumique :

- La subdivision régulière fournit une taille de noeuds identique à chaque niveau. Cette caractéristique est très intéressantes pour le stockage de briques de données régulières (*cf.*, section 3.3.1).
- Un stockage plus compact des données en diminuant le nombre d'indirections et donc de pointeurs nécessaires. Ce nombre réduit d'indirections est également intéressant dans le cadre du rendu sur GPU car cela limite ainsi le nombre d'accès dépendants en mémoire et améliore leur localité ce qui est également une propriété importante comme expliqué section 2.1.
- Une mise à jour progressive en fonction de l'évolution des données dans le cas de scènes dynamiques facilitée par sa structure régulière.

Structures linéaires

Toutes les structures hiérarchiques présentées jusqu'à présents sont généralement implémentées sur CPU à l'aide de pointeurs reliant les noeuds entre eux et indiquant les relations de parentés. Des structures hiérarchiques basées sur des arbres mais implémentées sans pointeurs ont également été proposées. Ces représentations permettent un stockage très compact des données, la contrepartie étant un accès plus complexe et moins direct à ces dernières. On peut citer les octree linéaires et leur utilisation pour le rendu de données volumiques proposée par [CGP04]. Une structure appelée *Geometry Images* a également été utilisée par [CHCH06] pour le rendu de scènes surfaciques par lancer de rayons. Cette structure stock le maillage des objets triangulé uniformément dans une texture (chaque texel stock un sommet) qui est mip-mappée et utilisée comme structure d'accélération pour le lancer de rayons. Une structure de hachage multidimensionelle a également été proposée par [LH06] pour permettre l'encodage compact de données volumiques tout en fournissant un accès aléatoire. Toutes ces méthodes sont très performantes en terme d'occupation mémoire mais leur principal inconvénient est qu'elles sont très peu adaptées au stockage de données dynamiques. La construction de ces structures est en effet très coûteuse et ne permet pas une mise à jour rapide.

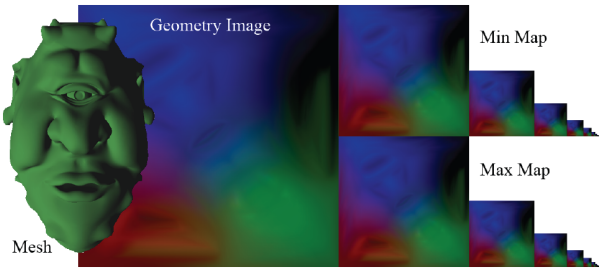


FIG. 2.22 – Illustration de la structure d'images géométriques. [CHCH06]

Chapitre 3

Contributions

3.1 Exposé de la problématique visée

3.1.1 Présentation

Le problème abordé pendant ce stage se décompose en plusieurs aspects. Le premier aspect est le choix d'une **structure de données** permettant un stockage dynamique efficace des scènes volumiques que nous souhaitons traiter, ce choix a été orienté par les propriétés intrinsèques de ces scènes qui seront décrites section 3.2. Pour éviter un gaspillage de la mémoire, cette structure doit permettre à la fois un stockage compact des données et un rendu volumique efficace sur le matériel graphique moderne, les données même compactifiées ne pouvant tenir intégralement en mémoire, cette structure inclura notamment des mécanismes de chargement progressif et dynamique. Bien que dans le contexte de ce stage, le travail se soit limité à des données statiques, la structure doit également être suffisamment flexible pour pouvoir s'adapter facilement à des données évoluant dans le temps (avalanches, nuages en mouvement etc.). Cette structure doit également permettre une mise à jour dynamique rapide en fonction du point de vue et de la résolution nécessaire des données.

Le deuxième aspect abordé concerne l'algorithme de **rendu de données volumiques** à proprement parler, compatible avec cette structure de données et tirant le meilleur parti possible du matériel graphique. De par l'évolution de ce dernier vers des architectures de calcul intensif massivement parallèles et génériques (*cf.*, section 2.1), il devient envisageable de développer des méthodes de rendu volumique basées entièrement sur le lancer de rayons qui permet de gérer individuellement chaque rayon le long de son parcourt et donc de s'adapter à des structures de données hétérogènes. Un certain nombre de travaux ont été menées ces dernières années sur le *ray-tracing* de surfaces sur GPU. Des évolution des méthodes de rendu volumique sur GPU basées sur le *ray-casting* commencent également à apparaître comme présenté section 2.2.4 mais restent dépendantes de la quantité de données visualisées du fait d'un mode de rendu encore très lié au découpage de la scène. Certaines méthodes permettent le traitement de scènes volumiques à haute résolution mais leur efficacité est limitée par la gestion de la structure de données. Elles découplent en effet la structure de stockage hiérarchique des données volumiques en gérant la hiérarchie sur le CPU. Ceci a pour effet de générer un nombre important de communications et d'envois de données pénalisantes entre CPU et GPU. Elles génèrent de plus une latence importante due à l'initialisation et à la traversée complète du pipeline graphique pour chaque rendu d'un bloc volumique (réalisé à partir de primitives surfaciques). Le coût de traitement de la géométrie utilisée par ce rendu peut également être identifié comme source de relentissement sur un matériel graphique de quatrième génération (*cf.*, section 2.1) ou les unités de traitement sont partagées par différentes étapes du pipeline graphique.

A partir de ces constatations, nous avons opté pour un mode de rendu par lancer de rayons totalement réalisé en espace image et en une seule passe dans lequel la structure hiérarchique est totalement prise en compte et utilisée comme structure d'accélération.

Le troisième aspect abordé pendant ce stage concerne la **génération et le chargement dyna-**

mique des données sur le GPU. Deux cas de figures sont possibles : soit les données sont générées ou chargées depuis le disque coté CPU et elles doivent dans ce cas être transférées sur le bus graphique, soit elles sont générées directement sur le GPU et le coût de cette génération peut dans ce cas affecter les performances du rendu lui même (unités de calcul monopolisées). Dans les deux cas, de par la taille limitée de la mémoire embarquée sur le matériel graphique, la faible vitesse de transfert des données avec la mémoire centrale, et la puissance de calcul limitée disponible sur le GPU au regard de la masse de données à traiter, une stratégie efficace de cache des données doit être mise en place. Un modèle de gestion de la mémoire GPU très flexible a été proposé par *Lefebvre et al.* [LDN04] dans le cadre des textures 2D appliquées sur des maillages surfaciques. Nous nous sommes inspiré de ce modèle afin de permettre la gestion des textures 3D stockant les données volumiques.

Pour être efficace en terme d'occupation mémoire, de vitesse de transfert et de calcul, ce système de cache doit également permettre le chargement prioritaire des données réellement visibles, c'est à dire localisées à l'intérieur de la zone de visibilité, mais également non occultées par des blocs opaques plus proches du point de vue et les masquants. Ce problème de *détermination de la visibilité* sur des données volumiques a rarement été abordé dans le littérature du domaine.

3.1.2 Cahier des charges

Le but de ce travail est donc de proposer une structure de données permettant un stockage dynamique et compact des scènes volumiques. Cette structure devrait permettre également un rendu efficace sur le matériel graphique moderne.

Après une étude de cas en section 3.2 visant à préciser les contraintes, nous détaillerons successivement l'ensemble des quatre problématiques listées précédemment. La structure de données proposée sera présentée section 3.3, l'algorithme de rendu section 3.4, la méthode et structure de chargement dynamique section 3.5 et pour finir la détermination de la visibilité en 3.6.

3.2 Études de cas des scènes à traiter

Comme expliqué en introduction, les scènes naturelles visées par ce travail sont principalement de type ciels nuageux ou avalanches, mais on peut également penser à une représentation intégralement volumique de paysages forestiers dans le prolongement de la représentation en textures volumiques proposée par [DN04]. En outre, nous avons également expérimenté des données volumiques réelles (des données médicales d'os *trabéculaire*) aux propriétés similaires. Nous allons dans cette partie montrer les caractéristiques principales de ce genre de scènes et en quoi celles-ci vont influer dans le choix de la structure de donnée utilisée pour leur représentation volumique.

Propriétés intrinsèques des scènes visées

Les scènes qui nous intéressent sont des espaces globalement vides dans lesquels sont plongées les données. Les zones non-vides sont formées de tâches denses et homogènes délimitées par de brèves zones de transitions hétérogènes et translucides. Ces zones sont celles qui nécessiteront une résolution des données importante.

La caméra est plongée à l'intérieur de ces scènes et peut naviguer de manière arbitraire aussi bien au sein des zones vides que des zones homogènes ou translucides.

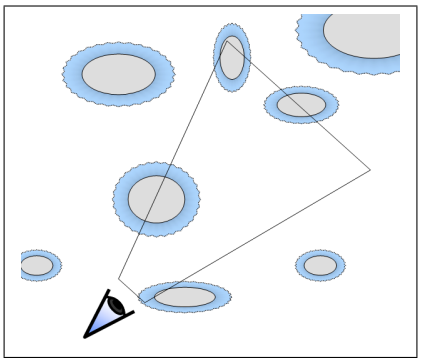


FIG. 3.1 – Structure caractéristique des scènes visées.

Exemple des ciels nuageux

Dans le cas d'une scène de ciel nuageux, l'espace globalement vide du ciel est rempli de nuages formés d'un coeur considéré comme globalement homogène entouré d'une zone de transition translucide contenant les volutes caractéristiques d'un nuage et très détaillée. Cette scène est illustrée figure 3.2.

Pour traiter ce genre de scènes, la structure de donnée doit tout d'abord permettre la compression efficace des grandes zones vides ainsi que leur traversée rapide lors du rendu. Elle doit également permettre un raffinement fin sur les zones de transition et en particulier celles situées à la frontière entre le nuage et le vide. La zone dense à faible variations à l'intérieur du nuage doit être stockée d'une manière compacte et également permettre un arrêt rapide des rayons y pénétrant.

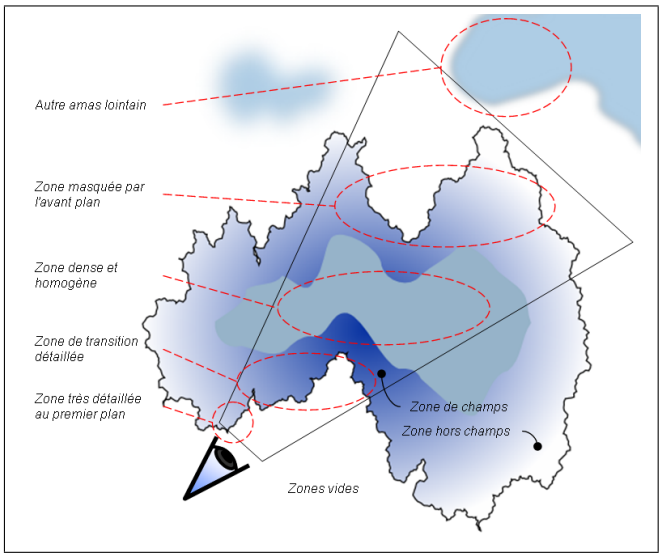


FIG. 3.2 – Illustration des zones remarquables dans le cas de la visualisation d'un nuage dans une scène de ciel nuageux.

Exemple des scènes d'avalanches

Dans le cas d'une scène d'avalanche, la zone translucide et hétérogène est constitué du front de l'avalanche, formé de multiples particules et amas de neige projetés vers l'avant à différentes vitesses et dans différentes directions, ainsi que de la surface de neige agitée sur le dessus de la coulée. La zone dense et constante est constituée d'une partie de la neige ensevelie ainsi que de la roche sous-jacente.

La problématique est ici très similaire à celle des nuages, la principale différence réside dans la répartition des données dans l'espace qui est plus dense avec moins de zones vides et un seul gros phénomènes modélisé de manière identique aux nuages.

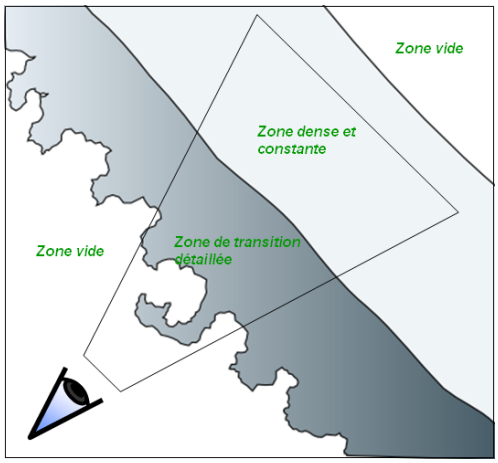


FIG. 3.3 – Illustration de la structure simple retenue pour la modélisation d'une avalanche.

Exemple des scènes forestières

Cette structure se retrouve également dans les paysages forestiers. Les arbres formant une zone de transition semi-transparente et fortement détaillée.

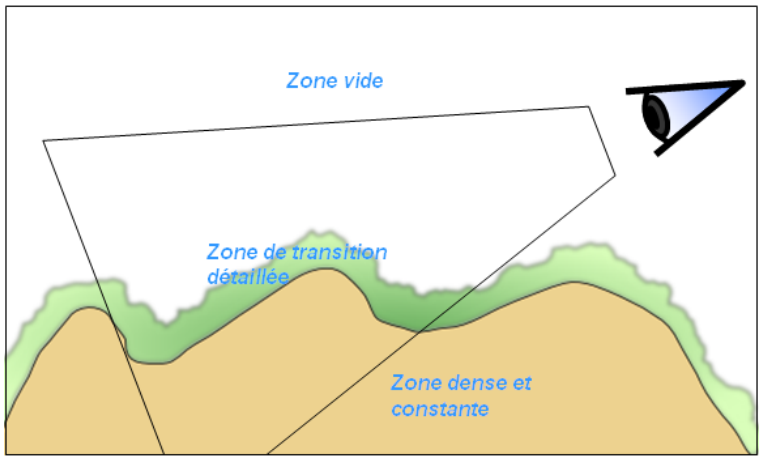


FIG. 3.4 – Structure pour la modélisation d'un paysage forestier.

3.3 Notre structure de stockage

La nature vaste, détaillée et intrinsèquement multi-échelle des scènes qui nous intéressent tout comme le besoin de raffinement adaptatif dans certaines zones localisées en fonction du point de vue et de la fréquence de variation des données élimine d'emblée l'utilisation de grilles régulières comme structure de stockage de scènes volumiques. De plus, dans le cadre du rendu, cette structure ne permet d'accélérer efficacement la traversée de zones creuses pour des scènes dynamiques (ou les structures d'accélération de type *proximity clouds* ne sont pas utilisables). Comme expliqué section 2.3.1, cette structure présente pourtant des caractéristiques très intéressantes pour l'implémentation sur GPU et que nous souhaitons exploiter.

Il ressort de l'étude exposée au paragraphe précédent le besoin principal d'une structure hiérarchique permettant à la fois un stockage efficace des données en "compressant" les zones vides ou denses et constantes présentes dans l'ensemble de ces scènes, mais également un raffinement à la demande des zones proches du point de vue et nécessitant des détails très fins.

Cette structure doit également permettre un rendu volumique efficace basé sur un lancer de rayons en espace image. Elle doit donc pouvoir être parcourue efficacement sur le GPU.

3.3.1 Nécessité de briques régulières

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

- Tout d'abord, le coût de structure en terme d'occupation mémoire doit être raisonable par rapport aux données.
- Ensuite comme expliqué section 2.3.1, le parcours des grilles régulières est bien plus simple et efficace que celui d'une structure plus complexe.
- De plus, ce genre de structure est nécessaire pour pouvoir bénéficier du filtrage tri-linéaire (linéaire sur chaque axe), voir quadrilinéaire (avec mip-mapping) des textures 3D offerte par les GPU (*cf.,* section 2.1.2).
- Pour finir, la génération de blocs de données régulière (ainsi que leur transfert vers le GPU), que ce soit à partir d'une lecture sur disque ou de manière procédurale est toujours plus rapide et efficace qu'une génération indépendante *voxel* par *voxel*.

Un compromis doit donc être trouvé entre les gains apportés par ces avantages et la place mémoire perdue ainsi le nombre de voxels visités inutilement lors du rendu (alors qu'ils se trouvent dans une zone constante ou vide) que cela implique.

3.3.2 Notre structure N^3-tree à briques régulières mip-mappées

Présentation

La structure que nous avons développée se décompose en deux niveaux : une structure hiérarchique en N^3-tree (*cf.*, section 2.3.1) formant une structure d'arbre à N^3 fils et des briques de données régulières stockées dans les feuilles de cette structure. Elle est très proche des *Brick Maps* proposées par Christensen et Batali [CB04] dans le cadre de l'illumination globale. Nous avons généralisé cette structure originalement basée sur un octree pour permettre une subdivision arbitraire des noeuds et autoriser ainsi une traversée efficace sur GPU. Nous avons adapté cette structure à la mise en place sur GPU à partir du modèle proposé par Sylvain Lefebvre [LHN05] pour les textures volumiques hiérarchiques sur GPU.

Cette structure permet de lier les avantages d'une structure hiérarchique en terme de compacité des données à ceux d'une grille régulière lors du parcours sur GPU. Elle inclus également un mécanisme de mip-mapping des briques de voxels présenté section 3.3.3.

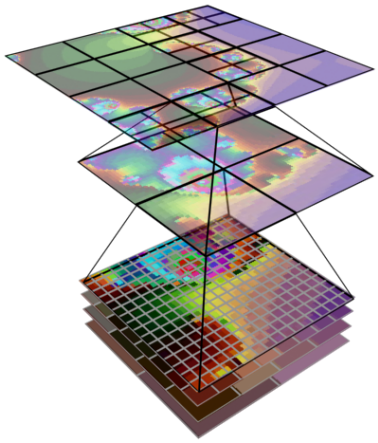


FIG. 3.5 – Illustration 2D de notre structure N^3-tree+briques de voxels.

Structure

Une approche classique pour implémenter un octree ou un N^3-tree sur CPU est d'utiliser une structure à base de pointeurs reliant les noeuds entre eux. Chaque noeud est ainsi constituée d'un tableau de pointeurs pointant vers ses fils, un fils pouvant etre un autre noeud ou une feuille ne contenant qu'une donnée. Nous avons suivi une approche similaire pour notre implémentation sur GPU mais en ajoutant aux pointeurs d'autres informations utiles lors de la traversée de la structure pour le rendu. Chaque élément de sous-noeud d'un noeud contient ainsi :

- Un identifiant de type. Celui ci permet d'identifier le type de sous-noeud : *"Noeud simple"*, *"Feuille"*, *"Brique"* ou *"brique vide"* (dont les données n'ont pas été chargées, *cf.*, section 3.3.3).
- Un pointeur vers le noeud fils.
- Un pointeur vers le noeud parent permettant de remonter la hiérarchie ce qui est utile pour l'une des méthodes de rendu présentée section 3.4. Le fait de stocker ce même pointeur pour chaque fils fournit une meilleur efficacité dans ce même cadre.
- Une valeur moyenne de l'ensemble des voxels présents dans la zone de l'espace couverte par le noeud.

Cette structure est illustrée figure 3.6.

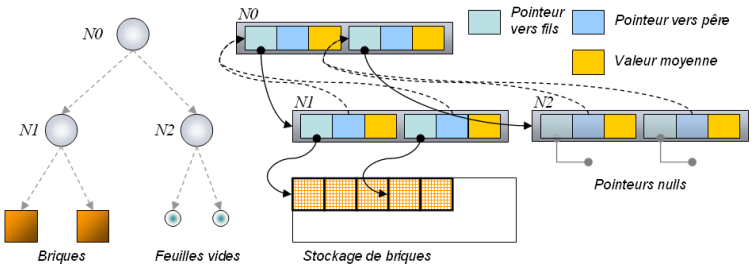


FIG. 3.6 – Illustration conceptuelle de l'implémentation de notre structure.

Implémentation GPU

L'arbre est stocké dans une texture 3D appelée *stockage d'indirection*. Cette texture est au format entier `RGBA` (Rouge, Vert, Bleu, Alpha) avec 32bits par composantes (`GL_RGBA32I_EXT`) disponible sur les GPU de quatrième génération (*cf.*, section 2.1). L'utilisation d'une texture 3D permet d'exploiter

au mieux le mécanisme de cache spatial du matériel. Chaque noeud de l'arbre est stocké comme une petite *grille d'indirection* dans ce *stockage*. Une grille d'indirection est composée de N^3 cellules (voxels) et chaque cellule contient les informations du sous-noeud correspondant décrites dans la section précédente. Ce stockage est illustré figure 3.7.

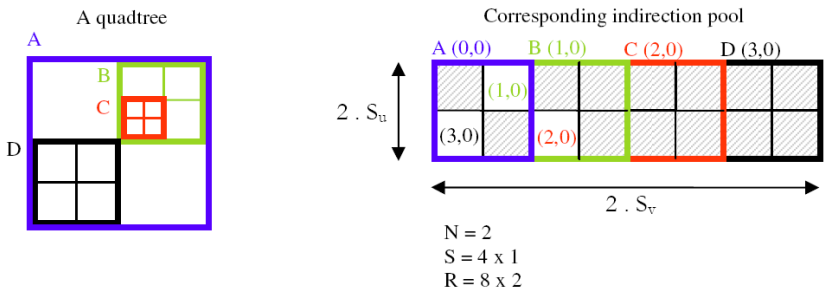


FIG. 3.7 – Illustration dans le cas 2D de l'organisation du stockage d'indirection pour un quadtree. (Source : [LHN05])

Organisation du stockage Pour exploiter au mieux le cache mis en place pour les textures par le matériel 3D, nous avons organisé les grilles dans ce *stockage* afin de favoriser au maximum la localité spatial des données. Lors d'un parcours de l'arbre, la lecture des données dans la texture est en effet plus efficace si les données sont spatialement proches. Nous les avons pour cela stockées en "courbe de Peano" (appelé *Swizzled Walk*) comme illustré figure 3.8 pour un *stockage* 2D. Cet organisation permet de maximiser les chances que des données lues successivement lors du rendu soient spatialement proches.

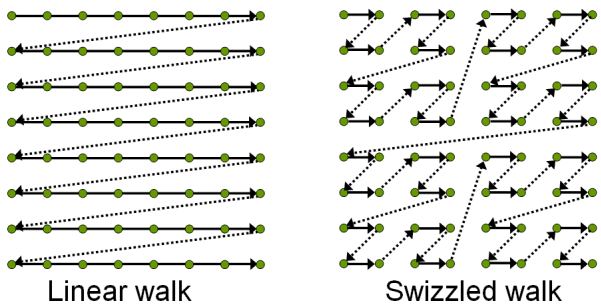


FIG. 3.8 – Illustration du stockage en *"courbe de Peano"* (*Swizzled Walk*).

Encodage des sous-noeuds Les informations de sous-noeuds sont encodées dans les composantes `RGBA` du voxel comme illustré figure 3.9. La composante rouge contient soit l'adresse[1] du sous noeud dans le *stockage* pointé par la cellule, soit l'adresse de la brique régulière contenue dans ce sous noeud dans le cas d'une feuille. Ces informations sont encodées en binaire ce qui permet de les compacter au maximum, cela apporte un gain à la fois en terme de taille de stockage mais également de vitesse de transfert des données de la mémoire vidéo vers les unités de *shaders*. Ces opérations binaires sont possibles grâces aux nouvelles capacité de manipulation de nombre entier des GPU de quatrième génération.

[1] Il s'agit ici d'adresses 3D correspondant aux coordonnées du voxel dans la texture.

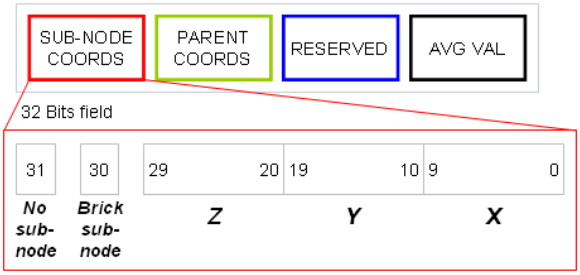


FIG. 3.9 – Encodage des informations de sous-noeud dans un voxel du stockage d'indirection de la structure hiérarchique.

Encodage des adresses La taille naturelle des données en texture peut être de 8, 16 ou 32 bits regroupés en éléments de 1, 2 ou 4 composantes correspondant aux composantes de couleurs qui y sont stockées généralement (RGBA). Dans le cas d'une texture 3D, ces éléments sont adressés par leurs 3 coordonnées. Sur les 32bits disponibles par sous-noeud, 10 bits sont affectés à chacune des coordonnées 3D (*X, Y et Z*), un bit permet d'identifier le type de sous noeud (brique ou noeud de l'arbre) et un bit permet d'identifier un noeud vide et sans descendant. Les 10 bits permettent d'adresser les éléments d'un *stockage d'indirection* ainsi que d'un *stockage de briques* d'une dimension maximum de 1024^3, ce qui parait largement suffisent sachant que cela représente une texture de 16Go de données (RGBA, 4x32bits par élément), déjà 23 fois plus volumineuse que le la plus volumineuse stockable sur les GPU actuels.

L'adresse du noeud parent dans le *stockage* est également stockée dans la composante verte du voxel et encodée de façon similaire. La composante bleu est réservée pour un usage futur et la composante alpha permet de stocker la valeur moyenne de l'ensemble des voxels de données contenus dans la zone couverte par le noeud. Cette valeur est ainsi utilisée lors de l'attente des données lorsque celles ci n'ont pas été chargées (*cf.*, section 3.5), ou quand une brique de voxels est inutile (zone vide ou constante).

Dimensionnement de la structure En ne considérant pas de mise à jour dynamique et dans le pire des cas, celui ou l'ensemble du volume de données doit être stocké à pleine résolution, le nombre de noeuds devant être stockés dans l'arbre est égale à :

$$\sum_{i=0}^{n} \left(\frac{Nb}{N^i}\right)^3 \text{, avec } n = \frac{\log(Nb)}{\log N} \text{ et } Nb = \frac{Rt}{Rb}$$

Rt étant la résolution maximale totale sur chaque axe d'une grille virtuelle contenant l'ensemble de la scène (volume de données) et Rb la résolution maximale d'une brique et N la résolution de chaque noeud ($N = 2$ pour un octree). Ce calcul suppose bien entendu des dimensions compatibles (multiples).

A titre d'exemple pour une résolution totale $Rt = 1024^3$, une résolution de noeud $N = 4$ et une résolution de briques $Nb = 16$, un stockage d'indirection de 64^3 est quasiment suffisant. Sachant qu'en conditions réelles l'ensemble du volume n'est jamais plein et l'ensemble de l'arbre n'est jamais chargé, on peut considérer qu'un stockage de 128^3 convient pour la grande majorité des scènes.

Le cache de briques

Les briques de données régulières sont stockées elles aussi dans une texture 3D. Il s'agit généralement de données scalaires de densité mais d'autres informations peuvent y être ajouté comme des informations de gradients par exemple. Cette texture est filtrée linéairement sur les trois axes par le matériel, et de ce fait accédé dans le shader par des coordonnées flottantes (les coordonnées entières stockées dans le *stockage d'indirection* sont transformées à la volée).

Le stockage est divisé en zones regroupant les briques de résolutions identiques. Chaque brique est constituée de ses données ainsi que d'une bordure d'un voxel l'entourant et permettant le filtrage

trilinéaire sans artefacts (sans cela les échantillons pris sur la bordure de la brique seraient interpolés avec des voxels de la brique adjacente dans le stockage).

Les dimensions et le découpage de ce stockage sont optimisées de façon à limiter le gaspillage de place lié aux différences de dimensions des briques. La séparation fixe du stockage en fonction de la résolution permet une gestion plus aisée des briques mais n'est pas contrainte par la structure. Une organisation plus efficace des données à l'intérieur de ce stockage pourrait être imaginée.

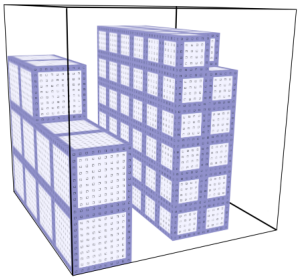


FIG. 3.10 – Illustration du stockage stockant les briques régulières de voxels.

3.3.3 Gestion du mip-mapping 3D

La structure de base présentée précédemment a été étendue afin de permettre l'implémentation d'un mécanisme de mip-mapping volumique. Ce mécanisme permet de palier au problème d'*aliasing* intervenant lors du sous échantillonnage d'une brique qui intervient lorsque l'on échantillonne des données de grande résolution par rapport à la résolution d'affichage.

Pour mettre ce mécanisme en oeuvre, une pyramide de mip-mapping est stockée pour chaque brique. Il s'agit d'une pile de briques organisées en niveaux dont la résolution de chacune est égale à la moitié de la résolution de la brique du niveau inférieur.

Pour l'implémentation de cette pyramide dans notre structure, deux possibilité se sont offertes à nous : Utiliser le mécanisme de mip-mapping fournit par le GPU pour les textures 3D ou ré-implémenter ce mécanisme.

Pour la réutilisation du mécanisme, un problème se pose avec la gestion des bordures stockées autour de nos briques de voxels pour permettre l'interpolation matérielle. Cette bordure doit en effet être dupliquée 2^n fois pour la brique de résolution maximale d'une pile de n niveaux, et divisée par deux à chaque sous-niveau pour permettre un fonctionnement correct. Cette implémentation devient alors rapidement très coûteuse en espace mémoire.

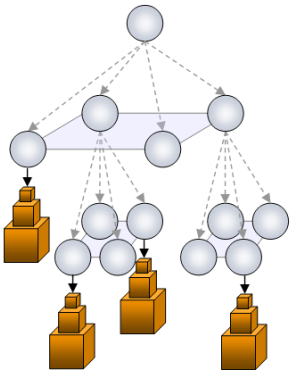


FIG. 3.11 – Illustration du modèle global de notre structure pour un quadtree (octree 2D), les cubes oranges représentent les pyramides de briques mip-mappées.

Pour remédier à ce problème, nous proposons de réimplémenter nous même ce mécanisme de mip-mapping. Bien que cela puisse paraître coûteux, nous nous somme aperçu après avoir effectué des tests que sur notre implémentation, le coût lors de l'utilisation du mip-mapping matériel se révèle être identique à celui de notre implémentation manuelle.

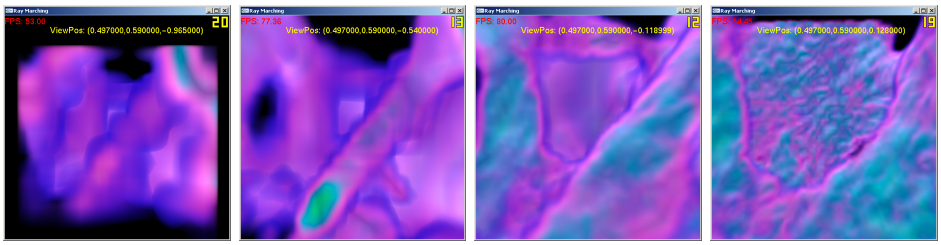


FIG. 3.12 – Illustration de l'effet amplifié du mip-mapping (flou) sur notre rendu.

Le stockage d'indirection de briques

Ce stockage permet de faire le lien entre la structure hiérarchique et les briques de données. Il est constitué d'une série d'entrées stockant pour chaque brique la liste des adresses correspondant à leurs données pour chaque niveau de mip-mapping dans le stockage de briques. Une valeur négative est affectée aux entrées dont les données ne sont pas chargées. Ce mécanisme permet ainsi de maintenir chargées plusieurs résolutions d'une même brique ce qui est utile pour le mécanisme de mip-mapping exposé section 3.3.3, mais également pour l'optimisation du chargement dynamique (*cf.*, section 3.5).

3.3.4 Construction de la structure

Degrés de libertés

Cette structure possède deux degrés de libertés : la résolution de subdivision des noeuds (N) et la résolution des briques de données régulières influant sur la profondeur des briques dans l'arbre (pour une résolution d'affichage donnée, la taille des voxels nécessaires à une certaine distance est fixée). Ces deux paramètres doivent être déterminés principalement en fonction de l'impact qu'ils ont sur les performances du parcours de la structure lors du rendu. La contrainte d'efficacité de la mise à jour de la structure doit également être prise en compte mais reste plus secondaire.

Algorithme de construction

Dans l'état actuel, la structure est construite à partir de données non fractales possédant une résolution maximale. Cette condition devra être levée dans des travaux futurs afin de permettre le rendu de scènes à détails infinis.Nous avons réalisé la construction de la structure hiérarchique à l'aide d'un algorithme récursif mis en oeuvre sur CPU. Le principe consiste à effectuer une descente de l'arbre virtuel en cours de construction jusqu'au niveau des briques de voxels. A ce niveau, les données correspondant aux briques sont lues et analysées. En fonction d'un critère de régularité des données, soit un noeud de type brique est créé mais sans en charger les données qui le seront lors de la navigation (*cf.*, section 3.5), soit c'est un noeud de type feuille sans données. Dans les deux cas la valeur moyenne du bloc est calculée et stockée dans le noeud. Ces noeuds sont en fait des noeuds provisoires qui ne seront créés que si au moins un des noeuds ayant le même père contient une brique. Au niveau récursif supérieur (fonction en charge du noeud ayant ces briques comme fils), un critère de subdivision des fils est donc testé, si aucun des fils n'a été créé, le noeud n'est pas créé, si au contraire au moins un fils a été créé, le noeud est créé avec l'ensemble de ses sous noeuds (grille d'indirection).

3.4 Rendu à l'intérieur de la structure

Il s'agit maintenant de proposer un algorithme de rendu compatible avec la structure de données introduite à la section précédente et aussi perforant que possible. Comme mentionné précédemment, nous nous basons sur une méthode de lancer de rayons, parcourant pur chaque pixel de l'image finale les données le long du rayon depuis l'oeil jusqu'a extinction (opacité totale atteinte).

Dans ce travail de DEA, nous nous limitons à un type de simulation de l'illumination volumique simple : un ray-casting volumique (ne prenants en compte ni inter-réflexions ni ombres) que nous avons réalisé à l'aide d'une méthode en une seule passe, inspirée des méthodes de lancer de rayons surfaciques sur GPU et étendant la méthode de ray-casting proposée par Scharsach ([Sch05]). La généricité et les propriétés que nous avons donné à la structure de donnée et à ce mode de visualisation permettra par la suite une prise en compte de modèles d'illumination plus élaborés.

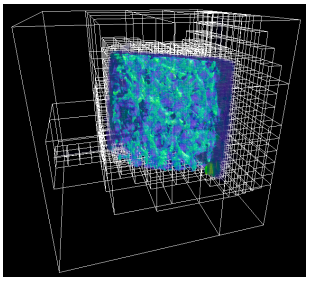


FIG. 3.13 – Rendu issu de notre implémentation avec les boites englobantes de la structure hiérarchique.

L'ensemble de la méthode a été implémentées en OpenGL en utilisant l'interface des shaders et c'est la raison pour laquelle les détails techniques sur la méthode sont données en utilisant ce point de vue. L'utilisation de l'interface CUDA a été envisagée mais n'a pas été utilisée pour cause de manque de maturité de l'API, les fonction d'interfacage avec l'API graphique ne sont en effet pas encore toutes fonctionnelles ni efficaces.

3.4.1 Principe général

La méthode proposée consiste à effectuer l'ensemble des calculs de traversée de la structure de stockage et d'accumulation de la couleur de chaque rayon (*cf.*, section 2.2) dans un programme de traitement de fragments (programme exécuté par le GPU pour chaque fragment d'une primitive graphique), chaque *fragment shader* réalisant le calcul complet pour un rayon lancé depuis le point de vue. Les fragments sont générés à l'aide d'un quadrilatère couvrant l'ensemble de la surface d'affichage et permettant d'initier le calcul. Pour chaque sommet du quadrilatère, un programme de traitement de sommets calcul la position et direction dans l'espace des données du rayon émis depuis le sommet. Ces rayons sont ensuite interpolés par le rasterizer et transmis au programme de traitement de fragments.

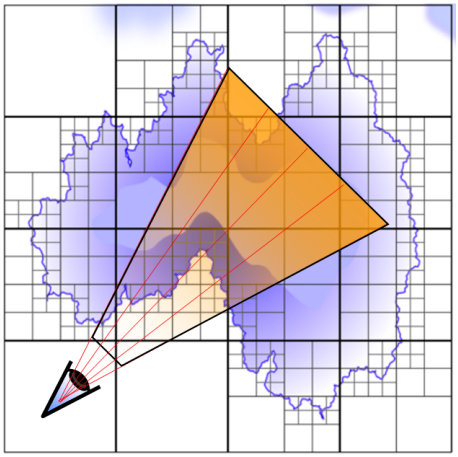


FIG. 3.14 – Illustration du lancer de rayons à l'intérieur de la structure hiérarchique.

3.4.2 Calcul des points d'entrée et de sortie du volume de donnée

La première étape pour chaque rayon consiste à gérer le cas ou le rayon est émis en dehors de la zone de définition des données, ou le cas ou la longueur maximale de ce dernier le fait sortir de cette zone. Il

faut alors calculer le point d'entrée et de sortie du volume à partir du point d'émission et de la direction du rayon pré-calculés. Ceci permet ainsi de traiter aussi bien les cas de point de vue à l'intérieur du volume qu'à l'extérieur. Les rayons n'intersectant pas le volume de données sont également détectés et l'exécution du *shader* est dans ce cas arrêté précocement. L'efficacité des branchements conditionnels sur les architectures des GPU de nouvelle génération (*cf.*, section 2.1) fait que ce genre de sortie prématurée du shader est relativement efficace, ce qui élimine la nécessité d'utiliser le mécanisme de test précoce de profondeur (*early depth test*)comme cela est proposé par [KW03].

3.4.3 Algorithme de parcours de l'arbre

Une fois le rayon initialisé, une succession d'échantillons doivent être pris dans la scène et accumulés selon la formule exposée section 2.2.2. Pour cela, un parcours de l'arbre permettant d'obtenir les noeuds traversés par le rayon, de son point de départ à son point d'arrivée, doit être effectué. Ce parcours alterne deux modalités :

- Le parcours *"horizontal"* parmi les cellules d'un bloc de N^3 sous noeuds (les grilles de sous noeuds décrites section 3.3.2).
- Le parcours *"vertical"* au travers des niveaux hiérarchique de l'arbre.

Les algorithmes CPU utilisent une pile afin d'optimiser le parcours en profondeur de l'arbre. Comme expliqué en 2.1, le modèle de *shaders* proposé par les API graphiques ne fournissait pas jusqu'à leur version 3.0 de zones mémoires temporaires indexables (typiquement des tableaux) accessibles par un shader pour des opérations de lecture et d'écriture.

Cette limitation a obligé jusqu'à aujourd'hui les implémentations à développer des méthodes sans piles moins performantes (*cf.*, section 2.3.1).

Nous avons implémenté et testé deux algorithmes de parcours, l'un utilisant une mini-pile grâce aux nouvelles capacités matérielles des GPU de quatrième génération, l'autre sans pile basé sur une série de descentes et remontées dans la hiérarchie en suivant les liens stockés dans la structure.

Dans le cas particulier des octree ($N = 2$), nous avons également testé d'autres algorithmes comme la méthode de traversée proposée par [FP02] et basée sur l'utilisation de codes de localisations binaires, ou la méthode de redescente proposée par [FS05] pour les Kd-tree et adaptée aux octree à l'aide le l'algorithme de [LHN05] pour la localisation de points dans une texture octree. Mais ces implémentations se sont montrée relativement lentes et pas assez flexible pour permettre la généralisation du parcours à une structure à N quelconque.

Parcours avec mini-pile

Le parcours avec mini-pile que nous avons conçu utilise une nouvelle fonctionnalité fournie par les *shaders models 4.0* (*cf.*, section 2.1 qui est la possibilité d'indexer dynamiquement des tableaux locaux aux *shaders*. L'algorithme développé est une extension aux N^3-tree de la méthode DDA de parcours de grilles régulières proposée par [AW87]. Le DDA est une généralisation de l'algorithme de *Bresenham* pour le tracé de lignes fournissant l'ensemble des cellules coupées par une ligne et utilisable en 3D. Cet algorithme présente l'avantage de permettre un parcours ordonné des cellules traversée par le rayon extrêmement rapide, avec un minimum de branchements conditionnels et sans recours à des tables de voisinage comme c'est le cas pour d'autres algorithmes.

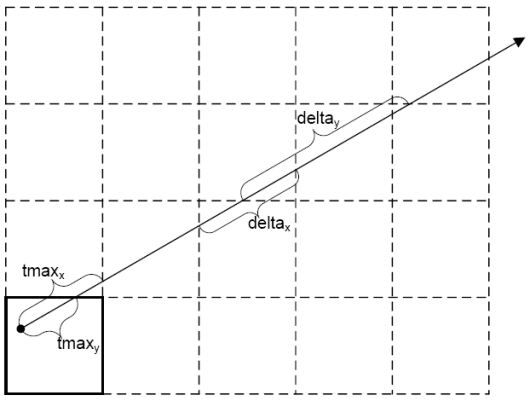


FIG. 3.15 – Illustration de l'algorithme DDA. (Source : *Niels Thrane et Lars Ole Simonsen*)

Structure de pile La pile est implémentée à l'aide d'un tableau dont la taille doit être fixée à la compilation et correspond à la profondeur maximale dans la structure hiérarchique. Le pointeur de pile est un entier stockant la position courante dans le tableau et incrémenté ou décrémenté à chaque insertion et suppression de valeurs.

Le tableau contient un certain nombre d'informations permettant la reprise de l'algorithme DDA lors de la remontée dans la hiérarchie. Pour une efficacité maximale, un équilibre doit être trouvé entre la quantité d'information stockées dans la pile et celles recalculées lors des remontées en fonction du coût de ces calcul et de la pénalité en terme d'utilisation des processeurs de flux induite par l'utilisation de mémoire et de registres supplémentaires.

Algorithme L'algorithme se décompose en deux phases : la descente jusqu'au point d'entrée du rayon dans le volume puis l'avancée le long du rayon. La descente initialise un algorithme DDA pour chaque grille de sous-noeuds traversée et stocke ses paramètre ainsi que l'adresse du noeud dans le pool d'indirection dans une mini-pile. Une fois la feuille contenant le point d'entrée atteinte, celle ci est traitée comme expliqué au paragraphe suivant puis l'algorithme DDA avance dans la grille en parcourant chaque sous noeud traversé par le rayon afin de les traiter. Une fois la limite de la grille atteinte, l'algorithme remonte dans la pile, reprend le DDA stocké et continue de façon similaire sur le modèle descentes/DDA/remontée jusqu'à atteindre la limite extérieur du dernier noeud parcouru. Cet algorithme est illustré figure 3.16.

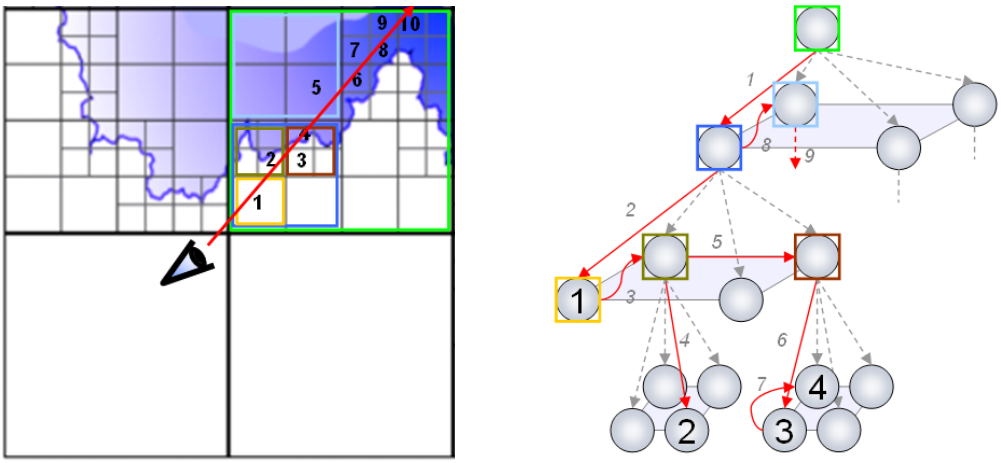


FIG. 3.16 – Illustration sur un quadtree de notre algorithme de parcours. Sur le schéma de droite, l'ordre de parcours des feuilles est numéroté dans celles ci. Les flèches rouges entre noeuds de différents niveaux représentent les opérations de descente, celles entre deux noeuds du même niveau un pas du DDA dans la grille de ce noeud.

L'algorithme global et simplifié est le suivant :

Algorithm 1 Global Mini-Stack traversal algorithm

```
LEVEL ⇐ 0
Node ⇐ GetEntryNode(LEVEL)
while LEVEL >= 0 do
  if NodeType(Node) = INTERNAL then
    LEVEL++
    Stack[LEVEL]=GoDown(Node)
    Node ⇐ GetEntryNode(LEVEL)
  else
    if NodeType(Node) = BRICK then
      RayMarchBrick(Node)
    end if
    Stack[LEVEL].curVoxel ⇐ DDA(Stack[LEVEL])
    while IsOutOfGrid(LEVEL, Stack[LEVEL].curVoxel) do
      LEVEL–
      Stack[LEVEL].curVoxel ⇐ DDA(Stack[LEVEL])
    end while
    Node ⇐ GetNode(Stack[LEVEL])
  end if
end while
```

Traitement des noeuds Les feuilles de la structure sont traités de façon différente en fonction de leur type :

- Si le noeud est une brique dont les données sont disponibles, un *ray-marching* régulier est effectué à l'intérieur de ses données (*cf.*, section 2.2.2).
- Si il s'agit d'une brique non chargée (en cas d'échec du chargement dynamique, *e.g.*, du à une surcharge soudaine du chargement dynamique présenté section 3.5), la couleur moyenne stockée dans le noeud est utilisée et accumulée en fonction de la distance traversée par le rayon dans cette brique (*cf.*, pré-intégration section 3.4.5).
- Si il s'agit d'un noeud vide, le noeud est simplement ignoré.

Parcours sans pile

Nous avons également développé un autre algorithme totalement dépourvu de pile. Celui ci se base sur le même principe que l'algorithme avec pile mais re-calcule l'ensemble des paramètres du DDA à chaque descente et remontée dans l'arbre. La remontée est effectuée grâce à l'information de parenté de chaque noeud stockée dans la structure hiérarchique. Ce lien pointant directement vers le sous noeud exact du noeud parent, l'information de cellule courante dans la grille du noeud parent peut en être extrait et ceci évite ainsi son re-calcul.

Nous avons conçu l'ensemble de l'algorithme de manière à limiter la taille des blocs conditionnels (corps des structures conditionnelles *"if...then...else..."*) ainsi que le nombre de registres utilisés. La limite dans la taille des blocs conditionnels permet de limiter le nombre de *threads* divergents exécutés par les multi-processeurs du GPU (*cf.*, section 2.1) et améliorer ainsi l'efficacité de l'algorithme sur ces architectures. La limite dans le nombre de registres permet aux multi-processeurs de traiter en parallèle un nombre plus important de *threads* ce qui permet de recouvrir plus efficacement les accès aux textures et augmente ainsi également les performances.

3.4.4 Pas d'échantillonnage volumique et arrêt précoce d'un rayon

Afin d'optimiser le rendu des données éloignés en diminuant leur fréquence d'échantillonnage (les données éloignées nécessitant moins de précision que les données proches), nous avons adapté le pas d'échantillonnage des briques en fonction de la distance de l'échantillon au point de vue. Ce pas adaptatif entraîne une inconsistance de l'opacité obtenue lors de l'intégration de la luminosité du volume si la fonction de transfert utilisée ne prend en compte qu'un pas constant. Pour remédier à ce problème, nous avons utilisé une fonction de transfert pré-intégré présentée dans la section suivante.

Lorsqu'un rayon a atteint une opacité totale, il n'est plus nécessaire de poursuivre l'évaluation du rayon. Nous avons donc mis en place un mécanisme permettant d'interrompre précocement le *shader* lorsque ce critère est atteint. Les tests dynamiques ayant un impact sur les performances, celui ci ne doit pas être effectué après chaque échantillon. Le test que nous avons mis en place est affecté après l'accumulation de chaque brique et une sortie prématurée du *shader* est effectuée en fonction du résultat de ce test. Nous avons effectué des tests afin d'observer l'efficacité de cette sortie conditionnelle et il s'est avéré qu'elle fournit effectivement de bonnes performances sur G80.

3.4.5 Pré-intégration

Lors de l'accumulation des propriétés optiques, une fonction de transfert pré-intégrée 3D est utilisée (*cf.*, section 2.2.2). Celle-ci permet en effet de traiter simplement les différences de fréquences d'échantillonnage entre chaque brique tout comme l'accumulation en un pas unique des briques constantes (utilisation de la valeur moyenne stockée directement dans la brique). Ceci permet ainsi de conserver une luminosité constante et elle fournit de plus une meilleur qualité de rendu.

Cette fonction est stockées dans une texture 3D et accédée par des coordonnées flottantes (valeurs interpolées). Afin de l'imiter l'impact de l'accès à cette texture qui doit être effectué pour chaque échantillon, sa taille est minimisée au maximum (en fonction de la qualité de rendu souhaitée) afin qu'un maximum de ses données soient conservée dans le cache de textures.

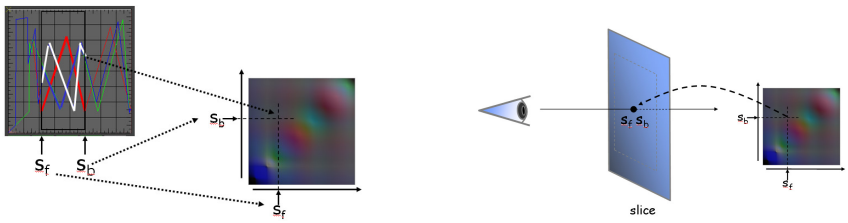


FIG. 3.17 – Illustration de l'utilisation d'une fonction de transfert pré-intégrée 2D lors de la prise d'un échantillon dans les données.

3.4.6 Performances des deux algorithmes

L'algorithme sans pile est de l'ordre de 50% plus rapide que l'algorithme avec mini-pile sur l'ensemble des tests réalisés (les résultats seront présentés section 3.7.1).

Bien que cet algorithme économise un grand nombre de calculs, nous expliquons cette différence de performances par le fait sur le G80, les tableaux pouvant être indexés dynamiquement sont placés par le compilateur de *shaders* dans la zone mémoire appelée (ironiquement ?) "locale" accessible par les multi-processeurs (*cf.*, section 2.1.2). Cette information nous a été confirmée récemment par NVidia et le problème est que l'accès à cette mémoire s'avère beaucoup plus lente que l'accès aux registres (d'un facteur pouvant aller jusqu'à 50x). Elle est en effet localisé dans la mémoire graphique (*device memory*), la même que celle utilisée pour les textures, mais est en plus non-cachée (les données ne sont pas transférées par groupes et conservées dans une zones temporaires pour ré-utilisation comme c'est le cas pour les l'accès aux textures) ce qui signifie que tout accès à cette mémoire aura le coût maximale de latence et de transfert depuis la mémoire graphique. L'interface CUDA offre une solution à ce problème : la zone de mémoire partagée entre *stream processeurs* (*cf.*, section 2.1). Celle ci a en effet un coût d'accès bien meilleur (le coût d'accès à un registre dans le meilleur des cas d'utilisation). Cette piste sera certainement explorée dans des travaux futurs.

3.5 Chargement dynamique et mise à jour progressive de la structure

Nous verrons dans cette partie la stratégie que nous avons développée afin de permettre la visualisation de scènes massives grâce à un *chargement dynamique* et *adaptatif* des données en fonction de la position du point de vue. L'idée générale derrière ce mécanisme est de borner l'utilisation mémoire en fonction de ce qui est visible, de la résolution de données nécessaire au point de vue courant (résolution dite *"utile"*), et non plus en fonction de la scène ce qui permet ainsi de traiter des scènes arbitrairement vastes.

Outre la détermination de ce qui est visible et qui sera abordé dans la section suivante, ce mécanisme suppose la mise en place de stratégies de niveaux de détails (LOD, *Level Of Details*) qui sera présenté section 3.5.1 et va de pair avec une méthode de gestion de la mémoire efficace permettant une ré-utilisation des données précédemment stockées présentée section 3.5.2.

Cette stratégie sera complétée dans la section 3.6 afin d'y ajouter la prise en compte de la visibilité des données (*i.e.,* masquage des volumes non visibles donc inutiles, des données d'arrière plan).

3.5.1 Gestion des niveaux de détails

La résolution utile dépend de deux facteurs : un facteur dynamique qui est la distance des données au point de vue et un facteur statique, intrinsèque aux données, qui est leur fréquence de variation. Des données fortement constantes et avec peu de variations nécessitent en effet moins de résolution que des données à fortes variations, c'est ce facteur qui nous a permis de construire notre structure hiérarchique (*cf.,* section 3.3.4) .

La stratégie de niveaux de détails intervient à deux niveaux dans la structure :

- Au niveau des briques, celles ci étant étant multi-résolution, les différentes résolutions utilisées devront être gérées.
- Au niveau de la structure N^3-tree, lorsque l'intervention sur la résolution des briques n'est plus suffisantes et que leur taille doit être remise en cause dans le but de couvrir un plus espace plus ou moins vaste. Dans ce cas, la feuille de l'arbre supportant la brique doit être soit subdivisée soit fusionnée avec ses feuilles soeurs pour remonter dans la hiérarchie.

Choix du niveau de LOD des briques

Le niveau de détail des briques est déterminé directement en fonction de la distance au point de vue. Le choix de cette résolution est dicté par la contraintes liée à la notion de MIP-mapping et qui est que la résolution idéale est celle pour laquelle un voxel projeté recouvre exactement la surface d'un pixel à l'écran.

Pour simplifier cette détermination, nous considérons des pixels et des voxels de formes sphériques et circulaires, la résolution ne dépendant ainsi pas de l'orientation des voxels par rapport à la surface de projection. La taille idéale d'un voxel devient alors proportionnelle à sa distance au point de vue. Une brique contenant plusieurs voxels de taille identique, un minorant de la résolution du bloc (dépendant de la distance du plus éloigné de ses angles au point de vue) doit être déterminé ainsi qu'un majorant (dépendant de la distance de l'angle le plus proche).

L'ensemble des niveaux de LOD contenus dans cet intervalle seront donc nécessaire lors de la traversée du bloc par les rayons et si ils ne sont pas déjà présents dans le *stockage*, doivent être chargés.

Mise à jour dynamique de la structure

Lorsque de nouvelles données deviennent nécessaires, la structure de données doit être mise à jour pour y inclure les nouvelles données. Le coût de cette mise à jour doit être limité afin d'impacter au minimum la fluidité de navigation. De ce fait, l'ensemble des données ne doivent pas être re-chargées en même temps à chaque déplacement de l'utilisateur entraînant une modification de la résolution utile.

Ce coût intervient principalement lors du chargement de données pour les briques qui sont les plus volumineuses (nous utilisons en effet des briques de N^3 voxels qui peuvent représenter de l'ordre de

16Ko à 1Mo de données, nécessitant ainsi entre 60 microsecondes et 4 millisecondes). La mise à jour de noeuds de l'arbre fait intervenir des données beaucoup plus petites (quelques octets par noeuds), mais également plus éparses en mémoire et de ce fait mises à jour individuellement. Ceci entraîne un coût supplémentaire lié aux latences de transferts.

De ce fait, un compromis doit être trouvé entre la fréquence de mise à jour des briques et celles de l'arbre afin de minimiser le coût moyen de mise à jour des données. Ce paramètre est controlé par la définition pour chaque brique d'une borne inférieure et supérieure de LOD. Ces bornes sont déterminées en fonction de bornes fixées globalement pour une scène, et également en fonction de la *fréquence de variation* des données.

Stratégie de mise à jour du N^3-tree

Lorsque les bornes de niveaux de LOD sont atteintes pour une brique, une stratégie de modification de la structure hiérarchique elle-même doit être mise en place. L'idée est de fusionner les briques reliées à un même noeud de l'arbre dans le cas ou le niveau de LOD utile est supérieur à la borne supérieure de ces briques, ou de décomposer la brique concernée en N^3 nouvelles briques dans le cas ou la borne inférieure est atteinte.

Cet aspect n'a pas été abordé pendant ce stage et fera l'objet de travaux futurs. Nous nous limiterons donc au mécanisme de gestion mémoire lié à la mise à jour du *stockage* des briques de voxels.

3.5.2 Le problème de la gestion mémoire

Le problème dans notre contexte concerne la limite en terme de vitesse de transfert existant entre la mémoire centrale dans laquelle sont générés ou stockées les données, et le GPU en charge de leur rendu.

Il s'agit en fait d'un problème très général intervenant dans toutes les problématiques de gestion de gros volumes de données. Les données volumineuses doivent en effet être stockées dans des mémoires à forte capacité de stockage, mais ces mémoires sont également celles ayant le temps d'accès et la vitesse de transfert la moins bonne sur leur bus. Lors de l'utilisation des données, celles-ci doivent alors être transférées dans des zones mémoires à la taille plus réduite, mais dotées de meilleurs capacités d'accès et plus proche des unités de traitement. Il s'agit en fait d'un problème intervenant à de multiples niveaux dans diverses architectures de traitement :

- Dans le cadre de l'utilisation par le CPU de données issues d'un disque dur : Entre disque dur et mémoire centrale, mémoire centrale et cache de niveau 1 du processeur, cache de niveau 1 et cache de niveau 2 au sein du processeur puis cache de niveau 2 et registres avant utilisation dans les unités de calcul.
- Pour l'utilisation par le CPU de données issues d'un réseau : Entre deux machines du réseau (*Streaming*), entre carte réseau et mémoire centrale de ces machine, mémoire centrale et CPU etc.
- Dans notre cadre dans le cas de données issues du disque dur : Entre disque et mémoire centrale, mémoire centrale et mémoire graphique du GPU, mémoire graphique du GPU puis registres des unités de *shaders* via le cache de textures.

A chaque niveau, l'utilisation de la mémoire disponible doit être optimisée et effectuée de manière efficace afin de limiter les transferts coûteux depuis la mémoire lente de plus grande capacité située en amont tout en permettant de répondre au maximum aux besoins de l'unité de traitement située en aval (schéma *producteur/consommateur*).

Dans cet optique, les données stockées doivent être prioritarisées afin de déterminer celles qui auront le plus de chances d'être réutilisées et celles dont l'occupation mémoire pourra être réutilisé pour le stockage d'autres blocs (stratégie dite *LRU*).

3.5.3 Gestion du cache de briques

Stratégie LRU

Pour permettre une réutilisation optimale de la mémoire dans le cache tout en conservant les données encore utiles, nous avons mis en place une stratégie LRU (*Last Recently Used*, cette stratégie

fait en sorte de réutiliser en priorité les données n'ayant pas été utilisées depuis le plus longtemps) au niveau du cache de briques décrit section 3.3.2. Lorsqu'une nouvelle brique ou résolution de brique devient nécessaire et doit être chargée, deux cas sont possibles : soit le cache n'est pas plein et dans ce cas la brique est simplement transférée dans le prochain emplacement libre, soit le cache est plein et dans ce cas l'emplacement contenant la brique n'ayant pas été utilisée depuis le plus longtemps est réutilisé.

Lorsque cette situation se produit et qu'une brique encore utilisée par la structure de stockage a été détruite, la structure hiérarchique est mise à jour afin d'invalider le niveau de LOD correspondant dans la feuille contenant la brique.

Implémentation

Alors que les données volumiques utiles sont maintenues sur GPU (le cache de briques), le fonctionnement de ce cache est gérée coté CPU. C'est en effet via l'API graphique que les transferts de données vers le GPU sont contrôlés (fonction *glTexSubImage* d'OpenGL). De plus, il n'existe pas à l'heure actuelle de méthode permettant le maintient d'un cache LRU avec mise à jour d'un drapeau d'utilisation (permettant de mémoriser la dernière utilisation de chaque élément et donc nécessaire à la gestion de la priorité dans le cache) coté GPU. Cela est dut au fait que le traitement en flux inhérent au modèle de shaders graphiques ne permet pas d'écriture aléatoire en mémoire. Cette limite est par contre partiellement levée par l'API CUDA introduite section 2.1.3 et l'utilisation de cette possibilité pourait faire partie d'un travail futur.

Maintient de la dernière utilisation La stratégie LRU suppose pour chaque niveau de LOD de chaque brique chargée le maintient et la mise à jour d'un *drapeau* de dernière utilisation. Ce drapeau est mise à jour et maintenu coté CPU pendant la phase de détermination de visibilité décrit section 3.6.

Structures de données Pour permettre une manipulation coté CPU de la structure N^3-tree (le *stockage* d'indirection présenté section 3.3.2), cette dernière est maintenue en parallèle en mémoire centrale et en mémoire graphique. Une liste ordonnée par ordre de dernière utilisation est également maintenue et permet la récupération du prochain emplacement libre dans le cache de briques. Une structure similaire au stockage d'indirection de briques (*cf.,* section 3.3.3) est également maintenue en mémoire centrale et permet l'accès direct aux données concernant le noeud propriétaire (qu'il faudra prévenir en cas de destruction de ses données.

Transferts asynchrones La mise à jour du cache de briques est effectuée via l'utilisation des PBO (*Pixel Buffer Object*) d'OpenGL [Groc]. Il s'agit de zones tampons pouvant être localisées dans des zones spéciales en mémoire centrale et permettant le transfert de données asynchrones avec le GPU via l'utilisation du contrôleur DMA (*Direct Memory Access*) présent sur le matériel. Ce mécanisme permet ainsi l'initiation d'un transfert de données depuis le CPU puis la reprise immédiate du fil d'exécution du programme pour la réalisation d'autres calculs, le transfert étant réalisé en parallèle sans intervention du processeur central ou du GPU.

Ces tampons peuvent être *mappées* dans la zone d'adressage de l'application afin de permettre l'adressage de leur contenu pour le remplir, ou utilisés directement lors d'opérations graphiques (typiquement la mise à jour d'une texture). Les données chargées depuis le disque ou générées à la volée sont ainsi directement placée dans un de ces tampons, celui ci étant ensuite utilisé pour l'opération de mise à jour de la texture 3D contenant le cache sur GPU.

3.5.4 Stratégies pour le temps réel

Malgré cette stratégie de mise en cache des données en cours d'exploration, il est possible que lors d'un mouvement de caméra trop brusque ou dans une zone non-encore visitée une quantité trop

importante de données deviennent visibles et doivent être chargées, nécessitant ainsi un transfert d'une durée supérieur au temps disponible par image pour le maintient d'une fréquence de rafraîchissement temps réel.

Il est important de limiter au maximum l'apparition de telles situation (par exemple par prévision et chargement anticipé) et lorsque celles-ci se produisent, il faut tout de même maintenir une fréquence de rafraîchissement correcte (au minimum 10Hz).

Pour limiter les phénomènes de ralentissement brutal ou de gèle de l'animation lors de ces chargements, on étale les chargements nécessaires sur plusieurs images en limitant le temps disponible pour cette opérations dans chaque image. Des mécanismes plus complexes de prédictions pourraient également être mis en oeuvre et feront peut être l'objet de travaux future.

3.5.5 Le modèle de producteur de données

Cette stratégie de gestion du cache s'inscrit dans un modèle producteur/consommateur plus général que nous avons développé sur le modèle proposé par [LDN04]. L'élément central de cette architecture est un modèle de producteur de données génériques. Ce producteur fournit une interface de requête de données qui permet, à partir d'une position et d'une taille dans la scène (définie entre 0.0 et 1.0 sur chaque axe) ainsi que d'une résolution désirée, d'obtenir les données directement dans une zone mémoire spécifiée (celle-ci pouvant être une zone classique ou un *buffer object* OpenGL pour le transfert sur GPU). Le schéma global de cette architecture de production de données est présenté figure 3.18.

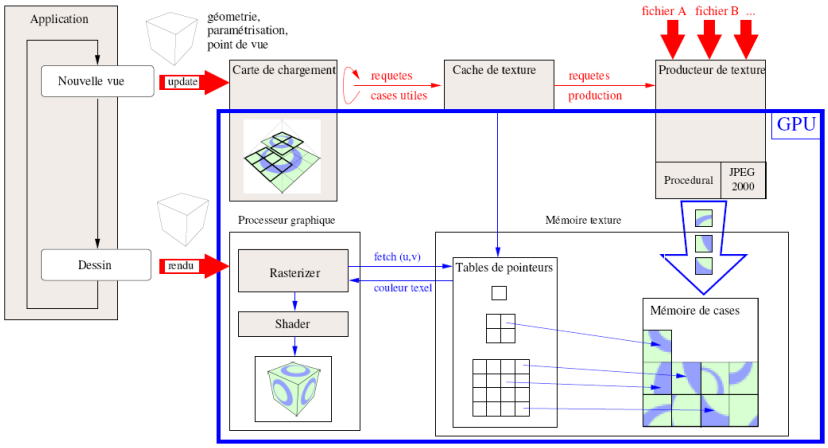


FIG. 3.18 – Architecture de gestion des données texture proposée par [LDN04] que nous avons adaptée à la gestion de données volumiques.

3.6 Prise en compte de la visibilité

Comme expliqué dans section précédente, la distance des briques au point de vue détermine la résolution qui sera chargée et rendu. En plus de cette distance, la prise en compte de la visibilité des briques, c'est à dire du fait qu'elles se trouvent soit hors du champs de vision (appelé *view frustum*), soit masquée par une autre brique se trouvant plus près du point de vue, permet une économie très important dans le nombre d'éléments générés et chargés sur le GPU. Cette économie est d'autant plus importante que dans notre contexte, l'occupation mémoire des données volumique atteint rapidement des tailles très importantes. A titre d'exemple, une simple grille de 256^3 voxels stockés sur 32bits représente 65Mo de données, une grille de 512^3 représente déjà 500Mo et tient tout juste en mémoire graphique. De plus, le transfert d'une telle masse nécessite en théorie de l'ordre de respectivement 200ms et 2s.

Frustum culling

La prise en compte de la visibilité des éléments en fonction de leur localisation à l'intérieur ou hors champs de vue (appelé *frustum culling*) est un éléments classique du rendu de scènes en informatique graphique. Mais il est généralement utilisé dans une optique d'économie de rendu dans le cadre du rendu par *z-buffer* et *rasterisation* (opération qui dans notre cas est réalisé directement par l'algorithme de rendu par lancer de rayons), l'économie en terme de transfert et d'espace mémoire étant généralement un élément secondaire du fait de la taille relativement faible des données de description de surfaces nécessaires. Dans notre cas, cet élément est crucial et doit être pris en compte avant même la génération des données et leur transfert. Ce *culling* est effectuée de manière hiérarchique directement sur la structure de données présentée section 3.3 et s'avère ainsi extrêmement rapide.

3.6.1 Description de notre méthode d'*occlusion culling* volumique

Le frustum culling permet d'éliminer efficacement l'ensemble des éléments à l'extérieur du champs de vision mais n'élimine pas les éléments présents dans le champs de vue qui sont occultés par des voxels plus près du point de vue.

Il s'agit d'un problème difficile dans le cadre du rendu de données volumiques. A la différence du cas surfacique traité par [LDN04], il n'est pas possible de prévoir la visibilité d'un bloc sans avoir effectué le rendu des blocs situés plus près du point de vue et l'occultant potentiellement. Dans le cadre de données potentiellement dynamiques dans lequel nous nous plaçons, il n'est absolument pas possible de pré-calculer cette visibilité.

Une méthode de détection de visibilité en espace image est donc indispensable. Nous nous sommes rapidement orienté vers l'utilisation des tests d'occlusions fournis pas les GPU. Il s'agit d'une fonctionnalité permettant d'obtenir le nombre de fragments ayant été inscrit dans le tampon image pour une série de primitives rendues. Utilisée en conjonction avec le test de profondeur (*depth test*), elle permet de détecter la visibilité des primitives à la granularité des pixels (seules les primitives partiellement ou totalement visibles auront des fragments inscrits dans l'image).

Adaptation des tests d'occlusion au rendu volumique

Dans notre cas, le critère de visibilité d'un bloc n'est pas simplement la présence d'un bloc plus proches du point de vue mais dépend également de l'opacité de ce bloc. Plus précisément, la visibilité d'un voxel le long d'un rayon est déterminée par la profondeur à laquelle le rayon a atteint l'opacité totale, si cette profondeur est inférieure à la profondeur du voxel, celui ci n'est pas visible, si elle est supérieure il l'est. Ce principe est illustré figure 3.19. Si le critère s'applique à un bloc entier de voxels, il faut considérer le bloc visible si un seul de ses voxels n'est pas occulté par l'avant-plan.

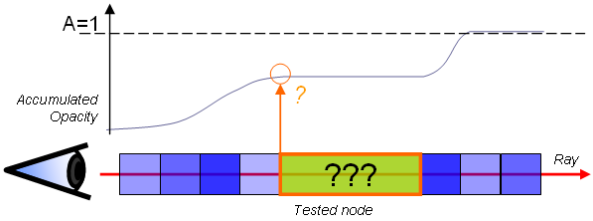


FIG. 3.19 – Illustration de la visibilité d'un bloc volumique le long d'un rayon en fonction de l'opacité accumulée.

Pour permettre la détection de la visibilité des briques de voxels, nous avons modifié notre algorithme de rendu afin d'inscrire dans le *framebuffer* pour chaque rayon la profondeur à laquelle celui-ci a atteint l'opacité maximale si il l'a atteinte et la profondeur maximale (1.0) dans le cas contraire.

La détection de la visibilité d'un bloc se fait ensuite en rendant la boite englobante de ce dernier (surface triangulée) en activant le mécanisme d'*occlusion query* (ou test d'occlusion) : une boite-test englobante est rasterisée (sans tracé réel) et, si au moins un fragment passe le test de profondeur cela signifie qu'au moins un rayon sur la trajectoire duquel il se trouve n'a pas atteint son opacité totale et le bloc doit être généré et chargé. En fonction de la *charge de chargement*, il est possible de jouer sur le nombre de fragments nécessaire au chargement d'un bloc afin de charger en priorité les blocs les plus visibles.

Il est important de noter que cette méthode suppose un parfait alignement entre les projections et position du point de vue du rendu OpenGL et de notre rendu par lancer de rayons. Il est en effet important de faire correspondre exactement la géométrie projetée des éléments dans les deux modes afin d'assurer une détection correcte de la visibilité.

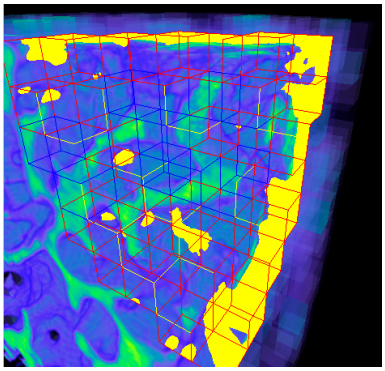


FIG. 3.20 – Illustration du mécanisme de détection de visibilité. Les cubes jaunes représentent la boite englobante des noeuds testés successivement.

Algorithme de détection

Ce test doit être appliquée de manière intelligente sur les éléments de notre structure de données afin de détecter et de pouvoir générer et charger de manière efficace les briques visibles. La principale difficulté réside dans l'asynchronisme : les taches de tracé de blocs et de test de visibilités de blocs sont complètement entremêlées. La question de la visibilité de tels blocs est posée peu avant d'avoir a le tracer, mais le mécanisme de test par occlusion *query* prend du temps (c'est lui-même un rendu simplifié), il faut donc anticiper la demande. Réciproquement, trop l'anticiper revient à ignorer la matière pas encore tracée qui peut pourtant participer à occulter le bloc testé. Il faut donc soigneusement choisir la stratégie d'ordonnancement des requêtes de tracé et de test afin de pouvoir recouvrir au maximum les temps de rendu nécessaires à ce mécanisme avec le chargement des blocs déjà détectés comme visibles.

Les tests d'occlusion peuvent en effet être effectués de façon totalement asynchrone par rapport à l'exécution du programme coté CPU. Après leur initialisation, le contrôle est rendu immédiatement au programme qui est libre de poursuivre son exécution et le GPU est en charge d'en réaliser le rendu. L'état des tests peut être interrogé afin de savoir si ils ont été exécutés et si le résultat (le nombre de fragments visibles) est disponible. L'interrogation du résultat est un point de synchronisation : si le test n'a pas encore été réalisé au moment de cette interrogation, le programme attend le résultat.

Pour pouvoir profiter de cet asynchronisme dans notre algorithme, nous avons organisés les tests de façon à permettre un recouvrement maximal des temps de rendu des tests par les temps de chargement (exécutions en parallèle des deux opérations) des briques. A partir de la liste des briques présentes dans le champs de vision mais non chargées fournie par l'étape de frustum culling (*cf.*, section 3.6.2), on commence par initialiser n tests d'occultation sur n briques prises à partir de la position $i = f(n)$

dans la liste. On effectue ensuite le chargement des i premières briques de la liste avant de récupérer successivement le résultat des n tests en relancant les tests pour les $n' = f^{-1}(v)$ (avec v le nombre de tests ayant renvoyé au moins un voxel visible) briques suivantes dans la liste. Les v briques visibles parmi les n testées sont alors chargées et l'algorithme est répété. La fonction f est choisie en fonction des temps de chargement et de rendu des tests de façon à permettre un recouvrement maximum des deux opérations. Dans le cas ou un nombre insuffisant de briques seraient détectées comme visibles lors d'une passe, des briques peuvent être préchargées sans test comme lors de la première itération. Cet algorithme est illustré figure 3.21.

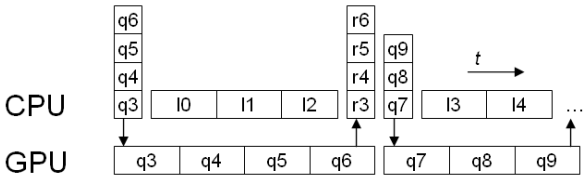


FIG. 3.21 – Illustration du principe de notre algorithme. Les numéros correspondent au numéro des briques dans la liste ordonnée. Les lettres q représentent les occlusion queries, les r leur résultat, les l les opérations de génération et de chargement.

Mise à jour du rendu

Entre chacune des passes de l'algorithme [2] présenté dans la section précédente, une passe de rendu volumique partiel de la scène peut être effectuée afin de compléter l'image et ainsi de mettre à jour les données occultantes afin de prendre en compte dans le test de visibilité les briques nouvellement chargées. L'opération de rendu total de la scène étant particulièrement coûteuse, la fréquence de réalisation de cette opération devra être choisie avec parcimonie.

Il serait intéressant dans ce cadre de fournir une mise à jour progressive du rendu via un mécanisme de reprise du lancer de rayons entre les passes. Nous pensons pour cela stocker en espace image, en plus de la couleur et de l'opacité accumulée, l'état de chaque rayon comprenant position spatiale, direction, position dans la structure hiérarchique et état de la pile le cas échéant (état des DDA). Cela semble possible en utilisant le mécanisme de rendu vers texture (*cf.,* section 2.1) ainsi que la possibilité de rendre vers de multiples tampons simultanément. Cette problématique sera abordée dans un travail futur.

3.6.2 Génération d'une liste ordonnée

Pour éviter au maximum d'effectuer des tests sur des blocs situés derrière d'autres blocs également non chargées et dont le résultat du test n'est pas connu (ceci pourrait en effet entraîner le chargement inutile d'un bloc occulté), il est préférable de fournir à l'étape de détection de visibilité une liste de briques ordonnées de la plus proche à la plus éloignée du point de vue,

Parcours ordonné de la structure

La structure de N^3-tree que nous avons utilisé a l'avantage de permettre un parcours aisé de ses noeuds dans l'ordre d'occultation, c'est à dire en imposant que tout noeuds déjà atteint ne pourra être occulté par un noeud atteint dans la suite du parcours.

Ce parcours consiste à effectuer une traversée de l'arbre en profondeur d'abord, en imposant à chaque niveau un ordre de parcours des sous noeuds (parcours de la grille représentant le noeud) selon les axes (x, y puis z ou z, x puis y etc.) choisit en fonction de la direction du point de vue. L'idée est d'effectuer en priorité le parcours selon le plan le plus proche du point de vue, cela est obtenu en choisissant les axes selon les composantes du vecteur de vue, l'axe ayant la plus petite composante sur le vecteur de vue étant pris en premier. Ce principe est illustré en 2D figure 3.22. L'élément de départ

[2] une passe regroupe l'initialisation d'une série de *queries*, le chargement d'un ensemble de blocs et la récupération du résultat des *queries*.

du parcours dans chaque grille est simplement déterminé par $E_{xyz} = step(0, View_{xyz}) * (N-1)$, avec $step(0, View_xyz) = \{0\ si\ View_{xyz} < 0,\ 1\ sinon\}$, $View_{xyz}$ le vecteur direction du point de vue et N le nombre de subdivision des noeuds.

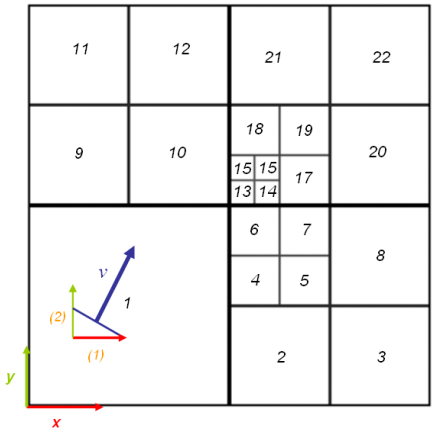


FIG. 3.22 – Illustration sur un quadtree de l'ordre de parcours de la structure hiérarchique en fonction de la direction du point de vue. Ici parcours selon l'axe x, puis y et départ en bas à gauche à chaque niveau (noeud (0,0)).

Ordre de distances strict

C'est donc l'ordre de parcours que nous utilisons lors de l'étape de *frustum culling* pour construire la liste de briques visibles. Bien que l'on obtienne ainsi un ordre de visibilité, cela ne fournit pas une liste ordonnée en fonction de la distance au point de vue. Un tel ordre serait pourtant utile afin de charger en priorité les briques les plus proches du point de vue. La liste de briques obtenue après le parcours décrit au paragraphe précédent est donc retriée dans une passe intermédiaire avant le test d'occlusion. Effectuer ce tri à partir d'une liste déjà ordonnée par un critère proche permet une convergence rapide de l'algorithme de tri et améliore ainsi grandement son efficacité. Ceci sans aucun surcoût, le parcours ordonné de l'arbre n'étant pas plus coûteux qu'un parcours quelconque. Ce tri permet de plus de beaucoup mieux répartir spatialement les blocs traités consécutivement par l'algorithme de détection de visibilité. Cette répartition diminue très largement les risques que deux noeuds testés simultanément puissent s'occulter l'un l'autre ce qui améliore grandement l'efficacité de l'algorithme de visibilité. La différence entre l'utilisation et la non utilisation de ce tri est illustrée figure 3.23.

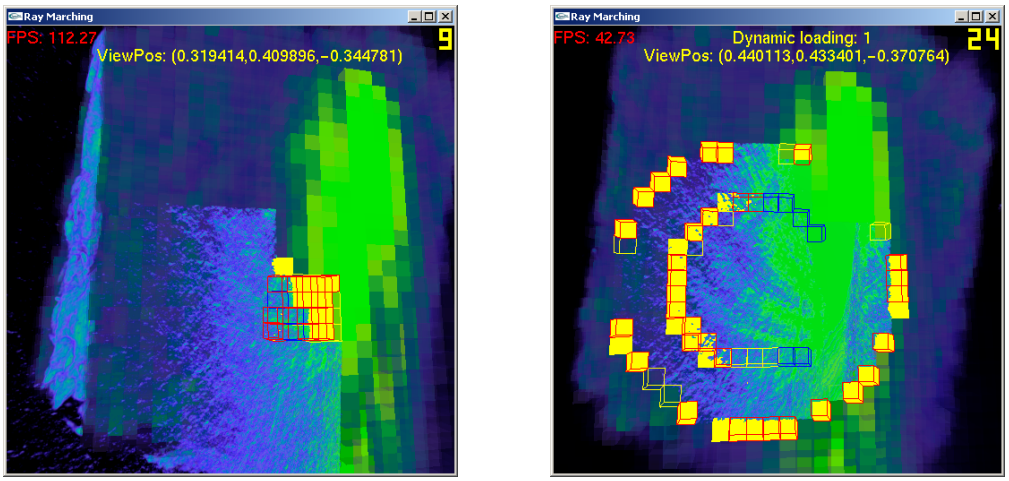


FIG. 3.23 – Comparaison de l'ordre de traitement des noeuds, à gauche sans étape de tri, à droite avec. Les blocs jaunes pleins représentent les briques dont l'occlusion est testée, les rouges les briques chargées et les bleus les briques occultés.

3.6.3 Stratégie pour le temps réel

L'algorithme de détection de visibilité que nous avons décris s'avère généralement trop coûteux pour permettre sa réalisation lors d'une unique passe de rendu tout en conservant un rendu interactif. Pour éviter le ralentissement brusque de l'affichage qu'il pourrait entraîner lors d'un changement brutal de point de vue par exemple, son exécution peut être étalée sur plusieurs images d'affichage. Le temps qui est alloué à l'algorithme est alors contraint pour assurer une visualisation interactive. Dans ce cas, les valeurs moyennes stockées dans la structure hiérarchique sont utilisées.

A noter que ceci n'handicape que marginalement l'utilisation : la camera peut continuer a explorer interactivement les données volumiques en présence. Au pire, la résolution d'un bloc peut être légèrement insuffisante, voire des blocs peuvent manquer (essentiellement en cas de changement total de vue, au démarrage et lors de changement brusques de direction du regard), ou plus probablement quelques pixels de blocs lointains.

3.6.4 Extension pour l'exploitation de la structure hiérarchique

Alors que nos données sont stockées de manière hiérarchique dons notre structure, cette hiérarchie n'est pas exploitée dans l'état actuel de l'algorithme pour accélérer les tests d'occlusion. L'idée serait d'utiliser le modèle proposé par [BWPP04] afin de tester hiérarchiquement l'occlusion des noeuds visibles. Cette méthode fournit en effet un mécanisme permettant d'exploiter la hiérarchie, sans pour autant introduire une latence liée à l'attente du résultat d'un noeud père pour tester les noeuds enfants.

3.7 Résultats et discussions

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

3.7.1 Performances comparées

Nous avons comparé les performances de nos deux algorithmes de rendu (Avec et sans mini-pile) avec les performances d'un rendu par *ray-marching* classique effectué dans une grille régulière contenant l'ensemble des données de la scène. Différentes position du point de vue ont été mesurées influant ainsi sur la proportion vu du volume total et deux résolutions différentes de la grille virtuelle contenant la scène ont été testées. Les résultats de ces tests sont présentés figure 3.24.

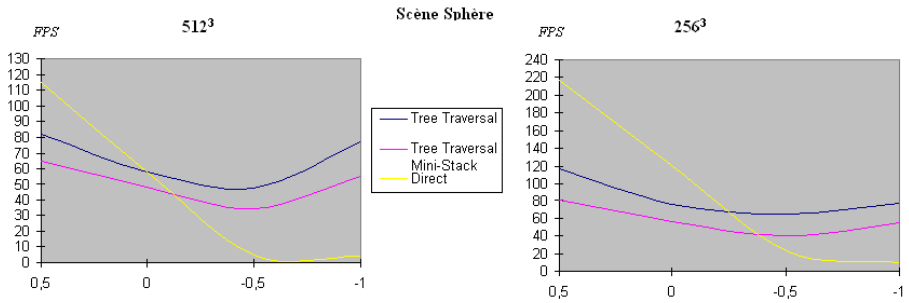


FIG. 3.24 – Performances en images par seconde du rendu de la scène *sphère* en fonction de la position du point de vue. A gauche les résultats pour une résolution virtuelle totale de 512^3, à droite pour une résolution de 256^3

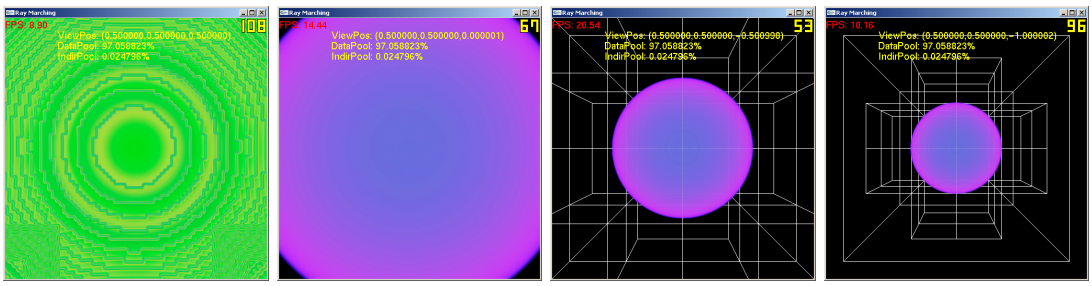


FIG. 3.25 – Positions des points de vue (0.5, 0.0, −0.5, −1.0) utilisés pour le test de performances.

Ces résultats montrent tout d'abord un impact sur les performances de la position du point de vue beaucoup moins important pour nos deux méthodes que pour la méthode de ray-marching direct. Ce plus faible impact peut s'expliquer par la capacité à "sauter" les espaces vides fournie par la structure qui fait que la distance au volume n'impacte que peu les performances ce qui n'est pas le cas pour la méthode directe. Il est important de noter que dans ces tests l'arrêt précoce des rayons était activé et que les deux méthodes arrêtaient donc relativement rapidement leur parcours après avoir atteint les données. Ce faible impact s'explique sans doute également par un coût de structure plus important dans le cas de nos méthodes et qui impacte les performances pour les points de vue à faibles données.

On remarque également un écart quasiment constant entre la méthode avec pile et la méthode sans, le même algorithme global étant utilisé cet écart est très certainement du au surcoût d'accès aux données de la pile par rapport au coût de leur re-calcul dans la première méthode. Notre coût de

structure n'est compensée que lorsqu'il y a assez d'espaces vides pour en accélérer le parcours et nos méthodes nécessitent donc de larges espaces de données pour être performantes. La méthode direct ne pouvant pas être utilisée avec des grilles de plus de 512^3, nous ne pouvons pas mesurer l'écart de performance pour des espaces de données plus vastes et complexes.

Un autre élément intéressant à noter concerne l'écart de performance des algorithmes en fonction de la résolution de la grille virtuelle globale. Alors que cet écart est de l'ordre de 50% pour la méthode directe, il n'est que d'un peu plus de 20% pour notre méthode. Cet écart est en fait dû à l'écart de fréquence d'échantillonage, lorsque la résolution des données double, le nombre d'échantillons pris le long des rayons est doublé également. On voit ainsi que la méthode directe est directement influencée par cette différence de fréquence d'enchantillonage et ses performances sont directement liées à ce paramètre alors que notre méthode en est moins dépendante.

3.7.2 Influence des paramètres de la structure sur le rendu

La figure 3.26 montre l'influence des paramètres N (subdivision des noeuds) et Br (résolution des briques) sur les performances du rendu (indiquées en images par seconde). A résolution de grille virtuelle identique, ces deux paramètres influencent la profondeur de l'arbre. La scène utilisée ici est une sphère creuses de diamètres extérieur 0.3 et intérieur 0.25 occupant 4.7% du volume total de la scène (scène cubique de 1^3). Les paramètres sont notés au format $N/Br(profondeur)$.

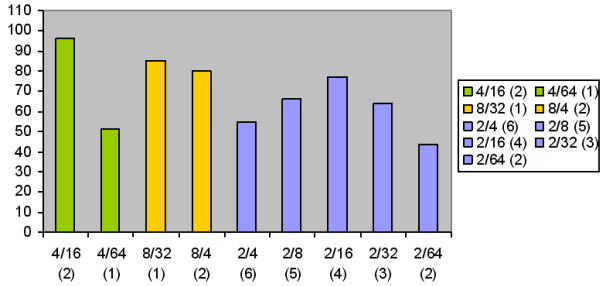


FIG. 3.26 – Performances du rendu de la scène *sphère* en fonction de divers paramètres de structure et indiquée en images par seconde.

Ces performances montrent tout d'abord que la configuration $N = 4$ et $Br = 16$ fournit les meilleures performances sur cette scène avec une profondeur de l'arbre égale à 2. On remarque également que les trois meilleures performances sont obtenues avec des subdivisions élevées des noeuds ($N = 4$ et $N = 8$) mais que la meilleure n'utilise qu'une subdivision de 4 et que la $4ieme$ meilleure performances n'utilise qu'une subdivision de 2. Ce résultat s'explique par l'intervention de deux phénomènes : le gain apporté à l'efficacité du rendu par l'utilisation de grilles régulières à chaque noeuds et l'espace vide qui a pu être ignoré grâce à l'utilisation de briques plus petites et d'une profondeur de l'arbre plus importante. Bien que l'on ne puisse pas tirer de conclusion sur une unique scène, ce test confirme notre intuition de départ sur le rôle de ces paramètres. La configuration la plus performante est en effet la plus équilibrée sur ces deux plans.

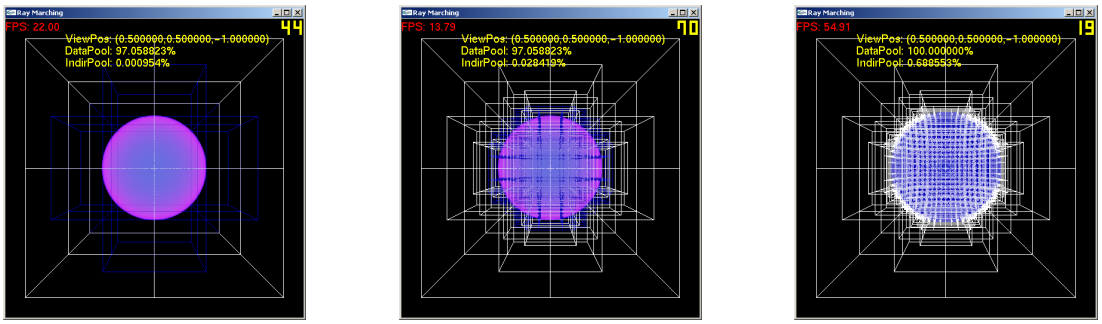


FIG. 3.27 – Rendus effectués sur la scène test sphère avec la structure de données de visualisée pour les paramètres 2/64, 2/16 et 2/4.

3.7.3 Discussion

Notre méthode permet de naviguer de manière interactive au sein de scènes volumiques vastes qui nécessiteraient des résolutions supérieures à 512^3 (qui est la limite de résolution de grille pouvant être stockée sur GPU en se limitant à une unique information de densité par voxel) si elles devaient être stockées dans une structure régulière. Nous avons pu visualiser des scènes allant jusqu'a 2048^3 et qui représenteraient quelques 34Go de données si elles devaient être stockées dans une grille régulière.

La navigation interactive est assurée au sein de ces données grâce aux mécanismes de gestion mémoire et de chargement dynamique contraint présentés section 3.5. Les données sont transférées de manière efficace vers le GPU en fonction de leur visibilité et de la résolution nécessaire à leur rendu.

Nous avons proposé une structure de données adaptée à la fois au rendu volumique interactif et à la gestion de grosses masses de données volumiques. Cette structure dispose en effet de bonnes qualités pour permettre un temps de rendu peu dépendant de la quantité de données visualisées. Elle fournit de plus une vision globales à partir du GPU des données qu'elle contient ce qui permettra un grand nombre d'extensions pour la gestion de modèle d'éclairages plus poussés, la gestion des ombres ou la mise en oeuvre de divers effets (par exemple, traiter la profondeur de champs serait assez facile).

La gestion de la structure de données entière sur GPU permettra également l'implémentation d'algorithmes de génération de données procédurales ou d'ajout de détails dynamiques directement sur le matériel.

La gestion du mip-mapping volumique ainsi que l'utilisation de fonctions de transfert pré-intégrées permet d'assurer une qualité optimale au rendu quelque soit la distance aux données.

La portée du mécanisme de gestion du niveau de détails des données est pour l'instant limitée à la résolution des briques de voxels, et les briques sont actuellement utilisées à partir d'une profondeur dans l'arbre fixée. Le travail sur la modification de la structure hiérarchique elle-même a été commencé et il permettra -nous l'espérons- une fois menée a son terme, la visualisation de scènes de tailles illimitées et aux détails fractales (sans limite de raffinement).

L'algorithme de détection de la visibilité pourra également être amélioré comme indiqué section 3.6.4. L'algorithme de rendu devrait également pouvoir bénéficier d'améliorations en permettant la reprise du parcours des rayons entre plusieures passes de rendu. Un gain important en performances pourrait également être obtenu via l'utilisation plus directe du matériel graphique à l'aide des nouvelles API de calcul (*cf.*, [nVIa]).

Conclusion et travaux futurs

Le travail réalisé pendant ce stage a permis d'apporter un certain nombre de contributions et d'idées nouvelles pour la visualisation et l'exploration interactive de très grandes étendus de données volumiques et les perspectives ouvertes par cette approche sont extrêmement nombreuses. Les résultats obtenus sont extrêmement encouragent pour l'utilisation de ce genre d'approche dans le cadre du rendu en temps réel de phénomènes volumiques vastes et complexes.

De nombreux problèmes restent ouverts comme la visualisation de volumes beaucoup plus profonds et larges, la gestion et le rendu de données dynamiques ou la prise en compte de l'illumination globale (ombres, inter-réflections etc.). Le développement de modèles de producteurs de données semi-procéduraux permettant l'ajout de détails fins de façon dynamique est également une piste qu'il reste à explorer.

Bibliographie

[AN06] Aymeric Augustin and Fabrice Neyret. Flow-noise en temps réel. Rapport de stage d'option, École polytechnique - Evasion, laboratoire GRAVIR, juin 2006.

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.

[BD06] Lionel Baboud and Xavier Décoret. Realistic water volumes in real-time. In *Eurographics Workshop on Natural Phenomena*. Eurographics, 2006.

[Bli82] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *SIGGRAPH '82 : Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 21–29, New York, NY, USA, 1982. ACM Press.

[BNL06] Antoine Bouthors, Fabrice Neyret, and Sylvain Lefebvre. Real-time realistic illumination and shading of stratiform clouds. In *Eurographics Workshop on Natural Phenomena*, sep 2006.

[BT04] Zoe Brawley and Natalya Tatarchuk. Parallax occlusion mapping : Self-shadowing, perspective-correct bump mapping using reverse height map tracing. In Wolfgang Engel, editor, *ShaderX3 : Advanced Rendering Techniques in DirectX and OpenGL*. Charles River Media, Cambridge, MA, 2004.

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling : Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3) :615–624, September 2004. Proceedings EUROGRAPHICS 2004.

[Car03] Christian Carvajal. Shaken and stirred, *XXX* visual effects. *CINEFEX*, 92, 2003.

[CB04] Per H. Christensen and Dana Batali. An irradiance atlas for global illumination in complex production scenes. In *Rendering Techniques*, pages 133–142, 2004.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94 : Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM Press.

[CDP95] Frédéric Cazals, George Drettakis, and Claude Puech. Filtering, clustering and hierarchy construction : a new solution for ray tracing very complex environments. In *Eurographics'95*, September 1995. Maastricht.

[CGP04] Sharat Chandran, Ajay K. Gupta, and Ashwini Patgawkar. A fast algorithm to display octrees, September 18 2004.

[CHCH06] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *GI '06 : Proceedings of the 2006 conference on Graphics interface*, pages 203–209, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[CN94] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3D texture hardware. Technical report, Chapel Hill, NC, USA, 1994.

[CS94] Daniel Cohen and Zvi Sheffer. Proximity clouds - an acceleration technique for 3D grid traversal. *Vis. Comput.*, 11(1) :27–38, 1994.

[DB89] Jevans D. and Wyvill B. Adaptive voxel subdivision for ray tracing. In *Graphics Interface'89*, pages 164–172, June 1989.

[DN04] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)*, pages 93–102, june 2004.

[Dom] Digital Domain. Digital domain web site. `http://www.digitaldomain.com`.

[Dun04] Jody Duncan. Freeze frames, *The Day After Tomorrow* visual effects. *CINEFEX*, 98, 2004.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, New York, NY, USA, 2001. ACM Press.

[EVG04] Manfred Ernst, Christian Vogelgsang, and Günther Greiner. Stack implementation on programmable graphics hardware. In *VMV*, pages 255–262, 2004.

[FP02] Frisken and Perry. Simple and efficient traversal methods for quadtrees and octrees. *JGTOOLS : Journal of Graphics Tools*, 7, 2002.

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

[GK96] Allen Van Gelder and Kwansik Kim. Direct volume rendering with shading via three-dimensional textures. In *VVS '96 : Proceedings of the 1996 symposium on Volume visualization*, pages 23–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[Groa] ARB/Khronos Group. Manuel de référence OpenGL 1.4. `http://www.opengl.org/documentation/blue_book_1.0/`.

[Grob] ARB/Khronos Group. Site officiel d'OpenGL. `http://www.opengl.org`.

[Groc] Khronos Group. Registre des extensions OpenGL. `http://www.opengl.org/registry/`.

[Had02] et al. Hadwiger, M. High-quality volume graphics on consumer pc hardware. In *Course Notes 42 - SIGGRAPH. 2002*. SIGGRAPH, 2002.

[Har] Hardware.fr. Présentation de DirectX 10 et des GPU compatibles. `http://www.hardware.fr/articles/631-1/directx-10-gpus.html`.

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

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. In *I3D '07 : Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM Press.

[Kap02] Alan Kapler. Evolution of a vfx voxel tool. In *SIGGRAPH 2002 Sketches and Applications*, page 179, New York, NY, USA, 2002. ACM Press.

[KH84] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *SIGGRAPH '84 : Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA, 1984. ACM Press.

[KH05] Joshua Krall and Cody Harrington. Modeling and rendering of clouds on "stealth". In *SIGGRAPH '05 : ACM SIGGRAPH 2005 Sketches*, page 85, New York, NY, USA, 2005. ACM Press.

[Kir] David Kirk. The CUDA hardware model. `courses.ece.uiuc.edu/ece498/al/lectures/lecture8-9-hardware.ppt`.

[KKH02] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3) :270–285, 2002.

[KPHE02] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *VIS '02 : Proceedings of the conference on Visualization '02*, pages 109–116, Washington, DC, USA, 2002. IEEE Computer Society.

[KTI+01] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi. Detailed shape representation with parallax mapping. In *In Proceedings of ICAT 2001*, pages 205–208, 2001.

[KW03] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[KW05] Jens Krüger and Rüdiger Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3), 2005.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes : A high resolution 3D surface construction algorithm. In *SIGGRAPH '87 : Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[LDN04] Sylvain Lefebvre, Jerome Darbon, and Fabrice Neyret. Unified texture management for arbitrary meshes. Technical Report RR5210-, INRIA, may 2004.

[LH06] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *SIGGRAPH '06 : ACM SIGGRAPH 2006 Papers*, pages 579–588, New York, NY, USA, 2006. ACM Press.

[LHN05] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Octree Textures on the GPU, pages 595–613. Addison Wesley, 2005.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94 : Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA, 1994. ACM Press.

[Max86] Nelson Max. Light diffusion through clouds and haze. *Comput. Vision Graph. Image Process.*, 33(3) :280–292, 1986.

[Max94] Nelson L. Max. Efficient Light Propagation for Multiple Anisotropic Volume Scattering. In *Fifth Eurographics Workshop on Rendering*, pages 87–104, Darmstadt, Germany, 1994.

[Max95] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2) :99–108, 1995.

[MHR02] K. Engel M. Hadwiger, J.M. Kniss and C. Rezksalama. High-quality volume graphics on consumer pc hardware. In *SIGGRAPH 2002, Course Notes 42*, New York, NY, USA, 2002. ACM Press.

[Mic] Microsoft. Site officiel de microsoft DirectX. `http://www.microsoft.com/france/msdn/technos/directx.mspx`.

[Ney98] Fabrice Neyret. Modeling animating and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1) :55–70, January–March 1998. ISSN 1077-2626.

[Ney03] Fabrice Neyret. Advected textures. In *ACM-SIGGRAPH/EG Symposium on Computer Animation (SCA)*, july 2003.

[Ney06] Fabrice Neyret. Créer, simuler, explorer des univers naturels sur ordinateur. `http://www-evasion.imag.fr/Publications/2006/Ney06`, 2006. invited paper.

[NS05] Thrane N. and Simonsen. *A comparison of acceleration structures for GPU assisted ray tracing*. PhD thesis, 2005.

[nVIa] nVIDIA. Nvidia Compute Unified Device Architecture programming guide. `http://developer.download.nvidia.com/compute/cuda/0_81/NVIDIA_CUDA_Programming_Guide_0.8.2.pdf`.

[nVIb] nVIDIA. Nvidia developer web site. `http://developer.nvidia.com`.

[nVIc] nVIDIA. Nvidia geforce 8800 technical brief. `http://www.nvidia.com/page/8800_tech_briefs.html`.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *SIGGRAPH '00 : Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 359–368, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3) :703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[Per85] Ken Perlin. An image synthesizer. In *SIGGRAPH '85 : Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[PH89] K. Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89 : Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, New York, NY, USA, 1989. ACM Press.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6) :311–317, 1975.

[PN01] Ken Perlin and Fabrice Neyret. Flow noise. In *Siggraph Technical Sketches and Applications*, page 187, Aug 2001.

[Pur04] Timothy John Purcell. *Ray tracing on a stream processor*. PhD thesis, 2004. Adviser-Patrick M. Hanrahan.

[RE02] Stefan Roettger and Thomas Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *VVS '02 : Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 23–28, Piscataway, NJ, USA, 2002. IEEE Press.

[RKE00] S. Rottger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection, 2000.

[RSEB+00] 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 *HWWS '00 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118, New York, NY, USA, 2000. ACM Press.

[Sch05] Henning Scharsach. Advanced GPU raycasting. In *Central European Seminar on Computer Graphics 2005*, pages 69–76, 2005.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98 : Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, New York, NY, USA, 1998. ACM Press.

[Wik] Wikipedia. Z-buffering algorithm description. `http://en.wikipedia.org/wiki/Z-buffering`.

[WM92] Peter L. Williams and Nelson Max. A volume density optical model. In *VVS '92 : Proceedings of the 1992 workshop on Volume visualization*, pages 61–68, New York, NY, USA, 1992. ACM Press.

[WZF+03] Xiaoming Wei, Ye Zhao, Zhe Fan, Wei Li, Suzanne Yoakum-Stover, and Arie Kaufman. Blowing in the wind. In *SCA '03 : Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 75–85, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.