Préservation de détails par placage d'octree-textures

Julien Lacoste

▶ To cite this version:

Julien Lacoste. Préservation de détails par placage d'octree-textures. Informatique [cs]. Université de Pau et des Pays de l'Adour, 2008. Français. NNT : . tel-00461725

HAL Id: tel-00461725
https://theses.hal.science/tel-00461725

Submitted on 5 Mar 2010

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.

UNIVERSITÉ DE PAU ET DES PAYS DE L'ADOUR

ÉCOLE DOCTORALE DES SCIENCES EXACTES ET DE LEURS APPLICATIONS ED 211

Préservation de détails par placage d'octree-textures

THÈSE

présentée et soutenue publiquement le 15 décembre 2008

pour l'obtention du

Doctorat de l'Université de Pau et des Pays de l'Adour
(spécialité informatique)

par

Julien Lacoste

Composition du jury

Président :	Pr. Christophe Schlick	LaBRI, Université de Bordeaux 2
Rapporteurs :	Pr. Jean-Michel Dischler	LSIIT, Université Louis Pasteur de Strasbourg
	Dr. Christian Bouville	IRISA, Université Rennes 1
Examinateurs :	Pr. Congduc Pham	LIUPPA, Université de Pau et des Pays de l'Adour
	MdC. Bruno Jobard	LIUPPA, Université de Pau et des Pays de l'Adour
Thèse dirigée par :	Pr. Congduc Pham	LIUPPA, Université de Pau et des Pays de l'Adour
Co-directeur :	MdC. Bruno Jobard	LIUPPA, Université de Pau et des Pays de l'Adour

Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour — EA 3000

LIUPPA

Mis en page avec la classe thloria.

Remerciements

Avant toute chose, je tiens particulièrement à remercier les êtres qui me sont le plus chers, mes parents, mes petits frères et bien évidemment tous les membres de ma famille. Je remercie en particulier mon grand-père pour m'avoir offert mon premier ordinateur, mes grands-mères pour toutes ces confitures et ces excellents repas du mercredi midi. Je remercie mes parents pour tout ce qu'ils ont fait pour nous trois, la confiance qu'ils ont eu en moi, et parce que durant cette thèse, me retrouver chez eux m'a fait un bien fou (malgré les moqueries incessantes de mes voyous de petits frères) quand il y avait quelques moments un peu pénibles. Je ne peux pas nommer tout le monde (on est trop nombreux...), mais je salue tous mes cousins et cousines (courage à Mélanie pour la fin de sa thèse), oncles et tantes, on a vraiment de la chance d'être dans une si belle famille, et pour moi il n'y a pas eu de plus grande motivation que celle de se savoir soutenu par vous.

D'un point de vue plus professionnel, je remercie les rapporteurs et membres du jury d'avoir prété attention à mes travaux. Je remercie Wilfrid Lefer de m'avoir offert la chance de démarrer cette thèse, et Congduc Pham la possibilité de la terminer. Je remercie plus particulièrement mon co-encadrant Bruno Jobard pour m'avoir lancé sur la thématique, et pour m'avoir laissé libre de mener ces travaux parfois à l'encontre de ses points de vue. Ses conseils et l'environnement qu'il a mis à ma disposition m'ont sans aucun doute permis de développer de manière favorable cette thèse.

Je remercie mes co-auteurs, avec qui j'ai eu grand plaisir à travailler. Les échanges que j'ai eu avec eux ont été riches d'enseignements et m'ont poussé à améliorer mes travaux. Je remercie donc Tamy Boubekeur, Christophe Schlick, Romain Pacanowski, Mickaël Raynaud, Xavier Granier et Camille Perin (bon courage pour ta future thèse !). Je remercie particulièrement Patrick Reuter qui m'a mis en contact par deux fois avec les membres de l'équipe IPARLA du LaBRI. Je fais une mention spéciale à Romain, Mickaël et Jérôme Baril pour l'agréable semaine passée à Los Angeles (ah ce bar de Beverly Hills, ces restaurants....).

Je remercie tous les membres du département informatique de l'UPPA qui m'ont aidé par leurs conseils avisés, ou avec qui j'ai pu travailler dans divers modules d'enseignements : Sophie et Éric Gouardères, Jean-Michel Bruel, Franck Barbier, Alain Teste et tous ceux que je pourrais oublier. Il est difficile pour moi de faire une dichotomie entre collègues et amis, tant la frontière n'existe plus pour une partie d'entre eux. Je tiens donc à faire un énorme merci à Annig et Laurent Lacayrelle (et leurs deux enfants Maïa et Malo) pour leur générosité et le nombre incalculable de fois où ils m'ont accueilli, et bien entendu pour les petits canards. Je remercie également Laurent pour le VTT (même si c'est un sport de con) et Magma, et d'une manière plus générale toutes les musiques *qui font mal à la tête*[1] qu'il m'a fait découvrir. Je n'oublie pas d'associer à ces hommages Éric Cariou, compagnon d'apéro et de vélo aussi, chez qui on peut toujours aller voir

[1] dixit Annig

son équipe de foot favorite perdre lamentablement (mais vraiment toujours, même si d'accord, la coupe du monde, c'est chez Laurent qu'on l'a perdue). C'est peut-être pour se venger du fait qu'il se cogne toujours (mais vraiment toujours) la tête sur le pare-soleil quand il rentre dans ma voiture.

Avant de poursuivre mes remerciements, je me permets de citer Hélène Raigné-Dabadie, parce que ça va lui faire plaisir. En plus elle est invitée à la soutenance (elle a promis d'agiter des trucs) ! Ça lui était jamais arrivé, alors qu'elle en a inscrit des doctorants dans son petit bureau.

Parmi toutes les personnes fréquentées durant ces ~~3~~ 4 années de thèse, c'est avec les autres doctorants que j'ai le plus échangé et discuté, ce paragraphe leur est dédié. Dans un premier temps je salue nos illustres prédécesseurs, qui lors de notre stage de DEA étaient en pleine rédaction, et n'ont malgré tout pas réussi à nous dissuader de démarrer une thèse : Nicolas Belloir, Mahmoud Tchikou et Pierre Laforcade (merci pour les vacances à Laval !). J'en profite pour saluer nos collègue de DEA de l'époque, Jérôme Lacouture et Arnaud Casteigts. Je remercie les doctorants avec qui j'ai partagé le bureau : Fabien Romeo (un peu au début), Fatou, puis le trio Pierre Loustau, Cyril Ballagny et Julien Lesbegueries. Ils ont supporté ma musique et mes blagues débiles sans jamais se plaindre, et Julien me supporte même depuis le DEUG. Ces quelques années à vos côtés, ponctuées de discussions ridicules ou sérieuses, de pas mal de rigolades et cie ont été très agréables. Je remercie chaleureusement Natacha Hoang avec qui, même si elle n'a (malheureusement) pas partagé notre bureau, j'ai eu grand plaisir à passer du temps, écouter ses rocambolesques aventures, boire des verres en terrasse au lieu de travailler.... J'ai adoré tout ces moments passés avec toi et où que j'aille maintenant je crois que je regretterai ta présence. Je n'oublie pas Guilhem Dupuy, tu sais je ne désespère pas d'arriver à faire cet échange ballon contre apéritif assez vite (mais bon le ballon est tout dégonflé sous mon lit, ton fils il en voudra pas) ! Pour finir je salue tous les doctorants du LIUPPA et leur souhaite une bonne réussite.

Il est temps maintenant de remercier toutes les personnes qui ne sont pas ou plus liées à la fac. Tout d'abord Florent et Christophe, amis depuis le primaire et pour longtemps encore j'espère (et non Christophe tu n'écriras pas de dédicace dans mon manuscrit). Fred le basque pour tous les bons moments passés au début des Sphinx, et après aussi. Clio parce qu'il est toujours de bons (ou mauvais) conseils, puis aussi parce que ça me fait toujours plaisir de le voir, et puis parce qu'on a une haine commune du type là haut qui nous joue tous ces sales tours (ah il doit bien se poiler l'enfoiré). Mes binômes successifs pendant l'université : Benjamin, Fred et Mirentxu. Zeuh, redoutable joueuse de Wordox, qui m'a hébergé lors de mes séjours bordelais. Merci pour ton hospitalité, ton poulet au soja, ton punch coco. Je remercie la merveilleuse Babycole, également redoutable joueuse, pour le week-end agréable dans le sud-est, mais surtout pour tout, tout le reste : les parties et discussions endiablées, les cartes et cadeaux, et tout cet amour qu'elle m'envoie par les réseaux.

Et puis il y a Frida, qui est belle comme un soleil.... Ah non ça s'est pas de moi, mais si on pouvait imaginer la même musique derrière, juste en remplaçant Frida par Princesse. Parce qu'elle aussi est belle comme un soleil, parce qu'elle est la personne la plus extraordinaire que je connaisse, parce que j'ai passé uniquement de très bons moments en sa compagnie, et que ses yeux et son sourire, et aussi ses imitations et ses bras splendides ont été d'un soutien précieux lors de ces quatre années, et que sans elle j'aurais peut-être arrêté plus tôt. C'est un grand bonheur que de t'avoir cotoyé.

Enfin pour terminer je remercie en vrac toutes les personnes ci dessous ainsi que toutes ces petites choses qui me rendent heureux au quotidien : les vinyles et leurs craquements quand on les écoute, la musique en général, le vélo, la bière, Alex et ses cours de batterie (quand est-ce qu'il vient ce nouvel album de Tapetto Traci hein ?), tous les étudiants que j'ai eu en cours, internet et ses innombrables sources de distractions futiles (koreus, bashfr...), les jolies filles, la bière, les kinder surprise, les bons bouquins, Bob Dylan, la bière, les bons repas, le cri de Roger Daltrey sur *Won't get fooled again*, le foot, les mots croisés, les repas en famille, Pink Floyd, tous les camarades que j'ai croisé pendant mon cursus (parmi eux Aurélien, Paul, Céline, Thomas, Ronan etc), les White Stripes, le bon whisky, Walter, le big Lebowski, Plastikman, les voix de Björk et Joan Baez, la mystérieuse fille aux sourires, le Club, la bière, les pantalons blancs, les tunisiens qui sont passés au LIUPPA (Wadi, Jihène et Asma), la ville de Pau et évidemment un immense merci au Garage[2], tout son personnel aimable et chaleureux (ah Florence ! Ses russes blancs, ses crêmes brûlées noyées de whisky et son petit ventre), leurs burgers, la musique qu'ils passent et .... leurs bières !

Il y un proverbe qui dit que l'important ce n'est pas où on va, mais comment on y va. Je ne sais pas vers où je vais, mais ce qui est sûr, c'est que grâce à vous tous (et à tous ceux que j'ai oublié de mentionner, désolé), le chemin qui m'y rapproche aura été des plus agréables et heureux. Merci pour tout.

[2] 49 rue Emile Garet, 64000 Pau

"Quand on est intelligent, il est plus facile de faire l'imbécile que l'inverse."
Woody Allen

Table des matières

Partie I Introduction et état de l'art 1

Chapitre 1
Introduction

Chapitre 2
Etat de l'art

2.1 Introduction . . . 10
2.2 Rappels sur le calcul d'illumination . . . 10
2.2.1 Algorithmes de rendu . . . 10
2.2.2 Calcul d'éclairage local . . . 11
2.3 Placage de texture et paramétrisation . . . 13
2.3.1 Textures 2D . . . 13
2.3.1.1 Paramétrisation de surface . . . 14
2.3.2 Filtrage et Mip-Mapping . . . 16
2.3.3 Textures volumiques . . . 19
2.3.4 Octree-Textures . . . 20
2.3.4.1 Filtrage et MIP-mapping des octree-textures . . . 22
2.3.5 Octree-Textures sur GPU . . . 23
2.3.5.1 Filtrage des Octree-Textures GPU . . . 25
2.4 Conclusion . . . 26
2.5 Préservation de détails . . . 27
2.5.1 Bump Mapping et Normal Mapping . . . 28
2.5.1.1 Bump Mapping . . . 28

2.5.1.2 Normal Mapping 29
2.5.2 Préservation d'apparence sur maillages simplifiés 30
2.5.2.1 *A general method for preserving attribute values on simplified meshes* 30
2.5.2.2 *Appearance-preserving simplification* 31
2.5.2.3 *Rapid Visualization of Large Point-Based Surfaces* . 33
2.5.3 Autres méthodes de préservation de détails 35
2.5.4 Conclusion 38

Partie II Contribution 39

Chapitre 3
Construction des *Appearance Preserving Octree-Textures*

3.1 Introduction 42
3.2 Subdivision de l'Octree 44
3.2.1 Paramètres et structures de données 44
3.2.2 Initialisation de l'octree et subdivision 46
3.2.3 Subdivision adaptative 47
3.2.4 Critère d'erreur : métrique $L^{2,1}$ 49
3.2.4.1 Influence de la valeur de tolérance sur la subdivision 51
3.2.4.2 Gestion de l'écart entre m et M 54
3.3 Assignation des normales aux feuilles de l'octree 58
3.3.1 Echantillonnage des normales par lancer de rayons 58
3.3.1.1 Choix de l'origine et de la direction du rayon 59
3.3.1.2 Liste des nœuds intersectés par le rayon 61
3.3.1.3 Recherche de la plus proche intersection 63
3.3.2 Echantillonnage par moyennage des normales des feuilles . . . 64
3.3.2.1 Calcul des normales pour les nœuds internes 66
3.4 Encodage de l'APO en texture 2D 67
3.4.1 Organisation en largeur de l'octree 67
3.4.2 Encodage sans valeurs MIP-map 68
3.4.2.1 Quantification sur trois octets des normales 69
3.4.3 Encodage avec valeurs MIP-map 70

3.4.4 Sauvegarde sous forme de texture 2D 72
3.5 Performances et discussion 73
3.5.1 Temps de construction des APO 73
3.5.2 Occupation mémoire des APO 74
3.5.2.1 Taille des textures APO 75
3.5.2.2 Occupation en mémoire vive lors de la construction . 77

Chapitre 4
Placage des APO par le GPU

4.1 Introduction 80
4.2 Parcours GPU de l'octree 80
4.2.1 Calcul de l'index du nœud fils contenant le fragment 81
4.2.2 Détermination de l'offset vers le nœud fils 83
4.2.2.1 Offset vers le premier fils 84
4.2.2.2 Décalage vers le nœud fils souhaité dans la fratrie . . 84
4.2.2.2.1 Encodage sans valeurs MIP-map : 85
4.2.2.2.2 Encodage avec valeurs MIP-map : 86
4.2.3 Lecture du nœud dans la texture *APO* 88
4.2.4 Mise à jour des informations spatiales de la cellule 88
4.2.5 Condition d'arrêt de la boucle de parcours 89
4.2.6 Lecture et décodage de la normale du fragment 89
4.2.7 Boucle de parcours complète 90
4.3 Filtrage des APO 91
4.3.1 Interpolation bilinéaire 92
4.3.1.1 Nouvelle condition d'arrêt de la boucle 92
4.3.1.2 Résultats et performances 94
4.3.1.3 Approximation du filtrage 95
4.3.2 Rendu adaptatif 97
4.3.2.1 Calcul de la profondeur par utilisation des fonctions GLSL 98
4.3.2.2 Filtrage trilinéaire 100
4.4 Rendus et Performances 101
4.4.1 Qualité visuelle 101

4.4.2 Performances d'affichage . . . 104
4.4.2.1 Performances en gros plan . . . 104
4.4.2.2 Performances en plan large . . . 104
4.4.2.3 Performances avec filtrage bilinéaire . . . 105
4.4.3 Discussion . . . 106

Chapitre 5
Optimisation et Atlas de texture 2D

5.1 Optimisation des performances . . . 109
5.1.1 Détection du nœud enveloppant les triangles . . . 110
5.1.2 Forêt d'Octrees . . . 111
5.1.2.1 Construction de l'octree et espace mémoire occupé . . . 112
5.1.2.2 Calcul de l'indice du nœud de départ . . . 112
5.1.2.3 Performances . . . 113
5.1.3 Améliorations possibles . . . 114
5.2 Conversion en atlas de texture 2D . . . 115
5.2.1 Paramétrisation en atlas de triangles . . . 115
5.2.1.1 Nombre de texels pour un triangle . . . 116
5.2.1.2 Positionnement des triangles dans le plan . . . 116
5.2.2 Création de la texture d'atlas . . . 118
5.2.2.1 Rendu dans texture . . . 118
5.2.2.2 Résultats . . . 119
5.2.3 Rendu avec l'atlas APO . . . 121
5.3 Conclusion . . . 122

Partie III Application **125**

Chapitre 6
Application : transmission progressive de l'apparence dans un visualisateur web

6.1 Introduction . . . 127
6.2 Visualisation de maillages sur environnement réseau . . . 129
6.2.1 Transmission de LOD géométrique . . . 129

6.2.1.1 Maillages progressifs et compressés 129
6.2.1.2 Structure hiérarchique 131
6.2.2 Rendu à distance 132
6.2.3 Transmission d'apparence 133
6.3 Architecture APOWeb3DViewer 133
6.3.1 Mise à jour de la texture 136
6.3.2 Fragment shader adapté 138
6.4 Téléchargement adaptatif 138
6.5 Implémentation JOGL et résultats 140

Partie IV Conclusion 143

Chapitre 7
Conclusion

7.1 Synthèse 145
7.2 Perspectives 149

Partie V Annexes 153

Annexe

Annexe A
Glossaire et définition des termes et notations employées

A.1 Octree et subdivision 155
A.2 Encodage 2D de l'octree 156
A.3 Placage GPU de l'APO 156

Annexe B
Statistiques de subdivision

Annexe C
Codes GLSL

C.1 Vertex Shader 161

C.2 Calcul de l'éclairage 161
C.3 Parcours de l'*APO*, boucle Tant que 162
C.4 Fragment shader complet 163

Annexe D
Rendus APO

Bibliographie **177**

Première partie

Introduction et état de l'art

Chapitre 1

Introduction

Les jeux vidéos, les outils de conception, de modélisation et de simulation numérique et maintenant les environnements virtuels ou encore les bases de données d'objets 3D, toutes ces applications doivent manipuler, sauvegarder et surtout afficher des représentations numériques d'objets du réel. La représentation informatique classique de ces objets est le maillage polygonal, qui est composé d'un ensemble de triangles connectant les sommets situés sur la surface dudit objet. Pour afficher et offrir une visualisation de ces objets, ceux-ci sont plongés dans une scène dans laquelle le calcul de l'éclairage à la surface des maillages permet de mettre en évidence la forme des objets. Un triangle faisant face à la source lumineuse réfléchira fortement la lumière et apparaîtra très illuminé, tandis qu'un autre dont l'orientation ne permet pas de réfléchir la lumière apparaîtra plus sombre. C'est donc l'orientation des triangles, donnée par leurs normales, qui nous permet de percevoir la géométrie et les détails de relief à la surface des maillages polygonaux.

Pour augmenter le réalisme des objets, il faut augmenter le nombre de détails des surfaces, et donc le nombre de triangles qui composent les objets. Si la représentation d'objets simples, comme des sphères ou cubes ne nécessite qu'un nombre limité de poly-

Fig. 1.1 – Plus le nombre de polygones composant un maillage est élevé, plus le maillage comporte de détails. Ici, les objets sont composés respectivement de 3000, 20000 et 800000 triangles.

gones, la représentation d'objets complexes, ou la représentation d'aspérités à la surface des objets nécessitant elles un grand nombre de triangles. Or la capacité des cartes graphiques à traiter en temps réel ces nombreux triangles n'a pas augmenté aussi vite que la capacité des logiciels de modélisation, ou encore des outils d'acquisition tels que les scanners 3D, à créer des objets de plus en plus détaillés. Par exemple, dans le cadre du projet *Digital MichelAngelo*, dont le but était d'archiver numériquement des sculptures antiques, les maillages scannés pouvaient être composés de plusieurs centaines de millions de polygones. Sans atteindre de tels ordres de grandeur, il est fréquent que des objets détaillés soient composés de quelques millions de polygones. Si il est possible d'arriver à visualiser interactivement un maillage de cette taille, l'affichage de plusieurs objets détaillés, ou l'insertion d'un objet dans une scène complexe reste problématique.

L'objectif des applications 3D étant de proposer une manipulation interactive sans sacrifier la richesse de détails des objets, il faut utiliser des méthodes permettant d'enrichir visuellement des géométries relativement simples. Or la richesse visuelle d'une scène 3D est plus dépendante de l'apparence des objets, c'est à dire des propriétés des matériaux de leurs surfaces, que de leurs formes géométriques. Un maillage simple permet de deviner la forme et le volume global d'un objet, tandis que l'apparence de sa surface nous permet de percevoir son aspect et ses détails de reliefs par exemple. Cette apparence est caractérisée par la variation de certaines propriétés sur la surface, comme la couleur ou la variation des normales. Il est donc possible de dissocier la géométrie des objets et leurs apparences, la géométrie étant le maillage, l'apparence est quant à elle sauvegardée sous forme de texture, introduites par Catmull [Cat74] dans le but de stocker les détails de couleur d'un maillage dans une image 2D. Le nombre de polygones à traiter est ainsi diminué, l'interactivité est assurée et la richesse visuelle de détails conservée. Cependant il reste un problème : comment stocker les détails d'un objet 3D dans une texture 2D ? C'est à dire comment passer d'une représentation dans l'espace de l'objet à une représentation planaire, pour obtenir un *patron* de l'objet. Cette problématique est connue sous le nom de paramétrisation 2D des surfaces, dont le principe est illustré par la figure 1.2.

La paramétrisation des surfaces soulève plusieurs problèmes, notamment ceux liés à sa construction. S'il est en effet trivial de déterminer la paramétrisation des objets simples comme les cubes ou les sphères, il n'en est pas de même pour les objets complexes. La paramétrisation des surfaces nécessite donc l'intervention d'un utilisateur pour guider le découpage en morceaux de l'objet. Chaque morceau est projeté sur la surface, son représentant est appelé *patch* en anglais. Les méthodes automatiques de calcul de la paramétrisation sont encore lourdes et coûteuses. De plus, la paramétrisation des surfaces doit veiller à minimiser les distorsions qui peuvent apparaitre lors de la projection de la surface sur le plan texture. Lors du calcul, des coordonnées 2D de textures sont associées à chaque sommet du maillage, ces coordonnées permettant de faire le lien entre la postion 3D des sommets et leurs projections sur la texture.

Fig. 1.2 – Le maillage 3D *Bunny* et une paramétrisation de sa surface. Image extraite de [LPRM02].

L'utilisation de textures volumiques, dans lesquelles les détails de surface ne sont pas sauvegardés sur un plan mais écrits dans un volume entourant la surface, permet d'affranchir le texturage du problème de la paramétrisation. Une texture 3D est donc une grille régulière dans laquelle les maillages sont plongés, les détails de texture étant sauvegardés dans les cellules qui intersectent la surface du maillage. Ainsi lors du rendu, il n'y a pas besoin de coordonnées de textures, la position interpolée des sommets de triangles dans l'espace permet de trouver directement la cellule de la grille dans laquelle le détail associé est stocké. Quelle que soit la complexité topologique des maillages, l'utilisation de textures volumiques offre une possibilité de les habiller, mais à un coût de stockage énorme. En effet, la grille présente beaucoup plus de cellules n'intersectant pas les maillages que de cellules portant réellement une information utile. La grande majorité des cellules de la texture 3D n'est pas utilisée, représentant une pure perte de mémoire. Pour obtenir une texture détaillée, des textures de résolutions 512^3 représentent la résolution minimale, le volume de données pour les sauvegarder est donc trop grand par rapport à la quantité d'informations utiles qu'elles portent. En pratique, les textures volumiques sont donc le plus souvent utilisées sous forme procédurale, une fonction générant le détail en fonction de la position des sommets. Si cette forme procédurale rend possible la création de surface ayant l'apparence de bois ou de marbre, il est impossible par exemple d'encoder le champs de normales d'un maillage complexe pour les plaquer sur un maillage simplifié.

Récemment, conjointement dans les travaux de 2002 de Benson et Davis [BD02] et de DeBry et al. [gDGPR02], le concept d'octree-texture a été introduit. Il s'agit de textures volumiques, mais au lieu de reposer sur un découpage régulier de l'espace, celles-ci consistent en une subdivision adaptative du volume, subdivision basée sur le principe des octrees : un volume racine est découpé suivant ses trois plans médians, formant huit nœuds fils qui peuvent eux même être récursivement subdivisés. Cette subdivision adaptative permet de concentrer les détails dans les zones qui intersectent le maillage, et ainsi d'avoir les avantages d'une texture volumique sans surcharger inutilement l'espace

de stockage nécessaire. Si les octree-textures ont tout d'abord été implémentées sur le CPU, les travaux de [LHN05] et [LSK+06] ont rendu ces structures disponibles sur les cartes graphiques, car ces textures ne sont pas matériellement présentes sur les GPU comme le sont les textures 2D ou 3D. Le passage sur GPU se fait donc par un encodage spécial de la structure de l'octree, associé à un programme pour carte graphique, un *shader*, permettant de parcourir la structure arborescente de l'octree.

Si les octree-textures ont été présentées dans le cadre de travaux de peinture sur objet, nous proposons de les utiliser pour effectuer une préservation d'apparence lors de la simplification de maillages, c'est à dire que nous allons encoder dans une octree-texture les détails d'un maillage original pour les plaquer sur une version simplifiée lors du rendu. Nos travaux s'inspirent de ceux présentés dans [CMR+99,CMSR98] et de [COM98], dans lesquels des textures 2D étaient utilisées pour sauvegarder l'apparence de maillages. Nous nous focalisons dans cette thèse sur l'apparence du relief, c'est à dire les normales du maillage original, et non pas sur les couleurs. Nous effectuons ainsi du *normal mapping* avec les octree-textures, ce qui comparativement à l'utilisation de textures 2D, offre plusieurs avantages liés à leur nature volumique et hiérarchique :

- **Absence de paramétrisation :** La nature volumique de l'octree permet de s'affranchir du calcul d'une paramétrisation de la surface des maillages, étape qui était présente dans les travaux cités ci-dessus, sous la forme d'une paramétrisation par morceaux ou par la génération d'un atlas de triangles.
- **Echantillonnage adaptatif :** Les subdivisions récursives des nœuds de l'octree permettent d'adapater le niveau de détail, et donc la taille des nœuds en fonction de la variation de détail à la surface du maillage original. Une zone présentant de fortes variations sera couverte par de nombreux nœuds de faible largeur, tandis qu'une zone ne présentant pas de variation sera couverte par des nœuds plus larges. On évite ainsi le sur et le sous échantillonnage inhérent à l'utilisation d'une taille de texel[3] fixe. Les octree-textures générées ont ainsi une taille plus compacte par rapport à une texture 2D ou 3D encodant les mêmes détails.
- **Rendu adaptatif :** La hiérarchie de l'octree permet d'encoder un détail aux feuilles de l'octree mais aussi à chaque nœud interne, formant ainsi une hiérarchie de niveaux de détails. Les détails encodés dans les feuilles sont les niveaux de détails les plus fins, les détails des nœuds internes sont les niveaux de détail intermédiaires. On peut faire une analogie entre cette hiérarchie et le mip-mapping des textures 2D, qui génére une *pyramide* de textures à différentes résolutions, la texture de résolution adaptée étant choisie au moment du rendu en fonction de la distance au point de vue, pour éviter l'aliasing, qui survient lorsque plusieurs détails devant se projeter sur le même pixel ne sont pas filtrés.

L'utilisation de l'octree comme structure de sauvegarde de l'apparence des maillages

[3]Element atomique d'une texture

nous permet donc de proposer une création de texture d'apparence **totalement automatique**, les différentes subdivisions étant pilotées par la **variation de normales** à l'intérieur des nœuds de l'octree. Nous proposons un **encodage plus compact** des octree-textures que ceux présentés précédemment, et nous montrons que cet encodage peut être utilisé à profit dans le cadre d'une application de visualisation distante en offrant la possibilité d'un **téléchargement progressif** des données de détails. L'utilisation de notre octree-texture de détails permet une **manipulation interactive** en plaquant sur des objets simplifiés une apparence de maillages originaux plus lourds. Les maillages ainsi manipulés ont l'**apparence des maillages originaux** et offrent la richesse de détails fournis par la géométrie complexe des versions originales.

Le mémoire est organisé comme suit : la première partie présente les travaux existants sur les domaines sur lesquels reposent nos travaux : texturage 2D et 3D, préservation de détails et structure hiérarchique pour les textures. La partie suivante présente la construction de nos octree-textures de détails, ainsi que leur utilisation lors du rendu, c'est à dire le programme à exécuter sur la carte graphique pour exploiter les détails contenus dans la texture. La troisième partie présente l'utilisation de la structure d'octree-texture de détails dans le cadre d'une application répartie de visualisation de maillages 3D.

Chapitre 2

Etat de l'art

Sommaire

2.1 Introduction **10**
2.2 Rappels sur le calcul d'illumination **10**
2.2.1 Algorithmes de rendu 10
2.2.2 Calcul d'éclairage local 11
2.3 Placage de texture et paramétrisation **13**
2.3.1 Textures 2D 13
2.3.1.1 Paramétrisation de surface 14
2.3.2 Filtrage et Mip-Mapping 16
2.3.3 Textures volumiques 19
2.3.4 Octree-Textures 20
2.3.4.1 Filtrage et MIP-mapping des octree-textures . . 22
2.3.5 Octree-Textures sur GPU 23
2.3.5.1 Filtrage des Octree-Textures GPU 25
2.4 Conclusion **26**
2.5 Préservation de détails **27**
2.5.1 Bump Mapping et Normal Mapping 28
2.5.1.1 Bump Mapping 28
2.5.1.2 Normal Mapping 29
2.5.2 Préservation d'apparence sur maillages simplifiés 30
2.5.2.1 *A general method for preserving attribute values on simplified meshes* 30
2.5.2.2 *Appearance-preserving simplification* 31
2.5.2.3 *Rapid Visualization of Large Point-Based Surfaces* 33
2.5.3 Autres méthodes de préservation de détails 35
2.5.4 Conclusion 38

2.1 Introduction

L'habillage de surfaces par des textures a depuis longtemps été utilisé pour enrichir de détails les géométries 3D des maillages polygonaux présents dans les scènes. Les textures peuvent porter les détails de couleur d'un objet, mais également des informations de normales qui permettent de rajouter des détails de reliefs. Dans ce chapitre d'état de l'art nous allons présenter les principes généraux du plaquage de textures ainsi que du *normal mapping*. Mais dans un premier temps nous faisons de brefs rappels sur le calcul de l'illumination lors du rendu des scènes 3D à l'écran.

Nous terminerons ce chapitre d'état de l'art en présentant plus particulièrement les travaux de préservation d'apparence, dont le but n'est pas uniquement d'enrichir visuellement les maillages polygonaux, mais plus précisément de sauvegarder dans une texture les informations de l'apparence de l'objet pour les plaquer sur une version basse résolution de l'objet. Cette sauvegarde d'apparence est le cœur de nos travaux, notre approche est détaillée dans la suite du document.

2.2 Rappels sur le calcul d'illumination

2.2.1 Algorithmes de rendu

Pour calculer et afficher les images de synthèse des scènes représentées numériquement, plusieurs algorithmes doivent être mis en place. Cette liste d'opérations qui permet de passer de la représentation informatique de la position et des propriétés de matériau des objets, des sources lumineuses et d'un point de vue donné à une image 2D composé de pixels s'appelle l'opération de rendu. Dans cette série d'opérations nous nous intéressons plus particulièrement aux algorithmes d'éclairages, qui calculent la couleur de chacun des pixels de l'image. Plusieurs familles d'algorithmes d'éclairage existent, qui varient en termes de réalisme et de complexité calculatoire. Leurs buts est de calculer la manière dont la lumière se propage à l'intérieur des scènes, et comment elle se reflète à la surface des objets.

Les familles d'algorithmes réalistes, basés sur des algorithmes de lancer de rayons [Gla89], déterminent le cheminement des rayons lumineux vers l'écran en remontant depuis le point de vue vers les sources lumineuses. À chaque intersection avec un objet de la scène, des rayons incidents calculés en fonction de la normale de la surface sont lancés vers les sources lumineuses pour savoir si elles contribuent à éclairer l'intersection trouvée. Une fois les conditions d'éclairage déterminée, le calcul de l'illumination, et donc la couleur du pixel, est effectué. Ces algorithmes sont réalistes dans le sens où ils prennent en compte les multiples reflets de la lumière, le parcours des rayons à travers la scène, c'est à dire qu'ils simulent l'optique réelle en tenant compte des effets indirects de la lumière, et calculant les échanges lumineux entre les différents objets de la scène.

Malgré des efforts pour accélérer le calcul de l'éclairage réaliste des scènes 3D, le rendu interactif par lancer de rayons reste problématique, les tests d'intersections nécessaires pour le calcul de l'illumination des objets étant très grand. Il existe une autre famille d'algorithmes dans laquelle, contrairement au lancer de rayon où les couleurs des pixels étaient déterminées depuis l'écran par le calcul de l'illumination de la plus proche intersection entre chaque rayon et les objets de la scène, les objets sont projetés sur l'écran. L'algorithme du *Z-buffer*, introduit par Catmull [Cat74], permet de sélectionner pour chaque pixel de l'écran la couleur de l'objet le plus proche, en sauvegardant dans un tampon la profondeur associée à chaque pixel. Si un fragment projeté sur un pixel a une profondeur plus faible que celle déjà présente, alors le nouveau fragment est conservé, il est rejeté sinon. À la fin de cette étape de calcul de profondeur, la couleur du fragment est calculée en fonction des attributs qui lui sont associés : couleur, normale etc. Ces attributs sont obtenus soit par interpolation linéaire des valeurs de ces mêmes attributs aux sommets, soit par la lecture de ces attributs dans une texture.

Cet algorithme de rendu ne permet pas de prendre en compte l'environnement extérieur de chaque fragment lors du calcul de l'éclairage, car seules les informations locales à chaque fragment sont disponibles. Cependant cet algorithme du *Z-buffer* est beaucoup plus rapide et a donc été retenu par les constructeurs pour être implémenté matériellement sur les cartes graphiques.

2.2.2 Calcul d'éclairage local

Le calcul de l'éclairage local consiste à déterminer pour un point donné de la surface et des conditions d'éclairage définies la manière dont la lumière réagit avec la surface, c'est à dire comment elle va être réfléchie par la surface. Ce calcul produit donc en sortie la couleur de lumière renvoyée, celle qui sera affectée au pixel sur lequel se projette le fragment en cours. On peut noter que les modèles d'illumination locale peuvent être calculés de deux façons différentes : l'éclairage par sommet ou l'éclairage par pixel. L'éclairage par sommet consiste à calculer une couleur pour chaque sommet d'un triangle, puis lors de la projection du triangle sur les pixels de l'écran, opération dite de *rasterisation*, à interpoler ces couleurs aux sommets pour chaque pixel à l'intérieur du triangle. Ce modèle est rapide mais peu précis, car si un reflet spéculaire se trouve au centre d'un triangle par exemple, il ne sera pas visible à l'écran. L'éclairage par pixel, consiste à interpoler pour chaque pixel la normale associée, et calculer l'éclairage avec cette normale. Cet éclairage est plus coûteux mais plus précis.

Le modèle d'illumination locale le plus courant est le modèle développé par Phong [Pho75], puis étendu par Blinn [Bli77]. C'est un modèle empirique que Phong a construit selon ses observations lors de ses expérimentations. Bien que non basé sur des fondements physiques ou optiques, ce modèle a été choisi par les API *OpenGL* ou *Direct3D* comme le modèle d'illumination par défaut dans les pipelines graphiques. Ce modèle étend celui de Lambert de réflexion diffuse en ajoutant une composante spéculaire qui représente les reflets brillants des objets, et une composante de lumière ambiante. La lumière réfléchie

par un point de la surface est donc constituée de trois composantes :

- **Composante ambiante :** La composante ambiante d'une scène correspond à la somme des interactions dites inter-diffuses entre les objets, c'est à dire qu'elle représente l'éclairage diffus reçu par réflexion entre les différents objets. Les algorithmes d'éclairage global permettent de calculer précisément cette composante, mais dans les applications temps réels elle est simplifiée et remplacée par une constante pour toute la scène, constante indépendante des sources lumineuses.
- **Composante diffuse :** C'est la lumière qui est réfléchie dans toutes les directions par la surface. Cette réflexion modélise le comportement dit lambertien des surfaces, du modèle du même nom. Ce modèle présente les surfaces comme réfléchissant la lumière de manière isotropique, c'est à dire que quel que soit l'angle d'observation de la surface, la luminance sera identique. La quantité de lumière réfléchie est dépendante de l'angle d'incidence du rayon lumineux avec la normale de la surface. Plus l'angle est faible, c'est à dire que la surface fait *face* à la source lumineuse, plus la quantité de lumière réfléchie est forte.
- **Composante spéculaire :** Pour représenter les reflets brillants des matériaux, comme ceux des métaux par exemple, Phong a ajouté la composante spéculaire dans son modèle d'illumination. Contrairement à la réflexion diffuse, la réflexion est anisotropique et a donc une direction privilégiée. La direction de cette réflexion est celle qu'a la lumière réfléchie sur un miroir parfait, c'est à dire que le rayon est réfléchie en formant avec la normale à la surface le même angle que le rayon incident. Pour observer cette réflexion il faut donc que le point de vue soit situé proche de cette direction. Plus l'angle entre la direction de vue et la direction spéculaire est grand, moins la composante spéculaire est importante.

La figure 2.1 illustre la combinaison de ces trois composantes pour la calcul final de l'illumination locale. En modifiant les coefficients associés aux trois composantes, il est possible de simuler plusieurs types de surfaces, plus ou moins brillants ou mats. Ce modèle simpliste a été amélioré pour le rendre plus réaliste, mais il n'en reste pas moins le modèle de référence intégré dans toutes les cartes graphiques.

Le calcul de l'éclairage en un point est dépendant de l'orientation locale de la surface, l'angle qu'elle forme avec la direction du rayon détermine la quantité de lumière réfléchie ainsi que la direction de la réflexion spéculaire. La normale de la surface influe donc directement sur la perception du relief des objets, en modifiant cette normale la perception du relief sera elle aussi modifiée. Cette propriété est l'idée fondamentale du *bump mapping* et du *normal mapping*, techniques dans lesquelles la normale est modifiée dans le but de donner l'illusion d'un relief sur des surfaces lisses, l'information de modification des normales pouvant être encodée dans des textures.

Avant de détailler le fonctionnement des méthodes de *bump mapping* et les méthodes

Fig. 2.1 – Décomposition du modèle d'illumination de Phong. De gauche à droite : éclairage ambiant, diffus et spéculaire. La capture de droite montre le résultat final.

de préservation de détails qui en ont découlées, nous décrivons dans la section suivante les principes du plaquage de texture.

2.3 Placage de texture et paramétrisation

2.3.1 Textures 2D

Le placage de texture pour enrichir les scènes de synthèse a été introduit par Catmull en 1974 [Cat74]. Dans ces travaux, seuls des détails de couleurs étaient sauvegardés dans les textures. Mais quelles que soient l'utilisation des textures, pour encoder des couleurs ou comme nous le verrons plus tard pour stocker les normales le long de la surface, la problématique principale de l'utilisation de texture est la paramétrisation de la surface, c'est à dire la relation entre les éléments de texture définis dans l'*espace texture* à deux dimensions et la surface définie dans un espace à trois dimensions. Le but est d'associer un couple de coordonnées 2D, généralement noté (u, v) à chacun des sommets du maillage polygonal. Lors du rendu et de la projection des fragments à l'écran, les coordonnées de textures sont interpolées, en prenant en compte la projection perspective, de façon à aller chercher le bon élément de texture, ou texel, à associer au fragment. La figure 2.2 montre un exemple de placage de texture sur une sphère.

Il existe quelques cas triviaux pour lesquels il est aisé de trouver une paramétrisation de la surface. Les formes géométriques simples, comme les sphères, cubes, cylindres cônes etc sont facilement développables sur des surfaces 2D. Malheureusement, la plupart des objets manipulés ne sont pas des formes, ni même des compositions de ces formes simples, mais des surfaces quelconques. La mise en correspondance de ces surfaces quelconques avec une texture 2D a motivé et motive encore de nombreux travaux de recherche. Nous n'allons pas dans le cadre de ce manuscrit détailler précisément les algorithmes de

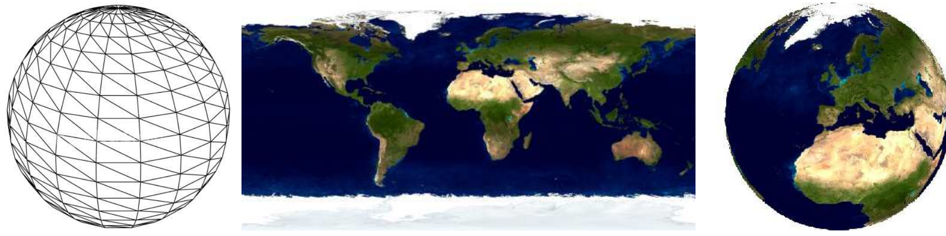


Fig. 2.2 – Placage de texture sur une sphère.

paramétrisation, mais décrire les problématiques et limites liées à cette opérations. Les algorithmes de paramétrisation sont détaillés par exemple dans l'état de l'art [FH05].

2.3.1.1 Paramétrisation de surface

Le but de la paramétrisation d'une surface est d'arriver à déplier celle-ci sur un plan, appelé l'espace texture. Ce dépliage, qui revient à obtenir un patron de la surface, introduit des distorsions qui nuisent à la qualité de la texture. Ces distorsions peuvent être de deux natures. Les distorsions angulaires sont observées quand les angles formés par les arêtes des triangles ne sont pas conservés par le dépliage sur le plan, les distorsions métriques sont observées quand les longureurs de ces mêmes arêtes ne sont pas conservées. Les distorsions ne sont pas souhaitables car le pas d'échantillonnage de la texture reste fixe, et donc des zones de la surface seront soit sous-échantillonné, soit sur-échantillonné.

Il est donc nécessaire de minimiser les effets de la distorsion lors de la paramétrisation des surfaces quelconques dans l'espace texture. Les surfaces quelconques, contrairement aux cônes ou aux cylindres, ne sont pas développables, et ne peuvent être aplanies qu'en introduisant des coupures dans la surface. Le positionnement des ces découpes est problématique, et si certains travaux permettent de les positionner automatiquement [SH02], l'intervention d'un utilisateur est souvent nécessaire pour positionner ces coupures sur les surfaces. Le choix des bordures doit permettre de minimiser les distortions lors de la projection de la surface sur l'espace texture, mais également ne pas faire apparaître d'artefacts visuels trop prononcés. En effet la découpe de la surface introduit obligatoirement des discontinuités. La figure 2.3 suivante illustre la découpe et la paramétrisation d'une surface.

Une autre contrainte sur les surfaces existe pour qu'elles puissent être dépliées sur un plan : elles doivent être homéomorphes à un disque. Or la plupart des surfaces ne présente pas cette propriété géométrique. La paramétrisation de telles surfaces implique donc une étape supplémentaire. Ces dernières doivent être découpées en morceaux, chacun des morceaux étant lui homéomorphe à un disque. Chaque morceau est ensuite paramétrisé indépendamment sur l'espace texture (voir figure 2.4). La texture ainsi créée est appelée un atlas de texture. Pour la création des atlas comme pour une paramétrisation globale

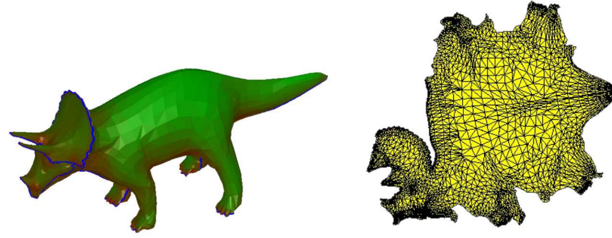


Fig. 2.3 – Paramétrisation globale d'un maillage 3D. Image extraite de [SH02].

Fig. 2.4 – Paramétrisation en atlas de texture d'un maillage 3D.

des surfaces, le problème est de trouver un découpage minimisant les discontinuités et distortions. Cette problématique peut-être traitée de manière automatique [LPRM02], ou semi-automatique, le guidage d'un utilisateur permettant d'outrepasser les limitations des méthodes automatiques.

La création des atlas de texture pose un autre problème : comment placer les différents morceaux sur la texture en minimisant l'espace non utilisé ? C'est à dire comment agencer au mieux les différents morceaux en les serrant au maximum. On peut voir ce problème comme la minimisation des chutes de tissu lors du dessin du patron d'un vêtement (voir [Lef05]). Ce problème d'empaquetage est malheureusement NP-complet, et quelque soit l'agencement des morceaux dans la texture, qui est généralement rectangle ou carrée, il y a toujours de l'espace perdu. Nous pouvons noter qu'avec l'utilisation d'une paramétrisation globale, il n'y a qu'un seul morceau mais qui sera rarement de forme rectangulaire, il y a donc également de l'espace texture non utilisé autour de la surface développée.

La notion d'atlas de texture peut être poussée à l'extrême, en utilisant le triangle comme unité atomique de l'atlas. On a alors un atlas où chaque patch est un triangle de la surface. Les problèmes de continuité sont alors multipliés, les triangles ne présentant pas de cohérence spatiale dans l'atlas de polygones. Si le problème de paramétrisation est amoindri, du fait que le découpage est automatique à la frontière de chaque triangle,

le problème du rangement des triangles dans la texture est lui toujours difficilement soluble. La figure 2.5 montre un exemple d'atlas de triangle.

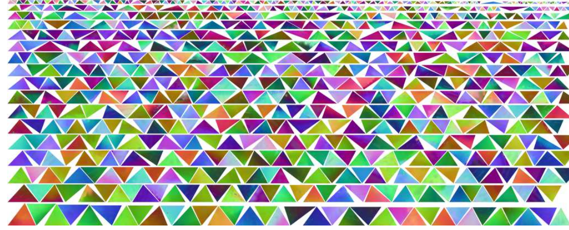


Fig. 2.5 – Exemple d'atlas de triangle.

2.3.2 Filtrage et Mip-Mapping

Le placage de textures fait apparaitre des artefacts visuels à la surface des objets. On peut distinguer deux types d'artefacts visibles selon l'éloignement de l'objet. Quand l'objet sur lequel s'applique la texture est proche, un effet de pixellisation est visible. Cet effet est lié à la nature discrète de l'image texture, et au fait que lorsque l'on est en gros plan, le niveau de détail de la texture peut ne pas être assez élevé par rapport à la résolution de la scène : un même texel se projette sur plusieurs pixels. On parle alors de *magnification* (agrandissement en français). Pour éviter cet effet de pixellisation, il faut reconstruire un signal continu à partir des éléments de la texture, en interpolant la valeur à l'intérieur du voisinage carré formé par les quatre texels entourant le fragment. Ce filtrage et ses résultats est illustré par la figure 2.6. Il faut dans ce cas considérer les texels comme des points et non pas comme des éléments de surface de la texture. D'autres schémas d'interpolation plus complexes, faisant intervenir plus de 4 texels, existent, mais sont plus lents que le filtrage bilinéaire simple que nous avons sommairement présenté.

Un autre effet d'aliasing se produit lorsque le niveau de détails d'une texture est plus grand que ce que peuvent représenter les pixels de l'image rendue. Les pixels de l'image, projetés sur la scène qu'ils représentent ont une surface non nulles qui peut abriter plusieurs détails de couleurs ou normales (voir figure 2.7). En quelque sorte, plusieurs texels se projettent sur le même pixel de la scène. On parle dans ce cas de *minification* (réduction en français). Or, une seule information est inscrite sur ce pixel. Il y a donc une perte d'information qui peut se traduire par des artefacts visuels. Concrètement, la fréquence de la texture est plus grande que la fréquence d'échantillonnage des pixels de l'écran, et la texture n'est donc pas suffisament échantillonnée, produisant un signal non continu à l'écran (voir la figure 2.7). Pour remédier à ce problème il faut donc filtrer le signal de la texture pour le faire correspondre à la fréquence de l'écran.

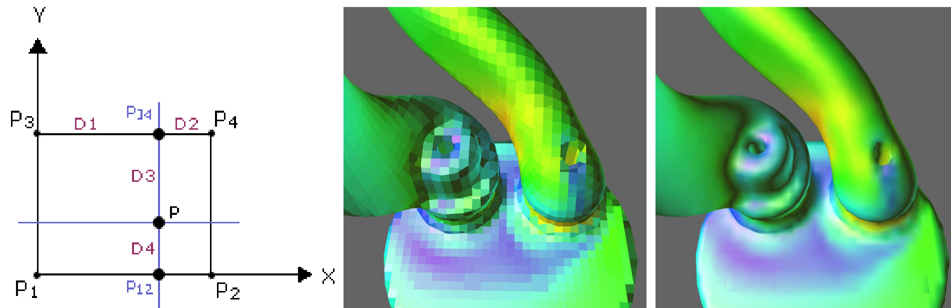


FIG. 2.6 – Interpolation linéaire d'une texture. Si la texture est accédée à la position P, la valeur sera interpolée entre les quatre texels notés P1 à P4 en pondérant selon les distances notées D1 à D4. Un signal continu est ainsi reconstruit. Les deux captures de droite montrent un placage de texture sans et avec l'utilisation du filtrage.

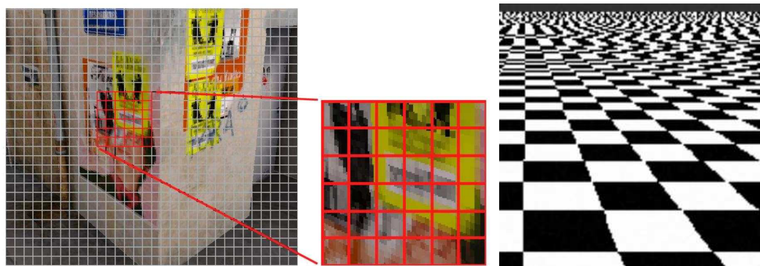


FIG. 2.7 – Les pixels de l'écran, représentés sur les grilles de gauche, contiennent plusieurs détails de couleur (image extraite de [Lef05]). L'image de droite présente un aliasing classique lors de l'application d'un motif de damier.

Fig. 2.8 – Exemple de MIP-map de textures. Chaque texel d'une texture de résolution inférieure est calculé en moyennant les quatre texels de la texture précédente. La texture originale forme la base de la hiérarchie, chaque texture inférieure a sa largeur divisée par deux jusqu'à atteindre la largeur d'un texel, formant le sommet de la hiérarchie, d'où l'image de *pyramide* utilisée.

Une solution est de calculer l'empreinte que fait le pixel rétro projectivement parlant sur la texture, puis de moyenner les valeurs à l'intérieur de cette empreinte. Cependant, le calcul de la forme du pixel rétro-projeté sur la surface est coûteux, et il faut parfois utiliser des formes prédéfinies pour accélérer ce calcul. Une autre solution a été présentée par Williams [Wil83], le *MIP-Mapping*. Le principe est de calculer une pyramide de textures (voir figure 2.8) à différents niveaux de résolution à partir de la texture originale. Chaque texel d'une texture contient la valeur moyennée des quatre texels correspondants de la texture de résolution directement supérieure, créant ainsi un lissage de la texture de plus haute résolution. L'ajout de cette hiérarchie de textures de résolution inférieure augmente la taille des textures d'environ 33%.

Lors du rendu, le niveau de détail adapté est sélectionné en fonction de la distance. Des travaux comme ceux de [EWWL98] traitent du calcul de niveau de MIP-map à appliquer. Le niveau de détail idéal se trouvant souvent entre deux textures de résolutions différentes, la valeur du texel est calculée en interpolant entre les valeurs préliminairement interpolée sur chacune des textures des deus résolutions encadrant le texel. On parle alors d'interpolation *tri-linéaire*. L'utilisation de cette technique permet d'éviter les problèmes d'aliasing, d'autant plus qu'il est possible de mettre en place des calculs plus précis qu'un simple moyennage pour la détermination des valeurs des textures de la hiérarchie. Citons par exemple [Tok05] qui permet de calculer spécifiquement le MIP-map pour les cartes de normales.

2.3.3 Textures volumiques

Une autre approche de texturage existe, permettant de s'affranchir de la nécessité d'une paramétrisation de la surface. Cette approche est celle des textures dites volumiques, qui ont été introduties par Perlin [Per85] et Peachey [Pea85]. Dans cette approche, l'apparence n'est pas encodée dans une image 2D, mais dans un volume enveloppant la surface à habiller. L'apparence étant définie en tout point à l'intérieur de ce volume, il suffit de lire les données à appliquer en fonction de la position des sommets de la surface. L'utilisation de textures volumiques évite tous les écueils propres aux textures 2D. Il n'y a pas de paramétrisation à calculer, pas plus que de problèmes de distorsion ou de continuité. L'objet peut être vu comme étant sculpté, ou extrait du volume d'apparence (voir figure 2.9).

Fig. 2.9 – Une théière habillée par une texture volumique.

Cependant les textures volumiques ne sont pas la panacée. Plus encore que pour les textures 2D, le problème majeur de leur utilisation est le gaspillage d'espace mémoire. En effet, pour sauvegarder une texture volumique, la solution classique est de découper régulièrement selon les trois axes le volume entourant la surface, formant ainsi une grille régulière de cellules. Seulement, il n'y a qu'un très faible pourcentage de cellules qui intersectent la surface de l'objet. Toutes les autres cellules peuvent être considérées comme une pure perte d'espace mémoire.

Les textures procédurales, c'est à dire des fonctions qui calculent l'apparence de la surface en fonction de la position 3D, permettent de générer des textures volumiques avec un coût de stockage quasiment nul. En effet, seul un algorithme et ses paramètres de contrôle sont nécessaires pour générer à la volée l'apparence de chaque fragment de la surface. Mais si les textures volumiques procédurales sont bien adpatées pour représenter des matérieux comme le bois ou le marbre, comme illustré sur la figure 2.10, il n'est pas possible d'étendre cette méthode à tous les types de matériau, et encore moins des surfaces d'apparence quelconque. Nous présentons dans la section suivante une alternative pour sauvegarder une texture volumique en minimisant l'espace mémoire requis pour la sauvegarde de la grille.

FIG. 2.10 – Le *Stanford Dragon* habillé par différentes textures volumiques procédurales, lui conférant des apparences de bois ou de roches. Images extraites de [KFCO+07].

2.3.4 Octree-Textures

Pour éviter le gaspillage d'espace mémoire d'une texture 3D classique, Debry et al. [gDGPR02] ainsi que Benson et al. [BD02] ont conjointement proposé l'utilisation d'une structure hiérarchique 3D pour stocker l'apparence de la surface. La structure utilisée est un *octree*, qui est une structure de données arborescente dont chaque nœud peut avoir au maximum huit fils. Utilisé comme structure de subdivision spatiale, l'octree permet de découper récursivement chacun de ses fils selon les axes médians de ces derniers. Un octree est simplement l'extension à la troisième dimension du *quadtree*. Le schéma de la figure 2.11 montre un exemple d'octree [4]. Lors du rendu, la position (x, y, z) d'un fragment permet, à l'instar d'une texture volumique classique, de déterminer la donnée à associer au dit fragment. Seulement, cette donnée n'est pas accessible directement, un parcours de l'octree de la racine jusqu'à la feuille contenant le fragment est nécessaire.

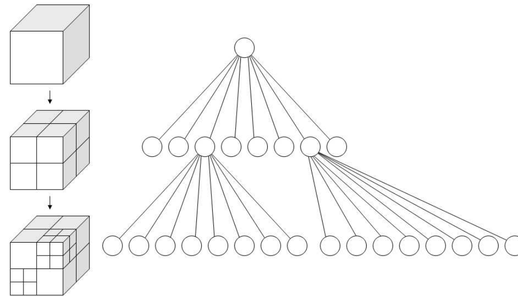


FIG. 2.11 – Exemple d'octree. La racine, ainsi que chacun des nœuds peut être décomposé en au maximum 8 fils lors du partitionnement de l'espace selon les trois axes du repère.

La structure de données utilisée par [BD02] pour représenter les nœuds l'octree consiste en un ensemble de marqueurs pour indiquer quels sont les fils du nœud existants,

[4]schéma issu de Wikipedia http://en.wikipedia.org/wiki/Octree

et de deux tableaux. Un premier tableau de pointeurs vers les nœuds fils, un deuxième tableau de valeurs dans le cas où les nœuds fils sont des feuilles de l'octree. L'utilisation d'un tableau de valeurs pour les fils qui sont des feuilles évitent le stockage de pointeurs vers ses feuilles, et économise donc un précieux espace mémoire. En effet, le stockage d'un octree complet est très coûteux. Pour la sauvegarde sur disque de ces octree-textures, les nœuds sont triés en largeur d'abord, ordre ne nécessitant qu'un pointeur d'un nœud vers ses fils étant donnés qu'ils sont tous contigus avec ce tri. Pour le rendu, effectué sur le CPU, la structure de l'octree est reconstruite, les pointeurs peuvent être déterminés en fonction du pointeur stocké sur disque et de l'ensemble des marqueurs. Ainsi, grâce à l'utilisation du tri en largeur d'abord, le stockage sur disque des octree-textures est le plus compact possible, sans perte de données pour l'exploitation lors du rendu. Les auteurs notent que l'utilisation de l'ordre en profondeur d'abord présentée dans [TS84] peut offrir une plus grande compression pour le stockage, mais que pour le rendu, l'ordre en largeur d'abord reste plus efficace.

Dans [gDGPR02], les auteurs font une estimation du poids supplémentaire qu'implique la sauvegarde de la structure hiérarchique de l'octree comparativement à une texture 2D classique. En effet, pour une texture de même résolution, il y aura autant de feuilles dans l'octree que de texels nécessaires pour encoder tous les détails de la texture. Il faut rajouter à ces feuilles la structure des parents des feuilles. Le poids rajouté est estimé à environ un tiers du poids de la texture. Ceci s'explique par le fait que la surface est localement planaire, et n'intersecte qu'une partie des fils[5]. En général on constate que la surface intersecte en moyenne quatre fils d'un nœud. Ce surpoids est estimé au maximum à 33%, ce qui correspond au poids rajouté par l'utilisation de MIP-mapping pour une texture 2D, or la structure hiérarchique peut elle aussi être vue comme une hiérarchie de niveaux de détails comme on le verra dans la suite.

Le surpoids engendré par la structure de l'octree peut être largement compensé, voire totalement effacé, de plusieurs manières. En effet l'estimation du poids de la structure faite par les auteurs de [gDGPR02] se base sur un nombre d'éléments de texture nécessaires pour sauvegarder l'apparence d'un objet. Or, comme nous l'avons vu précédemment, la paramétrisation d'un objet quelconque sur le plan texture, il y un espace mémoire non négligeable perdu autour des patches de texture. Ceci est dû à la difficulté d'agencer parfaitement les patches entre eux dans une texture rectangulaire (voir section 2.3.1.1 page 14). Une autre source de perte d'espace dans une texture 2D est le suréchantillonnage impliqué par une texture 2D. En effet, le niveau de détails est rarement fixe le long d'une surface, pourtant, pour pouvoir capturer toutes les variations haute fréquence de l'apparence sur la surface, il faut utiliser une discrétisation fine de l'espace, discrétisation qui s'applique également aux zones ne présentant pas de variation de détails. Ces zones sont sur-échantillonnées. Avec une octree-texture au contraire, les zones à haute fréquence peuvent être couvertes par des nœuds fins, tandis que les zones sans détails peuvent être couvertes par des nœuds de profondeur moins grandes. Les

[5] Les nœuds n'intersectant pas le maillage ne sont pas conservés

auteurs notent qu'avec cette possibilité d'échantillonnage adaptatif, une octree-texture ne pèse pas plus lourd qu'une texture 2D classique, et que l'augmentation de détails fait croître la consommation mémoire des textures de manière égale pour les textures 2D ou octree-texture, ce qui n'est pas le cas avec une texture volumique, où la taille explose du fait de la discrétisation régulière de la grille.

2.3.4.1 Filtrage et MIP-mapping des octree-textures

Pour que le placage des octree-textures sur un objet soit d'aussi bonne qualité que le placage de texture 2D, les opérations de filtrage et d'interpolation doivent être définies avec ces textures particulières, de manière à éviter les problèmes d'aliasing éventuels.

Pour l'interpolation des textures, qui permet d'éviter le crénelage apparent (voir figure 2.3.2), il faut utiliser un voisinage volumique (3x3x3 ou 5x5x5). Comme les nœuds de l'octree ne sont pas tous à la même profondeur, la méthode présentée dans [BD02] consiste à subdiviser le voisinage jusqu'à obtenir des cellules de même dimension, puis à interpoler à l'intérieur de ce voisinage. Ce principe est illustré par la figure 2.12. Lors de la subdivision, un soin particulier est apporté pour l'affectation des données aux nœuds vides auparavant. Celles-ci sont issues de moyennes pondérées des données des nœuds voisins existants.

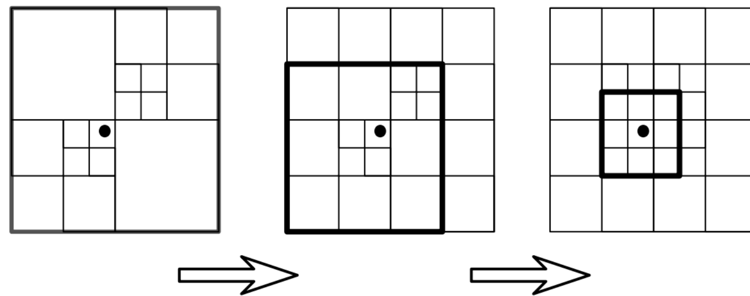


Fig. 2.12 – Subdivision du voisinage d'un nœud dans le but d'interpoler la texture hiérarchique.

Concernant le MIP-mapping, les deux articles [BD02, gDGPR02] proposent naturellement d'utiliser la structure hiérarchique de l'octree pour sauvegarder l'apparence à différents niveaux de détails. La méthode évidente pour calculer les donnéees à l'intérieur des nœuds est de calculer une valeur moyenne des valeurs des nœuds fils. Lors du rendu, on peut choisir d'utiliser les données contenues dans les nœuds internes plutôt que dans les feuilles, en évitant ainsi l'aliasing.

Seulement, du fait de la nature volumique de l'octree, il est des cas particuliers pour lesquels le moyennage simple provoque des erreurs. Dans le cas d'une surface très fine,

ayant des couleurs différentes de chaque côté, le moyennage pour les nœuds qui peuvent abriter les deux faces différentes mélange les couleurs alors qu'elles ne devraient pas l'être. Ces cas peuvent également se produire dans le cas général, pas uniquement dans le cas du MIP-mapping. La solution adoptée par les deux travaux est d'associer à ces nœuds posant problème des marqueurs de normales, et une couleur à chaque direction de normale. Ainsi lors du rendu d'un fragment se trouvant dans un nœud, la couleur est choisie en fonction de la direction du point de vue. Ce principe est illustré par la figure 2.13.

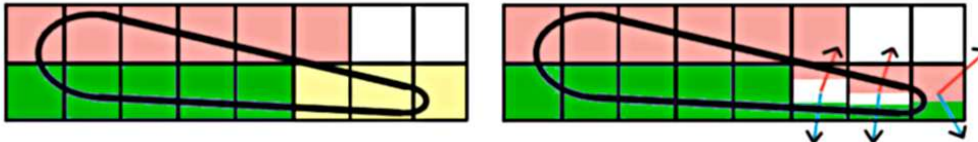


Fig. 2.13 – Dans le cas où plusieurs couleurs se mélangent dans un nœud (schéma de gauche), faisant apparaitre des couleurs n'existant pas dans l'apparence de l'objet, il est possible d'associer plusieurs données au même nœud, données dépendantes du point de vue (schéma de droite).

2.3.5 Octree-Textures sur GPU

Les octree-textures présentées dans [gDGPR02, BD02] ne sont pas supportées nativement par les cartes graphiques comme le sont les textures 2D ou 3D. Leur utilisation passe donc par une implémentation CPU qui ne permet pas un rendu interactif, le temps de calcul d'une image pouvant dépasser quelques secondes. La seule manière de rendre interactive les scènes utilisant les octree-textures était de les convertir en texture 2D, en perdant les avantages de la nature volumique des octree-textures.

Des travaux plus récents, réalisés par Lefebvre et al. [LHN05], ont permis de rendre disponibles les octree-textures sur les cartes graphiques supportant les *shaders*, c'est à dire les cartes graphiques pour lesquelles on peut remplacer le pipeline de traitement classique, plus précisément le traitement de la géométrie par les *vertex shader* et le traitement des fragments par les *fragment shader*, par des programmes spécifiques. Le lecteur désirant de plus amples informations sur la programmation et le fonctionnement des shaders pourra se référer à des livres comme [Ros05] par exemple.

Conjointement aux travaux de Lefebvre et al., les travaux présentés dans [KLS+05] présentent également une implémentation des octree-textures sur carte graphique. Ces derniers sont issus des travaux de Lefohn et al. [LSK+06] présentant une librairie de structure de données abstraites pour la programmation des cartes graphiques, qui était initialement de bas niveau.

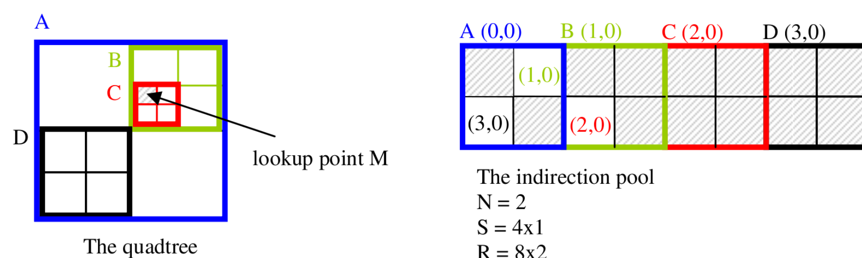
Pour rendre accessible les octree-textures aux cartes graphiques, il faut proposer une structure de données représentant l'octree, on choisit une structure que peuvent supporter les cartes graphiques, donc des textures 2D ou 3D, ainsi qu'un algorithme permettant de lire les informations dans cette structure de données. Le placage d'octree-texture, comme le placage de texture classique, s'applique lors du traitement de chaque fragment, c'est donc dans un *fragment shader* que doit être implémenté l'algorithme de lecture de l'octree-texture. Nous détaillons ici pour exemple la méthode présentée par [LHN05], c'est à dire l'encodage de l'octree dans une texture classique.

Dans l'encodage choisi par [LHN05], l'octree est sauvegardé dans une texture 3D. Chaque nœud y est représenté par un cube de 8 texels de la texture 3D. Chacun de ces texels représente un des fils du nœud. La valeur du canal alpha (la texture est donc au format RGBA) donne des informations sur le nœud fils correspondant :

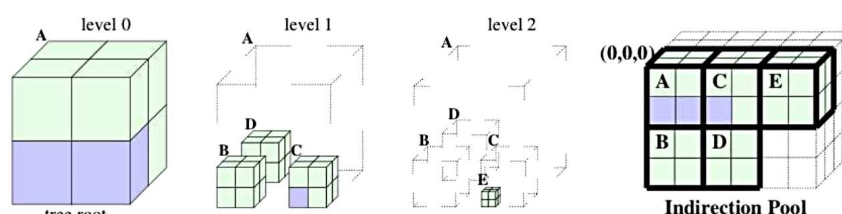
- Si alpha vaut 0, le fils correspondant est inexistant, les valeurs RGB sont ingorés
- Si alpha vaut 0.5, le fils correspondant est un nœud. Les valeurs RGB encodent un *pointeur* vers ce fils. Les pointeurs n'existant pas sur les cartes graphiques, les données sauvegardées sont en fait des coordonnées de textures permettant d'accéder au cube de texels représentant le fils dans la texture 3D.
- Si alpha vaut 1, le fils correspondant est une feuille, les valeurs RGB encodent la couleur associée à cette cellule dans l'octree.

La texture 3D représentant l'octree est appelée grille d'indirections car les cubes représentant les nœuds contiennent les *adresses* pour accéder à leurs fils respectifs. La figure 2.14 illustre l'encodage choisi par Lefebvre d'abord schématiquement sur un cas 2D, puis sur un cas 3D.

Le fragment shader doit permettre de trouver pour chaque fragment quelle valeur de l'octree-texture lui associer. Le principe est de parcourir l'octree de la racine jusqu'à la feuille contenant le fragment. Un placage d'octree-texture est donc plus coûteux qu'un placage de texture. En effet, plusieurs accès textures sont nécessaires pour accéder au texel requis. Pour parcourir la structure de l'octree il faut déterminer à chaque étape les coordonnées du fragment à l'intérieur du nœud et itérer jusqu'à rencontrer un texel dont la valeur alpha vaut 1. Nous pouvons noter que l'adressage des cubes se fait grâce aux coordonnées RGB des canaux, et que dans le cas de textures utilisant 8 bits par canal, l'espace adressable corespond donc à un cube de 256^3. Cette dimension étant la limite générale des textures 3D sur les cartes graphiques, l'espace adressable est suffisant. Cependant cette limitation pourrait être problématique dans le cas d'octrees très détaillés, la taille de la grille d'indirection étant limitée.



(a) Exemple de grilles d'indirection pour un quadtree.



(b) Exemple de grilles d'indirection pour un octree.

FIG. 2.14 – Grilles d'indirections encodant respectivement un quadtree (en haut) et un octree (en bas). La différence du genre des fils se fait grâce à la valeur du canal alpha.

2.3.5.1 Filtrage des Octree-Textures GPU

Comme pour les octree-textures sur CPU, le placage par les cartes graphiques doit permettre une interpolation linéaire et un filtrage des textures pour éviter les problèmes classiques d'aliasing. Pour l'interpolation linéaire, la solution retenue par [LHN05] est d'utiliser un voisinage de 8 cellules pour effectuer l'interpolation. Or toutes les cellules ne sont pas forcément présentes, puisque seuls les nœuds intersectant le maillage sont normalement conservés. Pour éviter le manque de données, tous les nœuds potentiellement nécessaires pour une interpolation sont donc intégrés dans la texture, même si ceux-ci n'intersectent pas le maillage. La texture est donc naturellement légèrement alourdie par la présence de ces nœuds.

Pour le MIP-mapping, la grille d'indirection ne présentant pas d'espace disponible, les valeurs moyennées des nœuds doivent être enregistrés dans une texture annexe, dont la taille est égale au nombre de cube d'indirection (donc de nœuds) dans la grille. Les auteurs notent qu'une conversion rapide des octree-texture en texture 2D est possible, ceci en connaissant une paramétrisation de la surface. L'utilisation de la texture 2D accélère les opérations de filtrage et d'interpolation, qui nécessitent plusieurs parcours de l'octree, et dégradent donc d'autant les performances déjà moindres du placage d'octree-textures. Pour éviter les problèmes de continuité, une méthode de débordement de couleurs est mise en place (voir figure 2.16).

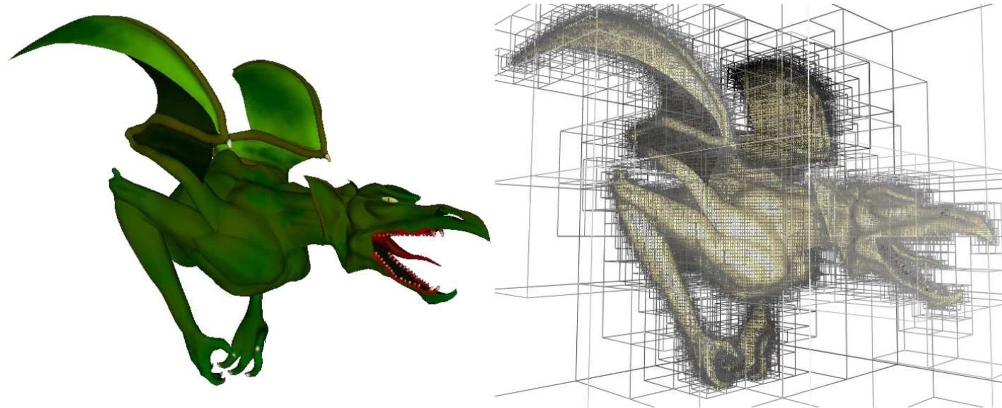


FIG. 2.15 – Un maillage habillé par une octree-texture.

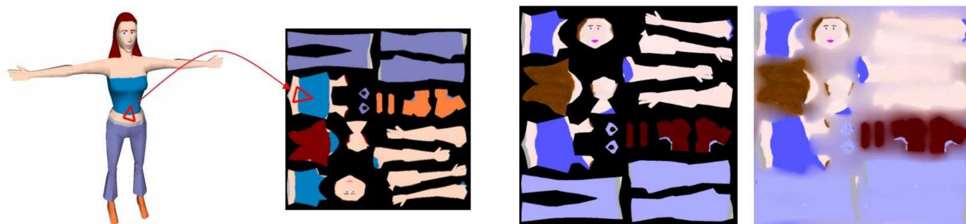


FIG. 2.16 – Lors de la conversion de l'octree-texture en atlas, les couleurs des patches débordent pour éviter les problèmes de continuité.

2.4 Conclusion

L'application de textures sur des modèles quelconques est un problème qui a été abordée de plusieurs manières. L'utilisation de textures 2D reste la méthode la plus répandue. Bien qu'elle présente des problématiques fortes, comme la difficulté de paramétriser les surfaces, des problèmes de distortion ou d'espace mémoire gaspillé, elle est la méthode d'habillage la plus performante.

Les textures volumiques permettent de faire disparaitre les problèmes des textures 2D, mais à un coût mémoire bien plus grand dans le cas général. La disponibilité des textures hiérarchiques comme les octree-textures est la solution au problème de mémoire des textures 3D, cependant leur placage, qui n'est pas implémenté nativement sur les cartes graphiques, doit se faire par un fragment shader idoine, les nombreux accès textures dégradant les performances. Toutefois, ces implémentations hardware des octree textures rendent possibles un rendu interactif, ce qui n'était pas le cas avec les premiers travaux sur les octree-textures.

Le choix du modèle de texture est donc toujours un compromis entre les performances de rendu, le temps de création et une consommation mémoire. Bien que n'étant pas la plus performante, les octree-textures sont intéressantes pour plusieurs raisons :

- Aucune paramétrisation n'est nécessaire, ce qui évite un traitement couteux voire une intervention d'un utilisateur pour la paramétrisation des surfaces.
- Echantillonnage adaptatif, la résolution de la texture est adaptée au niveau de détail réellement nécessaire.
- Niveau de détail intégré, les nœuds internes pouvant abriter des valeurs moyennées des valeurs de leurs fils.

Nous pouvons rajouter à cette liste l'indépendance vis à vis des déformations. Dans le cas d'une déformation d'un modèle, il suffit d'attacher à chaque sommet sa position originale pour trouver dans l'octree-texture la cellule qui contient les données à associer au fragment. Jusqu'alors, les octree-textures n'ont été utilisées que dans le cadre d'application de peinture interactive, dans le cadre d'utilisation de couleur d'une manière générale. Nous proposons dans cette thèse de les utiliser dans un cadre différent, celui de la préservation d'apparence. Avant de détailler notre encodage et notre programme de parcours de l'octree par le fragment shader, nous faisons dans la suite des rappels sur les travaux existants concernant la préservation d'apparence sur modèles simplifiés.

2.5 Préservation de détails

Dans les sections précédentes, nous avons présenté différentes manières de plaquer des textures 2D ou 3D. Pour chacune de ces méthodes nous avons abordé les problèmes et limitations qu'elles faisaient apparaitre, ainsi que les solutions qui avaient été apportées pour les résoudre.

Pour toutes ces techniques de placage de texture, l'utilisation sous entendue, parce qu'elle a été la première historiquement parlant, et parce qu'elle est la plus répandue, était le placage de couleurs sur un maillage. Si les maillages peuvent être habillés et rendus plus réalistes par ces textures de couleurs, tous les détails ne peuvent être mis en évidence. Par exemple les reliefs et petites aspérités d'un matériau ne seront pas visibles car les polygones sont toujours lisses, et le calcul de l'éclairage ne pourra pas prendre en compte ces reliefs (voir section 2.2.2 sur l'influence des normales dans le calcul de l'éclairage). C'est particulièrement le cas lorsqu'on utilise, pour des raisons d'économie de consommation mémoire ou de performances des versions simplifiées de modèles (voir figure 2.17).

Les textures peuvent être utilisées dans ce cas pour réintroduire sur le maillage les détails de relief qui ont été supprimés par les algorithmes de simplification, ou pour rajouter des détails de relief sur des surfaces n'en possédant pas. Nous présentons dans la suite ces algorithmes.

Fig. 2.17 – Les deux captures de gauche montrent le modèle du Neptune, un maillage de 4 millions de polygones. La capture de droite montre une version simplifiée de 100 mille polygones. Si la topologie du maillage est conservée, tous les détails de rugosité du matériau original ont disparus.

2.5.1 Bump Mapping et Normal Mapping

2.5.1.1 Bump Mapping

C'est James F. Blinn qui a le premier introduit, dans l'article [Bli78], une méthode, appelée *Bump Mapping*, pour perturber la normale le long de la surface des objets modélisés. La perturbation de la normale est dans ces travaux calculée à partir des dérivées de la texture appliquée, appelée *bump map*. Cette texture est une simple image en niveaux de gris, que l'on peut voir comme une carte d'élévation. En utilisant les dérivées partielles de l'image, la direction de la normale est modifiée. Plus les variations dans l'image sont fortes, plus la normale originale est modifiée. La figure 2.18 suivante est une illustration des résultats obtenus avec la méthode du bump mapping[6].

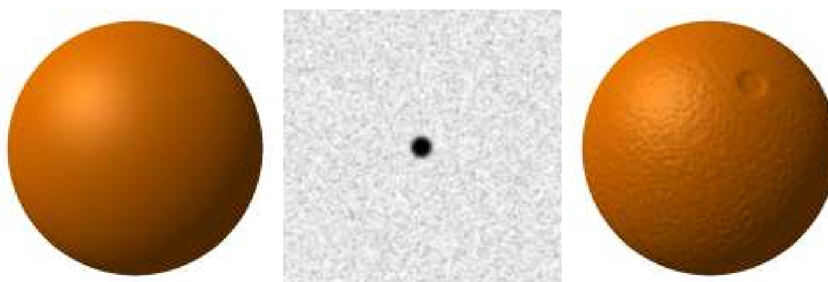


Fig. 2.18 – Une sphère lisse à laquelle on applique du *Bump Mapping*.

La normale calculée pour l'éclairage est obtenue en ajoutant à la normale du fragment une valeur dépendante des dérivées partielles de la texture. La formule de détermination de la normale est la suivante :

[6]Image extraite de l'article Wikipedia http://en.wikipedia.org/wiki/Bump_mapping

$$\vec{N}_{bump} = \vec{N} + \frac{G_u(\vec{N} \times \delta_v) - G_v(\vec{N} \times \delta_u)}{||\vec{N}||}$$

où $\vec{N}$ est la normale originale du fragment, δ_u et δ_u sont les dérivées partielles de la surface, G_u et G_v les gradients de la texture dans les directions $\vec{u}$ et $\vec{v}$ du repère de l'espace texture. Le calcul de cette normale nécessite donc la conversion des coordonnées du repère texture vers le repère tangent à la surface, de façon à ce que la perturbation soit correctement appliquée.

Pour éviter ces calculs, un faux bump mapping existe, méthode nommée *emboss bump mapping* (EBM). Il consiste à calculer dans une texture le résultat du bump mapping, en sauvegardant l'intensité lumineuse finale plutôt que la perturbation de la normale. Cette méthode produit cependant des résultats peu convaincants. Nous pouvons citer d'autres algorithmes de bump mapping, comme l'*Environment Mapped Bump Mapping* (EMBM), dont le principe est de modifier les coordonnées de texture avant d'accéder à une texture encodant l'éclairage. Ces méthodes varient par leur mode de fonctionnement, mais le principe reste souvent identique, c'est à dire se servir d'une perturbation encodée dans une texture pour modifier l'éclairage local de la surface. Nous décrivons dans la suite une méthode qui ne se base pas elle sur une perturbation, mais qui encode directement la normale à appliquer pour le calcul de l'éclairage.

2.5.1.2 Normal Mapping

Les *bump maps* encodent une variation à appliquer aux normales le long de la surface, calculée en fonction du gradient de la texture. Cette dernière est donc une image simple canal, typiquement un niveau de gris. Le placage implique donc de calculer pour chaque fragment la variation à appliquer. Or il est possible d'utiliser des textures qui vont sauvegarder non pas la variation, mais directement la normale à utiliser pour le calcul de l'éclairage, comme proposé dans les travaux de [Fou92] ou [PAC97]. La normale d'un fragment de la surface est remplacée par celle lue dans la texture à la coordonnée de texture correspondante, évitant ainsi le calcul de la variation et des changements de repère qu'il implique (voir figure 2.19). Les textures encodant la normale finale sont logiquement appelées *normal maps*. Contrairement aux bump maps, les normales maps nécessitent trois canaux (RGB), pour enregistrer les trois composantes des normales.

Le principal intérêt du normal mapping est de pouvoir améliorer l'apparence d'un maillage basse résolution avec des normales issues d'une version haute résolution du même objet. Plus que de simples motifs ou aspérités rajoutés à la surface d'un objet, c'est l'apparence réelle des objets qui peut être encodée dans les normal maps, permettant l'utilisation de maillages composés d'un faible nombre de polygones assurant un rendu rapide, tout en offrant la richesse visuelle d'un objet haute résolution. Dans la section suivante nous décrivons des méthodes permettant de générer ces textures d'apparence à partir des maillages haute résolution.

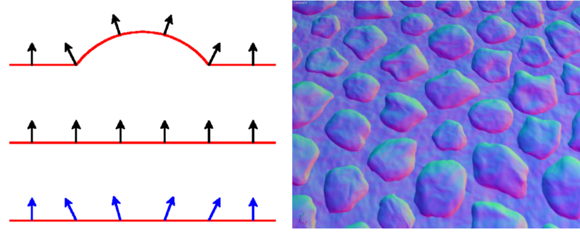


FIG. 2.19 – Le schéma de gauche illustre le principe du normal mapping, en haut se trouve le maillage original, au centre le modèle simplifié avec ses normales uniformes, en bas le maillage simplifié est habillé avec les normales du maillage original. La photo de droite est un exemple de *normal map*.

2.5.2 Préservation d'apparence sur maillages simplifiés

2.5.2.1 *A general method for preserving attribute values on simplified meshes*

Cignoni et al. ont présenté dans les articles [CMSR98] et [CMR+99] une des premières méthodes pour la création de textures sauvegardant l'apparence de maillages originaux. Leur méthode consiste à créer un atlas de polygones (voir section 2.3.1.1) pour le maillage simplifié.

Un taux d'échantillonnage définissant la qualité de la texture est défini. En fonction de ce taux, chaque triangle se voit attribuer un nombre d'échantillons représentatifs. Chaque triangle est en quelque sorte enveloppé par des cellules plus ou moins grandes selon le taux d'échantillonnage. Pour chacune de ces cellules, le point le plus proche sur le maillage original est déterminé, les données (couleur, normale) de ce point sont attribuées à la cellule. Dans le but d'identifier rapidement le point le plus proche, une grille régulière entourant la surface est utilisée. Ce partitionnement spatial permet de localiser efficacement les triangles du maillage original voisins du triangle du maillage basse résolution.

L'ensemble des cellules entourant un triangle forme ensuite le patch de texture pour ce triangle dans l'atlas. Un soin particulier est apporté au placement de triangles dans l'atlas. Deux méthodes sont employées (illustrées figure 2.20) :

- La première n'utilise que des patches ayant la forme de demi carrés. Il est ainsi facile d'agencer au mieux les triangles dans l'atlas, en évitant au maximum la perte d'espace. Cependant cette première méthode implique d'adapter le taux d'échantillonnage des triangles aux tailles de patches disponibles dans l'atlas (typiquement des dimension en puissance de 2).
- La seconde permet d'utiliser des patches de forme irrégulière. Le taux d'échantillonnage spécifié par l'utilisateur est mieux respecté, mais le placement des triangles dans l'atlas est plus problématique. Pour minimiser les pertes d'espace, les auteurs

proposent d'utiliser un nombre fixe de hauteurs différentes pour les triangles, et de déformer en les inclinant les triangles de façon à les faire s'agencer au mieux dans la texture.

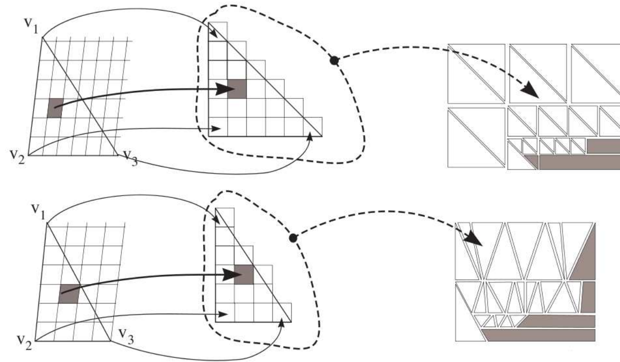


FIG. 2.20 – Méthodes de packing utilisées par [CMSR98, CMR+99]. En haut, utilisation de patches réguliers, en bas de patches irréguliers. Les zones grisées représentent l'espace texture inutilisé.

Bien que cette méthode évite le calcul d'une paramétrisation globale ou locale par morceaux de la surface, et permette de calculer l'atlas de texture de manière totalement indépendante de la méthode de simplification utilisée pour générer le maillage simplifié, quelques problèmes sont notables. Notons en premier que quelle que soit la méthode de placement des triangles utilisée, une déformation des triangles est inévitable et donc des distorsions, outre les soucis de continuité aux frontières de triangles, peuvent apparaitre.

Mais le souci majeur réside dans l'utilisation d'un taux d'échantillonnage fixe. Un taux d'échantillonnage faible permet de limiter la taille des textures, mais en perdant des données d'apparence. L'utilisation d'*over-sampling* est dans ce cas préconisée par les auteurs, de façon à inclure dans chaque texel la contribution de plusieurs éléments de la surface originale. Cette solution n'est toutefois pas capable de restituer tous les détails perdus. L'augmentation du taux d'échantillonnage n'est pas sans limite, car ce taux s'applique à tous les triangles de la surface, la taille de la texture peut donc vite exploser. Ces problèmes sont liés à la nature 2D de la texture, et donc au fait que le taux d'échantillonnage s'applique à toute la surface de l'objet, y compris aux zones qui ne présentent pas de variation forte dans l'apparence.

2.5.2.2 ***Appearance-preserving simplification***

Une autre approche pour la création des textures d'apparence a été proposée par Cohen et al. [COM98]. Il s'agit de créer les textures en même temps que l'opération de

FIG. 2.21 – Préservation d'apparence sur maillage simplifié par la méthode de Cignoni et al. [CMSR98, CMR+99]. De gauche à droite : maillage original, maillage simplifié, maillage simplifié avec application de texture d'apparence.

simplification du maillage est appliquée. Cette méthode repose sur une paramétrisation de la surface originale. Si celle-ci n'est pas disponible, il faut dans un premier temps la calculer. Lorsque la paramétrisation est connue, la première étape consiste à créer les textures. Ces dernières sont trivialement obtenues en projetant le maillage sur le plan texture grâce à la paramétrisation. La résolution de la texture est choisie de telle sorte que chaque texel n'accueille la contribution que d'un seul sommet du maillage au maximum, afin d'éviter toute perte de données. La position du texel dans la texture correspond aux coordonnées de texture du sommet.

Une fois que les textures ont été créées, l'algorithme de simplification est appliqué. Les auteurs ont choisi une simplification basée sur l'opération d'*edge collapse*, c'est à dire de *réduction d'arêtes*. Une arête d'un triangle est réduite à un seul sommet, réduisant ainsi peu à peu le nombre de triangles composant le maillage. Les algorithmes de simplification sont généralement basés sur une mesure d'erreur pour conserver au mieux la forme du maillage original. A ces mesures d'erreur, les auteurs de [COM98] rajoute un critère d'erreur calculé non pas sur le maillage, mais sur la texture elle même. Un nouveau sommet issu d'une opération de réduction d'arêtes se voit attribuer des coordonnées de textures. En retrouvant sur le maillage original le sommet possédant les mêmes coordonnées de texture, on peut calculer la distance entre les deux sommets partageant les mêmes coordonnées de textures sur les deux maillages différents. Une tolérance sur cette distance permet de contrôler la distorsion introduite par la simplification. Les captures de la figure 2.22 mettent en évidence les résultats obtenus par cette méthode.

Comme les méthodes décrites précédemment, cette dernière présente également des inconvénients notables, qui sont toujours liés à l'utilisation de textures 2D : taille de la texture, distorsion à prendre en compte etc. De plus, le calcul nécessaire de la paramétrisation de la surface peut nécessiter l'intervention d'un utilisateur comme nous l'avons déjà fait remarquer, et peut se révéler très compliqué à obtenir pour des maillages très détaillés.

Fig. 2.22 – Préservation d'apparence sur maillage simplifié par la méthode de Cohen et al. [COM98]. La rangée du haute présente des captures du modèle simplifié rendu avec les textures d'apparence, la rangée du bas présente elle le maillage simplifié uniquement. Les modèles utilisés sont composés de 250 mille à 975 polygones (de gauche à droite).

Une approche similaire, couplant génération de textures et simplification du maillage, avait été adoptée par les auteurs de [SGR96] mais pour des maillages colorés uniquement. La paramétrisation était évitée par l'utilisation d'un atlas de polygones, atlas dans lequel chaque triangle est représenté par un demi carré (voir section 2.5.2.1 précédente).

2.5.2.3 *Rapid Visualization of Large Point-Based Surfaces*

Dans l'article [BDS05], Boubekeur a présenté une méthode de préservation d'apparence, méthode qui s'applique à des objets n'étant pas sous la forme de maillages polygonaux, mais sous forme de nuages de points. Cette représentation des modèles consiste en un ensemble de points, possédant chacun ses informations de normale et couleur si nécessaire. Des algorithmes spécifiques doivent être mis en œuvre pour le rendu de ces structures, citons par exemple [RL00]. Les scanners 3D produisent en sortie des nuages de points, la chaine de traitement classique est de calculer une triangulation pour obtenir un maillage polygonal. Ce maillage peut ensuite être simplifié, et habillé de textures. Ce traitement est très coûteux que ce soit en temps ou en mémoire étant donné que les nuages de points peuvent être très volumineux.

Dans [BDS05], la méthode présentée permet d'éviter la triangulation en calculant à la fois le maillage simplifié et les *normal maps* directement à partir du nuages de points. Il n'y a dans ce cas pas de paramétrisation à calculer. Une simplification est appliquée au nuage de points, en plaçant les points dans une grille régulière, et en calculant un seul échantillon représentatif à l'intérieur de chaque cellule. Le nuage de points simplifié est plongé dans un octree, et sert de base pour générer le maillage simplifié. Le maillage n'est pas reconstruit globalement, mais à l'intérieur de chaque cellule une partie de surface est générée grâce à une triangulation de Delaunay 2D, l'ensemble de ces morceaux formant l'objet global.

Chaque morceau de surface se voit attribuer une *normal map*, qui est remplie par projection des normales du nuage de points original sur la surface. La figure 2.23 illustre les étapes de création du maillage et de la texture d'apparence.

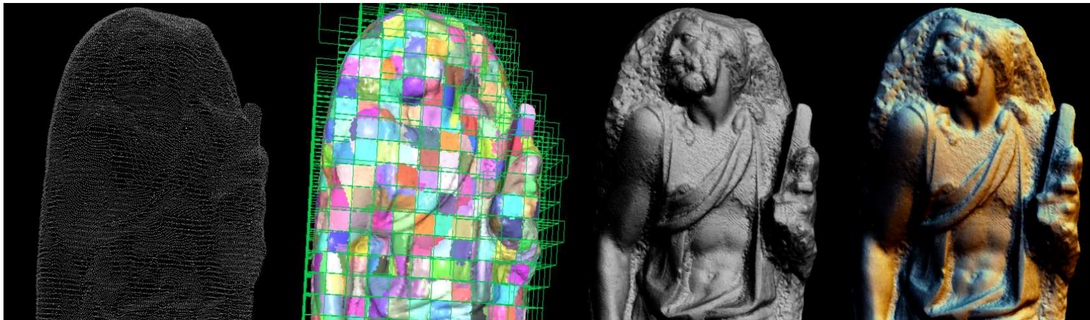


FIG. 2.23 – Construction de modèle simplifié avec carte d'apparence par la méthode de [BDS05]. De gauche à droite : le nuage de point rééchantillonné, la collection de morceaux de surface formant l'objet simplifié, puis l'objet texturé dans deux conditions d'éclairage différentes (une et trois sources lumineuses).

Grâce à cette méthode, le modèle simplifié et sa texture d'apparence sont créés sans faire appel à une quelconque paramétrisation globale. Il y a pour chaque morceau de surface une paramétrisation locale, mais la construction des morceaux de surface est faite de telle sorte que chaque morceau soit localement quasi planaire, la paramétrisation locale devient donc facile à déterminer. Cependant des distorsions pourraient apparaitre dans les zones de courbure forte. Les cartes de normales associées à chaque morceau de surface doivent ensuite être assemblés dans des textures de grandes dimensions. De plus, la création des textures par projection des normales sur les textures peut laisser des zones vides, ce qui implique la mise en place d'algorithme de diffusion sur la surface pour remplir tous les trous laissés vides de données.

2.5.3 Autres méthodes de préservation de détails

Le bump mapping, et son extension le normal mapping, n'est pas la seule méthode connue pour ajouter des détails à la surface d'un objet simplifié à partir de données contenues dans des textures. Nous présentons ici succintement quelques autres techniques répandues, bien que nous n'ayons pas dans nos travaux utilisé ou implémenté ces algorithmes.

L'idée d'utiliser sur des maillages simplifiés des textures pour enrichir l'apparence avait déjà été proposée dans des articles comme [Coo84, CCC87, KL96], il s'agissait de reconstruire le maillage original en subdivisant les polygones et déplaçant les nouveaux sommets en fonction d'une distance enregistrée dans une texture, c'est à dire une carte d'élévation. Cette méthode est connue sous le nom de *displacement mapping*. Elle n'est pas à proprement parler une méthode de placage d'apparence, puisque la géométrie orginale est reconstruite à partir de la carte d'élévation. Le rendu temps réel des maillages complexes reste donc problématique, puisque le nombre de polygones à manipuler reste identique. L'utilisation des processeurs graphiques pour le traitement de la géométrie, et la possibilité de raffiner localement le maillage (voir [BS08, Don05]) permet cependant d'envisager du *displacement mapping* en temps réel.

Une autre approche basée sur du rendu basé image est possible. Cette méthode présentée dans [Deb96] puis étendue dans [PDG05] consiste à capturer plusieurs images d'un même objet en utilisant différents points de vue. Lors du rendu, le maillage simplifié est utilisé, les texels à appliquer à chaque fragment sont pris dans les images correspondant au plus proche point de vue, voire interpolés entre différents points de vue. On parle alors de rendu projectif, les captures étant projetés sur le maillage simplifié. Cependant cette méthode nécessite un nombre non négligeable de captures autour de l'objet original, captures qui peuvent être automatisées, ou dirigées par un utilisateur (comme dans [PDG05]).

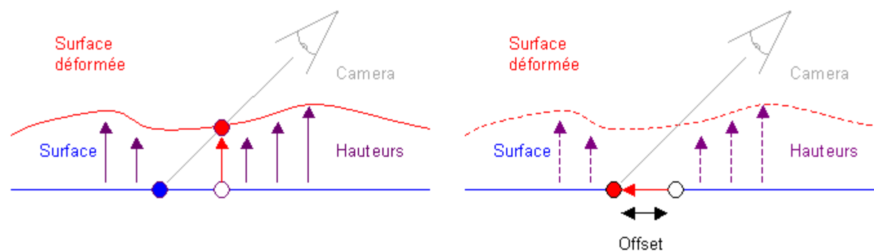


FIG. 2.24 – Parallax mapping : le texel colorié bleu est affiché, or le point de vue intersecte la surface à l'endroit où le texel blanc devrait être utilisé. Le but est de calculer un décalage dans les coordonnées de texture pour utiliser le bon texel.

Un des défauts du placage de normale, et qui n'est pas présent avec une méthode comme le *displacement mapping*, c'est que les silhouettes des objets ne sont pas correctement rendues. En effet, ce sont les silhouettes de objets simplifiés qui sont affichées, et non celles de l'objet original. La géométrie ayant été simplifiée, il est impossible de retrouver la forme originale des objets. Il existe cependant une méthode permettant de simuler les effets de la géométrie réelle, sans pour autant modifier l'objet simplifié comme le nécessite le *displacement mapping*, il s'agit du *parallax mapping*. Le principe est de décaler les coordonnées de texture en fonction du point de vue (voir schémas de la figure 2.24), de façon à trouver le texel correspondant au sommet réellement intersecté par la surface originale, et pas celui du maillage simplifié. Cette méthode implique donc la présence des informations de normale et d'élévation dans la texture.

Le *parallax mapping* donne une meilleure perception du relief, mais ne permet pas de déterminer les silhouettes originales. Si les travaux de [SGG+00] propose une solution aux problèmes des silhouettes, le *parallax mapping* peut être optimisé dans le but d'intégrer la gestion des silhouettes [POJ05,MMO05]. Le principe est d'implémenter une sorte de lancer de rayon entre le point de vue et le relief. Si le point de vue intersecte le relief (c'est à dire la carte d'élévation), alors le texel associé est utilisé. Si le point de vue n'intersecte pas le relief, alors le fragment est annulé, car il n'appartient pas à l'objet. Cette méthode permet d'obtenir les silhouettes réelles des objets (voir figure 2.25).

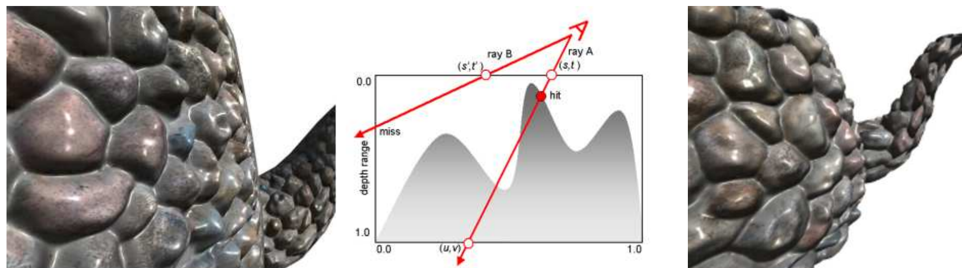


FIG. 2.25 – De gauche à droite : relief mapping sans gestion des silhouettes, schéma de test d'intersection entre le point de vue et la surface, résultat avec gestion des silhouttes.

L'état de l'art [SKU08] présente la plupart de ces méthodes de placage de détails sur des modèles simplifiés. Dans le cadre de cette thèse, l'application de telles méthodes est malheureusement inenvisageable à l'heure actuelle. En effet, la détection de l'intersection entre le point de vue et le relief nécessite de nombreux accès textures pour déterminer, soit par dichotomie, soit échantillonnage à intervalles de distance régulier le long du rayon, la position souhaitée. Or, comme nous nous proposons d'utiliser les octrees pour échantillonner le champs de normales du modèle haute résolution puis pour construire une octree-texture d'apparence, chaque accès texture nécessite un parcours d'octree. Ef-

fectuer 32 parcours[7] d'octree, soit environ 300 accès texture si l'octree est de profondeur 10, serait trop coûteux et ne permettrait pas d'obtenir un rendu temps réel interactif étant donné le temps d'accès aux textures sur les matériels actuels.

[7] nombre d'accès texture implémenté dans la méthode de [POJ05] pour trouver l'intersection entre le point de vue et le relief

2.5.4 Conclusion

Nous avons montré dans cet état de l'art comment les algorithmes de rendu prenaient en compte la géométrie des objets pour calculer l'éclairage d'une scène. Plus la géométrie est complexe, c'est à dire composée d'un grand nombre de sommets et triangles, plus les objets sont détaillés, mais la manipulation interactive devient problématique.

Les modèles d'habillage permettent d'augmenter le nombre de détails à la surface sans alourdir pour autant la géométrie des objets. Si les textures ont initialement été proposées pour colorer la surface des objets, nous avons vus comment elles ont été utilisées pour enrichir les modèles simplifiés de détails de relief. En modifiant la normale associée à chaque fragment de la surface, avec les informations contenues dans une texture, les détails supprimés par les algorithmes de simplification peuvent être réintroduits sur la surface. Les objets manipulés ont donc les détails d'un maillage haute résolution, mais leur complexité simplifiée permet d'obtenir des grandes vitesses d'affichage.

Si le *normal mapping* n'est pas la méthode d'ajouts de détails la plus sophistiquée (voir les autres méthodes en section 2.5.3), son utilisation offre tout de même une grande richesse visuelle de détails. De plus sa simplicité de mise en œuvre est un avantage indéniable quant il s'agit de l'implémentation du calcul de l'éclairage.

Comme pour les travaux présentés dans la section 2.5.2 nous nous proposons dans cette thèse de calculer des textures d'apparence à partir de maillages haute résolution afin de permettre une manipulation interactive d'objets simplifiés ayant l'apparence de modèles complexes. Cependant, nous avons également vus les problématiques liées à l'utilisation de textures 2D, ainsi que celles des textures 3D conventionnelles. Nous proposons donc l'utilsation des octree-textures comme support de l'apparence. Comme dans [BDS05, CMSR98, CMR+99], où une grille régulière volumique est employée, l'octree est dans un premier temps utilisé comme support pour l'échantillonnage du champs de normales du maillage original. L'octree est ensuite encodée de manière compacte dans une image 2D, image exploitée par un fragment shader lors du rendu.

L'utilisation d'octree-texture rend le processus de création des textures automatique car il évite la nécessité d'une paramétrisation, et il permet de bénéficier des avantages d'une structure hiérarchique. Nous détaillons dans la suite du manuscrit les différentes étapes de la création des textures, et leur fonctionnement pour le rendu temps réel. Enfin nous montrerons une application originale de la structure d'octree-texture dans un cadre d'application de visualisation à distance de maillages.

Deuxième partie

Contribution

Chapitre 3

Construction des *Appearance Preserving Octree-Textures*

Sommaire

3.1 Introduction **42**
3.2 Subdivision de l'Octree **44**
3.2.1 Paramètres et structures de données 44
3.2.2 Initialisation de l'octree et subdivision 46
3.2.3 Subdivision adaptative 47
3.2.4 Critère d'erreur : métrique $L^{2,1}$ 49
3.2.4.1 Influence de la valeur de tolérance sur la subdivision 51
3.2.4.2 Gestion de l'écart entre m et M 54
3.3 Assignation des normales aux feuilles de l'octree **58**
3.3.1 Echantillonnage des normales par lancer de rayons 58
3.3.1.1 Choix de l'origine et de la direction du rayon . . 59
3.3.1.2 Liste des nœuds intersectés par le rayon 61
3.3.1.3 Recherche de la plus proche intersection 63
3.3.2 Echantillonnage par moyennage des normales des feuilles . 64
3.3.2.1 Calcul des normales pour les nœuds internes . . . 66
3.4 Encodage de l'APO en texture 2D **67**
3.4.1 Organisation en largeur de l'octree 67
3.4.2 Encodage sans valeurs MIP-map 68
3.4.2.1 Quantification sur trois octets des normales . . . 69
3.4.3 Encodage avec valeurs MIP-map 70
3.4.4 Sauvegarde sous forme de texture 2D 72
3.5 Performances et discussion **73**
3.5.1 Temps de construction des APO 73
3.5.2 Occupation mémoire des APO 74
3.5.2.1 Taille des textures APO 75
3.5.2.2 Occupation en mémoire vive lors de la construction 77

3.1 Introduction

Nous allons utiliser les *octree-textures* (voir section 2.3.4.1 page 22) pour sauvegarder l'apparence d'un maillage 3D. Cette texture, que nous appellons *APO* pour *Appearance Preserving Octree-texture* sera plaquée sur un maillage simplifié issu du maillage original. Dans la suite nous noterons le maillage original M et le maillage simplifié m.

La création de la texture *APO* est indépendante de la méthode de simplification choisie, cette approche d'indépendance entre simplification et création de la texture de détails avait également été adoptée par [CMR+99]. Nous ne traitons donc pas dans nos travaux de la méthode de simplification de maillage utilisée pour créer m. Les maillages simplifiés utilisés ont été obtenus en utilisant les algorithmes basés sur l'erreur quadratique, tels que celui présenté dans [MP97], mais tous les algorithmes peuvent être employés. Dans la section 3.2.4.2 page 54 les effets du degré de simplification sur la création de l'*APO* seront discutés.

Étant donnés les deux maillages M et m nous nous proposons de créer une texture *APO* pour m encodant les détails, c'est à dire les normales, du maillage original. L'échantillonnage des normales de M se fait par la construction adaptative d'un octree autour du maillage simplifié, puis par une phase de lancers de rayons pour la récupération des normales. Après l'échantillonnage, l'octree en est encodé dans une texture 2D. La construction des *APO* est donc décomposée en trois étapes distinctes :

1. **Subdivision de l'octree :** Cette première étape consiste à créer la structure de l'octree. La subdivision s'effectue en deux temps. Dans un premier temps un octree grossier de profondeur limitée est construit autour du maillage m, puis une subdivision adaptative est appliquée sur chacune des feuilles de l'octree grossier. La problématique de la construction est la prise en compte des détails de M. Les zones présentant de fortes variations dans le champs de normales doivent être couvertes par des feuilles de profondeur élevée, les zones présentant moins de détails étant couvertes par des feuilles de profondeur moins grandes.
2. **Assignation des normales aux feuilles de l'octree :** Lors de cette étape, le champ de normales du maillage original est échantillonné. Chaque feuille de l'octree se voit assigner une normale représentative de M, normale qui est déterminée par un lancer de rayons de l'octree vers M. Les directions des rayons sont choisies pour éviter tout sous-échantillonnage de M. L'utilisation de la structure d'octree permet d'accélérer la phase d'échantillonnage en facilitant la localisation des intersections entre les rayons et M.
3. **Encodage :** Les octree-textures ne sont pas matériellement supportées par les cartes graphiques comme le sont les textures 2D ou 3D. Nous proposons un encodage de l'octree-texture qui rend les données disponibles pour les cartes graphiques. L'octree sera encodé dans une texture 2D, nous traiterons du rendu dans le chapitre 4. L'encodage proposé contient toutes les informations nécessaires au rendu, c'est à

dire les normales mais également la structure de l'octree. Celle-ci est utilisée pour identifier la normale à associer à chaque fragment lors du calcul de l'éclairage. De plus, l'encodage est le plus compact possible pour rendre les *APO* les plus légères possibles.

Ces trois étapes sont détaillées dans la suite de ce chapitre. Si les textures *APO* construites sont plaquées lors du rendu par le processeur graphique (*GPU*), toutes les étapes de la création sont elles exécutées par le processeur central (*CPU*). La création d'un octree sur *GPU* est en effet difficile, puisque la gestion des pointeurs ou encore l'allocation dynamique ne sont pas disponibles sur les cartes graphiques même récentes.

3.2 Subdivision de l'Octree

3.2.1 Paramètres et structures de données

L'octree contenant les détails est construit durant cette étape. La construction se fait par une subdivision récursive appliquée aux feuilles de l'octree. Cette subdivision est pilotée par un critère d'erreur qui permet d'adapter la résolution de l'octree aux détails du maillage M, en mesurant la variation des normales à l'intérieur de chacun des nœuds.

Pour arrêter les subdivisions récursives, outre le critère d'erreur, une profondeur maximale est spécifiée. Ces deux paramètres sont liés et permettent de contrôler le niveau de détails des APO et également leur taille mémoire. En effet le critère d'erreur seul ne permet pas de limiter la profondeur de l'octree, car sur les arêtes vives, ou sur une zone présentant une très haute fréquence de variation de normales, l'erreur autorisée ne sera jamais satisfaite, forçant ainsi les subdivisions à s'effectuer. Ce cas est illustré en 2D sur la figure 3.1. La profondeur maximale limite ainsi le nombre de subdivisions et évite

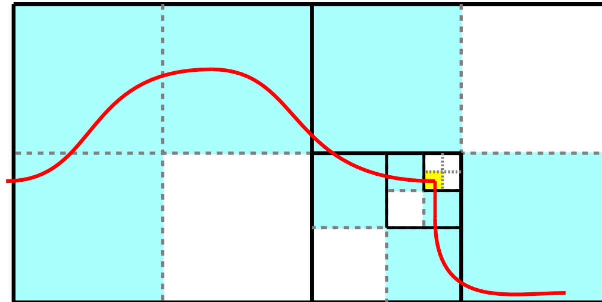


FIG. 3.1 – Le maillage (courbe en rouge) est plongé dans un octree. La feuille marquée en jaune présente une arête vive, la variation mesurée sera toujours au dessus de la tolérance de variation autorisée si celle-ci est faible, forçant la subdivision. L'utilisation d'une profondeur maximale permet d'éviter ce cas de figure.

à la texture APO une occupation mémoire trop grande. Notons que comme la texture de détails est construite pour le maillage simplifié m, seules les feuilles de l'octree qui intersectent ce maillage sont conservées lors des subdivisions.

Pour mesurer la variation des normales du maillages M, une liste des triangles de M intersectés est placée dans chacun des nœuds de l'octree. De même, une liste des triangles de m est maintenue. Nous notons ces deux listes comme suit :

- T^N : liste des indices des triangles du maillage original M étant contenus ou intersectés par le nœud N.
- t^N : liste des indices des triangles du maillage simplifié m étant contenus ou intersectés par le nœud N.

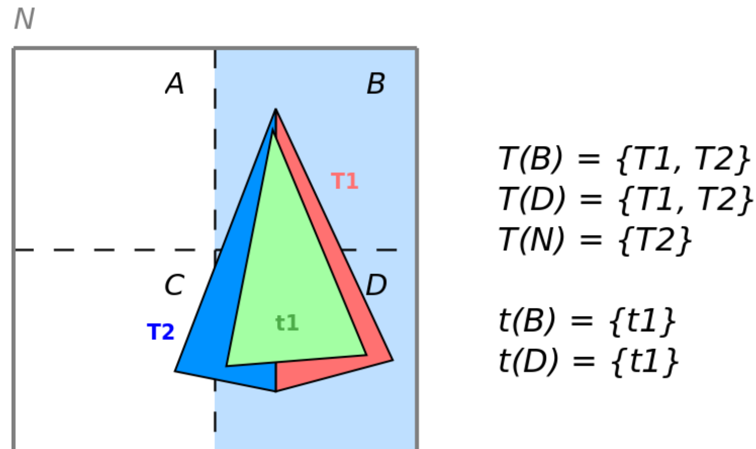


FIG. 3.2 – Les trianges *T1* et *T2* appartiennent au maillage original *M*, tandis que le triangle *t1* appartient au maillage simplifié *m*. Le nœud N a uniquement deux fils, les feuilles B et D qui intersecent le triangle *t1*. Le triangle T2 est ajouté à la liste T^N de N car il n'est pas entièrement inclus dans les deux feuilles existantes.

Ces deux listes permettent d'identifier rapidement les triangles nécessaires pour l'échantillonnage du champs de normales dans chaque cellule de l'octree. Comme l'octree est construit autour du maillage *m*, ce dernier est entièrement compris dans l'enveloppe formée par les feuilles de l'octree. Le maillage *M* peut par contre dépasser cette enveloppe. Les indices des triangles de *M* sont inscrits dans les listes T^N des feuilles qu'ils intersectent, mais également dans celles des nœuds parents (voir l'illustration en 2D sur la figure 3.2).

Le début du processus de subdivision peut-être considérablement accéléré en construisant un octree d'une profondeur fixée autour de *m*. En effet, les détails de *M* seront placés dans les feuilles de profondeur élevée, il n'est donc pas nécessaire d'effectuer des tests de subdivision sur les premiers niveaux de l'octree. Les subdivisions peuvent commencer au niveau 6 ou 7 par exemple.

Pour résumer, la construction de l'octree est pilotée par 3 paramètres :

- La profondeur initiale de l'octree construit autour de *m*, qui évite les premiers tests de subdivsion inutiles, notée $profIn$.
- La tolérance sur la variation de normales dans les feuilles qui permet d'ajuster le niveau de détails, c'est à dire la qualité de l'APO. Nous notons cette mesure de variation V^N.
- La profondeur maximale autorisée lors des subdivisions des nœuds de l'octree, qui permet de limiter la taille mémoire des APO. Elle est notée $profMax$.

3.2.2 Initialisation de l'octree et subdivision

L'octree que nous construisons est centré sur le cube unitaire, c'est à dire le cube de largeur 1.0 centré autour de la position $(0.5, 0.5, 0.5)$. Dans un premier temps les maillages m et M sont donc redimensionnés à l'échelle du cube unitaire.

L'octree initial est ensuite construit autour de m. Pour ce faire, chaque triangle de m est inséré dans l'octree initialement vide. Les nœuds sont subdivisés jusqu'à ce que le triangle courant soit entièrement compris dans des feuilles de profondeur $profIn$. L'algorithme 1 présente la manière de construire cet octree. Lors de la subdivision, seuls les octants fils intersectant m sont conservés.

```
Données : Maillages m, Octree O
Entrées : entier profIn : profondeur initiale de l'octree
Résultat : Octree O initial de profondeur profIn construit autour de m

// Boucle Principale. La fonction racine retourne la racine de l'octree
// La fonction fils retourne le i^ème du nœud
pour chaque triangle t de m faire
    InsererTriangleOctree(t,Racine(O))

// Procedure récursive d'ajout des triangles à l'octree
procedure InsererTriangleOctree(triangle t, nœud N)
début
    si (Profondeur(N) == profIn) alors
        si (Intersecte(t,N)) alors
            Ajouter le triangle t à la liste t^N de N;
    sinon
        Subdiviser(N);
        pour i de 1 à 8 faire
            si (Intersecte(t, Fils(N,i))) alors
                // Appel récursif sur les fils existants
                InsererTriangleOctree(t, Fils(N,i));
            sinon
                si (liste t^N du Fils(N,i) est vide) alors
                    Supprimernœud(Fils(N,i));
fin
```

Algorithme 1 : Algorithme de création de l'octree initial.

Chaque triangle est inséré dans la racine de l'octree, puis passé récursivement aux fils des nœuds intersectés. L'indice du triangle est ajouté à la liste t^N des feuilles de profondeur $profIn$ qui intersectent le triangle courant. Lors du parcours de l'octree, si un nœud dont la profondeur est inférieure à $profIn$ est recontré, il est subdivisé. Lors de la subdivision, les nœuds fils obtenus n'intersectant aucun triangle sont supprimés.

A la fin de la procédure, un octree de profondeur $profIn$ enveloppe m, chacune de ses feuilles contient dans sa liste t^N les indices des triangles de m qu'elle intersecte.

Le maillage M est ensuite lui aussi plongé dans l'octree initial. Chaque triangle de M est ajouté dans les listes T^N des feuilles qu'il intersecte. Si un triangle dépasse de l'enveloppe formée par les feuilles (tel *T2* dans la figure 3.2) alors il est aussi ajouté dans les listes T^N des nœuds parents (N dans la figure 3.2).

3.2.3 Subdivision adaptative

Nous détaillons maintenant la subdivision adaptative appliquée aux feuilles de l'octree initial. L'algorithme principal est un algorithme récursif (voir algo 2) qui est appelé depuis chaque feuille de profondeur $profIn$. Le but est de subdiviser l'octree autour du maillage m, car l'octree-texture est appliquée à ce maillage lors du rendu. Les subdivisions sont effectuées tout pendant que la variation des normales des triangles de M intersectant les nœuds (liste T^N) est supérieure à une tolérance fixée.

```
Entrées : entier profMax : profondeur maximale autorisée
Entrées : réel tol : tolérance sur la variation de normales

procedure SubdiviseOctreeRec(nœud N, entier profMax) début
    si (N est un nœud) alors
        pour i de 1 à 8 faire
            si (Fils(N,i) existe) alors
                SubdiviseOctreeRec(Fils(N,i), profMax);
    sinon
        // N est une feuille
        si (Profondeur(N) < profMax) alors
            // La profondeur maximale n'a pas été atteinte
            V^N ← CalculVariation(T^N);
            si (V^N > tol) alors
                Subdiviser(N);
                Supprimer_feuilles_n_intersectant_pas_m ();
                Mise_à_jour_des_listes_d_indices();
                // Appel récursif sur les nouveaux fils
                pour i de 1 à 8 faire
                    si (Fils(N,i) existe) alors
                        SubdiviseOctreeRec(Fils(N,i), profMax);
fin
```

Algorithme 2 : Algorithme de subdivision adaptative de l'octree.

La variation V^N des normales de M est mesurée dans chaque feuille, et si celle-ci dé-

passe une valeur seuil passée en paramètre, la feuille est subdivisée, les fils intersectant m sont conservés puis le même traitement leur est appliqué. Nous détaillerons le calcul de la variation V^N dans la section 3.2.4 suivante.

Lors de ces subdivisions, une des problématiques est la mise à jour des listes t^N et T^N des nœuds. La mise à jour de ces listes se fait de manière indépendante, en effet il faut respecter les conditions vues précédemment (voir la partie 3.2.1 page 44 concernant le contenu des listes d'indices).

La mise à jour des listes t^N, c'est à dire celles qui sauvegardent la liste des triangles de m intersectés par les cellules de l'octree, est aisée. Lors de la subdivision, les triangles de la liste t^N d'un nœud sont passés aux fils créés, puis la liste du nœud est vidée. En effet, m est enveloppé par les feuilles de l'octree, les nœuds internes ont donc leur liste t^N vide. Cette opération est effectuée suivant l'algorithme 3.

```
Données : nœud N de l'octree

procedure Mise_a_jour_tN(N)  // Passage des indices aux cellules filles
pour chaque triangle t de la liste t^N de N faire
    pour i de 1 à 8 faire
        si (Intersecte(t,Fils(N,i))) alors
            Ajouter le triangle t à la liste t^N de Fils(N,i);
// Vider la liste t^N de N
Vider(t^N de N);
// Les fils n'intersectant aucun triangle de m sont supprimés
pour i de 1 à 8 faire
    si (liste t^N de Fils(N,i) est vide) alors
        SupprimerNœud(Fils(N,i));
```

Algorithme 3 : Algorithme de mise à jour des indices de t^N lors de la subdivision d'une feuille.

Pour la mise à jour des listes T^N, l'algorithme est identique à la différence près que les triangles de la liste T^N de chaque feuille sont supprimés uniquement si ils sont entièrement enveloppés dans les nouvelles feuilles générées lors de la subdivision. En effet, comme nous l'avons vu précédemment, un nœud continue de référencer les triangles de M qui ne sont pas complètement enveloppés par ses fils (voir figure 3.2). Un triangle de la liste T^N qui n'est pas entièrement enveloppé par les fils d'un nœud est donc conservé dans la liste du nœud. Pour déterminer si un triangle est entièrement enveloppé par les fils, nous calculons l'intersection des triangles avec les fils du nœud qui ne sont pas conservés lors de la subdivision. Si un triangle de T^N intersecte un de ces fils, c'est qu'il n'est pas entièrement enveloppé par les fils subsistants, son indice est donc conservé dans la liste T^N du nœud. Dans le cas contraire, il est supprimé.

Dans l'algorithme 3 de passage des triangles aux nœuds fils lors de la subdivision adaptative, ainsi que dans celui de création de l'octree initial (algorithme 1 page 46), nous retrouvons une opération commune qui consiste à faire passer les triangles des nœuds vers leurs fils. Un triangle est ajouté à un fils si celui-ci l'intersecte ou si il y est entièrement inclus. Si le test d'inclusion d'un triangle dans un nœud est trivial, celui d'intersection entre un triangle et une boîte (nœud de l'octree) est plus difficile. Comme ce test est effectué un grand nombre de fois lors des différentes subdivisions, son efficacité est critique pour les performances de création des *APO*. Nous avons choisi la méthode des auteurs décrite dans [AM01], dont une implémentation est présentée dans [AMH02] et téléchargeable sur le site relatif au livre [8].

3.2.4 Critère d'erreur : métrique $L^{2,1}$

Nous avons jusqu'alors détaillé l'algorithme de subdivision utilisé pour construire l'octree supportant l'*APO*. Dans cet algorithme une mesure de la variation du champ de normales de M est calculée dans chaque feuille. Cette variation V^N correspond à une erreur tolérée lors de la construction des feuilles échantillonnant M. Nous détaillons dans cette section la mesure de ce critère d'erreur. L'objectif de cette mesure est d'adapter aux variations fortes du champ de normales de M le niveau de détail de l'octree (voir 3.3 suivante).

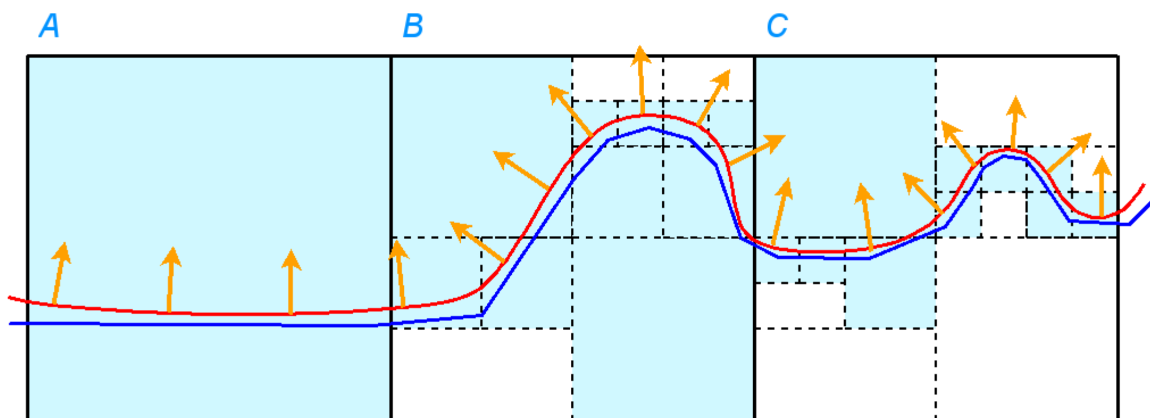


Fig. 3.3 – Les normales des triangles du maillage original M (rouge) contenues dans le nœud A étant quasiment identiques, le nœud n'est pas subdivisé. Dans les nœuds B et C, comme les normales présentent des variations fortes, la subdivision doit être effectuée jusqu'à obtenir un champ de normales homogène dans chaque feuille.

Une première idée est de mesurer l'angle maximum entre deux normales de la liste

[8] http://www.realtimerendering.com/int/

T^N. Un angle trop grand entre deux normales est un indicateur de non homogénéité à l'intérieur d'un nœud de l'octree. Cependant cette mesure triviale présente plusieurs inconvénients :

- **Complexité élevée :** Pour déterminer l'angle maximal entre deux normales il faut mesurer deux à deux les angles formés par chacun des couples de normales des triangles de T^N. Ceci se fait par un algorithme dont la complexité est de classe $\Theta(n^2)$, à cause du parcours imbriqué des triangles. Cette complexité est d'autant plus problématique que lors des premières subdivisions des nœuds de faible profondeur, les listes T^N indexent un grand nombre de triangles de M.
- **Perte de détails :** Une zone présentant une quantité de petites variations pourra ne pas être subdivisée, si les angles ne dépassent pas une seule fois l'angle maximal choisi.

Ces constats motivent la recherche d'un critère de mesure de variation plus précis, plus sensible aux variations et qui se calculera plus rapidement. Notre choix s'est porté sur une mesure de variation basée sur la métrique $L^{2,1}$ présentée dans l'article [CSAD04]. Cette métrique basée sur les normales permet de mesurer la variation à l'intérieur d'une zone, en l'occurence les triangles de M intersectés par les nœuds de l'octree, recensés dans la liste T^N. Comme pour la métrique $L^{2,1}$, la mesure est une somme des différences de chaque normale par rapport à la normale moyenne des triangles de T^N. La formule utilisée est donc :

$$V^N = \sum_{T_i \in T^N} ||N_{T_i} - N_{T^N}||^2$$

où N_{T^N} est la normale moyenne des triangles intersectant le nœud N, N_{T_i} est la normale de chacun des triangles. Ce calcul se fait en deux parcours, le premier pour calculer la moyenne N_{T^N}, le second pour faire la somme des différences. La complexité du calcul est donc de classe $\Theta(n)$.

Par rapport à la métrique $L^{2,1}$ présentée dans l'article original, nous ne prenons pas en compte la pondération par l'aire de chacun des triangles. Cette pondération minimise l'influence des petits triangles. Dans l'article [CSAD04], un des objectifs était de déterminer des zones homogènes sur la surface des maillages, un triangle de surface faible présentant une normale différente de triangles voisins plus vastes ne doit pas altérer cette mesure d'homogénéité. La pondération par l'aire permet de donner une plus grande importance aux grands triangles, ceux qui vont le plus correspondre aux zones homogènes, c'est à dire à la géométrie globale de l'objet.

Nous sommes justement intéressés par ces variations hautes fréquences dans les variations de normales à la surface de M, le maintien de la pondération par l'aire pourrait faire manquer des détails (voir figure 3.4).

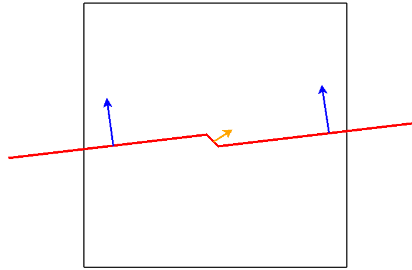


FIG. 3.4 – La suppression de la pondération par l'aire des triangles dans la métrique $L^{2,1}$ permet d'adapter la subdivision de l'octree aux détails les plus fins de M (normale orange sur notre figure).

3.2.4.1 Influence de la valeur de tolérance sur la subdivision

Un seuil de tolérance élevé fait intervenir l'arrêt des subdivisions tôt, l'octree est moins détaillé. L'*APO* construite a moins de feuilles et occupe donc un espace mémoire moindre, les texels couvrant le maillage m sont plus larges. Au contraire, un seuil bas permet de construire un octree détaillé avec des feuilles de profondeur élevée. La figure 3.5 montre un octree autour du maillage *bunny*, où l'on peut voir différentes subdivisions avec des tolérance différentes.

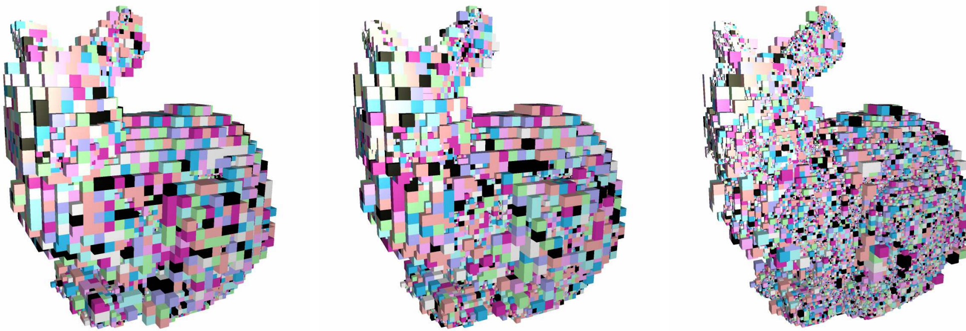


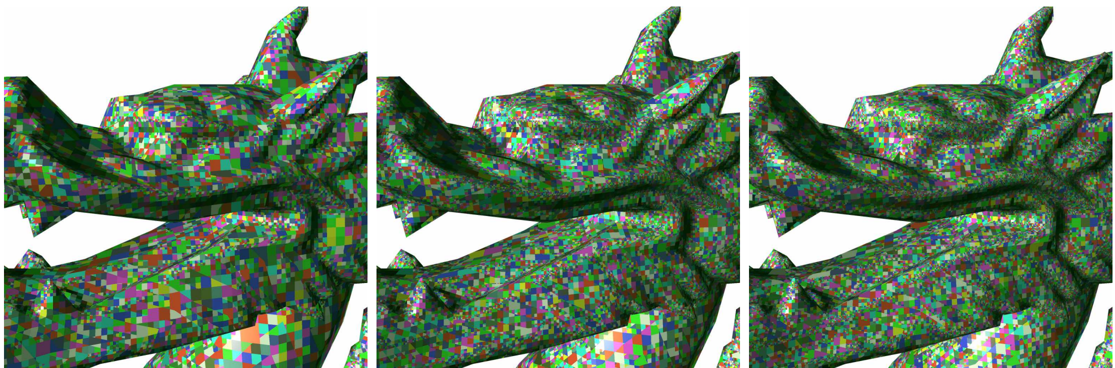
FIG. 3.5 – Subdivsion de l'octree autour du bunny. Les valeurs de tolérance sélectionnées sont respectivement 1.0, 0.5 et 0.05.

Les tableaux suivants montrent les statistiques des *APO* en termes de nombre de nœuds internes et de feuilles sur des maillages de tailles différentes, avec des seuils de tolérance variés. Pour ces tests, nous avons limité la profondeur maximale de l'octree au niveau 13. Nous présentons les résultats pour deux maillages, les modèles de dragon de Stanford et XYZ RGB. Dans l'annexe B se trouvent les statistiques concernant d'autres maillages.

Stanford Dragon : Le tableau de la figure 3.6 présente les tailles des octree construits lors de la subdivision avec différentes valeurs de tolérance pour la mesure de variation V^N. Nous constatons naturellement que plus la tolérance est faible, plus le nombre de cellules créés augmente. Cependant, il n'y pas de relation linéaire directe entre la valeur de la tolérance et le nombre de nœuds créés. On ne peut pas prédire exactement la taille des *APO* en fonction de la tolérance, cette relation est dépendante de chaque maillage, de la fréquence des détails à sa surface.

Stanford Dragon	Original : Simplifié :	874 414 triangles 3 000 triangles		
Octree	**Profondeur initiale :** 5 **Profondeur maximale :** 13			
Tolérance	0.5	0.3	0.1	0.05
Nombre total de cellules	216254	297450	615497	992585
Feuilles	168885	231089	473029	758177
Nœuds	47369	66361	142468	234408
Ratio Feuilles/Nœuds	3.56	3.48	3.32	3.23

(a) Composition de l'octree en fonction de la tolérance sur V^N.



(b) De gauche à droite les seuils de tolérance sont respectivement $0.5, 0.1$ et 0.05.

Fig. 3.6 – Résultats de la subdivision pour le *Stanford Dragon*.

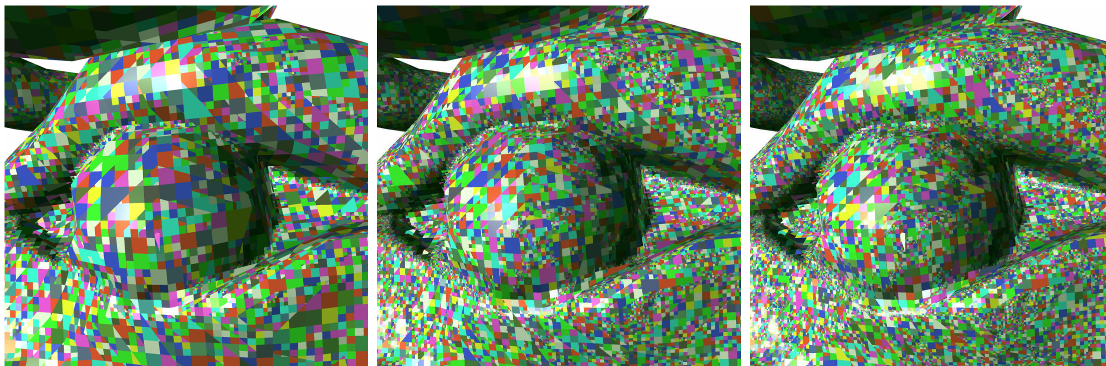
Nous constatons par contre que le nombre de feuilles par rapport au nombre de nœuds interne reste relativement constant, même si il décroît naturellement quand la tolérance diminue. La structure de l'octree pèse en effet de plus en plus lourd quand le niveau de détail et donc la profondeur augmentent. Les captures d'écran de la figure 3.6 mettent en évidence la différence de profondeur des feuilles en fonction de la tolérance sur V^N.

XYZRGB Dragon : Les constats que nous pouvons faire à propos des statistiques de création d'*APO* sur le maillage du dragon XYZRGB, présentés au tableau de la figure

3.7, sont les mêmes que pour le maillage plus léger du fragon de Stanford. Le nombre de cellules ne croît pas de manière proportionnelle avec la tolérance, et le taux de feuilles par rapport au nombre de nœuds est lui aussi relativement constant. Comme le maillage XYZRGB dragon est plus détaillé que le maillage précédemment décrit, le nombre de cellules est plus grand pour une valeur de tolérance sur V^N égale. Il faut en effet des cellules plus fines, et donc plus profondes, pour assurer un même taux de variation dans chaque feuille sur un maillage présentant des triangles plus fins et des détails à plus hautes fréquences.

XYZRGB Dragon	Original : Simplifié :	7 218 906 triangles 15 000 triangles		
Octree	**Profondeur initiale :** 5 **Profondeur maximale :** 13			
Tolérance	0.5	0.3	0.1	0.05
Nombre total de cellules	1664260	2433306	5444837	8673849
Feuilles	1297945	1887068	4173866	6612123
Nœuds	366315	546238	1270971	2061726
Ratio Feuilles/Nœuds	3.54	3.45	3.28	3.21

(a) Composition de l'octree en fonction de la tolérance sur V^N.



(b) De gauche à droite les seuils de tolérance sont respectivement $0.5, 0.1$ et 0.05.

FIG. 3.7 – Résultats de la subdivision pour le *Dragon XYZRGB*.

Les captures d'écran de la figure 3.7 montrent un gros plan sur l'oeil du dragon XYZRGB. Sur ces captures, ainsi que sur celles de la figure 3.6 on peut noter l'adaptativité de l'échantillonnage, les feuilles profondes se situant dans les zones à hautes fréquences des maillages, c'est à dire présentant des variations fortes dans le champ de normales.

Il est également intéressant de noter que pour les deux maillages présentés ici, ainsi que pour ceux dont les statistiques figurent dans l'annexe B, le rapport $Feuilles/Noeuds$

est quasiment identique pour chaque maillage. Si N_n est le nombre de nœuds internes et N_f le nombre de feuilles, nous avons

$$N_f \approx 3.5 \times N_n$$

Cette approximation est dépendante du niveau de détail du maillage M et des paramètres sélectionnés lors de la création de l'*APO*, mais l'ordre de grandeur est constant quelque soit le maillage sélectionné. Ce taux de feuilles par rapport au nombre de nœuds internes avait déjà été mis en évidence dans les travaux de [gDGPR02], où les auteurs notent qu'en pratique, les surfaces étant localement planaires, les nœuds ont environ 4 fils.

3.2.4.2 Gestion de l'écart entre m et M

Le critère de subdivision mesure la variation du champ de normales de M au sein de chacune des feuilles de l'octree. Cette mesure implique donc une contrainte sur la subdivision adaptative. Il faut en effet que chaque feuille intersecte des triangles de M pour pouvoir effectuer cette mesure de variation. Or il peut arriver que lors de la subdivision, les feuilles de l'octree qui sont conservées, c'est à dire celles qui sont autour du maillage simplifié m, n'intersectent plus aucun triangle du maillage original. Ce cas de figure est illustré par la figure 3.8.

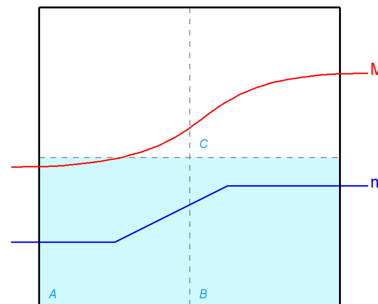


FIG. 3.8 – Lors de la subdivision de la feuille, seuls seront conservés les deux fils A et B intersectant m. A l'étape de subdivision suivante, la mesure de variation ne pourra pas être effectuée pour la feuille B qui n'intersecte pas le maillage M.

Lorsque ce cas de figure se produit, la subdivision de la feuille ne peut avoir lieu, et des détails peuvent donc être perdus dans l'*APO*. Les images présentées dans la figure 3.9 illustre les effets sur le rendu induits par cette perte de détails.

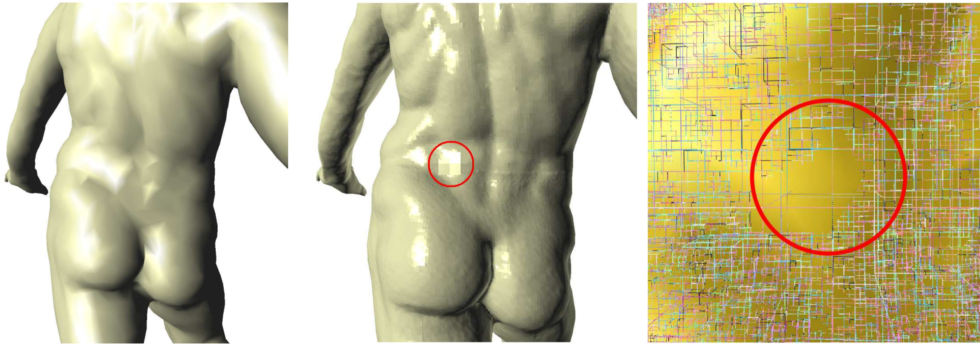


Fig. 3.9 – Maillage simplifié (gauche), le même avec texture APO (milieu), nœuds de l'octree intersectant le maillage (droite). Certaines zones, comme celle encerclée en rouge, sont couvertes par des nœuds très larges, ce qui est dû à l'arrêt prématuré de la subdivision de l'APO causé par une trop grande distance entre le maillage original et sa version simplifiée. Le gros plan de l'image de droite montre le *trou* provoqué par l'arrêt prématuré des subdivisions.

Ce problème survient surtout lorsque la profondeur des feuilles de l'octree est élevée, et que l'écart entre les maillages M et m est significatif par rapport à la taille du nœud. Pour éviter que ce cas se produise fréquemment, ou qu'il se produise sur les feuilles de faible profondeur, ce qui est nuisible pour la qualité de rendu, il faut imposer que le maillage m soit une bonne simplification du maillage original M. Il faut plus précisément que le maillage m ne soit pas trop éloigné de M pour que chaque feuille ait toujours une liste T^N non vide qui lui permette de mesurer le critère d'erreur. Par exemple, pour une APO de profondeur maximale p, il serait souhaitable que la distance entre les deux maillages (une fois redimensionnés dans le cube unitaire) soit au maximum de 2^{-p}. Cette distance correspond à la largeur d'une feuille de profondeur p. Cependant, même avec cette condition, le cas de figure décrit par la figure 3.8 peut se produire, en effet une subdivision pourrait toujours venir découper entre les deux maillages même si ceux-ci sont très proches.

Une solution consiste à lancer un rayon lorsque la liste T^N vide, pour identifier la zone de M qui sera échantillonnée par la feuille. On peut alors mesurer la variation V^N dans la première cellule intersectée par le rayon. Le principe de cette méthode est illustré par la figure 3.10. Cette méthode permet de corriger le problème de distance entre les deux maillages de V^N en donnant à chaque étape la zone d'échantillonnage dans laquelle la mesure peut être effectuée. Pour adapter cette zone d'échantillonnage à la largeur de chaque feuille, la cellule intersectée par le rayon est également subdivisée. Ceci implique que des nœuds n'intersectant pas le maillage m sont créés, ils doivent être supprimés à la fin de la subdivision.

Cette méthode permet de toujours avoir des données sur lesquelles mesurer la varia-

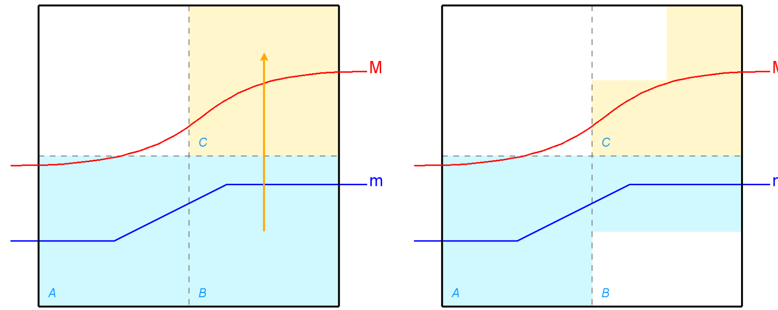


FIG. 3.10 – La feuille B a une liste T^N vide. Pour tester si la subdivision doit être effectuée, la zone d'échantillonnage de B est déterminée par le lancer du rayon. La variation V^N est mesurée dans la cellule C. Dans le cas où la subdivision doit être effectuée, les deux cellules B et C sont subdivisées.

tion V^N. Cependant, elle rend la création des textures très longue. En effet le lancer de rayon est une opération coûteuse qui doit être répétée pour chacun des fils d'un nœud à partir du moment où le manque de données apparaît (si un nœud n'intersecte pas M, aucun de ses fils ne l'intersectera non plus, la méthode du lancer de rayons devant donc être appliquée à tous ces fils). De plus la solution requiert un parcours d'octree supplémentaire pour supprimer les feuilles n'intersectant pas m.

Nous avons donc choisi d'implémenter une méthode approximative qui, utilisée sur des maillages bien simplifiés, produit des textures de détails très convenables. Lorsque un nœud manque de données, il n'est tout simplement plus subdivisé. Cette approche laisse quelques erreurs présentes dans les textures de détails, mais rend la création des textures de 5 à 10 fois plus rapides. Les artefacts, comme ceux présentés en figure 3.9, sont peu visibles lors du rendu, car ils sont peu nombreux même sur des maillages drastiquement simplifiés[9]. Plus la simplification est proche du maillage original, moins ces artefacts sont nombreux.

Ne traitant pas dans cette thèse de la simplification des maillages, nous ne pouvons pas assurer que ces artefacts ne seront pas présents dans les *APO* construites. Pour nos expérimentations, nous avons généré des maillages simplifiés avec une simlpification basée sur l'erreur quadratique [MP97]. Une première sécurité pour éviter la perte potentielle de détails dans l'*APO* serait de paramétrer les algorithmes de simplification pour introduire la prise en compte de la distance maximale entre les maillages, en utilisant par exemple une simplification basée sur les travaux de [DZ91] ou de [CVM+96]. On pourrait également créer un algorithme qui couplerait la simplification à la création de l'octree, et ainsi contraindre la distance maximale entre les deux maillages. Les travaux présentés

[9] La plupart des maillages utilisés pour les figures de ce manuscrit ont été réduits de quelques centaines de milliers, voire quelques millions de triangles à environ 15 ou 20 mille triangles

dans [BHP06], dans lequel une simplification de maillages basée sur octree sont décrits, peuvent servir de base à la mise en place de cet algorithme couplant simplification et création de l'*APO*.

3.3 Assignation des normales aux feuilles de l'octree

La subdivision décrite dans la section 3.2 précédente permet de construire la structure d'octree de l'*APO*. Cette structure permet d'obtenir une couverture adaptative du maillage original M, et donc d'adapter la subdivision au niveau de détails du maillage M. Dans cette section nous détaillons comment les normales représentatives assignées aux feuilles de l'octree sont récupérées dans le champ de normales de M. Nous détaillerons dans un premier temps une méthode exacte par lancer de rayons, puis une méthode plus rapide mais moins précise par simple moyennage des normales à l'intérieur des feuilles.

Dans les deux cas, le calcul de la normale représentative de chaque feuille s'appuie sur la structure de l'octree pour à la fois pour localiser et accélérer l'échantillonnage. L'utilisation de structure spatiale pour la création de texture d'apparence était également présentée dans [CMSR98, CMR+99], les auteurs créant une division régulière de l'espace autour des maillages pour identifier les échantillons à affecter à chaque texel de leurs textures. Cependant, les points sur M correspondants aux texels n'étaient pas obtenus par lancer de rayon, mais en cherchant les points les plus proches. Dans [SGG+00] les auteurs font remarquer que l'échantillonnage par lancer de rayons, qu'ils nomment *normal shooting* est plus précis et présente moins de discontinuités que le choix du point le plus proche.

Dans les sections suivantes, nous présentons les méthodes de calcul de la normale représentative qui sont appliquées à chacune des feuilles de l'octree. L'algorithme général d'échantillonnage pour l'ensemble des feuilles se fait par un parcours récursif ou itératif de l'octree, le calcul étant lancé chaque fois qu'une feuille est rencontrée lors du parcours.

3.3.1 Echantillonnage des normales par lancer de rayons

Depuis chaque feuille de l'octree, un rayon dont l'origine est le centre de la feuille est lancé, il intersecte le maillage M, la normale à cette intersection est affectée à la feuille. Plus précisément, sa contribtion est ajoutée à la feuille, car en fait un rayon R_i est lancé pour chaque triangle $t_i \in t^N$. Il est en effet nécessaire de lancer un rayon pour chaque triangle du maillage m intersecté par la feuille dont on calcule l'échantillon, pour prendre en compte la contribution locale de toutes les normales de M, dans le cas où la feuille intersecte plusieurs triangles de m. Si on ne lance qu'un seul rayon, des discontinuités apparaîtront dans l'échantillonnage (voir figure 3.11).

La recherche de la plus proche intersection entre le rayon et M est une opération coûteuse, qui est acclérée par l'utilisation de l'octree comme structure de partitionnement spatial. Grâce aux listes T^N qui indexent les triangles de M intersectés par chaque feuille, nous avons une information locale sur les triangles de M situés proches de l'origine du rayon. Les tests d'intersection se font donc entre le rayon et les triangles des listes T^N des nœuds traversés par le rayon. L'algorithme 4 décrit les étapes successives du lancer de rayon, étapes qui seront chacune détaillées dans les sections suivantes.

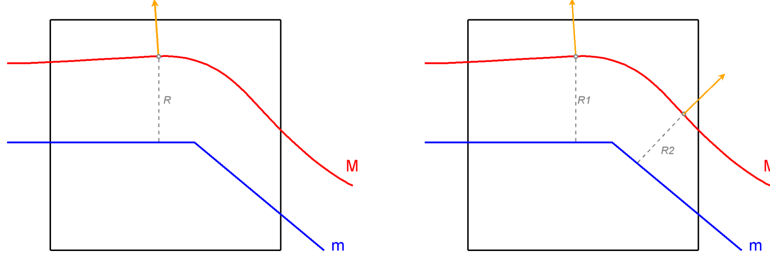


FIG. 3.11 – En lançant un rayon par triangle, la normale échantillonnée, qui sera la moyenne des normales trouvées, sera plus juste car elle prendra en compte plus de détails de M.

Données : Feuille F de l'octree, Maillage haute résolution M
Normale $\vec{N} \leftarrow (0,0,0)$;
pour chaque *triangle* $t_i \in t^N$ *de* F **faire**
    déterminer le rayon $\vec{R}(Origine, Direction)$;
    $L_N \leftarrow$ liste des nœuds de l'octree intersectés par $\vec{R}$;
    Trier L_N selon distance des cellules à *Origine* ;
    Position $P_{inte} \leftarrow$ plus proche intersection entre et $\vec{R}$ et M dans L_N ;
    $\vec{N} \leftarrow$`Normale(`P_{inte}, M`)` + $\vec{N}$;
`Normaliser(`$\vec{N}$`);`

Algorithme 4 : Algorithme d'échantillonnage par lancer de rayons depuis une feuille.

3.3.1.1 Choix de l'origine et de la direction du rayon

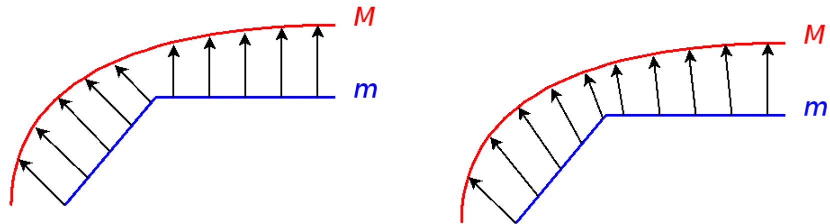
La première étape est de déterminer, étant donné un nœud N et un triangle t_i de sa liste t^N quel est le rayon qui servira à trouver l'échantillon correspondant sur M. Si l'origine peut trivialement être le centre du nœud, la détermination de la direction est plus problématique. Le choix se porte entre deux options :

- **Normale du triangle :** Cette direction est obtenue directement en lisant les informations contenues dans la structure du maillage, ce qui rend la phase d'échantillonnage plus rapide.
- **Normale interpolée du triangle :** Direction obtenue en prenant la normale interpolée du triangle au point le plus proche de l'origine du rayon, c'est à dire du centre de N.

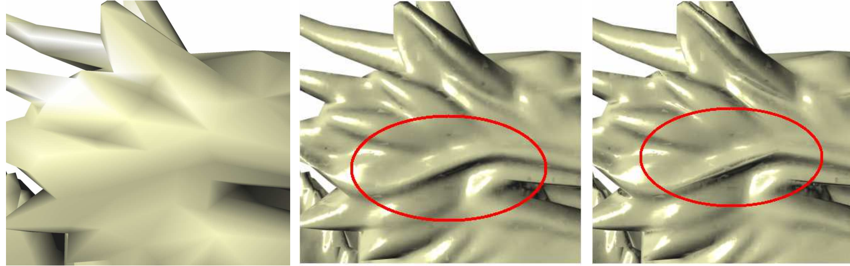
Le choix de la normale interpolée permet un échantillonnage plus uniforme de M et donne de meilleurs résultats, comme le notent les auteurs de [SGG+00] dans leurs travaux. Le schéma de la figure 3.12 illustre la cause du sous échantillonnage tandis que les captures d'écran montrent les différences de qualité entre les deux directions de rayon

choisie. La détermination du rayon utilisée pour le test d'intersection se fait donc par les deux étapes suivantes :

1. Projection du centre de la cellule sur le triangle t_i
2. Interpolation de la normale au point trouvé



(a) Normales des triangles vs. normales interpolées



(b) Différences sur l'échantillonnage : à gauche le maillage simplifié, au milieu l'APO est calculée avec les normales interpolées, à droite avec les normales des triangles. Notez que l'échantillonnage est de moins bonne qualité.

FIG. 3.12 – Direction des rayons.

Comme l'intersection entre le rayon et le maillage M peut se trouver devant ou derrière le maillage m, le rayon est lancé dans les directions positives et négatives. L'intersection que l'on cherche alors est l'intersection entre M et le rayon la plus proche de l'origine. La normale représentative qui sera assignée à la feuille est donc issue d'un point voisin de l'origine, et pour cette raison il n'est pas nécessaire de chercher des intersections sur toute la longueur du rayon, une distance maximale d_{max} est définie. La recherche se transforme alors en la recherche de la plus proche intersection sur le segment $[-d_{max}, d_{max}]$ centré sur le centre de la cellule.

La valeur d_{max} influe de plusieurs manières sur le lancer de rayon :

- Une valeur trop grande ralentira l'exécution de la phase de lancer de rayon, en indiquant un trop grand nombre de nœuds intersectés par le rayon, et donc en augmentant le nombre de tests d'intersection entre le rayon et les triangles de M.

- Une valeur trop petite risque de faire manquer l'intersection, si la distance entre l'origine du rayon et le maillage est supérieure à d_{max}.

Assigner à d_{max} la plus grande distance entre M et m serait la solution optimale, mais il faudrait calculer cette distance au préalable. Nous avons dans nos expérimentations utilisé la largeur d'un nœud de profondeur 5. Les détails fins se trouvant plutôt dans les feuilles de profondeur plus élevée, cette valeur de d_{max} nous assure de trouver les intersections sans sacrifier aux performances. La figure 3.13 résume la méthode que nous venons de décrire.

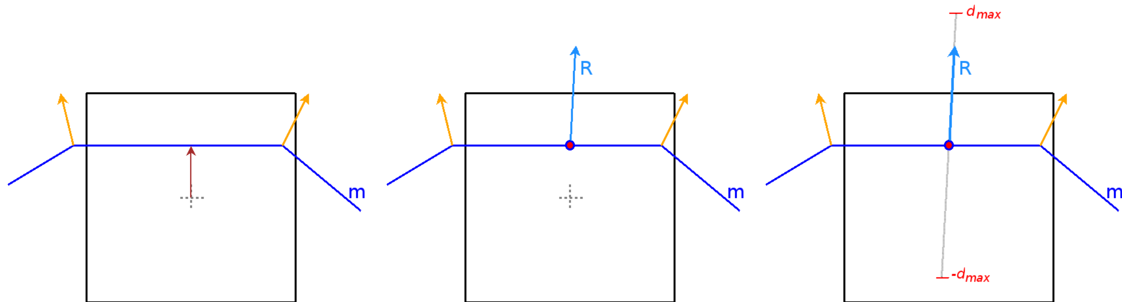


Fig. 3.13 – Le centre de la cellule est projeté sur le triangle pour lequel on va lancer le rayon. Au point le plus proche trouvé on interpole la normale pour obtenir la direction du rayon. L'intersection sera cherchée sur le segment bornée entre d_{max} et $-d_{max}$.

3.3.1.2 Liste des nœuds intersectés par le rayon

Une fois que le rayon à lancer a été déterminé, l'intersection entre ce dernier et le maillage M doit être trouvée. L'octree qui a été construit contient pour chaque nœud et feuille la liste des triangles de M qu'ils intersectent, cette information de localisation permet de tirer avantage de la structuration spatiale, et ainsi d'accélérer la recherche de l'intersection. Seuls les triangles qui sont référencés par la liste des cellules qui sont traversées par le rayon sont testés.

Nous décrivons ci-dessous comment obtenir puis trier cette liste de cellules de façon à faciliter la recherche d'intersection. Nous avons vu à la section 3.2.1 page 44 que le maillage M n'était pas obligatoirement entièrement contenu dans l'enveloppe formée par les feuilles de l'octree. L'intersection entre les rayons et M pourra donc se trouver dans les nœuds internes de l'octree. Pour cette raison la liste des cellules dans laquelle la recherche de l'intersection est composée des feuilles intersectant le segment, mais également des nœuds internes dont la liste T^N n'est pas vide.

Recherche des nœuds intersectés : L'algorithme récursif 5 simple permet de créer la liste des cellules intersectées par le segment défini par le rayon et la distance maximale de recherche. Il est appelé sur la racine de l'octree, la liste de cellules est construite lors d'un parcours en profondeur de l'octree. A la fin de l'exécution de cet algorithme, la liste L_N contient toutes les cellules de l'octree qui intersectent le rayon.

Données : Rayon $\vec{R}$
Données : Distance d_{max}
Résultat : Liste L_N des cellules intersectées par le segment

```
procedure cellulesIntersecteesParRayon(nœud N, R⃗, d_max, L_N) début
    si IntersectionBoiteSegment(N,Segment(R⃗,d_max)) alors
        si Liste T^N de N non vide alors
            AjouteAListe(N,L_N) ;
        pour i de 1 à 8 faire
            si Fils(i,N) non vide alors
                cellulesIntersecteesParRayon(Fils(i,N),R⃗, d_max, L_N) ;
fin
```

Algorithme 5 : Algorithme de recherche des cellules de l'octree intersectées par un rayon.

Dans cet algorithme il est primordial d'avoir un test d'intersection rayon/boîte très rapide pour conserver des performances optimales. Nous avons dans un premier temps testé l'algorithme présenté dans [Woo90], qui donne une réponse booléenne sur l'intersection ou non de la boîte par le rayon, ce qui obligeait à opérer un second test de distance pour vérifier si la boîte se trouvait dans le segment $[-d_{max}, d_{max}]$. L'implémentation du test d'intersection proposée dans [WBMS05a] et [WBMS05b] s'est révélée plus efficace, incluant en outre l'intervalle d'intersection. Cette dernière solution teste donc directement l'intersection avec un intervalle, ce qui nous libère de l'opération de vérification de distance des cellules.

Tri de la liste L_N : Pour accélérer la recherche de la plus proche intersection entre le rayon et M, outre l'identification des cellules de l'octree intersectées par les différents rayons, nous avons également mis en place un tri de ces cellules, de la plus proche de l'origine à la plus lointaine. En utilisant ce tri la recherche pourra être prématurément stoppée lorsque l'on sera sûr de ne plus trouver de plus proche intersection dans les cellules plus lointaines.

Pour trier les cellules intersectées par le rayon, la distance entre l'origine et leur première intersection avec le rayon est utilisée (le rayon intersecte deux fois chaque cellule, une fois en *rentrant*, une fois en *sortant*). Si deux cellules sont à la même distance

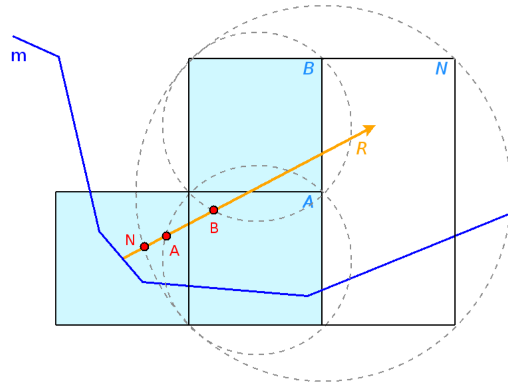


Fig. 3.14 – Le nœud N, dont l'intersection de la sphère englobante avec le rayon se trouve avant celles des cellules filles A et B, sera traîté avant ces dernières dans l'algorithme de recherche.

de l'origine, celle qui a la profondeur la plus élevée est classée avant. Comme le calcul des coordonnées d'intersection entre un cube et un rayon est complexe, nous avons choisi de faire une approximation qui permet d'accélérer le tri, en calculant la distance entre l'origine du rayon et la sphère englobante de chaque nœud, le calcul d'intersection entre une sphère et un rayon étant plus rapide que celui entre un rayon et un cube. Nous avons choisi l'algorithme présenté dans le livre [Gla89]. Si cette approximation peut rendre un tri incorrect, un nœud interne dont la sphère englobante est plus grande que celles de ses fils pourra être rangé avant ses fils dans la liste (voir figure 3.14), elle assure toutefois que la plus proche intersection sera trouvée. Les quelques erreurs de tri ne pénalisent pas les performances, puisque l'utilisation de sphères englobantes rend la recherche plus rapide que le calcul des intersections rayons/cubes.

3.3.1.3 Recherche de la plus proche intersection

La plus proche intersection entre le rayon et les triangles de M est dans un premier temps recherchée dans la liste T^N de la feuille dont est issue le rayon. La liste L_N n'est calculée que si aucune intersection n'est trouvée dans cette feuille originale. Dans le cas où la recherche dans la liste liste L_N est nécessaire, les nœuds sont parcourus dans l'ordre du tri, la recherche étant arrêtée dès que l'on est assuré de ne plus pouvoir trouver une plus proche intersection dans les cellules qui n'ont pas encore été traitées. En effet si une intersection déjà trouvée est plus proche de l'origine du rayon que l'intersection de la sphère englobante de la cellule suivante, il n'est pas nécessaire de parcourir les triangles des dernières cellules, elles ne pourront pas contenir d'intersection plus proche. L'algorithme 6 suivant décrit ce processus de recherche.

Cet algorithme fait appel à un test d'intersection entre un rayon et un triangle, test qui se doit d'être le plus efficace possible pour ne pas affecter les performances de

Données : Feuille F de l'octree
Données : Maillage haute résolution M
$Normale\ \vec{N} \leftarrow (0, 0, 0)$;
pour chaque *triangle* $t_i \in t^N$ **faire**
  déterminer le rayon $\vec{R}(Origine, Direction)$;
  Point $P_{inte} \leftarrow$ plus proche intersection entre $\vec{R}$ et les triangles de T^N ;
  si $P_{inte} == \emptyset$ **alors**
    $L_N \leftarrow$ cellules de l'octree traversées par $\vec{R}$;
    Trier L_N selon distance des cellules à $Origine$;
    Point $P_{inte} \leftarrow$ plus proche intersection entre et $\vec{R}$ et M dans L_N ;
    si $P_{inte} == \emptyset$ **alors**
      Point $P_{inte} \leftarrow$ point le plus proche de $Origine$ sur triangles de T^N ;
  ajouter `Normale(`P_{inte}, M`)` à $\vec{N}$;
`Normaliser(`$\vec{N}$`)`;

Algorithme 6 : Algorithme de récupération de la normale pour une feuille de l'octree.

recherche. Nous avons testé les méthodes de [GD03] et de [AM01]. Bien que les différences en termes de temps de création des APO se soient révélées très faibles, nous avons constatés un léger avantage à celle de [AM01]. Plusieurs implémentations de sa méthode, présentant un ordonnancement différent des étapes, sont disponibles sur internet[10]. Nous avons testé ces différentes méthodes, mais l'implémentation originale de l'article [AM01] reste la plus efficace. Comme le fait remarquer l'auteur, ces variantes de l'implémentation sont plus ou moins efficaces selon l'architecture et le processeur utilisé. Dans toutes les expérimentations que nous avons menées, la différence de performance entre les variantes étaient infimes.

Il arrive que dans certains cas il n'y ait pas d'intersection entre le rayon et M. Lorsque ce cas se produit, la recherche s'effectue une deuxième fois dans la feuille originale, puis dans la liste L_N si nécessaire, mais en cherchant non plus l'intersection sur le rayon, mais le point le plus proche de l'origine sur chaque triangle. Cette alternative avait également été proposée par les auteurs de [SGG+00].

3.3.2 Echantillonnage par moyennage des normales des feuilles

Nous avons vu plus tôt que les nœuds de l'octree contiennent une liste référençant les triangles de M qu'ils intersectent. Grâce à cette liste, nous savons pour chaque point P du maillage m quels sont les triangles de M voisins, c'est à dire ceux intersectés par la feuille qui contient P. Pour assigner un échantillon représentatif de M à chaque feuille, on peut utiliser les triangles de la liste T^N plutôt que d'effectuer un lancer de rayons. Comme chaque feuille peut-être intersectée par plusieurs triangles, la contribution de

[10] http://www.cs.lth.se/home/Tomas_Akenine_Moller/raytri/

chacun d'eux est prise en compte en calculant la normale moyenne des normales des triangles appartenant à T^N. Cette méthode d'échantillonnage est très rapide car elle ne nécessite que très peu d'opérations pour obtenir la normale représentative de chaque feuille.

Cependant cette méthode peut laisser des feuilles non échantillonnées. En effet si une feuille a une liste T^N vide, on ne pourra pas calculer de normale moyenne. Ce problème est le même que celui soulevé lors de la mesure de V^N lors de la subdivision de l'octree (voir 3.2.4.2 page 54). Plus l'octree sera détaillé, plus la fréquence de ce manque de triangles intersectant les feuilles sera grande, là où la méthode du lancer de rayon assure de trouver une normale pour chaque feuille.

On peut observer sur la figure 3.15 le même maillage avec deux *APO* utilisées, une obtenue par lancer de rayons, l'autre par la méthode que nous venons de décrire. Il est intéressant de noter, outre les erreurs d'échantillonnage déjà décrites, que cette méthode de moyennage est moins précise que celle du lancer de rayons. Elle présente cependant l'avantage d'être plus rapide, tout en offrant une qualité de conservation des détails convenables.

Fig. 3.15 – Comparaison des méthodes d'échantillonnage. A gauche le maillage simplifié, au centre le même avec utilisation de l'*APO* obtenue par lancer de rayons, et à droite obtenue par moyennage des normales des triangles. On observe la qualité moindre d'échantillonnage de cette dernière méthode et les erreurs d'échantillonnage.

Il est possible de combiner les deux méthodes pour obtenir de meilleures performances pour le temps de calcul des *APO*. Lors du parcours récursif de l'octree, on peut décider d'utiliser le moyennage des normales si la feuille intersecte des triangles de M, ou le lancer de rayons dans le cas contraire. Chaque feuille est ainsi assurée de récupérer une normale représentative. Cette méthode n'est à employer que si l'on cherche à améliorer le temps de création des textures *APO*, qui est comme on le montrera relativement faible (de l'ordre de quelques minutes), le lancer de rayons offrant la meilleure qualité

d'échantillonnage.

3.3.2.1 Calcul des normales pour les nœuds internes

Les algorithmes précédents présentent la méthode permettant d'affecter une normale à chacune des feuilles de l'octree. Les nœuds internes n'ont à ce stade pas de normales. Nous avons décidé d'assigner aux nœuds la normale moyenne de leurs fils, en utilisant un processus remontant des feuilles vers la racine. Ce moyennage peut-être vu comme un niveau de détails sur la texture APO, c'est à dire un MIP-mapping interne de l'APO. Le simple moyennage des normales n'est pas un filtrage totalement correct. Des filtrages plus fins peuvent être mis en place, comme celui présenté dans [HSRG07] ou [Tok05].

Nous verrons dans la section 4.3.2 page 97 comment utiliser ces valeurs moyennées aux nœuds internes pour effectuer un rendu adaptatif en fonction de la distance de l'objet à l'écran.

3.4 Encodage de l'APO en texture 2D

Cette partie aborde la dernière étape de la construction des *APO*, c'est à dire l'encodage de la structure de l'octree dans un format exploitable par les cartes graphiques. En effet nous avons vu dans la partie 2.3.4.1 que les octree-textures ne sont pas nativement supportées par les *GPU* comme le sont les textures 2D ou 3D. Le décodage de l'*APO* sur *GPU* par un *fragment shader* est décrit dans la partie 4 page 79.

Nous proposons un encodage différent, plus compact, des précédentes implémentations d'octree-texture *GPU* présentées dans [LHN05] et [KLS+05, LSK+06]. Nous déclinons cet encodage en deux versions différentes, une première très compacte ne stockant pas les normales aux nœuds, une seconde contenant toutes les informations de l'octree créé. Nous détaillons également toutes les informations nécessaires au parcours de l'octree par le *GPU* qui sont écrites aux nœuds lors de cet encodage.

3.4.1 Organisation en largeur de l'octree

Avant d'être sauvegardé dans une texture 2D, l'octree est dans un premier temps écrit dans un tableau 1D, dans lequel chaque nœud et feuille sont représentés par une structure sauvegardant les informations de ces derniers. Dans le but de rendre l'octree le plus compact possible, l'octree est rangé en largeur d'abord dans le tableau. Ainsi tous les fils d'un nœud sont voisins dans le tableau, et un seul pointeur vers le premier est suffisant pour y accéder. Cet encodage est classique et a été présenté dans plusieurs travaux comme ceux de [HW91].

Toujours dans le souci de créér une texture la plus petite possible, nous nous inspirons des travaux de [BD02] qui présentaient des octree textures sur *CPU*, et nous ne stockons dans le tableau que les nœuds non vides de l'octree. La figure 3.16 illustre ce tri en largeur d'abord ainsi que le stockage des nœuds non vides.

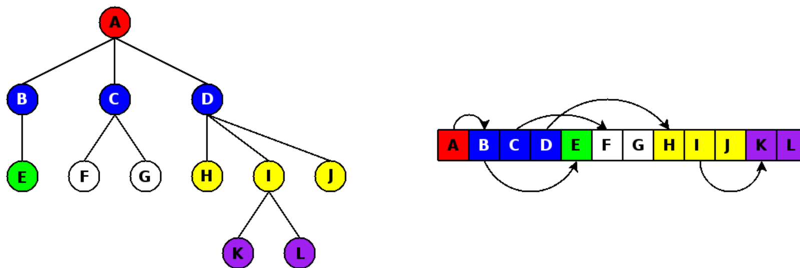


FIG. 3.16 – Conversion de la structure d'arbre de l'octree dans un tableau 1D, en largeur d'abord. Lors de ce classement, seuls les fils existants d'un nœud sont sauvegardés dans le tableau.

Dans les sections suivantes nous présentons les différentes structures employées pour

enregistrer les nœuds dans le tableau 1D. Les processeurs GPU n'étant capables de manipuler que des textures, il faut encoder l'octree dans une texture qui sera exploitable par la carte graphique. Les textures 1D étant limitées en taille, et l'accès aux textures 2D étant optimisé matériellement, nous choisissons de couper le tableau 1D sauvegardant l'octree en plusieurs lignes, de façon à former une texture 2D qui encodera la structure et les valeurs de l'*APO*. L'utilisation des textures implique une autre contrainte : les tailles des structures représentant les nœuds de l'octree doivent être alignées sur les tailles des pixels, c'est à dire 3 ou 4 octets selon que l'on utilise des images RGB ou RGBA.

3.4.2 Encodage sans valeurs MIP-map

Pour stocker un nœud ou une feuille dans le tableau, il faut enregistrer toutes les informations que nécessitera le GPU pour le parcours de l'octree lors du rendu. Ces informations et leurs encodages respectifs, illustrés par le schéma de la figure 3.17, sont les suivantes :

- **Nœud :** Un nœud doit contenir les informations concernant ses fils qui sont utilisées lors du parcours de l'octree par le GPU. Ces informations se résument au pointeur vers le premier fils, et comme nous ne stockons que les fils non vides, un ensemble de marqueurs pour indiquer quels sont les fils existants. Comme dans les travaux de [BD02], ces indicateurs sont écrits dans un ensemble de 8 bits, chacun correspondant au fils de son rang, le bit de poids faible étant associé au premier fils. Un bit à 1 signifiera que le fils correspondant existe. Ce masque d'existence des fils étant stocké sur un octet, il reste 3 octets qui sont utilisés pour sauvegarder le pointeur vers le premier fils. Sur ces 3 octets, il est possible d'indexer jusqu'à 2^{24} valeurs, soit 16 millions d'indices. Cette valeur est suffisante pour stocker les indices sur une texture de dimension 4096×4096, ce qui permet comme nous allons le voir plus tard de créer des octree ayant une profondeur maximale de niveau 12 ou 13, avec un bon niveau de détail. Pour pouvoir indexer les nœuds sur des textures plus grandes, nous ne stockons pas le pointeur absolu vers le premier fils, mais un offset local depuis la position du nœud courant.
- **Feuille :** Une feuille de l'octree ne contient que la normale représentative qui a été calculée. Cette normale, exprimée dans le repère monde, est quantifiée sur 3 octets, un par composante. Le dernier octet est utilisé comme masque d'existence des fils, comme pour les nœuds. La distinction entre feuilles et nœuds dans l'octree se fait grâce à ce masque, les feuilles n'ayant pas de fils, leurs masques ont pour valeur 0 tandis que ceux des nœuds ont une valeur non nulle.

La conversion de l'octree en tableau 1D se fait en deux étapes. La première consiste en la numérotation de chacun des nœuds et feuilles lors d'un parcours en largeur d'abord de l'octree, parcours illustré par l'algorithme 7. Comme l'arbre n'est pas complet, il est impossible de deviner l'index de chaque nœud, et donc de calculer l'offset depuis chaque nœud vers son premier fils. Le premier parcours de l'octree permet donc d'identifier la

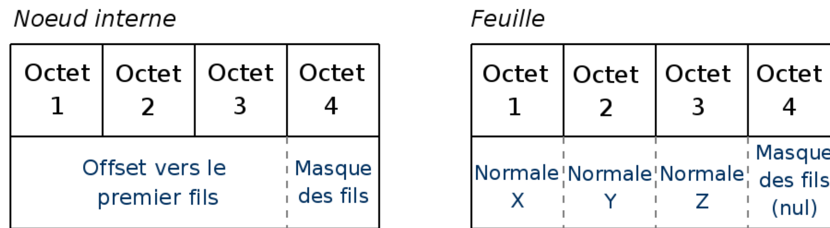


FIG. 3.17 – Encodage sur 4 octets des nœuds et feuilles de l'octree.

position dans le tableau de chacun des nœuds de l'octree, le second sert à placer la structure sur quatre octets représentant ces cellules à leus positions respectives dans le tableau.

Données : Octree Oct
entier *indice* $\leftarrow 0$;
file *fileNoeuds* ;
`enfiler`(*fileNoeud*, *`racine`*(Oct)) ;

tant que $fileNoeud \neq \emptyset$ **faire**
  Nœud $N \leftarrow$ `debut`(*fileNoeud*) ;
  `defiler`(*fileNoeud*) ;
  `affecteIndice`(N,*indice*) ;
  $index \leftarrow indice + 1$;
  pour *i de 1 à 8* **faire**
    si *`Fils`*(N,i) *existe* **alors**
      `enfiler`(*fileNoeud*,*`Fils`*(N,i))

Algorithme 7 : Algorithme de numérotation des nœuds pour l'encodage de l'octree sans sauvegarde des valeurs MIP-map.

Lors du deuxième parcours, l'initialisation du masque d'existence des fils est effectuée. Les bits sont positionnés ou non selon que les fils sont présents ou pas. L'offset du nœud parent vers son premier fils est écrit sur les 3 octets restants. Les quatre octets sont inscrits à l'indice correspondant au nœud dans le tableau. Pour le traitement d'une feuille, le masque d'exsitence est initialisé à 0, la normale est inscrite sur trois octets. La quantification sur trois octets de la normale est décrite dans la partie 3.4.2.1 suivante.

3.4.2.1 Quantification sur trois octets des normales

Les normales associées à chaque feuille de l'octree sont quantifiées sur trois octets dans le tableau 1D. Nous utilisons une écriture directe, chaque composante étant encodée sur un octet. Les vecteurs normales exprimés dans le repère monde et normalisés vérifient

l'assertion suivante :

$$\vec{N}(x, y, z) \in [-1, 1]^3$$

Chaque composante doit être encodée sur un canal RGB, c'est à dire une valeur positive appartenant à l'intervalle $[0, 255]$. Pour effectuer la translation de l'intervalle réel $[-1, 1]$ vers l'intervalle entier $[0, 255]$, chaque composante se voit appliquer l'opération suivante :

$$(\vec{N}(x, y, z) + 1.0) \times 127.5$$

Chacune des composantes se retrouve ainsi avec une valeur positive entre 0 et 255, cette valeur est convertie sur 8 bits et écrite dans le canal correspondant.

Cette quantification, bien qu'intuitive et rapide à implémenter, présente quelques défauts. En effet la quantification choisie n'est pas isotrope, l'angle entre les différentes normales n'étant pas uniforme. Cependant, un tel encodage sera très rapidement déquantifié pendant le rendu, en ne nécessitant que très peu d'opérations mathématiques basiques, et en ne faisant appel à aucune table d'indexation.

Bien que cette quantification non uniforme sur 24 bits des normales ne présente pas d'artefacts visuels, il serait possible de l'améliorer en implémentant par exemple une quantification classique, comme celle présentée dans [Dee95]. L'auteur présente un partitionnement de l'espace des directions. Il assure qu'une distribution de 100000 normales dans la sphère unitaire est suffisante pour éviter les artefacts, ceci correspond à un angle maximal de 0.01 radians. Avec notre quantification, l'angle maximal entre deux normales est inférieur à cette limite, car il y a plus de 100000 normales encodées sur 24 octets. L'angle maximal est environ de 0.003 radians.

3.4.3 Encodage avec valeurs MIP-map

L'encodage de l'octree que nous venons de présenter ne sauvegarde pas les normales moyennées qui ont été calculées aux nœuds. Pour rajouter ces normales, qui peuvent être utilisées pour du rendu adaptatif, la manière la plus simple est de créer un deuxième tableau qui contient à l'indice correspondant de chaque nœud la normale associée. Cette méthode est cependant trop coûteuse en terme d'occupation mémoire, puisqu'elle implique que l'espace occupé pour les APO est doublé alors que le deuxième texel est inutile pour les feuilles. Nous présentons donc un encodage qui ne double l'espace occupé que pour les nœuds internes de l'octree.

Pour sauvegarder les normales filtrées des nœuds internes, il faut rajouter 3 octets dans la structure représentative de ces derniers, ce qui implique l'utilisation d'un deuxième texel dans l'optique d'une sauvegarde sous forme de texture 2D. L'octet restant est utilisé comme masque d'information sur le genre des fils du nœud. En effet, dans le premier encodage, tous les élements d'un octree étaient stockés sur un texel, donc le calul de l'indice d'un fils était évident. Par exemple, pour accéder au $i^{ème}$ fils, il suffit de décaler de $i \times 4$ octets dans la liste des fils. Si les nœuds sont stockés sur deux

texels, et les feuilles sur un seulement, ce calcul n'est plus valable. On sauvegarde donc un deuxième ensemble de 8 marqueurs permettant d'indiquer le genre des fils : un bit i positionné informera que le fils i est un nœud interne. Le nouvel encodage des nœuds est illustré par la figure 3.18.

Noeud interne

Octet 1	Octet 2	Octet 3	Octet 4	Octet 1	Octet 2	Octet 3	Octet 4
Offset vers le premier fils			Masque des fils	Normale X	Normale Y	Normale Z	Genre des fils

Fig. 3.18 – Encodage sur 8 octets des nœuds de l'octree. L'encodage des feuilles reste le même que celui présenté en figure 3.17.

Cet encodage sur 8 octets des nœuds a une deuxième conséquence : la numérotation dans le tableau 1D est différente. En effet, un nœud occupe désormais deux indices du tableau 1D, la numérotation doit être adaptée. L'algorithme 8 montre cette prise en compte du nouvel espace alloué aux nœuds.

Données : Octree Oct
entier $indice \leftarrow 0$;
file $fileNoeuds$;
`enfiler`($fileNoeud$, `racine`(Oct)) ;

tant que $fileNoeud \neq \emptyset$ **faire**
- Nœud $N \leftarrow$ `debut`($fileNoeud$) ;
- `defiler`($fileNoeud$) ;
- `affecteIndice`(N,$indice$) ;
- **si** N *est une feuille* **alors**
  - $index \leftarrow indice + 1$;
- **sinon**
  - //N est un noeud
  - $indice \leftarrow indice + 2$;
- **pour** *i de 1 à 8* **faire**
  - **si** `Fils`(N,i) *existe* **alors**
    - `enfiler`($fileNoeud$,`Fils`(N,i))

Algorithme 8 : Algorithme de numérotation des nœuds pour l'encodage avec valeurs MIP-map de l'octree.

3.4.4 Sauvegarde sous forme de texture 2D

La dernière étape de l'encodage est la sauvegarde du tableau 1D créé dans une image 2D. Les indices du tableau stockés sur 4 octets sont directement convertis en pixels RGBA. Nous choisissons d'enregistrer les masques dans les canaux alpha de ces pixels, les composantes RGB sont donc dévolues au stockage des normales ou des offsets vers les premiers fils (voir figure 3.19). Le tableau 1D est sauvegardé dans des textures rectangles

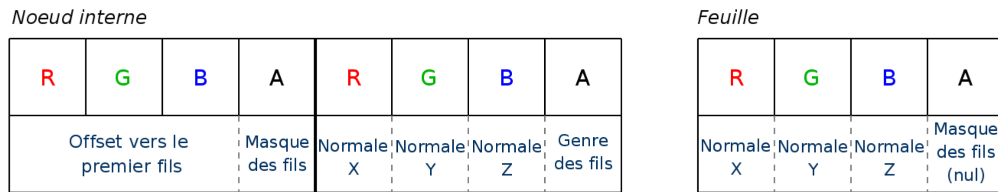


FIG. 3.19 – Sauvegarde des nœuds et feuilles de l'octree dans des pixels RGBA.

dont la largeur est de la forme 2^K, tel que K vérifie :

$$(2^{K-1} \times 2^{K-1}) < dim_T \leq (2^K \times 2^K)$$

où dim_T est la taille du tableau 1D. On prend ainsi la plus petite texture carrée qui peut contenir l'ensemble du tableau, les lignes vides de la texture sont tronquées pour ne pas utiliser inutilement de l'espace mémoire.

Pour le stockage sur disque, les textures *APO* sont sauvegardées au format PNG. Ce format a l'avantage de proposer une compression sans perte des données. En effet les *APO* seraient rendues inexploitables par une altération des valeurs des masques. La figure 3.20 montre une texture *APO* de faible résolution. Pour cette illustration, les valeurs alpha des pixels n'ont pas été prises en compte. La plupart des masques ont une valeur positives, et donc la plupart des pixels sont en partie transparents sur la texture *APO* produite.

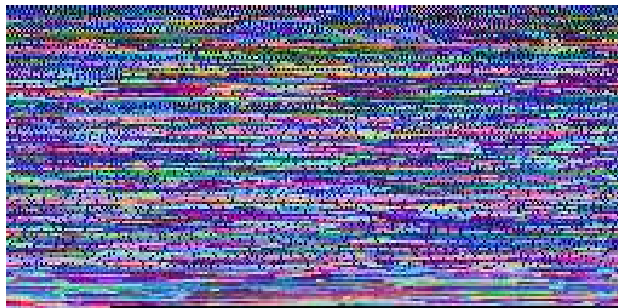


FIG. 3.20 – Une texture *APO* au format RGBA.

3.5 Performances et discussion

Nous allons présenter dans cette partie les résultats de création des *APO*, d'abord en terme de temps de création, puis en terme d'occupation mémoire. De plus, nous commentons et discutons les différents résultats exposés.

3.5.1 Temps de construction des APO

Le tableau 3.1 présente les temps de création des *APO* pour différents maillages et différentes valeurs de tolérance V^N. Comme pour les tableaux de la sections 3.2.4.1 page 51 et ceux de l'annexe B page 157, la profondeur maximale de l'octree est limitée au niveau 13. Les tailles des maillages utilisés sont également indiquées sur ces tableaux. Toutes les *APO* créées pour les statistiques ont été calculées sur un PC AMD 3500+ (mono cœur) avec 2Go de mémoire vive, sur un système Linux/Debian 32 bits.

Maillages	V^N			
	0.5	**0.3**	**0.1**	**0.05**
Stanford Dragon	23s 1024 × 258	27s 1024 × 356	37s 1024 × 741	49s 2048 × 600
Vase Lion	33s 1024 × 794	40s 2048 × 647	1'18s 2048 × 1722	1'40s 4096 × 1421
Happy Bouddah	28s 1024 × 610	35s 1024 × 903	52s 2048 × 1082	1'19s 2048 × 1870
Moulage de main	28s 1024 × 391	35s 1024 × 626	55s 2048 × 949	1'12s 2048 × 1860
Neptune	1'41s 1024 × 668	1'51s 1024 × 947	2'27s 2048 × 1049	3'07s 2048 × 1766
XYZRGB Dragon	3'23s 2048 × 992	3'59s 2048 × 1455	5'32s 4096 × 1640	8'37s 4096 × 2621
XYZRGB Statue thaïe	6'02s 2048 × 1902	6'58s 4096 × 1300	11'54s 4096 × 2522	n.a swap

Tab. 3.1 – Temps de création et dimensions des *APO* créées.

Les temps présentés prennent en compte toutes les étapes de création, c'est à dire la lecture des maillages, la création de l'octree initilal, la subdivision adaptative, l'échantillonnage du champs de normales puis l'encodage et l'écriture de la texture. Les options de création des *APO* sont les suivantes :

- Pas de gestion d'erreur de manque de données (voir section 3.2.4.2 page 54)
- Echantillonnage par lancer de rayons, directions des normales interpolées. Distance maximale de recherche d'intersection : 0.035 (largeur d'un nœud de profondeur 5,

voir section 3.3.1.1 page 59).
- Encodage de l'octree avec valeur MIP-map (voir section 3.4.3 page 70), sauvegarde au format PNG.

Bien que la lecture des maillages haute résolution puisse prendre un temps non négligeable, la majorité du temps de création est dépensée dans les étapes de subdivision et d'échantillonnage, qui prennent à elles deux environ 80% à 90% du temps total d'exécution lors des créations des *APO* les plus précises.

Si la phase de subdivision semble directe et donc ne semble pas proposer de grandes libertés quant à son optimisation, la phase d'échantillonnage est plus propice à la mise en œuvre d'améliorations. En effet, bien que nous ayons implémenté les tests d'intersections entre rayons et triangles les plus efficaces qui soient, la recherche des cellules intersectées (section 3.3.1.2 page 61) par chacun des rayons est un domaine largement traité dans la littérature concernant le rendu par ray-tracing.

Il existe deux familles d'algorithmes de parcours pour tester l'intersection entre un octree et un rayon. Les méthodes ascendantes partent de la première cellule intersectée par le rayon et détectent les cellules voisines intersectées par le rayon. Les travaux de [Sam89] présentent une telle méthode.

Nous avons implémenté une méthode descendante, partant de la racine vers les nœuds de plus bas niveau, basée sur une représentation paramétrique du rayon, qui peut s'apparenter à des méthodes déjà existantes, comme celle de [RUL00] ou de [AGL91]. Cette méthode paramétrique a été optimisée par les auteurs de [SW91] qui présentent une manière d'effectuer les tests d'intersection entre le rayon et l'octree uniquement avec de l'arithmétique entière.

D'autres travaux sur l'optimisation du lancer de rayons comme [Jan86, WSC+95, Sun91], pourraient être adaptés à nos besoins pour tenter d'optimiser la phase d'échantillonnage. Cependant, notre implémentation de recherche descendante, combinée à un test d'intersection efficace entre le rayon et les boites [WBMS05a, WBMS05b], nous permet d'obtenir des temps de création satisfaisants.

3.5.2 Occupation mémoire des APO

Dans cette section, nous traitons de la place occupée par les *APO*. Nous discutons dans un premier temps de l'espace de stockage des textures, puis de l'espace mémoire nécessaire pendant la création des textures. Dans la suite, nous notons N_n le nombre de nœuds internes, N_f le nombre de feuilles et N_o le nombre total de nœuds de l'octree.

Les tailles indiquées des textures *APO* servent à donner un ordre de grandeur des textures, des plus détaillées aux plus grossières. Nous ne faisons pas de comparaisons

avec la taille du maillage original, le but de nos travaux n'est pas de proposer une compression de maillages, mais de proposer un avatar ayant l'apparence du maillage original qui pourra être manipulé de manière interactive.

3.5.2.1 Taille des textures APO

La taille des textures *APO* créées est dépendante des paramètres de construction et subdivision présentés plus tôt, mais également du nombre de détails et des variations dans le champs de normales présents sur le maillage original. Les textures peuvent avoir des tailles allant de quelques centaines de kilo-octets jusqu'à environ 40 Mio pour les plus détaillées d'entre elles.

Si la profondeur et la tolérance V^N permettent de limiter et contrôler le niveau de détails et donc l'espace qu'occupent les textures *APO*, ils ne peuvent pas aider à prédire exactement la taille qui sera utilisée au final par les *APO*. En effet, pour deux maillages ayant environ le même nombre de polygones et pour les mêmes tolérances et profondeurs maximales autorisées, la taille de l'*APO* peut être très variable. Le maillage ayant les plus hautes fréquences de variation dans son champs de normales devra voir son octree subdivisé plus profondément pour atteindre le même niveau de détail dans chaque feuille.

Ainsi la tolérance V^N doit être vue comme un critère de qualité des *APO* plus qu'une valeur permettant de limiter la taille des *APO*. Il serait par contre possible, dans le cas où plus de contrôle était nécessaire, d'ajouter un autre paramètre à la subdivision pour limiter la profondeur en fonction d'une occupation mémoire maximale autorisée. Par exemple, en ne subdivisant pas les branches de l'arbre en profondeur jusqu'à satisfaire la tolérance sur V^N, mais en rangeant les nœuds dans une file de priorité, il serait envisageable de contrôler la dimension maximale des *APO*. Les nœuds de la file seraient subdivisés jusqu'à ce que la dimension maximale soit atteinte. L'utilisation de cette file de priorité sur la variation V^N des nœuds assurerait la couverture homogène de la subdivision sur la surface du maillage, évitant la subdivision des premières branches jusqu'à la profondeur maximale qui risque de saturer la mémoire maximale autorisée sans offrir un échantillonnage régulier.

Une autre piste pour limiter la taille des textures *APO* serait de prédire la tolérance requise pour atteindre la dimension maximale de texture autorisée. En étudiant individuellement pour chaque maillage la courbe d'évolution du nombre de nœud en fonction de V^N, courbe qui peut être obtenue par une étude statistique détaillée des résultats de plusieurs subdivisions d'un même maillage avec des V^N différents, on serait dans la capacité de prédire le seuil nécessaire pour atteindre une dimension de texture souhaitée. Cependant cette méthode nécessiterait la génération de plusieurs textures pour permettre de déterminer les courbes d'évolution du nombre de nœuds d'un modèle.

Encodage sans valeurs MIP-map : Pour cet encodage, décrit dans la partie 3.4.2 page 68, les nœuds de l'octree sont encodés sur un texel, qu'ils soient des feuilles ou des

nœuds internes. La taille T_{APO} des APO est donc dans ces cas là calculée par la formule suivante :

$$T_{APO} = (N_f + N_n) \times 4 \; octets = (N_o \times 4) \; octets$$

Rappelons que chaque texel est codé sur 4 octets dans le cas d'une texture RGBA. Nous prenons pour exemple la texture APO construite pour le maillage XYZRGB Dragon (voir tableau 3.7), avec la valeur de tolérance V^N de 0.05, nous avons un octree composé de 8 673 849 nœuds ce qui fait une taille de 33 Mio. Cette taille correspond à la taille d'une texture bitmap, la taille pour le stockage sera moindre si l'on utilise la compression PNG.

Pour une texture moins détaillée, si l'on garde le même exemple de maillage, mais cette fois avec une valeur de tolérance plus forte, prenons 0.3, le nombre de nœuds est cette fois de 2 433 306, ce qui donne une texture de 9.3 Mio. Dans la partie 4 les différences impliquées par la variation de V^N seront illustrées.

Encodage avec valeurs MIP-map : Avec cet encodage, décrit dans la partie 3.4.3 page 70, les nœuds occupent 2 octets chacun. La formule pour calculer l'espace mémoire devient donc :

$$T_{APO} = (N_f + 2 \times N_n) \times 4 \; octets$$

Pour les deux mêmes exemples que précédemment, avec les tolérances de 0.3 et 0.05, les tailles des APO sont respectivement de 11.4 Mio et de 41 Mio. Avec le format PNG, ces images sont ramenées à 9.2 et 32 Mio.

Comparaison avec l'encodage classique des octree-textures sur GPU : Dans l'encodage proposé par [LHN05], un nœud est représenté par un cube de 8 texels dans une texture 3D. Dans chacun de ces texels, on trouve soit une adresse vers un autre cube si le fils correspondant est un nœud, soit directement la valeur correspondante si le fils est une feuille. La distinction entre nœud et feuille, voire fils vide, se fait par la valeur du canal alpha. Pour sauvegarder une valeur filtrée aux nœuds, une deuxième texture contenant un pixel par nœud est utilisée. Nous avons donc un espace mémoire calculé par la formule :

$$T = ((N_n \times 8) + N_n) \times 4 = N_n \times 9 \times 4$$

$$T = N_n \times 36 \; octets$$

Un nœud est représenté par 8 texels, plus un dans la texture sauvegardant les valeurs filtrées. Nous avons constaté avec nos expérimentations que $N_f \approx 3.5 \times N_n$, donc en le remplaçant dans notre encodage nous obtenons :

$$T_{APO} = (N_f + 2 \times N_n) \times 4 \approx (3.5 \times N_n + 2 \times N_n) \times 4$$

$$T_{APO} \approx N_n \times 22 \; octets$$

Nous pouvons donc supposer que notre encodage est plus compact que celui présenté dans [LHN05], la taille des *APO* étant environ 40% plus petite pour un octree équivalent.

Sur l'exemple du dragon XYZRGB avec la tolérance V^N de 0.05, le calcul donne pour notre encodage :

$$T_{APO} = (N_f + 2 \times N_n) \times 4 = (6\ 612\ 123 + 2 \times 2\ 061\ 726) \times 4$$

$$T_{APO} = 42\ 942\ 300\ octets$$

Soit environ 41 Mio. Avec l'encodage de [LHN05], la taille de la même octree-texture serait de :

$$S = 2\ 061\ 726 \times 36 = 74\ 222\ 136\ octets$$

Soit environ 71 Mio. Notre encodage permet donc d'économiser 30 Mio. Cette différence sur l'occupation mémoire s'explique par le fait que dans notre encodage, les nœuds vides ne sont pas sauvegardés, alors qu'ils ont un représentant dans l'encodage de [LHN05].

3.5.2.2 Occupation en mémoire vive lors de la construction

Nous venons de voir l'espace mémoire qu'occupent les *APO* après leurs créations, nous allons dans cette partie commenter brièvement l'espace nécessaire en mémoire vive pendant la création de ces *APO*.

Si pour l'encodage de l'octree en texture on utilise le rangement de l'octree dans un tableau 1D, ne nécessitant qu'un pointeur d'un nœud vers ses fils, cette structuration est difficilement utilisable lors de la création de l'octree. On utilise une implémentation classique avec un tableau de huit pointeurs vers les fils respectifs. Dans chaque cellule sont également présentes les listes t^N et T^N qui référencent les triangles intersectées de m et M. Comme les triangles peuvent intersecter plusieurs nœuds et feuilles, les indices des triangles sont dupliqués dans chacune des cellules concernées.

Les nœuds et feuilles contiennent également la normale représentative associée, un indice pour sauvegarder la position qu'aura le nœud dans le tableau 1D ainsi que toutes les informations nécessaires sur sa géométrie et sa position : le centre, la largeur, la profondeur. Toutes ces données ajoutées font que les nœuds et feuilles peuvent chacun occuper jusqu'à une centaine d'octets sur un système 32 bits (pointeurs sur 4 octets).

L'octree complet, pour les objets très détaillés, occupe donc un vaste espace mémoire, dépassant 1 Go et plus. Il faut donc avoir assez de mémoire vive pour éviter le swap qui dégrade fortement les performances de création. Lors de nos expérimentations, nous avons constaté que la mémoire pouvait être saturé pour les maillages composés de plus de 10 millions de polygones. Les textures alors générées sont très proches de la dimension 4096×4096.

L'utilisation d'un *pointerless octree* durant la phase de construction des *APO*, voire la conversion out-of-core de notre algorithme de subdivision et d'échantillonnage permettrait de s'affranchir de cette nécessité d'avoir assez de mémoire vive disponible. Une autre possibilité serait d'utiliser plusieurs textures pour le même objet.

Chapitre 4

Placage des APO par le GPU

Sommaire

4.1 Introduction **80**
4.2 Parcours GPU de l'octree **80**
4.2.1 Calcul de l'index du nœud fils contenant le fragment . . . 81
4.2.2 Détermination de l'offset vers le nœud fils 83
4.2.2.1 Offset vers le premier fils 84
4.2.2.2 Décalage vers le nœud fils souhaité dans la fratrie 84
4.2.2.2.1 Encodage sans valeurs MIP-map : . . . 85
4.2.2.2.2 Encodage avec valeurs MIP-map : . . . 86
4.2.3 Lecture du nœud dans la texture *APO* 88
4.2.4 Mise à jour des informations spatiales de la cellule 88
4.2.5 Condition d'arrêt de la boucle de parcours 89
4.2.6 Lecture et décodage de la normale du fragment 89
4.2.7 Boucle de parcours complète 90
4.3 Filtrage des APO **91**
4.3.1 Interpolation bilinéaire 92
4.3.1.1 Nouvelle condition d'arrêt de la boucle 92
4.3.1.2 Résultats et performances 94
4.3.1.3 Approximation du filtrage 95
4.3.2 Rendu adaptatif 97
4.3.2.1 Calcul de la profondeur par utilisation des fonctions GLSL 98
4.3.2.2 Filtrage trilinéaire 100
4.4 Rendus et Performances **101**
4.4.1 Qualité visuelle 101
4.4.2 Performances d'affichage 104
4.4.2.1 Performances en gros plan 104
4.4.2.2 Performances en plan large 104
4.4.2.3 Performances avec filtrage bilinéaire 105
4.4.3 Discussion 106

4.1 Introduction

Les *APO* sauvegardent le champs de normales du maillage *M*, pour le plaquer sur un maillage simplifié m lors du rendu. Cette substitution permet de manipuler un avatar ayant les détails du maillage original de manière interactive, sans avoir la lourdeur d'un objet à la géométrie complexe. Nous utilisons donc le maillage m simplifié avec l'*APO* comme texture de détails. Les octree-textures n'étant pas matériellement supportées par les cartes graphiques comme le sont les textures 2D ou 3D, un programme pour GPU, ou shader, permet de retrouver dans cette structure hiérarchique la normale à associer à chaque fragment. Le placage des *APO* n'est donc possible que sur les cartes graphiques programmables, celles qui supportent les shaders. Un shader est un programme utilisateur qui remplace le pipeline classique de traitement des primitives graphiques.

L'association d'une normale à chaque fragment du maillage simplifié par le fragment shader est l'objet de ce chapitre. Nous verrons comment l'*APO* est parcourue, quelles sont les différences d'algorithme selon les encodages utilisés ainsi que l'utilisation de la structure hiérarchique pour effectuer un rendu adaptatif. Les codes associés sont donnés dans le langage GLSL. Nous présentons à la fin de ce chapitre des résultats, puis nous les commentons termes de vitesse d'exécution, mais aussi en termes de rendu visuel.

4.2 Parcours GPU de l'octree

Le parcours de l'octree permettant de déterminer la normale associée à chaque fragment pour le calcul de l'éclairage est effectué par un fragment shader. La texture *APO* est accessible depuis le fragment shader sous la forme d'une texture 2D classique. Pour retrouver la normale associée au fragment, il n'y a pas de paramétrisation, seule la position (x, y, z) dudit fragment, que l'on notera $Frag$, est nécessaire pour localiser la feuille de l'octree contenant la normale. Le fragment shader, outre la position du fragment et la texture *APO*, doit également connaitre les dimensions de la texture *dimXTexture* et *dimYTexture*, pour pouvoir convertir les indices 1D des nœuds dans le tableau en coordonnées de textures 2D. Les dimensions de l'*APO* sont envoyée au fragment shader sous forme de variable uniforme.

Le premier nœud à être traversé est la racine, qui se trouve à l'indice 0 dans le tableau 1D. La texture *APO* ne contenant que les informations sur la structure de l'octree et sur les normales associées à chaque cellule, il faut indiquer les informations spatiales des cellules associées aux nœuds traversés. La racine est centrée sur la positon $(0.5, 0.5, 0.5)$ et sa largeur vaut 1.0. A chaque itération, la largeur de cellule est divisée par 2, tandis que le centre est déplacé en fonction du nœud fils accédé.

L'*APO* est parcourue jusqu'à ce qu'une feuille soit atteinte. A chaque itération, il faut :

1. Déterminer si le nœud courant est un nœud interne ou une feuille.
2. Dans le cas d'une feuille, le parcours est stoppé.
3. Dans le cas d'un nœud interne, trouver quel est le fils qui contient le fragment.
4. Calculer la position du nœud fils souhaité :
   - Lire l'offset vers le premier fils
   - Ajouter le décalage pour atteindre le nœud souhaité.
5. Lire le texel contenant le nœud fils qui sera traité à l'itération suivante.
6. Mettre à jour les informations spatiales (centre et largeur).

L'algorithme de parcours de l'*APO* effectuant toutes ces opérations est décrit par l'algorithme 9. Dans cet algorithme les fonctions listées ci-dessous sont utilisées. Chacune d'entre elles est détaillée dans les sections suivantes.

- **accesTexture2D :** permet d'accèder à la texture 2D à un indice donné. Cette fonction convertit l'indice donné en paramètre en coordonnée de texture 2D.
- **noeudFils :** retourne le numéro d'octant du nœud fils contenant le fragment. Ce numéro est calculé en fonction de la position du fragment dans la cellule.
- **offsetPremierFils :** donne l'offset vers le premier fils. Cet offset est lu dans les canaux RGB du texel.
- **positionNoeudFils :** donne la position d'un nœud dans la fratrie des nœuds fils.
- **miseAJourRacineCourante :** modifie la largeur et la position du centre de la cellule courante.

La plupart des opérations de cet algorithme nécessitent des manipulations de bits, pour lire la valeur d'offset ou décoder les masques qui ont été stockés dans l'*APO*. Ces manipulations binaires n'étant pas disponibles sur tous les processeurs GPU, nous les avons simulés grâce à des opérations mathématiques simples, comme par exemple la division par deux pour effectuer un décalage de bits. Les codes présentés sont donc écrits avec ces simulations d'opérations binaires.

4.2.1 Calcul de l'index du nœud fils contenant le fragment

Pour déterminer dans quel octant se trouve le fragment, la solution directe est de comparer la position de ce dernier par rapport aux plans médians qui découpent la cellule. Cette méthode est cependant coûteuse car elle nécessite des comparaisons sur les trois axes. Nous proposons une méthode permettant de calculer l'indice de la cellule fille contenant le fragment de manière directe.

Notre approche consiste à extraire un code binaire sur 3 bits du vecteur entre le centre de la cellule et la position du fragment. Ce code binaire converti en valeur décimale donne l'indice de l'octant contenant le fragment. L'extraction du code binaire se fait en deux

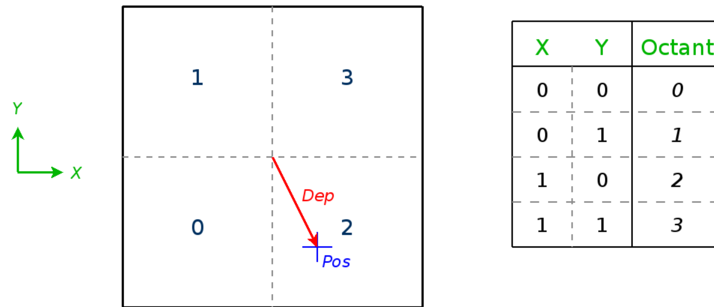
Données : Texture APO $texAPO$
Données : Position du fragment F_{xyz}
Données : Entier $dimXTexture$

// Initialisation
Entier $indice \leftarrow 0$;
Position $centre \leftarrow (0.5, 0.5, 0.5)$;
Réel $largeur \leftarrow 1.0$;
Nœud $N \leftarrow$ `accesTexture2D(`$texAPO$, $indice$, $dimXTexture$`)` ;

// Boucle de parcours
tant que *N est un nœud* **faire**
  Noeud $N_f \leftarrow$ `noeudFils(`N,F_{xyz}`)` ;
  Entier $offset \leftarrow$ `offsetPremierFils(`N`)` ;
  $offset \leftarrow offset$+`positionNoeudFils(`N,N_f`)` ;
  $indice \leftarrow indice + offset$;
  $N \leftarrow$ `accesTexture2D(`$texAPO$, $indice$, $dimXTexture$`)` ;
  `miseAJourRacineCourante(`$centre$, $largeur$`)` ;

Algorithme 9 : Algorithme de parcours de l'*APO*

temps : on calcule le vecteur $\overrightarrow{dep} = F_{xyz} - centre$ entre le centre de la cellule et la position du fragment, dans un deuxième temps les composantes positives de $\overrightarrow{dep}$ sont remplacées par 1, les composantes négatives par 0. On obtient alors un code binaire qui donne l'indice de la cellule fille qui contient le fragment. Le principe de ce calcul est illustré en 2D par la figure 4.1.

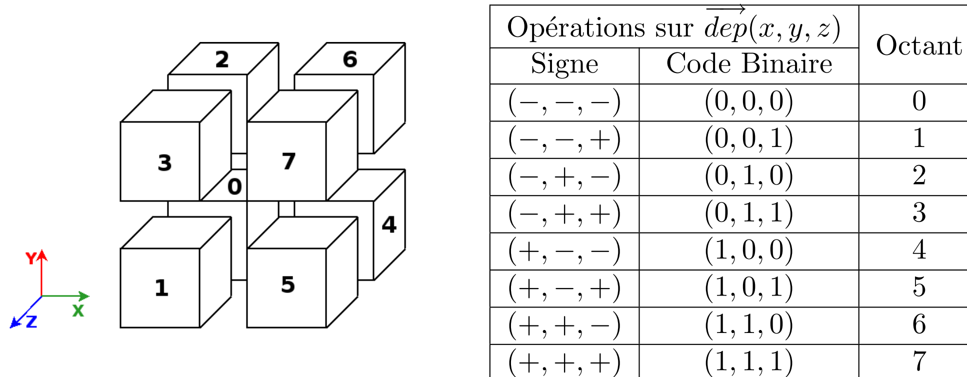


X	Y	Octant
0	0	*0*
0	1	*1*
1	0	*2*
1	1	*3*

FIG. 4.1 – Le vecteur entre la position du fragment et le centre de la cellule permet d'extraire un code binaire dans lequel une valeur 0 signifie une valeur négative de déplacement sur l'axe en question. Ce code binaire est directement converti en indice de cellule.

La numérotation des octants d'un nœud a été choisie de façon à correspondre à ce calcul d'indice. Le tableau de la figure 4.2 récapitule ce calcul d'index et la numérotation

choisie.



Opérations sur $\overrightarrow{dep}(x, y, z)$		Octant
Signe	Code Binaire	
$(-, -, -)$	$(0, 0, 0)$	0
$(-, -, +)$	$(0, 0, 1)$	1
$(-, +, -)$	$(0, 1, 0)$	2
$(-, +, +)$	$(0, 1, 1)$	3
$(+, -, -)$	$(1, 0, 0)$	4
$(+, -, +)$	$(1, 0, 1)$	5
$(+, +, -)$	$(1, 1, 0)$	6
$(+, +, +)$	$(1, 1, 1)$	7

FIG. 4.2 – Numérotation des octants et opérations binaires calculant les indices.

La fonction noeudFils en charge de cette conversion s'écrit en GLSL de la manière suivante :

```
int noeudFils(vec3 Center, vec3 Pos) {
  vec3 dep = Pos - Center;
  dep = step(0.0, sign(dep));
  return int(dep.x*4.0 + dep.y*2.0 + dep.z);
}
```

Les opérations décrites ci-dessus sont codées avec les fonctions GLSL existantes, la fonction sign appliquée sur le vecteur $\overrightarrow{dep}$ permet de récupèrer pour chaque composante son signe, c'est à dire que chaque composante est remplacée par la valeur -1.0 si elle est négative ou 1.0 si elle est positive. Il faut ensuite remplacer chaque valeur négative par 0.0. Cette opération se fait par la fonction step(x,v) qui retourne 0 si $v <= x$, 1 sinon. En choisissant 0.0 comme seuil les valeurs négatives sont ramenées à 0. On obtient donc un vecteur ayant pour chacune des composantes la valeur 1 ou 0, vecteur qui peut être interprété comme un code binaire. L'indice est calculé par une simple multiplication, la composante x étant associée au bit de poids fort, z au bit de poids faible. Rappelons que ces opérations et leurs résultats sont illustrés par la figure 4.2.

4.2.2 Détermination de l'offset vers le nœud fils

L'étape décrite précédemment a permis de trouver quel nœud fils du nœud courant doit être parcouru pour atteindre la feuille la plus profonde contenant la position F_{xyz} du fragment $Frag$, ou plutôt son numéro d'octant, notons le no, associé. En supposant que le nœud courant N soit positionné à l'indice N_i dans le tableau 1D de l'APO, l'indice du nœud fils N_f est calculé par la formule :

$$N_f = N_i + offsetPremierFils(N) + decalageFils(N, no)$$

La fonction offsetPremierFils donne l'offset vers le premier fils, encodé dans les canaux RGB du texel (voir la section 3.4 sur l'encodage). La seconde quant à elle donne le

décalage à ajouter à l'offset pour atteindre l'octant *no*. En effet, comme les branches vides ne sont pas sauvegardées dans l'*APO*, l'octant numéroté i n'est presque jamais à la position i dans la liste des enfants du nœud. Ce cas de figure est illustré par la figure 4.3.

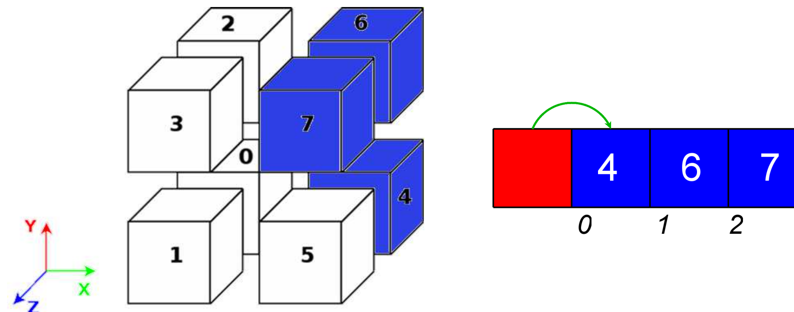


Fig. 4.3 – Les cellules bleutées sont les seules non vides. Dans la liste 1D des fils du nœud, représentée à droite, les nœuds 4, 6 et 7 auront respectivement pour indice 0, 1 et 2.

Nous détaillons ci-dessous les deux fonctions permettant le calcul de l'offset vers le prochain nœud à être traversé lors du parcours de l'*APO*.

4.2.2.1 Offset vers le premier fils

L'offset vers le premier fils est directement lu dans les canaux RGB du texel représentant le nœud. Les valeurs lues dans le texel pour chaque canal sont des valeurs réelles comprises dans l'intervalle $[0, 1]$. Pour retrouver la valeur décimale qui était sauvegardée dans l'octet correspondant, la valeur réelle est mutipliée par 255. Le canal R est associé aux bits de poids forts, tandis que le canal B est lui associé aux bits de poids faibles, ce qui donne la fontion de décodage de l'offset vers le premier fils suivante :

```
int offsetPremierFils(vec4 N) {
  return int(N.b * 255.0 +
             N.g * 255.0 * 256.0 +
             N.r * 255.0 * 65536.0);
}
```

Le paramètre de cette fonction est un vecteur de 4 flottants, c'est à dire un texel qui a été lu au préablable dans la texture RGBA par une fonction d'accès aux textures 2D, dans notre cas la fonction GLSL texture2D.

4.2.2.2 Décalage vers le nœud fils souhaité dans la fratrie

Pour ajuster l'offset vers le nœud fils souhaité, il est nécessaire de déterminer sa position dans la fratrie, c'est à dire dans la liste nœuds fils. On utilise pour ce faire le

masque des fils, sauvegardés dans le canal alpha du texel représentant le nœud. Le but est de compter le nombre de fils existants avant le nœud que nous souhaitons accéder, c'est à dire le nombre de bits positionnés à 1 avant le bit associé au prochain nœud traversé. Ce comptage est illustré par l'algorithme trivial 10.

Données : Masque ME des fils existants
Données : Entier $offsetPF$: offset vers le premier fils du nœud
Données : Entier *no* : numéro d'octant du prochain nœud à être accédé.

```
pour i de 0 à (no − 1) faire
    si (bit(i,ME) == 1) alors
        augmenteOffset(offsetPF)
```

Algorithme 10 : Algorithme général d'ajustage de l'offset vers un nœud.

Nous différençons cependant plusieurs manières d'augmenter l'offset en fonction des encodages choisis, que nous détaillons chacune dans les parties suivantes.

4.2.2.2.1 Encodage sans valeurs MIP-map : Avec cet encodage, traité sections 3.4.2 (pages 68) les nœuds internes et feuilles sont encodés sur un seul texel, il suffit donc d'incrémenter l'offset dès qu'on rencontre un bit à 1 dans la boucle de l'algorithme 10. Nous choisissons une implémentation GLSL qui ne fait pas de tests, mais ajoute directement la valeur du bit. Si le bit i est positionné à 1, alors il faut compter un nœud avant celui que nous souhaitons accéder et donc incrémenter l'offset. Si le bit i est à 0, l'ajouter à l'offset n'aura pas de conséquence sur le parcours.

Dans l'implémentation GLSL, la valeur du canal alpha est multipliée par 255 pour obtenir la valeur décimale représentée par le masque. Les valeurs individuelles de chaque bit sont obtenues en récupérant le reste de la division du masque par 2, ce qui est effectué par la fonction mod(masque, 2.0). Pour simuler le décalage de bit, la valeur du reste est soustraite à la valeur du masque, de façon à toujours obtenir un nombre pair, puis on divise le masque par 2 pour accéder à la valeur du bit de rang directement supérieur.

Avec cette simulation des opérations binaires, on ne peut accéder aux valeurs des bits que de manière séquentielle. Le bit de poids faible ayant été choisi pour correspondre à l'octant numéroté 0, la série de division par deux est compatible avec l'algorithme 10 dans le sens où l'on veut compter tous les fils existants avant le nœud de l'octant *no* souhaité, c'est à dire tous les fils existants dans l'intervalle d'indice $[0, no[$.

Le code GLSL suivant correspond à l'algorithme que nous venons de décrire. On passe en paramètre le nœud N, toujours sous la forme du vecteur de 4 réels lus dans la texture, l'indice *no* de l'octant qui sera traité à l'étape suivante ainsi que l'offset, en variable inout de façon à ce qu'il puisse être modifié dans la boucle de parcours. Le mot clé inout correspond à un passage par référence de la variable.

```
void calculDecalageDansFratrie1(vec4 N, int no, inout int offset) {
  float masque = N.a * 255.0;
  float bitFrereCourant;

  for(int it=0 ; it<no ; ++it) {
    // récupère le reste de la division par 2 du masque
    bitFrereCourant = mod(masque, 2.0);
    // incrémente l'offset (si 0, effet nul)
    offset += int(bitFrereCourant);
    // enlève le reste et divise par 2 => décalage de bit simulé
    masque -= bitFrereCourant;
    masque /= 2.0;
  }
}
```

4.2.2.2.2 Encodage avec valeurs MIP-map : Dans cet encodage, décrit dans la section 3.4.3 page 70, les nœuds internes occupent deux texels alors que les feuilles n'en occupent qu'un. Il faut donc prendre en compte cette différence, et augmenter l'offset de deux lorsqu'un fils précédant le nœud souhaité dans la fratrie est un nœud interne, simplement incrémenter lorsque l'on rencontre une feuille. Cette différenciation est nécessaire car les feuilles ne se trouvent pas forcément toutes à la même profondeur, étant donnée la subdivision adaptative appliquée à l'octree.

Pour connaitre le genre des nœuds fils, les informations du deuxième masque sont utilisées (voir figure 3.18 page 71). Le parcours de l'octree avec cet encodage nécessite donc deux accès texture pour lire un nœud interne. Les deux texels étant voisins, ce deuxième accès texture n'est pas pénalisant pour les performances. L'algorithme modifié 11 de mise à jour de l'offset est présenté ci dessous.

Données : Masque masqueExistence des fils existants
Données : Masque masqueGenre du genre des fils
Données : Entier $offsetPF$: offset vers le premier fils du nœud
Données : Entier no : numéro d'octant du prochain nœud à être accédé.

pour *i de 0 à* $(no - 1)$ **faire**
  si *(bit(*i,masqueExistence*) == 1)* **alors**
    si *bit(*i,masqueGenre*) == 1* **alors**
      offsetPF ← offsetPF + 2 ;
    sinon
      offsetPF ← offsetPF + 1 ;

Algorithme 11 : Algorithme d'ajustage de l'offset vers un nœud pour l'encodage avec valeur MIP-map de l'octree.

L'implémentation GLSL de cet algorithme est similaire à celle vue précédemment quant à la simulation des opérations binaires.

```
void calculDecalageDansFratrie2(vec4 N1, vec4 N2, int no, inout int offset) {
  float masqueExistence = N1.a * 255.0;
  float masqueGenre = N2.a * 255.0;
  float bitExistence, bitGenre;

  for(int it=0 ; it<no ; ++it) {
    bitExistence = mod(masqueExistence, 2.0);
    bitGenre = mod(masqueGenre, 2.0);
    offset += int((bitGenre + 1.0) * bitExistence);

    // simulation du décalage de bits
    masqueGenre -= bitGenre;
    masqueGenre /= 2.0;
    //
    masqueExistence -= bitExistence;
    masqueExistence /= 2.0;
  }
}
```

Le calcul du décalage à appliquer pour chaque fils présent ou non se fait par l'instruction suivante, dont nous analysons le comportement ci-dessous :

$$offset+ = int((bitGenre + 1.0) * bitExistence);$$

- Dans le cas où le fils de l'octant n'existe pas, le bit correspondant dans le masque d'existence, récupéré dans la variable bitExistence est à 0. La somme est donc multipliée par 0 et l'offset n'est pas modifié.
- Dans le cas où le sous fils existe, le bit bitExistence vaut 1 et l'offset est donc augmenté de la somme bitGenre+1. Deux cas se distinguent :
  - Si le fils est un nœud interne, le bit correspondant à l'octant dans le masque de genre est positionné à 1, la somme bitGenre+1 vaut 2, l'offset est bien décalé de deux texels.
  - Si le fils est une feuille, le bit dans le masque de genre vaut 0, la somme vaut 1, et l'offset est bien décalé d'un seul texel.

Les opérations sur les masques de bits étant similaires (division par deux, soustraction du reste), le placement de ces deux masques dans un vecteur deux dimensions (vec2) permet de bénéficier du parallélisme de calcul des processeurs graphiques, et donc de rendre l'opération de calcul du décalage de l'offset plus rapide. Le code devient le suivant :

```
void calculDecalageDansFratrie2(vec4 N1, vec4 N2, int no, inout int offset) {
  vec2 masques = vec2(N1.a * 255.0, N2.a * 255.0);
  vec2 bits;

  for(int it=0 ; it<no ; ++it) {
    bits = mod(masques, 2.0);
    offset += int((bits[0] + 1.0) * bits[1]);

    // simulation du décalage de bits
    masques -= bits;
    masques /= 2.0;
  }
}
```

4.2.3 Lecture du nœud dans la texture *APO*

Après l'exécution des deux étapes précédentes, l'indice du prochain nœud est connu. La conversion de l'indice en coordonnées de texture se fait de manière triviale par la fonction suivante. L'indice 0 de la racine est le coin supérieur gauche de la texture, correspondant à la ligne et colonne 0.

```
vec2 coordTexture2D(int indice) {
  // y : nombre de lignes
  int y = indice / dimXTexture;
  // x : colonnes
  int x = indice - y * dimXTexture;
  return vec2(float(x)/float(dimXTexture), float(y)/float(dimYTexture));
}
```

Cette fonction fait appel aux constantes *dimXTexture* et *dimYTexture*, qui sont les dimensions de la texture *APO*, envoyées comme variable uniforme au fragment shader. Les deux premiers calculs permettent d'identifier les coordonnées entières de la position du texel dans la texture 2D. La division entière de l'indice par la largeur de la texture nous donne le numéro de la ligne contenant le texel cherché, en soustrayant ensuite à l'indice le nombre de texels contenus dans les lignes précédentes nous obtenons le numéro de la colonne. Les coordonnées sont ensuite normalisées dans l'intervalle $[0, 1]^2$ en divisant les valeurs par les dimensions de la texture.

4.2.4 Mise à jour des informations spatiales de la cellule

La dernière étape de la boucle de parcours de l'octree consiste à actualiser les informations géométriques du nœud courant. Durant le parcours, le centre du nœud et sa largeur, qui permettent de définir la cellule dans l'espace, doivent être mis à jour à chaque itération. Si la largeur ne nécessite qu'une division par deux pour être actualisée, la nouvelle position du centre doit être déduite du centre de la cellule courante et de l'octant qui contient le fragment. On utilise pour ce faire une méthode similaire à celle présentée pour le calcul de l'indice du fils à traverser. Le principe, illustré en 2D sur la figure 4.4, est de calculer le vecteur à appliquer au centre de la cellule pour pointer vers le centre du nœud fils, calcul qui est basé sur le vecteur entre la position du centre et la position du fragment.

Pour calculer ce vecteur, on applique d'abord l'opérateur GLSL `sign` au vecteur $\overrightarrow{dep}$, pour l'orienter vers le centre de cellule. En effet, comme la fonction `sign` remplace chaque composante par ± 1, la direction du vecteur obtenu sera orientée vers les centres des cellules filles. Ensuite, chacune des composantes du vecteur se voit assigner comme valeur le quart de la largeur de la cellule courante pour ajuster la norme du vecteur. Le code GLSL compilant ces opérations est le suivant :

```
// vec3 centre : centre de la cellule
// vec3 pos : position du fragment
vec3 dep = sign(pos - centre);
dep = dep * (largeurCellule / 4.0);
// deplacement du centre et mise à jour de la largeur de la cellule
centre += dep; largeurCellule /= 2.0;
```

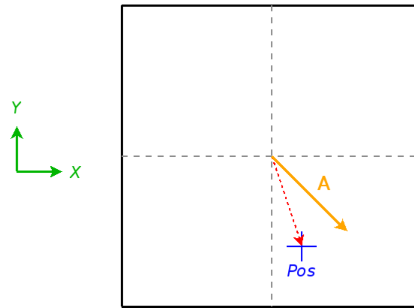


FIG. 4.4 – Le vecteur A, calculé à partir du vecteur $\vec{dep}$ entre le centre de la cellule courante et la position du fragment, permet de déplacer le centre vers celui du prochain sous fils contenant le fragment.

4.2.5 Condition d'arrêt de la boucle de parcours

Même si nous verrons dans la suite différentes conditions d'arrêt de la boucle de parcours de l'octree, celle qui est le plus naturelle est l'arrêt sur une feuille. La boucle de parcours doit être effectuée tant que les texels accédés dans la texture *APO* sont des texels représentant des nœuds internes.

Nous avons vu à la section 3.4 page 67 que la différence entre nœuds internes et feuilles se fait par la valeur du masque d'existence des fils. Une feuille n'étant pas subdivisée, elle ne possède pas de sous fils, et donc son masque est nul. Au contraire, le masque d'un nœud interne aura une valeur positive. Les deux fonctions permettant de tester le genre d'un texel sont donc les suivantes :

```
bool estUnNoeud(vec4 N) {
  return (N.a != 0.0);
}

bool estUneFeuille(vec4 N) {
  return (N.a == 0.0);
}
```

4.2.6 Lecture et décodage de la normale du fragment

Quand la boucle s'arrête, il faut extraire la normale du fragment des canaux RGB du texel. Chaque canal contient une valeur réelle de l'intervalle $[0, 1]$. Or les normales encodées avaient leurs composantes dans l'intervalle $[-1, 1]$. Il faut effectuer l'opération inverse de celle présentée section 3.4.2.1 page 69 pour retrouver la valeur de chaque composante, c'est à dire dans un premier temps multiplier par deux chaque valeur pour retrouver un intervalle de longueur 2, puis oter 1 pour centrer ce dernier autour de la valeur 0. Cette normale est ensuite multipliée par la matrice gl_NormalMatrix pour obtenir les coordonnées dans le repère caméra :

```
N.xyz = (N.rgb * 2.0) - 1.0;
vec3 fragmentNormal = gl_NormalMatrix * N.xyz;
```

4.2.7 Boucle de parcours complète

Nous présentons ici la boucle complète de parcours de l'octree. Le vertex shader, qui envoie les coordonnées du fragment ainsi que les différents vecteurs nécessaires au calcul de l'éclairage, est présenté en annexes C page 161. Dans cette annexe se trouve également le code du calcul de l'éclairage.

Le code présenté ci-dessous n'est pas complet, le parcours de l'octree doit se trouver dans la fonction main du fragment shader, ou dans une fonction appelée dans la fonction main. Nous présentons le code le plus complexe, celui correspondant à l'encodage de l'octree avec valeurs MIP-map, où les nœuds sont encodés sur 2 texels.

```
varying vec4 pos;
uniform sampler2D texAPO;
//....
vec3 centre = vec3(0.5, 0.5, 0.5);
float largeurCellule = 1.0;

vec4 N1, N2;
vec2 texCoords
int indice = 0;

do {
  // lecture premier pixel
  texCoords = coordTexture2D(indice);
  N1 = texture2D(texAPO, texCoords.st);

  // lecture deuxième pixel
  texCoords = coordTexture2D(indice+1);
  N2 = texture2D(texAPO, texCoords.st);

  // trouve dans quel octant se trouve le fragment
  int no = celluleFille(centre, pos);

  // calcul de l'offset
  int offset = offsetPremierFils(N1);
  calculDecalageDansFratrie2(N1, N2, no, offset);

  // mise à jour indice courant
  indice += offset;

  // mise à jour cellule
  vec3 dep = sign(pos - centre);
  dep = dep * (largeurCellule / 4.0);
  centre += dep;
  largeurCellule /= 2.0;

} while( N1.a != 0.0 );

N1 = N1 * 2.0 - 1.0;
vec3 fragmentNormal = gl_NormalMatrix * N1.xyz;

//..... Calcul de l'éclairage dans la suite
```

À la fin de la boucle, l'octree s'arrête sur une feuille, lue dans le texel N1. Le texel N2 est tout de même lu, le test sur le genre du nœud est fait après la lecture. On effectue donc un accès texture de trop pour chaque feuille. La mise en place de tests pour éviter cet accès texture s'est avérée plus lente que la lecture du deuxième texel. En effet le texel lu est voisin du premier, et donc l'accès texture n'est pas coûteux, la texture accédée se trouvant dans le cache.

Le code de la boucle complet est présenté dans l'annexe C.3 page 162. La figure 4.5 présente une capture d'un objet simplifié rendu avec les détails de normales lus dans l'*APO*. D'autres résultats seront présentés dans la partie 4.4 page 101, mais avant nous détaillons le filtrage et le rendu adaptatif.

Fig. 4.5 – L'image à gauche présente l'objet original, la statue *Happy Buddah* de 1 million de polygones, celle du centre une version simplifiée de 10000 polygones. La capture de droite présente le maillage simplifié, avec une texture *APO* construite avec une tolérance de 0.01, dont les dimensions sont 4096×2768. Les objets originaux et simplifiés avec APO sont difficilement différenciables.

4.3 Filtrage des APO

Les *APO* comme toutes les autres textures doivent être filtrés pour améliorer la qualité de rendu. On distingue deux cas de filtrage différents, le filtrage sous magnification (zoom) et sous minification (objet éloigné).

- Dans le cas de la magnification, c'est à dire lorsqu'un texel se projette sur plusieurs pixels de l'écran, nous proposons une interpolation linéaire dans le voisinage du texel pour faire disparaitre l'effet d'aliasing. Cette interpolation est décrite dans la section 4.3.1.
- Dans le cas de la minification, quand plusieurs texels sont projetés sur un seul pixel de l'écran, nous adoptons une méthode similaire au mip-mapping en utilisant une normale moyennée qui permet d'éviter les hautes fréquences. Ce filtrage est décrit en section 4.3.2.

4.3.1 Interpolation bilinéaire

Le placage des normales lues dans les feuilles tel qu'on l'a vu à la section précédente implique un crénelage visible entre les fragments voisins appartenant à des feuilles différentes. Ce crénelage est classique (voir section 2.3.2 de l'état de l'art), l'effet de cet *aliasing* est illustré sur la figure 4.6.

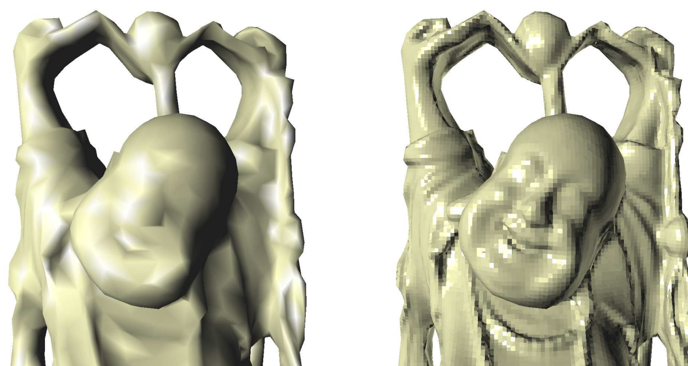


FIG. 4.6 – Le maillage de 10000 polygones est rendu avec une texture *APO*. La profondeur a volontairement été limitée à 8 pour mettre en évidence l'aliasing.

Nous proposons un filtrage bilinéaire pour lisser la texture. Comme l'octree-texture est une texture volumique, il faut effectuer le filtrage dans un voisinage de 8 cellules. Le principe, illustré par la figure 4.7, est le même que pour du filtrage bilinéaire en 2D. Pour être sûr d'interpoler avec les cellules les plus profondes, et donc les plus précises, nous prenons comme coordonnées pour la recherche des cellules voisines des positions se trouvant juste derrière les frontières de la cellule courante, en ajoutant une petite distance δ à la distance entre le fragment et la frontière de la cellule. On évite ainsi, dans le cas où les cellules voisines sont plus profondes, et donc de dimensions inégales, de chercher un voisin au delà des voisins directs (voir 4.7).

4.3.1.1 Nouvelle condition d'arrêt de la boucle

On récupère par 7 parcours supplémentaires d'*APO* les valeurs des normales voisines, on effectue l'interpolation à l'intérieur de ce cube. Contrairement aux travaux de [LHN05], où l'octree est construit de façon à ce que toutes les cellules nécessaires à l'interpolation soient présentes, ce qui alourdit la taille de l'octree, les cellules voisines recherchées peuvent être inexistantes. En effet, toutes les cellules de ce 8-voisinage n'intersectent pas forcément m. Dans le cas où une de ces cellules est manquante, nous utilisons la valeur moyennée du dernier nœud parcouru lors de la recherche de la normale de la cellule voisine. L'utilisation de la valeur moyennée plutôt que la duplication de la valeur initiale obtenue pour la cellule courante, qui aurait été une méthode plus proche

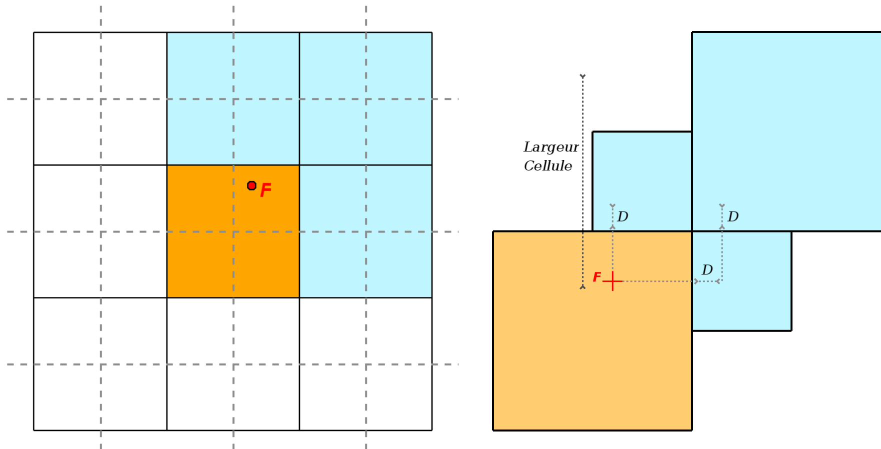


Fig. 4.7 – Si le fragment *Frag* est positionné dans le quart supérieur droit de la cellule, les cellules utilisées pour l'interpolation bilinéaire sont celles dont le fond est grisée. Dans le schéma de droite est illustré le principe de recherche des cellules voisines. On ajoute une distance faible δ pour choisir les bonnes cellules pour l'interpolation.

de celle présentée dans [LHN05], offre un plus grand lissage quand la profondeur est limitée, de par l'utilisation d'une normale déjà filtrée en sus des normales utilisées pour l'interpolation.

Il faut donc rajouter une condition d'arrêt dans la boucle, en testant l'existence du fils que l'on veut accéder à chaque itération. Cette condition n'est pas nécessaire si le filtrage n'est pas utilisé, car on sait que les nœuds existent toujours dans ce cas là. Le test de l'existence se fait en vérifiant la valeur du bit correspondant au fils dans le masque des fils. Pour atteindre ce bit, toujours dans le cas où les opérations binaires ne sont pas disponibles, il faut effectuer des divisions successives par 2 jusqu'à atteindre la position souhaitée. Or cette série de divisions par deux était déjà effectuée par la fonction comptant le nombre de fils existants avant celui que l'on voulait accéder (voir codes pages 85 et 86). Nous utilisons donc cette même fonction, en lui faisant retourner une valeur booléenne indiquant si le fils existe ou non. Pour ce faire, le reste de la division par deux du masque est retourné. Les fonctions sont donc modifiées avec le code suivant :

```
bool calculDecalageDansFratrie( vec4 N1, /*....*/ ) {
  //....
  // boucle de comptage des fils
  for(int it=0 ; it<no ; ++it) {
    // ...
    masqueExistencee /= 2.0;
  }
  // test de l'existence du fils à atteindre
  return (mod(masqueExistence, 2.0) == 1.0);
}
```

La condition de la boucle de parcours devient :

```
bool filsExiste;
//....
do {
  //.....
  filsExiste = calculDecalageDansFratrie2(N1, N2, no, offset);
  //....
}  while( N1.a != 0.0 && filsExiste);
```

4.3.1.2 Résultats et performances

Les effets du filtrage bilinéaire sont bien visibles et permettent d'atténuer voire même de faire disparaitre les effets de l'aliasing. Ces effets sont illustrés par la figure 4.9. Cependant, malgré les bénéfices indéniables du filtrage bilinéaire, nous pouvons remarquer deux défauts persistants :

Performance : Le problème majeur du filtrage est la dégradation des performances qu'il implique. En effet, il faut parcourir l'octree à huit reprises pour récupérer les normales du voisinage du fragment $Frag$. Le nombre d'images par seconde est donc logiquement divisé par environ huit lorsque l'on utilise le filtrage.

Nous avons imaginé une solution pour accélérer le parcours, en partant du constat que les cellules du voisinage étant proches, le parcours pour les atteindre doit être sensiblement le même. En sauvegardant dans un tableau les 5 derniers nœuds traversés, on peut vérifier si les cellules voisines appartiennent à ces nœuds et ainsi raccourcir le parcours, ou le faire partir de la racine si les fragments voisins n'appartiennent pas à la même branche de l'octree. Malheureusement, la gestion de la sauvegarde de la trace du premier parcours est trop coûteuse et ne permet pas d'accélérer le nombre d'images par seconde. Il faut en effet sauvegarder non seulement l'indice des cellules, mais également leurs informations spatiales (centre, profondeur, largeur), les mettre à jour à chaque itération de la boucle de parcours, puis enfin vérifier lors du filtrage si les cellules voisines appartiennent à ces nœuds précédemment traversés. Cette lourdeur ne compense pas le gain fait sur le nombre d'accès textures.

Problèmes de continuité : Le filtrage implémenté présente des défauts de continuité, ceux-ci sont dus au fait que les cellules ne sont pas toutes de même dimension. Le filtrage ne se fait donc pas automatiquement dans un cube régulier formé par les 8 voisins, mais entre des cellules voisines de différentes tailles formant un volume irrégulier (voir l'exemple en 2D sur la figure 4.8).

Notre filtrage est donc une approximation de l'interpolation réelle qu'il faudrait calculer, ce qui explique les erreurs de continuité apparentes. Les défauts de continuité sont mis en évidence par les captures de la figure 4.8. Pour réduire ces discontinuités, il

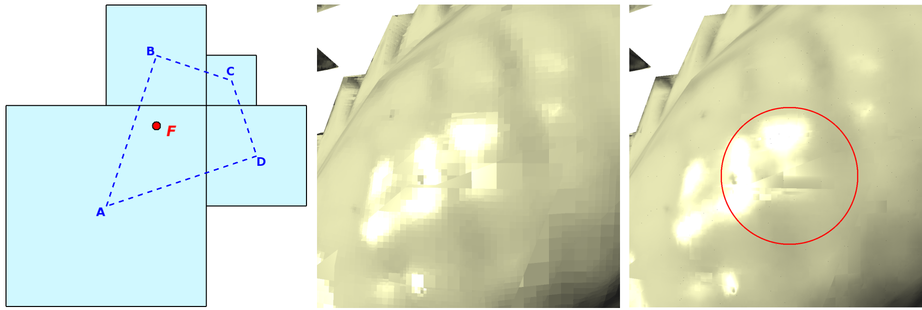


FIG. 4.8 – La normale interpolée du fragment $Frag$, du fait des dimensions inégales des cellules au voisinage, est calculée à l'intérieur du polygone irrégulier $ABCD$. Les effets sont visibles sur les deux captures de droite.

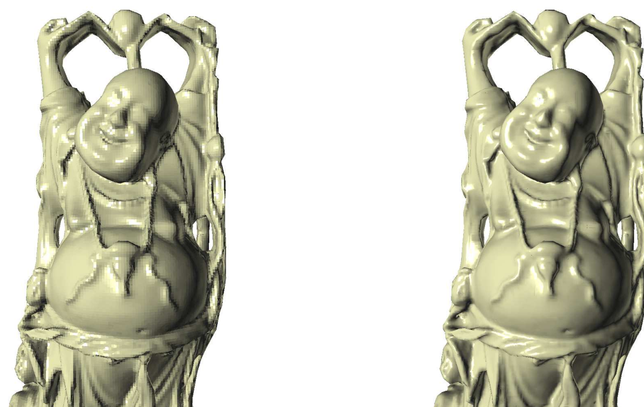
faudrait calculer une pondération différente pour chaque cellule en fonction de la profondeur de celles-ci. Les travaux présentés dans [LLY06] peuvent servir de base à ce calcul d'interpolation entre cellules de dimensions variables.

Comme on peut le constater sur la figure 4.9, les effets du filtrage sont visibles quand la profondeur est faible, ou lorsque l'on est en gros plan. Lorsque l'on est pas dans un de ces deux cas, les différences visuelles sont minimes (voir sur la figure 4.9 les captures du centre). Les performances en termes de fréquence d'affichage seront discutées dans la partie 4.4, mais d'une manière générale, l'utilisation du filtrage ne permet pas toujours une manipulation interactive des modèles texturés par l'*APO*, surtout lorsque ceux-ci sont visualisés en gros plan.

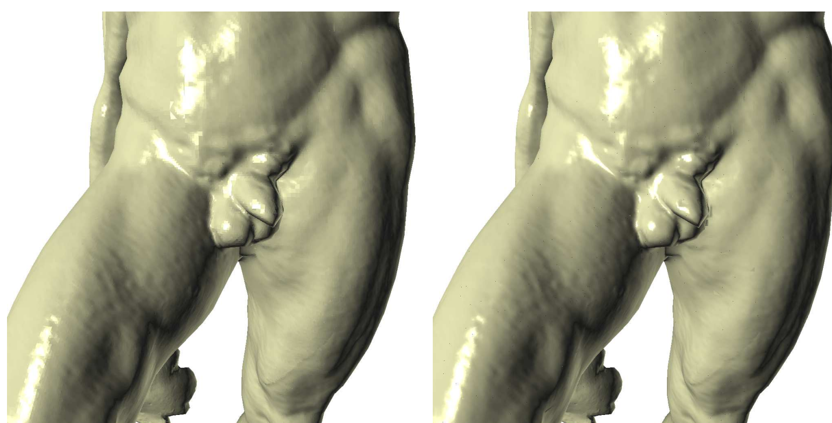
Dans [BD02], les auteurs présentent un algorithme de filtrage trilinéaire dans un voisinage 3^3 autour de la cellule. Leur méthode, inspirée des travaux de [KSS97] sur le filtrage à l'intérieur d'un quadtree, est basée sur une subdivision récursive du voisinage jusqu'à obtenir uniquement des cellules de même dimension. Leur filtrage, plus précis que celui que nous venons de présenter, ne peut malheureusement être implémenté dans notre cas. En effet leur octree CPU permet cette subdivision du voisinage, tandis que le notre a une structure fixée lors du rendu. De plus, un voisinage 3^3 impliquerait une dégradation encore plus forte des performances qui sont déjà bien affectées par l'utilisation du filtrage bilinéaire.

4.3.1.3 Approximation du filtrage

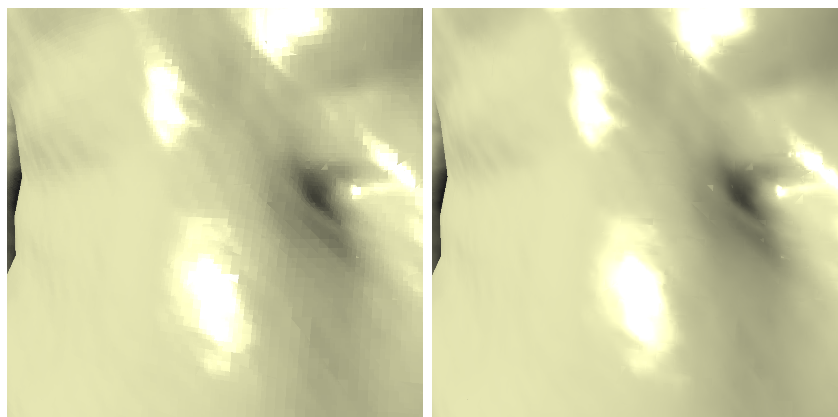
Dans le but d'améliorer les performances du filtrage, nous avons mis en place une approximation qui permet de diminuer le nombre de parcours d'octree nécessaires. Plutôt que la réelle interpolation de la normale au sein du volume formé par l'octree, nous allons effectuer un lissage entre fragments voisins sur la surface, fragments que nous allons déterminer grâce aux fonctions dérivatives GLSL `dFdx` et `dFdy`. Ces deux fonctions cal-



(a) Effet du filtrage sur le modèle *Happy Buddah*. La profondeur est limitée à 8.



(b) Effet du filtrage sur la statue *Neptune*. La profondeur n'est pas limitée.



(c) Effet du filtrage sur la statue *Neptune* (détail).

FIG. 4.9 – Captures mettant en exergue les effets du filtrage bilinéaire des textures *APO* implémenté. Les captures de gauche sont prises sans filtrage, celles de droite avec.

culent le taux de variation d'une variable de type `varying`[11]. C'est à dire qu'étant donné un fragment, `dFdx(v)` donne la différence entre les valeurs de v du fragment courant et de celle du fragment voisin selon l'axe x, `dFdy(v)` donne la différence avec le voisin selon l'axe y. Si on les utilise sur la variable représentant la position du fragment dans le repère monde, on peut déterminer la position de fragments voisins sur la surface.

Notre approximation consiste à combiner les deux vecteurs obtenus par les fonctions `dFdx` et `dFdy` pour déterminer un vecteur sur la surface, nous cherchons ensuite les normales dans les sens positifs et négatifs autour du fragment courant en suivant la direction déterminée. Une pondération entre la normale du fragment courant et les deux des fragments voisins est effectuée. On obtient ainsi un lissage de la texture *APO*, qui n'est pas l'interpolation correcte, mais qui est obtenu en trois parcours d'octree au lieu de huit. Ce lissage est donc deux à trois fois plus rapide que la réelle interpolation, les rendus obtenus sont illustrés par les captures de la figure 4.10 suivante.

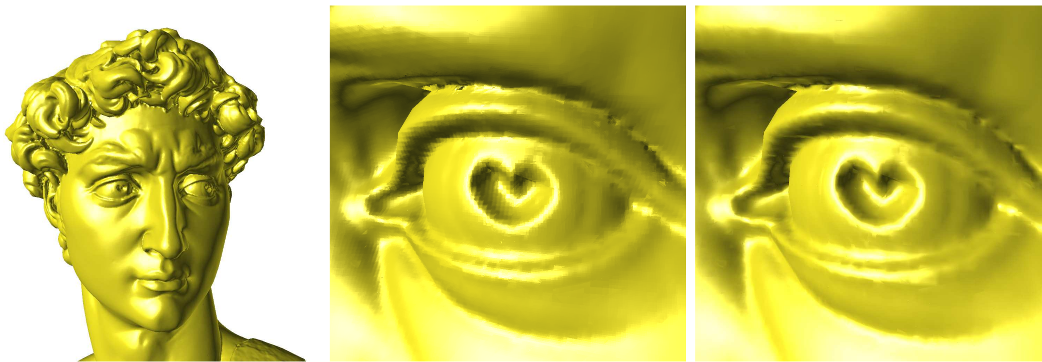


Fig. 4.10 – Approximation de l'interpolation bilinéaire par pondération avec les normales des fragments voisins. On peut remarquer que les différences ne sont visibles qu'en gros plan (la capture de droite est l'interpolation correcte, celle du milieu l'interpolation approximée), le lissage approximatif étant de bonne qualité (voir capture de gauche).

4.3.2 Rendu adaptatif

La structure hiérarchique de l'*APO*, avec ses normales moyennées aux nœuds internes (voir 3.3.2.1 page 66), peut être vue comme une hiérarchie de textures à différents niveaux de détails. L'*APO* est en quelque sorte une texture MIP-mapping, nous pouvons donc utiliser ces différents niveaux de détails lors du rendu pour éviter les hautes fréquences qui peuvent apparaitre lorsque l'objet est éloigné. Le niveau de détail le plus adapté est choisi en fonction de la distance de l'objet à la caméra. Plus précisément, on fera en sorte qu'un pixel de l'écran soit couvert par un seul nœud de l'octree. On utilise

[11]c'est à dire transmise par le vertex shader et interpolée pour chaque fragment d'une primitive graphique

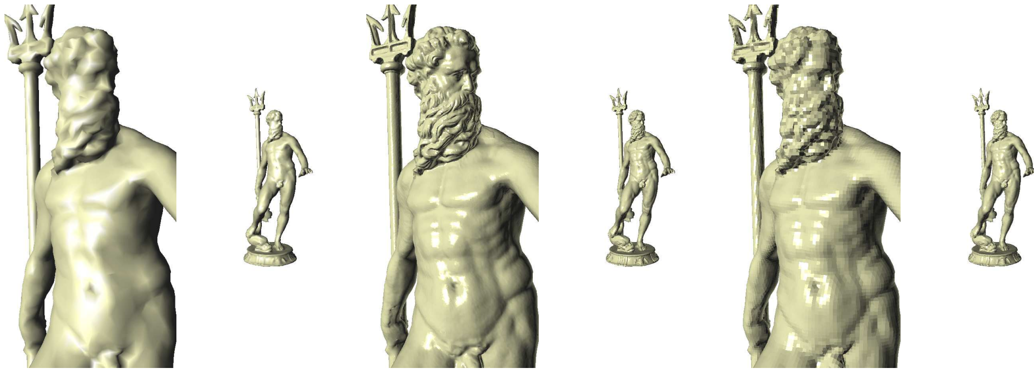


FIG. 4.11 – La différence de détail lié à une profondeur moindre se distingue moins quand l'objet est éloigné.

ainsi une normale filtrée, plutôt que les normales des nœuds les plus fins qui peuvent introduire des hautes fréquences lorsqu'un même pixel de l'écran peut abriter plusieurs nœuds.

La figure 4.11 montre des captures d'une même texture *APO* à différents niveaux de détails, placée sur un objet à distance variable de la caméra. Il est également intéressant de noter que ce rendu adaptatif sera bénéfique pour les performances, car en arrêtant prématurément le parcours de l'octree des accès textures sont économisés.

4.3.2.1 Calcul de la profondeur par utilisation des fonctions GLSL

La méthode de rendu adaptatif que nous présentons se base sur un constat simple : si deux fragments voisins d'un même objet sont issus de posiitions éloignées sur cet objet, alors c'est que l'objet est éloigné du point de vue. Au contraire, si ces deux fragments sont issus de positions proches sur l'objet, cela signifie que ce dernier est proche de la caméra. En évaluant la distance (dans le repère monde) qui sépare les positions dont sont issus deux fragments voisins, on peut estimer le niveau de détail nécessaire.

En utilisant les fonctions dérivatives déjà présentées plus haut (voir section 4.3.1.3) dFdx et dFdy sur la variable varying représentant la position des sommets dans le repère monde, on obtient la différence entre les positions dont sont issues les fragments, on estime ainsi la distance entre le fragment et ses fragments voisins. La profondeur nécessaire pour obtenir le niveau détail correspondant à la distance est ensuite calculée. Pour obtenir le niveau de détail le plus juste, on se base sur la plus petite distance donnée par soit la dérivée en x, soit celle en y. Notons avant de décrire le calcul du niveau de détail que ce calcul de distance est une approximation de la dérivée car nous n'avons pas accès aux valeurs des positions pour tous les fragments voisins. Les fonctions ne permettent donc pas de calculer une différence centrale qui aurait été plus précise. La figure 4.12

montre l'évolution du résultat des fonctions `dFdx` et `dFdy` selon l'éloignement de l'objet.

FIG. 4.12 – Evolution des fonctions dérivatives : plus l'objet est rouge, plus la dérivée locale des positions des fragments est forte.

Il reste à calculer le niveau de détail, c'est à dire la profondeur maximale lors du parcours, en fonction de cette distance de positions entre deux fragments voisins. Cette distance est notée D_{xy}. Nous avons décidé de prendre la profondeur maximale p telle que :

$$2^{-p} < D_{xy} \leq 2^{-(p+1)}$$

C'est à dire que nous choisissons comme profondeur maximale la première profondeur où les cellules de l'octree ont leur largeur plus petite que D_{xy}. Nous cherchons en effet la profondeur la plus adaptée pour le fragment, c'est à dire le pixel, courant. Prendre une profondeur plus élevée reviendrait à couvrir la distance voulue par plusieurs cellules, qui se retrouveraient projetées sur le même fragment. Comme nous l'avons déjà dit, il vaut mieux dans ces cas là utiliser le nœud parent pour éviter le sur-échantillonnage. Pour calculer p, nous utilisons la formule suivante :

$$p = \lceil \log_2(\frac{1}{D_{xy}}) \rceil$$

Ce calcul se base sur le fait que nous choisissons la cellule de profondeur p dont la largeur correspond à peu près à D_{xy}, nous avons donc (rappelons que la largeur d'une cellule de l'octree de profondeur p est de 2^{-p} ou $1/2^p$) :

$$D_{xy} \approx \frac{1}{2^p}$$

$$2^p \approx \frac{1}{D_{xy}}$$

$$p \approx \log_2(\frac{1}{D_{xy}})$$

En choisissant la valeur entière de ce résultat, nous sommes sûrs d'obtenir la première profondeur dont la largeur des cellules ne couvre pas totalement D_{xy}. La figure 4.13 illustre le principe complet du calcul du niveau de détail avec les fonctions dFdy et dFdy.

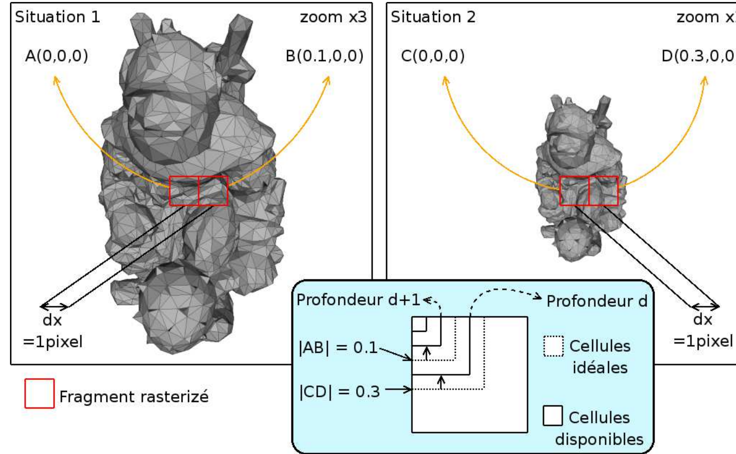


FIG. 4.13 – Les fonctions dFdx et dFdy donnent l'éloignement sur l'objet entre deux fragments voisins. La profondeur est choisie en fonction de cette distance.

Le calcul de la profondeur maximale par cette méthode est plus rapide que celui présenté auparavant, car il ne nécessite de calculer la limite maximale qu'une seule fois par fragment. Ce calcul est de plus basé sur des opérations (fonctions dérivées) rapides car implémentées matériellement.

Sur une scène complexe, comme celle présentée à la figure 4.14, composée de plusieurs objets chacun à des distances différentes, l'utilisation du rendu adaptatif permet d'obtenir des taux de rafraichissement plus grands. Sur cet exemple, avec une résolution de 1920×1125, nous obtenons des augmentations du nombres d'images par seconde de l'ordre de 15% lorsque l'on visualise la scène en plan rapproché, de l'ordre de 25% quand on la visualise en plan large. En effet les bénéfices du rendu adaptatif sont plus grand quand les objets sont éloignés et qu'il permet donc d'éviter un nombre d'accès textures conséquent en stoppant le parcours sur des nœuds intermédiaires. Ce gain se fait sans perte de qualité visuelle.

4.3.2.2 Filtrage trilinéaire

Avec la structuration hiérarchique d'octree, la mise en place du filtrage trilinéaire, c'est à dire l'interpolation entre les deux couches de détails entre lesquelles se situe le niveau de détail du fragment courant, est triviale. Cette interpolation a été proposée avec les textures MIP-map pour atténuer les transitions brutales qui se produisent lorsque l'on change de couche de détails. Cependant, nous n'avons pas remarqué d'apport après la mise en place de ce filtrage trilinéaire avec les *APO*.

FIG. 4.14 – Sur une scène complexe composée de plusieurs objets, le rendu adaptatif permet d'augmenter la vitesse d'affichage de 15 à 20% en moyenne. Le nombre d'images par seconde passe de 12 à 15 en gros plan, de 25 à 32 en plan large pour une résolution de 1920×1125.

4.4 Rendus et Performances

Nous allons dans cette partie nous intéresser aux résultats du placage des *APO* selon deux aspects. Dans un premier temps nous nous intéressons aux résultats visuels, en comparant les modèles simplifiés avec *APO* et les modèles originaux. Nous montrons les différences de qualité en fonction du choix de tolérance V^N. Dans un deuxième temps nous discutons des performances obtenues en termes de nombre d'images par seconde. Nous discutons également des limites d'utilisation des *APO*.

4.4.1 Qualité visuelle

Dans cette section seront exposées des captures d'objets simplifiés rendus avec l'utilisation de leurs *APO* pour faire apparaitre les détails des maillages originaux correspondants. Les erreurs introduites peuvent avoir plusieurs sources :

- La simplification de M apporte déjà une erreur. Notre placage d'*APO* s'apparentant à du normal mapping en ce qui concerne le calcul de l'éclairage, les silhouettes des objets rendus avec *APO* ne peuvent être les silhouettes exactes des objets originaux.
- Une autre erreur apportée par la simplification de M a déjà été traitée en partie 3.2.4.2 page 54. Si M et m sont trop éloignés, les *APO* ne peuvent pas échantillonner correctement le champ de normales de M, et donc des détails sont perdus.
- Dans le cas de surfaces très fines ayant deux côtés opposés, les *APO* ne peuvent pas capturer les normales des deux faces, puisque une seule normale est encodée dans chaque feuille ou nœud. Ce problème survient également aux angles vifs des

maillages, où il serait souhaitable d'avoir plusieurs normales. Dans les travaux de [BD02], les auteurs présentent une méthode pour sauvegarder plusieurs normales par nœuds. Chacune des 6 directions des faces des cubes peut avoir une normale représentative, repérée par un marqueur (flag). La normale à utiliser lors du rendu est celle associée à la face dont la direction est opposée au point de vue. Cette méthode était cependant implémentée en CPU. Il serait intéressant d'implémenter une telle approche sur GPU. Nous pouvons toutefois déjà entrevoir la complexité qu'imposerait une gestion d'un nombre variable de normales par nœud.

Il est difficile de mesurer et de quantifier ces erreurs introduites. Nous allons exposer différentes captures d'objets avec utilisation de textures de détails APO, nous remarquons qu'il est parfois difficile de distinguer l'original du maillage simplifié rendu avec les détails des APO. Il serait vain de calculer une différence de ces images pour essayer d'en extraire une *quantité* d'erreur. En effet, même si tous les détails de M sont présents, ils sont légèrement décalés sur m du fait de la simplification. On se retrouverait donc à calculer une erreur qui pourrait être grande, tout en étant quasiment incapable de distinguer visuellement l'objet original de son avatar avec APO. La figure 4.15 illustre cette différence entre les images des maillages originaux et simplifiés avec APO. Nous pouvons noter que si la différence d'image comme mesure d'erreur permet une première quantification de l'erreur introduite, cette approche reste une méthode naïve et imprécise. Pour mesurer de manière plus précise les erreurs introduites, il faudrait mettre en place un calcul prenant en compte la distance entre les maillages et le seuil de tolérance utilisé lors de la subdivision de l'octree. Cette mesure devrait s'effectuer non pas dans l'espace écran, mais plutôt dans l'espace objet, lors de la construction de l'APO par exemple.

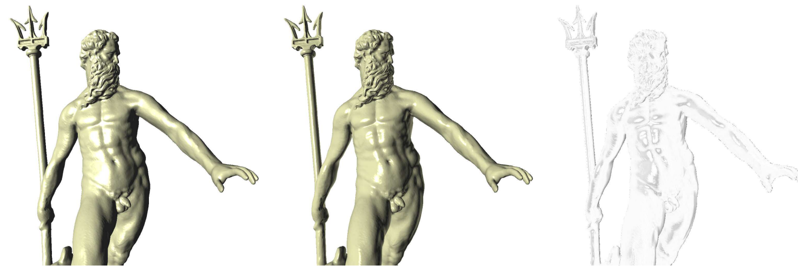
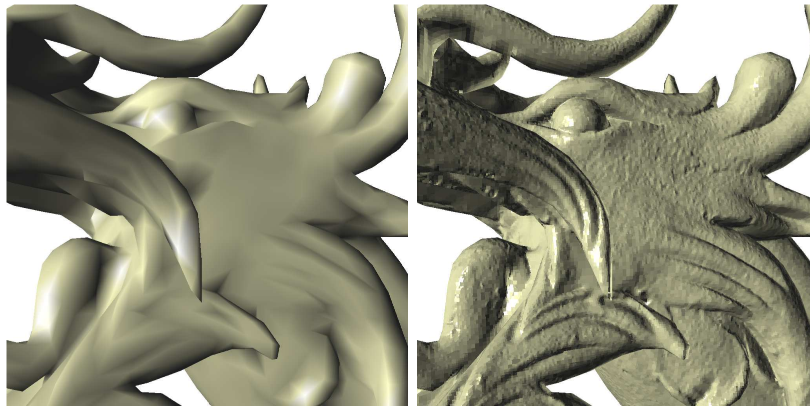
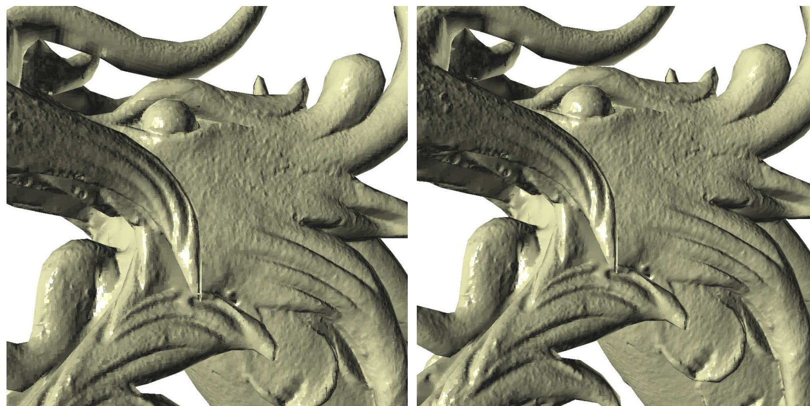


Fig. 4.15 – Gauche : maillage original de 4 millions de triangles. Centre : maillage simplifié de 30 mille polygones. Droite : différence entre les deux images. L'APO utilisée a été construite avec une tolérance de 0.2, la texture est de dimension 4096×2680.

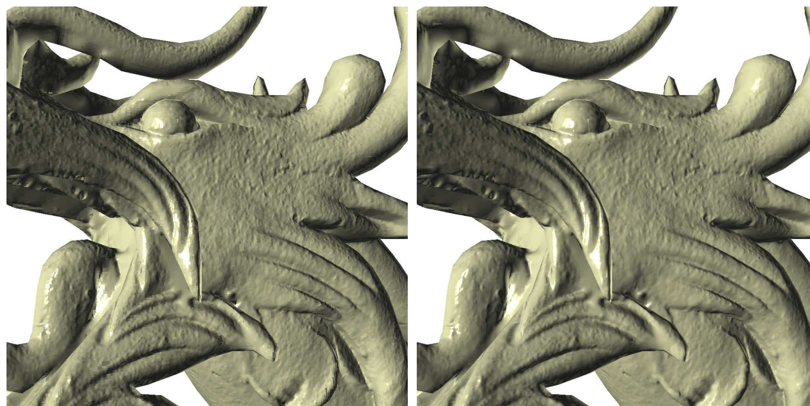
Les erreurs visibles sont celles des silhouettes, et celles de la qualité d'échantillonnage, qui est elle contrôlée par la valeur de la tolérance V^N. Les effets de la variation de la valeur de tolérance V^N sont illustrés par les captures de la figure 4.16. D'une manière générale on peut remarquer que les erreurs, ainsi que l'aliasing sont visibles quand l'objet est visualisé en très gros plan. D'autres captures se trouvent en annexe D.



(a) A gauche le maillage simplifié. A droite, il est rendu avec une *APO* dont la tolérance était de 0.5



(b) A gauche la tolérance est de 0.3, à droite la tolérance est de 0.1



(c) A gauche la tolérance est de 0.05, à droite la tolérance est toujours 0.05, mais le filtrage est activé

Fig. 4.16 – Différences de rendu avec des *APO* construites avec des tolérances diférentes. Les statistiques des *APO* peuvent se trouver partie 3.2.4.1 ou en annexes.

4.4.2 Performances d'affichage

Le parcours de l'octree étant effectué par le fragment shader, les performances d'affichage de notre méthode sont uniquement dépendantes du nombre de pixels de l'écran pour lesquels le parcours de l'octree est nécessaire. Ainsi, le nombre d'images par secondes diminue quand on zoome sur l'objet. Il est donc délicat de mesurer les performances d'affichage lors de la visualisation des objets, une manipulation alternant fréquemment les visualisations lointaines puis les plans rapprochés. Pour les résultats que nous présentons, nous avons décidé de nous placer dans un premier temps dans les pires des cas, c'est à dire que nous avons relevé les taux de rafraichissement lorsque les objets sont visualisés en gros plan. Dans un deuxième temps nous présentons les performances obtenues quand l'objet est visualisé dans son ensemble. La procédure de mesure consiste à positionner l'objet puis à lui faire faire une rotation selon l'axe y, on relève le FPS moyen à la fin de cette rotation.

Les résultats obtenus ont été relevés avec les maillages et *APO* correspondantes qui ont été exposés dans les tableaux de la partie 3.2.4.1 ou en annexe B. Nous présentons les résultats pour le rendu avec une textures créée avec la valeur de tolérance 0.05. Tous les rendus ont été exécutés sur un AMD 3500+ mono-cœur, la carte graphique étant une NVidia GeForce 8800 GTS avec 320 Mo de mémoire vive.

4.4.2.1 Performances en gros plan

Dans cette section les résultats présentés sont ceux obtenus lors de la visualisation en gros plan des objets. La figure 4.17 illustre pour quelques uns des maillages pour lesquels nous avons mené les expérimentations les conditions de mesure en gros plan. Le tableau 4.1 présente les résultats obtenus avec des textures dont la tolérance est de 0.05. Cette faible tolérance implique la création de textures APO détaillées (les dimensions respectives des textures utilisées pour ces mesures de performance sont indiquées dans le tableau 3.1 page 73), forçant la plupart des feuilles à être à la profondeur maximale. On essaie ainsi de se placer dans le pire des cas pour les mesures, c'est à dire en ayant un parcours d'arbre jusqu'à la profondeur maximale pour tous les fragments. Cette tolérance permet aussi d'avoir des rendus très détaillés des objets.

On peut constater que même en gros plan et sur des résolutions importantes, la manipulation de maillages avec *APO* reste interactive. En effet, le nombre d'images par seconde moyen ne descend pas en dessous de 15 images par seconde avec une résolution de 1280×1024. Nous avons constaté des chutes ponctuelles autour de 10 images par secondes, mais ces ralentissements n'empêchent pas la manipulation interactive des maillages.

4.4.2.2 Performances en plan large

Dans cette section les résultats présentés sont ceux obtenus lors de la visualisation en plan large des objets. Par plan large nous entendons que les objets sont visualisés en

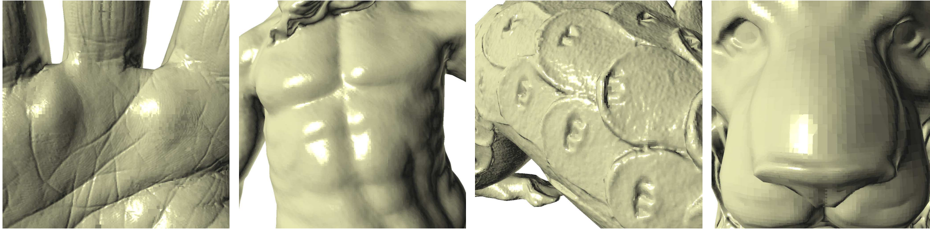


FIG. 4.17 – Exemples de maillages en gros plans utilisés pour les mesures de performances.

Modèles	**Résolution (gros plan)**		
	800 × 600	1024 × 768	1280 × 1024
Stanford Dragon	50.3 im/s	33.3 im/s	24.1 im/s
Vase Lion	38 im/s	24.8 im/s	17.4 im/s
Happy Bouddah	38.2 im/s	26.2 im/s	16.8 im/s
Moulage de main	39.7 im/s	28.2 im/s	18.2 im/s
Neptune	50.3 im/s	35.1 im/s	22.4 im/s
XYZRGB Dragon	30.1 im/s	21.5 im/s	14.8 im/s
XYZRGB Statue thaïe	38.2 im/s	29.6 im/s	18.7 im/s

TAB. 4.1 – Taux de rafraichissement en gros plan, APO construite avec la tolérance 0.05

intégralité à l'écran. Comme précédemment les résultats exposés sont obtenus avec une texture ayant été construite avec une tolérances de 0.05 (tableau 4.2).

Les taux de rafraichissement sont ici élevés, car le nombre de parcours d'octree est moindre que lorsque l'on est en gros plan. De plus, quand les objets sont moins proches du point de vue, le rendu adaptatif permet de limiter la profondeur de parcours, et donc d'économiser des accès textures coûteux.

4.4.2.3 Performances avec filtrage bilinéaire

Les deux tableaux 4.3 et 4.4 présente les performances de rendu lorsque le filtrage bilinéaire est activé. Les APO utilisées sont celles dont la tolérance V^N lors de la construction est de 0.05. Nous constatons que les performances son très dégradées, puisqu'une diminution d'un facteur 8 des taux de rafraichissement est notable. Cette diminution s'explique par les 8 parcours d'octree nécessaires pour calculer la normale interpolée de chaque fragment. Toutefois, avec des résolutions faibles et moyennes, les taux de rafraichissement permettent une manipulation interactive des objets si ceux ci sont éloignés ou visualisés en plan large. Par contre, sur des résolutions plus grandes, ou si l'on zoome sur les maillages, les taux de rafraichissement tombent sous les 10 images par seconde, voire autour de 2 images par seconde, ce qui est incompatible avec une utilisation temps

Modèles	**Résolution (plan large)**		
	800×600	1024×768	1280×1024
Stanford Dragon	141.3 im/s	97.1 im/s	66.6 im/s
Vase Lion	82.6 im/s	60.7 im/s	42.1 im/s
Happy Bouddah	111.1 im/s	82.5 im/s	58.3 im/s
Moulage de main	125.8 im/s	93.8 im/s	62.1 im/s
Neptune	121.5 im/s	84.5 im/s	70.1 im/s
XYZRGB Dragon	113.5 im/s	88.6 im/s	63.4 im/s
XYZRGB Statue thaïe	82.1 im/s	63.6 im/s	50.8 im/s

Tab. 4.2 – Taux de rafraichissement en plan large, *APO* construite avec la tolérance 0.05

réel des maillages. Il est important de noter que les mesures ont été effectuées avec l'utilisation du filtrage volumique, pas avec l'approximation présentée section 4.3.1.3 page 95 qui permettrait de multiplier par deux voire plus les vitesses mesurées.

Modèles	**Résolution (Gros plan)**		
	800×600	1024×768	1280×1024
Stanford Dragon	6.8 im/s	4.3 im/s	2.7 im/s
Vase Lion	4.7 im/s	3.1 im/s	≈ 2 im/s
Happy Bouddah	5 im/s	3.3 im/s	≈ 2 im/s
Moulage de main	5.2 im/s	3.4 im/s	≈ 2 im/s
Neptune	6.7 im/s	4.3 im/s	≈ 2 im/s
XYZRGB Dragon	3.8 im/s	2.6 im/s	< 2 im/s
XYZRGB Statue thaïe	5.3 im/s	3.5 im/s	≈ 2 im/s

Tab. 4.3 – Taux de rafraichissement, *APO* construite avec la tolérance 0.05, le filtrage bilinéaire est activé

4.4.3 Discussion

L'utilisation des textures *APO* sur des maillages simplifiés permet, hormis dans le cas où le filtrage bilinéaire est activé, de manipuler en temps réel des avatars ayant l'apparence des objets M originaux. Les performances en termes de nombre d'images par seconde sont dépendantes du nombre de fragments pour lesquels un parcours de l'*APO* est nécessaire, et dans une moindre mesure, du niveau de détail des textures. Les opérations coûteuses lors du traitement par le fragment shader sont les accès textures réalisés lors du parcours de l'octree. Ces accès, même si ils sont peu nombreux [12], ne sont pas contigus dans la texture, car les fils d'un nœud ne sont pas voisins de ce dernier dans le tableau 1D. Il n'y a donc pas de cohérence lors des accès, ce qui est la principale cause

[12] au maximum $2 \times p$ accès par fragment quand la profondeur maximale est p.

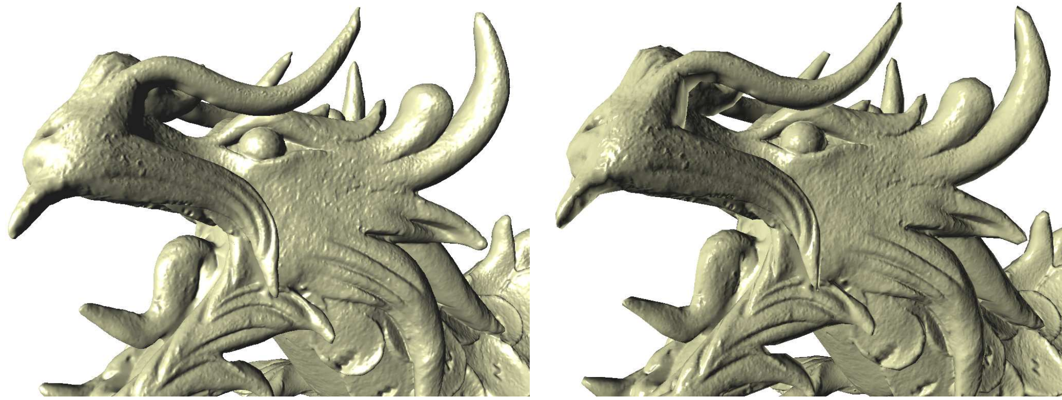
Modèles	**Résolution (Plan Large)**		
	800 × 600	1024 × 768	1280 × 1024
Stanford Dragon	18.4 im/s	12.8 im/s	8.4 im/s
Vase Lion	10.1 im/s	7.2 im/s	4.8 im/s
Happy Bouddah	14.2 im/s	10.1 im/s	6.8 im/s
Moulage de main	16.9 im/s	11.6 im/s	7.5 im/s
Neptune	14.6 im/s	11.1 im/s	8.1 im/s
XYZRGB Dragon	14.6 im/s	10.6 im/s	7.4 im/s
XYZRGB Statue thaïe	10.3 im/s	7.6 im/s	5.4 im/s

Tab. 4.4 – Taux de rafraichissement, *APO* construite avec la tolérance 0.05, le filtrage bilinéaire est activé

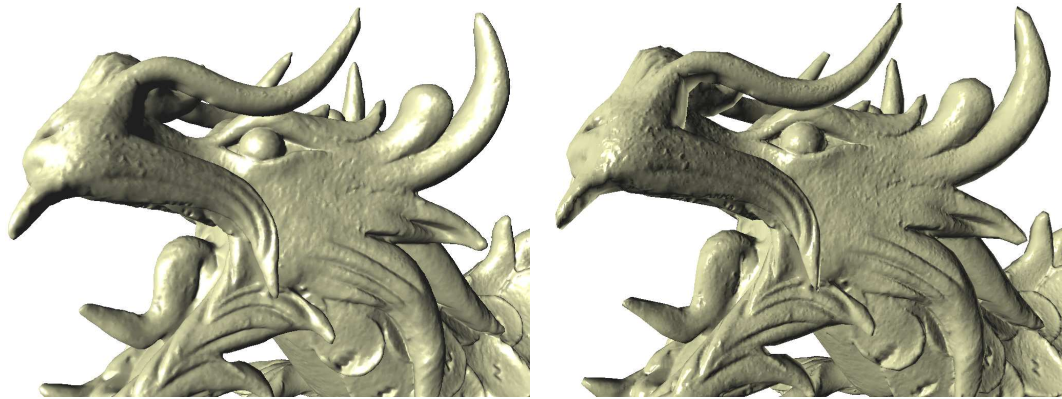
des vitesses d'affichage relativement basses que nous obtenons. Pour preuve, la limitation artificielle de la profondeur au niveau 9 lors des parcours augmente de 20% les taux de rafraichissement mesurés.

Cependant, avec des taux de rafraichissement supérieurs à 40 images par seconde, l'*APO* permet de rendre en temps réel des avatars ayant l'apparence de modèles qui ne pourraient directement être visualisés autrement. En effet, il est difficile d'avoir un rendu temps réel pour des objets composés de plusieurs millions de polygones. On peut toutefois se poser une question : n'est il pas envisageable de remplacer le modèle avec *APO* par une simplification de M de bonne qualité qui pourrait être rendue en temps réel avec de meilleures performances ? Il est évident qu'un maillage de 1 million ou 500 mille polygones peut être rendu aisément en temps réel. Il est pourtant important de noter qu'une texture *APO* peut apporter plus de détails qu'une simplification, comme on peut le voir sur la figure 4.18.

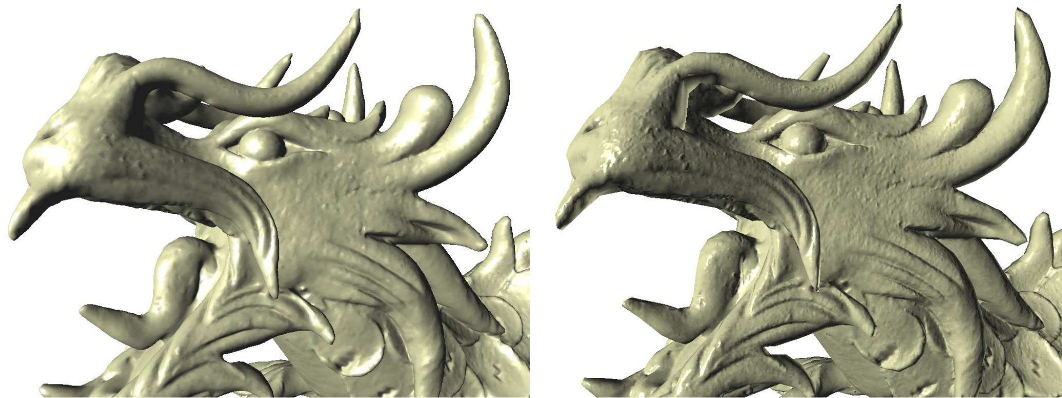
Nous pouvons également remarquer que comme on applique les *APO* sur des maillages m qui sont grandement simplifiés, et donc composés de quelques milliers de polygones, il est possible d'utiliser les objets *APO* dans des scènes d'une grande complexité. Ainsi, comme l'illustre la figure 4.14 page 101, il est possible de rendre plus d'une dizaine d'objets ayant l'apparence des originaux, ce qui serait impossible dans le cas d'utilisation de simplifications d'un million de triangles.



(a) A gauche une simplification de 3 millions de polygones, à droite un rendu avec une *APO* de dimension 4096×2621 pour une tolérance de 0.05



(b) A gauche une simplification de 1 million de polygones, à droite un rendu avec une *APO* de dimension 2048×1455 pour une tolérance de 0.3



(c) A gauche une simplification de 500 mille polygones, à droite un rendu avec une *APO* de dimension 2048×992 pour une tolérance de 0.5

Fig. 4.18 – Comparaison des simplifications géométriques et des rendus avec *APO*. On remarque que les *APO* peuvent apporter plus de détails qu'une simplification, même avec une simplification drastique de m et des V^N faibles.

Chapitre 5

Optimisation et Atlas de texture 2D

Sommaire

5.1 Optimisation des performances 109
5.1.1 Détection du nœud enveloppant les triangles 110
5.1.2 Forêt d'Octrees 111
5.1.2.1 Construction de l'octree et espace mémoire occupé 112
5.1.2.2 Calcul de l'indice du nœud de départ 112
5.1.2.3 Performances 113
5.1.3 Améliorations possibles 114
5.2 Conversion en atlas de texture 2D 115
5.2.1 Paramétrisation en atlas de triangles 115
5.2.1.1 Nombre de texels pour un triangle 116
5.2.1.2 Positionnement des triangles dans le plan 116
5.2.2 Création de la texture d'atlas 118
5.2.2.1 Rendu dans texture 118
5.2.2.2 Résultats 119
5.2.3 Rendu avec l'atlas APO 121
5.3 Conclusion 122

5.1 Optimisation des performances

Nous avons montré que les performances du rendu avec *APO* étaient limitées par le nombre d'accès textures nécessaires pour le parcours des octrees. Ces accès textures n'étant pas contigus, les texels accédés ne se trouvent pas dans le cache et donc les accès mémoires ne sont pas cohérents, ce qui explique les taux de rafraichissements moins performants que ceux obtenus avec du texturage 2D classique pour lequel un seul accès texture est suffisant. Pour améliorer ces performances une seule solution directe serait

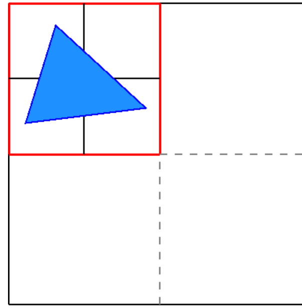


Fig. 5.1 – Pour tous les fragments issus du triangle ici repéré en bleu, le parcours de l'octree peut démarrer de la cellule marquée en rouge.

de diminuer le nombre d'accès texture. Les positions des cellules filles étant déduites du nœud courant lors du parcours, et les offsets étant calculés en fonction des masques indiquant l'existence ou non des fils, il est impossible de sauter des niveaux en fin de parcours. La seule possibilité est donc d'éviter des accès textures en ne démarrant pas ce parcours par la racine, mais par un nœud interne plus profond contenant le fragment $Frag$.

Nous avons imaginé deux méthodes pour atteindre cet objectif, les deux donnant des résultats différents. Les sections suivantes s'attachent à détailler ces méthodes d'optimisation. La dernière partie de ce chapitre sera consacrée à détailler la conversion de nos octree-textures en atlas de texture 2D, ce qui permet d'obtenir des performances équivalentes au texturage 2D, mais en perdant les avantages des APO.

5.1.1 Détection du nœud enveloppant les triangles

La première optimisation que nous avons mis en place pour diminuer le nombre d'accès texture est basée sur une idée simple : le parcours de l'APO ne démarrera pas de la racine, mais de la cellule la plus profonde qui enveloppe intégralement le triangle dont est issu $Frag$. Ce principe est illustré par la figure 5.1.

Il faut donc enregistrer pour chacun des triangles de m les informations permettant d'identifier le nœud le plus profond contenant le triangle. Comme les APO ne contiennent que les informations sur la structure arborescente de l'octree, et pas les informations spatiales sur chacun des nœuds, qui sont déduites du nœud parent lors du parcours, il faut enregistrer, outre l'indice du nœud dans le tableau 1D, les informations permettant de le localiser : son centre et sa largeur. La détection de ces informations se fait lors de la phase de création des APO. Pour chaque triangle de m l'octree construit est parcouru, le parcours s'arrête dès que le triangle n'est plus totalement enveloppé dans une cellule. Les informations de celle-ci sont alors associées au triangle.

Pour associer ces informations à chaque triangle et les transmettre au fragment shader lorsqu'un fragment du triangle est traité, les informations sur le nœud sont liées au triangle sous forme de coordonnées de textures. Une coordonnée de texture est composée de deux valeurs flottantes, or nous en avons besoin de cinq[13]. On utilise donc le Multi-Texturing pour pouvoir associer plusieurs coordonnées de textures aux triangles de m. Avec trois coordonnées de texture on peut enregistrer toutes les informations nécessaires, la sixième valeur peut par exemple être utilisée pour sauvegarder la profondeur du nœud. Les coordonnées de textures étant interpolées pour chaque fragment, les informations sur la cellule sont écrites aux trois sommets du triangle de sorte que l'interpolation n'altère pas ces informations. Notons également que les informations de la cellule sont associées aux sommets pour chacun des triangles. Si un sommet appartient à deux triangles et que ceux-ci ne sont pas contenus dans la même cellule, il faut deux jeux de coordonnées différents pour ce sommet. Les coordonnées de textures sont donc associées *par sommet et par triangle*.

L'utilisation de cette optimisation n'a pas produit de résultats réellement convaincants. Ceci s'explique par le fait que les triangles de m, qui est un maillage simplifié, sont relativement grands par rapport au cube unitaire, et donc les cellules qui les enveloppent entièrement sont rarement dans des couches au du niveau 3. De plus, la profondeur de ces cellules de départ est évidemment dépendante de la position des triangles à l'intérieur de l'octree : si un triangle, même petit se trouve traversé par un des plans médians du cube de l'octree, la cellule la plus profonde qui le contient est la racine. Sans être toujours dans ce cas extrême, la position des triangles implique une limitation quant au nombre d'accès textures économisés. De plus, cette méthode alourdit sensiblement la gestion du rendu APO, puisqu'il faut prendre en compte les multiples coordonnées textures associées aux triangles.

Ces constats et les résultats décevants nous ont poussé à trouver une autre optimisation indépendante du maillage m, qui permet d'économiser de manière sûre des accès textures. Nous présentons cette optimisation dans la section suivante.

5.1.2 Forêt d'Octrees

Pour pouvoir se dispenser d'un nombre fixe et défini d'accès textures, il faut une méthode indépendante de la géométrie du maillage m. La solution retenue est de proposer une optimisation basée sur la structure d'octree des APO. Le parcours de l'octree par le fragment shader démarre de la racine car il est impossible de deviner directement la feuille abritant chaque fragment $Frag$, l'arbre n'étant pas complet. Il est évidemment impensable de compléter entièrement l'octree, on ne pourrait pas faire tenir un arbre complet, ne serait-ce composé que de neuf niveaux seulement, en mémoire centrale lors de la construction, puis dans une texture 2D ensuite. Par contre, il est possible de compléter les premiers niveaux de l'arbre, jusqu'à une profondeur que nous notons Pc, il

[13] indice, largeur et les trois coordonnées du barycentre

est alors possible de calculer l'indice de la cellule de niveau Pc qui contient $Frag$. Avec cette solution, $Pc \times 2$ accès textures sont évités dans le cas d'utilisation de l'encodage des nœuds sur deux texels. On peut voir cette optimisation comme une forêt d'octrees enveloppant le maillage m, chacun ayant sa racine sur une cellule du niveau Pc.

5.1.2.1 Construction de l'octree et espace mémoire occupé

Lors de la construction de l'octree initial (voir section 3.2.2 page 46) on force l'existence de chacun des 8 fils d'un nœud même si ceux-ci n'intersectent pas m, on obtient ainsi un arbre complet sur les premiers niveaux. Il se pose alors la question du choix de la valeur de Pc. Dans cette optimisation, Pc est typiquement un paramètre d'ajustage de performance en relation avec la taille des APO. Plus Pc sera grand, plus le gain de performance augmentera, mais la taille des textures augmentera elle aussi. On peut estimer le nombre de texels occupé par un arbre complet avec la formule suivante :

$$N_{Pc} \approx (\sum_{i=0}^{Pc-1} 8^i) \times 2 + 8^{Pc}$$

Nous présentons seulement une estimation car la majorité des nœuds du niveau P_c n'intersectent pas m, et sont donc des feuilles encodées sur un seul texel, tandis que celles des niveaux inférieurs sont des nœuds sauvegardés sur deux pixels dans la texture finale. On peut par contre borner le nombre de texels occupés :

$$(\sum_{i=0}^{Pc-1} 8^i) \times 2 + 8^{Pc} < N_{Pc} < (\sum_{i=0}^{Pc} 8^i) \times 2$$

La valeur N_{Pc} est cependant plus proche de la partie gauche de l'inéquation, car un faible pourcentage des cellules du niveau Pc intersecte m, seules ces cellules des nœuds internes (les autres n'intersectant pas le maillage ne peuvent être subdivisées et sont donc des feuilles de l'octree). Cette équation est valable dans le pire des cas, c'est à dire quand les nœuds internes sont encodés sur deux texels. Dans le cas où l'on ne stocke pas les valeurs moyennes des normales aux nœuds internes, le nombre de texels supplémentaires est simplement $\sum_{i=0}^{Pc} 8^i$.

5.1.2.2 Calcul de l'indice du nœud de départ

Avant de démarrer le parcours de l'octree, l'indice de la racine du sous-octree contenant $Frag$ doit être déterminé. Les informations spatiales (centre et largeur) doivent également être déduites. Nous utilisons donc la même boucle que pour le parcours vu précédemment (voir section 4.2.7 page 90), à la différence qu'il n'y pas d'accès texture pour déterminer l'offset vers le fils que l'on veut atteindre.

L'indice est trouvé par récurrence en calculant à chaque étape l'indice du nœud fils en fonction de l'indice courant. Si ind est l'indice courant de la cellule dans la couche p, l'indice de son fils c dans la couche $p+1$ est obtenu par $ind_{p+1} = ind_p \times 8 + c$. En

effet, chacune des cellules précédent *ind* dans le tableau en largeur d'abord aura 8 fils, on ajoute ensuite c pour atteindre le fils souhaité dans la fratrie de *ind*. L'indice calculé ainsi représente uniquement l'indice du nœud au sein de la couche n, il faut donc rajouter la somme des cellules des premières couches pour obtenir l'index réel de la cellule. La boucle GLSL est donc la suivante (les fonctions utilisées ont été décrites à la section 4) :

```
int cindex = 0;
int pow8 = 1;
int nbNoeuds = 1;

for(int i=0 ; i<n-1 ; ++i) {
    // calcul de l'indice dans la couche suivante
    fille = celluleFille(Center, fragPosition);
    cindex = cindex * 8 + fille;
    // mise à jour indice réel
    index = (cindex + nbNoeuds) * 2;
    // mise à jour du nombre total de noeuds
    nbNoeuds += 8 * pow8;
    pow8 *= 8;
    // mise à jour du centre de la cellule
    ...
    //
    ++depth;
  }
```

La variable pow8 permet d'effectuer la somme $\sum_{i=0}^{n} 8^i$ calculant le nombre total de nœuds, qui est initialisé à 1 pour représenter le nombre de nœuds de la couche 0. On multiplie par deux l'indice réel pour représenter le fait que chaque nœud occupe deux texels. Cette boucle s'arrête au niveau $n-1$, car la couche de profondeur n est composée de nœuds et de feuilles, il est donc impossible de prédire l'indice de la cellule contenant le fragment[14]. Après cette boucle, le parcours classique de l'*APO* peut avoir lieu en démarrant à l'indice calculé. Nous commentons dans la section suivante les gains de performance obtenus avec cette méthode.

5.1.2.3 Performances

Nous présentons dans cette section les performances obtenues avec l'optimisation dite des forêts d'octree. Pour les expérimentations, nous avons positionné Pc à la valeur 6, ce qui ajoute au maximum 600 mille texels à la texture. En termes de stockage disque, ces texels supplémentaires sont peu coûteux : tous les texels étant identiques[15], la compression PNG permet d'absorber ce surplus de texels en n'augmentant la taille des textures sur le disque que très légèrement, de moins de 1 Mo en général. Le tableau 5.1 suivant montre les taux de rafraichissement obtenus, dans les conditions suivantes : texture construite avec une tolérance de 0.05, profondeur maximale de l'arbre de 13. La résolution choisie pour le rendu est de 1280×1024.

[14] à moins que les nœuds ne soient encodés que sur un seul pixel

[15] ce sont quasiment tous des nœuds internes n'ayant pas de normale moyenne, car ils n'intersectent pas le maillage, avec les masques indiquant que tous leurs fils existent et sont eux même des nœuds internes. Seuls les nœuds intersectant le maillage, représentant un faible pourcentage de l'arbre complet, ont un codage différent.

Modèles	**Résolution** 1280 × 1024**, tolérance** 0.05			
	Sans filtrage		**Avec filtrage**	
	Plan large	Gros plan	Plan large	Gros Plan
Stanford Dragon	108.5 im/s	35.9 im/s	33.4 im/s	11.5 im/s
Vase Lion	61.5 im/s	27.2 im/s	23.1 im/s	8 im/s
Happy Bouddah	87.6 im/s	27.3 im/s	35.4im/s	10.8 im/s
Moulage de main	85.2 im/s	25.3 im/s	36 im/s	10.9 im/s
Neptune	100.3 im/s	35.6 im/s	43.5 im/s	13.3 im/s
XYZRGB Dragon	96.8 im/s	24.6 im/s	42.9 im/s	9.2 im/s
XYZRGB Statue thaïe	71.8 im/s	23.2 im/s	31.9 im/s	11.1 im/s

Tab. 5.1 – Taux de rafraichissement avec l'optimisation *forêt d'octrees*. Comparativement aux résultats présentés dans les tableaux de la section 4.4 (page 101) les performances sont en moyenne 40% supérieurs.

Nous pouvons constater grâce à ce tableau que les gains de performances sont substantiels comparativement au placage classique sans utilisation de l'optimisation, puisque nous obtenons au minimum des taux de rafraichissements en moyenne supérieurs de 40%, parfois même plus. Ceci s'explique par le fait que toutes les feuilles ne sont pas à la profondeur 13, une majorité de celles-ci sont placées dans des niveaux inférieurs. Pour ces feuilles, avec $P_c = 6$, l'optimisation forêt d'octree permet d'éviter presque la moitié des accès textures nécessaires pour le parcours de l'arbre. En choisissant une valeur de 5 pour P_c, le gain de performance est approximativement de 30%, pour une surcharge de moins de 75 mille texels dans la texture, ce qui est négligeable étant donné que les textures sont composées de millions de texels.

Cette optimisation permet donc d'augmenter la vitesse d'affichage, même si il est toujours difficile d'obtenir un rendu interactif en gros plan lorsque l'on visualise les objets avec du filtrage bilinéaire.

5.1.3 Améliorations possibles

L'optimisation présentée dite des forêts d'octree a prouvé son efficacité quant à l'amélioration des vitesses d'affichage obtenues. Cependant, cette optimisation se fait au dépend de la taille des textures. Or, il est possible d'envisager le même gain de performances sans alourdir la taille des APO. En effet, comme l'arbre est complet sur les Pc premiers niveaux, il serait possible de calculer directement l'indice du nœud interne de profondeur Pc contenant le fragment affiché, sans avoir le calcul par récurrence que nous avons présenté.

En supposant connue cette formulation directe du calcul de l'indice, il serait possible de ne pas stocker les premières couches de l'octree dans l'APO, et donc de ne pas alourdir

la structure de l'octree avec les branches inutiles rajoutées par notre optimisation. Les normales moyennes des nœuds internes des profondeurs inférieures à Pc seraient dans ce cas perdues. En même temps que le calcul de l'indice, les informations spatiales de la cellule devront également être calculées directement sans s'appuyer par une mise à jour par récurrences des informations des nœuds parents.

5.2 Conversion en atlas de texture 2D

Les textures APO nécessitent pour être exploitées une carte graphique supportant les shaders. Il est impossible d'utiliser les APO sur les autres types de carte, c'est pourquoi nous avons proposé une méthode de conversion des APO en textures 2D classiques. Cette conversion ne s'appuie pas sur une paramétrisation globale de la surface des maillages, mais sur un placement individuel des triangles dans un atlas de polygones, méthode utilisée dans les travaux de [CMSR98, CMR+99]. La conversion en textures 2D des APO fait perdre les avantages liées à la structure hiérarchique de l'octree, c'est à dire l'échantillonnage et le rendu adaptatif, ainsi que la non nécessité d'utiliser des coordonnées de texture. Nous verrons que les textures converties occupent plus d'espace mémoire que leurs homologues APO. Cependant, le passage en texture 2D classique fait que les taux de rafraichissement des rendus sont désormais ceux d'un placage de texture classique, nécessitant un seul accès texture par fragment.

Les sections suivantes exposent les deux étapes de conversion des APO, qui sont sont dans l'ordre chronologique la paramétrisation de m en un atlas de polygones, puis le remplissage de cet atlas par les normales des nœuds de l'octree. La première étape est effectuée par le CPU, la seconde par le GPU. Tout au long de ce chapitre, nous commentons les résultats de cette conversion en texture 2D des APO, que ce soit en termes d'occupation mémoire, de temps de création ou de rendu. Enfin, avant de commencer, notons que nous faisons un léger abus de langage en nommant cette section *Conversion en atlas de texture 2D* puisque nous présentons plutôt un algorithme de création d'atlas de texture 2D à partir de la structure d'octree. En effet, nous ne convertissons pas directement une texture APO en une texture 2D classique, mais nous créons un atlas de triangles qui sera *colorié* par les valeurs de normales lues dans l'APO. Cette création de l'atlas de texture se positionne donc directement après la création de la texture APO, à la suite de l'encodage puis de la sauvegarde en un tableau 1D[16] de l'octree construit. Notons que pour créer un atlas de polygones avec notre méthode, il faut absolument un GPU capable d'effectuer le parcours de l'APO.

5.2.1 Paramétrisation en atlas de triangles

Cette étape, effectuée sur le CPU, consiste à trouver la position dans l'atlas de chaque triangle du maillage simplifié m. Il faut pour ce faire déterminer la surface nécessaire pour représenter chaque triangle dans la texture, puis un positionnement des triangles

[16] puis en texture 2D

dans un plan qui limite l'espace inoccupé. Ce plan deviendra la texture 2D de l'atlas. On appelle cette étape paramétrisation 2D de m bien que l'on n'applique pas une réelle paramétrisation par morceaux ou autre de la surface, mais un simple *packing* des triangles dans une texture. On calcule également dans cette étape les coordonnées de texture de chaque triangle. Nous utilisons une structure intermédiaire pour représenter les triangles de m dans la texture. Celle-ci contient une référence du triangle qu'elle représente, les coordonnées dans l'espace monde des sommets du triangles et des informations concernant le positionnement du triangle dans le plan texture. La collection de ces structures représentatives des triangles dans le plan constitue la structure de l'atlas de triangles 2D.

5.2.1.1 Nombre de texels pour un triangle

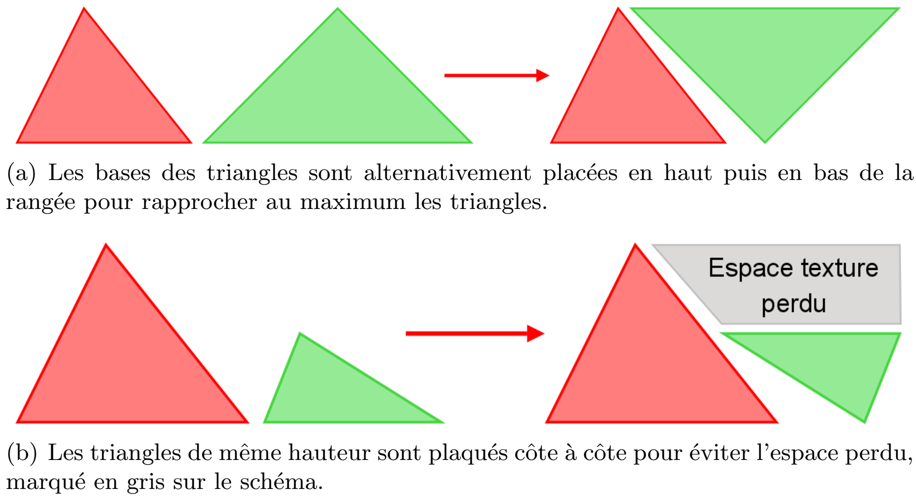
Pour représenter chaque triangle de m dans l'atlas de polygones, il est nécessaire de savoir combien de texels seront utilisés pour restituer tous les détails de chacun d'entre eux. Comme une texture 2D ne permet pas comme une APO un échantillonnage adaptatif, tous les texels d'un même triangle doivent avoir la même dimension. Dans l'octree cependant, les cellules couvrant un triangle peuvent être de profondeurs, et donc de dimensions différentes si le niveau de détail du maillage M n'est pas uniforme sur la surface du triangle. Pour simuler l'échantillonnage uniforme, et ainsi compter le nombre d'échantillons couvrant un triangle de m, les cellules intersectant chaque triangle sont subdivisées jusqu'à ce qu'elles aient toutes la même dimension, c'est à dire la profondeur maximale des feuilles couvrant le triangle dans l'APO.

Ces subdivisions temporaires peuvent être évitées en faisant une approximation. En connaissant la profondeur maximale des feuilles intersectant le triangle, une estimation du nombre de cellules de même dimension recouvrant l'aire du triangle peut être effectuée. Cete estimation est cependant moins précise. Dans la texture finale, nous décidons d'affecter un texel pour chaque feuille de profondeur maximale de l'octree qui intersecte le triangle.

Il est à noter que la taille d'un triangle dans l'atlas de texture ne dépendra pas de sa taille à la surface de l'objet 3D mais de la dimension du plus petit nœud l'intersectant et donc de la richesse de détails à sa surface.

5.2.1.2 Positionnement des triangles dans le plan

Une fois que l'on sait combien de texels doivent couvrir chaque triangle, on possède une estimation du nombre total de texels de la texture 2D finale, ce qui permet d'en estimer ses dimensions. L'étape suivante est donc de positionner la liste des triangles dans le plan 2D. L'objectif principal est d'agencer au mieux les triangles entre eux pour éviter les zones inoccupées dans la texture. Notre algorithme fonctionne sur une idée simple : les triangles de même hauteur sont placés côte à côté pour former des rangées, leurs bases étant alternativement alignées sur le haut ou sur le bas de la rangée, dans



(a) Les bases des triangles sont alternativement placées en haut puis en bas de la rangée pour rapprocher au maximum les triangles.

(b) Les triangles de même hauteur sont plaqués côte à côte pour éviter l'espace perdu, marqué en gris sur le schéma.

Fig. 5.2 – Principes du packing de triangle dans l'atlas de texture.

le but de limiter l'espace entre deux triangles successifs. Le principe est illustré par la figure 5.2. La première étape du placement consiste donc à trier les triangles selon leur hauteur. Ce tri évite de positionner deux triangles de hauteur différente à côté, ce qui ferait perdre trop de place dans la texture (voir figure 5.2).

Avant de trouver les coordonnées d'un triangle dans la texture, il faut donc calculer la rotation à appliquer à celui-ci pour qu'il soit coplanaire au plan supportant l'atlas, puis la rotation à appliquer pour aligner la base sur soit le haut soit le bas de la rangée. On applique ensuite les translations pour placer les triangles dans le plan. On choisit arbitrairement de placer le plan texture dans le plan $(\vec{x}, \vec{y})$.

Dans [CMSR98, CMR+99], les triangles sont déformés de façon à faire correspondre les angles des arêtes et ainsi éviter toute place perdue entre les triangles. Nous avons décidé de ne pas adopter ce comportement qui rajoute de légères distortions dans la texture. Avec notre méthode, les triangles sont juxtaposés, jusqu'à quasiment coller les sommets mais en évitant tout chevauchement. Cet algorithme grossier donne de bons résultats, cependant une optimisation consisterait à chercher non pas le triangle suivant, mais le triangle parmi les n suivants dont l'angle correspondrait le mieux à l'angle du triangle qui vient d'être inséré. Chaque fois qu'une rangée de la texture est complétée, on positionne le triangle suivant dans la rangée supérieure.

À la fin de cette étape, nous possédons une liste de structures qui indiquent pour chaque triangle les coordonnées réelles de ses sommets dans le maillage m, mais également les coordonnées après les rotations et translations qui permettent de positionner le triangle dans l'atlas. Cette collection représente en quelque sorte la structure de l'atlas,

puisque nous avons en sus les dimensions $dim_x \times dim_y$ de la texture, la surface en texels de chaque triangle ayant été calculée. Les coordonnées 2D dans l'atlas représente les coordonnées de textures associées au sommet. Le maillage m est enrichi de ces coordonnées de texture pour pouvoir exploiter l'atlas lors du rendu.

5.2.2 Création de la texture d'atlas

La première étape a permis de calculer la structure de l'atlas, en s'appuyant sur la structure de l'octree supportant l'*APO*. La seconde étape que nous décrivons dans cette section a pour but de créer physiquement l'atlas en remplissant la structure par les normales encodées dans l'*APO*. Ce dessin de la texture va s'appuyer sur la structure calculée précédemment, mais également sur l'*APO* construite pour m et sur la technologie openGL de rendu sur texture. Le principe est de faire un rendu éclaté du maillage m avec les positions dans la texture 2D, de façon à obtenir une image plane, mais en utilisant les coordonnées réelles de m pour faire un parcours de l'*APO*. On affecte ainsi à chaque pixel la couleur RGB qui encode la normale du fragment. Ce rendu ne se fait pas à l'écran, mais dans une texture qui est récupérée : c'est l'atlas de polygone calculé à partir de l'*APO*.

5.2.2.1 Rendu dans texture

L'utilisation de la technologie openGL des FBO, pour *FrameBuffer Object* permet de détourner le rendu classique pour faire du rendu offscreen, et plus particulièrement du rendu dans des textures[17]. Un rendu des triangles est effectué dans un FBO, on récupère son contenu pour sauvegarder la texture d'atlas. Pour chaque triangle, deux jeux de coordonnées sont disponibles : les coordonnées dans le plan 2D, et les coordonnées originales des sommets du triangles dans m. On cherche à associer à chaque texel de la texture la normale représentative associée, c'est à dire la normale sur la surface du maillage m, il faut donc que le parcours de l'*APO* s'effectue avec les coordonnées du fragment original, et pas avec les coordonnées dans le plan 2D. Le *fragment shader* est donc modifié : on associe les coordonnées originelles aux coordonnées de textures du sommet, et le parcours de l'octree se fait avec les coordonnées de textures du triangle. Ces dernières étant interpolées pour chaque fragment, le parcours de l'*APO* est effectué pour chaque texel avec sa position originale sur m, ce qui permet d'associer correctement chaque texel à sa normale représentative. Le pseudo code suivant illustre le principe utilisé :

```
...
glDisable(GL_LIGHTING);
for(int i=0 ; i<listeTriangles->taille() ; ++i) {
  for(int s=0 ; s<3 ; ++s) {
    glTexCoord3fv(listeTriangles[i]->positionOriginale[s]);
    glVertex3fv(listeTriangles[i]->positionTexture2D[s]);
  }
  glEnd();
}
```

[17]Cette technologie existe aussi sous DirectX, sous l'appellation render targets model

Le pseudo code du shader utilisé pour le dessin des triangles est le suivant :

```
// coordonnées de textures interpolées envoyées par le vertex shader
varying vec3 texCoordsPosition;
...
vec3 Normale = parcoursAPO(texCoordsPosition.xyz, APO);
...
// La couleur encodant la normale est associée au fragment
gl_FragColor = Normale.rgb;
```

Pour ne pas altérer les normales encodées, il n'y pas d'éclairage calculé lors du rendu des triangles. En effet, pour récuperer directement le code RGB qui sauvegarde la normale de chaque fragment, il faut juste lire la couleur dans l'APO, et ne pas la modifier par un calcul d'éclairage. De plus, une projection orthogonale est appliquée lors du rendu, pour assurer que les triangles rendus aient la même taille que celle déterminée lors du calcul de la structure de l'atlas.

Un des problèmes quand un atlas de textures est utilisé, c'est que les frontières entre les triangles lors du placage de l'atlas sur m peuvent être visibles, surtout dans le cas de l'utilisation du filtrage bilinéaire, où les texels sont interpolés avec les voisins pour obtenir le texel final à appliquer. Dans [LHN05], où une méthode de conversion 2D est aussi présentée, mais elle basée sur une paramétrisation de la surface du maillage, les auteurs évitent l'artefact des frontières en faisant *baver* les couleurs hors des limites des patches.

Nous adoptons la même méthode en dessinant dans la texture deux fois chaque triangle : un première fois le triangle est dessiné légèrement plus grand que ses coordonnées réelles, une seconde fois il est dessiné par dessus avec ses coordonnées réelles à l'intérieur du triangle précédemment dessiné. On obtient ainsi une enveloppe autour du triangle qui permet lors du filtrage de ne pas effectuer l'interpolation avec des texels vides. La figure 5.3 montre les effets de l'artefact et de la solution retenue pour les éviter. Lors du calcul de la structure de l'atlas, ce double dessin des triangles est pris en compte en écartant de quelques pixels supplémentaires chaque triangle, de façon à ce que lorsqu'ils sont une première fois dessinés plus grand que leur échelle réelle, il n'y ait pas de recouvrement des ces enveloppes étendues.

5.2.2.2 Résultats

L'algorithme de placement des triangles dans l'atlas, bien que simpliste, permet de créer des atlas minimisant de manière suffisante les espaces perdus dans la texture. La figure 5.4 montre deux atlas, pour des maillages de 600 et 3000 polygones, pour lesquels on peut observer l'espace perdu, qui est l'espace non occupé par des triangles de m.

Comme nous l'avons déjà évoqué, une approche pour minimiser de manière plus optimale consisterait à chercher dans les triangles suivant celui dont l'angle correspond le mieux à l'angle laissé par le dernier triangle inséré. Pour des raisons d'efficacité la

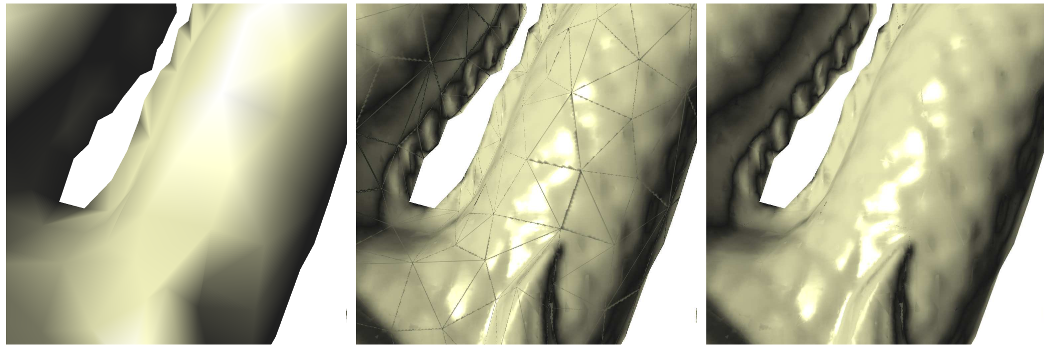


FIG. 5.3 – A gauche le maillage non texturé, au centre le maillage est texturé avec l'atlas. On aperçoit les frontières entre les triangles. À droite, l'atlas a été rempli en dessinant deux fois chaque triangle pour éviter la visibilité des frontières. Les frontières sont moins visibles car les triangles de l'atlas sont entourés par des texels non vides grâce à leurs dessins à plus grande échelle autour d'eux.

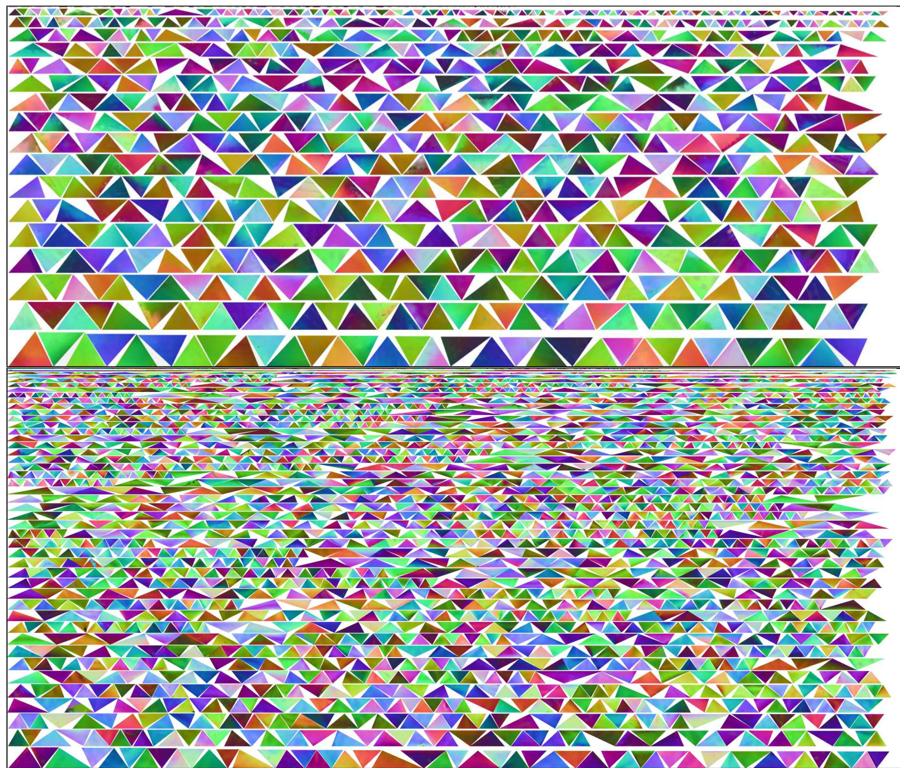


FIG. 5.4 – Exemples d'atlas de textures pour des maillages de respectivement 600 et 3000 polygones.

recherche doit se limiter à un nombre limité de triangles, au risque de ralentir grandement la création de l'atlas en rendant l'algorithme de class $\Theta(n^2)$. Dans les travaux de [CMSR98, CMR+99], les textures créées ne présentaient aucun espace perdu, car les triangles sont adaptées à des dimensions prédéfinies agencées parfaitement dans la texture. Cette méthode implique toutefois une déformation des triangles qui peut faire apparaitre des effets d'aliasing si les déformations sont trop grandes.

L'algorithme simpliste que nous avons mis en place est rapide, il ajoute entre une et deux minutes au temps de création des *APO* classiques. Le tableau 5.2 montre les statistiques de création pour deux maillages différents. Nous pouvons constater dans celui-ci que le désavantage majeur de l'atlas est l'augmentation de la taille des textures créées, qui peuvent doubler les dimensions des textures générées. Cette augmentation s'explique en partie par l'espace perdu entre les triangles, mais surtout par la perte de l'échantillonnage adaptatif, qui permet d'avoir des feuilles de dimensions différentes couvrant un même triangle. Sans cette adaptativité, construire une texture avec un pas d'échantillonnage fixe revient presque à créer un octree de profondeur maximale, 13 dans les exemples du tableau 5.2. Sur les *APO* les plus détaillées, il est même impossible de créer les atlas sans réduire l'échantillonnage, c'est à dire en n'allouant pas un texel par feuille de l'octree intersectant chaque triangle, mais par exemple un taux de 0.5 texel par feuille, au risque de perdre des détails dans les textures. Il faut toutefois pondérer cette augmentation par le fait que les atlas de textures n'ont pas besoin d'être sauvegardés au format RGBA, mais seulement au format RGB, le canal alpha indiquant les informations sur la structure de l'arbre étant désormais inutile. Cependant les tailles en octets des images sont tout de même doublées, au moins, par rapport aux équivalents *APO*.

Maillages	*APO*		Atlas	
	Temps	**Dimension**	**Temps**	**Dimension**
Stanford Dragon (3K/870K)	27s	1024×356	1min10s	8192×2037
Neptune (15K/4M)	1min51s	1024×947	2min04s	4096×2496

Tab. 5.2 – Temps de création et dimensions comparatives des atlas de texture générés. La profondeur maximale de l'octree est limitée à 13, la tolérance est fixée à 0.03.

5.2.3 Rendu avec l'atlas APO

Si les atlas augmentent considérablement la taille des textures utilisées, les performances sont quant à elles largement supérieures aux résultats obtenus avec les *APO*. D'une dizaine d'accès texture environ par fragment pour le parcours de l'*APO*, le rendu passe à un accès direct pour le rendu avec l'atlas de triangles. Les performances sont donc similaires à celles que l'on peut attendre d'un texturage 2D classique, c'est à dire que les taux de rafraichissement peuvent dépasser 200 images par secondes. De plus, le rendu avec atlas permet une manipulation interactive dans les conditions où le rendu

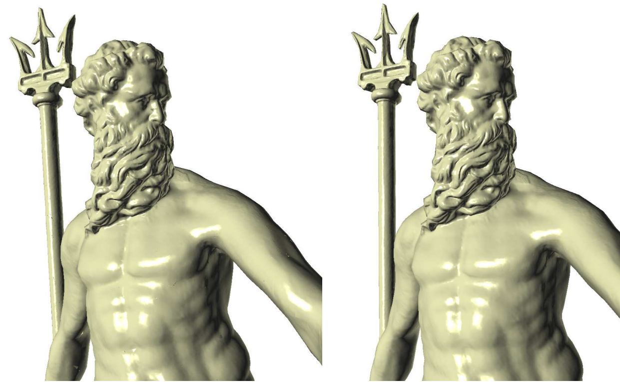
avec *APO* pouvait souffrir de baisse de performances : gros plan et filtrage bilinéaire activé. Tous ces avantages sont bien évidemment liés à la nature 2D de l'atlas, qui est donc un texturage classique implémenté en hardware sur les cartes graphiques.

En termes de qualité visuelle de rendu, nous n'avons pas constaté de perte de détails dans les atlas par rapport aux détails contenus dans les *APO*. Les captures de la figure 5.5 illustrent une comparaison de rendu avec *APO* et atlas, tout deux construits avec les mêmes paramètres. La différence notable entre les deux rendus est l'apparition des frontières entre les triangles que l'on peut déceler avec le rendu par atlas. En effet, le dessin d'un triangle étendu autour de chaque triangle de l'atlas permet d'avoir des valeurs non nulles pour effectuer le filtrage bilinéaire aux arêtes des triangles, mais les échantillons utilisés pour ce filtrage ne sont pas les réels voisins des fragments. Une disontinuité qui ne serait pas présente avec une paramétrisation globale de la surface apparait donc. On peut remarquer que ces frontières ne sont réellement détectables qu'en gros plan sur les objets, et n'altèrent pas significativement la qualité du rendu.

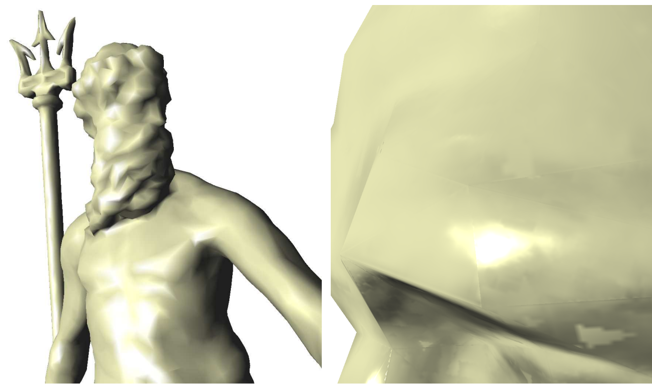
5.3 Conclusion

Nous avons présenté dans cette section deux méthodes dont le but est d'accélérer le rendu *APO*. La première a consisté à remplir les premiers niveaux de l'octree de façon à faire démarrer le parcours de l'*APO* non pas de la racine, mais d'un nœud de profondeur n, ce qui permet ainsi d'éviter des accès textures et d'améliorer les performances de vitesse de rendu d'environ 30%. Cette méthode efficace conserve toutefois les désavantages liés à la nature même de l'*APO*, c'est à dire le parcours de l'octree nécessaire pour trouver les normales associées aux fragments. Même avec des performances améliorées, la manipulation des objets en gros plans et surtout avec le filtrage bilinéaire actif reste difficile.

La deuxième solution quant à elle permet de s'affranchir de la nature intrinsèque des *APO*, en proposant une conversion en atlas de textures. Cette solution offre l'avantage d'un rendu rapide, puisqu'un seul accès texture par fragment est nécessaire pour trouver la normale. De plus cette solution est utilisable lors du rendu sur des cartes graphiques ne supportant pas les shaders. Ce gain de performance a naturellement un coût : la perte de l'échantillonnage adaptatif oblige à la construction d'atlas de textures de dimensions particulièrement grandes. Une optimisation de l'algorithme de placement des triangles dans l'atlas permettrait d'économiser de l'espace texture qui est laissé vide dans l'atlas final, mais cette économie serait non significative par rapport au nombre de texels final. La conversion en texture 2D des *APO* s'apparente fortement aux travaux de [CMSR98, CMR+99], qui traitait déjà de l'échantillonnage dans une structure 3D, mais régulière, d'un maillage original, puis du placement des triangles dans un atlas. Cependant contrairement aux travaux précédents se basant sur la taille des triangles dans l'espace objet ou sur un taux d'échantillonnage fixe, notre création d'atlas à l'aide de la structure d'*APO* permet d'adapter la taille des triangles dans l'espace texture à la densité réelle des détails à leur surface. Notre étude a donc permis de prouver l'adéqua-



(a) A gauche rendu par atlas de texture, à droite par *APO*



(b) A gauche le maillage simplifié, à droite un zoom permettant d'observer les légères discontinuités aux frontières des triangles lors du rendu par atlas.

FIG. 5.5 – Comparaison rendu par atlas et par *APO*.

tion de la structure *APO* avec la génération d'un atlas, mais de nombreuses pistes pour l'amélioration de cette conversion peuvent être explorées :

- Placement optimisé des triangles dans l'atlas, dans le but de minimiser l'espace perdu dans l'atlas. La solution est de soit chercher les triangles qui s'emboîtent le mieux entre eux, soit d'adopter la méthode de [CMSR98, CMR+99], en modifiant la forme des triangles pour les faire rentrer dans des zones prédéfinies de l'atlas.
- Calcul optimal du nombre de texels par triangle pour éviter la création de texture gigantesques.
- Gestion améliorée des discontinuités entre les triangles, bien que les artefacts générés soient similaires à ceux qui apparaissent lors du rendu par *APO*.

Troisième partie

Application

Chapitre 6

Application : transmission progressive de l'apparence dans un visualisateur web

Sommaire

6.1 Introduction . . . **127**
6.2 Visualisation de maillages sur environnement réseau . . . **129**
6.2.1 Transmission de LOD géométrique . . . 129
6.2.1.1 Maillages progressifs et compressés . . . 129
6.2.1.2 Structure hiérarchique . . . 131
6.2.2 Rendu à distance . . . 132
6.2.3 Transmission d'apparence . . . 133
6.3 Architecture APOWeb3DViewer . . . **133**
6.3.1 Mise à jour de la texture . . . 136
6.3.2 Fragment shader adapté . . . 138
6.4 Téléchargement adaptatif . . . **138**
6.5 Implémentation JOGL et résultats . . . **140**

6.1 Introduction

La visualisation d'objets 3D est un besoin gra,ndissant dans des applications réseaux telles que les environnements virtuels. Dans les applications de type base de données en ligne d'objets 3D, aussi appelées *shape repositories*[18], la visualisation et la manipulation interactive des maillages peut apporter un plus grand confort d'utilisation, une plus grande richesse de détails et d'informations qu'un ensemble de rendus fixes. Grâce à la visualisation de maillages simplifiés avec textures *APO*, les utilisateurs ont à disposition un outil qui permet de manipuler des versions dégradés des maillages ayant

[18] un exemple est le site *aim at shape* http://shapes.aim-at-shape.net/index.php

toutefois l'apparence des originaux. Ainsi le téléchargement de l'objet original n'est pas nécessaire. Cette méthode a aussi l'avantage d'assurer la confidentialité des données : les sites ne sont pas tenus de mettre à disposition les maillage originaux dans le cas où ces objets sont soumis à des contraintes de droits d'auteur ou autres. Pour ces applications réparties, il faut adapter les applications de visualisation aux contraintes liées à la nature réseau de l'environnement, c'est à dire minimiser le volume de données à transférer pour limiter l'utilisation de la bande passante, ou encore pour s'adapter à des environnements ayant des limitations sur la quantité de mémoire utilisable.

Dans cette section nous présentons une application originale des textures *APO* que nous avons proposé. Les textures de normales *APO* vont être utilisées pour effectuer une visualisation de maillages sur un environnement distribué. Cette application des *APO* est envisageable de par la structuration hiérarchique de l'encodage de l'octree : le stockage en largeur d'abord des nœuds implique que les nœuds de faible profondeur sont situés avant les nœuds de profondeur élevée, ils sont placés dans les premiers pixels de la texture *APO*. Cette structuration permet d'envisager une transmission progressive des données. Le maillage m peut être visualisé alors que les détails contenus dans l'*APO* sont téléchargés. Les détails sont plaqués progressivement sur m, au début seuls les détails grossiers seront disponibles, des raffinements successifs auront lieu au fur et à mesure du téléchargement de la texture.

Cette application des *APO* peut-être comparée à l'utilisation du format d'image JPEG progressif. Lors de l'affichage dans un navigateur d'images dans ce format, les détails des images apparaissent progresivement comme une série de différents raffinements successifs sur une image grossière, permettant au cours du téléchargement de visualiser rapidement des versions basse résolution de l'image. Ceci permet de faire patienter l'utilisateur, mais alourdit par contre un peu le fichier, et demande plus de ressources machine pour la décompression. Notre application est donc similaire : on visualisera rapidement le maillage simplifié avec des détails de normales grossiers, les raffinements successifs faisant apparaître progressivement les détails fins à la surface de m. La visualisation est ainsi permise dès la réception des premières données, sans forcer l'attente du téléchargement total de la texture de détails ou du maillage original complet.

Nous décrivons dans ce chapitre l'architecture choisie pour la visualisation réseau de maillages avec *APO*, ainsi que les choix techniques d'implémentation du visualiseur. Nous commençons dans un premier temps par présenter les méthodes existantes pour effectuer de la visualisation répartie de maillages 3D.

6.2 Visualisation de maillages sur environnement réseau

6.2.1 Transmission de LOD géométrique

Pour effectuer une visualisation rapide sur un environnement réseau, la solution classique est d'adopter une méthode multi-résolution pour le maillage : la version basse résolution, étant la moins lourde, est transmise initialement pour commencer l'interaction. Chaque fois qu'une version de plus haute résolution est disponible, on remplace le modèle par la dernière version reçue. Pour générer une hiérarchie de maillages de différentes résolutions, les algorithmes de simplification peuvent être utilisés. La méthode des *Simplification envelopes* [CVM+96] permet de générer une série de simplification d'un même maillage, chaque version étant située à l'intérieur d'une enveloppe construite autour de M. Plus l'enveloppe est large et donc plus la tolérance sur la modification de la position des sommets est grande, plus la simplification est forte. Lors de la visualisation, les différentes versions sont transmises successivement. Cette méthode souffre de plusieurs défauts majeurs :

- Le serveur est dans l'obligation de stocker et gérer un ensemble de modèles pour la transmission d'un seul objet, surchargeant ainsi les périphériques de stockage.
- La transition vers un modèle d'une résolution supérieure ne se fait pas de manière douce, mais au contraire de manière abrupte. Deux modèles de résolutions différentes ne présentent pas de cohérence spatiale forte, leurs sommets n'ont pas d'équivalent direct sur les maillages respectifs. Lors de la transition, le modèle de plus basse résolution est remplacé par le modèle suivant, qui présente de forts changements. L'artefact visuel du changement brutal est parfois appelé *popping* en anglais, et nuit à la qualité du rendu : l'impression rendue est que le modèle basse résolution est remplacé plutôt que raffiné.
- L'utilisation de la bande passante est loin d'être optimale : avant de pouvoir visualiser le modèle haute résolution, tous les objets de résolutions inférieures doivent être intégralement transmis. Les maillages de différentes résolutions sont indépendants et la transmission des géométries des premiers modèles ne peut pas servir à réduire le volume de données à transmettre pour les maillages de plus haute résolution.

6.2.1.1 Maillages progressifs et compressés

L'utilisation d'une représentation hiérarchique des maillages, comme par exemple les maillages progressifs [Hop96] permet d'éviter la plupart de ces écueils. Un maillage progressif est constitué d'un maillage simplifié, et d'un ensemble d'opérations permettant de le raffiner. Ces opérations sont des suites de *vertex split*[19] qui permettent de remplacer un sommet par deux nouveaux, en faisant apparaître de nouveaux triangles. La suite de ces opérations de séparation est l'inverse de la séquence d'opérations d'*edge collapse*[20]

[19] séparation de sommets
[20] suppression d'arêtes

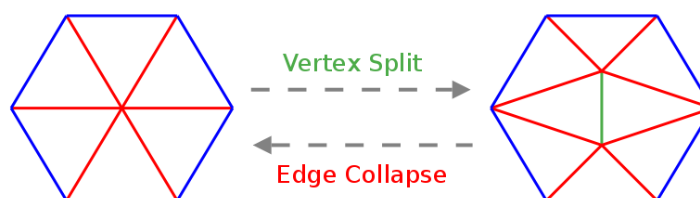


Fig. 6.1 – Opérations d'*edge collapse* et *vertex split* utilisées dans la structure des maillages progressifs.

qui a permis de générer le maillage m à partir de M. Le principe de ces opérations est illustré sur la figure 6.1.

Pour une transmission des données sur le réseau, le maillage simplifié est tout d'abord transmis rapidement, il pèse en général à peine 1% du poids de la structure totale, ensuite les opérations encodées de séparation de sommets sont successivement transmises, ajoutant les sommets un par un, raffinant ainsi de manière douce et progressive le maillage, sans effet de *popping* notable. La position des nouveaux sommets peut de plus être interpolée avec une fonction de temps pour adoucir d'autant plus le raffinement du maillage. Dans l'article [Hop97] la technique est améliorée de façon à ne transmettre sur le réseau que les opérations de raffinement nécessaires. Les sommets situés à l'extérieur du cône de vision, ou encore qui sont au dos des objets visualisés ne sont pas transmis. De plus, une mesure d'erreur est effectuée lors de la projection à l'écran, si le niveau de détail est suffisant par rapport à un seuil d'erreur, les raffinements sont bloqués. Seules sont transmises les informations concernant les raffinements réellement nécessaires, l'utilisation de la bande passante est ainsi optimisée.

Avec cette représentation des maillages progressifs, les données transmises sont désormais interdépendantes, la position des nouveaux sommets étant déduites de ceux déjà présents. La qualité du réseau est donc cruciale lors de la transmission, car la perte de données peut impliquer l'impossibilité de décoder l'ensemble du maillage. Les auteurs de [COM+07] présentent une analyse et un modèle de transmission des paquets, groupant des ensembles d'opérations de *vertex split*, ces paquets étant plus ou moins dépendant les uns des autres, ajustant la priorité entre rapidité de transmission, au risque de perdre des données pour le décodage, et qualité de transfert, avec un contrôle d'erreur.

Notons également que les maillages progressifs ont été étendus de façon à les rendre plus compacts, en groupant entre elles plusieurs opérations de séparation de sommets. Ces travaux [PR00, TGHL98] permettent d'encoder de manière plus compacte les séries d'opérations. D'une manière générale, les travaux traitant de la compression de maillages, comme ceux présentés dans [AD01, BPZ99, COLR99, KSS00] peuvent être utilisés dans le cadre de transmission réseau dans la mesure où ils proposent un décodage progressif des géométries compressées.

Fig. 6.2 – Transmission progressive de Q-Splats. Les images sont prises à respectivement 1, 10 et 60s sur un réseau de 384 kb/s.

6.2.1.2 Structure hiérarchique

Plutôt qu'une hiérarchie de modèles, ou une représentation progressive du maillage, les auteurs de [RL01] propose l'utilisation de leur structure hiérarchique arborescente de sphères englobantes, présentée dans [RL00], structure qui contient les différents niveaux de détails du modèle. Les auteurs utilisent non pas un maillage triangulé, mais un nuage de points, effectuant un rendu à base de splats. La structure contient pour chaque nœud de l'arbre une liste des nœuds fils si ils existent, mais aussi des informations sur la sphère englobante du nœud. Lors du rendu, un parcours de l'arbre est effectué, et si un nœud n'a pas encore de fils disponibles, un splat dont le rayon est celui de la sphère englobante est dessiné.

Cette structure hiérarchique permet, à l'instar des maillages progressifs, de faire une transmission adaptative des données, en ne transférant que les nœuds qui sont visibles à l'écran. Cepdendant cette méthode présente le désavantage de transmettre une plus grande quantité de données que les méthodes basées sur les maillages progressifs. De plus, les données initialement transmises, c'est à dire les nœuds les moins profonds, qui sont rendus par des splats de grande surface, portent moins de détails sur la géométrie et la topologie de l'objet qu'un maillage polygonal simplifié. Ce désavantage est à nuancer car étant donné l'augmentation croissante des taux de transferts disponibles, les splats larges ne sont affichés que l'espace de quelques secondes pour être remplacés rapidement par un rendu par points assez précis (voir figure 6.2).

Nous tenions à présenter cette méthode, car on peut voir dans la structure arborescente hiérarchique utilisée une analogie avec la transmission de l'APO et donc d'un octree, mais qui n'encode que l'apparance d'un maillage, et pas sa géométrie comme dans [RL01]. La structure hiérarchique de l'octree nous permettra également de contrôler les données transmises en fonction du point de vue.

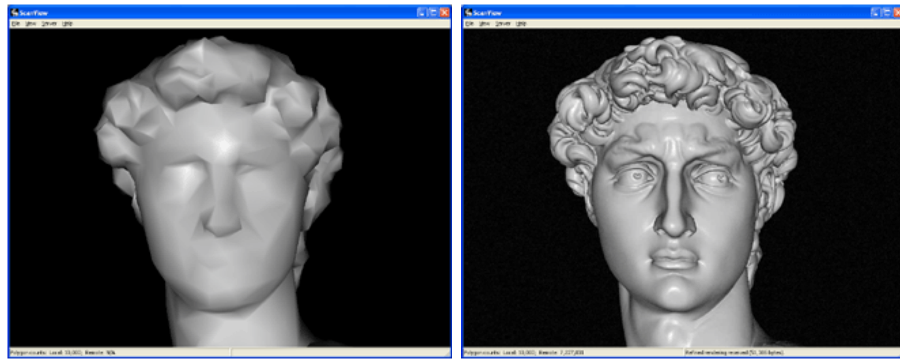


Fig. 6.3 – A gauche le maillage manipulé par le client, à droite l'image rendue calculée par le serveur et transmise au client.

6.2.2 Rendu à distance

Les méthodes que nous avons présentés jusqu'à présent ont toutes un point commun : les données stockées sur un serveur distant sont transmises à un client qui effectue le rendu en local. Ceci implique que pour rendre le maillage original, le modèle complet doit être transféré et mis à disposition du client. Dans le cas où, pour des raisons de droits ou de confidentialité, ce transfert ne doit pas être autorisé, les clients n'ont pas moyen de visualiser les maillages. Les auteurs de [KTL+04] et [KL05] présentent une méthode, dans le cadre du projet *Digital Michel Angelo*[21], qui permet aux clients de visualiser les modèles numérisés sans avoir à disposition le maillage original.

Le principe est de fournir aux clients un visualiseur qui ne manipule que des maillages simplifiés. Dès que l'utilisateur client arrête de manipuler l'objet, les coordonnées du point de vue sont envoyées au serveur, qui calcule le rendu et envoie comme réponse au client l'image du rendu avec le maillage haute résolution (voir figure 6.3). Ces images étant compressées, le temps de latence entre la fin de la manipulation et l'affichage de l'image résultat est court. Du côté serveur, plusieurs versions des maillages sont disponibles, représentant plusieurs niveaux de détails de l'objet, le rendu est calculé sur celui qui convient le mieux en fonction de la distance entre le point de vue et l'objet.

Pour éviter toute reconstruction des maillages à partir des images rendues, et ainsi renforcer d'autant plus la sécurité des données, les rendus sont légèrement altérés, en modifiant les conditions d'éclairage, ou encore en modifiant légèrement les positions des pixels. Ces altérations ne nuisent pas à l'interprétation de l'utilisateur, mais empêche seulement une reconstruction des objets par des algorithmes *malveillants*.

[21] http ://graphics.stanford.edu/projects/mich/

6.2.3 Transmission d'apparence

Nous avons décrits plusieurs méthodes de transmission des maillages sur le réseau, dans le cadre d'une visualisation sur un environnement répartie. Ces méthodes reposent sur une compression ou un encodage compact de la géométrie pour réduire au maximum le taux d'utilisation de la bande passante. Les auteurs de [HBR+07] présentent une méthode originale, basée elle sur une quantification de l'apparence des objets (couleurs et normales), qui permet de réduire le volume de données transmises, tout en minimisant le travail du CPU du côté serveur.

Pour éviter l'utilisation intensive du CPU, la quantification des données étant effectuée à la volée, cette opération est laissée à la charge du GPU. Deux tables (une pour les couleurs, une pour les normales) contenant les normales et couleurs quantifiées sont précalculées et sauvegardées. Les valeurs réelles des données d'apparence sont utilisées pour retrouver l'indice correspondants dans les tables, les indices sont transmis sur le réseau. Les tables sont stockées sous formes de textures, ce qui permet l'exécution de la quantification sur le GPU. Les auteurs appliquent leurs méthodes aux modèles de nuages de points, où chaque point est représenté par 9 flottants (3 flottants pour chaque donnée : position, couleur, normale). Si les positions ne sont pas quantifiées, les normales et couleurs sont encodées sur mot de 32 bits, réduisant ainsi le volume de données à transférer de 36 octets[22] par points à 16 octets[23].

Le client recevant les données doit déquantifier les normales et couleurs avant d'effectuer le rendu final. Cette technique permet de diminuer l'utilisation de la bande passante et également celle du CPU, bien qu'elle nécessite un GPU capable d'effectuer les traitements requis sur le serveur. De plus, cette méthode implique que la géométrie complète est transférée au client, qui doit donc être capable d'éventuellement manipuler des géométries complexes.

6.3 Architecture APOWeb3DViewer

En utilisant les *APO*, il est possible d'effectuer une transmission progressive des détails d'apparence d'un modèle alors que le maillage simplifié est affiché sur le client. Cette possibilité est offerte par la structuration en largeur d'abord de l'octree, qui fait que les niveaux de détails les plus grossiers sont rangés dans les premiers pixels de la texture. Nous proposons donc un visualiseur à distance de maillages qui présentent les caractéristiques suivantes :

- Seule une version simplifiée du maillage est transmise au client. Ce dernier étant léger et donc vite transmis, une visualisation est très rapidement disponible sur le

[22] 1 flottant (4 octets) pour chacune des données : position, normale, couleur (3 valeurs pour chaque attribut)

[23] 1 flottant (4 octets) encodant les normales et couleurs quantifiées, plus un flottant pour chacune des composantes des sommets

client, qui peut manipuler immédiatement m. Comme dans les travaux de [KTL+04, KL05], notre méthode permet donc d'assurer la confidentialité et la sécurité des données originales.

- La texture *APO* est téléchargée par le client, et dès qu'un nombre suffisant de niveaux de détails est disponible, typiquement dès que 7 niveaux de détails sont totalement téléchargées (les normales moyennées des niveaux inférieurs sont trop peu précises), le placage de l'*APO* peut être appliqué. La texture est ensuite régulièrement mise à jour quand des nouveaux détails sont disponibles.
- Le téléchargement peut être piloté par le niveau de détail nécessité par la résolution de l'écran. Si les détails téléchargés sont suffisants pour un affichage de qualité, la transmission est momentanément suspendue. Elle sera reprise en cas de besoin.
- Le transmission du maillage simplifié et de l'*APO* est moins lourde que la transmission d'un maillage haute résolution complet. Si l'utilisation de techniques de compression de maillages peut également réduire le volume de données transitant sur le réseau, l'utilisation de l'*APO* permettra toujours la manipulation interactive des modèles du côté client, là où la manipulation d'un maillage haute résolution reste problématique.

Notre visualiseur client est donc composé de deux modules : le module de rendu, et le module de téléchargement et mise à jour de la texture *APO*. Contrairement aux méthodes présentés dans la section précédentes, qui nécessitent toutes un serveur effectuant des traitements lors de la visualisation répartie, soit pour effectuer le rendu comme dans [KTL+04], soit pour quantifier les données, ou encore marquer les données en fonction du point de vue, notre architecture client/serveur ne nécessite aucun travail côté serveur, hormis celui de stocker les données. Le client calcule lui-même le niveau de détail nécessaire au rendu et télécharge les données suivant ses besoins.

Cette architecture légère permet de créer le client comme une simple applet web, les données étant stockées sur un serveur HTTP. Il n'y a dans ce cas pas d'installation logicielle nécessaire. Comparativement à la création d'un logiciel dédié, notre visualiseur ne nécessite qu'un navigateur web pour sa mise en œuvre. De plus, l'utilisation du protocole HTTP nous affranchit de la gestion du téléchargement. La figure 6.4 illustre l'architecture de notre visualiseur à distance basée sur les *APO*.

Dans le cadre d'une implémentation du client en tant qu'applet Java pour navigateur web (voir section 6.5), lors de l'affichage de la page hébergeant le visualiseur, le premier téléchargement à être lancé est celui de l'archive *jar* contenant le viewer. Cette archive pèse seulement 300 Ko, et son temps de téléchargement est donc négligeable par rapport à celui du téléchargement de la texture *APO*. Une fois que l'applet est téléchargée, son exécution démarre. Dans un premier temps, le maillage simplifié est téléchargé. Avant de démarrer le téléchargement de la texture de détails *APO* un fichier texte contenant des informations sur celle-ci est lui aussi téléchargé. Il contient les dimensions de la texture,

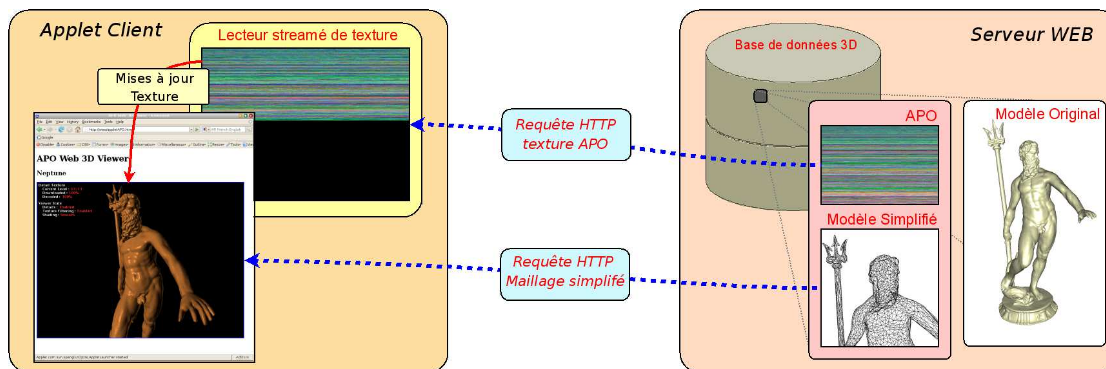


FIG. 6.4 – Architecture client/serveur du visualiseur à distance basée sur les *APO*. Le serveur est simplement utilisé comme serveur de stockage des données. Le client est composé de deux modules : le module de rendu et le module de téléchargement progressif de la texture *APO*.

ainsi que le nombre de texels qui composent chaque niveau de détail de l'*APO*. Cette information est nécessaire pour déterminer quelle est la profondeur maximale disponible lors du parcours de l'*APO* pour le rendu.

Une fois que le maillage simplifié et le fichier d'information sont téléchargés, le visualiseur affiche le maillage et le module de téléchargement de texture est lancé. Lors du lancement de la visualisation du client, une texture vide de la dimension de la texture finale est créée. Comme les détails ne sont pas encore disponibles, seul le maillage simplifié est affiché. La texture *APO* n'est pas utilisée dès que les premiers détails sont disponibles, car les premiers niveaux, c'est à dire les nœuds de faible profondeur, n'apportent pas plus de détails que le maillage simplifié. Idéalement, l'utilisation de l'*APO* ne devrait être activée que lorsque les nœuds disponibles sont ceux dont la largeur est plus petite que les dimensions des triangles. On aurait ainsi plusieurs échantillons par triangle, apportant plus de détail que la normale interpolée à partir des sommets de chaque triangle. En pratique, sur les maillages simplifiés que nous avons testé, il était intéressant d'activer l'utilisation de l'*APO* dès que les 7 premièrs niveaux de détails étaient entièrement téléchargées, représentant un faible pourcentage du poids total de la texture à transférer, ce qui permet d'obtenir rapidement les bénéfices de l'*APO* (voir figure 6.5).

Les sections suivantes décrivent le téléchargement progressif de l'*APO* ainsi que les modifications à apporter au *fragment shader* pour prendre en compte le fait que la texture n'est pas complète lors du rendu.

FIG. 6.5 – A gauche le maillage simplifié est d'abord affiché. Ensuite de gauche à droite les niveaux complétés sont respectivement de 7, 8 et 10. On remarque que le niveau de détail 7 contient autant de détails que le maillage simplifié. Le maillage simplifié est composé de 20 mille polygones.

6.3.1 Mise à jour de la texture

Le processus de téléchargement de la texture reçoit continuellement les texels constituant la texture *APO*. Pour éviter une mise à jour incessante de la texture sur la carte graphique, et donc un ralentissement des performances impliqué par des transferts trop nombreux entre la mémoire centrale et la mémoire GPU, ces texels sont stockés dans un tableau intermédiaire créé sur la mémoire centrale. Pour contourner la limite des 64Mo de mémoire alloué par défaut aux applets Java, notre implémentation alloue un tableau de taille variable qui recevra des portions de l'*APO* téléchargée. Une fois plein, ce buffer sera concaténé à la texture sauvegardée sur la carte graphique par la commande glTexSubImage2D et il pourra recevoir les texels suivants après avoir été vidé. Cette implémentation permet la manipulation de textures *APO* volumineuses sans toucher à la limite des 64Mo.

Il reste donc à mettre en place une politique de mise à jour de la texture sur le GPU. Nous avons testé deux politiques, chacune présentant des avantages et inconvénients.

- **Mise à jour par niveaux :** Une mise à jour de la texture, c'est à dire un appel à la fonction glTexSubImage2D pour copier les texels qui viennent d'être téléchargés dans la texture sur le GPU, est effectuée chaque fois qu'un nouveau niveau de l'octree est entièrement disponible. Cette politique limite le nombre de mise à jour et donc les échanges entre la mémoire centrale et la mémoire graphique. Cependant, la dimension des niveaux n'est pas régulière et donc leurs temps de téléchargement respectifs non plus. Par exemple, les 7 premiers niveaux de la texture utilisée pour la figure 6.6 occupent 200 Ko, les tailles des niveaux suivants étant dans l'ordre

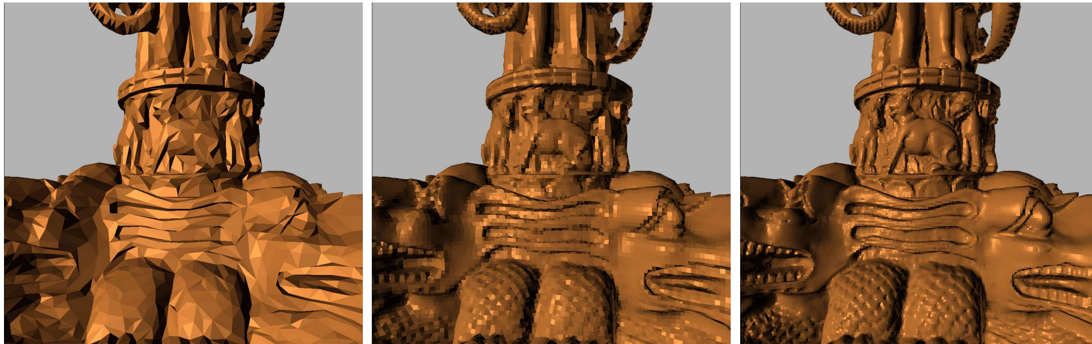


Fig. 6.6 – A gauche, le rendu après le téléchargement de 200 Ko de données, taille occupée par le maillage simplifié. Au centre, l'utilisation de l'*APO* est possible car les niveaux de détails téléchargés (ici 7) offrent une plus grande richesse que le maillage simplifié. Ces 7 niveaux nécessitent 200 Ko de téléchargement. A droite, le rendu après un téléchargement supplémentaire de 3 Mo de données (couches 8 et 9).

(couches 8 à 13) : 700 Ko, 2.4 Mo, 6.5 Mo, 11.3 Mo, 12.5 Mo et 8.4 Mo. L'utilisateur pourra observer très rapidement le maillage avec les niveaux de détails faibles de l'*APO*, l'attente pour chaque raffinement sera elle assez longue.

- **Mise à jour réguliere de la texture :** La mise à jour de la texture se fait de manière régulière, de façon à faire apparaitre plus progressivement les détails à la surface de m. L'utilisateur voit ainsi les détails qui sont téléchargés même si la couche en cours de téléchargement n'est pas complète. Pour effectuer la mise à jour, un timer est utilisé, la texture étant mise à jour à intervalles réguliers[24]. On peut aussi limiter la taille du buffer à un nombre fixe de lignes de la texture. Ainsi, dès que n lignes sont complètes, elles sont ajoutées à la texture.

Après chaque mise à jour de la texture, indépendamment de la politique choisie, le buffer de stockage des texels est vidé, de façon à conserver une occupation mémoire constante de l'applet tout au long de l'exécution. Les captures de la figure 6.6 montre l'évolution du rendu en fonction du volume de données téléchargé. Le maillage original est composé de 10 millions de triangles, le maillage simplifié qui est téléchargé par le visualiseur de 20 mille triangles. La texture *APO* occupe quant à elle 42 Mo.

Dans la section suivante, nous décrivons les légères adaptations qu'il faut apporter au *fragment shader* pour prendre en compte le fait que la texture n'est pas complète lors du rendu.

[24]Dans notre implémentation, nous avons choisi de mettre à jour la texture tous les quarts de seconde.

6.3.2 Fragment shader adapté

Lors du rendu des maillages simplifiés avec leurs *APO* dans le visualiseur à distance, la totalité des détails de la texture n'est disponible qu'après le téléchargement du dernier niveau de détail. Avant, la texture est donc logiquement incomplète et le parcours de l'octree doit être interrompu précocement. Dans le cadre d'une mise à jour de la texture à chaque niveau complet reçu, cet arrêt est facile à déterminer : la parcours de l'*APO* s'arrête dès que la profondeur de la dernière couche totalement téléchargée a été atteinte. La valeur numérique de ce niveau est transmise au programme sous forme de variable uniforme.

Dans le cas où l'on applique une mise à jour régulière de la texture, le dernier niveau est incomplet, c'est à dire composée des texels noirs de la texture vide initiale, mais aussi des texels déjà téléchargés au début du niveau. Pour exploiter ces détails, il faut donc parcourir l'*APO* jusqu'à la profondeur de la dernière couche entièrement téléchargée, puis descendre d'un niveau supplémentaire pour vérifier si le fils du nœud courant a été téléchargé, et l'appliquer au maillage le cas échéant. Un test qui décidera de la normale à appliquer, c'est à dire soit celle présente au niveau de la dernière couche totalement téléchargée ou celle du niveau suivant si elle est déjà disponible, est effectué. Lors du parcours les deux nœuds des deux derniers niveaux parcourus doivent donc être sauvegardés. Pour éviter l'instruction conditionnelle nécessaire au choix de la normale à appliquer, nous avons mis en place une opération d'interpolation, par le biais de la fonction GLSL mix. Si le dernier texel accédé est noir, c'est qu'il n'a pas encore été téléchargé, un coefficient de valeur 0 est donc calculé de façon à ce que cette normale ne soit pas appliquée. Au contraire, si le texel contient des données non nulles, le coefficient d'interpolation aura 1 pour valeur, ce qui permet d'affecter la normale de la couche la plus profonde au fragment. Cette opération est réalisée par le pseudo-code suivant :

```
// calcule le coefficient
float coeff;
coeff = sign(dernierNoeud.r + dernierNoeud.g + dernierNoeud.b);

// lit la normale
normale = mix(avantDernierNoeud, dernierNoeud, coeff);
```

L'opérateur sign retourne 1 si au moins une des composantes de dernierNoeud est non nulle, c'est à dire que le ce dernier nœud a été téléchargé, il retourne 0 sinon.

6.4 Téléchargement adaptatif

La structure hiérarchique peut également être utile pour contrôler le téléchargement, c'est à dire le stopper quand les détails ne sont pas nécessaires à l'écran. Nous appliquons au téléchargement de la texture le même principe que pour le rendu adaptatif (filtrage sous minification, 4.3.2 page 97) : si les nœuds de détail téléchargés se projettent sur moins d'un pixel à l'écran, nous pouvons suspendre momentanément le téléchargement des détails plus fins.

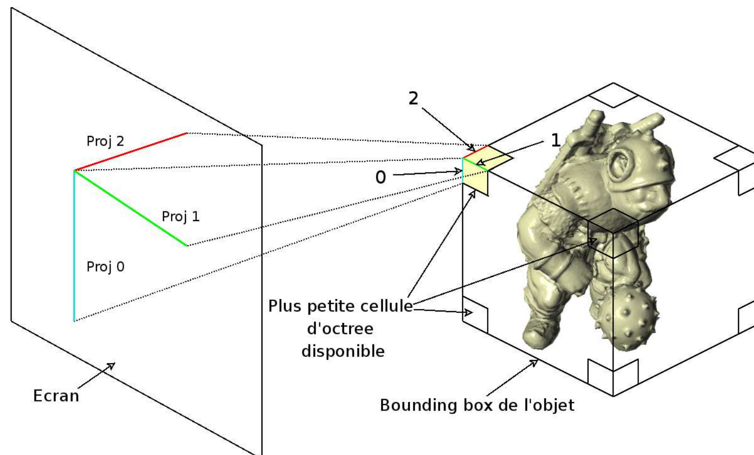


FIG. 6.7 – Pour déterminer si le téléchargement doit être interrompu, le nœud le plus fin déjà téléchargé est projeté sur l'écran. Si toutes les dimensions sont plus petites que un pixel, alors il n'y a plus besoin de détail supplémentaire et le transfert est stoppé.

Pour le rendu adaptatif, la profondeur idéale était calculée au début du parcours de l'octree, en tenant compte de l'éloignement du plan écran de chaque fragment (voir section 4.3.2 page 97). Ce calcul se basait sur la distance séparant les positions d'origines sur l'objet de deux fragments voisins. Dans le module de téléchargement de texture, qui est exécuté sur le CPU, nous n'avons pas accès aux fragments qui vont être projetés, ce travail étant dévolu au processeur graphique. Pour déterminer si les détails téléchargés sont suffisants, nous adoptons donc une méthode qui consiste à calculer la dimension en pixels de la projection sur l'écran d'un nœud de la couche la plus profonde (c'est à dire celle qui est en cours de téléchargement). Comme cette projection est dépendante de la position du nœud dans la scène, il faut calculer cette projection pour le cas où le nœud se trouve au plus proche de l'écran, là où le besoin de détail est le plus grand. Nous faisons là une approximation, en supposant que la position la plus proche de l'écran sera située sur un des 8 angles de la boite englobante de l'objet. Le schéma de la figure 6.7 illustre ce principe.

La fonction openGL gluProject, qui prend en paramètre une matrice de projection et les coordonnées d'un point, permet de calculer la position à l'écran de la projection de ce point, et donc nous permet de calculer la projection à l'écran d'un nœud. Si aucune des dimensions du nœud projeté n'est plus grande qu'un pixel de l'image, alors le téléchargement peut-être interrompu, car les détails supplémentaires ne seront pas visibles, ils se projetteront sur moins d'un pixel de l'écran. Dans le cas contraire, le téléchargement des données de la texture *APO* peut continuer. Cette vérification se fait à chaque frame du rendu, pour permettre la reprise du téléchargement si les détails

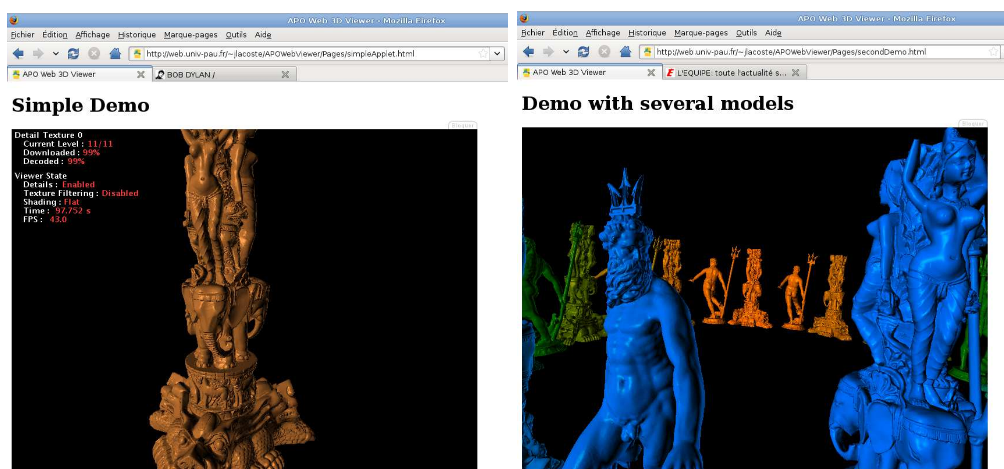


Fig. 6.8 – Deux exemples d'applet web de visualisation de maillages avec textures *APO* intégrées dans une page web standard.

sont de nouveau nécessaire. Ce calcul est rapide et n'affecte pas la vitesse d'affichage du visualiseur.

6.5 Implémentation JOGL et résultats

Le visualiseur a été implémenté et testé sous forme d'applet web Java, la partie graphique étant assurée par l'API JOGL. Cette implémentation permet d'intégrer aisément le visualiseur dans un navigateur (figure 6.8). Pour le transfert des données, nous n'avons pas enregistré du côté server les textures sous leur format image PNG, mais sous forme d'un fichier contenant le tableau 1D sous forme binaire. Ce fichier est stocké compressé au format GZIP, de façon à limiter le volume de données qui vont transiter sur le réseau. Du côté client, la décompression est effectuée à la volée grâce aux fonctions de lecture de flux compressé fournies par le langage Java, les objets GZIPInputStream. Cette décompression est efficace et non pénalisante en termes de performances lors de la lecture de la texture. Le fichier contenant le maillage est lui aussi stocké compressé au format GZIP.

Le programme principal de l'applet télécharge le maillage simplifié et le fichier d'information, dont les url sont passées en paramètres dans le code HTML de la page contenant le visualiseur. Après ce téléchargement, la visualisation commence tandis que le thread de téléchargement est lancé. Les *APO* détaillées pouvant peser plusieurs dizaines de Mo (jusqu'à 40Mo compressé pour les modèles présentés dans ce manuscrit), le temps de téléchargement total peut être long sur des réseaux ayant de faibles débits, mais les détails portés par les bas niveaux de l'octree peuvent être rapidement transmis. Par exemple, sur un réseau ADSL basique (64 Ko/s), la transmission des 9 premiers niveaux, qui occupent en moyenne 3Mo, prend moins d'une minute. Le téléchargement total prendra lui

10 minutes. Sur des réseaux plus performants (réseau local filaire, Wi-Fi, ADSL+ etc), les temps de transferts seront moindres, et le transfert des 10 premières couches peut être effectué en moins d'une minute. Notre visualiseur permet donc de manipuler rapidement des maillages, l'apparence étant elle peu à peu raffinée à la surface de l'objet. De plus, cette apparence d'abord grossière est rapidement enrichie par les niveaux de détails de l'*APO*.

Quatrième partie

Conclusion

Chapitre 7

Conclusion

Sommaire

7.1 Synthèse 145
7.2 Perspectives 149

7.1 Synthèse

Nous avons présenté dans ce manuscrit une chaine complète d'outils permettant la création d'octree-textures d'apparence pour maillages simplifiés, allant de la construction des textures par l'échantillonnage du champ de normales du maillage original, jusqu'à leur utilisation dans des applications 3D interactives. Nous avons également présenté une application originale de transmission des textures dans le cadre d'une application de visualisation de maillages à distance avec téléchargement progressif des détails. Cette application peut être intéressante pour les sites de dépôt de maillages 3D par exemple.

Pour la création des textures, nous avons détaillé les trois grandes étapes qui permettent d'obtenir les octree-textures d'apparence :

- **Subdivision de l'octree :** C'est dans cette première étape que la structure de la texture est mise en place. L'octree, support de nos textures d'apparence, est positionné autour du maillage simplifié, puis récursivement subdivisé. La politique de subdivision des nœuds de l'octree permet d'adapter la résolution aux détails sous-jacents du modèle haute résolution. En effet, si les triangles du maillage haute résolution positionnés dans un nœud présentent une forte variation dans leurs normales, ce dernier est subdivisé. Si au contraire, les normales sont identiques, le nœud ne sera pas subdivisé. Pour mesurer les variations à l'intérieur d'un nœud, nous avons mis en place un calcul de métrique basé sur la métrique $L^{2,1}$ présentée dans [CSAD04]. Dans ce calcul de métrique, qui est une somme des différences des normales par rapport à la normale moyenne, nous ôtons la pondération par l'aire

présente dans la version originale, de façon à ne pas minimiser la contribution de petits triangles dans le relief et l'apparence de la surface. Grâce à cette métrique, les nœuds les plus fins sont uniquement placés sur les zones qui présentent des fortes variations dans les normales, c'est à dire les zones ayant le plus de détails haute fréquence. Outre la métrique, une profondeur maximale est également spécifiée pour stopper les subdivisions.

- **Récupération des normales du maillage original :** Une fois que la structure est fixée par les subdivisions, les feuilles de l'octree se voient attribuer une normale représentative du maillage original. Cette assignation se fait avec une opération de lancer de rayons, depuis la feuille vers le maillage original. À l'intersection entre le maillage et le rayon, la normale est récupérée et associée au nœud. Le lancer de rayons est accéléré par la structuration spatiale que représente l'octree, il permet en effet de localiser rapidement les triangles du maillage original qui sont le plus susceptibles d'intersecter le rayon. Après la phase de lancer de rayons, les nœuds internes de l'octree sont eux aussi associés à une normale représentative, qui n'est pas échantillonnée sur le maillage original, mais qui est la moyenne des normales de leurs fils. Ce moyennage, calculé avec un processus remontant des feuilles vers la racine, peut être vu comme les différents niveaux hiérarchiques de la texture, à l'instar des textures MIP-map dans le cadre des textures 2D traditionnelles.

- **Encodage 2D de l'octree :** La dernière étape de construction des octree-textures d'apparence est l'encodage sous forme d'images 2D de la structure d'octree. Les octrees n'étant pas supportés sur les processeurs graphiques, cet encodage est nécessaire pour rendre disponibles au GPU les données de normales lors du rendu. Nous avons proposé un encodage dans lequel les nœuds sont dans un premier temps rangés en largeur d'abord dans un tableau 1D, ce tableau 1D étant ensuite découpé de façon à former les lignes de l'image finale. Le tri en largeur d'abord permet de ne stocker qu'un offset du nœud parent vers ses fils, ceux-ci sont contigus dans le tableau. Ce tri, combiné au fait que nous ne sauvegardons pas les feuilles vides de l'octree, permet de rendre l'encodage de l'octree-texture très compact. Nous estimons que notre encodage est en moyenne 30% plus compact que celui présenté dans [LHN05]. Les textures sont stockés sur disque au format PNG car il est nécessaire de ne pas altérer le contenu des texels encodant les nœuds.

Nous avons appelé nos textures d'apparence *APO*, de l'acronyme anglais d'*Appearance Preserving Octree-textures*. Nous avons détaillé chacune des étapes de création ainsi que leurs problématiques respectives dans le manuscrit. Durant les travaux, nous avons toujours eu le souci de proposer une construction qui sauvegarde au mieux l'apparence des maillages, en mettant par exemple en place le pilotage des subdivisions par les détails du

maillage original, puis la récupération des normales par le lancer de rayons. Mais nous avons également toujours eu comme souci de proposer une création des *APO* efficace, calculée en un temps minime. Les temps de création que nous avons obtenus sont dépendants de la taille des maillages donnés en entrée, ainsi que de la précision souhaitée, mais sont de l'ordre de quelques minutes.

Le principal avantage des *APO* est que la structuration d'octree évite la paramétrisation des maillages. Cette paramétrisation est très ardue à obtenir pour des maillages très détaillés, faisant souvent appel à l'intervention d'un utilisateur. Au contraire, la création des *APO* est totalement automatique et indépendante de la méthode de simplification utilisée.

Dans la deuxième partie du manuscrit, nous avons montré que les *APO* pouvaient être utilisées dans des applications 3D temps réel. En effet, le placage des *APO*, bien que coûteux sur des maillages simplifiés (un parcours d'octree est nécessaire pour chaque fragment affiché) atteint des vitesses d'affichage supérieures aux vitesses nécessaires pour obtenir des applications interactives. Cependant, lorque l'on est en gros plan le nombre de fragments pour lesquels il est nécessaire de parcourir l'octree augmente, limitant donc les performances d'affichage. Le même constat peut être fait lorsque le filtrage bilinéaire est activé. Nous avons donc proposé des optimisations qui évitent une partie du parcours de l'arbre, ou encore une approximation du filtrage qui permet de limiter le nombre de parcours. Dans le cas extrême, une conversion des *APO* en atlas de texture 2D classique est possible, cependant les bénéfices de la structure hiérarchique (échantillonnage adaptatif, niveaux MIP-map internes) sont alors perdus.

Un exemple d'application dans laquelle les *APO* ont été utilisées avec succès est présentée dans les travaux de Pacanowski et al. [PRL+08]. Dans ces travaux, les maillages simplifiés enrichis de textures *APO* sont utilisés dans des scènes dans lesquelles l'illumination est calculée de manière indirecte. L'éclairage indirect étant peu sensible aux variations hautes fréquences de la géométrie, il est précalculé sur des maillages simplifiés. Au moment du rendu, les *APO* sont utilisées pour restituer les détails de la surface des objets. Ainsi, le calcul de l'éclairage et le rendu sont effectués avec des maillages basses résolutions. Il est alors possible d'obtenir des vitesses d'affichage atteignant plus de 60 images par seconde, ce qui serait inenvisageable si les modèles originaux étaient utilisés. L'image de la figure 7.1 est tirée d'une de ces scènes.

Enfin notre méthode de création de texture de détails nous a permis de proposer une application originale dans le cadre d'une application de visualisation à distance de maillages 3D. Notre approche est de transmettre intégralement un maillage simplifié léger, puis de télécharger progressivement la texture de détails tout en visualisant le modèle. La structure hiérarchique ainsi que le tri en largeur d'abord des nœuds permet de raffiner progressivement les détails de normales à la surface des objets, puisque les premiers texels téléchargés correspondent aux faibles niveaux de détails des *APO*,

Fig. 7.1 – Scène composée de trois statues issues du bassin méditerranéen. La géométrie complète est composée de 13 millions de polygones, la géométrie simplifiée de 280 mille polygones. Les détails des modèles sont restitués par l'utilisation d'une *APO* pour chacun d'entre eux, la source lumineuse est orientée vers le plafond, tout l'éclairage des objets est donc indirect. Les vitesses d'affichage obtenues sont de 30 im/s pour l'utilisation des *APO*, 60 si l'on utilise les atlas de texture.

les texels suivants correspondant eux aux détails situés dans des feuilles plus profondes. L'utilisateur peut visualiser les maillages avec leurs *APO* sans attendre le téléchargement complet des textures. Notre implémentation sous forme d'applet web est légère tant au niveau client qu'au niveau serveur, ou un simple serveur web est nécessaire pour stocker les données. De plus, seuls les maillages simplifiés sont mis à disposition des clients, ce qui est un avantage lorsque la confidentialité des données est requise.

Les différents travaux et résultats ont été mis en évidence dans l'article [LBJS07] pour ce qui concerne la création et l'utilisation des *APO*, dans l'article [LPJ08] pour ce qui concerne l'application de visualisation distribuée.

7.2 Perspectives

Il existe plusieurs pistes pour améliorer les différents pans des travaux présentés dans ce manuscrit. Nous donnons d'abord celles qui concernent la création des *APO*, puis nous nous concentrerons plus longuement sur un des points critiques de l'utilisation de l'*APO* : la vitesse d'affichage limitée par les accès textures. De plus, une autre piste intéressante serait l'extension des *APO* au stockage de propriétés autre que les normales : couleurs, coefficients d'occlusion ambiante, de réfraction ou réflexion etc. Des stratégies de subdivision devront être adaptées à la variation de l'ensemble des propriétés à échantillonner et des algorithmes de quantification adéqats limiteront la taille nécessaire à leur stockage.

Bien que les temps de création des *APO* ne soient pas rédhibitoires, les textures pouvant être créées en quelques minutes seulement, la phase de construction peut être améliorée du point de vue de la rapidité, de la précision et de la consommation mémoire. Pour ce qui est de la mémoire utilisée pendant la création, l'utilisation d'un octree sans pointeur, trié dans un tableau 1D, pourrait réduire la quantité d'informations requise par la sauvegarde des liens des nœuds vers leurs fils. Si la gestion de la structure d'octree deviendrait plus compliquée, car il faudrait gérer les décalages des nœuds dans le tableau lors des subdivisions et donc des créations des nouveaux nœuds, la taille occupée par l'octree durant la création des *APO* serait considérablement réduite, car on éviterait le stockage de 8 pointeurs pour chacun des nœuds. De plus, un des avantages de cette solution serait la conversion plus rapide de l'octree en texture 2D finale. En effet, lors de l'encodage de l'octree en texture 2D, on passe tout d'abord par une tableau 1D intermédiaire dans lequel l'octree est trié en largeur d'abord, ce tableau étant ensuite placé dans la texture 2D. Avec un octree sans pointeur directement stocké dans un tableau 1D, cette conversion serait évitée et la transformation en texture serait plus directe. Comme nous l'avons déjà évoqué plus tôt, la mise en place d'un algorithme *out-of-core* est également une solution pour limiter l'occupation mémoire lors de la création des *APO*.

Les autres pistes pour l'amélioration de la vitesse de création des *APO* concernent la phase de lancer de rayons. Cette étape nécessite de nombreuses opérations. Nous avons déjà mis en place les calculs d'intersection les plus efficaces, mais nous n'avons testé qu'un seul parcours de l'octree pour la recherche des nœuds intersectés par le rayon. Ce parcours est un parcours descendant de la racine vers les feuilles, il donne directement les nœuds intersectés, sans nécessiter de remontée dans l'arbre. Un test d'un parcours montant de la feuille contenant l'origine du rayon pourrait permettre de vérifier si notre implémentation est la plus efficace. Nous avons l'intuition que le parcours remontant sera plus coûteux, car il nécessitera de nombreuses remontées et descentes dans l'arbre pour identifier les nœuds voisins de l'origine du rayon. Cependant, en enrichissant la structure d'octree d'informations sur le voisinage (informations nécessaires uniquement pendant la phase de création des *APO*), la recherche des nœuds intersectés par le rayon pourrait être accélérée.

Si les temps de création peuvent être améliorés, ils ne sont pas à l'heure actuelle rédhibitoires à l'utilisation des *APO*. Le point le plus critique de notre approche est le nombre d'accès texture réalisés pendant le parcours de l'octree, accès qui pénalisent les vitesses d'affichage des maillages avec textures de détails *APO*. De plus, ces accès textures ne sont pas cohérents spatialement dans la texture 2D encodant l'octree, les fils pouvant être très éloignés de leurs parents. Nous avons mis en place des optimisations permettant d'éviter les premiers accès textures, ceci en complétant les premiers niveaux de l'octree. Cette optimisation peut éviter la moitié des accès textures nécessaires au parcours de l'arbre, mais les accès finaux sont toujours incohérents.

Pour améliorer plus encore les vitesses d'affichage, il est donc nécessaire de mettre en place de nouvelles structures permettant d'améliorer la cohérence des accès textures. En poussant plus l'idée de l'optimisation que nous avons proposé, dite de *forêts d'octree*, on peut imaginer modifier la structure des *APO* dans le but de rendre les accès textures du parcours de l'arbre plus cohérent spatialement parlant. L'idée serait de ne pas avoir un, mais plusieurs octrees recouvrant le maillage. Chacun de ces octree aurait sa racine à l'intérieur d'une cellule d'une grille régulière placée autour du maillage simplifié. On encoderait ensuite chacun de ces octrees en largeur d'abord, les tableaux 1D obtenus seraient ensuite concaténés dans la texture finale (voir figure 2D 7.2).

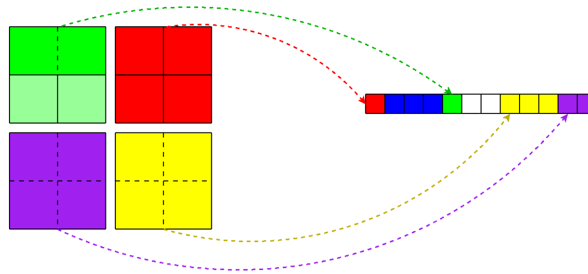


Fig. 7.2 – Chacune des cellules de la grille régulière (à gauche) est la racine d'un octree. Chacun de ces octree est trié en largeur d'abord, les tableaux respectifs sont concaténés dans un tableau global (à droite).

Avec cette solution, les différents octrees auraient tous leurs texels représentatifs contigus, au contraire de notre encodage actuel. Cet encodage nécessiterait par contre une texture 3D supplémentaire pour sauvegarder la grille régulière, chacune des cellules non vides contenant un pointeur vers l'octree encodée dans la texture de données. En choisissant une grille régulière de résolution assez haute, chacun des octree pourrait avoir une profondeur maximale relativement faible, ce qui limiterait le nombre d'accès texture nécessaires au parcours des arbres. De plus, les feuilles seraient plus proches de leurs parents respectifs dans la texture 2D, la cohérence des différents accès textures serait plus forte, ce qui améliorerait les performances en termes de vitesse d'affichage. La première grille 3D régulière d'indirection serait en grande partie vide, elle pourrait être écrite avec une table de hachage présentée dans [LH06] pour éviter l'espace perdu. Il faut toutefois

noter que cet encodage ne serait pas adapté à la transmission progressive des données sur le réseau, car les différents niveaux de détails ne serait plus consécutifs dans la texture, mais bel et bien mélangés.

Nous pensons que les textures hiérarchiques, avec les avantages qu'elles offrent sur l'échantillonnage adaptatif, la représentation multi-niveaux et l'encodage compact, seront de plus en plus utilisées dans les années à venir. C'est la raison pour laquelle il faut trouver des structures nouvelles pour offrir de nouvelles possibilités de texturage. Dans [LD07] une structure hybride est présentée. Un octree est construit autour des maillages, des portions de texture 2D sont ensuite collés sur les faces des cellules (voir figure 7.3). Bien que minimisés car appliqués sur des portions quasiment planaires, cette méthode souffre de problèmes de distorsion, et l'on ne bénéficie pas naturellement d'une représentation multi résolution, les morceaux de texture 2D ayant de plus un pas d'échantillonnage fixe.

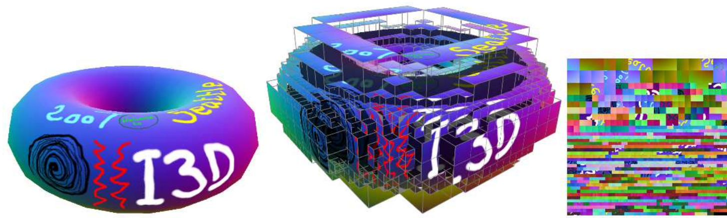


Fig. 7.3 – Tiletree : les morceaux de surface 2D sont plaqués sur les parois des cellules de l'octree.

Une piste intéressante pour pallier ces défauts serait d'utiliser la structure VS-Tree présentée par Boubekeur et al. [BHGS06]. Dans celle-ci, un octree est construit sur les premiers niveaux, ensuite, quand la surface est localement planaire à l'intérieur des nœuds, les subdivisions sont appliquées à un quadtree 2D (voir figure 7.4). Bien que

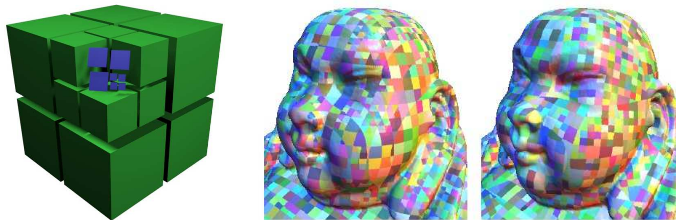


Fig. 7.4 – A gauche la représentation VS-Tree : un octree est utilisé sur les premiers niveaux, les feuilles contenant un quadtree. Le VS-Tree (à droite) présente une meilleure couverture des surfaces que l'octree (au centre).

cette structure puisse nécessiter plus de calcul pour effectuer les rotations de la grille d'octree vers le plan 2D supportant le quadtree, cette dernière représenterait plusieurs avantages : la structure 2D serait par nature moins lourde que l'octree et l'adaptation de l'orientation du quadtree permettrait une meilleure couverture de la surface que le découpage régulier de l'octree (voir figure 7.4). De plus, le filtrage bilinéaire sur le quadtree 2D ne souffrirait pas des problèmes de continuité liés aux feuilles de l'octree manquante, et serait également plus efficace en termes de performances.

La mise en place de cette *VS-Tree texture*, ou d'autres structures de support pour les textures hiérarchiques (comme le KD-Tree qui a été implémenté sur GPU par [FS05]) serait une évolution naturelle de nos travaux, et permettrait d'offrir une plus grande variété de structures hiérarchiques pour la préservation d'apparence et d'une manière plus générale pour l'habillage des modèles 3D.

Cinquième partie

Annexes

Annexe A

Glossaire et définition des termes et notations employées

A.1 Octree et subdivision

- **Maillages :**
  - M : le maillage polygonal original haute résolution.
  - m : version basse résolution du maillage.
  - $\vec{N}$: normale des sommets et triangles, mais également normales associées aux nœuds de l'octree.
- **Octree :**
  - L'octree est composé de nœuds qui peuvent être soit des *feuilles* soit des *nœuds internes*.
  - Dans les différents algorithmes et figures, les lettres N et F font référence respectivement aux nœuds et feuilles de l'octree.
  - Deux nœuds *frères* sont des nœuds ayant le même nœud *parent*.
  - La liste des fils d'un nœud est appelée *fratrie*.
- **Structure de données :**
  - t^N : liste des triangles de m intersectés par un nœud.
  - T^N : liste des triangles de M intersectés par un nœud.
- **Subdivision de l'octree :**
  - $profIn$: *profondeur initiale* des feuilles de l'octree.
  - $profMax$: *profondeur maximale* autorisée des feuilles lors de la subdivision adaptative.
  - V^N : mesure de *variation* du champ de normales de M à l'intérieur des nœuds de l'octree. Cette mesure est basée sur la métrique $L^{2,1}$ [CSAD04].
- **Echantillonnage par lancer de rayons**
  - $\vec{R}$: *rayons* pour la détection d'intersection
  - d_{max} : *distance maximale* de recherche des intersections entre les rayons et M.
  - L_N : *liste des nœuds* de l'octree qu'intersecte un rayon particulier.

A.2 Encodage 2D de l'octree

- **Tri en largeur d'abord**
  - La structure arborescente de l'octree est triée en largeur d'abord dans un tableau 1D. La racine est à l'indice 0.
  - Seuls les nœuds non vides sont sauvegardés dans le tableau.
  - Pour assurer la conversion en texture RGBA, les cases du tableau ont une taille fixées à 4 octets.
- **Position des nœuds dans le tableau 1D**
  - Une feuille occupe un indice du tableau (trois octets pour encoder les trois composantes de la normale associée,)
  - Selon les encodages choisis, les nœuds occupent :
    - un indice si les normales aux nœuds ne sont pas sauvegardés.
    - deux indices dans le cas contraire.
  - *indice* : fait référence à la position des nœuds dans le tableau 1D. Dans le cas où un nœud interne occupe deux cases, l'*indice* fait référence à la position de la première case.
  - Chaque nœud interne possède un offset pour déterminer la position de son premier nœud fils.
  - Pour déterminer la position d'un fils particulier (pas le premier), il faut ajouter un décalage supplémentaire à l'offset vers le premier fils. Pour différencier ces deux notions, nous parlerons uniquement d'offset pour le pointeur vers le premier fils, le terme décalage étant dévolu au décalage pour atteindre un fils particulier dans la fratrie.
  - *Masque des fils* : Ensemble de 8 marqueurs (1 bit chacun) indiquant quels sont les nœuds fils existants.
  - *Masque du genre* : Ensemble de 8 marqueurs (1 bit chacun) la nature des nœuds fils (nœud interne ou feuille).
- **Sauvegarde de l'octree sous forme de texture 2D**
  - *APO* : texture 2D encodant l'octree, construite à partir du tableau 1D, acronyme de l'anglais *Appearance Preserving Octree-texture.*
  - Une case du tableau 1D est représentée par un texel de la texture.
  - Les composantes (x, y, z) des normales et les offsets sont encodées sur les canaux RGB des nœuds.
  - Les masques sont sauvegardés dans les canaux Alpha des texels.
  - *dimXTexture, dimYTexture* : dimensions de la texture 2D de l'*APO.*

A.3 Placage GPU de l'APO

- $Frag$: le fragment courant.
- F_{xyz} : la position dont est issu le fragment courant.
- $\underline{no}$: numéro d'octant d'un nœud de l'octree.
- $\overrightarrow{dep}$: vecteur entre le centre des cellules et la position F_{xyz} des fragments.

Annexe B

Statistiques de subdivision

Ces tableaux contiennent les statistiques de création des *APO*, c'est à dire les nombres de nœuds créés en fonction de la tolérance V^N. Il est intéressant de noter que quel que soit le seuil donné et le maillage, le taux de feuilles par rapport au nombre de nœud est quasiment constant.

Vase Lion	Original : Simplifié :	400 000 triangles 13 212 triangles		
Octree	**Profondeur initiale :** 5 **Profondeur maximale :** 13			
Tolérance	0.5	0.3	0.1	0.05
Nombre total de cellules	670912	1086174	2868614	4718789
Feuilles	529150	849104	2211187	3618852
Nœuds	141762	237070	657427	1099937
Ratio Feuilles/Nœuds	3.73	3.58	3.36	3.29

TAB. B.1 – Composition de l'octree en fonction de la tolérance sur V^N.

Happy Bouddah	Original : Simplifié :	1 087 716 triangles 9 968 triangles		
Octree	**Profondeur initiale :** 5 **Profondeur maximale :** 13			
Tolérance	0.5	0.3	0.1	0.05
Nombre total de cellules	514216	758816	1806252	3113260
Feuilles	403967	593126	1398304	2398172
Nœuds	110249	165690	407948	715088
Ratio Feuilles/Nœuds	3.66	3.58	3.42	3.35

Tab. B.2 – Composition de l'octree en fonction de la tolérance sur V^N.

Moulage de main	Original : Simplifié :	1 546 926 triangles 5 000 triangles		
Octree	**Profondeur initiale :** 5 **Profondeur maximale :** 13			
Tolérance	0.5	0.3	0.1	0.05
Nombre total de cellules	328283	823722	1578067	3081047
Feuilles	256387	407104	1213189	2354193
Nœuds	71896	116618	364878	726854
Ratio Feuilles/Nœuds	3.56	3.49	3.32	3.24

Tab. B.3 – Composition de l'octree en fonction de la tolérance sur V^N.

Grog	Original : Simplifié :	1 752 348 triangles 5 000 triangles		
Octree	**Profondeur initiale :** 5 **Profondeur maximale :** 13			
Tolérance	0.5	0.3	0.1	0.05
Nombre total de cellules	656907	974514	2180268	3455237
Feuilles	508304	750097	1663969	2627258
Nœuds	148873	224417	516299	827979
Ratio Feuilles/Nœuds	3.41	3.34	3.22	3.17

Tab. B.4 – Composition de l'octree en fonction de la tolérance sur V^N.

Neptune	Original : Simplifié :	4 007 872 triangles 15 000 triangles		
Octree	**Profondeur initiale :** 5 **Profondeur maximale :** 13			
Tolérance	0.5	0.3	0.1	0.05
Nombre total de cellules	560452	792601	1745074	2928098
Feuilles	437379	616054	1343299	2239737
Nœuds	123073	176547	401775	688361
Ratio Feuilles/Nœuds	3.55	3.48	3.34	3.25

TAB. B.5 – Composition de l'octree en fonction de la tolérance sur V^N.

XYZRGB Statue	Original : Simplifié :	10 000 000 triangles 20 000 triangles		
Octree	**Profondeur initiale :** 5 **Profondeur maximale :** 13			
Tolérance	0.5	0.3	0.1	0.05
Nombre total de cellules	3170154	4323182	8353868	12251436
Feuilles	2445503	3323350	6380745	9329456
Nœuds	724651	999832	1973123	2921980
Ratio Feuilles/Nœuds	3.37	3.32	3.23	3.19

TAB. B.6 – Composition de l'octree en fonction de la tolérance sur V^N.

Annexe C

Codes GLSL

C.1 Vertex Shader

```
  varying vec4 pos;
  varying vec3 normal, lightDir, eyeVec;

  void main() {
    normal = gl_NormalMatrix * gl_Normal; // facultatif
    vec3 vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightDir = vec3(gl_LightSource[0].position.xyz - vVertex);
    eyeVec = -vVertex;
    pos = gl_Vertex;
    gl_Position = ftransform();
}
```

C.2 Calcul de l'éclairage

Éclairage de Phong classique dans lequel la normale du fragment est fournie par la texture *APO*

```
  vec4 final_color=gl_FrontLightModelProduct.sceneColor * gl_FrontMaterial.ambient
    + gl_LightSource[0].ambient *  gl_FrontMaterial.ambient;

  vec3 N = normalize( fragmentNormal ); // lue dans APO
  vec3 L = normalize(lightDir);
  float lambertTerm = abs(dot(N,L));

  if(lambertTerm > 0.0) {
      final_color += gl_LightSource[0].diffuse *
                     gl_FrontMaterial.diffuse *
                     lambertTerm;
      vec3 E = normalize(eyeVec);
      vec3 R = reflect(-L, N);
      float specular = pow( max(dot(R, E), 0.0), gl_FrontMaterial.shininess);
      final_color += gl_LightSource[0].specular *
                     gl_FrontMaterial.specular *
                     specular;
    }
  gl_FragColor = final_color;
```

C.3 Parcours de l'*APO*, boucle Tant que

```
vec3 centreCellule = vec3(0.5, 0.5, 0.5);
float largeurCellule = 1.0;

vec4 N1, N2;
vec2 texCoords
int index = 0;
int offset;

// lecture premier pixel
texCoords = coordTexture2D(index);
N1 = texture2DRect(APO, texCoords.st);

// lecture deuxième pixel
texCoords = coordTexture2D(index+1);
vec4 N2 = texture2DRect(APO, texCoords.st);

// trouve dans quel octant se trouve le fragment
int oct = celluleFille(Centre, Pos);

while(N1.a != 0) {

  // calcul de l'offset
  offset = offsetPremierFils(N1);
  calculDecalageDansFratrie(N1, N2, oct, offset);

  // mise à jour indice courant
  index += offset;

  // lecture premier pixel
  texCoords = coordTexture2D(index);
  N1 = texture2DRect(APO, texCoords.st);

  // lecture deuxième pixel
  texCoords = coordTexture2D(index+1);
  vec4 N2 = texture2DRect(APO, texCoords.st);

  // trouve dans quel octant se trouve le fragment
  int oct = celluleFille(Centre, Pos);

  // mise à jour cellule
  vec3 dep = sign(Pos - centreCellule);
  dep = dep * (largeurCellule / 4.0);
  CentreCellule += dep;
  largeurCellule /= 2.0;
}
```

C.4 Fragment shader complet

```
//         +--------+        +--------+
//        /    2   /|       /    6   /|
//       +--------+ |      +--------+ |
//       |        | |      |        | |
//   +---------+  | |  +---------+  | |
//  /         /|  | + /         /|  | +
// +---------+ |  |/ +---------+ |  |/
// |         | | -+  |         | |--+
// |   3     | |     |   7     | |
// |         | +     |         | +
// |         |/      |         |/
// +---------+       +---------+
//
//         +--------+        +--------+
//        /   0    /|       /    4   /|
//       +--------+ |      +--------+ |
//       |        | |      |        | |
//   +---------+  | |  +---------+  | |
//  /         /|  | + /         /|  | +
// +---------+ |  |/ +---------+ |  |/
// |         | | -+  |         | |--+
// |    1    | |     |   5     | |
// |         | +     |         | +
// |         |/      |         |/
// +---------+       +---------+

//     Y
//     ^
//     |
//     |
//     |
//     +-------> X
//    /
//   /
//  Z

varying vec4 pos;
varying vec3 normal, lightDir, eyeVec;

uniform int OctreeTextureWidth;
uniform int OctreeTextureHeight;
uniform sampler2D OctreeStructureTexture;
uniform int UseInterpolation;

const float DEP = 0.5;

int adaptativeStop;

// given an index, returns the texture coordinates to access
// element in the 2d texture
vec2 getTextureCoordinates(int index)
{
  int y = index / OctreeTextureWidth;
  int x = index - y*OctreeTextureWidth;
  return vec2((float(x)+DEP) / float(OctreeTextureWidth),
              (float(y)+DEP) / float(OctreeTextureHeight));
}
```

```
// given an octree element center and a position
// gives the index of the subchild in which the position lies
int getChildIndex(vec3 octCenter, vec3 pos)
{
  vec3 dep = pos - octCenter;
  dep = step(0.0, sign(dep));
  return int(dep.x*4.0 + dep.y*2.0 + dep.z);
}


// test if a node is a leaf or not
float isANode(float nodeMask)
{
  return sign(nodeMask);
}


// get the node's first child offset
int getFirstChildOffset(vec4 node)
{
  // <MSB ------------------------------ LSB>
  // <-------------------------------------->
  // <--- R ---><--- G ---><--- B --->
  // <------------- OFFSET ------------->
  // convert RGB bits in an integer value
  return int(node.b * 255.0 +
       node.g * 255.0 * 256.0 +
       node.r * 255.0 * 65536.0);
}



//
bool updateOffsetV2(float childrenMask, float kindMask, int wantedChild,
        out int offsetAdjust)
{
  vec2 masques = vec2(kindMask * 255.0, childrenMask * 255.0);
  vec2 bitValues;
  offsetAdjust = 0;

  for(int it = 0 ; it<wantedChild ; ++it)
    {
      //
      bitValues = mod(masques, 2.0);
      offsetAdjust += int((bitValues[0] + 1.0) * bitValues[1]);

      // divide by 2 each mask
      masques -= bitValues;
      masques /= 2.0;
    }

  // test id the wanted child exists
  return (mod(masques[1], 2.0) == 1.0);
}
```

```
// traverse the octree encoded int the textures
// return the reached depth
int traverseOctree(in vec3 position,
        out vec3 fragmentNormal,
        out float leafEnd, out bool childExistEnd, out bool outside,
        out float OEwidth, out vec3 octreeElementCenter)
{
  //
  vec4 octreeElement;
  vec4 extraOctreeElement;

  ///////////////////////////////////////////////
  // init the traversal
  int index;
  int depth = 0;

  // initially the octree is centered on the point 0.5 0.5 0.5
  octreeElementCenter.xyz = vec3(0.5, 0.5, 0.5);
  // the width of current octree element
  OEwidth = 1.0;
  // first index
  index = 0;

  //
  int child;
  int offset;
  float isnode;
  bool childExist;
  vec2 texCoords;
  int offsetAdjust;
  //
  int lastOffset;

  // get root
  texCoords = getTextureCoordinates(index);
  octreeElement = texture2D(OctreeStructureTexture, texCoords.st);
  isnode = sign(octreeElement.a);//isANode(octreeElement.a);

  // find extra data
  // read infos about child kind in the node 2nd pixel data
  texCoords = getTextureCoordinates(index + 1);
  extraOctreeElement = texture2D(OctreeStructureTexture, texCoords.st);

  // find in which child lies the fragment
  child = getChildIndex(octreeElementCenter, position);

  // test its existance, find offset adjust to reach it
  childExist = updateOffsetV2(octreeElement.a, extraOctreeElement.a,
          child, offsetAdjust);

  // main traversal loop
  while(octreeElement.a != 0.0 && childExist && (depth < adaptativeStop))
    {
      // update index, index now points to the desired child
      // ad offset to first child, plus offset to desired child
      lastOffset = getFirstChildOffset(octreeElement);
      index += lastOffset + offsetAdjust;

      if (lastOffset < 16777216) // condition magique sur certaines cartes
  {
    // convert index in texture coordinates
    // get octree element
```

```
    texCoords = getTextureCoordinates(index);
    // get node
    octreeElement = texture2D(OctreeStructureTexture, texCoords.st);

    // test whether the element is a node
    isnode = sign(octreeElement.a);//isANode(octreeElement.a);

    // find extra data
    // read infos about child genus in the node 2nd pixel data
    texCoords = getTextureCoordinates(index + 1);
    extraOctreeElement = texture2D(OctreeStructureTexture, texCoords.st);

    // update infos about cell width
    OEwidth /= 2.0;

    // update the octreeElement center
    octreeElementCenter += sign(position - octreeElementCenter) * (OEwidth / 2.0);

    // find in which child lies the fragment
    child = getChildIndex(octreeElementCenter, position);

    // test its existance, find offset adjust to reach it
    childExist = updateOffsetV2(octreeElement.a, extraOctreeElement.a, child,
              offsetAdjust);

    ++depth;
  }
    }


  octreeElement = mix(octreeElement, extraOctreeElement, isnode);

   // get Normal
  octreeElement = octreeElement * 2.0 - 1.0;
  fragmentNormal = gl_NormalMatrix * octreeElement.xyz;
  //
  childExistEnd = childExist;
  leafEnd = 1.0-sign(octreeElement.a);

  return depth;
}


// the main program
void main()
{
  // final texture normal
  vec3 usedNormal = normal;
  vec3 octreeElementCenter;

  //int depth;
  float leafEnd;
  bool childExist;
  float OEwidth;
  float tempWidth;

  vec3 tempCenter, tempCenter2;

  float dx, dy, dz;
  vec3 nP0, nP1;
  vec4 cP0, cP1;
  float leP0, leP1;
```

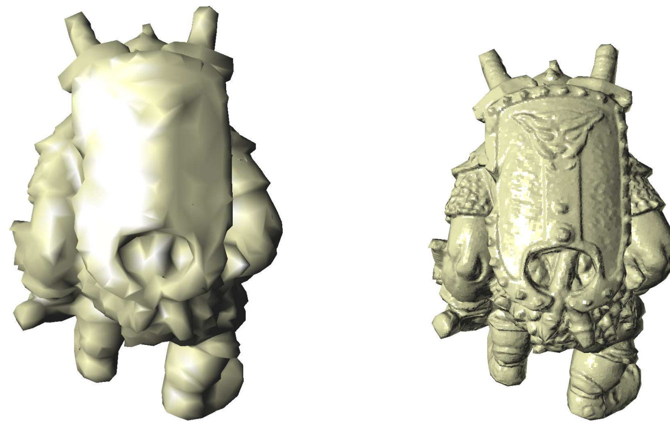
```
bool ceP0, ceP1;
bool outside;
vec3 centerP1;
vec3 normal0, normal1, normalE, normalF;
vec3 dep;

//
vec3 dxpixpos = dFdx(pos.xyz);
vec3 dypixpos = dFdy(pos.xyz);
float lengthDF = length(dxpixpos)<length(dypixpos)?length(dxpixpos):length(
    dypixpos);
adaptativeStop = int(ceil(log2(2.0/lengthDF)));



// find color of the fragment
int def, defN;
def = traverseOctree(pos.xyz,
         nP0,
         leP0, ceP0, outside,
         OEwidth, octreeElementCenter);


usedNormal = nP0;
leafEnd = leP0;
childExist = ceP0;


if (UseInterpolation == 1)
  {
    // find the color of eight neighbours
    // to do a bilinear filtering of the fragment color

    //
    dep = pos.xyz - octreeElementCenter;
    dep = sign(dep);


    //      +----D-----------------------+
    //     /    /|                      /|
    //    /    / |                     / |
    //   /    /  |                    /  |
    //  /    /   |                   /   |
    // +----B-----------------------+    |
    // |    |    |                  |    |
    // |    |    |                  |    |
    // |    |    |                  |    |
    // |    |    F                  |    |
    // |    |   /|                  |    |
    // |    |  S |                  |    |
    // |    | /  |                  |    |
    // |    |/   |                  |    |
    // |    E    |                  |    |
    // |    |  - C -  -  -  -  -  - |-  -+
    // |   /|   /                   |   /
    // |  / |  /                    |  /
    // | /  | /                     | /
    // |/   |/                      |/
    // +----A-----------------------+
```

```
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
// find the colors of the first plane
// point A
float dpx = OEwidth/2.0 - abs(octreeElementCenter.x - pos.x);
dpx *= dep.x * 1.2;
float dpy = OEwidth/2.0 - abs(octreeElementCenter.y - pos.y);
dpy *= dep.y * 1.2;
float dpz = OEwidth/2.0 - abs(octreeElementCenter.z - pos.z);
dpz *= dep.z * 1.2;

//
centerP1 = pos.xyz + vec3(dpx, 0.0, 0.0);
defN = traverseOctree(centerP1,
    nP1,
    leP1, ceP1, outside,
    tempWidth, tempCenter);

//
dx = abs(pos.x - octreeElementCenter.x) / (OEwidth);

// point A normal
normal0 = mix(nP0, nP1, dx);

//
centerP1 =  pos.xyz + vec3(0.0, dpy, 0.0);
defN = traverseOctree(centerP1,
    nP0,
    leP0, ceP0, outside,
    tempWidth, tempCenter);

//
centerP1 = pos.xyz + vec3(dpx, dpy, 0.0);
defN = traverseOctree(centerP1,
    nP1,
    leP1, ceP1, outside,
    tempWidth, tempCenter2);

// point B normal
normal1 = mix(nP0, nP1, dx);


//
dy = abs(pos.y - octreeElementCenter.y) / (OEwidth);

// point E normal
normalE = mix(normal0, normal1, dy);


////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
// find the colors of the second plane
// point C
centerP1 = pos.xyz + vec3(0.0, 0.0, dpz);
defN = traverseOctree(centerP1,
    nP0,
    leP0, ceP0, outside,
    tempWidth, tempCenter);

//
centerP1 = pos.xyz + vec3(dpx, 0.0, dpz);
defN = traverseOctree(centerP1,
```

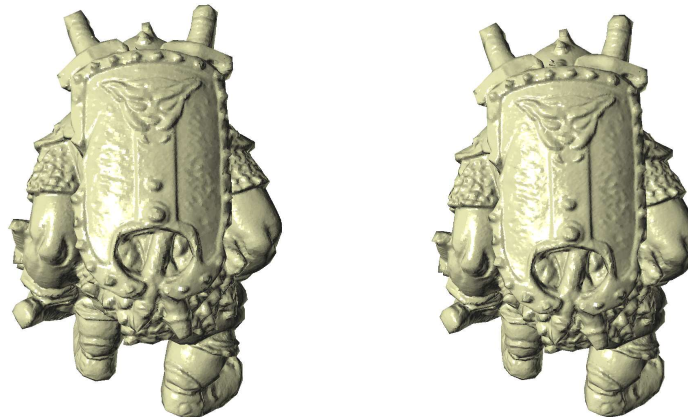
```
            nP1,
            leP1, ceP1, outside,
            tempWidth, tempCenter);

        //
        normal0 = mix(nP0, nP1, dx);


        // point D
        centerP1 = pos.xyz + vec3(0.0, dpy, dpz);
        defN = traverseOctree(centerP1,
            nP0,
            leP0, ceP0, outside,
            tempWidth, tempCenter);

        //
        centerP1 = pos.xyz + vec3(dpx, dpy, dpz);
        defN = traverseOctree(centerP1,
            nP1,
            leP1, ceP1, outside,
            tempWidth, tempCenter);

        //
        normal1 = mix(nP0, nP1, dx);

        // sample point final color and normal
        dz = abs(pos.z - octreeElementCenter.z) / (OEwidth);

        // compute final normal
        usedNormal = mix(normalE, normalF, dz);
    }


  // compute color
  vec4 final_color=gl_FrontLightModelProduct.sceneColor * gl_FrontMaterial.ambient
    + gl_LightSource[0].ambient *  gl_FrontMaterial.ambient;

  vec3 N = normalize(usedNormal);
  vec3 L = normalize(lightDir);

  float lambertTerm = abs(dot(N,L));

  if(lambertTerm > 0.0)
    {
      final_color += gl_LightSource[0].diffuse *
  gl_FrontMaterial.diffuse *
        lambertTerm;

      vec3 E = normalize(eyeVec);
      vec3 R = reflect(-L, N);
      float specular = pow( max(dot(R, E), 0.0),
                            gl_FrontMaterial.shininess * 0.2);
      final_color += gl_LightSource[0].specular *
        gl_FrontMaterial.specular *
        specular;
    }
  gl_FragColor = final_color;
}
```

Annexe D

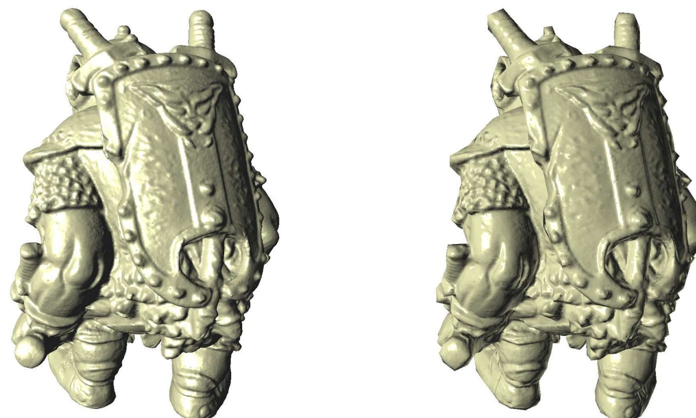
Rendus APO



(a) A gauche le maillage simplifié, à droite rendu avec une *APO*, la tolérance est de 0.5

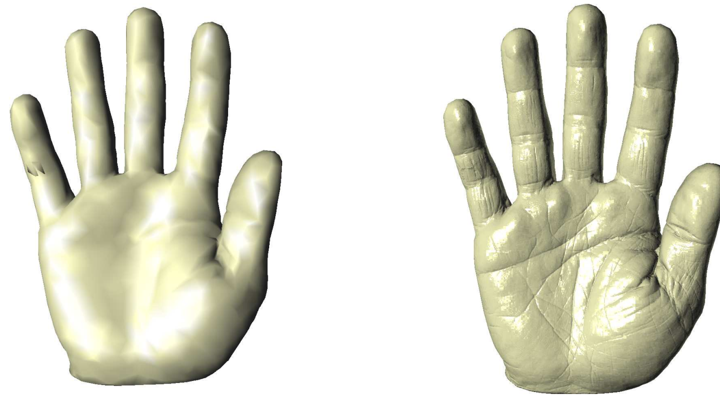


(b) Tolérance respectives de 0.1 et 0.01

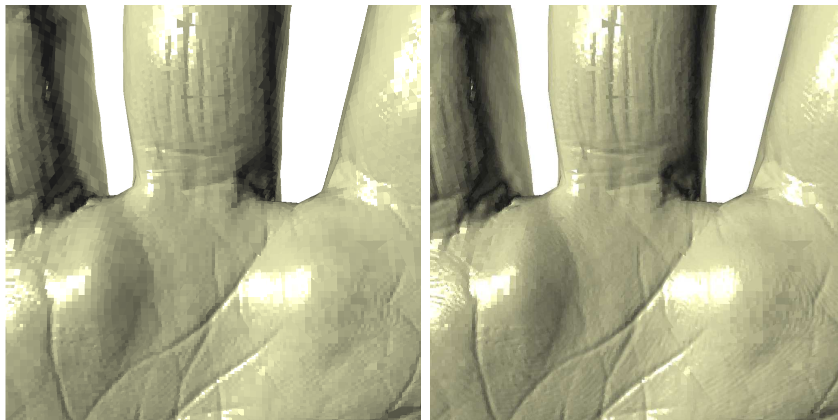


(c) A gauche le maillage original, à droite rendu avec l'*APO* de tolérance 0.01, filtrage bilinéaire activé.

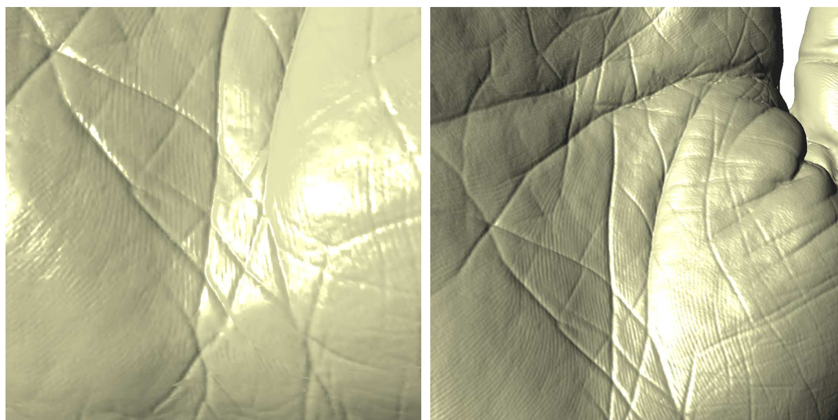
FIG. D.1 – L'objet simplifié est composé de 5000 polygones, l'original de 1.7 millions.



(a) A droite, le maillage simplifié, à gauche une vue d'ensemble de la main de Pierre Alliez rendue avec une *APO* de tolérance 0.5



(b) Tolérance respectives de 0.5 et 0.1



(c) A gauche la tolérance est de 0.01, à droite se trouve le maillage original

FIG. D.2 – L'objet simplifié est composé de 5000 polygones, l'original de 1.5 millions.

(a) A droite, le maillage simplifié, à gauche une vue d'ensemble de la statue rendue avec une *APO* de tolérance 0.5

(b) Détails de la statue, avec tolérances respectives de 0.5 et 0.1

FIG. D.3 – L'objet simplifié est composé de 20 mille polygones, l'original de 10 millions.

FIG. D.4 – Le maillage simplifié est cette fois composé de 100 mille triangles (M) de 10 millions.

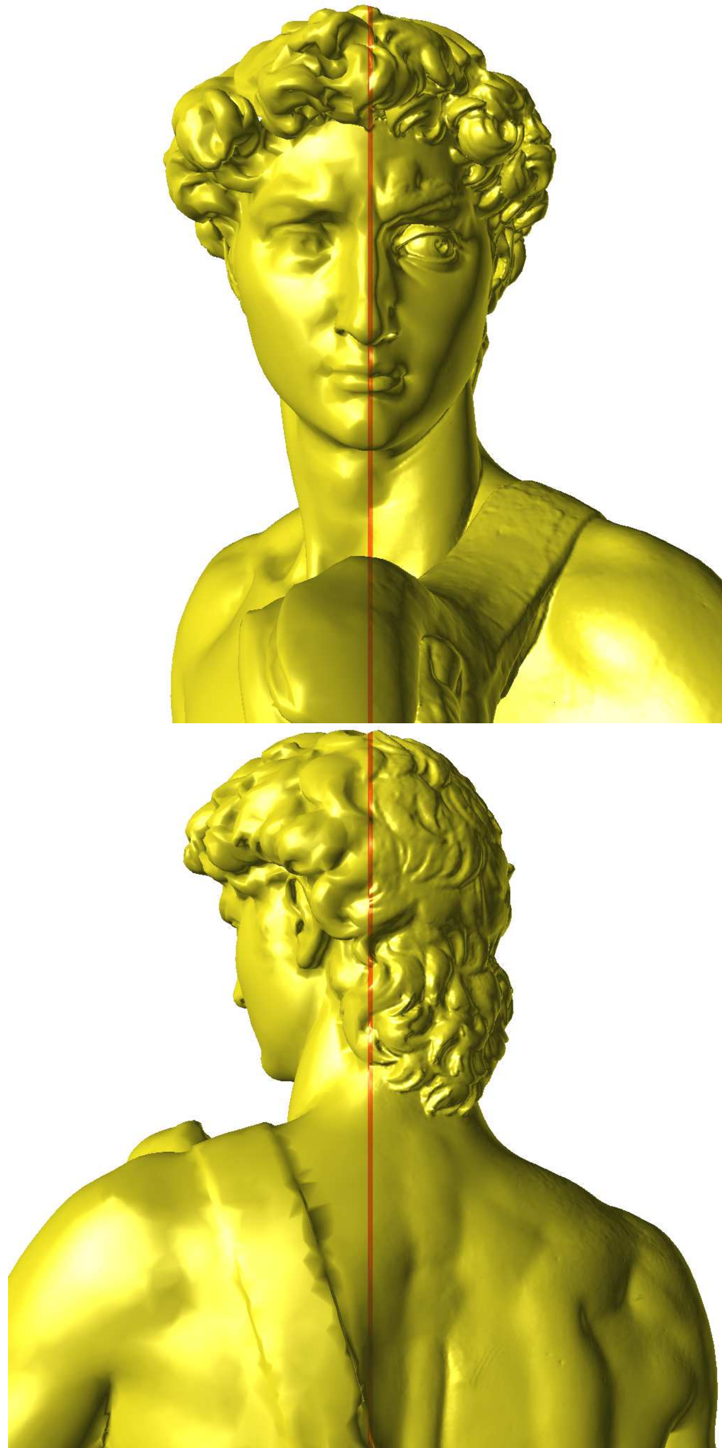


FIG. D.5 – Détails sur la statue du David. L'objet original est composé de 7 millions de triangles, le simplifié de 100 mille.

Bibliographie

[AD01] Pierre Alliez and Mathieu Desbrun. Progressive compression for lossless transmission of triangle meshes. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 195–202. ACM Press / ACM SIGGRAPH, 2001.

[AGL91] Mark Agate, Richard L. Grimsdale, and Paul F. Lister. The hero algorithm for ray-tracing octrees. In *Advances in Computer Graphics Hardware IV (Eurographics'89 Workshop)*, pages 61–73, London, UK, 1991. Springer-Verlag.

[AM01] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. *journal of graphics tools*, 6(1) :29–33, 2001.

[AMH02] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering Second Edition.* A K Peters, Ltd, 2002.

[BD02] David Benson and Joel Davis. Octree textures. *ACM Transactions on Graphics*, 21(3) :785–790, July 2002.

[BDS05] Tamy Boubekeur, Florent Duguet, and Christophe Schlick. Rapid visualization of large point-based surfaces. In M. Mudge, N. Ryan, and R. Scopigno, editors, *Proceedings of the International Symposium on Virtual Reality, Archeology and Cultural Heritage (VAST).* Eurographics, Eurographics, November 2005.

[BHGS06] Tamy Boubekeur, Wolfgang Heidrich, Xavier Granier, and Christophe Schlick. Volume-surface trees. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2006)*, 25(3) :399–406, 2006.

[BHP06] Ying Bai, Xiao Han, and Jerry L. Prince. Octree-based topology-preserving isosurface simplification. In *CVPRW '06 : Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop*, page 81, Washington, DC, USA, 2006. IEEE Computer Society.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11(2) :192–198, July 1977.

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3) :286–292, 1978.

[BPZ99] Chandrajit L. Bajaj, Valerio Pascucci, and Guozhong Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 307–316, San Francisco, 1999.

[BS08] Tamy Boubekeur and Christophe Schlick. A flexible kernel for adaptive mesh refinement on gpu. *Computer Graphics Forum*, 27(1) :102–114, 2008. doi : 10.1111/j.1467-8659.2007.01040.x.

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, University of Utah, December 1974.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987.

[CMR+99] Paolo Cignoni, Claudio Montani, Claudio Rocchini, Roberto Scopigno, and Marco Tarini. Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer*, 15, 1999.

[CMSR98] P. Cignoni, C. Montani, R. Scopigno, and C. Rocchini. A general method for preserving attribute values on simplified meshes. In *VIS '98 : Proceedings of the conference on Visualization '98*, pages 59–66, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[COLR99] Daniel Cohen-Or, David Levin, and Offir Remez. Progressive compression of arbitrary triangular meshes. In *VIS '99 : Proceedings of the conference on Visualization '99*, pages 67–72, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[COM98] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *SIGGRAPH '98 : Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122, New York, NY, USA, 1998. ACM.

[COM+07] Wei Cheng, Wei Tsang Ooi, Sebastien Mondet, Romulus Grigoras, and Géraldine Morin. An analytical model for progressive mesh streaming. In *MULTIMEDIA '07 : Proceedings of the 15th international conference on Multimedia*, pages 737–746, New York, NY, USA, 2007. ACM.

[Coo84] Robert L. Cook. Shade trees. In *SIGGRAPH '84 : Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.

[CSAD04] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. In *SIGGRAPH '04 : ACM SIGGRAPH 2004 Papers*, pages 905–914, New York, NY, USA, 2004. ACM.

[CVM+96] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *SIGGRAPH '96 : Proceedings of the 23rd annual*

conference on Computer graphics and interactive techniques, pages 119–128, New York, NY, USA, 1996. ACM.

[Deb96] Paul E. Debevec. *Modeling and Rendering Architecture from Photographs.* PhD thesis, University of California at Berkeley, Computer Science Division, Berkeley CA, 1996.

[Dee95] Michael Deering. Geometry compression. *Computer Graphics*, 29(Annual Conference Series) :13–20, 1995.

[Don05] William Donnely. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Per-Pixel Displacement Mapping with Distance Functions, pages 123–136. Addison Wesley, 2005.

[DZ91] Michael J. DeHaemer, Jr. and Michael J. Zyda. Simplification of objects rendered by polygonal approximations. *Computers and Graphics*, 15(2) :175–184, 1991.

[EWWL98] Jon P. Ewins, Marcus D. Waller, Martin White, and Paul F. Lister. MIP-Map level selection for texture mapping. *IEEE Transactions on Visualization and Computer Graphics*, 4(4), October 1998.

[FH05] Michael S. Floater and Kai Hormann. Surface parameterization : a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in multiresolution for geometric modelling*, pages 157–186. Springer Verlag, 2005.

[Fou92] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, Vancouver, BC, Canada, 11 May 1992.

[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.

[GD03] Philippe Guigue and Olivier Devillers. Fast and robust triangle-triangle overlap test using orientation predicates. *journal of graphics tools*, 8(1) :25–42, 2003.

[gDGPR02] David (grue) DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. Painting and rendering textures on unparameterized models. In *SIGGRAPH '02 : Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 763–768, New York, NY, USA, 2002. ACM.

[Gla89] Andrew S. Glassner, editor. *An introduction to ray tracing.* Academic Press Ltd., London, UK, UK, 1989.

[HBR+07] Julien Hadim, Tamy Boubekeur, Mickaël Raynaud, Xavier Granier, and Christophe Schlick. On-the-fly appearance quantization on gpu for 3d broadcasting. In *ACM SIGGRAPH Web3D*. ACM, ACM Press, 2007.

[Hop96] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96 : Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, New York, NY, USA, 1996. ACM.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series) :189–198, 1997.

[HSRG07] Charles Han, Bo Sun, Ravi Ramamoorthi, and Eitan Grinspun. Frequency domain normal map filtering. *ACM Trans. Graph.*, 26(3) :28, 2007.

[HW91] Andrew Hunter and Philip Willis. Classification of quad-encoding techniques. *Computer Graphics Forum*, 10(2), jun 1991.

[Jan86] Frederik Jansen. Data structures for ray tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, editors, *Data Structures for Raster Graphics*, pages 57–73. Springer-Verlag, 1986.

[KFCO+07] Johannes Kopf, Chi-Wing Fu, Daniel Cohen-Or, Oliver Deussen, Dani Lischinski, and Tien-Tsin Wong. Solid texture synthesis from 2d exemplars. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3) :to appear, 2007.

[KL96] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *SIGGRAPH '96 : Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 313–324, New York, NY, USA, 1996. ACM.

[KL05] David Koller and Marc Levoy. Protecting 3d graphics content. *Commun. ACM*, 48(6) :74–80, 2005.

[KLS+05] Joe Kniss, Aaron Lefohn, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Octree textures on graphics hardware. In *SIGGRAPH '05 : ACM SIGGRAPH 2005 Sketches*, page 16, New York, NY, USA, 2005. ACM.

[KSS97] Leif Kobbelt, Marc Stamminger, and Hans-Peter Seidel. Using subdivision on hierarchical data to reconstruct radiosity distribution. *Computer Graphics Forum*, 16(3) :C347–C355, 1997.

[KSS00] Andrei Khodakovsky, Peter Schröder, and Wim Sweldens. Progressive geometry compression. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 271–278. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[KTL+04] David Koller, Michael Turitzin, Marc Levoy, Marco Tarini, Giuseppe Croccia, Paolo Cignoni, and Roberto Scopigno. Protected interactive 3d graphics via remote rendering. In *SIGGRAPH '04 : ACM SIGGRAPH 2004 Papers*, pages 695–703, New York, NY, USA, 2004. ACM.

[LBJS07] Julien Lacoste, Tamy Boubekeur, Bruno Jobard, and Christophe Schlick. Appearance preserving octree-textures. In *GRAPHITE '07 : Proceedings of the 5th international conference on Computer graphics and interactive*

techniques in Australia and Southeast Asia, pages 87–93, New York, NY, USA, 2007. ACM.

[LD07] Sylvain Lefebvre and Carsten Dachsbacher. Tiletrees. In *I3D '07 : Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 25–31, New York, NY, USA, 2007. ACM.

[Lef05] Sylvain Lefebvre. *Modeles d'habillage de surface pour la synthese d'images.* PhD thesis, Universite Joseph Fourier, Grenoble, France., avril 2005.

[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.

[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.

[LLY06] Patric Ljung, Claes Lundström, and Anders Ynnerman. Multiresolution interblock interpolation in direct volume rendering. *EUROVIS*, 2006.

[LPJ08] Julien Lacoste, Camille Perin, and Bruno Jobard. Progressive transmission of appearance preserving octree-textures. In *Web3D '08 : Proceedings of the 13th international symposium on 3D web technology*, pages 19–22, New York, NY, USA, 2008. ACM.

[LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21(3) :362–371, 2002.

[LSK+06] Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. Glift : Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, 25(1) :60–99, 2006.

[MMO05] Fabio Policarpo Manuel M. Oliveira. An efficient representation for surface details. Technical report, january 2005.

[MP97] M.Garland and P.Heckbert. Surface simplification using quadric error metrics. *ACM SIGGRAPH*, 1997.

[PAC97] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *SIGGRAPH '97 : Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 303–306, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[PDG05] Damien Porquet, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Real-time high-quality view-dependent texture mapping using per-pixel visibility. In *GRAPHITE '05 : Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 213–220, New York, NY, USA, 2005. ACM.

[Pea85] Darwyn R. Peachey. Solid texturing of complex surfaces. In *SIGGRAPH '85 : Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 279–286, New York, NY, USA, 1985. ACM Press.

[Per85] K. Perlin. An image synthesizer. *Computer Graphics*, 19(3) :287–296, July 1985.

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

[POJ05] Fábio Policarpo, Manuel M. Oliveira, and ao L. D. Comba Jo˙ Real-time relief mapping on arbitrary polygonal surfaces. In *I3D '05 : Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 155–162, New York, NY, USA, 2005. ACM.

[PR00] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1) :79–93, /2000.

[PRL+08] Romain Pacanowski, Mickaël Raynaud, Julien Lacoste, Xavier Granier, Patrick Reuter, Christophe Schlick, and Pierre Poulin. Compact strucutures for interactive global illumination on large cultural objects. In *Proceedings of the 9th International Symposium on Virtual Reality, Archeology and Cultural Heritage (VAST)*. Eurographics, December 2008.

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat : A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[RL01] Szymon Rusinkiewicz and Marc Levoy. Streaming qsplat : a viewer for networked visualization of large, dense models. In *I3D '01 : Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 63–68, New York, NY, USA, 2001. ACM.

[Ros05] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[RUL00] J. Revelles, Carlos Ureña, and Miguel Lastra. An efficient parametric algorithm for octree traversal. In *WSCG*, 2000.

[Sam89] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4) :445–60, 1989. includes code.

[SGG+00] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *SIGGRAPH '00 : Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 327–334, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[SGR96] Marc Soucy, Guy Godin, and Marc Rioux. A texture-mapping approach for the compression of colored 3D triangulations. *The Visual Computer*, 12(10) :503–514, 1996.

[SH02] Alla Sheffer and John C. Hart. Seamster : inconspicuous low-distortion texture seam layout. In *VIS '02 : Proceedings of the conference on Visualization '02*, pages 291–298, Washington, DC, USA, 2002. IEEE Computer Society.

[SKU08] L. Szirmay-Kalos and T. Umenhoffer. Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum*, 27(1), 2008.

[Sun91] Kelvin Sung. A DDA octree traversal algorithm for ray tracing. *Eurographics '91*, pages 73–85, September 1991.

[SW91] John Spackman and Philip J. Willis. The smart navigation of a ray through an oct-tree. *Computers & Graphics*, 15(2) :185–194, 1991.

[TGHL98] Gabriel Taubin, André Gueziec, William Horn, and Francis Lazarus. Progressive forest split compression. *Computer Graphics*, 32(Annual Conference Series) :123–132, 1998.

[Tok05] Michael Toksvig. Mipmapping normal maps. *journal of graphics tools*, 10(3) :65–71, 2005.

[TS84] Markku Tamminen and Hanan Samet. Efficient octree conversion by connectivity labeling. *SIGGRAPH Comput. Graph.*, 18(3) :43–51, 1984.

[WBMS05a] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. *Journal of graphics tool*, 10(1) :49–54, 2005.

[WBMS05b] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. In *SIGGRAPH '05 : ACM SIGGRAPH 2005 Courses*, page 9, New York, NY, USA, 2005. ACM.

[Wil83] Lance Williams. Pyramidal parametrics. In *SIGGRAPH '83 : Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, New York, NY, USA, 1983. ACM Press.

[Woo90] Andrew Woo. *Fast ray-box intersection*, pages 395–396. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[WSC+95] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. Octree-R : An Adaptive Octree for Efficient Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4) :343–349, December 1995.

Abstract

Graphical applications tend to offer a more and more realistic rendering of 3D scenes thanks to the growing complexity of displayed objects. However the meshes of these objects are sometimes too big to be rendered in realtime even on recent GPUs, and they must be simplified to reach interactive framerates. One clever method to replace the lost details is to store the normals of the meshes in textures, and to use them while rendering to compute the shading of each fragment. This appearance preserving method, known as normal mapping relies on a 2D parameterization of the 3D meshes, which sometimes can't be obtained without any user control.

We propose in this thesis an alternative method of appearance preserving based on *octree-textures*. The octree creation is driven by the normal variation accross the object's surface, thus the sampling can be adapted to local surface details. Thanks to the volumetric nature of the octree, no parameterizaton is required, making the creation process totally automatic. We propose a compact encoding of the octrees in 2D textures which can be used by programmable GPUs for realtime rendering. We also present a conversion of our octree-textures in standard 2D normal maps in which all the details are preserved. Finally we present an original application of our encoding, by showing the usability of our textures in a network visualization system. The simplified mesh is rapidly transmitted, and while the user can interact with this model, the octree-texture details are progressively downloaded and refined on the mesh.

Keywords: appearance preserving, texture, octree-texture, GPU programming, computer graphics, 3D, mesh

Résumé

Les applications graphiques tendent à offrir un rendu interactif de scènes 3D de plus en plus réaliste, résultant en partie de l'accroissement de la richesse géométrique des objets affichés. Le maillage de certains de ces objets est toutefois trop complexe au regard des capacités de traitement des cartes graphiques et doit donc être simplifié afin de conserver un bon niveau d'interactivité. Une méthode élégante pour pallier à la perte des détails les plus fins induits par la simplification consiste à les stocker sous forme de normales dans des textures de grande résolution et de les utiliser lors du calcul de l'éclairage de ces objets. Cette technique de *préservation d'apparence* repose cependant traditionnellement sur des opérations complexes de paramétrisation 2D de maillages 3D qui sont encore souvent impossibles à réaliser sans l'intervention d'un utilisateur.

Nous proposons dans ce mémoire une méthode alternative de préservation de détails basée sur les *octree-textures*. La création de l'octree est pilotée par la variation des normales à la surface des maillages originaux, adaptant l'échantillonnage des normales à la richesse de détails locale de la surface. Grâce à la nature volumique des octree-textures, aucune opération de paramétrisation n'est requise, rendant le processus de création des textures totalement automatique. Nous proposons un encodage compact de l'octree sous forme de textures 2D exploitables par les GPU programmables, et nous détaillons l'utilisation de ces textures pour le rendu interactif. Nous présentons également un processus de conversion en atlas de textures 2D classiques dans lesquels tous les détails de l'octree-texture sont conservés. Enfin nous montrons l'adéquation de cette représentation des maillages détaillés avec leur visualisation à distance via le réseau. La transmission instantanée du maillage simplifié permet une interaction immédiate avec l'objet 3D pendant que l'affichage se raffine progressivement à mesure du téléchargement des normales les plus précises.

Mots-clés: préservation d'apparence, texture, octree-texture, programmation GPU, informatique graphique, 3D, maillage