Étude et développement d'un système multi-échelle pour la visualisation réaliste et interactive de végétaux (rendu volumique)

Vincent Vidal

PROJET DE TROISIÈME ANNÉE

RAPPORT FINAL

SEPTEMBRE 2007

TITRE DU SUJET :

Étude et développement d'un système multi-échelle pour la visualisation réaliste et interactive de végétaux (rendu volumique)

Laboratoires :

INRIA Rhône-Alpes **Équipe EVASION** Inovallée 655 Avenue de l'Europe Montbonnot 38334 Saint Ismier cedex, France	**Institute of Automation** **Chinese Academy of Sciences** **LIAMA** (Laboratoire Franco-Chinois d'Informatique, d'Automatique et de Mathématiques Appliquées) 95 Zhongguancun East Road Beijing 2728 P.O. 100080 P.R. China

Période du stage : Du lundi 5 février au mardi 31 juillet 2007. 4 mois ½ passés au LIAMA à Pékin et 1 mois ½ au sein de l'équipe EVASION de l'INRIA Rhône-Alpes.

Responsable du projet : M. Philippe DECAUDIN (Philippe.Decaudin<at>imag.fr)

Tuteur ENSIMAG : M. Antoine BOUTHORS (Antoine.Bouthors<at>imag.fr)

Étudiant : M. Vincent VIDAL (vidal.vince<at>gmail.com)

Filière : Images et Réalité Virtuelle

ENSIMAG
681 rue de la Passerelle – Domaine Universitaire – BP 72
38402 ST MARTIN D'HERES Cedex – France
Tél : (33) 04.76.82.72.22 – Fax : (33) 04.76.82.72.50

PROJET DE TROISIÈME ANNÉE

RAPPORT FINAL

SEPTEMBRE 2007

TITRE DU SUJET :

Étude et développement d'un système multi-échelle pour la visualisation réaliste et interactive de végétaux (rendu volumique)

SUJET:

Le LIAMA (via son projet GreenLab) développe des modèles de croissance de plantes qui leur permettent de générer des scènes virtuelles peuplées de végétaux. Le résultat est une représentation en volume (voxels 3D) de la scène. L'utilisation de représentations voxels dédiées permet une visualisation interactive de plantes lorsqu'elles sont vues de loin (voir par exemple http://evasion.imag.fr/Publications/2004/DN04), et d'autres représentations (comme les maillages 3D) peuvent être utilisées pour les vues rapprochées.

L'objet du stage est d'étudier et d'implémenter une représentation adaptative basée sur les données volumiques de plantes et d'arbres et qui permette d'éliminer des espaces vides. L'algorithme exploitera la forme de l'arbre (ou de la plante) pour le représenter de près par un groupe de volumes (ensembles de branches et leurs feuillages) et pour les agréger ou désagréger dynamiquement lorsque l'observateur s'en éloigne ou s'en rapproche.

L'implémentation des algorithmes et structures de données se fera en C++ et nécessitera aussi de programmer la carte graphique (pixel et vertex *shaders*).

RESUME DU RAPPORT:

Ce rapport introduit une approche intéressante pour le rendu volumique basé sur des textures 3D. En utilisant une structure de *KD-Tree*, construite à partir d'un maillage 3D ou d'une représentation volumique d'arbre ou de plante, une liste de sous-volumes englobants les parties non-vide est générée. Un nombre relativement petit de sous-volumes suffit à englober finement les espaces non-vides. La liste est ensuite utilisée pour ne rendre que les parties correspondantes de la représentation volumique associée. Ce qui permet un gain important au niveau de la vitesse d'affichage.

Après avoir introduit le rendu volumique basé sur des textures 3D, ainsi que la construction et l'intérêt de l'utilisation des *KD-Trees*, une présentation des différentes heuristiques utilisées pour la construction est faite. Elle met en avant les possibilités et limites de chaque approche. Les points techniques les plus intéressants sont détaillés. Les résultats obtenus ainsi que les perspectives résultantes montrent l'intérêt de notre méthode.

M O T S - C L E S :
Rendu volumique ; KD-Tree ; Suppression d'espaces vides

Table des matières

Remerciements

Je remercie Monsieur Philippe DECAUDIN pour ses idées et pour le savoir-faire qu'il m'a transmis pendant ce stage. J'ai vraiment apprécié le travail sous sa supervision, car Philippe est un chercheur de haut niveau qui m'a beaucoup touché par ses qualités humaines et par son soutien tout au long de mon stage.

Je tiens aussi à remercier toutes les personnes qui m'ont aidé pour que mon séjour en Chine se passe dans les meilleurs conditions, en particulier Mme Anne Pierson et les secrétaires du LIAMA.

I - Introduction et objectifs du projet

A - Le rendu volumique

1 - Introduction générale

L'expression « rendu volumique » désigne un ensemble de techniques qui permettent la visualisation de données 3D (échantillons) sous la forme d'images 2D (projection de données volumiques). Les données 3D sont associées à une grille de points 3D, qui va définir leur position, leur voisinage et l'algorithme de visualisation à utiliser. Nous travaillerons par la suite avec des grilles cubiques rectilignes (isotropes ou anisotropes). Sachez néanmoins qu'il existe des grilles curvilignes et des grilles non structurées (confère figure 1), car les données volumiques ne sont pas toujours superposables sur une grille rectiligne. Chaque élément « volumique » de la grille est un voxel (*volume element* ou *volumetric pixel*), qui peut être vu comme un petit cube. Un voxel est le plus petit élément de volume indivisible dans un système à trois dimensions. La résolution utilisée pour la grille définit le nombre total de voxels.

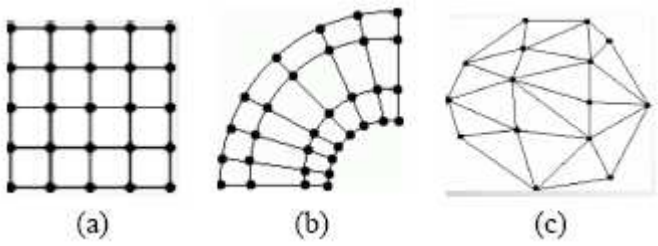


Fig. 1: (a) Grille rectiligne ; (b) Grille curviligne ; (c) Grille non structurée.

Un voxel émet et absorbe de la lumière. L'émission détermine la couleur et l'intensité de du voxel, tandis que l'absorption est liée à son opacité. Plus un voxel est opaque, plus son absorption de lumière sera importante. Lorsqu'on cherche la quantité de lumière qui traverse un voxel, on parle de la notion duale de l'opacité, la transparence. En définissant l'opacité O et la transparence T d'un voxel v comme un scalaire sur l'intervalle [0.0; 1.0], la relation élémentaire **T(v) = 1 – O(v)** se déduit. La couleur et l'opacité d'un voxel peuvent être utilisées pour classifier certaines parties du volume.

Les données volumiques peuvent avoir des origines et formes diverses. Il y a des nuages de points 3D associés à un scalaire qui proviennent de scanners (rayons-X), de CT (*Computed Tomography*), de PET (*Positron Emission Tomography*), d'ultrasons, d'IRM (Imagerie par Résonance Magnétique), ou de modèles mathématiques (qui intègrent en général des propriétés physiques comme en mécanique des fluides). Ces données 3D sont représentées sous la forme d'un couple (Position 3D, Valeur scalaire ou champ de vecteurs), les valeurs scalaires pouvant être des intensités ou couleurs, des densités de matière ou d'autres propriétés (salinité d'un sol, concentration d'un composant,...). Il est clair que la signification de cette valeur scalaire est à la base du type de visualisation qui va être réalisé. Parfois les données volumiques sont incomplètes ou même fausses

(trou ou valeur incohérente). C'est pourquoi il est possible de passer par une phase d'analyse et de correction des données pour produire un ensemble cohérent et utilisable de données.

Avant d'utiliser une technique de rendu spécifique pour visualiser les données, une étape supplémentaire de sélection/modification d'un sous-ensemble de données, une classification, peut être réalisée (au moyen d'une fonction de transfert). La sélection est une étape de réduction de la masse de donnée pour la rendre plus intelligible, car des données volumiques qui sont « volumineuses et complexes » sont difficiles à interpréter. La modification consiste à transformer notre ensemble de voxels en des voxels visualisables, c'est-à-dire associés à une intensité ou une couleur, mais aussi à une opacité. Une étape de segmentation peut être aussi réalisée (de manière semi-automatique). Elle permet la détection de sous-ensembles différents et de leur limites, mais ce n'est pas toujours évident puisque des sous-ensembles différents peuvent avoir des valeurs scalaires ou intensités identiques (en particulier les tissus mous pour les rayons-X)...

Les fonctions de transfert utilisées dépendent du type des données et de l'effet visuel recherché. Par exemple, pour les données volumiques d'un arbre, au moyen d'une fonction de transfert, la couleur des feuilles peut être changée ou carrément les feuilles peuvent devenir totalement transparentes pour ne rendre que le tronc de l'arbre. Le rendu volumique offre beaucoup de possibilités grâce à la prise en compte de la transparence. D'une manière générale, le choix de la fonction de transfert est difficile et souvent contre-intuitif. Heureusement il existe des méthodes de génération automatique de fonctions de transfert [KINDLMANN and DURKIN 1998] qui permettent de visualiser par exemple les frontières entre différents matériels et de générer les normales associées aux voxels. Les normales sont utilisées pour le calcul de l'ombrage (*shading*), c'est-à-dire la prise en compte des différentes sources de lumière pour le rendu final. Elles sont généralement déterminées par le calcul du gradient.

Les 2 principaux domaines d'application du rendu volumique sont la visualisation scientifique (physique, chimie, biologie, etc...) et l'imagerie médicale (tomographie (reconstruction volumique) puis détection de tumeurs, diagnostics, planification d'une opération chirurgicale etc...). D'une manière globale, le rendu volumique est utilisé dans des applications où la navigation interactive est essentielle. C'est pourquoi la communauté scientifique développe des méthodes et outils pour optimiser le rendu volumique. D'un autre côté, chaque domaine a ses spécificités, en visualisation scientifique des gros volumes de données, des champs scalaires, vectoriels, etc... sont manipulés.

Il existe trois principales méthodes pour visualiser les données volumiques:

- **Par extractions de surfaces** (iso-surfaces avec normales): représenter les parties d'intérêt (limites) par des surfaces et rendre ces surfaces (*Marching cubes* [LORENSEN and CLINE 1987],...)
- **Par pixel**: Faire du rendu par pixel en intégrant le long d'un rayon, avec un pas de discrétisation donné, l'intensité ou la couleur associée aux données volumiques (lancer de rayons: *Ray Marching* ou *Ray-Casting*)
- **Par volume**: Faire une projection des cellules volumiques (les voxels) puis une interpolation (*Splatting* ou éclaboussures), faire une projection orthogonale des voxels après déformation (*Shear-Warp*) ou rééchantillonner les données volumiques sous forme de textures 3D par des tranches (*3D Texture-Mapping*) grâce au hardware

Les méthodes par pixel ou par volume sont des méthodes qui nécessitent des intensités avec une opacité ou des données RGBA (*Red* ou Rouge, *Green* ou Vert, *Blue* ou Bleu et A étant l'*alpha-value* ou l'opacité). C'est pourquoi il faut transformer les données initiales à l'aide d'une fonction de transfert, si elles ne sont pas directement accessibles sous la forme RGBA. Notons aussi que lors de l'échantillonnage le long du rayon pour le Ray-Casting, il est préférable d'utiliser une interpolation de qualité (trilinéaire par exemple), car la plupart du temps on ne sera pas sur un point de la grille.

Pour le rendu par surfaces [LORENSEN and CLINE 1987 ; MONTANI et al. 1994], le coût de stockage des polygones peut être plus important que le coût de stockage de la représentation volumique, mais encore la précision de la méthode dépend fortement de la taille d'une cellule de la grille 3D utilisée. En particulier des artefacts visuels peuvent apparaître.

Pour le rendu par pixel dans le cas du Ray-Casting (confère figure 2), la projection 2D du volume coloré semi-transparent est calculée en utilisant l'information sur l'opacité. Des images de type rayon-X ou MIP (*Maximum Intensity Projection* ou Projection de l'Intensité Maximum) peuvent être obtenues lorsque les données se présentent sous la forme de niveaux de gris. Le parcours le long du rayon lancé (depuis la caméra) pour chaque pixel, peut se faire dans le sens derrière-devant (*back-to-front*) ou devant-arrière (*front-to-back*). Le *front-to-back* offre la possibilité de s'arrêter avant de sortir du volume, en utilisant une valeur seuil pour l'opacité (par exemple 95% de l'opacité maximale), mais nécessite la mise à jour en parallèle de l'opacité accumulée, ce qui n'est pas le cas du *back-to-front*. L'opération de mise à jour de la couleur finale s'appelle le *compositing* ou mélange de couleur (confère figure 3).

Initialisation du *compositing* sous la forme RGBA (Rouge, Vert, Bleu et Alpha-value ou opacité):
$C_{dest} = (0.0, 0.0, 0.0, 0.0)$

Le *compositing* dans le cas *back-to-front*:
$C_{dest} = (1- \alpha_{src}).C_{dest} + \alpha_{src}.C_{src}$

Le *compositing* dans le cas *front-to-back*:
$C_{dest} = C_{dest} + (1-\alpha_{dest}).\alpha_{src}.C_{src}$
$\alpha_{dest} = \alpha_{dest} + (1-\alpha_{dest}).\alpha_{src}$

Ici C_{dest} désigne la couleur finale calculée, celle qui se trouvera dans un *color buffer* (tampon mémoire qui contient des couleurs pour afficher l'image correspondante selon la résolution de la fenêtre d'affichage ; ex: *backbuffer* (invisible) ou *frontbuffer* (visible)). C_{src} désigne la couleur actuellement échantillonnée dans les données volumiques (en général, échantillonnage par interpolation trilinéaire). Un *color buffer* invisible, c'est-à-dire qui n'est pas directement affiché dans la fenêtre de l'application, peut être utile pour faire des calculs qui vont intervenir dans le rendu du *frontbuffer*, qui lui sera affiché.

Actuellement des algorithmes performants de *Ray-Casting* sur GPU (Unité de traitement graphique (carte graphique) ou *Graphics Processing Unit*) existent et sont simples à implémenter [KRÜGER and WESTERMANN 2003]. Un GPU est un processeur spécialisé dans les calculs graphiques, il est donc moins flexible qu'un CPU (*Central Processing Unit* ou Unité Centrale de Traitement), mais permet en contre-partie des exécutions en parallèle et l'obtention de

meilleures performances d'affichage (spécialisé pour le traitement des sommets et des pixels).

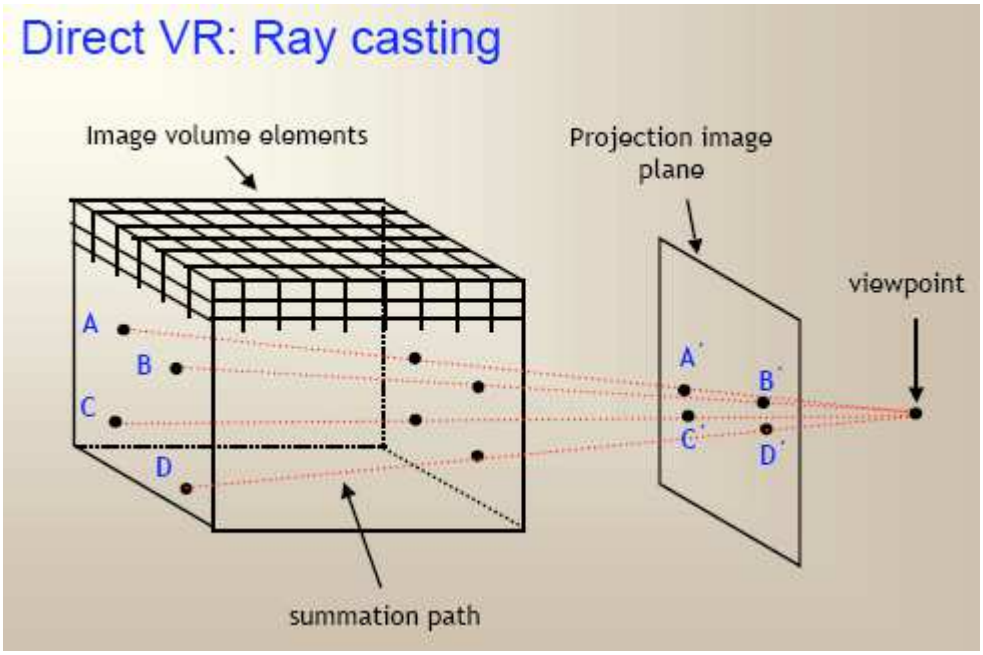


Fig. 2: Rendu volumique direct: Cas du Ray-Casting.

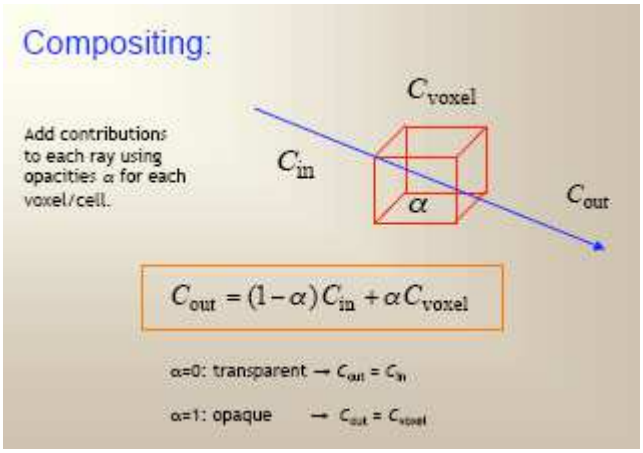


Fig. 3: Ray-Casting: *Compositing* dans le cas du parcours *back-to-front*. Un voxel transparent ne modifie pas la couleur obtenue après sa traversé, tandis qu'un voxel opaque va cacher les voxels précédents.

Pour le rendu par volume dans le cas du *Splatting*, des voxels sous forme de sphère sont projetés sur l'image dans le sens *front-to-back* (et sont mélangés en fonction de la contribution de chaque cellule). Le *Splatting* est plus rapide que le *Ray-Casting*, car les interpolations se font dans l'espace image 2D et pas dans le volume 3D. Dans le cas du *Shear-Warp*, introduit par Cameron et Undrill, et rendu populaire par Philippe Lacroute et Marc Levoy [LACROUTE and LEVOY 1994], les voxels sont déplacés de telle sorte que les rayons soient parallèles aux tranches (pour faire une

projection orthogonale des voxels), et le résultat obtenu est une image déformée (par Ray-Casting). Ensuite cette image est plaquée (*warping*) sur le plan image original, de telle sorte à obtenir le résultat escompté. Ces 2 méthodes sont de qualité inférieure au Ray-Casting usuel. Le terme anglais *Shear* (cisaillement) désigne une transformation qui garde le parallélisme des plans, et qui déplace des plans parallèles dans la même direction (le sens pouvant être différent, en particulier pour les extrémités). Par exemple un rectangle est transformé en parallélogramme. Pour mieux comprendre, voici une illustration de la transformation de type *shear*:

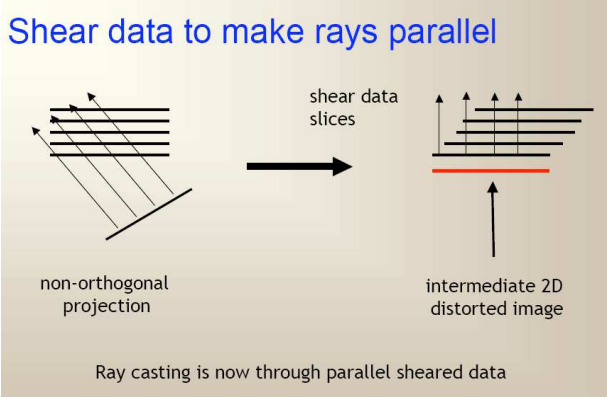


Fig. 4: *Shear*: Transformation des données pour permettre une projection othogonale plus facile à calculer.

La méthode de rendu volumique utilisant les textures 3D et une pile de tranches est détaillée dans une partie I - A - 5, car c'est la méthode utilisée pour rendre nos données volumiques dans le cadre de ce projet.

Maintenant que les méthodes de rendu volumique les plus utilisées ont été présentées, revenons sur le *compositing*. Le mélange des couleurs peut se faire sans prise en compte des sources de lumière et des calculs de réflexion, mais les résultats obtenus seront limités. Dans le cas contraire, avant de faire le *compositing*, on va faire du *shading*, pour prendre en compte la couleur ambiante de la scène, la couleur diffuse et la couleur spéculaire du voxel (en tant que matériel) et des sources de lumières. Cela nécessitera la connaissance de la normale du voxel. Les bases théoriques du *shading* reposent sur sur le VRI (*Volume Rendering Integral*), qui décrit le transport de la lumière au sein d'un volume. Pour plus de détail sur le VRI, j'ai mis un lien sur un site Web en bibliographie (VII). Un tutoriel intéressant sur différents calculs d'illumination pour le rendu volumique [MAX et al. 1995] est aussi cité. Le *shading* peut être pré-calculé et réutilisé par la suite (*pre-shaded volume*).

2 - Récapitulatif sur le rendu volumique

Après avoir fait une introduction sur le rendu volumique, il est important de rappeler les différentes étapes du processus de rendu:

- Correction des données
- Classification
- Segmentation
- *Shading* (ombrage) ou calculs d'illumination

- *Compositing* ou mélange des couleurs
- *Filtering* (filtrage)

Les quatre premières étapes peuvent être pré-calculées une fois pour toute. Rappelons aussi l'utilisation de fonctions de transfert pour la classification et la segmentation. L'estimation des normales des voxels pour le *shading* peut être réalisée à l'aide de filtres spéciaux sur les intensités ou à l'aide d'une fonction de transfert.

Il est courant de mémoriser les données volumiques sous la forme d'une texture 3D, car la carte graphique va nous simplifier la vie.

Des structures de données hiérarchiques (ex: *Octree* [LAMAR et al. 1999]) ou des enveloppes de texture (*texture hulls* [LI and KAUFMAN 2002]) peuvent être utilisées pour déterminer des sous-volumes vides/non-vides au sein des données volumiques initiales (ou des zones de moindre intérêt qui peuvent être rendues avec des résolutions de textures inférieures). Cela permettra de diminuer le nombre de calculs inutiles dans l'étape de *compositing* et le nombre d'accès à la texture 3D (sans oublier l'interpolation trilinéaire), donc cela accélérera le rendu de l'image finale.

3 - Rendu volumique interactif et cadre du projet

Pour le rendu volumique interactif, qui est le cadre de notre projet, les 2 techniques principales sont l'utilisation de textures 3D avec une pile de tranches pour les rééchantillonner et le Ray-Casting. Cela s'explique car ces 2 méthodes utilisent les capacités des cartes graphiques (GPU) actuelles et des algorithmes efficaces utilisant les vertex et pixel *shaders* ont été développés [KRÜGER and WESTERMANN 2003]. En fait les méthodes basées sur CPU ont montré leurs limites quant à traiter de grandes quantités de voxels pour le rendu volumique interactif.

Le majeur problème du rendu volumique, c'est la grande quantité de voxels. Par exemple pour un modèle avec une résolution usuelle de 256x256x256, il y a 16 777 216 voxels. Des optimisations pour obtenir une vitesse d'affichage des images (*framerate*) permettant l'interactivité sont donc plus que nécessaires. Un *framerate per second* ou *fps* est dit interactif s'il est > 15 fps.

Heureusement dans les données volumiques, il y a en général beaucoup de voxels sans intérêts (transparents ou cachés). Par exemple dans nos modèles d'arbre ou de plante, il y a beaucoup de voxels vides autour du tronc ou autour du feuillage (ils représentent entre 50% et 80% du volume initial). Les premières techniques développées pour accélérer le rendu volumique, en particulier dans le cas du Ray-Casting, utilisaient des structures de données hiérarchiques, les *Octrees* ou les *KD-Trees*, pour rapidement traverser les zones vides présentes au sein des données volumiques. Un *Octree* divise chaque volume en 8 sous-volumes de même taille, tandis qu'un *KD-Tree* divise un volume en 2 sous-volumes, mais le plan de découpe peut être choisi librement selon un des trois axes Ox, Oy et Oz.

Avec ces partitions hiérarchiques de l'espace, qui utilisent des plans de découpe parallèles aux axes, la construction peut s'arrêter à partir d'un certain niveau de raffinement, et une information « vide ou ne pas prendre en compte » peut être associée à chaque noeud. Un noeud contient un volume apparaissant sous la forme d'une boite englobante parallèle aux axes, une *AABB*

(*Axis-Aligned Bounding Box*). Actuellement des algorithmes de parcours de ces structures implantés sur GPU existent.

D'autres méthodes existent pour partitionner l'*AABB* des données volumiques initiales, comme des méthodes basées sur des subdivisions uniformes ou sur des algorithmes de regroupement (*clustering*). Une taille minimale pour un sous-volume est en générale fixée. Par exemple les *Growing Boxes* ou boites grossissantes [LI and KAUFMAN 2003 ; LI et al. 2003] est une méthode qui partitionne les données volumiques initiales selon les propriétés des voxels (position et valeur associée par une fonction de transfert (opacité, gradient, etc...)). De telles méthodes produisent en général beaucoup de sous-volumes et il est parfois coûteux de les hiérarchiser (pour faciliter leur parcours) ou de les mettre à jour en cas de découpe d'une partie du volume initial.

D'autres techniques d'accélération, basées cette fois sur la visibilité (opacité accumulée) des voxels ont été développées (*Early Ray Termination* [LEVOY 1990 ; LACROUTE and LEVOY 1994], *Voxel Culling Scheme for Splatting* [MUELLER et al. 1999], carte d'opacité (*Opacity Map*) [LI et al. 2003]). La technique *Early Ray Termination* peut être mise en oeuvre sur GPU en utilisant le *z-test* ou *Depth test* et en modifiant la profondeur à 0 par exemple quand l'opacité accumulée dépasse un certain seuil pour rejeter les futurs fragments et à 1 sinon (l'*Empty-Space Skipping* peut être réalisé de la même façon si on détecte que l'on se trouve dans une zone de vide). Un fragment est un morceau de polygone texturé de la taille d'un pixel. Ici les profondeurs sont normalisées sur [0,1] et donc 0 est la valeur de profondeur du fragment le plus proche et 1 du plus éloigné. J'ai pu lire dans certains articles qu'en moyenne 60% des voxels non-vides ne sont pas utilisés pour le rendu de l'image finale, et donc la vitesse de rendu peut être augmentée de manière considérable avec de telles techniques.

Les techniques d'accélération qui évitent de prendre en compte des sous-volumes vides (*Empty-Space Skipping*) et celles qui tiennent compte de l'opacité accumulée pour ne pas rendre des voxels cachés (*Occlusion Clipping*) sont complémentaires. Le premier type d'accélération permettra un gain considérable lorsqu'il y aura beaucoup de voxels vides « groupables » et le second genre lorsqu'il y aura beaucoup de voxels non-vides. Ainsi en utilisant les 2 sortes de techniques, on pourra accélérer le rendu de la plupart des données volumiques. Ici j'utilise le terme « groupable » pour dire qu'il doit y avoir plusieurs voxels vides disposés les uns à la suite des autres et que des sous-ensembles de ces voxels peuvent être délimités par une *AABB*. Il serait inutile de d'éviter seulement un ou 2 voxels vides isolés. Si les données volumiques ne changent pas au cours de leur visualisation, les structures de données hiérarchiques pour sauter les espaces vides peuvent être déterminées statiquement (pré-calculs), ce qui permettra l'utilisation de méthodes plus complexes pour les construire. Sinon il faudrait utiliser des structures plus adaptées à la mise à jour dynamique lorsque ces changements ne sont pas trop brusques (par exemple une hiérarchie de boites englobantes ou *Bounding Volume Hierarchy*). Par contre les accélérations dues à la visibilité nécessiteront toujours des mises à jour dynamiques en fonction des mouvements de la caméra.

Le cadre de ce projet est l'utilisation d'un petit nombre d'*AABBs* sur un modèle volumique d'arbre ou de plante pour optimiser son rendu dans le cas de la technique basée sur les textures 3D « tranchées ». L'optimisation ne concernera que la suppression des voxels vides, car l'objectif est d'étudier le gain en vitesse d'affichage si on ne prend pas en compte ces derniers (qui devrait être important vu qu'il y a plus d'espace vide que d'espace non-vide pour de tels modèles).

Dans le cadre d'une approche qui manipule plusieurs sous-volumes, le nombre d'*AABBs* est en effet critique car:

- un tri d'*AABBs* est effectué pour afficher les sous-volumes dans le bon ordre (de plus amples détails seront donnés par la suite)
- un surcoût est lié aux structures de données utilisées pour manipuler les sous-volumes (en particulier pour leur affichage)
- chaque sous-volume est rendu avec la technique des textures 3D tranchées, ce qui nécessite l'initialisation et le chargement de la texture dans une zone mémoire dédiée (sur GPU), mais encore des calculs d'intersections entre les tranches et les *AABBs* des sous-volumes doivent réalisés pour déterminer les bonnes coordonnées de textures (et de manière dynamique lorsque les tranches sont choisies orthogonales à l'axe de vue)

C'est pourquoi notre approche sur le problème de la partition hiérarchique des voxels non-vides est assez différente des approches existantes [BOADA et al. 2001 ; LAMAR et al. 1999 ; LI et al. 2003] qui ne se soucient pas vraiment du nombre total de feuilles obtenu pour une structure hiérarchique donnée.

4 - Textures 3D et termes associés aux textures

Une texture 3D peut être vue comme un tableau 3D, dont l'accès à un élément est possible avec des coordonnées réelles, en particulier avec u, v et w sur [0,1]. L'élément de base d'une texture 3D est un texel (*texture element*) ou voxel. Le terme texel peut être aussi utilisé pour des textures 2D, tandis que le terme voxel est spécifique aux textures 3D. Lorsque les données sont associées à des intensités ou des couleurs ainsi qu'à une certaine opacité (directement ou après traitement), une texture 3D peut s'avérer être un bon outil pour stocker les données volumiques, surtout si une carte graphique qui supporte les textures 3D est à notre disposition (sinon il faudrait utiliser une pile de textures 2D selon les 3 axes).

Le terme *MIP-maps* (*Multi In Parvo)* ou *mipmaps* désigne une série de textures de résolutions décroissantes qui seront utilisées à la place de la texture originale, selon la distance du point de vue de l'objet texturé et le niveau de détails nécessaire. La texture utilisée lors du rendu dépendra du nombre de texels qui correspondent à un pixel sur l'image finale. En utilisant des résolutions obtenues par moyennage, cela permet de nuancer la perte d'information et de diminuer l'effet d'aliassage (*aliasing* ou effet d'escalier). Plusieurs niveaux de *MIP-map* peuvent être créés pour une texture 3D, mais le nombre de niveaux dépend de la résolution initiale en voxels de la texture 3D ainsi que de la méthode de filtrage utilisée.

Le filtrage de texture est une méthode utilisée pour déterminer la couleur à afficher dans un pixel lorsqu'une texture lui est associée. Plus un filtrage « lisse » le résultat dans le sens où il diminue les artefacts visuels, plus il sera coûteux à réaliser. En fait la carte graphique cherche la position du centre du pixel dans la texture, ainsi le pixel peut être associé à un groupe de texels. Lorsqu'un texel est projeté sur une image, en général il est plus petit ou plus gros qu'un pixel. Supposons qu'étant donné la distance de l'objet texturé et le pixel regardé, la projection du texel corresponde exactement au pixel. Si l'objet se rapproche, le texel va devenir plus gros que le pixel. Il y aura donc plusieurs pixels associés à ce texel. C'est ce qui s'appelle la *magnification* ou agrandissement. En fonction du type de filtrage utilisé l'image peut être « joufflue » (*chunky*). La *magnification* concerne plutôt les objets ou parties d'objets proches de la caméra. Si l'objet

s'éloigne, le texel va devenir plus petit que le pixel. Il y aura donc plusieurs texels associés à ce pixel. C'est ce qui s'appelle la *minification* ou rétrécissement. En fonction du type de filtre utilisé pour traiter cela, l'image résultante peut être aliasée ou floue (pour les pixels concernés). La *minification* concerne les objets ou parties d'objets éloignés de la caméra et les angles d'incidence rasante (*grazing angle*). Finalement en utilisant plusieurs niveaux de *MIP-map*, une partie de la *minification* peut-être déjà réalisée.

Plusieurs filtres existent pour traiter ces problèmes. Les 3 principaux sont le texel le plus proche du centre du pixel (*nearest_neighbor* dans les API graphiques), le filtrage bilinéaire (en 2D) ou trilinéaire (en 3D) (*linear* dans les API graphiques) qui est en fait la moyenne des 4 texels (en 2D) ou des 8 texels (en 3D) les plus proches du centre du pixel (interpolation), et le filtrage anisotrope. Le filtrage anisotrope est une alternative au filtrage bilinéaire ou trilinéaire qui préserve plus les détails (moins de flou ou *blur*), mais qui est beaucoup plus coûteux en nombre de calculs (surtout dans le cas des textures 3D). Le filtre *nearest_neighbor* donne une image avec des petits blocs (*blockiness*) pour la *magnification* et une image aliasée pour la *minification*. Le résultat peut être amélioré en utilisant ce filtre avec du mipmapping (car le niveau de *MIP-map* le plus adapté sera choisi).

Dans le cadre de ce projet nous travaillons avec des textures 3D (de plantes ou d'arbres) avec des voxels pré-éclairés. Le *shading* est effectué en pré-calcul et est stocké dans les canaux RGB de la texture. L'opacité est stockée dans le canal Alpha.

5 - Rendu Volumique utilisant des « tranches » de textures 3D

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

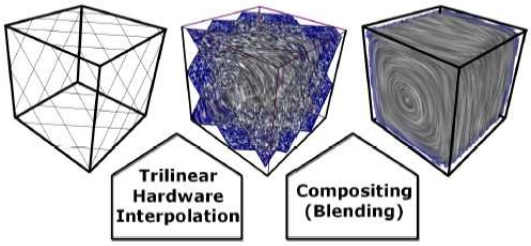


Fig. 5: Illustration du rendu volumique à base d'une texture 3D échantillonnée (interpolation trilinéaire selon des plans parallèles au plan d'observation) [KRÜGER and WESTERMANN 2003].

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

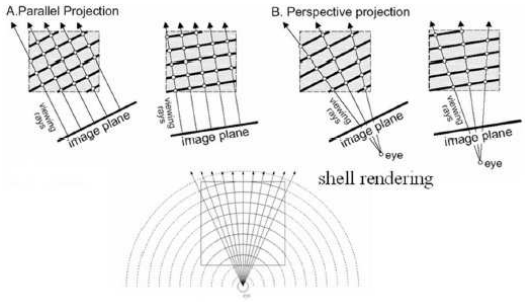


Fig. 6: Utilisation des textures 3D: on peut faire une projection orthogonale ou en perspective [KRÜGER and WESTERMANN 2003].

6 - Voxelisation

La voxelisation est une technique de conversion d'une représentation géométrique (courbe dans l'espace, polygone, maillage texturé, ...) en une représentation volumique à base de voxels (grille 3D). Un pas de discrétisation est fixé selon les 3 directions pour mettre en place le processus d'échantillonnage. Le problème est de représenter des formes géométriques avec suffisamment de voxels (séparabilité) pour distinguer les limites de chaque forme, mais aussi de ne pas en utiliser trop par rapport à la qualité fixée (minimalité). Pour des détails techniques, veuillez regarder [COHEN-OR and KAUFMAN 1995 ; HUANG et al. 1998].

La voxelisation permet de manipuler des données volumiques par la suite (plus de maillage), qui ont une complexité liée à la résolution de la grille 3D. Elle permettra de rendre plus facile l'intégration d'autres données volumiques (par mélanges ou ajouts), et de choisir plus

facilement un sous-ensemble de voxels à afficher via une fonction de transfert. Cependant à cause de l'échantillonnage, des informations seront perdues et la représentation volumique sera moins précise (moins de détails) que la représentation utilisant un maillage 3D texturé. Les problèmes liés à la visibilité sont traités de manière différente pour une représentation volumique, car la visibilité des voxels à l'arrière ne dépend plus des faces qui cachent les autres (*culling*), mais de l'opacité des voxels situés devant...

B - Pourquoi utiliser du rendu volumique?

En parlant de voxelisation dans la partie précédente, nous avons présenté certaines possibilités intéressantes pour le rendu volumique, comme la modification des données initiales par une fonction de transfert. Maintenant nous allons expliquer pourquoi nous utilisons des données volumiques de plantes et d'arbres dans ce projet sous forme de textures 3D (obtenues par voxelisation de maillage 3D).

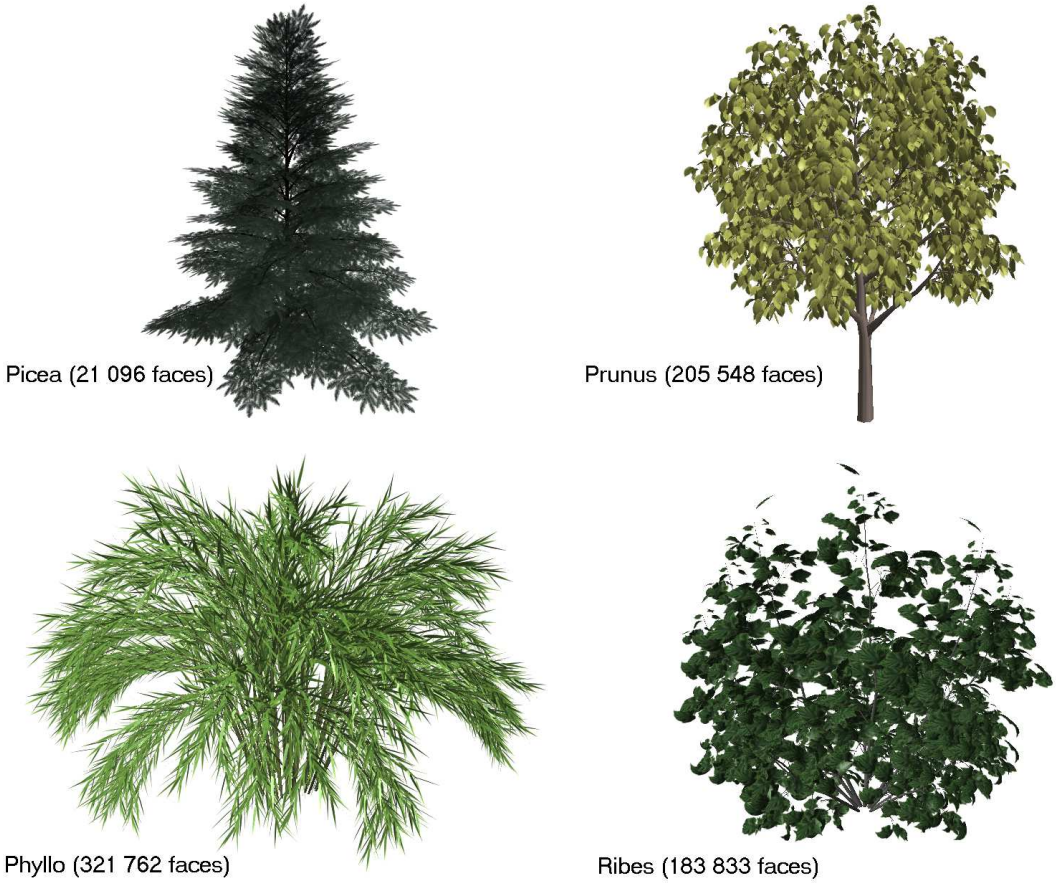


Fig. 7: Les maillages de d'arbres ou de plantes utilisés présentent parfois plusieurs centaines de milliers de faces.

Les données initiales sont des maillages 3D d'arbres et de plantes obtenus à partir de modèles statistiques sur leur croissance. Ces modèles sophistiqués permettent de créer des maillages avec beaucoup de détails, pour rester le plus fidèle possible à la forme générale d'une plante, mais aussi à la forme des ses feuilles. La figure 7 donne un aperçu de quelques maillages. Dans notre

ensemble de départ, il y a plus de 400 000 faces triangulaires en moyenne par maillage 3D d'arbre ou de plante, donc la complexité géométrique n'est pas négligeable.

Nous créons des textures 3D de ces maillages par voxelisation (discrétisation), le plus souvent avec une résolution 256^3. Ainsi, pour une direction de vue selon un des 3 axes, au maximum 256 tranches de textures (niveau de *mipmap* 0) pourront être utilisées pour rééchantillonner la texture 3D initiale. Un plan nécessitant 2 faces triangulaires, cela fait 256 x 2 = 512 faces triangulaires à gérer au lieu des 400 000 en moyenne. La complexité géométrique n'est donc pas comparable (400 000/512 = 781,25)... Les maillages présentent plus de détails que les représentations volumiques et donc pour une représentation d'un arbre ou d'une plante sur plusieurs niveaux de détails, ils seraient utilisables pour le premier niveau (proximité de la caméra). Ensuite pour un niveau de détails intermédiaire ou éloigné, une représentation volumique aurait une complexité visuelle équivalente et permettrait en général un gain conséquent en *framerate*, car sa complexité géométrique est moindre et parce que le filtrage est géré par le GPU. Pour les niveaux de détails les plus éloignés, un imposteur ou *billboard* (un plan texturé) pourrait aussi être utilisé.

En utilisant une représentation volumique avec plusieurs niveaux de *MIP-map*, plus le modèle sera éloigné de la caméra, plus le niveau de *MIP-map* utilisé sera élevé, donc moins de tranches texturées seront utilisées (en général le nombre de tranches est une puissance de 2, donc chaque fois que le niveau de *mipmap* sélectionné augmente, la complexité géométrique est divisée par 2). C'est pourquoi la complexité géométrique de la représentation volumique devient très intéressante pour des positions relativement éloignée de la caméra, et cela en conservant suffisamment de détails (complexité visuelle équivalente). Un autre avantage du rendu volumique basé sur une texture 3D avec différents niveaux de *MIP-map*, est la diminution de l'effet d'aliasing (scintillement) en utilisant du filtrage trilinéaire par exemple. Mais cela nécessite la mise en mémoire des différents niveaux de *MIP-map*. Il serait donc intéressant d'étudier les techniques existantes pour compresser les données volumiques, mais ce n'est pas l'objectif de notre projet.

Le fait qu'un détail ne soit pas plat n'implique pas nécessairement qu'il doit être représenté par un maillage 3D complet et trop coûteux pour faire du rendu interactif lorsque plusieurs objets sont nécessaires (arbres et forêt) [MEYER and NEYRET 1998]. L'impression de 3D est une notion progressive qui dépend de plusieurs paramètres, dont l'apparence locale et les contours associés à l'axe de vue, les mouvements de parallaxe, l'occlusion etc..., et elle peut être aussi obtenue à l'aide de textures 3D.

Les textures volumiques permettent donc de garder l'impression de 3D, et cela à un coup faible, pour une qualité de rendu tout à fait acceptable grâce au filtrage géré facilement par le GPU (pas d'*aliasing*). En plus, il n'y a pas les effets de *popping* (apparition soudaine) des imposteurs, en particulier des modèles volumiques peuvent être disposés les uns à côté des autres, sans que l'un d'entre eux apparaisse soudainement. La complexité visuelle est meilleure que celle des modèles texturés simplifiés et il n'y a pas le problème de la transformation géométrique à appliquer à tous les points d'une représentation basée sur des points.

Dans le cadre du rendu interactif de forêt, beaucoup d'arbres sont situés relativement loin de la caméra et il y a beaucoup trop de feuillage pour espérer un rendu temps-réel direct des feuilles. Dans ce cas des représentations volumiques d'arbre peuvent être utilisées et permettront de conserver les effets d'impression 3D. Il y a une certaine répétitivité dans les forêt et donc un petit

ensemble de modèles d'arbre suffit pour faire du rendu réaliste, mais la disposition des arbres doit apparaître naturelle, en particulier il ne faut pas que l'oeil repère des règles d'agencement. Des techniques ont été développées pour faire du *mapping* apériodique de textures volumiques sur des terrains 3D, pour éviter que la répétition des motifs soit visible dans le cadre du rendu de forêt dense temps-réel [DECAUDIN and NEYRET 2004].

Fig. 8: Rendu interactif de forêt dense utilisant des textures volumiques et un *mapping* apériodique [DECAUDIN and NEYRET 2004].

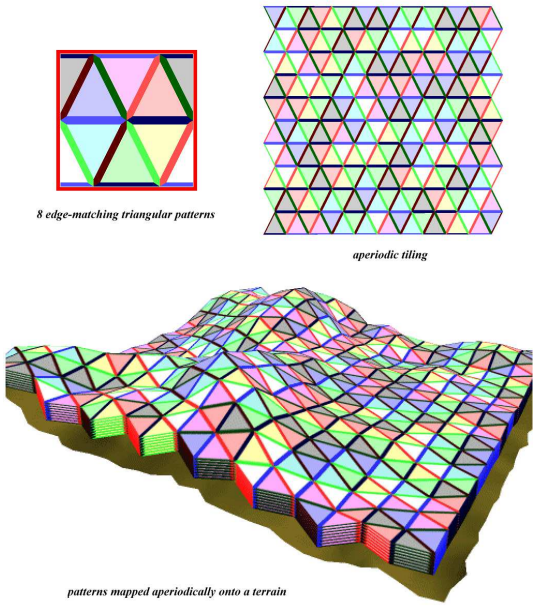


Fig. 9: *Mapping* apériodique sur un terrain [DECAUDIN and NEYRET 2004].

C - Les KD-Trees

Les *KD-Trees* ou *K-Dimentional Trees* sont des *BSP Trees* orthogonaux qui manipulent des données dans un espace de K dimensions. *BSP* signifie *Binary Space Partitioning* ou encore partitionnement binaire de l'espace et *Tree* signifie arbre. Les *BSP Trees* et les *KD-Trees* sont des

structures de données hiérarchiques qui permettent de partitionner l'espace en sous-volumes, l'*AABB* initiale étant coupée en 2 et ainsi de suite jusqu'à ce qu'un critère d'arrêt soit atteint pour un noeud de l'arbre donné. La particularité des *KD-Trees* réside dans le fait que les plans de découpe doivent être choisis parallèles aux axes (ou le long des axes) et donc chaque noeud d'un *KD-Tree* est associé à une boite dont les côtés sont parallèles aux axes.

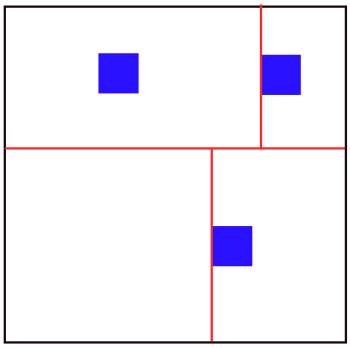


Fig. 10: Illustration 2D d'un partitionnement de l'espace obtenu à l'aide d'un *KD-Tree*.

Dans l'illustration ci-dessus, les plans de découpe sont en rouge, le carré noir représente la boite associée à la racine du *KD-Tree*, et les carrés bleus des primitives géométriques. On peut remarquer qu'un plan de découpe peut être choisi selon n'importe quel axe et qu'un plan de découpe n'est pas obligatoirement contre une primitive géométrique ou contre une boite englobant des primitives géométriques. La boite associée à un noeud n'est pas forcément une *AABB*, dans le sens où il est possible (comme dans notre exemple) de rétrécir cette dernière pour qu'elle soit minimale (et encore englobante). Des noeuds vides peuvent être obtenus, et en général un noeud vide n'est plus subdivisé (c'est un critère d'arrêt).

Pour choisir le plan de découpe d'un noeud d'un *KD-Tree*, une fonction de coût est habituellement utilisée. Elle va associer une valeur à chaque plan candidat pour permettre de trouver le meilleur. La recherche du meilleur plan se fait dans la plupart des cas en choisissant le minimum de la fonction de coût. La qualité du *KD-Tree* construit (en fonction du problème donné) repose sur la place du plan de découpe pour chaque noeud de l'arbre et sur les critères d'arrêt. Les *KD-Trees* sont souvent utilisés pour partitionner l'espace de manière à déterminer les espaces vides et les espaces non-vides. Pour améliorer la qualité du *KD-Tree* en diminuant la profondeur de certaines de ses branches, la fonction de coût utilisée peut être légèrement biaisée, de manière à favoriser un plan de découpe qui engendrerait un fils vide, mais qui ne serait pas trop éloigné du plan optimal au sens du minimum de la fonction de coût non biaisée.

Les *KD-Trees* sont très utilisés pour le *Ray-Tracing* [HAVRAN 2001 ; HAVRAN 2002 ; HURLEY et al. 2002 ; STOLL 2005 ; WALD et al. 2001 ; WALD 2004]. Dans ce cas précis, il y a beaucoup de calculs d'intersection entre les rayons lancés et les primitives géométriques de la scène. C'est pourquoi la fonction de coût utilisée, l'heuristique SAH (*Surface Area Heuristic*) [MACDONALD and BOOTH 1990], tend à minimiser le nombre de calculs d'intersection, en se basant sur une approche probabiliste. Une heuristique peut être vue comme une méthode intuitive (ou une recette de cuisine) qui favorise ou accélère l'obtention de résultats sans garantie sur leur appartenance à l'ensemble des solutions optimales. Cela est intéressant lorsqu'une solution optimale

nécessiterait des heures de calculs ou si elle est très difficile à résoudre (problème NP-complet). Dans le cas précédent, l'heuristique SAH accélère le rendu final de l'image, car elle permet de diminuer le nombre de calculs d'intersection inutiles. Pour améliorer la qualité du *KD-Tree* construit, l'heuristique SAH est généralement biaisée par un facteur de 0,80 ou 0,85 lorsqu'un des 2 fils est vide pour un plan de découpe donné.

D - Problématique

Dans le cadre du rendu volumique interactif, la problématique de notre projet est l'optimisation du rendu volumique de modèles volumiques de plante ou d'arbre. Il y a beaucoup de voxels vides dans ces modèles (plus de 50% pour tous les modèles utilisés), en particulier autour du tronc et du feuillage. Cette optimisation est orientée vers du rendu volumique basé sur des textures 3D « tranchées ». Il faut chercher à diminuer le volume vide total de la représentation de base en segmentant la boite englobante initiale (de type *AABB*) en plusieurs sous-boites.

Pour faire cela, nous utiliserons une structure hiérarchique, un *KD-Tree*, qui nous permettra de faire une partition de l'ensemble des voxels non-vides des données volumiques initiales. Les bibliothèques actuelles qui permettent de faire du rendu basé sur une texture 3D, ne sont pas encore optimisées pour la gestion de plusieurs sous-volumes. Donc un surcoût est envisageable pour la manipulation de plusieurs sous-volumes. Le travail est donc orienté vers une solution utilisant peu de sous-volumes, tout en supprimant une large quantité d'espace vide. Il faudra donc déterminer un ensemble de critères pour le choix des plans de découpe et pour ne pas continuer la découpe si ça « n'est plus intéressant ». Remarquons que la contrainte « d'avoir peu de sous-volumes » a mené à préférer la structure de *KD-Tree* à celle d'*Octree*, car pour un *KD-Tree*, le choix du plan de découpe peut être orienté pour minimiser le nombre total de sous-volumes utilisés.

Une fois que ce *KD-Tree* sera construit, ce dernier sera utilisé pour choisir un sous-ensemble de volumes (des *AABBs*) qui seront affichés à l'aide de la technique du rendu volumique basé sur des textures 3D.

E - Objectifs

1 - Trouver les bons critères d'arrêt pour la construction du KD-Tree

Le premier objectif du stage est de trouver des bons critères d'arrêt pour la construction du *KD-Tree* associé à un maillage 3D d'arbre ou de plante. L'objectif étant d'avoir un nombre peu élevé de boites pour que le rendu des sous-volumes soit plus intéressant que le rendu initial du volume associé au maillage du végétal. La manière de construire les noeuds du *KD-Tree* pourra être adaptée au problème étudié.

2 - Gestion de plusieurs niveaux de subdivision

Pour cette étude, nous nous restreignons à 3 niveaux de subdivisions, car l'objectif principal est de voir si notre technique est utilisable, et non de déterminer le nombre optimal de niveaux de subdivision pour chaque modèle.

Le second objectif du stage est donc de déterminer trois niveaux de subdivision pour un modèle donné. Le niveau 0 sera le volume de base (intéressant pour les arbres ou plantes éloignés) et les 2 autres niveaux restent à déterminer.

Il faut aussi s'occuper du problème des transitions entre 2 niveaux de subdivision si nécessaire. A priori il n'y aura pas de problème car le volume rendu visible sera le même, seul le nombre de sous-volumes changera. Mais il est possible d'avoir certains artefacts visuels, en particulier à cause de la transparence.

Il faut résoudre le problème du passage automatique d'un niveau de subdivision à un autre afin de maximiser le *framerate* par seconde.

Remarque: Ici on parle de niveaux de subdivision et non de niveaux de détail, car les détails des représentations volumiques ne changent pas à cause du nombre de sous-volumes utilisés, mais à cause du niveau de *MIP-map* utilisé.

II - Choix effectués pour la construction du KD-Tree et pour la génération des 3 niveaux de subdivision

La solution finale proposée repose sur la construction d'un *KD-Tree* pour un modèle d'arbre ou de plante donné. En particulier nous avons fait des expériences et des choix pour la génération des plans de découpe candidats, pour le choix de la fonction de coût utilisée et pour diminuer le nombre total de ses feuilles. Ensuite, le *KD-Tree* est utilisé afin de générer 3 listes de sous-volumes, qui correspondront à 3 niveaux de subdivision.

Dans ce qui suit, je présente donc les fonctions de coût utilisées, les méthodes de génération des plans candidats et les optimisations et critères d'arrêt pour la construction du *KD-Tree*. Ensuite j'aborde la génération des 3 niveaux de subdivisions ainsi que les différents problèmes liés à leur affichage.

Il est important de signaler que j'ai réalisé mon travail à partir de maillages 3D de plantes et d'arbres (confère figure 7) au format DirectX *.x*. En particulier la construction des *KD-Trees* a été réalisée à partir de maillages 3D afin d'optimiser le rendu des données volumiques associées au maillage. Cependant elle peut s'étendre facilement aux représentations volumiques si on possède l'information sur la transparence (ou l'opacité) des voxels.

J'ai réalisé ce projet sous Windows XP, j'ai programmé en C++ avec l'API graphique Microsoft DirectX 9 sous Microsoft Visual C++ 2003 et 2005.

A - Construction du KD-Tree: Sélection d'un plan de découpe

La sélection d'un plan de découpe, nécessite dans un premier temps de générer des plans candidats, puis dans un second temps de choisir le meilleur en prenant le minimum d'une fonction de coût. La qualité d'une partition repose essentiellement sur le choix d'une bonne fonction de coût, car même si la génération des plans candidats était supposée idéale, une mauvaise fonction de coût engendrerait une partition inutilisable. C'est pourquoi nous nous sommes intéressés en premier au choix de la fonction de coût (en générant simplement une liste de plans candidats uniformes pour les 3 axes).

1 - Fonctions de coût

i - Heuristique SAH (Surface Area Heuristic)

Au cours de mon étude bibliographique, j'ai pu voir que cette heuristique était la plus utilisée pour la partition spatiale basée sur les *KD-Trees* dans le cas du Ray-Tracing [MACDONALD and BOOTH 1990 ; STOLL 2005]. En fait c'est une approche probabiliste qui

regarde la probabilité qu'un rayon intersecte un fils sachant que le volume père a été touché (basé sur l'hypothèse de la distribution uniforme des rayons). L'heuristique SAH prend en compte un coût estimé pour la découpe du noeud en cours, avec un coût de traversé du noeud et un coût d'intersection (approximé par le nombre de primitives se trouvant dans un fils). L'objectif dans le cas du Ray-Tracing est d'obtenir un *KD-Tree* qui minimise le nombre de calculs d'intersection. C'est pourquoi il faut choisir des plans qui vont séparer des espaces restreints chargés en primitives des autres zones du volume initial (pour avoir moins de calculs d'intersection inutiles dans des zones qui ont plus de probabilité d'être touchées).

Il est clair que notre problème est de nature différente, car on cherche des plans de découpe qui permettent de supprimer un maximum d'espace vide avec peu de sous-volumes. C'est pourquoi une partition obtenue avec l'heuristique SAH ne sera pas forcement optimale pour minimiser le volume total à rendre avec un nombre relativement petits de sous-volumes. Après une série de tests sur l'ensemble des modèles d'arbre et de plante à disposition, je me suis rendu compte qu'elle n'était pas adaptée à la résolution de notre problème. L'heuristique SAH essaie d'isoler les parties les plus « chargées » en primitives géométriques (faces...) du volume conteneur. Cela provient de sa fonction de coût:

```
SAH(N,p) = TC + ( NB_L(N,p) * SA_L(N,p) + NB_R(N,p) * SA_R(N,p) )/SA(N)
```

N est un noeud associé à un volume englobant (*AABB*) ; p = plan de découpe ; TC = *Transversal Cost* ou coût de traversée ;
NB_L / NB_R = Nombre de primitives à gauche / droite du plan de découpe ;
SA = demi-aire du volume initial ;
SA_L / SA_R = demi-aire du volume initial à gauche / droite du plan de découpe

La demi-aire ou l'aire peuvent être utilisée de manière indifférente pour la fonction de coût. Cette dernière va donner un coût minimal pour un plan de découpe qui permet de mettre un maximum de primitives sur une partie relativement petite du volume initial.

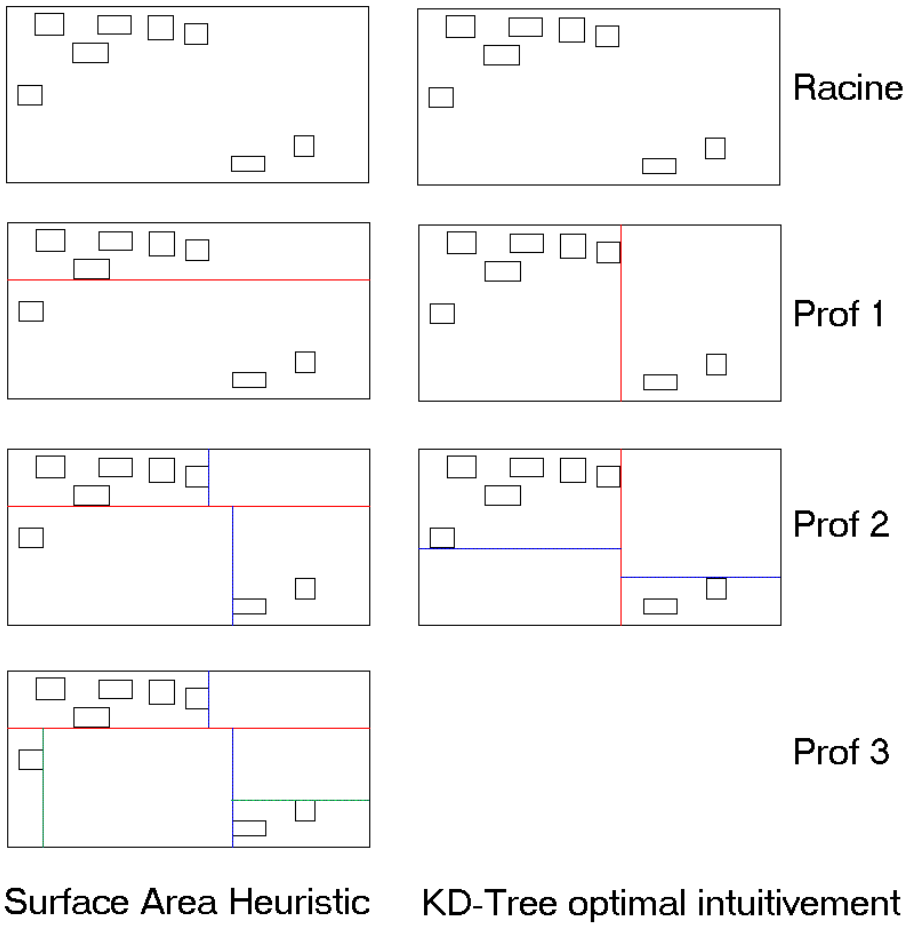


Fig. 11: Illustrations 2D montrant qu'un *KD-Tree* élémentaire construit avec l'heuristique SAH n'est pas optimal en nombre de feuilles non-vides.

La figure 11 illustre le fait que l'heuristique SAH tend à choisir un plan de découpe qui permet d'isoler relativement beaucoup de primitives sur une zone d'aire petite, sans tenir compte des primitives géométriques voisines assez proches, ce qui tend à générer plus de sous-volumes qu'une solution intuitivement optimale pour notre problème donné. Dans l'exemple, une seule primitive géométrique « sort » du groupe de primitives en haut à gauche, et cela va nous conduire à un *KD-Tree* construit avec SAH qui possédera 6 feuilles, dont 3 non-vides, tandis que la solution optimale n'en possède que 4 dont 2 non-vides (et cela pour un espace vide retiré équivalent).

Pour notre problème de base, nous n'avons finalement pas besoin des feuilles vides du *KD-Tree* car nous allons seulement rendre les sous-volumes non-vides. J'ai donc décidé de ne plus autoriser de feuilles vides au sein du *KD-Tree*, en calculant l'*AABB* de chaque fils après une découpe et en utilisant cette dernière pour la découpe suivante. L'*AABB* exacte des données volumiques est calculée avant de commencer la construction de l'arbre. Cette manière de construire le *KD-Tree* permet de diminuer sa profondeur, et nous donne une hiérarchie de boites englobantes, qui épouse la forme des données de plus en plus au fur et à mesure que l'on avance dans l'arbre (confère figure 12). Cela nous sera utile pour le choix des sous-volumes à visualiser par la suite. A partir de maintenant tous les noeuds d'un *KD-Tree* que l'on construit contiendront leur *AABB* exacte.

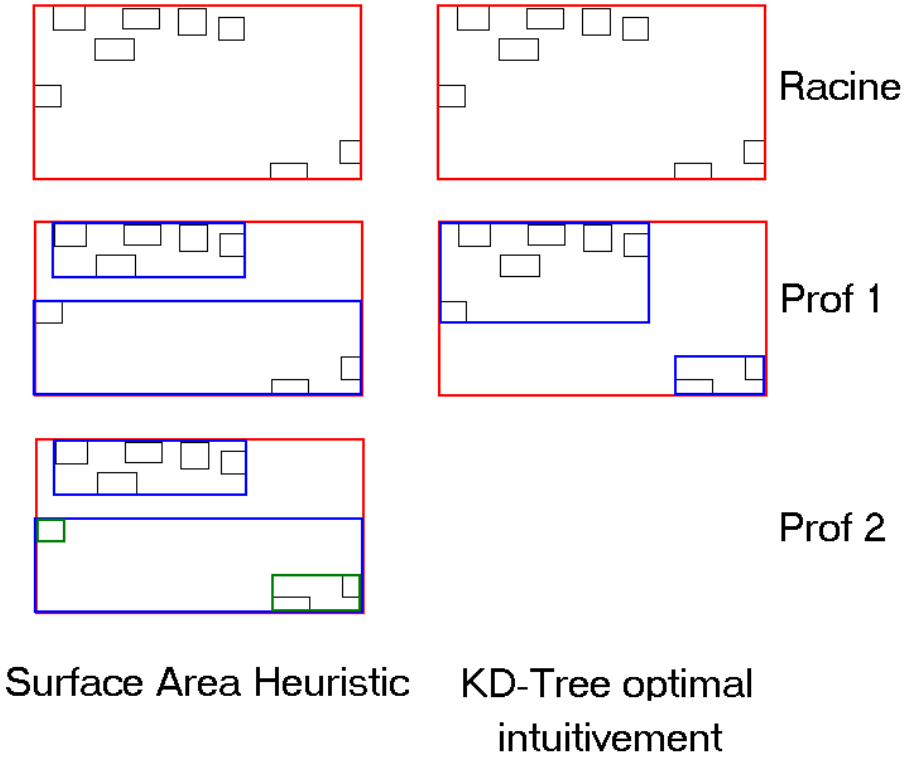


Fig. 12: Illustrations 2D montrant qu'un *KD-Tree* élémentaire construit avec l'heuristique SAH avec l'ajustement des *AABBs* des fils n'est pas optimal en nombre de feuilles non-vides.
Les plans de découpe n'ont pas été représentés pour une question de visibilité.

J'ai abandonné l'heuristique SAH (sans et avec l'ajustement des *AABBs*), car elle générait trop de sous-volumes. Elle posait vraiment un problème pour notre objectif d'une perte importante de volume vide avec un petit nombre de boites. Par exemple pour le cas d'un modèle 3D d'arbre pour lequel le tronc est visible (prunier), l'heuristique SAH ne permet pas d'isoler avec un seul plan de découpe le tronc des feuilles. Cela est du au fait que le feuillage est beaucoup plus chargé en primitives que le tronc, et qu'un petit nombre de feuilles sort de l'alignement global du feuillage. Ce qui va engendrer beaucoup de sous-volumes (une quarantaine) pour arriver à enlever suffisamment de vide (au moins 40 % du volume total) lorsque les critères d'arrêts sont « peu contraignants » ou qui ne permettra pas d'enlever suffisamment de vide.

De l'échec de l'heuristique SAH (sans et avec ajustement de l'*AABB* des fils), j'ai compris 2 choses importantes:

- Si le nombre de primitives influence le plan de découpe, cela peut nous empêcher d'obtenir un gain volumique maximal avec un nombre de boites relativement petit. En fait, il n'y a pas de rapport entre le volume de l'*AABB* d'un ensemble de primitives géométriques et le nombre de primitives que l'*AABB* contient. Il ne faut donc pas prendre en compte le nombre de primitives dans notre fonction de coût pour notre problème.
- La fonction de coût doit permettre d'obtenir une perte maximale en volume vide, c'est-à-dire permettre de choisir un plan de découpe qui minimise le volume total à rendre.

Remarque: J'ai aussi fait des tests avec des heuristiques complémentaires. Je me suis en partie inspiré de mes lectures bibliographiques. Dans [HUNT et al. 2006 ; WALD and HAVRAN 2006], l'heuristique SAH est biaisée (x 0.85 ou 0.80) pour favoriser des plans de découpe qui permettent de supprimer du vide (cas où un des 2 fils est vide). Mais même si les résultats observés sont meilleurs (on perd du volume vide plus rapidement), certaines configurations comme celle présentée à la figure 11 ne peuvent pas être réglées, le nombre de découpes successives pour isoler les parties non vides est encore trop important sur nos modèles d'arbres ou de plantes. Ce qui a 2 conséquences fâcheuses. D'abord une floraison de petits sous-volumes non vides pour les feuilles isolées peut s'observer. Ensuite ces sous-volumes sont trop petits et souvent dissymétriques.

Cela montre, qu'il est inutile d'essayer d'améliorer une heuristique de base si cette dernière n'est pas adaptée au problème étudié. Nous pouvons aussi remarquer qu'il n'est pas la peine dans notre cas de spécifier un coup de traversée dans notre fonction de coût, car par la suite le *KD-Tree* ne sera plus parcouru, puisque des volumes associés à un niveau de subdivision fixé seront affichés.

ii - Heuristique VVH (Voxel Volume Heuristic)

Cette heuristique est une version approximée de la SAH heuristique. Le ratio SA(N_enfant)/SA(N_parent) est approximé par VOL(N_enfant)/VOL(N_parent). Au lieu d'utiliser la demi-aire d'un volume, le volume (scalaire) est utilisé. Les résultats obtenus sont similaires à ceux de SAH.

Nous avons donc abandonné cette heuristique.

iii - Le gain volumique relatif maximal d'un fils

Cette heuristique consiste à choisir le plan de découpe qui permet d'avoir le plus grand gain volumique relatif (la plus grosse suppression de vide), en ne considérant que le gain volumique relatif maximal parmi les 2 fils (ou le volume relatif à rendre minimal). Après chaque découpe les *AABBs* des 2 nouveaux fils sont ajustées pour obtenir une hiérarchie de boites englobantes qui progressivement va épouser la forme des données.

Puisque les *AABBs* sont ajustées après chaque découpe et qu'il y a relativement peu de vide à l'intérieur d'un modèle d'arbre ou de plante, pour un noeud et un plan de découpe donnés, le plus souvent une seule des 2 nouvelles *AABBs* filles sera inférieure à la partie de l'*AABB* mère correspondante (confère figure 13). Cela nous a amené à ne considérer que le gain relatif maximal des 2 fils.

Voici la fonction de coût utilisée (*Volumetric Gain*):

```
VG(N,p)  =  min( BV(N_L) / BV_L(N,p), BV(N_R) / BV_R(N,p) )
```

BV = volume englobant (scalaire) de l'*AABB* du noeud ; N = noeud ; p = plan de découpe ;
_L = à gauche du plan ; _R = à droite du plan ; N_L et N_R sont déterminés avec N et le plan de

découpe (en associant à N_L/N_R toutes les primitives géométriques de N à gauche/droite du plan de découpe)

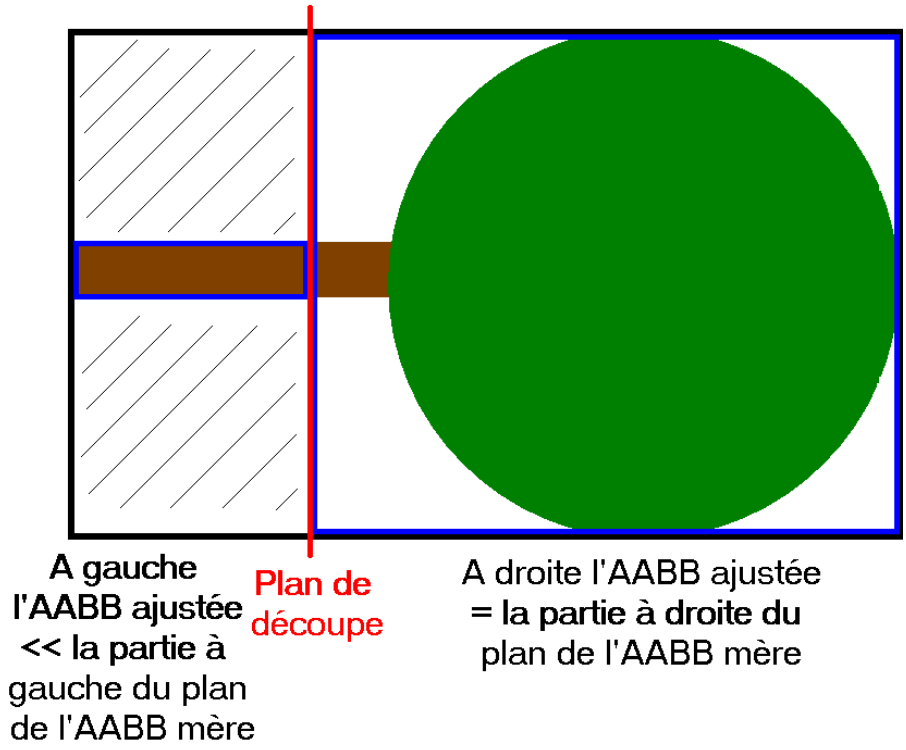


Fig. 13: Le plus souvent une seule des *AABBs* ajustées sera plus petite que la partie correspondante de l'*AABB* mère.

Cela a l'avantage de séparer les extrémités telles que le tronc ou certains groupes de feuilles du volume de départ. Par contre, vu qu'on ne prend pas en compte le volume à rendre de l'*AABB* du second fils, notre plan de découpe ne sera jamais un plan de découpe optimal (car il peut y avoir des configuration ou le gain relatif des 2 fils est plus important que celui pour le plan choisi par cette fonction de coût). De plus un gain relatif ne choisira pas forcément le gain réel optimal (confère figure 14). L'intérêt de cette heuristique réside dans le fait qu'un certain nombre de découpes peuvent être faites au bord du volume de départ, puis à partir des *AABBs* calculées, les *AABBs* voisines qui sont intéressantes à fusionner (le volume vide ajouté du à la fusion est négligeable) seront regroupées. Mais la fusion d'*AABBs* est assez coûteuse (si le nombre d'*AABBs* est important) car pour chaque *AABB*, il faut chercher ses candidats « fusionnables » avec cette dernière, et la fusionner avec le meilleur candidat (union d'*AABBs*). Pour voir si 2 *AABBs* sont fusionnables, il faut déterminer les 2 faces de contact entre ces 2 *AABBs*, contrôler qu'il n'y a pas d'autre *AABB* en contact sur une face de contact et vérifier que l'union des 2 *AABBs* n'intersectera pas une autre *AABB*... Cela nous a poussé à abandonner cette heuristique, car nous avions besoin d'un algorithme peu complexe pour la génération d'un niveau de subdivision.

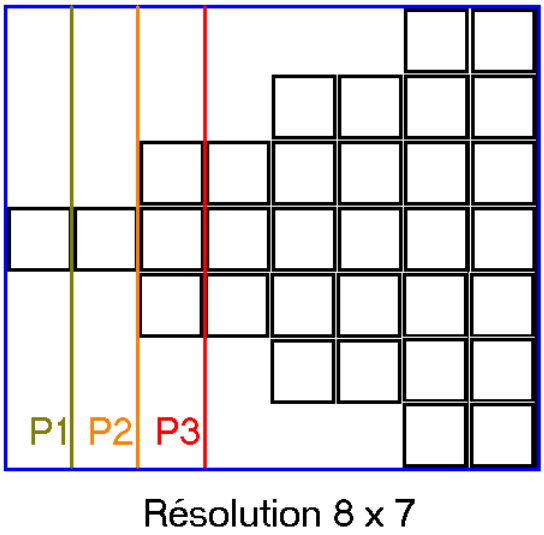


Fig. 14: Illustration 2D des défauts de l'heuristique gain volumique relatif maximal d'un fils. Seules les cellules non-vides sont affichées. Le plan de découpe P1 (resp. P2, P3) permettra d'enlever 6 (resp. 12, 12) cellules vides, soit un gain volumique du meilleur fils de 6/7 (resp. 12/14, 12/21) et un gain volumique total de 6/56 (resp. 12/56, 12/56).

La figure 14 illustre le fait que l'heuristique gain volumique relatif maximal d'un fils ne permet pas de trouver le plan qui permettra la suppression de la plus grande quantité de cellules vides, car par exemple elle va déterminer que le plan P3 est moins bon que le plan P1 ce qui est faux, car il permet de supprimer 12 cellules vides contre les 6 de P1. De plus cette heuristique ne sera pas capable de choisir entre le plan P1 et le plan P2 alors que le plan P2 est meilleur. Le gain relatif n'est pas un bon choix dans l'optique d'un gain volumique optimal direct. Par contre cette heuristique choisira un plan de découpe proche d'une extrémité pour laquelle le gain volumique est assez important, ce qui peut être intéressant si on peut fusionner les *AABBs* par la suite.

iv - Heuristique Volume_A_Rendre

Après l'échec de l'heuristique SAH et du gain volumique relatif maximal d'un fils, j'ai eu l'idée d'utiliser le nouveau volume à rendre comme fonction de coût. Il faut calculer les 2 *AABBs* pour un plan de découpe donné, une pour chaque fils. Mais pour éviter d'avoir 2 *AABBs* qui se chevauchent (car il y a des faces coupées par le plan de découpe qui sont gardées dans les 2 enfants) on fait une intersection d'*AABBs* entre l'*AABB* calculée et la partie de l'*AABB* du père correspondante (à gauche ou à droite du plan de découpe). Pour plus de détails (schéma), veuillez consulter la partie correspondante dans les points techniques (III - E).

Voici la fonction de coût utilisée (*Volume to Render*):

```
VR(N,p)  =  BV(N_L) + BV(N_R)
```

BV (*bounding volume*) est une fonction qui calcule le volume englobant minimal (scalaire) ;
p = plan de découpe ; N_L et N_R sont déterminés avec N et le plan de découpe (en associant à

N_L/N_R toutes les primitives géométriques de N à gauche/droite du plan de découpe).

Cette fonction nous donne le nouveau volume à rendre pour un plan de découpe donné. Et comme nous cherchons le minimum de cette fonction, on va maximiser la perte de volume vide. On obtiendra un minimum local (découpage d'un sous-volume) pour le volume total à rendre. Ce qui est intéressant car nous sommes limités par le nombre de boites.

Justification de notre fonction de coût (EV signifie *Empty Volume* ou volume vide): Minimiser notre fonction de coût revient à maximiser le volume vide retiré:

```
EV(N, p)   = BV(N) - ( BV(N_L) + BV(N_R) )
           = BV(volume) - VR(volume, plane)
```

Les résultats obtenus sont meilleurs que ceux obtenus avec SAH ou avec le gain volumique relatif maximal. D'une manière générale les résultats obtenus sont très bons pour l'élimination du vide autour du feuillage et pour la séparation tronc-feuillage.

Le choix du volume minimal à rendre (direct) n'est pas forcément une heuristique qui permette de trouver un minimum global (pour le même nombre de sous-volumes, il existe peut-être une autre configuration qui permet d'enlever plus de vide). Néanmoins, cette heuristique donne des résultats utilisables pour notre problème de départ. C'est donc la fonction de coût que nous avons retenu.

2 - Génération de la liste des plans de découpe candidats

Un plan de découpe doit être inclus strictement dans l'*AABB* initiale du noeud à partitionner pour permettre d'enlever du volume vide. Dans tout ce qui suit un plan de découpe candidat sera donc obligatoirement un plan qui se trouve strictement entre les 2 côtés de l'*AABB* selon l'axe de découpe. Puisque nous construisons un *KD-Tree*, les plans de découpe doivent être selon un des axes Ox, Oy ou Oz.

i - Les plans des AABBs des faces

Intuitivement les plans de découpe optimaux pour la construction d'un *KD-Tree* utilisé pour le *Ray-Tracing*, sont ceux qui se trouvent sur les *AABBs* des primitives ou des groupes de primitives proches les unes des autres, car ils permettent de faire des regroupements, ce qui limitera par la suite les calculs d'intersection.

Mais il est assez coûteux de regrouper les primitives d'un modèle complexe d'arbre, car il faudrait utiliser un algorithme de *clustering* (k-means ou k-nearest neighbors). C'est pourquoi nous avons décidé de seulement ajouter les plans des *AABBs* des faces (confère figure 15) qui sont strictement inclus dans l'*AABB* de départ. Nous faisons aussi attention à ne pas avoir de doublon. Ainsi pour chaque face, au maximum 6 plans de découpe seront ajoutés à la liste des plans. Si une face est orthogonale à un des 3 axes, son *AABB* sera plate et un seul plan sera ajouté.

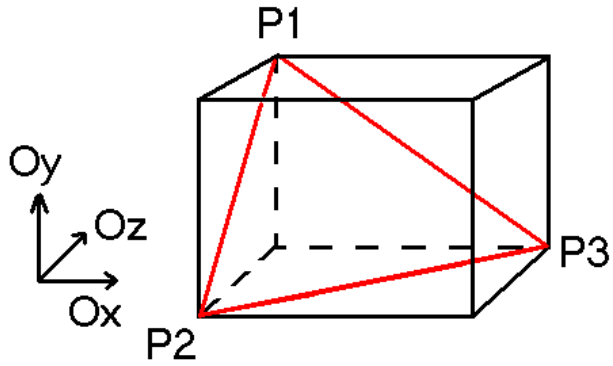


Fig. 15: *AABB* associée à une face triangulaire (P1, P2, P3).

Il n'est pas envisageable d'ajouter tous les plans des *AABBs* des faces pour nos modèles de plante ou d'arbre qui comportent plusieurs centaines de millier de faces, car le nombre de plans de découpe serait alors trop important pour permettre de calculer toutes les partitions possibles.

Après des séries de tests sur des maillages relativement simples (< 25 000 faces), nous avons conclu que cette heuristique ne donne pas de bon résultat, car le plan de découpe qui minimise le volume à rendre n'est pas forcément un plan qui passe par l'*AABB* d'une face. Le problème vient du fait que l'on cherche à partitionner un maillage 3D et non un ensemble d'objets placés dans une scène.

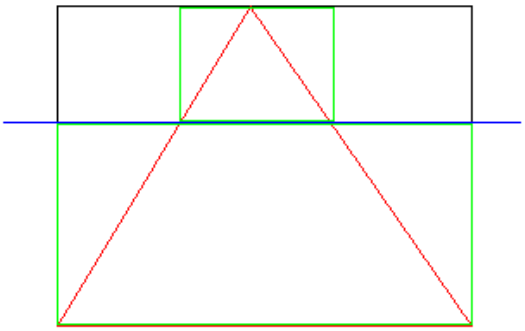


Fig. 16: Illustration 2D du fait qu'un plan de découpe optimal pour notre problème donné, ne passe pas forcément par un plan de l'*AABB* d'une face.

Dans la figure 16, aucun plan sur l'*AABB* de la face triangulaire ne permet d'enlever du vide, tandis que le plan bleu permet d'enlever relativement beaucoup de vide (plus de 20%). Pour des situations plus complexes, les plans de découpe donnés par l'*AABB* d'une face ont donc très peu de chance d'être optimaux pour notre problème donné.

Cela montre aussi qu'un plan de découpe optimal risque fortement de couper des faces. Ainsi, il faut à tout prix bannir les idées qui chercheraient à favoriser les plans de découpe qui intersectent le moins de faces possible.

ii - Subdivision aléatoire

La subdivision aléatoire consiste à ne tester qu'un sous-ensemble de plans parmi un ensemble de plans assez important. Par exemple, nous pouvons faire la subdivision aléatoire de tous les plans des *AABBs* des faces strictement incluses dans l'*AABB* initiale. Vu que les plans des *AABBs* des faces ne contiennent pas forcement les plans optimaux, cette manière de procéder ne peut pas donner de bon résultat pour notre problème initial. De plus en tirant de manière aléatoire les plans de découpe, il y a une forte probabilité de tirer des plans seulement dans la partie la plus "chargée" en primitives du volume, et donc on risque de n'avoir que des plans candidats qui ne permettent pas d'enlever beaucoup de volume. Ce qui n'est pas acceptable pour notre problème. C'est pourquoi nous avons abandonné cette heuristique.

On aurait pu tester cette heuristique en tirant aléatoirement des plans parmi des plans découpés uniformément, mais pour nos modèles, le nombre de plans nécessaires pour obtenir une bonne approximation du minimum de la fonction de coût, était petit et donc cela n'aurait pas été intéressant. En plus, vu que notre approche consiste en une construction statique pour notre *KD-Tree*, il vaut mieux utiliser une construction déterministe plus sûre.

iii - Distribution spatiale des données

Cette heuristique consiste à générer un petit nombre de plans qui prennent en compte la répartition spatiale des données (des sommets). Intuitivement pour notre problème donné, on voudrait trouver des plans qui permettent d'enlever une partie relativement peu dense en primitives (comme certains groupes de feuilles situés au bord du feuillage), car de telles parties sont susceptibles d'avoir une *AABB* plus petite, ce qui permettrait de diminuer le volume vide total à rendre. Nous avons pensé utiliser des plans tels que les plans moyens de la moitié des données triées selon un axe, le plan moyen, le plan médian... Mais de tels plans, ne prennent pas en compte la forme des faces, et donc ne sont pas optimaux le plus souvent. De plus, il n'y a pas de rapport à priori entre le volume occupé et la répartition des primitives géométriques qui s'y trouvent, car par exemple il est possible d'avoir 2 ensembles de faces avec un nombre de faces très différent, mais qui ont la même *AABB*.

Après quelques séries de tests, nous avons conclu que cette heuristique n'est pas intéressante pour notre problème initial.

iv - Subdivision uniforme

La suite logique fut de tester les plans de découpe répartis de manière uniforme. Cela peut être vu comme une discrétisation de l'ensemble infini des plans de découpe possibles (puisque le corps des réels R est dense).

Après lecture de quelques articles, ainsi que des séries de tests, nous avons conclu que l'on doit choisir entre 8 et 64 plans candidats (ou entre 8 et 64 zones découpées = 9 ou 65 plans candidats) pour chaque axe de découpe. En effet, l'utilisation d'un nombre de candidats supérieur (ex: 128 ou 129 plans) pour nos maillages 3D d'arbre et de plante n'apporte pas de gain volumique

supplémentaire (confère les propriétés des *KD-Trees* construits avec 65 et 129 plans de découpe en ANNEXES VI - B).

Plus on raffine la subdivision uniforme, meilleur le plan de découpe sera choisi (au sens du minimum de la fonction de coût), ce qui va engendrer un meilleur *KD-Tree*. En particulier sa profondeur a tendance à diminuer avec l'augmentation du nombre de subdivisions. Les résultats obtenus pour 64 ou 65 plans de découpe sont très bons pour notre ensemble de modèles (confère II - E). Nous avons aussi testé 8, 16 et 32 plans de découpe uniformes avec une approximation du minimum de la fonction de coût au moyen d'une interpolation quadratique (confère III - A), mais les résultats étaient moins bons qu'avec la découpe de 65 plans répartis de manière uniforme.

v - Subdivision uniforme et nouvelle subdivision sur la meilleure zone

Avec une subdivision uniforme simple de 64 plans selon un axe de découpe, beaucoup de calculs inutiles sont réalisés car des zones voisines “mauvaises” sont testées, alors que seulement la meilleure partie au sens du minimum de la fonction de coût utilisée serait intéressante.

C'est pourquoi une autre approche consiste à faire dans un premier temps une subdivision uniforme avec 8 plans de découpe. Puis la zone dans laquelle se trouve le minimum global est sélectionnée et elle est subdivisée uniformément avec 8 plans.

Ainsi on obtient un plan de découpe aussi bon au sens du minimum de fonction de coût que celui obtenu pour une subdivision uniforme simple de 64 plans (pour 1 axe), pour 16 plans en tout (pour 1 axe), mais avec 2 balayages.

Le problème de cette méthode, est que l'on doit faire un nouveau balayage sur la zone où se trouve le minimum. Or pour des maillages qui comportent plus de 800 000 faces le balayage est une opération très coûteuse et il est plus intéressant (en temps de calculs) d'utiliser une subdivision uniforme simple de 64 plans de découpe. C'est pourquoi cette solution n'a pas été retenue.

vi - Subdivision uniforme régressive

Pour une subdivision uniforme simple, le nombre de plans de découpe candidats nécessaire pour obtenir une partition de qualité dépend de la taille du côté le plus long de l'*AABB*. Par exemple, nous avons vu que 64 plans de découpe pour l'*AABB* initiale étaient suffisant, mais que par la suite pour des volumes plus petits, on peut utiliser moins de plans de découpe (32, 16).

Nous avons donc mis en place une subdivision uniforme régressive qui diminue le nombre de plans candidats lorsque la longueur de l'axe de découpe de l'*AABB* en cours est en dessous d'une certaine longueur (6 x et et 4 x épaisseur minimale d'une tranche (expliquée plus loin) sont les seuils de sélection).

Les résultats sont aussi bons que pour la subdivision uniforme simple, pour un gain en temps notable. Cette approche montre finalement qu'il faudrait déterminer un pas de discrétisation

entre 2 plans de découpe successifs et utiliser ce dernier pour la génération de notre ensemble de plans candidats. Mais dans ce cas on ne contrôlerait plus le nombre maximal de plans candidats. Comme la construction d'un *KD-Tree* est actuellement très performante pour la méthode de subdivision usuelle, nous avons décidé de ne pas prendre de pas de découpe, car même si pour certaines configurations, il y aura moins de plans, peut-être que pour les premières étapes de construction, il y aura plus de plans et on n'est pas sûr au final d'être gagnant en temps de calculs et en qualité.

vii - Subdivision uniforme près du bord intérieur des AABBs

Les *AABBs* de maillages d'arbre ou de plante contiennent beaucoup de vide à proximité des 6 faces de l'*AABB* (le tronc, certains groupes de feuilles). C'est pourquoi nous nous sommes dit que l'on pouvait faire seulement une subdivision uniforme près du bord intérieur des *AABBs* et éviter ainsi de tester des plans candidats dans la zone centrale de l'*AABB* qui paraissent moins bons.

Nous avons fait quelques essais et nous nous sommes rendu compte que cette heuristique était mauvaise, car il y a certains modèles pour lequel le meilleur plan de découpe au sens de la fonction de coup se trouvait vers le centre de l'*AABB*. De plus, il arrive souvent qu'après une ou 2 découpes successives, plus rien ne peut être supposé sur la position du meilleur plan de découpe.

viii - Choix de l'axe de découpe le plus long

Cette heuristique consiste à choisir l'axe de découpe le plus long pour une *AABB* donnée. Cela permet de diviser par trois le nombre de plans de découpe à tester dans le cas d'un nombre fixe pour chaque axe de découpe (en général 8, 16, 32 ou 64 plans de découpe à déterminer pour un axe). Ce choix ou celui d'un seul axe (par ex: profondeur du noeud courant **mod** 3) sont des techniques couramment utilisées pour accélérer la construction des *KD-Trees*, sans pour autant produire des *KD-Trees* très éloignés (en qualité) de la solution obtenue en considérant les 3 axes.

Un autre avantage de cette heuristique, est qu'elle permet en général de découper selon des axes différents pour 2 découpes successives lorsque l'*AABB* de départ n'est pas trop disproportionnée. Ce qui est le cas pour un modèle d'arbre ou de plante (de qualité). Donc les *AABBs* finales ne seront pas trop « étirées ».

En fait cette heuristique est bien adaptée à la morphologie des arbres ou végétaux. Pour mieux comprendre cela, imaginons les étapes successives sur un arbre de type prunier (confère figure 17):

- 1ère étape, l'axe le plus long est celui sur lequel on va séparer le tronc du feuillage et la subdivision du tronc s'arrête car son volume est trop petit (c'est un critère d'arrêt)

- 2ème étape, on s'occupe du feuillage. Comme le volume vide direct qui peut être gagné est celui sur les 8 coins du volume du feuillage, le plan de découpe choisi sera en général près du bord du feuillage. Le plus petit volume n'est plus subdivisé et on continue ainsi de suite sur le reste du

feuillage. Comme la forme générale du feuillage est proche de celle d'une sphère, chaque fois que du volume vide sera enlevé le long d'un axe de découpe, à l'étape suivante, du volume selon un autre axe sera supprimé si nécessaire. Cela permettra d'avoir une partition du volume initiale symétrique dans la plupart des cas.

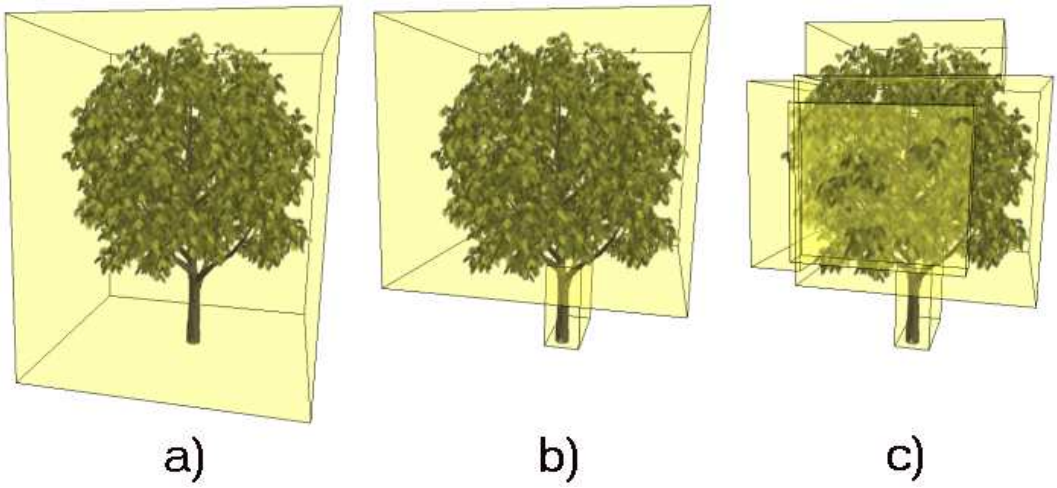


Fig. 17: Résultats obtenus avec le choix de l'axe le plus long.
Les *AABBs* sont ajustées après chaque découpe.
a) Racine du *KD-Tree*
b) 1ère subdivision
c) Subdivision finale (toutes les feuilles du *KD-Tree*)

Après quelques séries de tests, nous avons conclu que cette heuristique est très intéressante car elle permet de perdre pratiquement autant de vide qu'avec les 3 axes de découpe (42,38% contre 43,57% en moyenne pour les modèles d'arbre et de plante à disposition) et avec un nombre plus petit de sous-volumes à rendre (confère ANNEXES VI - B 1 et 3).

B - Construction du KD-Tree: Minimisation du nombre de feuilles et critères d'arrêt

Une fois que nous savons générer notre liste de plans candidats et que nous avons notre fonction de coût pour sélectionner le meilleur (pour un noeud donné), il faut trouver des bonnes heuristiques et des bons critères d'arrêt afin de diminuer le nombre total de feuilles du *KD-Tree*.

1 - Ajustement des *AABBs* des nouveaux noeuds

Cela a été introduit avec la présentation de l'heuristique SAH, mais il faut rappeler que l'ajustement des *AABBs* des 2 nouveaux noeuds fils après chaque découpe est essentiel pour diminuer la profondeur du *KD-Tree* et le nombre de feuilles total. En particulier cet ajustement évite d'avoir des noeuds vides au sein du *KD-Tree* (partition de l'espace non-vide) et permet d'avoir une structure hiérarchique d'*AABBs* qui va de plus en plus épouser la géométrie d'un modèle d'arbre ou de plante (confère figure 17).

2 - Épaisseur de tranche minimale le long de l'axe de découpe

Ce critère de sélection des plans de découpe candidats, consiste à choisir seulement les plans qui permettent d'avoir une "tranche" d'épaisseur minimale (confère figure 18). Il nous donne les bornes de début et de fin pour la découpe uniforme. Cela permet de ne pas obtenir de volumes trop « aplatis » et par la même occasion permet de diminuer le nombre total de feuilles du *KD-Tree*, puisqu'un noeud dont la dimension maximale de son *AABB* est inférieure strictement à 2*épaisseur minimale d'une tranche ne sera pas subdivisé (dans ce cas c'est un critère d'arrêt). De plus, cela a tendance à générer des volumes de tailles équivalentes sur les contours des modèles (fait observé), zones où l'on cherche à retirer du vide. Et cela est intéressant pour l'obtention de découpes symétriques.

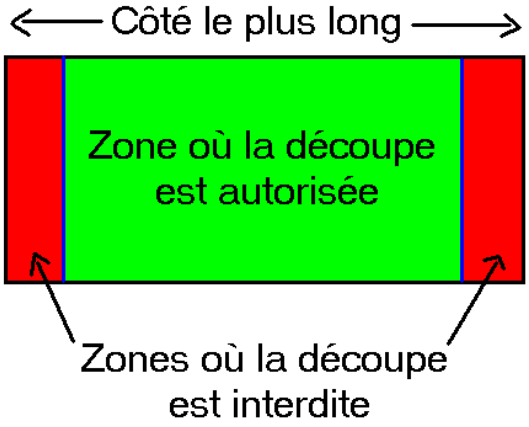


Fig. 18: Illustration 2D de l'heuristique tranche minimale lorsque la découpe est faite selon l'axe de plus long de l'*AABB*.

Cette heuristique, à elle seule, permet d'arrêter la construction du *KD-Tree*, car il arrivera un moment où il ne sera plus possible de « couper » des tranches d'épaisseur minimale pour l'ensemble des sous-volumes non-vides du *KD-Tree*. Cette limite nous donnera le volume d'espace vide maximal que l'on peut retirer. Cette valeur nous sera utile pour déterminer si les autres heuristiques sont efficaces ou non.

Nous avons fait des séries de tests et il ressort que 10% de la valeur de la plus petite dimension de l'*AABB* initiale est une valeur bien adaptée pour l'ensemble des modèles. En utilisant seulement cette heuristique, nous avons obtenu en moyenne 132 feuilles non-vides par *KD-Tree*, pour un volume d'espace vide retiré de 63% du volume initial en moyenne (voir ANNEXES C).

3 - Volume minimal

Ce critère d'arrêt consiste à ne subdiviser une *AABB* associée à un noeud, que si cette dernière a un volume suffisamment important par rapport au volume initial de l'*AABB* du modèle étudié.

Cela se justifie parce que notre objectif est d'avoir un gain volumique important par

rapport au volume initial avec un petit nombre de sous-volumes, est donc il est préférable de ne pas subdiviser des volumes très inférieurs au volume de référence.

Nous avons fait des séries de tests et il s'est avéré que le meilleur choix pour les modèles d'arbre ou de plante utilisés, est un volume minimal égal à 10% du volume initial (scalaire). Il est clair que pour d'autres types de maillages 3D, ou pour des scènes contenant plusieurs objets, le choix du volume minimal pourra être adapté pour permettre d'obtenir le raffinement désiré.

4 - Gain volumique minimal direct

Toujours dans le but de minimiser le nombre de sous-volumes générés, on peut éviter la découpe d'un volume, si cette dernière ne permet pas de perdre plus de 3% du volume courant avec le meilleur plan de découpe trouvé. C'est la valeur la plus adaptée à l'ensemble des modèles d'arbre ou de plante étudiés.

Cela permet en outre d'éviter de subdiviser la partie centrale d'une plante ou du feuillage d'un arbre qui est en général très chargée en primitives géométriques (faces) et qui est souvent beaucoup plus grande que le volume minimal.

Le nouveau volume à rendre associé au meilleur plan est calculé (somme des volumes englobants des 2 fils) pour savoir si le noeud courant va être effectivement subdivisé ou non. Un inconvénient de ce critère d'arrêt est donc que la sélection du meilleur plan doit être effectuée même si finalement la subdivision n'aura pas lieu.

5 - Profondeur maximale Pmax

Un moyen simple pour contrôler le nombre de feuilles non-vides du *KD-Tree* construit, est de limiter sa profondeur. Cela nous donne une borne maximale du nombre de sous-volumes total (2^{Pmax}). Ainsi on peut limiter la consommation en mémoire vive. Au cours de l'étude bibliographique on a pu constater qu'en général pour le ray-tracing on utilise une profondeur maximale de 20. Nous avons aussi décidé de prendre 20 comme profondeur maximale, même si la nature de notre problème est différente (donc $Pmax \leq 20$) pour limiter les besoins en mémoire vive. Cette profondeur nous permet aussi de regarder si les heuristiques utilisées sont de bonne qualité, c'est-à-dire si l'on obtient un petit nombre de boites, à fortiori une petite profondeur maximale pour un *KD-Tree* construit.

Après des séries de tests, il s'est avéré qu'une profondeur de 7 serait aussi adaptée pour nos modèles d'arbres et de plantes, car les *KD-Trees* construits ont en moyenne une profondeur maximale inférieure ou égale à 7 (pour la subdivision uniforme avec 65 plans de découpe selon l'axe le plus long et avec les paramètres de références pour la tranche minimale, le volume minimal et le gain volumique direct). Ainsi, si on limite la profondeur du *KD-Tree* pour les modèles d'arbre ou de plante, on ne doit pas prendre de $Pmax < 7$. Une profondeur maximale ≥ 7 permet de regarder si la découpe près des 6 faces de l'*AABB* initiale est intéressante. Ici grâce à l'heuristique du volume minimal, qui va empêcher de raffiner les volumes trop petits, en général seul le plus gros volume restant va être subdivisé lors de la construction du *KD-Tree*. Et à la profondeur 7 le *KD-Tree* en

cours de construction est très proche du *KD-Tree* final, mais avec un nombre de sous-volumes plus intéressant.

Il est important d'insister sur le fait que l'heuristique de la profondeur maximale à 7 fonctionne bien pour des modèles d'arbre ou de plante, mais qu'elle ne doit pas être utilisée à priori pour d'autre types de maillages 3D. Ensuite, si on modifie la dimension minimale ou le volume minimal, il faudra adapter la profondeur maximale en conséquence.

Pour tous nos tests, nous avons pris une profondeur maximale de 20, pour pouvoir évaluer la qualité de toutes nos heuristiques.

6 - Densité maximale

Les modèles utilisés sont de qualités, c'est-à-dire que le maillage de départ est assez raffiné. Ainsi, d'une manière générale les faces triangulaires sont petites et bien reparties. Si on a un moyen intelligible de calculer la densité (voir points techniques pour plus de détails), qui intuitivement représente le « chargement » en primitives géométriques d'un volume (en fait d'un noeud de l'arbre), on pourra estimer le vide restant dans le volume. Par exemple, si la densité est trop élevée, alors il n'y aura plus beaucoup de vide. Donc au-dessus d'un certain seuil il sera inutile de continuer la subdivision d'un noeud du *KD-Tree*.

Nous avons fait plusieurs tests, et il s'est avérer que le meilleur critère utilisant la densité est d'autoriser la subdivision d'un sous-volume si au-moins un des 2 futurs fils a une densité inférieure à la densité maximale autorisée (seuil). Donc si les 2 futurs fils ont une densité supérieure au seuil fixé, la construction du noeud en cours s'arrêtera.

La valeur utilisée pour la densité maximale est la densité initiale du modèle. Ainsi, si un plan de découpe ne permet pas d'extraire une partie moins dense que la densité initiale, on le rejettera. Il est important d'utiliser la valeur de la densité initiale du modèle, car cette dernière varie fortement d'un maillage à un autre. Plus un modèle est complexe, c'est-à-dire plus il contient de faces, plus sa densité initiale sera élevée.

Il faut quand même faire attention à la taille des faces triangulaires, car le calcul de la densité tel qu'il est, n'aurait plus de sens s'il y avait des faces triangulaires de tailles très différentes ou très grandes. Dans ce dernier cas les zones de chevauchement des *AABBs* des faces seraient trop importantes pour donner du sens à notre calcul de densité...

Ce critère d'arrêt n'a pas été retenu pour la version finale basée sur les maillages 3D, car les résultats obtenus étaient très similaires à ceux obtenus sans ce dernier. En plus le volume engobant d'une même face peut varier en fonction de son inclinaison par rapport à un plan parallèle aux axes (car son *AABB* peut avoir un volume plus ou moins important), et donc la densité d'un noeud peut varier fortement avec les mêmes faces triangulaires, simplement en modifiant leur position spatiale. Il serait donc difficile de justifier son utilisation. Par contre il serait intéressant d'utiliser ce critère d'arrêt pour un *KD-Tree* qui partitionne directement des voxels, car en manipulant des données volumiques, la densité en voxels non-vides aurait du sens.

C - Création d'un niveau de subdivision

Pour le choix des sous-volumes associés avec un niveau de subdivision, il y a 2 possibilités. A partir du *KD-Tree*, soit les volumes à rendre sont sélectionnés « à la main », soit un procédé de sélection automatique est mis en place. Nous avons décidé de faire une génération automatique pour les trois niveaux de subdivision, pour éviter de passer par une phase d'analyse avec sélection des sous-volumes désirés car le nombre de subdivisions possibles peut être assez important.

Pour chaque plan de découpe (donc pour chaque noeud interne (≠feuille) du *KD-Tree*), il y a 2 possibilités, soit on découpe et on continue le parcours de l'arbre, soit on s'arrête. Le nombre de subdivisions possibles (NB_SUB) pour un noeud interne est donc égal au nombre de subdivisions possibles de son fils gauche fois celui de son fils droit plus 1. Pour une feuille, il n'y a qu'un seul choix. On peut donc calculer le nombre de subdivisions possibles de manière récursive:

NB_SUB(N**) =** **si** N est une feuille
retourner 1
sinon
retourner **NB_SUB(**N.G**)**x**NB_SUB(**N.D**)** +1

En fait avec notre manière de construire le *KD-Tree*, grâce aux critères d'arrêt choisis, la plupart du temps un seul des 2 fils d'un noeud sera subdivisé à nouveau pour un modèle d'arbre, c'est-à-dire qu'un des 2 fils sera une feuille et que son nombre de subdivisions associé sera 1 (confère figure 19). Pour un modèle d'arbuste ou de plante, il y a le gros volume central qui est subdivisé le long de la plus longue branche du *KD-Tree*, mais il y a parfois une subdivision en plus pour un noeud qui ne se trouve pas sur cette branche principale. Cela vient du fait que le feuillage est moins dense en faces triangulaires que pour nos modèles d'arbre.

Pour un modèle d'arbre qui a un feuillage compact, le nombre de subdivisions possibles devient alors 1 pour la racine et on ajoute 1 chaque fois que l'on rencontre un nouveau noeud intérieur et on ajoute encore 1 à la fin. Ce qui donne comme valeur finale le nombre de noeuds d'un plus long chemin ou encore la profondeur du *KD-Tree* + 1. Comme pour nos modèles d'arbre ou de plante, nous obtenons en moyenne une profondeur maximale de 7, il y aura en moyenne 8 niveaux de subdivision possibles au minimum. Cela justifie l'utilisation d'une méthode automatique ou semi-automatique pour le choix des 3 niveaux de subdivision.

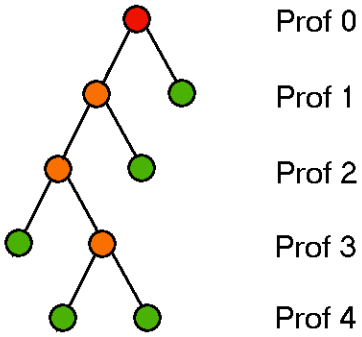


Fig. 19: *KD-Tree* particulier pour lequel 5 niveaux de subdivision sont possibles.
En rouge la racine du *KD-Tree*, en orange les noeuds intérieurs autres que la racine et en vert les feuilles.

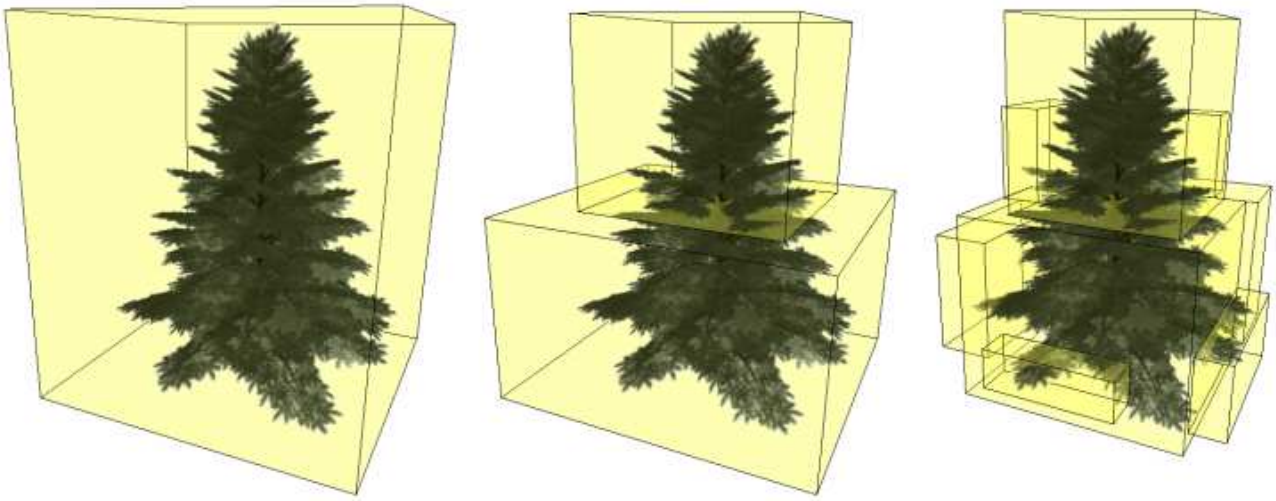


Fig. 20: Des niveaux de subdivision possibles pour le modèle *Picea* (avec 1, 2 ou 8 sous-volumes).

1 - Sous-volume et texture 3D associée

Maintenant nous avons à notre disposition un ensemble de sous-volumes ou d'*AABBs* et un maillage 3D texturé. Et le problème consiste à rendre le volume associé à notre maillage 3D, en utilisant des textures 3D. Nous disposons d'un processus de voxelisation qui nous permet de générer une texture 3D à partir d'un maillage 3D. Une texture 3D peut donc être générée pour chaque sous-volumes, ou alors la texture 3D pourra être générée pour le maillage initial et ensuite il faudra recalculer les coordonnées de texture associées à chaque sous-volume. La première méthode est plus coûteuse en mémoire car il y aura une texture 3D par sous-volume à stocker, sans oublier la génération des différents niveaux de *MIP-map*.

Pour la voxelisation d'un maillage 3D donnée par « *slicing* » ou tranchage, les dimensions de la résolution utilisée pour une tranche doivent être spécifiées, comme la résolution en voxels du volume, sizeX*sizeY*sizeZ où size est une puissance de 2 (nombre de tranches selon l'axe). En fait certaines cartes graphiques (GPU) ne supportent pas les textures qui ont des dimensions autres que des puissances de 2, et justement le procédé de *slicing* utilisé par mon

superviseur pour créer des textures 3D ne permettait que la génération de telles textures 3D (avec 256 comme résolution maximale selon un axe). Une texture 3D (3 piles de textures 2D si la carte graphique ne supporte pas les textures 3D) est générée avec ses niveaux de *MIP-map*.

Pour le niveau de subdivision 0, la texture 3D de base en résolution 256x256x256 est utilisée. Pour les 2 autres niveaux de subdivision, sans regarder le coût mémoire, si on garde la même résolution que pour la texture 3D initiale pour chaque sous-volume, cela va augmenter la résolution de base car il y aura plus de tranches au total. Et le *framerate* va diminuer au lieu d'augmenter comme souhaité, car le gain en volume vide non rendu sera inférieur au surcoût du à la résolution utilisée.

Toujours sans regarder le coût mémoire, une première approche fut de choisir la puissance de 2 la plus proche de la résolution du volume de base pour un sous-volume. Nous avions découpé le maillage initial en sous-maillages et nous avions appliqué le processus de *slicing* sur chaque sous-maillage. Dans ce cas on peut avoir des résolutions différentes de part et d'autre d'un plan de découpe (entre un petit et gros sous-volume par exemple) et cela selon les trois axes. L'artefact visuel se caractérise par un léger gonflement et une texture plus floue si la résolution diminue. Cela reste acceptable lorsque la différence de résolution entre 2 sous-volumes voisins n'est pas trop grande. Le problème principale ici, c'est que l'on approxime la résolution voulue pour un seul bloc (256x256x256) pour un sous-bloc, en calculant la résolution correspondante pour le sous-volume, puis en prenant la puissance de 2 la plus proche. Et justement cette approximation est la cause principale des artefacts visuels. De plus il y a des configurations où la résolution choisie est plus importante que la résolution globale souhaitée, et donc il arrive parfois qu'une représentation avec seulement 2 ou 3 boites pour un gain en volume vide supérieur à 30% soit moins intéressante que la texture 3D du niveau de subdivision 0. De plus, dans les cas où le *framerate* serait plus intéressant, on ne pourrait pas affirmer que cela provient de notre méthode, car cela pourrait aussi bien venir du fait que la résolution utilisée pour certains sous-volumes est inférieure à celle de référence. Donc cette manière de procéder est inacceptable.

Certaines cartes graphiques permettent d'utiliser des textures de n'importe quelle résolution, et pas seulement des puissances de 2. Cela permettrait d'éviter tous les inconvénients dus à l'approximation à la puissance de 2 la plus proche. En particulier il n'y aurait plus de différence de résolution notable entre 2 sous-volumes voisins (par rapport à la résolution de référence). Cela permettrait aussi d'avoir une variation du *framerate* en fonction du gain volumique beaucoup plus prédictible. Dans les travaux futurs, il faudra donc adapter notre algorithme de voxelisation pour créer des textures 3D avec des dimensions quelconques.

Notre deuxième approche consiste à utiliser une seule texture, celle du *mapping* initial, et de spécifier les bonnes coordonnées de texture pour chaque sous-volume (confère figure 21). Cela a plusieurs avantages, d'abord il n'y a plus les différences de résolution visibles entre 2 sous-volumes voisins, mais aussi l'espace mémoire pour le stockage des différents niveaux de *MIP-map* est beaucoup plus petit qu'avec la méthode précédente. Ce qui est bien avec cette méthode, c'est que le gain en *framerate* sera du à la perte de volume vide, et non pas au fait que des sous-volumes peuvent utiliser des textures avec des résolutions plus petites que celle de départ. C'est cette approche qui a été retenue pour la méthode finale.

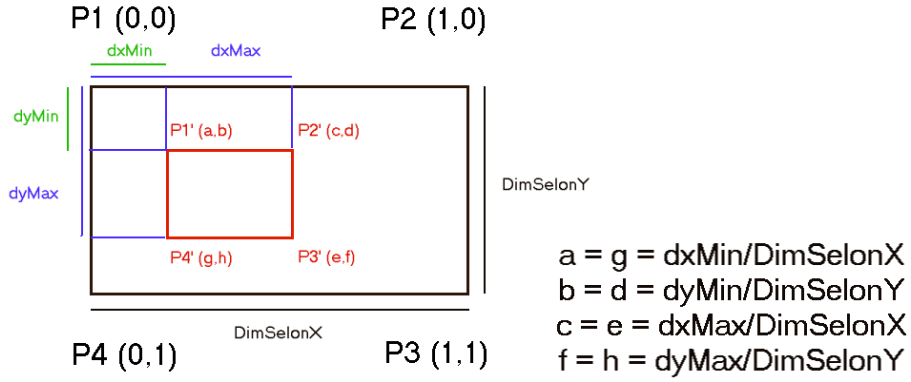


Fig. 21: Illustration 2D de la détermination des coordonnées de texture d'un sous-volume connaissant les sommets de référence.
A droite de chaque point, il y a les coordonnées de texture.
Les dx, dy et Dim sont des distances (scalaires).

2 - Sous-volumes, transparence et alpha-value

Pour prendre en compte la transparence dans l'affichage de l'ensemble des sous-volumes associés à un niveau de subdivision, il faut afficher les volumes les plus éloignés (par rapport à la position de la caméra) en premier et ainsi de suite. Il faut donc trier les *AABBs* de manière à gérer correctement la transparence (confère figure 22).

Fig. 22: Tri d'*AABBs* pour les afficher dans l'ordre *back-to-front* (de la plus éloignée à la plus proche).

Le tri d'*AABBs* n'est pas évident, car on peut toujours trouver des cas critiques. Par exemple, ce qui semblerait une bonne solution, dire qu'une *AABB* est située devant une autre *AABB* si la distance entre le point le plus proche de l'*AABB* et la position de la caméra est inférieure à celle

de l'autre *AABB*, ne marcherait pas en fait (confère figure 23).

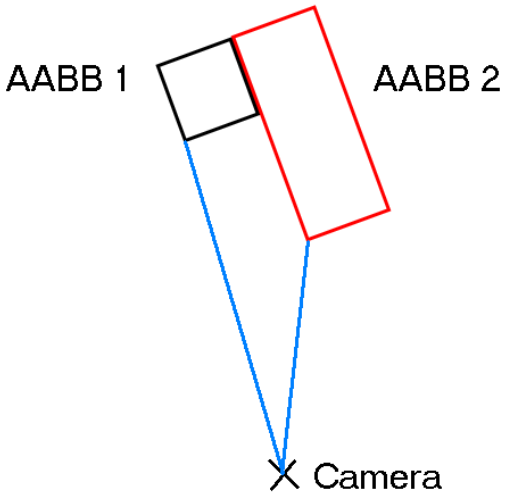


Fig. 23: Le tri d'*AABBs* n'est pas simple.
L'*AABB* 2 est plus proche de la camera,
mais doit être affichée avant l'*AABB* 1
pour gérer correctement la transparence.

En fait, il semblerait qu'il n'y ait pas de solution simple à ce problème (peu coûteuse en calculs), car il faudrait analyser la priorité d'affichage de chacune des faces d'une *AABB* par rapport à une autre *AABB* pour finalement déterminer l'ordre d'affichage des 2 *AABBs*. Lorsque le nombre d'*AABBs* devient important, on voit que le coût du tri risque de faire perdre le gain en *framerate* du au gain volumique. C'est pourquoi par la suite nous proposons des solutions simples à mettre en oeuvre (peu de calculs) qui sont certes imparfaites, mais qui approximent bien le tri d'*AABBs* (peu d'erreurs).

Nous avons choisi comme première approche de dire qu'une *AABB* est située avant une autre *AABB* par rapport à la position de la camera, si le point d'intersection entre l'axe de vue (qui est déterminé avec la position de la caméra et la direction de vue) et le plan le plus orthogonal et le plus proche de la première *AABB* est à une distance de la camera plus petite que celui de l'autre *AABB*. Pour déterminer le côté de l'*AABB* le plus othogonal à la direction de la caméra, on regarde les 2 plans de l'*AABB* dont la normale est la plus proche de la direction de la caméra, et on choisit le plus proche (en regardant la distance axiale adéquate).

Notre seconde approche, qui donne une meilleure approximation du tri d'*AABBs*, consiste à comparer les distances entre les centres des plans des *AABBs* les plus orthogonaux à la direction de vue et les plus proches (de la caméra) et la position de la caméra. Les distances les plus grandes nous donnerons les *AABBs* les plus éloignées.

Avec les méthodes de tri précédente, le tri d'*AABBs* est plutôt correct, mais sur certaines positions limites, une bande blanche due à l'affichage des sous-volumes dans un mauvais ordre (et donc une mauvaise gestion de la transparence) peut s'observer. On peut utiliser l'*alpha-test* avec une *alpha-value* de référence adéquate pour diminuer cet effet de bande blanche. L'*alpha-test* est un test

qui compare la valeur stockée dans le canal alpha d'un voxel avec la valeur de référence à l'aide d'une fonction de comparaison choisie à l'avance (<, ≤, >, ≥, =, != et composition des fonctions précédentes). Il doit être explicitement activé (*enabled*) pour être appliqué. Si le test réussi, la couleur du voxel (en fait la couleur d'un *fragment*) sera prise en compte pour le rendu de la couleur du pixel associé, avec la formule de mélange adéquate. Si le test échoue, le *fragment* est rejeté et ne sera pas pris en compte. L'alpha-value de référence doit être choisie entre 0 (transparent) et 1 (opaque). L'effet bande blanche est moins remarquable lorsqu'on utilise un alpha-test avec pour alpha-value de référence 0,220 et pour fonction de comparaison <.

Le tri d'*AABBs* est un problème complexe qui ne dépend pas que de la position et de la direction de la caméra, et je n'ai pas trouvé de solution simple lors de mes recherches. Néanmoins, les résultats obtenus sont satisfaisants car il n'y a pas de surcoût important du au tri et la plupart du temps la gestion de la transparence est correcte.

D - Choix d'un niveau de subdivision et transition

Pour pouvoir être capable de choisir le meilleur niveau de subdivision, c'est-à-dire celui qui nous permettra d'obtenir le meilleur *framerate* par seconde, il faut étudier attentivement les différents paramètres qui interviennent dans le coût du rendu. Le choix des volumes utilisés dans un niveau de subdivision doit être aussi soigneusement étudié, car il arrive par exemple qu'un niveau de subdivision avec k boites ne soit pas intéressant car son coût est toujours plus cher que le niveau de subdivision de base (avec une seule boite) ou pas assez différent d'un autre niveau de subdivision (surtout si la transparence est correctement gérée ou si le calcul d'ombre avec des *shadow maps* [WILLIAMS 1978] est actif).

Plus un niveau de subdivision sera grand, plus il contiendra des sous-volumes. Pour les vues de près, nous utiliserons le niveau L2 (le plus grand niveau de subdivision), pour les vues intermédiaires L1 et pour les vues éloignées le niveau de subdivision L0. La restriction à trois niveaux de subdivision peut être assez contraignante, car pour certains modèles peut-être plus de niveaux seraient nécessaires pour toujours maximiser le *framerate*.

1 - Choix des trois niveaux de subdivision

Les *KD-Trees* construits pour chaque modèle, ont la bonne propriété de fournir un petit nombre de sous-volumes (moins d'une dizaine en moyenne) pour un gain volumique assez important (entre 30% et 54%) (confère ANNEXES VI - B - 1). Cela va restreindre notre choix de niveaux de subdivision possibles.

Après plusieurs séries de tests, nous avons conclu que le meilleur choix pour le niveau de subdivision L2 était la configuration dans laquelle on prend toutes les feuilles associées au *KD-Tree*. Pour les vues proches, c'est la configuration qui permet d'obtenir le meilleur *framerate*. Cela s'explique car il y a peu de sous-volumes à gérer et que c'est la configuration où il y a le plus de gain volumique. Le meilleur niveau de subdivision L0 reste le volume de base pour les vues éloignées. Et donc il nous reste à déterminer les critères de sélection du L1.

Intuitivement, un bon candidat pour L1 est un L1 pour lequel le gain volumique est

assez proche de la moitié du gain volumique du L2. Il faut aussi que le nombre de sous-volumes utilisés soit proche de la moitié du nombre de volumes utilisés dans le L2 pour que le coût du au nombre de volumes augmente proportionnellement avec le niveau de subdivision. Le choix du L1 devra donc être adapté au modèle en essayant de trouver le meilleure compromis entre gain volumique et nombre de sous-volumes, pour que le L1 soit bien distinct des 2 autres niveaux de subdivision. Nous avons écrit une méthode qui prend en paramètres le pourcentage d'espace vide à retirer et le nombre de sous-volumes maximal pour le faire, puis elle essaye de trouver le meilleur compromis comme désiré. Pour les modèles testés, j'ai pris 50% du gain volumique maximal et Nb_Feuilles_Non_Vides/2+2 pour le nombre maximal de sous-volumes. Cela donne un bon niveau L1 de subdivision. Pour plus de détails, veuillez consulter la partie technique correspondante (III - G).

2 - Niveau de subdivision et framerate

La *résolution de la texture 3D*, c'est-à-dire le nombre de tranches utilisé à un niveau de *MIP-map* donné influence le *framerate*, car il y a plus d'accès mémoire et plus de calculs (transparence par exemple) pour chaque pixel affiché.

L'*éloignement du modèle par rapport à la position de la caméra*, influence aussi le *framerate*, car plus un modèle est éloigné, moins il y a de pixels à afficher, mais aussi si le niveau de *MIP-map* est adaptatif, la résolution utilisée sera moindre. Ce qui contribuera à l'augmentation du *framerate*. Toujours dans le même ordre d'idées, les *dimensions ou l'aire des tranches* est à prendre en compte, car elle change selon l'axe de découpe et selon le modèle étudié.

Pour ce qui est du coût du *tri des AABBs* des différents volumes à afficher, il s'est avéré que le coût du tri est négligeable si le modèle est assez proche de la camera (avec le niveau de subdivision L2). Par contre si le modèle est plus éloigné (toujours avec la partition donnée par L2), le coût du tri ne devient plus négligeable et le *framerate* peut diminuer. Cela s'explique car le niveau de *MIP-map* choisi sera plus élevé, donc le coût nécessaire pour afficher un pixel associé à des texels de la texture sera moindre. Il est tout naturel de se poser la question suivante, est-ce que le tri serait nécessaire pour un modèle très éloigné de la camera? Pour le moment il est nécessaire, car il y a toujours des configurations pour lesquelles une bande blanche pourrait être visible. Mais cela n'est pas trop grave, puisque si le modèle est très éloigné, le niveau de subdivision choisi sera certainement L1 ou même L0 (avec un seul volume) et le coût du tri sera négligeable.

Après les tests effectués sur des données observées, nous sommes arrivés à la conclusion que le *framerate* ne pouvait pas se déduire linéairement, et même en cantonnant les paramètres calculés sur des petits intervalles de variation. Cela vient du fait que l'influence de chaque paramètre (nombre de tranches, aire, distance, gain volumique) n'est pas linéaire.

3 - Critère de passage d'un niveau de subdivision à un autre

Nous voulions un critère simple à calculer pour ne pas faire diminuer le *framerate*, mais aussi un critère qui ne fasse pas trop de mauvais choix pour le meilleur niveau de subdivision. De plus nous avons construit des niveaux de subdivision très différents pour diminuer la zone d'incertitude entre 2 niveaux de subdivision voisins (L2 et L1, L1 et L0).

Le critère simple que l'on a identifié, est le ratio entre la dimension maximale de l'*AABB* initiale et la distance entre le centre du modèle et la caméra:
R = MaxDimInitAABB / Distance(Camera, ModelCentre).

Ce ratio n'étant calculé que si **Distance(Camera, ModelCentre)** est différent de zéro. Dans le cas contraire on fixe le niveau de subdivision à L2. La valeur de ce ratio pour le passage d'un niveau de subdivision à un autre change en fonction de la résolution de base utilisée, mais aussi en fonction de la qualité de rendu désirée (ex: ombres ou non). Il faudra donc adapter les valeurs de référence pour le choix d'un niveau de subdivision en fonction de la résolution de base des données volumiques, mais aussi en fonction du coup de gestion des sous-volumes.

Pour une résolution de texture 3D de 256x256x256, sans utiliser la gestion de la transparence et le calcul d'ombre (par *Shadow Map*), il est ressorti que si R < 0,08 alors on doit choisir le L0, sinon si R < 0,14 on doit choisir L1 et sinon L2. Pour les modèles de plante ou d'arbre de notre banc d'essais, ce critère de passage d'un niveau de subdivision à un autre donne des résultats correct, dans le sens où le nombre de mauvais choix est relativement faible, mais aussi lorsqu'il y a un mauvais choix le *framerate* n'est pas trop éloigné de l'optimal.

Pour passer d'un niveau de subdivision à un autre, tout en utilisant le tri d'*AABBs* pour gérer la transparence et/ou les ombres, il faut translater légèrement les valeurs de passage d'un niveau de subdivision à un autre. Cette translation traduit le fait que la gestion des sous-volumes a un coût et qu'il faut le prendre en compte. Les différents essais sur notre ensemble de modèles nous ont mené à utiliser eps=0,06 et si R < 0,08+eps alors on choisit L0, sinon si R < 0,14+eps on choisit L1, sinon L2.

4 - Transition et artefact visuel

Quand est-il du problème des transitions? Y-a-t-il des artefacts visuels? La réponse dépend en fait si on gère la transparence ou non.

Si la transparence n'est pas gérée correctement (pas de tri d'*AABBs*), il y a des coupures visibles entre les sous-volumes utilisés pour un niveau de subdivision donné (confère figure 22). Et comme le nombre de sous-volumes change en fonction du niveau de subdivision utilisé, les transitions seront visibles.

Par contre si la transparence est bien gérée (tri des *AABBs* en fonction de la position de la caméra pour afficher les plus éloignées en premier), il n'y a pas de problème de transition, car rien de visible permet de constater qu'il y a eu un changement. En fait la seule différence constatable est la variation du *framerate*, qui est en fait pratiquement continue lorsqu'on a bien choisi ses paramètres de transition.

E - Solution retenue et résultats visuels

La solution retenue pour notre problème de base consiste à utiliser la subdivision uniforme régressive avec 65 plans de découpe candidats selon l'axe le plus long d'une *AABB* d'un noeud du *KD-Tree*, et de choisir le meilleur plan en minimisant la valeur de la fonction de coût

Volume_A_Rendre. A chaque découpe les *AABBs* filles sont ajustées pour permettre une partition des volumes non-vides. Les critères d'arrêt utilisés sont la tranche d'épaisseur minimale (10% de la longueur de la plus petite dimension de l'*AABB* initiale), le volume minimal (10% du volume initial) et le gain volumique direct (3% du volume parent) (la profondeur maximale n'est en pratique jamais utilisée car la construction du *KD-Tree* s'arrête rapidement). Une fois le *KD-Tree* construit, la génération des 3 niveaux de subdivision est faite automatiquement. Le niveau L0 correspond à la texture 3D initiale (générée par voxelisation d'un maillage 3D). Le niveau L2 correspond à l'ensemble des feuilles du *KD-Tree* construit et le niveau intermédiaire L1 est déterminé en recherchant la première subdivision qui permette de retirée 50% du volume vide retiré avec L2 avec un nombre de sous-volumes maximal (nombre de feuilles non-vides /2 +2).

Voici quelques résultats visuels des 3 niveaux de subdivision associés à différents modèles volumiques. Les *AABBs* sont mises en évidence pour faciliter leur visualisation.

1 - Données volumiques d'arbres et de plantes

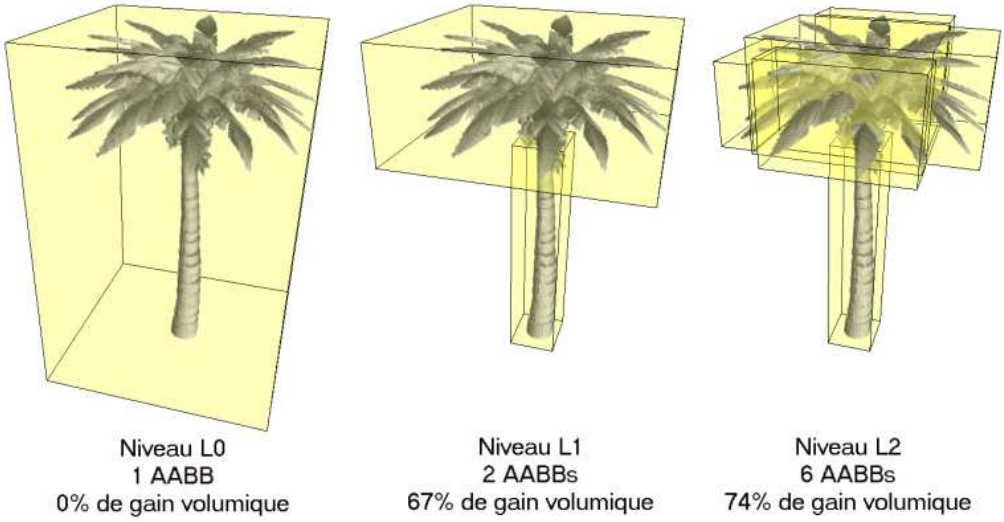


Fig. 24: Palmier: Modèle volumique d'un arbre pour lequel il y a beaucoup de vide autour du tronc et du feuillage.

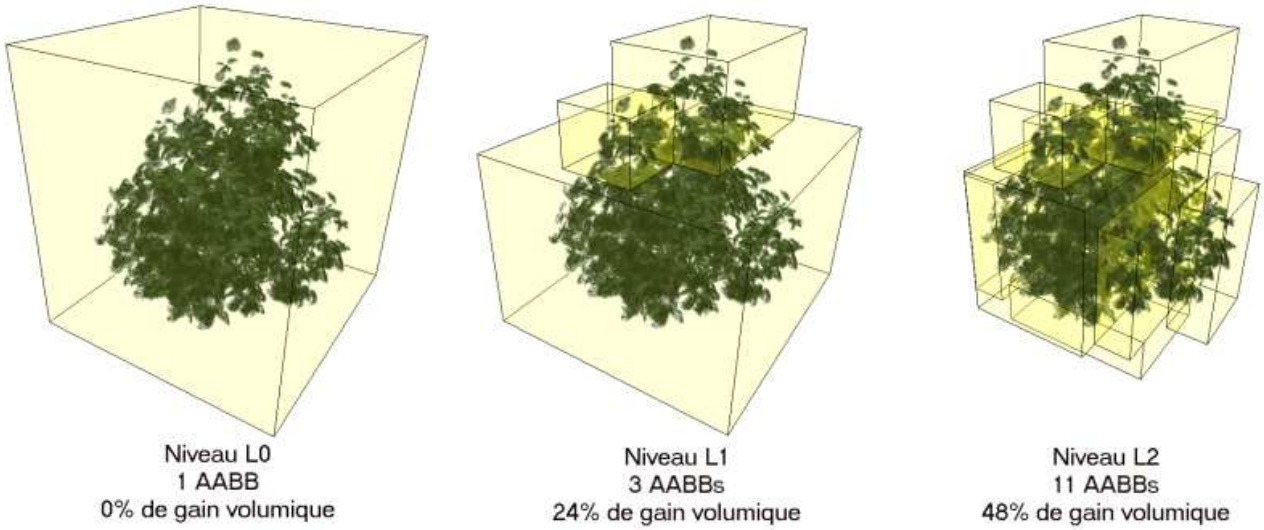


Fig. 25: Ribes: Modèle d'un arbrisseau, le tronc est ramifié dès la base.

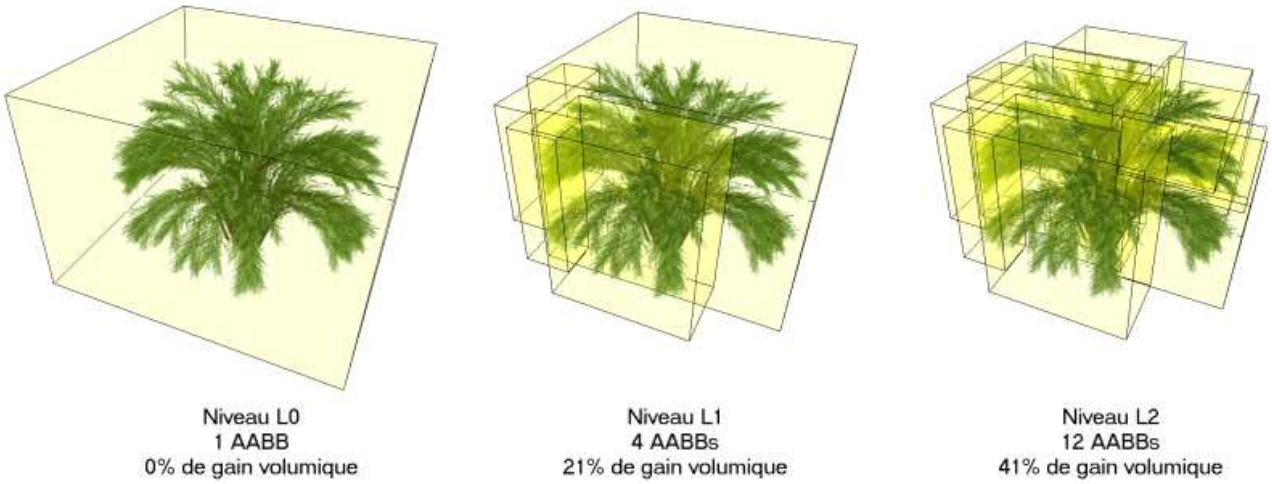


Fig. 26: Phyllo: Modèle pour lequel le vide autour du feuillage est assez important.

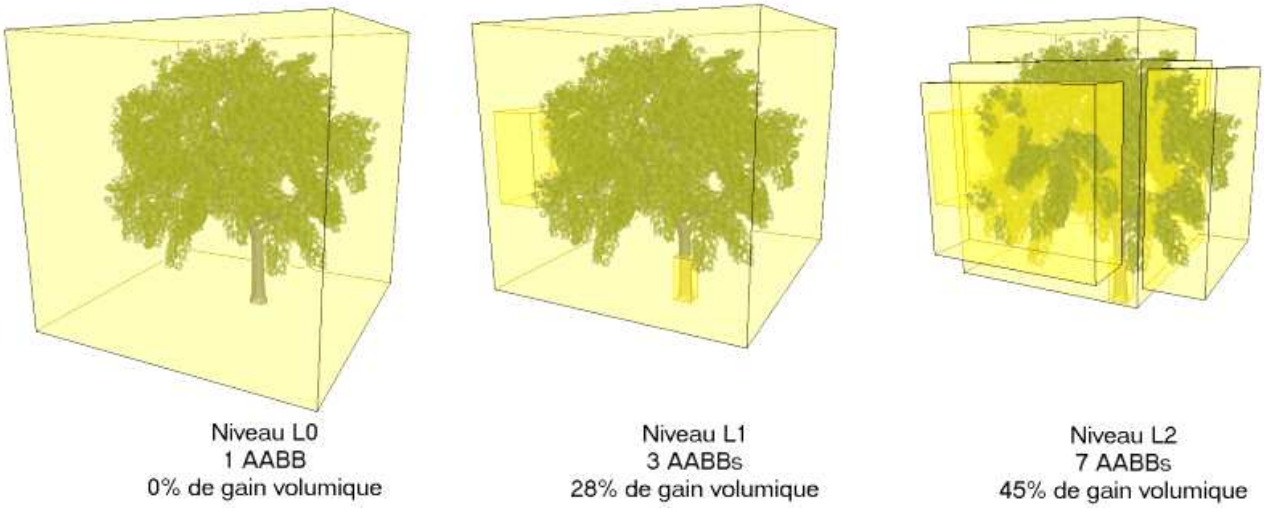


Fig. 27: acer_monsp: cas typique d'un arbre pour lequel le *KD-Tree* est simple (seul le gros volume central est subdivisé au fur et à mesure selon ses 6 côtés).

Pour les figures 25, 26 et 27, le choix du niveau de subdivision L1 n'est pas symétrique. Cela est du à la méthode de sélection automatique du niveau de subdivision qui ne prend pas en compte des critères de qualité visuels, mais seulement des critères quantitatifs (vide à retirer et le nombre de boites maximal pour le faire). Mais cela n'est pas grave, puisque les sous-volumes ne sont pas affichés pour le rendu volumique (ils ne servent qu'à partitionner la texture 3D initiale). Néanmoins, l'inconvénient de découpes non-symétriques, est que des *framerates* assez différents pour des position symétriques de la caméra autour du modèle volumique peuvent être observés.

2 - Autres types de données volumiques

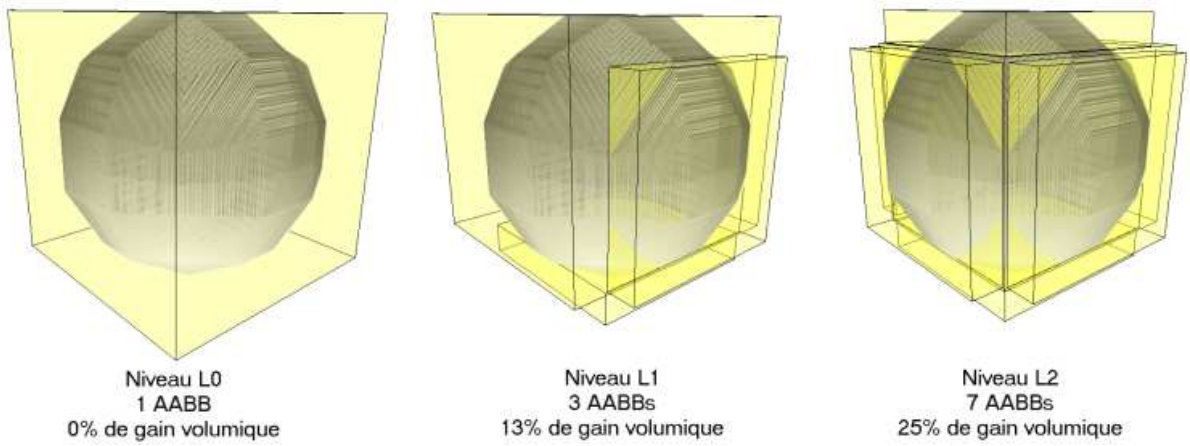


Fig. 28: Une sphère: l'espace vide à l'intérieur ne pourra pas être retiré.

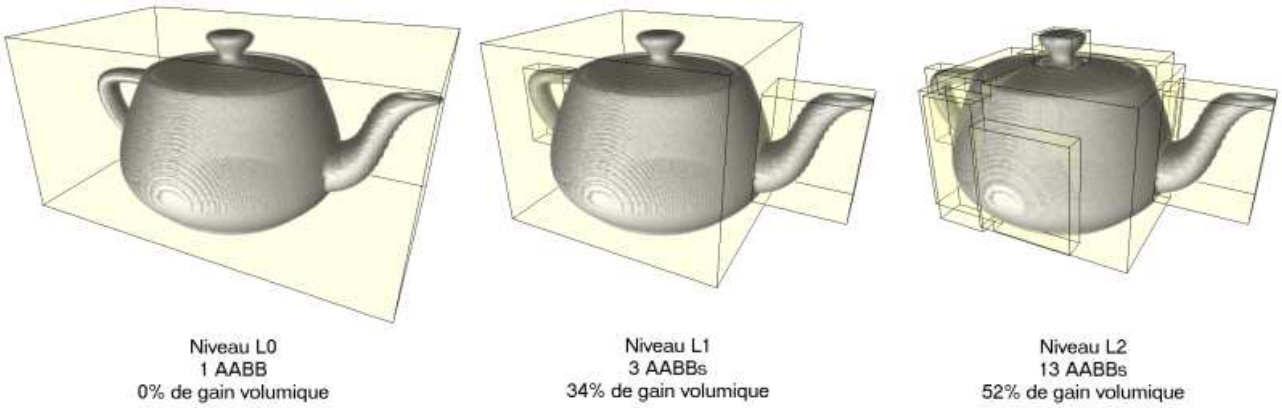


Fig. 29: La *teapot* ou théière.

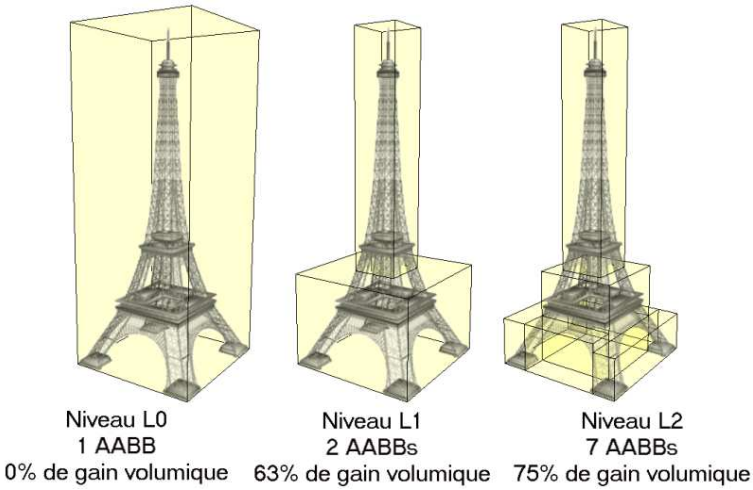


Fig. 30: La tour Eiffel.

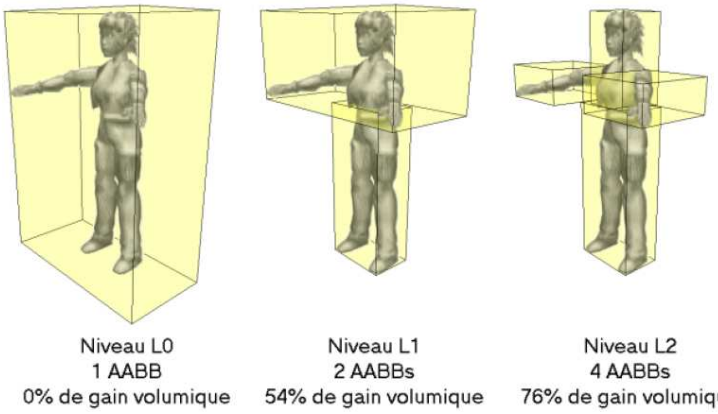


Fig. 31: Personnage: la forme du corps est épousée en un nombre minimal de boites.

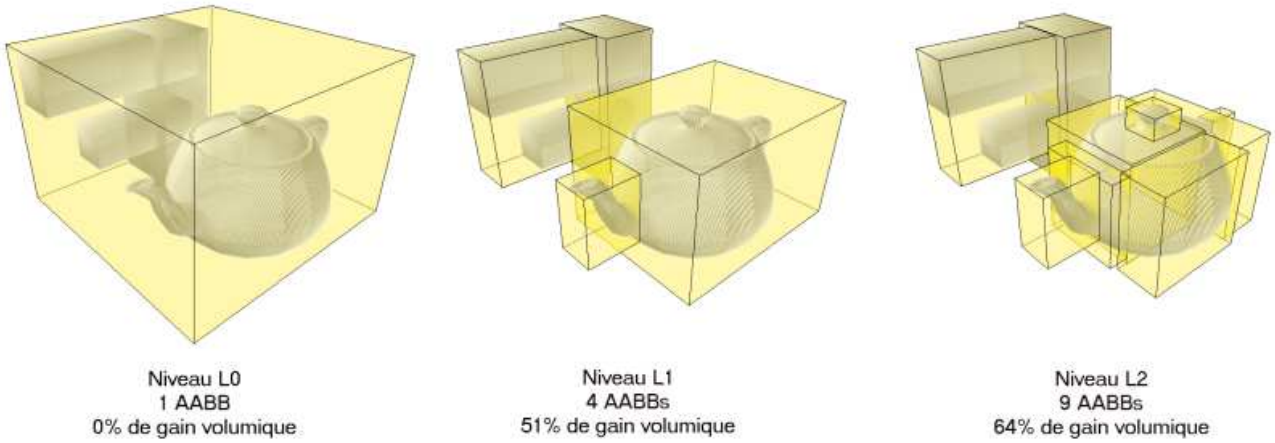


Fig. 32: Mire: notre méthode fonctionne aussi pour des groupes d'objets volumiques (au sein d'une texture 3D).

L'espace vide autour des modèles autres que des arbres ou des plantes, est aussi supprimé avec notre méthode. Mais l'espace vide à l'intérieur de la sphère (figure 28) ou de la théière (figure 29) ne peut pas être supprimé.

III - Points techniques intéressants

A - KD-Tree: construction

Commençons par souligner que notre construction du *KD-Tree* n'est pas orientée dans l'objectif d'avoir des résultats temps-réel (car précalculs, construction statique), mais plutôt orientée délais acceptables pour une très haute qualité, en gros quelques secondes en version *release*. Notre construction est récursive, on divise un sous-volume en 2 et on recommence sur chacun des fils.
Schéma de construction:

- Génération de la liste des plans de découpe possibles selon l'axe le plus long.
- Génération de la liste des coûts associée (en un seul balayage).
- Sélection du meilleure plan (celui qui minimise le coût ou interpolation quadratique pour déterminer un minimum approximé).
- Si un critère d'arrêt est rencontré on arrête la construction de cette branche.
- Sinon découpe effective avec création des 2 fils.

Il y a un *vertex buffer* ou tampon de sommets commun au *KD-Tree* entier. Un *vertex buffer* est un tableau de sommets. Un noeud de l'arbre contient un *index buffer* ou un tableau d'indices correspondant aux indices des sommets des faces qui sont contenues dans le noeud. En fait comme nous travaillons avec des faces triangulaires, trois indices de sommets successifs représentent une face. Par exemple les 3 premiers indices sont les indices des sommets de la face 0, etc... Pour représenter un sous-volume, on doit mémoriser à la fois les indices des faces, mais aussi leur matériel associé (si on veut découper le maillage 3D initial ou si on s'intéresse à des sous-volumes particuliers comme le tronc, le feuillage), c'est pourquoi nous avons décidé d'utiliser un tableau d'*index buffer* pour chaque matériel utilisé. Un noeud contient aussi l'*AABB* associée à l'ensemble des faces qu'il contient.

Notre manière de créer la racine du *KD-Tree* et les 2 fils d'un noeud permet de diminuer le nombre d'étapes nécessaires pour enlever suffisamment de vide. En effet, l'*AABB* utilisée pour la racine du *KD-Tree* est l'*AABB* de l'ensemble de toutes les faces du modèle 3D, et l'*AABB* utilisée pour la création d'un fils, est l'*AABB* calculée lors de la création des *index buffers* des 2 fils (et adaptée pour ne pas avoir de problème de chevauchement). Cela permet de ne plus prendre en compte la zone de vide située à l'extérieure des *AABBs* des 2 fils, mais à l'intérieure de l'*AABB* du noeud parent dans les subdivisons à venir.

Par contre il n'y aura jamais de noeud vide, car un plan de découpe strictement inclus dans l'*AABB* du noeud courant créera forcement 2 fils non vides sinon cela signifierait que le calcul d'*AABB* était faux. Donc notre partition de l'espace n'est pas celle d'un *KD-Tree* habituel, c'est une partition de l'espace contenant des primitives géométriques. La profondeur de l'arbre est optimisée par rapport à un *KD-Tree* qui prendrait en compte les volumes vides, car pour arriver à un volume plein d'une certaine dimension, il faudrait atteindre des profondeurs plus importantes.

Pour la génération de la liste des coûts des plans candidats, on utilise une approximation de l'*AABB* des 2 fils d'un noeud. Cette dernière n'est valide que si les faces sont homogènes et petites (Confère calcul d'*AABBs* dans la partie technique III - E). Notons aussi que nos plans

candidats sont triés par ordre croissant (en fait dès la génération de la liste), et donc nous avons pu faire une légère optimisation lors du balayage: si on trouve une face à gauche d'un plan de découpe, alors elle sera forcément à gauche de tous les plans candidats restants. Cela nous évite de faire un test inutile pour chacun des plans restants.

Pour le calcul du minimum de la fonction de coût, nous avons fait des tests avec des interpolations quadratiques (par morceaux & polynôme de Newton) avec 8, 16, et 32 plans de découpe, mais hélas les résultats n'étaient pas bons. Cela est du au fait que notre fonction de coût peut varier brusquement, aux erreurs numériques dues au petit pas de discrétisation et aussi au fait que l'on utilise une approximation pour déterminer le coût d'un plan candidat. On pourrait faire d'autres essais avec une interpolation basée sur des splines (Cardinal Spline) par morceaux ou des courbes de Bézier (en ne considérant que des segments de courbe cubiques), mais les résultats obtenus sans interpolation avec 64 ou 65 plans découpés uniformément sont suffisamment précis pour notre ensemble de modèles. Remarquons qu'une interpolation linéaire n'apporterait pas d'information supplémentaire pour la recherche du minimum de la fonction de coût, et donc il est inutile d'en faire.

B - KD-Tree: critères d'arrêt

Lors de la construction du *KD-Tree*, une partie très importante est le choix des critères d'arrêts, qui vont déterminer si on subdivise un noeud de l'arbre en 2 fils ou non. Voici la liste des critères d'arrêt généraux testés dans la construction de notre *KD-Tree*. La subdivision du noeud courant n'aura pas lieu si:

- Pas de plan de découpe valide trouvé (il faut qu'un plan permette de couper une tranche d'une épaisseur minimale).
- La profondeur du noeud courant est la profondeur maximale.
- Si le noeud ne contient pas de primitive géométrique (noeud vide).
- Si le volume de l'*AABB* du noeud courant est trop petit.
- Si le gain volumique direct sur le noeud courant avec le meilleur plan de découpe n'est pas assez important.
- Si la densité des 2 futurs fils avec le meilleur plan de découpe serait trop grande (testé, mais pas conservé).

Remarquons que le critère du noeud vide est inutile pour la construction du *KD-Tree* proposée, mais c'est un critère important à ne pas oublier pour les partitions de l'espace qui essaient d'éliminer du vide. En plus, notre toute première version de la construction du *KD-Tree* autorisait les noeuds vides (et peut-être dans des versions futures il y aura à nouveaux des noeuds vides). C'est pourquoi nous avons gardé ce critère.

Les 2 derniers critères d'arrêt sont testés après avoir sélectionné le meilleur plan de découpe, tandis que les autres peuvent être directement testés avant la génération des plans de découpe candidats.

Chaque fois que l'on a un nouveau critère d'arrêt en tête, il faut bien faire attention à le valider, au moins avec son bon sens, car sinon on pourrait obtenir un *KD-Tree* inutilisable. Par exemple, un mauvais critère d'arrêt serait d'avoir atteint un nombre maximal de sous-volumes

autorisés, car le nombre de volumes nécessaires dépend fortement du modèle utilisé, et aussi car notre construction récursive à gauche déséquilibrerait notre *KD-Tree* final (il faudrait alors gérer la construction avec une file de priorité, avec le plus petit index comme choix de tri).

C - KD-Tree: critère d'évaluation

Avant de réfléchir à un critère d'évaluation quelconque, on doit se demander quelles sont les valeurs importantes liées à la construction du *KD-Tree*.
Il y a:

- Le volume total à rendre ou le volume vide total éliminé (en %).
- Le nombre total de feuilles non vides, qui nous donne le nombre maximal de sous-volumes.
- Le temps de construction.
- La profondeur maximale atteinte.

Pour des durées de construction acceptables (quelques secondes) avec une profondeur relativement petite, un critère de qualité pour notre *KD-Tree* est:
gain_volumique(%) / Nb_feuilles_non_vides.

Ce critère traduit l'aptitude à retirer du volume vide avec un petit nombre de sous-volumes, à fortiori un petit nombre de découpes. Il serait aussi intéressant de regarder la valeur du plus petit volume non-vide pour chaque *KD-Tree* construit, pour vérifier que les volumes à rendre ne sont pas trop petits.

D - La séparation des données pour les 2 enfants

Pour notre problème qui consiste à générer un ensemble de sous-volumes à partir d'un volume de base, on doit être capable de créer 2 nouveaux noeuds enfants (fils gauche et fils droit) à partir d'un noeud et d'un plan de découpe. Le problème est qu'il y a beaucoup de faces coupées par le plan de découpe que l'on doit conserver pour les 2 fils, pour ne pas perdre d'information sur les coordonnées de textures finales.

La seule optimisation que l'on a faite est de comptabiliser les faces qui sont dans le plan de découpe une seule fois si:

- Il y a seulement des faces dans le plan de découpe, donc on ne conservera les faces que pour le fils le plus petit (si on prenait l'*AABB* découpée directement) dans un volume élargi sur la gauche ou la droite du plan de la valeur de la dimension minimale.

- Il y a seulement les faces incluses dans le plan de découpe pour un fils et dans l'autre fils il y a au moins une face supplémentaire. Dans ce cas, on ne gardera les faces que pour le fils qui contient d'autres faces.

Cette optimisation ne sera pas intéressante pour notre manière de construire le *KD-Tree*, car le fait de recalculer des *AABBs* après chaque découpe enlève toute possibilité d'enlever directement un morceau de vide de l'*AABB* courante. Néanmoins, cette optimisation nous a servi pour nos premières versions pour lesquelles on pouvaient avoir des noeuds vides. Et son surcoût est négligeable, donc nous l'avons conservée (peut-être qu'on réutilisera une méthode qui permet de

prendre en compte des noeuds vides dans le futur).

Comme nous n'avons pas le droit de déplacer le plan de découpe pour réduire le nombre de faces partagées entre les 2 fils, puisque le plan de découpe a été choisi de manière optimale pour le gain volumique, aucune autre optimisation pour réduire le nombre de faces partagées n'est possible.

Pour la répartition des faces entre les 2 noeuds enfants, on regarde pour chaque face, d'abord si elle se trouve dans le plan de découpe et on l'ajoute aux 2 fils (et on incrémente un compteur de faces dans le plan), ensuite si elle est à gauche et on l'ajoute au fils gauche, puis si elle est à droite et on l'ajoute au fils droit et finalement si elle est coupée par le plan et alors on l'ajoute aux 2 fils. A la fin de la répartition, on compare le nombre total de faces dans un fils et le nombre total de faces dans le plan de découpe pour vider l'*index buffer* du fils si nécessaire (confère optimisation précédente). Remarquons tout de même que le test « la face est dans le plan de découpe » est effectué avant ceux « la face est à gauche » ou « la face est à droite » car le premier test est un sous-cas des 2 autres.

E - Le calcul des AABBs filles lors d'une découpe

Lorsqu'on fait la séparation des données pour les 2 enfants, on met à jour en parallèle les *AABBs* associées. S'il y a des faces découpées par le plan de découpe, il faudra ajuster les *AABBs* sur ce plan (et sur les plans de découpe précédents) pour ne pas avoir de chevauchement. Pour cela, on gardera l'intersection d'*AABBs* entre l'*AABB* calculée à la fin de la séparation des données et la partie de l'*AABB* parent associée (confère figure 33).

AABB_A_GAUCHE = AABB_CALCULEE _A_GAUCHE ∩ PARTIE_GAUCHE_AABB_PARENT

AABB_A_DROITE = AABB_CALCULEE _A_DROITE ∩ PARTIE_DROITE_AABB_PARENT

S'il n'y a aucune face dans une *AABB*, alors on prendra la partie de l'*AABB* parent associée. De toute façon si un noeud contient une *AABB* vide, il ne sera pas utilisé par la suite pour le rendu volumique.

AABB_A_GAUCHE_VIDE = PARTIE_GAUCHE_AABB_PARENT

AABB_A_DROITE_VIDE = PARTIE_DROITE_AABB_PARENT

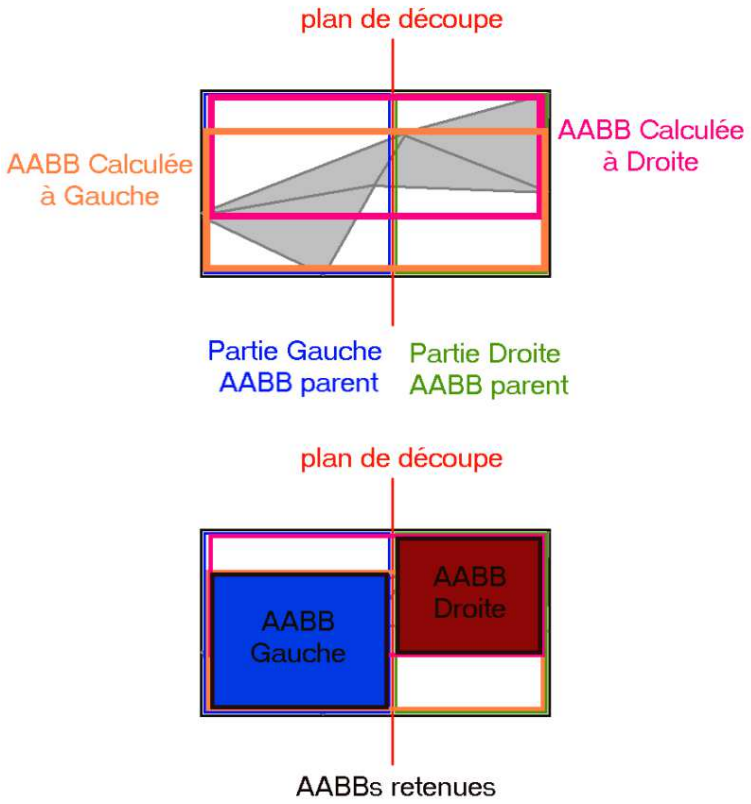


Fig. 33: Illustration 2D de l'ajustement des *AABBs* filles lors de la découpe d'un maillage composé de faces triangulaires.

Cas particulier des *AABBs* plates que nous avions avec nos premières méthodes de construction du *K-Tree*: Pour le cas d'une *AABB* plate (donc non vide) selon le plan de découpe, on ajuste l'*AABB* pour que l'*AABB* ne soit pas plate (son volume ne sera pas nul). L'ajustement se fera à gauche pour le fils gauche (modification du vecteur Min de l'*AABB*) et à droite pour le fils droit (modification du vecteur Max de l'*AABB*). En fait on « élargit » l'*AABB* de la valeur de la dimension minimale pour découper. On fait cela, car un plan de découpe valide coupe des tranches d'épaisseur d'au moins cette dimension minimale. Et cela nous assure qu'il n'y aura pas de chevauchement d'*AABBs* suite à cet élargissement. La valeur de la dimension minimale est petite, mais on fait cela pour ne pas avoir d'*AABB* plate, ce qui facilite la validation visuelle de la position des *AABBs* des sous-volumes.

Remarquons que notre manière de calculer les *AABBs*, nous assure que l'on n'oubliera aucun morceau de face car on raisonne sur des faces entières lors du calcul des *bounding boxes* englobantes. Mais ce n'est pas optimal, car pour perdre plus de volume, il faudrait calculer les 2 points d'intersection entre chaque face triangulaire coupée et le plan de découpe (confère figure 34).

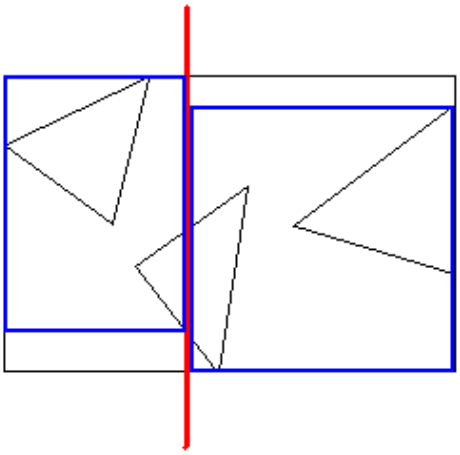


Fig. 34: Illustration 2D montrant qu'un calcul d'*AABBs* (en bleu) optimal nécessite de calculer les points d'intersections entre les faces triangulaires et le plan de découpe (en rouge).

Mais calculer les points d'intersection pour l'ensemble de toutes les faces coupées au moment de la génération de liste des coûts associée à la liste des plans de découpe candidats est assez coûteux. En fait, la perte volumique due à l'imprécision de la première méthode est acceptable si les faces ne sont pas trop grandes, pour un temps de calcul beaucoup plus court. Une solution intéressante consiste à faire le calcul de sélection du meilleur plan de découpe en utilisant la première méthode de calcul des *AABBs*, mais pour faire la véritable séparation des données on utilise la deuxième méthode. Cela permet d'avoir des volumes englobants découpés de manière optimale et un temps de calcul de l'ordre de quelques secondes.

F - Le calcul de la densité d'un sous-volume

Pour calculer la densité d'un noeud (ou de l'*AABB* associée), nous faisons simplement le ratio entre la somme des volumes équilibrés des *AABBs* des faces contenues dans le noeud et le volume équilibré de l'*AABB* du noeud. Un volume équilibré (*balanced volume*) est le volume d'une *AABB*, agrandi d'un epsilon strictement positif sur la gauche et sur la droite des 3 axes OX, OY et OZ. Epsilon doit être très petit (<< 1), et nous avons choisi epsilon = 0,2 pour nos tests.

L'utilisation de volumes équilibrés avec notre epsilon permet de prendre en compte les faces qui ont une normale (normalisée) de la forme $[1, 0, 0]^T$ ou $[0, 1, 0]^T$ ou $[0, 0, 1]^T$. En effet, pour de telles faces, l'*AABB* associée est plate et donc son volume est nul. C'est pourquoi un calcul de densité qui ne tiendrait pas compte de ce type de face serait forcement faux.

Il ne faut pas perdre de vu que ce calcul de densité n'aura du sens que pour des faces relativement petites et de tailles voisines. Le problème majeur de ce calcul est qu'il est basé sur l'*AABB* de chaque face, or cette dernière varie en fonction de la position spatiale de la face (ses angles d'inclinaison selon les 3 axes). Plus une face sera grande, plus cette variation peut être importante. Une autre solution pour le calcul de densité serait d'utiliser seulement le nombre de faces divisé par le volume englobant (scalaire). Dans ce cas chaque face aurait la même influence,

ce qui est aussi discutable car il faudrait justifier le fait que 2 faces de taille très différente aient la même importance.

En fait, le calcul de densité pour une représentation volumique aurait plus de sens, car il suffirait de compter le nombre de voxels vides (opacité à 0 ou par rapport à un seuil). Ensuite la valeur de la densité en voxels non-vides peut être obtenue par la formule:
D = (#Voxels - #EmptyVoxels) x VolVoxel / (#Voxels x VolVoxel)
= (#Voxels - #EmptyVoxels)/#Voxel

G - Choix automatique d'un niveau de subdivision à partir du KD-Tree

Le *KD-Tree* nous offre la possibilité de choisir un ensemble de sous-volumes en le parcourant, et selon certains critères, pour savoir si pour un noeud donnée, on va continuer le parcours dans les 2 branches filles, ou seulement dans une branche ou carrément s'arrêter et générer un seul volume.

Nous avons pensé à 2 critères simples pour effectuer cette sélection pour un noeud donné, le pourcentage d'espace vide à retirer et le nombre de boites maximal pour le faire. Ainsi, en partant de la racine du *KD-Tree*, nous générons un ensemble de sous-volumes qui seront utilisés par la suite pour le rendu volumique. Nous utilisons un algorithme récursif, avec des critères d'arrêt et de sélection.

Génération_Subdivision(Noeud, %Gain_en_Espace_vide, #Vol, liste_vol)
{
si critère d'arrêt rencontré **alors**
ajouter le volume (AABB) du noeud à liste_vol ;
sortir (*return*) ;

si en générant toutes les feuilles du Noeud le gain est suffisant
et le nombre de feuilles + le nombre de volumes dans liste_vol <= #Vol **alors**
ajouter toutes les feuilles du Noeud à liste_vol;
sortir ;

si le fils gauche semble meilleur **alors**
{
si le fils gauche permet de gagner assez d'espace vide
et son nombre de feuilles + le nombre de volumes dans liste_vol < #Vol **alors**
{
ajouter les feuilles du fils gauche à liste_vol ;
ajouter le volume (AABB) du fils droit à liste_vol ;
sortir ;
}

si le fils droit permet de gagner assez d'espace vide
et son nombre de feuilles + le nombre de volumes dans liste_vol < #Vol **alors**
{
ajouter les feuilles du fils droit à liste_vol ;

```
                    ajouter le volume (AABB) du fils gauche à liste_vol ;
                    sortir ;
            }

            #VolG = min(#feuillesFG,#Vol-1);
            #VolD = min(#feuillesFD,#Vol-#VolG);
            %GainPossibleG = min(%GainPossible(Noeud.FG), %Gain_en_Espace_vide);
            %GainRestantD = min(%GainPossible(Noeud.FD),%Gain_en_Espace_vide-%GainPossibleG);

            Génération_Subdivision(Noeud.FG, %GainPossibleG, #VolG, liste_vol);
            Génération_Subdivision(Noeud.FD, %GainRestantD, #VolD, liste_vol);
    }
sinon
    {
            // pseudo-code symétrique (en inversant le fils droit/D et le fils gauche/G)
    }

}
```

Pour les critères d'arrêt, nous avons pris:
- le noeud n'est pas valide
- pas de *KD-Tree* généré (car cela est nécessaire pour calculer le pourcentage d'espace vide retiré)
- le noeud est une feuille (car il ne sera pas possible de gagner plus de vide)
- si en allant à la profondeur suivante, le nombre maximal de sous-volumes autorisé serait dépassé
- l'espace à retirer est trop petit (<= 3%)

Pour déterminer si un fils est meilleur qu'un autre nous avons pris le maximum du ratio **GainVolFils (%) / #feuillesNonVidesFils** qui a tendance à favoriser la meilleure branche, celle qui permet d'avoir un bon gain volumique pour un petit nombre de volumes.

H - Implémentation d'un Ray-Casting sur GPU avec DirectX 9 et HLSL

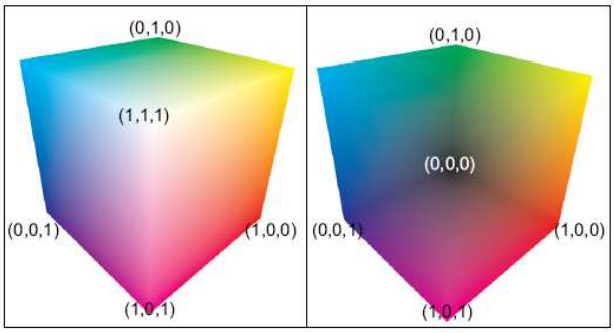
Beaucoup de personnes ont choisi de faire du rendu volumique en utilisant la technique du Ray-Casting, qui consiste à lancer un rayon pour chaque pixel à partir du point de vu, et de mélanger les couleurs RGBA de chaque voxel (en utilisant l'opacité stockée dans le canal Alpha). Le parcours le long du rayon peut se faire en *back-to-front* et en *front-to-back* (la mise à jour de la couleur courante d'un pixel sera alors différente).

Nous voulions comparer les résultats de notre méthode avec ceux obtenus par Ray-Casting. De plus je voulais apprendre la programmation de vertex et de pixel *shaders*. C'est pourquoi nous avons décidé d'implémenter un Ray-Casting sur GPU. Pour cela je me suis servi des tutoriaux que j'ai pu trouver sur le Net, ainsi que de l'article de J. Krüger et de R. Westermann [KRÜGER and WESTERMANN 2003]. L'idée de base est de rendre 2 fois un parallélépipède rectangle qui correspond à la *bounding box* de l'objet associé à la texture 3D, une fois en *back faces culling*, une autre en *front faces culling* (confère figure 35). Il faut spécifier les coordonnées

adéquates de la texture 3D pour chaque sommet de la *BBOX* en tant que couleurs, afin qu'elles soient automatiquement interpolées (au niveau du pixel *shader*). Un *shader* est un petit programme chargé sur la carte graphique qui va remplacer une partie de la chaîne de rendu du pipeline graphique (un vertex *shader* va s'occuper des transformations et des calculs d'illumination, tandis qu'un pixel *shader* déterminera la couleur à afficher pour chaque pixel). Pour du *back faces culling*, seules les *front faces* seront rendues et cela nous donnera les points de départ des rayons dans l'espace des textures 3D. Au contraire, pour du *front faces culling*, on obtiendra les points d'arrivée. Ainsi, sans les passes d'optimisation, le Ray-Casting peut être réalisé sur GPU en 2 passes. La première passe consiste à faire du rendu dans une texture des points d'entré (ou de sortie). Puis dans la seconde passe les points de sortie (ou d'entrée) sont déterminés, ce qui permet de calculer la direction normalisée pour itérer (boucle *for* ou *while*) dans la texture 3D et rendre le pixel final.

Pour ce qui concerne l'API DirectX 9, en fait il y a trois choses importantes à dire:
- Garder un pointeur sur la surface de rendu courante, que l'on nomme *backbuffer*
- Créer une texture utilisée pour le rendu (comme surface de rendu) à la passe 1
- Utiliser la texture résultante dans la passe 2 (set texture) pour pouvoir accéder à cette dernière dans le pixel *shader* et sélectionner la surface de rendu pointée par *backbuffer*

Pour ce qui concerne le *back/front faces culling*, on peut l'activer à partir de DirectX ou directement dans la technique du fichier effect (*.fx*). Le Ray-Casting que j'ai implémenté est simple, dans le sens où je n'ai pas utilisé de structure hiérarchique pour éviter de parcourir des zones vides dans le volume à rendre (pas assez de temps). Le code du *shader* HLSL (High Level Shading Language) est accessible en ANNEXES (VI - D - 2 - ii).



Rendering front faces (left) and back faces (right) of the volume bounding box in order to generate ray directions and texture coordinates of first ray intersection points.

Fig. 35: Ray-Casting sur GPU [KRÜGER and WESTERMANN 2003]:
Génération des points d'entré et de sortie pour chaque rayon.

IV - Conclusion: les résultats obtenus et les perspectives résultantes

A - Résumé du travail effectué pendant le stage

Pendant ce stage nous avons élaboré une méthode pour optimiser le rendu volumique basé sur des tranches de texture 3D de modèles volumiques d'arbre ou de plante. Cette optimisation ne concerne que la suppression d'espace vide et ne prend pas en compte les optimisations possibles dues à la visibilité. En utilisant une structure de données hiérarchique, un *KD-Tree*, pour faire une partition des sous-volumes non-vides des données volumiques initiales (en fait une hiérarchie de boites englobantes ou *Bounding Volume Hierarchy* car nous ajustons les *AABBs*), nous sommes capables de générer 3 niveaux de subdivision qui épousent de plus en plus la forme des données avec un petit nombre de sous-volumes (contrairement aux méthodes existantes) et qui permet un gain volumique considérable.

Pour la construction du *KD-Tree*, nous avons testé plusieurs heuristiques et techniques pour la génération des plans candidats, pour la sélection du meilleur plan et pour la mise à jour des noeuds fils. Nous avons retenu la génération de 65 plans de découpe répartis de manière uniforme et régressive selon l'axe le plus long d'une *AABB* associée à un noeud. La sélection du meilleur plan se fait en minimisant le nouveau volume à rendre déterminé grâce au volume des *AABBs* des 2 fils associés à un plan de découpe pour un noeud donné. Si aucun critère d'arrêt n'est rencontré, la découpe du noeud courant a effectivement lieu et dans ce cas les *AABBs* filles sont ajustées et stockées dans les noeuds fils adéquates. La qualité du *KD-Tree* final est en partie due aux critères d'arrêt (épaisseur minimale d'une tranche, volume minimal, gain volumique direct minimal) qui permettent d'obtenir un petit nombre de sous-volumes et répartis de manière assez symétrique.

Ce *KD-Tree* est ensuite utilisé pour sélectionner 3 niveaux de subdivision. Le niveau de subdivision le plus adapté en fonction de la distance du centre du modèle à la caméra est choisi de manière à optimiser le *framerate*. La transparence est assez bien gérée grâce à une bonne approximation du tri des *AABBs*. Cette approximation permet de ne pas avoir de surcoût important pour le tri.

Nous avons soumis un article à VRST 2007, et cela a été ma première expérience dans la rédaction d'un article. A la fin du stage, un Ray-Casting non-optimisé a été implémenté sur GPU en utilisant DirectX9 et HLSL. Cela constitue ma première expérience en programmation de carte graphique (vertex et pixel *shaders*).

B - Amélioration du framerate

L'objectif principal de notre méthode était d'améliorer le *framerate*, le nombre d'images affichées par seconde, pour augmenter les possibilités d'interactivité. Qu'en est-il?

En ANNEXES (VI - C - 2), les résultats concernant le *framerate* montrent que pour un

affichage de la texture 3D initiale (en résolution 256x256x256, sans calcul d'ombre et sans gestion de la transparence) dans une fenêtre 800x600, de telle sorte qu'elle prenne le maximum d'espace disponible sur la fenêtre, le *framerate* augmente lorsque le niveau de subdivision choisi augmente. Comme on pouvait s'y attendre, le *framerate* augmente proportionnellement au gain volumique. C'est-à-dire que plus le gain volumique est important, plus il augmente.

Mais le gain du *framerate* n'est pas idéal. Dans un cas parfait, si le *framerate* F était obtenu pour l'affichage d'un seul volume, alors pour la subdivision Li avec un gain volumique G, un *framerate* F' = F/(1-G) serait attendu. Par exemple pour G = 50%, F' = 2.F (G = 75%, F' = 4.F). D'une manière générale le nouveau *framerate* obtenu F' est inférieur à F/(1-G). Il y a trois principales raisons. La première est que le *framerate* obtenu dépend du point de vu, car selon certaines directions il y aura moins ou beaucoup moins de voxels à traverser que pour la subdivision 0 avec un seul volume. Par exemple pour un arbre de type prunus avec une subdivision en 2 sous-volumes, un pour le tronc et un pour le feuillage, pour la vu de dessus le gain sera inférieur que pour une vue de côté (en comparant avec la vue correspondante sur la texture 3D initiale). En effet un *framerate* de 34 fps est conservé pour la vue de dessus, tandis que l'on passe de 25 fps à 34 fps pour une vue de côté (ce qui correspond au *framerate* idéal, car il y a un gain volumique de 26%). La deuxième raison est le surcoût du à la gestion des sous-volumes, mais cela ne se ressent que lorsque le modèle est éloigné de la caméra. La troisième raison est due aux accès mémoires qui ne sont pas optimisés pour des plans de vue qui ne sont pas parallèles aux axes, car dans ces cas là il y a plus d'opération pour aller chercher des blocs de textures qui ne sont pas dans le cache.

En se basant sur les variations du *framerate* de chaque niveau de subdivision, nous avons établi un procédé de passage automatique d'un niveau de subdivision à un autre pour optimiser le *framerate* pour tous les points de vue. Ainsi nous avons rempli notre objectif d'augmenter le *framerate* pour la texture 3D initiale. Mais si nous voulons gérer la transparence correctement et si nous voulons faire des calculs d'ombre, le *framerate* peut-il être encore amélioré?

La réponse est oui, mais l'amélioration du *framerate* est moindre, car le *framerate* de départ est moindre aussi (cas des ombres). En ANNEXES, dans la partie C - 3 - *Gestion de plusieurs boites*, nous disons que pour ne pas avoir de surcoût du au calcul des ombres et à la gestion de la transparence pour les différents niveaux de subdivision affichés (sur leur zone d'existence) il faut un nombre de sous-volumes inférieur à 13 pour L2 et un nombre de sous-volumes inférieur à 7 pour L1. Et nos *KD-Trees* construits permettent en général de respecter ces contraintes. Ce petit nombre de sous-volumes maximal, traduit le fait que plus un modèle volumique est éloigné de la caméra, plus le niveau de mipmaps choisi sera élevé et donc le coût du à l'affichage diminue rapidement avec l'éloignement (d'où l'importance progressive du nombre de sous-volumes avec l'éloignement). Signalons tout de même que nous avons du adapter notre procédé de passage d'un niveau de subdivision à un autre lorsque la gestion de la transparence ou le calcul d'ombre sont activés. Ainsi nous avons réussi à améliorer le *framerate* dans le cas du calcul des ombres et de la gestion correcte de la transparence.

C - Qualité de la partition

Nos heuristiques sont bien adaptées à la morphologie des arbres, et la partition des voxels non-vides obtenue est assez bonne, dans le sens où elle contient peu de sous-volumes, et que ces derniers sont généralement disposés de manière symétrique. Et le temps de calculs (quelques

secondes) pour la construction du *KD-Tree* est négligeable pour une approche statique comme la notre.

Mais la partition ne permet pas d'enlever les espaces vides situés à l'intérieur d'un modèle, cela à cause de notre fonction de coût. Par exemple, les résultats pour des objets simples tels qu'une boite vide ou une sphère vide, ne sont pas bons, dans le sens où la grande quantité de vide à l'intérieur de la boite ou de la sphère ne peut pas être retirée. Pour la boite cela s'explique par le fait qu'il n'y ait pas de plan de découpe initial qui permette d'enlever du vide. Donc dans le cas d'une boite, l'heuristique du gain volumique direct ne peut pas fonctionner et le vide à l'intérieur de la boite ne pourra pas être retiré. Il faudrait utiliser l'heuristique SAH, qui va choisir des plans près des côtés de la boite et donc le vide ne sera pas éliminé du premier coup, mais en plusieurs étapes...

Le plus important, c'est que nous avons résolu le problème de base, qui consistait à enlever du vide pour nos modèles d'arbre ou de plante, qui ne contiennent pas de zones de vide importantes à l'intérieur (dans le feuillage et dans le tronc).

D - Applications

Notre méthode peut être appliquée à tout rendu volumique basé sur des textures 3D et le *KD-Tree* construit peut être aussi utilisé comme structure hiérarchique pour le Ray-Casting. Elle nécessite de pré-calculer statiquement un *KD-Tree* (donc pas de mise à jour nécessaire par la suite), ainsi que les 3 niveaux de subdivision associés. Le *KD-Tree* pouvant être construit soit à partir d'un maillage 3D, soit directement à partir d'une représentation en voxels en utilisant l'information sur l'opacité.

Notre méthode permet d'améliorer la vitesse d'affichage, et ouvre des perspectives intéressantes dans le rendu de forêt. Par exemple, nous pouvons grouper des maillages d'arbres de manière « naturelle », puis voxeliser l'ensemble pour finalement appliquer notre méthode.

E - Limitations

Il y a 2 principales limitations pour notre méthode. La première est le fait que le *KD-Tree* construit ne permet pas de retirer du vide à l'intérieur d'objets simples comme une sphère ou une boite. Cela est du à notre fonction de coût qui n'est pas capable de trouver un meilleur plan si aucun plan de découpe ne permet de retirer directement du vide. Néanmoins ce n'est pas un problème pour les modèles d'arbre ou de plante qui ne possèdent pas de zones vides importantes à l'intérieur en général. En plus il n'est pas évident qu'une méthode qui permettrait d'enlever du vide à l'intérieur des modèles volumiques, le ferait avec peu de sous-volumes comme nous le faisons.

La deuxième limitation est que notre méthode dépend de constantes « magiques » (critères d'arrêt) qu'on ne peut pas être en mesure de justifier théoriquement et qui devront certainement être adaptées pour d'autres ensembles de modèles.

F - Travaux futurs

La construction du *KD-Tree* pourra être améliorée de manière à retirer des espaces vides à l'intérieur des représentations volumiques tout en continuant d'obtenir un petit nombre de volumes pour un niveau de subdivision donné. En particulier, nous pourrons explorer l'approche des subdivisions successives pour le choix du meilleur plan de découpe. Cela permettra d'utiliser notre méthode sur une plus grande variété de modèles et sera utile pour l'optimisation du *framerate*.

L'utilisation de la mémoire devra être optimisée: pourquoi mémoriser les pixels « vides » dans notre texture 3D? Il faudra chercher à réduire la taille mémoire nécessaire pour stocker les textures 3D utilisées avec leur différents niveaux de *MIP-map*. On pourra étudier l'algorithme de compactage de texture présenté dans [KÄHLER et al. 2003]. Il serait aussi intéressant d'optimiser les accès mémoires à notre texture 3D [WEISKOPF et al. 2004], car sans cela il est possible d'avoir des variations de la vitesse d'affichage en fonction de la position de la caméra (quand le plan de vue n'est plus parallèle aux axes, le frame rate a tendance à diminuer), ce qui est inacceptable pour de la visualisation interactive.

Il serait intéressant d'améliorer le tri d'*AABBs* pour augmenter la qualité du rendu final (meilleure gestion de la transparence) et aussi pour faire un état de l'art des techniques actuelles de tri d'*AABBs* (car c'est un problème intéressant). Le rendu utilisant les *AABBs* pourra être aussi fait directement à partir de plusieurs sous-volumes (sans rendre les volumes séparément), et donc le tri d'*AABBs* ne sera plus nécessaire. On pourra aussi vérifier s'il n'y a pas d'optimisation possible pour nos calculs d'ombre.

Nous n'avons fait aucune optimisation concernant les voxels non-vides, mais cachés (*Occlusion Clipping* [LI et al. 2003]). Or beaucoup de voxels non-vides ne participent pas au rendu de l'image finale. Donc sans prendre en compte ces voxels, nous pourrions améliorer de manière considérable la vitesse d'affichage sans pour autant perdre en qualité.

Nous pourrions mettre à jour les bibliothèques actuelles pour que notre algorithme de voxelisation nous permette de générer des textures 3D en résolution 512^3, car il est important d'étudier les résultats obtenus pour des résolutions plus grandes que 256^3.

Pour ce qui est du rendu de forêt, il serait intéressant de faire des expériences sur des groupes d'arbres, sur des données volumiques beaucoup plus grandes (qui devraient alors être subdivisées en blocs pour tenir sur la mémoire de la carte graphique)...

Notre méthode devra être comparée avec l'optimisation du rendu basé sur des tranches de textures 3D proposée par [LI et al. 2003]. Il serait aussi intéressant de comparer les résultats obtenus avec ceux d'un Ray-Casting à l'état de l'art actuel, pour voir les avantages et inconvénients de chaque méthode. D'où nos travaux sur le Ray-Casting sur GPU.

V - Planning réalisé

le lundi 5 février:	arrivée à l'aéroport de Beijing et présentations
6 – 8 février:	lecture bibliographique d'introduction au rendu volumique
9 février – 1 mars:	formation à DirectX 9 (tutoriels, affichage de plantes au format ***.x*** et manipulation du vertex buffer) et fermeture du LIAMA pendant 10 jours (festival du printemps)
2 mars – 9 mars:	étude de la construction des *KD-Trees* pour isoler les volumes vides avec SAH (*Surface Area Heuristic*): recherche bibliographique ; l'objectif étant l'élaboration d'un algorithme de construction automatique de l'arbre des volumes vides et de l'arbre des volumes pleins à partir d'un fichier au format ***.x***. écriture d'un document spécifiant la construction du *KD-Tree* (arbre des volumes vides et arbre des volumes pleins) ; cela comprendra un algorithme détaillé, mais clair ; les critères d'arrêt de la récursion (pour la construction d'un arbre) devront être justifiés ; bien réfléchir au contenu des noeuds, en particulier la position de toutes les boites doit être connue ; détails sur le format utilisé
12 mars – 6 avril:	implémentation en C++: les résultats doivent être visuels: les volumes doivent être visibles => réfléchir à d'éventuels paramètres intéressants pour la visualisation
9 – 13 avril:	tests et validation
16 avril – 18 mai:	niveaux de subdivision: déterminer les critères de passage d'un niveau de subdivision à l'autre et implémenter le passage d'un niveau de détail à un autre ; les résultats doivent être visuels: on doit pouvoir voir de manière interactive le nombre de volumes qui varie en fonction de la position de la camera (du niveau de détail) ; fermeture du LIAMA pendant 1 semaine (fête du travail)
21 mai – 14 juin:	rédaction d'un *short paper* (soumission à VRST 2007)
14 juin:	retour en France
15 juin – 13 juillet:	étude de HLSL et de l'utilisation des techniques sur DirectX 9 ; implémentation d'un Raycasting sur GPU ; rédaction du rapport de PFE
16 juillet – 31 juillet:	tests Raycasting ; présentation du travail effectué (séminaire) ; rédaction du rapport de PFE

VI - ANNEXES

A - DirectX

1 - Généralités

Pour ce qui concerne les généralités sur l'utilisation de DirectX 9, en particulier la manipulation du *device*, le rendu de maillages 3D, l'utilisation des fichiers au format *.x*, les différentes étapes nécessaires pour afficher un maillages 3D ou des données volumiques sous la forme d'une texture 3D, la documentation de DirectX9 SDK est assez complète, en plus il y a des tutoriaux pour apprendre les bases.

Un cours intéressant avec des codes source commentés (on peut en particulier trouver des exemples pour créer le *vertex buffer* et l'*index buffer* d'objets simples comme un cube, une sphère):
http://mathinfo.univ-reims.fr/image/PageBuilder.php?dir=DirectX&show=Tutorial&menu=tutorial&act=1

Autres sources:
http://www.gamedev.net/reference/programming/

DirectX SDK: http://www.microsoft.com
msdn:http://msdn2.microsoft.com/fr-fr/default.aspx

2 - GPU programming et HLSL (High Level Shading Language)

nVidia: http://www.nvidia.com (http://news.developer.nvidia.com/events/index.html)
ATI: http://www.ati.com
Matrox: http://www.matrox.com
3Dlabs: http://www.3dlabs.com
SiS: http://www.xabre.com

HLSL:
http://en.wikipedia.org/wiki/High_Level_Shader_Language
http://www.daionet.gr.jp/~masa/rthdribl/
http://www.two-kings.de/tutorials/dxgraphics/dxgraphics16.html

GPU Ray Casting tutorial: http://www.daimi.au.dk/~trier/?page_id=98

Outils d'aide au développement de *shaders*:
GPUShaderAnalyser (pas génial).
ATI RenderMonkey 1.62 (les exemples fournis sont assez formateurs)

3 - Utilisation de textures volumiques

Les textures volumiques de mon projet sont sous le format *.dds*. Voici quelques explications en anglais:

L'extension .dds:

The DirectDraw Surface (.dds) file format is used to store textures and cubic environment maps, both with and without mipmap levels. This format can store uncompressed and compressed pixel formats, and is the preferred file format for storing DXTn compressed data. This file format is supported by the DirectX Texture tool (DirectX Texture Editor (Dxtex.exe)), as well as some non-Microsoft tools, and by the D3DX library.

DDS File Layout for Volume Textures:

A volume texture is an extension of the standard texture for Direct3D 9, and can be defined with or without mipmaps. If a file contains a volume texture, DDSCAPS_COMPLEX, DDSCAPS2_VOLUME, DDSD_DEPTH, and dwDepth should be set. For volumes without mipmaps, each depth slice is written to the file in order. If mipmaps are included, all depth slices for a given mipmap level are written together, with each level containing half as many slices as the previous level with a minimum of 1. Volume textures do not support DXTn compression as of Direct3D 9.

B - Tests construction KD-Tree sur des maillages 3D

Les résultats suivants ont été obtenus avec un Pentium IV 2,40 GHz duo core et une GeForce 8800 GTX.

1 - 65 plans de découpe uniformes selon l'axe le plus long

Nom Modèle (.x)	Nombre de sommets	Nombre de faces	Nombre de feuilles non-vides F	Gain volumique G (% initial)	Prof Max	Temps construction en secondes	Ratio G/F
picea	16 328	21 096	9	50,71	5	0,24	5,63
pistacia	200 013	288 222	9	41,77	8	3,12	4,64
prunus	143 068	205 548	7	46,89	6	2,56	6,70
quercus	563 239	808 557	8	40,78	7	11,51	5,10
ribes	121 160	183 833	11	48,48	7	1,99	4,41
castanea	317 150	430 788	13	39,12	7	4,85	3,01
citrus	29 950	37 398	5	30,15	4	0,37	6,03
cornus	593 916	686 151	7	30,37	5	5,40	4,34
cotoneaster	443 449	751 399	11	53,77	6	7,56	4,89
elaeagnus	272 188	392 070	10	40,97	6	4,27	4,10
phyllo	254 411	321 762	12	40,60	6	3,12	3,38
acer_monsp	573 078	796 640	7	44,64	6	9,50	6,38
acer_saccar	661 778	844 472	13	42,69	7	7,62	3,28
Moyenne	322 286,77	443 687,38	9,38	42,38	6,15	4,78	4,76
Min	16 328	21 096	5	30,15	4	0,24	3,01
Max	661 778	844 472	13	53,77	8	11,51	6,70

2 - 129 plans de découpe uniformes selon l'axe le plus long

Nom Modèle (.x)	Nombre de sommets	Nombre de faces	Nombre de feuilles non-vides F	Gain volumique G (% initial)	Prof Max	Temps construction en secondes	Ratio G/F
picea	16 328	21 096	9	50,83	5	0,41	5,65
pistacia	200 013	288 222	9	41,39	8	5,46	4,60
prunus	143 068	205 548	7	47,25	6	4,52	6,75
quercus	563 239	808 557	6	38,14	5	16,34	6,36
ribes	121 160	183 833	12	48,68	8	3,66	4,06
castanea	317 150	430 788	13	39,29	7	8,76	3,02
citrus	29 950	37 398	5	30,40	4	0,64	6,08
cornus	593 916	686 151	7	30,33	5	9,81	4,33
cotoneaster	443 449	751 399	8	49,66	6	14,27	6,21
elaeagnus	272 188	392 070	10	40,92	6	7,67	4,09
phyllo	254 411	321 762	12	40,56	6	5,56	3,38
acer_monsp	573 078	796 640	7	44,99	6	17,16	6,43
acer_saccar	661 778	844 472	12	41,58	7	12,76	3,47
Moyenne	322 286,77	443 687,38	9	41,85	6,08	8,23	4,96
Min	16 328	21 096	5	30,33	4	0,41	3,02
Max	661 778	844 472	13	50,83	8	17,16	6,75

3 - 65 plans de découpe uniformes selon les 3 axes

Nom Modèle (.x)	Nombre de sommets	Nombre de faces	Nombre de feuilles non-vides F	Gain volumique G (% initial)	Prof Max	Temps construction en secondes	Ratio G/F
picea	16 328	21 096	10	51,76	6	0,43	5,18
pistacia	200 013	288 222	10	42,19	7	5,59	4,22
prunus	143 068	205 548	7	46,11	6	6,46	6,59
quercus	563 239	808 557	11	41,96	7	21,03	3,81
ribes	121 160	183 833	9	48,40	7	4,56	5,38
castanea	317 150	430 788	9	41,17	6	13,61	4,57
citrus	29 950	37 398	8	35,40	6	1,21	4,43
cornus	593 916	686 151	8	33,22	5	14,71	4,15
cotoneaster	443 449	751 399	9	53,20	6	14,42	5,91
elaeagnus	272 188	392 070	11	45,32	6	10,54	4,12
phyllo	254 411	321 762	12	40,42	7	9,84	3,37
acer_monsp	573 078	796 640	8	47,17	7	29,71	5,90
acer_saccar	661 778	844 472	11	40,03	7	21,34	3,64
Moyenne	322 286,77	443 687,38	9,46	43,57	6,38	11,80	4,71
Min	16 328	21 096	7	33,22	5	0,43	3,37
Max	661 778	844 472	12	53,20	7	29,71	6,59

C - Les intérêts et limites de l'implémentation actuelle

1 - Gain volumique et gain volumique maximal

Voici un tableau qui expose le gain volumique réalisé avec notre méthode et le gain volumique maximal possible pour différents modèles de plantes et d'arbres.

Nom Modèle (.x)	Nombre de feuilles non-vides F	Nombre de feuilles non-vides max FM	Gain volumique G (%)	Gain volumique max GM (%)	F/FM	G/GM
picea	9	108	50,71	67,84	0,08	0,75
pistacia	9	111	41,77	61,78	0,08	0,68
prunus	7	99	46,89	64,57	0,07	0,73
quercus	8	140	40,78	61,84	0,06	0,66
ribes	11	89	48,48	66,69	0,12	0,73
castanea	13	140	39,12	58,16	0,09	0,67
citrus	5	177	30,15	58,70	0,03	0,51
cornus	7	161	30,37	50,08	0,04	0,61
cotoneaster	11	136	53,77	72,09	0,08	0,75
elaeagnus	10	130	40,97	64,52	0,08	0,63
phyllo	12	219	40,60	65,36	0,05	0,62
acer_monsp	7	90	44,64	64,55	0,08	0,69
acer_saccar	13	119	42,69	59,61	0,11	0,72
Moyenne	9,38	132,23	42,38	62,75	0,08	0,67
Min	5	89	30,15	50,08	0,03	0,51
Max	13	219	53,77	72,09	0,12	0,75

Les résultats sont bons car on arrive en moyenne à retirer 67 % du volume maximal que l'on peut retirer avec notre méthode et cela avec moins de 10 sous-volumes en moyenne.

2 - Niveau de subdivision et évolution du framerate

Les tests suivants ont été effectués avec une fenêtre d'affichage de résolution 800x600, en résolution texture 3D initiale 256x256x256 et pour un modèle qui prend le maximum d'espace sur l'écran. L'ordinateur utilisé était un Intel Xeon à 4 coeurs à 1,86 GHz avec 8 GB de RAM, avec pour carte graphique une GeForce 8800 GTS.

Voici des benchmarks pour des textures 3D (gestion correcte de la transparence et sans aucun calcul d'ombre (avec les ombres, les FPS sont divisés par 1,5 en moyenne)):

Nom Modèle (.x)	Nombre de sous-volumes	FPS	Nombre de sous-volumes	Gain (% total)	FPS	Nombre de sous-volumes	Gain (% total)	FPS
picea	1	26	5	36	43	9	50,71	58
pistacia	1	31	3	27,27	41	9	41,77	50
prunus	1	25	2	26,25	34	7	46,89	47
quercus	1	27	3	22,24	32	8	40,78	40
ribes	1	30	3	23,52	38	11	48,48	50
castanea	1	28	3	15,29	31	13	39,12	42
citrus	1	35	3	19,80	45	5	30,15	52
cornus	1	32	3	10,94	36	7	30,37	48
cotoneaster	1	29	4	34,31	45	11	53,77	63
elaeagnus	1	30	4	25,43	42	10	40,97	50
phyllo	1	36	4	20,52	46	12	40,60	60
acer_monsp	1	31	3	28,29	43	7	44,64	53
acer_saccar	1	29	5	23,71	36	13	42,69	43
Moyenne	1	29,92	3,46	24,12		9,38	42,38	
Min	1	25	2	10,94		5	30,15	
Max	1	36	5	36		13	53,77	

3 - Gestion de plusieurs boites

Nous avons vu qu'il était nécessaire d'utiliser une représentation adaptative des modèles volumiques d'arbre ou de plante, pour optimiser le coût du rendu. Plus une texture 3D sera proche de la caméra, plus le nombre de sous-volumes utilisés sera élevé, la limite étant fixée au nombre total de feuilles du *KD-Tree*.

Il est intéressant d'étudier plus en détails l'influence du nombre de sous-volumes, par exemple pour déterminer un nombre maximal de sous-volumes en fonction de la qualité de rendu demandée. Il faut faire attention à ce que les sous-volumes choisis ne soient pas trop différents, c'est-à-dire que pour nos tests nous devons fixer les paramètres une fois pour toute (% vol min et % vol père à supprimer), et générer le *KD-Tree*, pour finalement utiliser la totalité des feuilles. En effet, si on cherchait un sous-ensemble de sous-volumes (plus petit que celui de départ), on risquerait d'avoir un raffinement inégal au sein de la représentation obtenue. Et cela pourrait fausser

les résultats.

Dans ce qui suit, nous avons effectué des tests en résolution fenêtre d'affichage 800x600 et en résolution 256x256x256 pour les textures 3D. Pour un affichage de la texture 3D qui maximise sa surface projetée sur l'image finale, on obtient:

Sans tri *back-to-front* des volumes et sans ombre, les séries de tests effectuées montrent qu'un nombre de volumes supérieur à 100 ne permet pas de gain attractif. De plus, à cause de notre heuristique sur la largeur minimale d'une tranche, lorsqu'on est proche de 100 sous-volumes, on ne pourra pratiquement plus subdiviser les sous-volumes du *KD-Tree* en cours de construction. En effet, le *KD-Tree* complet obtenu sans les contraintes sur la perte volumique directe et sur le volume de taille minimale contient 132 feuilles en moyenne pour notre ensemble de modèles. Finalement 100 est une bonne borne, car le gain volumique du *KD-Tree* obtenu en cette borne sera très proche du gain volumique maximal.

Avec tri *back-to-front* des volumes et sans ombre, les séries de tests effectuées montrent qu'on doit utiliser un nombre de volumes inférieur à 50. Sinon le coût de gestion des sous-volumes devient plus coûteux que l'affichage d'un seul volume.

Sans tri *back-to-front* des volumes et avec les ombres, les séries de tests effectuées montrent qu'on doit utiliser un nombre de volumes inférieur à 45.

Avec tri *back-to-front* des volumes et avec les ombres, les séries de tests effectuées montrent qu'on doit utiliser un nombre de volumes inférieur à 45.

Pour une texture 3D qui est affichée dans une fenêtre de résolution 800x600, il faut utiliser un nombre de volumes inférieur à 45. Cela justifie l'idée directrice qu'un petit nombre de volumes était nécessaire. De plus, lorsque la texture 3D est affichée plus loin de la caméra (avec certainement un niveau de *MIP-map* plus grand), le nombre de volumes doit être encore plus petit, car le nombre de calculs nécessaires pour l'affichage des pixels diminue, et donc le temps passé pour la gestion des volumes devient encore plus critique.

Après avoir manipulé des représentations volumiques en utilisant la gestion de la transparence et les *shadow maps* et en étudiant attentivement l'évolution du *framerate* pour différents point de vue, il est apparu que l'on doit utiliser un nombre de sous-volumes inférieur à 13 pour le dernier niveau de subdivision, un nombre de sous-volumes inférieur à 7 pour le niveau de subdivision intermédiaire et un seul volume pour les textures 3D les plus éloignées. Avec ce nombre de sous-volumes, on sera la plupart du temps dans une configuration qui ne limitera pas le *framerate*, et lorsqu'on se rapprochera d'une zone de passage d'un niveau de subdivision à un autre, il arrivera que la gestion des sous-volumes limite le *framerate*, mais seulement légèrement... Ce droit droit à l'erreur nous permet d'utiliser un plus grand ensemble de *KD-Trees*, mais pour permettre un gain de *framerate* plus important pour les vues les plus proches. Ce qui est très important dans notre cas, car nous sommes intéressés par les vues les plus proches, car c'est pour ces dernières que la suppression d'espace vide est la plus utile.

Ces résultats montrent que les *KD-Trees* construits avec nos heuristiques sont bons, car ils permettent d'obtenir un petit nombre de volumes inférieur à 13 pour l'ensemble de nos modèles

de plante et d'arbre. Et donc on peut utiliser directement leurs feuilles pour générer le niveau de subdivision L2.

D - Implémentation en C++

1 - Présentation des différentes classes

La mise en oeuvre de notre méthode pour optimiser le rendu volumique basé sur des tranches de textures 3D a été réalisée en C++ à l'aide du logiciel de développement Microsoft Visual C++. Je vais donc vous présenter les classes principales (programmation objet) implémentées.

La construction d'un *KD-Tree* (partition binaire) nécessite une structure de noeud dans laquelle nous avons un pointeur sur les 2 enfants (qui sont non-nuls tous les 2 pour un noeud intérieur ou nuls tous les 2 pour une feuille), mais aussi un pointeur sur le noeud père (nul pour la racine du *KD-Tree*) pour ne pas perdre la notion de parenté au sein de l'arbre (essentielle pour parcourir l'arbre dans le sens père-fils mais aussi fils-père). Ensuite, il est utile dans notre cas d'avoir une *AABB* par noeud qui délimite le volume associé aux données du noeud (faces triangulaires ou voxels). Nous avons trouvé intéressant d'avoir aussi une structure spécifique pour manipuler un plan, car par exemple il est utile de pouvoir trier les plans, de faire des calculs d'intersection entre un plan et une droite, de déterminer si un plan intersecte une face, de regarder si une face se trouve à gauche ou à droite d'un plan, etc...

Ainsi les classes principales de notre projet sont *plane*, *node*, *interiornode*, *aabb* et *kdtree*. *interiornode* est juste une classe qui hérite de la classe *node* et qui contient en plus le plan de découpe pour séparer les 2 enfants, le coût associé et l'*index buffer* de toutes les faces du noeud. Il est intéressant de garder l'information sur les faces présentent au sein d'un noeud pour savoir s'il est vide, pour le découper (ou le fusionner avec un autre noeud), mais aussi pour générer un maillage DirectX *.x* associé à la zone définie par l'*AABB* du noeud. *aabb* est une classe qui permet de connaître la position de l'*AABB* d'un noeud (position Min et position Max) et qui offre des méthodes pour englober des faces ou des *AABBs*, pour générer des plans (les côtés), pour comparer la position de 2 *AABBs* par rapport à la position de la caméra, pour faire des intersections d'*AABBs*, etc... *kdtree* est une classe qui contient un *vertex buffer*, un *index buffer* qui définit les faces triangulaires du maillage, une liste de plans de découpe candidats selon les trois axes (mise à jour pour chaque noeud à découper), une liste de coûts associée (qui est mise à jour après la génération des plans candidats), un pointeur sur un objet de type *node* (la racine du *KD-Tree*) et trois listes de sous-volumes (avec une structure d'affichage par sous-volume) pour gérer les 3 niveaux de subdivision. Elle contient aussi plusieurs paramètres constants fixés au moment de la création d'une instance de la classe, tels que le pourcentage du volume de l'*AABB* initiale à utiliser pour le volume minimal, le pourcentage du volume père à utiliser pour le gain volumique directe minimal, le nombre de découpes uniformes selon un axe, si l'axe le plus long est utilisé pour chaque découpe ou les 3 axes, le choix de la fonction de coût pour sélectionner le meilleur plan candidat, etc...

La classe *kdtree* s'occupe du chargement d'un maillage DirectX, de la création du *KD-Tree*, de la création des 3 niveaux de subdivision à partir du *KD-Tree* construit et de l'affichage du meilleur niveau de subdivision par rapport à la distance entre le centre du modèle et la position de la

caméra. Elle possède aussi des méthodes statistiques et de visualisation pour valider numériquement et visuellement et les résultats obtenus. Dans les méthodes statistiques, il y a par exemple une méthode qui indique le nombre de feuilles, une autre le nombre de feuilles non-vides, une autre le volume total à afficher associé aux feuilles non-vides du *KD-Tree*, etc... Les résultats visuels consistent principalement en de la visualisation des *AABBs* associées à un niveau de subdivision, ou alors à celles associées à une profondeur pour voir progressivement les *AABBs* épouser les données au fur et à mesure que la profondeur augmente. Il y a des méthodes pour créer des morceaux de maillages 3D DirectX dans la classe *kdtree*...

2 - Codes sources intéressants

i - La création des 2 noeuds fils à partir d'un plan de découpe et d'un noeud

Ici j'ai décidé de vous montrer un exemple typique de balayage des faces d'un noeud de l'arbre en cours de construction, au cours duquel les *AABBs* des 2 nouveaux fils, leur densité et leur *index buffer* sont déterminés. Comme vous pouvez le remarquer, j'ai utilisé la biblithèque *Standard Template Library* (*std::vector*) pour manipuler des listes d'objets.

```
// The main function which divides data node
// for the two children and also computes
// the AABBs and density child
void kdtree::DivideDataNodeForChildren(
                                    const node * n,
                                    const plane& p, // splitting plane
                                    const std::vector< std::vector< unsigned int > >& IndexBuffer,
                                    std::vector< std::vector< unsigned int > >& IndexBufferChildL,
                                    std::vector< std::vector< unsigned int > >& IndexBufferChildR,
                                    aabb &AABBL,
                                    aabb &AABBR,
                                    double & densL,
                                    double & densR)
{
            if(!n || !n->GetAABB().StrictlyContains(p) ) return;

            Vec3f s1, s2,s3;
            unsigned int sharedfaces = 0;
            // initialization
            AABBL=aabb();
            AABBR=aabb();
            densL = densR = 0.0;
            IndexBufferChildL.clear();
            IndexBufferChildR.clear();
            Vec3f Inter1, Inter2;
            std::vector< unsigned int > Left, Right; // index buffer
            unsigned int nbmat = (unsigned int)IndexBuffer.size();
            for(unsigned int i=0; i<nbmat; ++i) // for each material
            {
                        Left.clear();
                        Right.clear();
                        unsigned int nbindices = (unsigned int)IndexBuffer[i].size();
                        std::vector< unsigned int >::const_iterator iter = IndexBuffer[i].begin();

                                    for(unsigned int j=0; j<nbindices; j+=3) // for each face
                                    {
                                                s1 = m_vertexBuffer[*iter];
                                                s2 = m_vertexBuffer[*(iter+1)];
                                                s3 = m_vertexBuffer[*(iter+2)];
```

```
                        if(p.FaceInPlane(s1,s2,s3))
                        {   // on the splitting plane: shared face
                                sharedfaces++;

                                Left.push_back(*iter);
                                Left.push_back(*(iter+1));
                                Left.push_back(*(iter+2));
                                AABBL.IncludePlane(p);
                                densL += aabb(s1, s2, s3
                                        ).GetBalancedVolume(m_eps);

                                Right.push_back(*iter);
                                Right.push_back(*(iter+1));
                                Right.push_back(*(iter+2));
                                AABBR.IncludePlane(p);
                                densR += aabb(s1, s2, s3)
                                        .GetBalancedVolume(m_eps);
                        }
                        else if(p.FaceToLeft(s1,s2,s3))
                        {   // to the left
                                Left.push_back(*iter);
                                Left.push_back(*(iter+1));
                                Left.push_back(*(iter+2));
                                AABBL.IncludeFace(s1, s2, s3);
                                densL += aabb(s1, s2, s3
                                        ).GetBalancedVolume(m_eps);
                        }
                        else if(p.FaceToRight(s1,s2,s3))
                        {   // to the right
                                Right.push_back(*iter);
                                Right.push_back(*(iter+1));
                                Right.push_back(*(iter+2));
                                AABBR.IncludeFace(s1, s2, s3);
                                densR += aabb(s1, s2, s3
                                        ).GetBalancedVolume(m_eps);
                        }
                        else
                        {           // split face (at least one vertex
                                    // strictly to the left
                                    // and one strictly to the right)
                                Left.push_back(*iter);
                                Left.push_back(*(iter+1));
                                Left.push_back(*(iter+2));

                                densL += aabb(s1, s2, s3
                                        ).GetBalancedVolume(m_eps);

                                Right.push_back(*iter);
                                Right.push_back(*(iter+1));
                                Right.push_back(*(iter+2));

                                densR += aabb(s1, s2, s3
                                        ).GetBalancedVolume(m_eps);

                                aabb::IncludeSplitFaces(s1, s2, s3, p,
                                                        AABBL, AABBR);
                        }

                        iter+=3;
                }// 2nd for

                IndexBufferChildL.push_back(Left);
                IndexBufferChildR.push_back(Right);
        } // 1er for
```

```
                ////////////////////////////
                // Indexbuffer optimization //
                ////////////////////////////
                // optimization to minimize the number of stored indices
                if(         sharedfaces &&
                            CountFacesNumber(IndexBufferChildL)==sharedfaces &&
                            CountFacesNumber(IndexBufferChildR)==sharedfaces )
                {
                  // case where there are faces only on the splitting plane
                  // we just choose to keep those faces in the smaller box
                  if( n->GetAABB().GetVolumeLeft(p) < n->GetAABB().GetVolumeRight(p) )
                  { // the left one is the smaller
                            IndexBufferChildR.clear();
                            AABBR=n->GetAABB().GetAABBRight(p);
                            densR = 0.0;
                  }
                  else
                  { // the right one is the smaller
                            IndexBufferChildL.clear();
                            AABBL=n->GetAABB().GetAABBLeft(p);
                            densL = 0.0;
                  }
                }
                else if(sharedfaces &&  CountFacesNumber(IndexBufferChildL)==sharedfaces &&
                                        CountFacesNumber(IndexBufferChildR)>sharedfaces)
                {
                            // case where we can erase data to the left
                            IndexBufferChildL.clear();
                            AABBL=n->GetAABB().GetAABBLeft(p);
                            densL = 0.0;
                }
                else if(    sharedfaces && CountFacesNumber(IndexBufferChildR)==sharedfaces
                                        && CountFacesNumber(IndexBufferChildL)>sharedfaces)
                {
                            // case where we can erase data to the right
                            IndexBufferChildR.clear();
                            AABBR=n->GetAABB().GetAABBRight(p);
                            densL = 0.0;
                }

                ////////////////////
                // AABB checking //
                ////////////////////
                if( !AABBL.isValid() )
                            AABBL=n->GetAABB().GetAABBLeft(p);

                if( !AABBR.isValid() )
                            AABBR=n->GetAABB().GetAABBRight(p);

                ////////////////////
                // AABB updates //
                ////////////////////
                if(AABBL.isFlatAlongPlane(p))
                            AABBL=n->GetAABB().AdjustAABBLeft(p, m_minimalLengthDimAABB);
                else
                            AABBL.Intersection(n->GetAABB().GetAABBLeft(p));

                if(AABBR.isFlatAlongPlane(p))
                            AABBR=n->GetAABB().AdjustAABBRight(p, m_minimalLengthDimAABB);
                else
                            AABBR.Intersection(n->GetAABB().GetAABBRight(p));

                ///////////////////////
                // density updates //
                ///////////////////////
                densL = densL/n->GetAABB().GetBalancedVolume(m_eps);
                densR = densR/n->GetAABB().GetBalancedVolume(m_eps);
}
```

ii - Le code HLSL intéressant du Ray-Casting sur GPU

Le fichier *Raycasting.fx*:

```
//-----------------------------------------------------------------------------------------
// global variables
//-----------------------------------------------------------------------------------------
float4x4 g_mWorldViewProjection; // World * View * Projection transformation

// textures
texture g_RenderedTexture;
texture g_MeshTexture;

// associated samplers
sampler2D g_backside = sampler_state {
                            Texture = <g_RenderedTexture>;
                            magfilter = LINEAR; //ANISOTROPIC;
                            minfilter = LINEAR; //ANISOTROPIC;
                            mipfilter = LINEAR; //ANISOTROPIC;
                            AddressU = border;
                            AddressV = border;
                            };

sampler3D g_volume_tex = sampler_state {
                            Texture = <g_MeshTexture>;
                            magfilter = LINEAR; //ANISOTROPIC;
                            minfilter = LINEAR; //ANISOTROPIC;
                            mipfilter = LINEAR; //ANISOTROPIC;
                            AddressU = border;
                            AddressV = border;
                            AddressW = border;
                            };

float g_stepsize = 1.0f/256.0f; // texture sampling factor, the final
                                // step = ray_length*g_stepsize
                                // to do 256 sampling steps



//-----------------------------------------------------------------------------------------
// structures
//-----------------------------------------------------------------------------------------

// Define interface between the application and the vertex program
// the initial AABB (triangle vertices with texture coordinates)
struct VS_INPUT
{
   float4 Position: POSITION;     // geometrical input (cube = BBOX associated with the
                                  // 3D texture)
   float4 TexCoord: TEXCOORD0;    // 3D texture coordinates associated with the input
                                  // position (vertex)
};
```

```
// Define the interface between the vertex and the fragment programs for pass one
struct VS_OUTPUT_P0
{
   float4 Position: POSITION;     // For the rasterizer
                                  // (At a very basic level, rasterizers simply take a
                                  // stream of vertices, transform them into
                                  // corresponding 2-dimensional points on the viewer's
                                  // monitor and fill in the transformed 2-dimensional
                                  // triangles as appropriate)

   float4 InterpTex: COLOR0;      // 3D texture coordinates in the unit cube to allow
                                  // coordinate interpolation
                                  // it is an intelligible way to easily find the
                                  // starting and ending points in the 3D texture
};


// Define the interface between the vertex and the fragment programs for pass two
struct VS_OUTPUT_P1
{
   float4 Position: POSITION;     // For the rasterizer
                                  // (At a very basic level, rasterizers simply take a
                                  // stream of vertices, transform them into
                                  // corresponding 2-dimensional points on the viewer's
                                  // monitor and fill in the transformed 2-dimensional
                                  // triangles as appropriate)

   float4 Pos: TEXCOORD0;         // memorize the projected position on the screen (it
                                  // is the rasterized coordinates)

   float4 InterpTex: COLOR0;      // 3D texture coordinates in the unit cube to allow
                                  // coordinate interpolation
                                  // it is an intelligible way to easily find the
                                  // starting and ending points in the 3D texture
};


//--------------------------------------------------------------------------------------
// Vertex shader
//--------------------------------------------------------------------------------------

// specific to the first pass
VS_OUTPUT_P0 VSmainP0( VS_INPUT IN )
{
  // structure init
  VS_OUTPUT_P0 OUT = (VS_OUTPUT_P0)0;

  // Transform vertex
  // we get the vertex position on the window
  OUT.Position =  mul(IN.Position, g_mWorldViewProjection);

  // Texture coordinates
  OUT.InterpTex = IN.TexCoord; // set a texture coordinates as color
                               // to interpolate it
  return OUT;
}
```

```
// specific to the second pass
VS_OUTPUT_P1 VSmainP1( VS_INPUT IN )
{
  // structure init
  VS_OUTPUT_P1 OUT = (VS_OUTPUT_P1)0;

  // Transformed vertex
  // we get the vertex position on the window
  OUT.Position =  mul(IN.Position, g_mWorldViewProjection);

  OUT.Pos = mul(IN.Position, g_mWorldViewProjection);

  // Texture coordinates
  OUT.InterpTex = IN.TexCoord; // set a texture coordinates as color
                               // to interpolate it

  return OUT;
}


//--------------------------------------------------------------------------------------
// Pixel shader
//--------------------------------------------------------------------------------------

// specific to the first pass
float4 PSmainP0( VS_OUTPUT_P0 IN ) : COLOR0
{
    return IN.InterpTex;
}


// specific to the second pass
// Raycasting fragment program implementation
float4 PSmainP1(VS_OUTPUT_P1 IN ) : COLOR0
{
  // find the right place to lookup in the backside buffer
  // from the cuboïd [-1,1]x[-1,1]x[0,1] to coordinate space [0,1]x[0,1]
  float2 texc;
  texc.x = ((IN.Pos.x / IN.Pos.w) + 1)*0.5f;
  texc.y = 1-((IN.Pos.y / IN.Pos.w) + 1)*0.5f;

  // the start position of the ray in the texture coordinate space
  float4 start_position = IN.InterpTex;

  // the end position of the ray in the texture coordinate space
  float4 back_position  = tex2D(g_backside, texc);

  // vector dir in texture space
  float3 dir = back_position - start_position;

  // the length from front to back is calculated and used to
  // terminate the ray
  float len = length(dir.xyz);

  // direction normalization (some vector may be too small)
  float3 norm_dir = normalize(dir);

  float step = g_stepsize * len;

  float3 delta_dir = norm_dir * step;
```

```
  ////////////////////////////////////////////////////
  // Main source code for the Ray-Casting implementation//
  // with a back-to-front blending order                //
  ////////////////////////////////////////////////////

  // starting position
  float3 vec = back_position;

  // accumulated color
  float4 col_acc = float4(0.0f,0.0f,0.0f,0.0f);

  float4 color_sample;
  float alpha_sample;

  float length_acc = 0.0f;

 while( length_acc < len )
 {
      color_sample = tex3Dlod(g_volume_tex, float4(vec,0.0f));

      alpha_sample = color_sample.a;

      // color blending for back-to-front order
      col_acc = (1.0 - alpha_sample) * col_acc + alpha_sample*color_sample;

      // updates
      vec -= delta_dir;
      length_acc += step;
    }

  return col_acc;
}


//--------------------------------------------------------------------------------------
// Techniques (effect)
//--------------------------------------------------------------------------------------
technique RenderScene
{
    pass P0
    {

        VertexShader = compile vs_1_1 VSmainP0();
        PixelShader  = compile ps_2_0 PSmainP0();
    }

    pass P1
    {
        VertexShader = compile vs_1_1 VSmainP1();
        PixelShader  = compile ps_3_0 PSmainP1();
    }
}
```

Il n'y a aucune optimisation dans notre implémentation, car l'objectif était d'apprendre la programmation de vertex et pixel *shaders* et aussi de manipuler des textures 3D avec des méthodes de DirectX 9 comme *D3DXCreateVolumeTextureFromFileEx* (avec un fichier *.dds* en entré). Une fois notre fichier *.fx* écrit, pour l'utiliser, nous créons un *Effect* (au moyen de l'interface *ID3DXEffect*) avec la méthode *D3DXCreateEffectFromFile*. Cela permet d'intégrer nos vertex et pixel *shaders* dans le pipeline graphique. Le code intéressant du côté de l'API (*Application Programming Interface*) graphique est composé de la méthode *RenderToTexture* et de la méthode de rendu *OnFrameRender* appelée à la fin de chaque *frame*.

```
// some global variables declaration

LPDIRECT3DVERTEXDECLARATION9 m_pVertexDeclaration = NULL;
LPDIRECT3DVERTEXBUFFER9      g_pVB = NULL;
// for render to texture we need the following variables:
LPDIRECT3DTEXTURE9 pRenderTexture = NULL;
LPDIRECT3DSURFACE9 pRenderSurface = NULL, pBackBuffer = NULL;


//////////////////////
// Texture handling //
//////////////////////

// Rendering to a texture
// The most common case of creating an empty texture to be filled with data during runtime
// is the case where an application wants to render to a texture and then use the results
// of the rendering operation in a subsequent pass. Textures created with this purpose should
// specify default usage.
// Rendering to a texture enables many possibilities, but there are also some restrictions you
// have to take care of. First the depth stencil surface must always be greater or equal to the
// size of the render target. Furthermore the format of the render target and the depth stencil
// surface must be compatible and the multisample type must be the same.

inline void RenderToTexture(LPDIRECT3DDEVICE9 pDevice)
{
            HRESULT hr;
            D3DVIEWPORT9 Viewport;

#ifdef DEBUG
            IDirect3D9* pD3D;
#endif
            ////////////////////////////////////
            // retrieve the current viewport //
            ////////////////////////////////////
            V( pDevice->GetViewport(&Viewport) );

#ifdef DEBUG
            //////////////////////
            // format checking //
            //////////////////////
            pDevice->GetDirect3D((IDirect3D9**)&pD3D);

            V( ((Idirect3D9*)pD3D)->CheckDeviceFormat(
                                                  D3DADAPTER_DEFAULT, // UINT Adapter,
                                                  D3DDEVTYPE_HAL, // D3DDEVTYPE DeviceType,
                                                  D3DFMT_X8R8G8B8, // D3DFORMAT AdapterFormat,
                                                  D3DUSAGE_RENDERTARGET, //DWORD Usage
                                                  D3DRTYPE_TEXTURE, // D3DRESOURCETYPE RType,
                                                  D3DFMT_X8R8G8B8 //D3DFORMAT CheckFormat
                                                 ) );
            SAFE_RELEASE(pD3D);
#endif
            ////////////////////////////////////////
            // Creates a texture with the same     //
            // resolution that the current viewport//
            ////////////////////////////////////////
            // some hardware requires power of two for a texture dim

            //SAFE_RELEASE(pRenderTexture);
            if(!pRenderTexture)
                        V( D3DXCreateTexture(
                                          pDevice,
                                          Viewport.Width, //UINT Width / D3DX_DEFAULT
                                          Viewport.Height, //UINT Height / D3DX_DEFAULT
                                          1, //a complete mipmap chain is created with D3DX_DEFAULT
                                          D3DUSAGE_RENDERTARGET, //DWORD Usage
                                          D3DFMT_X8R8G8B8, // D3DFORMAT Format
                                          D3DPOOL_DEFAULT, // When using the texture as a render
                                                           // target the memory pool has to be
                                                           // D3DPOOL_DEFAULT
                                          &pRenderTexture
                                          ) );
```

```
      // In order to access the texture memory a surface object is needed, because a texture
      // object in Direct3D always uses such a surface to store its data.
      // To get the underlying surface we call the method GetSurfaceLevel() of the texture.
      // The parameters are the index of the texture level to get and a pointer to a surface object.
            if(!pRenderSurface)
                        pRenderTexture->GetSurfaceLevel(0, &pRenderSurface);

            // the render target has to be set to the texture surface for texture rendering
            V( pDevice->SetRenderTarget(0, (IDirect3DSurface9*)pRenderSurface ) );
}


...
...
...

//--------------------------------------------------------------------------------------
// This callback function will be called at the end of every frame to perform all the
// rendering calls for the scene, and it will also be called if the window needs to be
// repainted. After this function has returned, DXUT will call
// IDirect3DDevice9::Present to display the contents of the next buffer in the swap chain
//--------------------------------------------------------------------------------------
void CALLBACK OnFrameRender( IDirect3DDevice9* pd3dDevice, double fTime, float fElapsedTime,
                             void* pUserContext )
{
    // If the settings dialog is being shown, then
    // render it instead of rendering the app's scene
    if( g_SettingsDlg.IsActive() )
    {
        g_SettingsDlg.OnRender( fElapsedTime );
        return;
    }

    HRESULT hr;
    D3DXMATRIXA16 mWorldViewProjection;
    UINT iPass, cPasses;
    D3DXMATRIXA16 mWorld;
    D3DXMATRIXA16 mView;
    D3DXMATRIXA16 mProj;

    // Clear the render target and the zbuffer
    V( pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                                                    D3DXCOLOR(0.0f,0.25f,0.25f,0.55f), 1.0f, 0) );
    // Get a pointer on the current surface used for rendering
    pd3dDevice->GetRenderTarget(0,&pBackBuffer);

    ////////////////////////
    // Render the scene //
    ////////////////////////
    if( SUCCEEDED( pd3dDevice->BeginScene() ) )
    {
        // Get the projection & view matrix from the camera class
        mWorld = g_mCenterWorld * *g_Camera.GetWorldMatrix();
        mProj = *g_Camera.GetProjMatrix();
        mView = *g_Camera.GetViewMatrix();

        mWorldViewProjection = mWorld * mView * mProj;

        ////////////////////////////////////////
        // our starting BBOX made of triangles //
        ////////////////////////////////////////
        CreateCubeVertexBuffer(pd3dDevice, Min, Max, g_pVB); // Create a Bbox used for rendering
                                                             // with a Min and a Max vectors
        ////////////////////////
        // device settings //
        ////////////////////////
        V( pd3dDevice->SetFVF( CUSTOMVERTEX::D3DFVF_CUSTOMVERTEX ) );
        V( pd3dDevice->CreateVertexDeclaration(VertexElements, &m_pVertexDeclaration) );
        V( pd3dDevice->SetVertexDeclaration( m_pVertexDeclaration ) );
        V( pd3dDevice->SetStreamSource( 0, g_pVB, 0, sizeof(CUSTOMVERTEX) ) );
```

```
        // Update the effect's variables.  Instead of using strings, it would
        // be more efficient to cache a handle to the parameter by calling
        // ID3DXEffect::GetParameterByName
        V( g_pEffect->SetMatrix( "g_mWorldViewProjection", &mWorldViewProjection ) );


        //////////////////////////
        // render state param //
        //////////////////////////
        V( pd3dDevice->SetRenderState( D3DRS_FILLMODE, D3DFILL_SOLID) );

        // we can set the mesh texture which has been created
        V( g_pEffect->SetTexture( "g_MeshTexture", g_pMeshTexture) );

        V( g_pEffect->SetTechnique( "RenderScene" ) );

        // Apply the technique contained in the effect
        V( g_pEffect->Begin(&cPasses, 0) );

        for (iPass = 0; iPass < cPasses; iPass++)
        {
            V( g_pEffect->BeginPass(iPass) );

            if(iPass==0)
            {  // rendering to the texture

               // CW = front faces culling => rendering back faces
               V( pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW) );
               ////////////////////////////////////
               // set texture rendering target //
               ////////////////////////////////////
               RenderToTexture(pd3dDevice); // creates the pRenderTexture texture
            }
            else
            {  // iPass == 1
               // set the backbuffer

               V( pd3dDevice->SetRenderTarget(0,pBackBuffer) );
               // CCW = back faces culling => rendering front faces
               V( pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW) );

               // set rendered texture
               // we need this texture for the last pass
               V( pd3dDevice->SetTexture(0, pRenderTexture) );
               V( g_pEffect->SetTexture( "g_RenderedTexture", pRenderTexture) );
            }

            V( pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, g_nbtriangles ) );

            V( g_pEffect->EndPass() );
        }
        V( g_pEffect->End() );

        RenderText( fTime ); // to display the FPS
        V( pd3dDevice->EndScene() );
    }

    SAFE_RELEASE(g_pVB);
    SAFE_RELEASE(m_pVertexDeclaration);

    SAFE_RELEASE(pRenderTexture);
    SAFE_RELEASE(pRenderSurface);
    SAFE_RELEASE(pBackBuffer);
}
```

E - Le LIAMA

L'Institut National de Recherche en Informatique et en Automatique (INRIA) et l'Académie des Sciences de Chine (CAS) ont créé à Pékin, en janvier 1997, un Laboratoire Franco-Chinois de Recherche en Informatique, Automatique et Mathématiques Appliquées (LIAMA), hébergé dès l'origine par l'Institut d'Automatique de l'Académie des Sciences de Chine (CASIA, http://www.ia.ac.cn/new/english/info.asp?column=191).

J'ai passé la plus grande partie de mon stage au sein du LIAMA. Dans mon bureau, j'étais avec une chercheuse chinoise et une thésarde chinoise. Le LIAMA est un endroit que je recommande aux futurs étudiants, car il y a une bonne ambiance de travail, il y a beaucoup d'étudiants chinois qui sont très accueillants et qui parlent anglais, et même pour certains le français. L'inconvénient majeur était la mauvaise gestion du réseau, et à cause de cela le débit des connexions Internet était très faible. Par contre, il y a la possibilité de manger très bien pour 2 ou 3 euros pour le déjeuner.

Site Web: http://liama.ia.ac.cn/wiki/doku.php
Sur ce site vous trouverez les projets (en particulier le projet Greenlab), ainsi que des informations sur les coopérations, les publications, les offres de stage...

Quelques informations complémentaires sur:
http://www-direction.inria.fr/international/LIAMA.html

F - Un petit mot sur mon séjour en Chine

Ne pas partager mon expérience en Chine serait regrettable, car je pense qu'un séjour en Chine est quelque chose d'unique, tant la culture, la cuisine, les temples, la grande muraille et les chinois font de la Chine une destination typique et inoubliable.

Commençons par parler des Chinois. D'une manière générale, ils sont accueillants et gentils avec les étrangers. Chaque fois que j'ai demandé de l'aide à un Chinois pour ma route par exemple, je suis toujours tombé sur une personne qui m'a aidé et parfois qui est allée m'accompagner jusqu'à la destination souhaitée. Le seul problème, c'est qu'il y a très peu de Chinois qui parlent anglais, et donc ce n'est pas toujours évident de se faire comprendre. Une fois que j'ai appris les bases en Chinois, j'ai beaucoup plus apprécié ma vie en Chine!
Les Chinois sont très conviviaux et j'ai eu l'occasion de me faire de nombreuses connaissances et même des amis dans Beijing.

A Beijing, les pauvres côtoient les riches et les très riches. En fait il n'y a pas vraiment de classe moyenne, il y a de gros écarts de niveaux de vie. Les vélos et les gros 4x4 ou les Audi A6 se croisent...
C'est une ville très intéressante sur le plan historique (place Tian An Men, la cite interdite, la Grande Muraille, les temples et les tombeaux...), mais aussi sur le point de vu culinaire (Canard laqué, nourriture impériale, plats épicés, ...). La pollution et les mauvaises odeurs dans les rues sont les plus points négatifs à signaler. Dans l'air il y a un mélange de poussière et de pollution qui fait qu'il est vraiment difficile de faire du sport en plein air. J'ai essayé de faire du foot, mais j'avais l'impression de manquer d'air. Je ne sais pas comment ils vont faire les athlètes en août 2008 pour

bouger pendant les Jeux Olympiques!

Le coût de la vie est environ 10 fois moins élevé pour ce qui est de la nourriture. On peut très bien manger dans un restaurant pour environ 2 euros (ou dans un Macdo). Pendant mon séjour en Chine, je n'ai jamais fait la cuisine une seule fois, je suis toujours allé manger dans un restaurant. C'est normal, puisque je pouvais manger quelque chose de délicieux pour 1 ou 2 euros.
J'ai eu la bonne expérience d'aller chez le coiffeur, pour 1,5 euros j'ai eu trois personnes qui se sont occupés de moi et pendant 1h30! Une femme pour le shampooing, un homme pour la coupe et une autre femme pour le rinçage!

J'ai eu l'occasion d'aller à Xi'an voir l'armée enterrée (Terra Cotta Warriors and Horses), à Shengde voir 3 des 8 fameux temples, d'aller à Shanghai voir le poumon économique de la Chine (et l'oriental pearl tower!) et d'aller à Hangzhou et Suzhou acheter un peu de soie et apprécier leur paysages sauvages vraiment magnifiques. J'ai aussi rencontré des gens intéressants et adorables pendant mon séjour. Mon premier contact avec l'Asie restera gravé dans ma mémoire à tout jamais! Avant de venir en Chine, j'étais assez intéressé par l'Asie et en particulier par la Corée du Sud. Cela s'explique par le fait que je vis dans une résidence à Grenoble, dans laquelle les coréens sont omniprésents. Maintenant je suis fan! J'espère que plus tard j'aurais l'occasion de revenir en Chine et de visiter d'autre pays...

G - EVASION

EVASION : Environnements Virtuels pour l'Animation et la Synthèse d'Images d'Objets Naturels (équipe-projet créée en 2003) (http://www.inria.org/recherche/equipes/evasion.fr.html)

Site Web: http://www-evasion.imag.fr/

J'ai passé la fin de mon PFE au sein de l'équipe EVASION (1 mois 1/2), et j'ai vraiment apprécié l'environnement de travail, les JT du jeudi au cours desquels les chercheurs présentent leur travail, car cela permet d'enrichir sa culture personnelle en synthèse d'images et en animation. J'ai été entouré de gens dynamiques, sympathiques et experts dans leur domaine.

VII - Bibliographie

A - Articles scientifiques

1 - Articles d'introduction au rendu volumique et au cadre du projet

[DECAUDIN and NEYRET 2004]
P. DECAUDIN and F. NEYRET, *Rendering Forest Scenes in Real-Time*, Eurographics Symposium on Rendering '04, 2004, H. W. Jensen A. Keller (Editors), pp. 93-102.

[JESCHKE et al. 2005]
S. JESCHKE, M. WIMMER and W. PURGATHOFER, *Image-based Representations for Accelerated Rendering of Complex Scenes*, State of the Art Report, EUROGRAPHICS 2005.

[MEYER and NEYRET 1998]
A. MEYER and F. NEYRET, *Interactive Volumetric Textures*, Rendering Techniques '98 (*Proceedings of the Eurographics Workshop on Rendering 98*), June 1998, Drettakis G., Max N., Eurographics, Springer-Verlag Wien New York, pp. 157-168.

2 - Autres références

[BOADA et al. 2001]
I. BOADA, I. NAVAZO AND R. SCOPIGNO, *Multiresolution Volume Visualization with a Texture-Based Octree*, 2001, The Visual Computer 17, 3, 185–197.

[CABRAL et al. 1994]
B. CABRAL, N. CAM AND J. FORAN, *Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware*, In VVS '94: Proceedings of the 1994 symposium on Volume visualization, 91–98.

[COHEN-OR and KAUFMAN 1995]
D. COHEN-OR and A. KAUFMAN, *Fundamentals of Surface Voxelization*, Graphical Models and Image Processing, 57, 6 (November 1995), 453-461.

[CROW 1984]
F. C. CROW, *Summed-area Tables for Texture Mapping*, Proceedings of SIGGRAPH '84, *Computer Graphics,*18(3):207-212.

[DACHILLE et al. 1998]
F. DACHILLE, K. KREEGER, B. CHEN, I. BITTER and A. KAUFMAN, *High-quality Volume Rendering using Texture Mapping Hardware,* In Eurographics / SIGGRAPH Workshop on Graphics Hardware, 1998, pages 69–76.

[DEVILLERS 1989]
O. DEVILLERS, *The macro-regions: an Efficient Space Subdivision Structure for Ray Tracing*, Eurographics (September 1989), 27–38.

[ENGEL and ERTL 2002]
K. ENGEL AND T. ERTL, *Interactive High-quality Volume Rendering with Flexible Consumer Graphics Hardware*, Eurographics '02, State of the Art Report, 2002.

[GLASSNER 1990]
A. S. GLASSNER, *Multidimensional Sum Tables*, In Graphics Gems, 376-381. Academic Press, New York, 1990.

[HADWIGER et al. 2006]
M. HADWIGER, J. M. KNISS, C. REZK-SALAMA, D. WEISKOPF AND K. ENGEL, *Real-Time Volume Graphics*, A. K. Peters, 2006.

[HAVRAN 2001]
V. HAVRAN, *Heuristic Ray Shooting Algorithms*, PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[HAVRAN 2002]
V. HAVRAN and J. BITTNER, *On Improving Kd-tree for Ray Shooting*, In Proceedings of WSCG, 2002, pp. 209-216.

[HUANG et al. 1998]
J. HUANG, R. YAGEL, V. FILIPPOV, and Y. KURZION, *An Accurate Method for Voxelizing Polygon Meshes*, IEEE Symposium on Volume Visualization, 1998.

[HUNT et al. 2006]
W. HUNT, W. R. MARK and G. STOLL, *Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic*, 2006 IEEE Symposium on Interactive Ray Tracing, Sept. 2006. http://www-csl.csres.utexas.edu/gps/publications/rt06-fast-kd/

[HURLEY et al. 2002]
J. T. HURLEY, A. KAPUSTIN, A. RESHETOV and A. SOUPIKOV, *Fast Ray Tracing for Modern General Purpose CPU*, In Proceedings of Graphicon, 2002.

[KÄHLER et al. 2003]
R. KÄHLER, M. SIMON, AND H.-C. HEGE, *Interactive Volume Rendering of Large Sparse Data Sets using Adaptive Mesh Refinement Hierarchies,* IEEE Transactions on Visualization and Computer Graphics ,2003, 9, 3, 341–351.

[KINDLMANN and DURKIN 1998]
G. KINDLMANN and J. W. DURKIN, *Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering*, IEEE 1998 Symposium on Volume Visualization, S. 79-86.

[KNISS et al. 2001]
J. KNISS, P. MCCORMICK, A. MCPHERSON, J. AHRENS, J. PAINTER, A. KEAHEY and C. HANSEN, *Interactive Texture-based Volume Rendering for Large Data Sets*, IEEE Computer Graphics and Applications, 2001, 21(4):52–61.

[KRÜGER and WESTERMANN 2003]
J. KRÜGER and R. WESTERMANN, *Acceleration Techniques for GPU-based Volume Rendering*, in VIS '03: Proceedings of the 14th IEEE Visualization 2003 Conference, 38–42.

[LACROUTE and LEVOY 1994]
P. LACROUTE and M. LEVOY, *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*, COMPUTER GRAPHICS Proceedings, Annual Conference Series, 1994.

[LAMAR et al. 1999]
E. C. LAMAR, B. HAMANN and K. I. JOY, *Multiresolution Techniques for Interactive Texture-based Volume Visualization*, In IEEE Visualization 1999, Proceedings of the 10th IEEE Visualization 1999 Conference, pages 355–362.

[LEVOY 1990]
M. LEVOY, *Efficient Ray Tracing of Volume Data*, ACM Transactions on Graphics, Vol. 9, No. 3, July 1990, pp. 245-261.

[LI and KAUFMAN 2002]
W. LI and A. KAUFMAN, *Accelerating Volume Rendering with Texture Hulls*, IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics (October 2002), 115-122.

[LI and KAUFMAN 2003]
W. LI and A. KAUFMAN, *Texture Partitioning and Packing for Accelerating Texture-based Volume Rendering*, Graphics Interface, 81-88.

[LI et al. 2003]
W. LI, K. MUELLER and A. KAUFMAN, *Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering,* Visualization, 2003. VIS 2003. IEEE Volume , Issue , 19-24 Oct. 2003 Page(s): 317 - 324.

[LORENSEN and CLINE 1987]
W. E. LORENSEN and H. E. CLINE, *Marching cubes : a high resolution 3D surface construction algorithm*, *SIGGRAPH '87 Conference Proceedings (Anaheim, CA, July 27–31, 1987)*, In M. C. Stone, editor, pages 163–170 (Computer Graphics, Volume 21, Number 4, July 1987).

[MACDONALD and BOOTH 1990]
J. MACDONALD and K. BOOTH: *Heuristics for Ray Tracing using Space Subdivision*, The Visual Computer, 1990, Vol. 6, No. 3, pp. 153–166.

[MAX et al. 1995]
N. MAX, D. LIVERMORE and L. LIVERMORE, *Optical Models for Direct Volume Rendering, tutorial*, University of California and National Laboratory, 1995.

[MEISSNER et al. 2000]
M. MEISSNER, H. PFISTER, R. WESTERMANN and C.M. WITTENBRINK, *Volume Visualization and Volume Rendering Techniques*, In Proceedings Of EUROGRAPHICS, Tutorial 6, 2000.

[MONTANI et al. 1994]
C. MONTANI, R. SCATENI and R. SCOPIGNO, *Discretized Marching Cubes* Visualization, 1994., Visualization apos, 94, Proceedings, IEEE Conference on Volume , Issue , 17-21 Oct 1994 Page(s):281 - 287, CP32.

[MUELLER et al. 1999]
K. MUELLER, N. SHAREEF, J. HUANG and R. CRAWFIS, *High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels*, IEEE Transactions on Visualization and Computer Graphics 5, 2 (April-June 1999), 116-134.

[STEGMAIER et al. 2005]
S. STEGMAIER, M. STRENGERT, T. KLEIN AND T. ERTL, *A Simple and Flexible Volume Rendering Framework for Graphics-hardware-based Raycasting*, In Fourth International Workshop on Volume Graphics, 2005, 187–241.

[STOLL 2005]
G. STOLL, *Part II: Achieving Real Time Optimization Techniques*, In SIGGRAPH 2005 Course on Interactive Ray Tracing, 2005.

[SUBRAMANIAN and FUSSELL 1990]
K. R. SUBRAMANIAN and D. S. FUSSELL, *Applying Space Subdivision Techniques to Volume Rendering*, In VIS ’90: Proceedings of the 1st conference on Visualization ’90, 150–159.

[TONG et al. 1999]
X. TONG, W. WANG, W. TSANG and Z. TANG, *Efficiently Rendering Large Volume Data Using Texture Mapping Hardware*, In Proceedings of Joint Eurographics/IEEE TVCG Symposium on Visualization.

[WALD et al. 2001]
I. WALD, P. SLUSALLEK, C. BENTHIN and M. WAGNER, *Interactive Rendering with Coherent Ray Tracing*, Computer Graphics Forum 20, 3 (Proceedings of Eurographics), 2001, pp. 153-164.

[WALD 2004]
I. WALD, *Realtime Ray Tracing and Interactive Global Illumination*, PhD thesis, Computer Graphics Group, Saarland University, 2004.

[WALD and HAVRAN 2006]
I. WALD and V. HAVRAN, *On building fast kd-trees for Ray Tracing, and on doing that in O(Nlog N)*, In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing. 61–70.

[WEISKOPF et al. 2004]
D. WEISKOPF, M. WEILER and T. ERTL, *Maintaining Constant Frame Rates in 3D Texture-Based Volume Rendering*, Computer Graphics International 2004 (CGI'04), Proceedings,16-19 June 2004, On page(s): 604- 607.

[WESTERMANN and ERTL 1998]
R. WESTERMANN and T. ERTL, *Efficiently using Graphics Hardware in Volume Rendering Applications,* In Proceedings of ACM SIGGRAPH 1998, pages 169–178.

[WILLIAMS 1978]
L. WILLIAMS, *Casting Curved Shadows on Curved Surfaces*, International Conference on Computer Graphics and Interactive Techniques, Proceedings of the 5th annual conference on Computer graphics and interactive techniques, 1978, Pages: 270 – 274.

B - Pages Web

Sur le rendu volumique:
http://www.gris.uni-tuebingen.de/people/staff/meissner/
http://www.gris.uni-tuebingen.de/people/staff/bartz/
http://www.cs.sunysb.edu/~mueller/
http://graphics.stanford.edu/papers/lacroute_thesis/
http://en.wikipedia.org/wiki/Volume_rendering
http://en.wikipedia.org/wiki/Marc_Levoy
http://www.cs.sunysb.edu/~mueller/research/pop/presHtml/sld001.htm

Volume Rendering Integral:
http://www.cs.unm.edu/~kmorel/documents/dissertation/thesis_full/node8.html

Textures:
http://en.wikipedia.org/wiki/Texture_filtering