Visualisation haute-qualite de forets en temps-reel a l'aide de representations alternatives

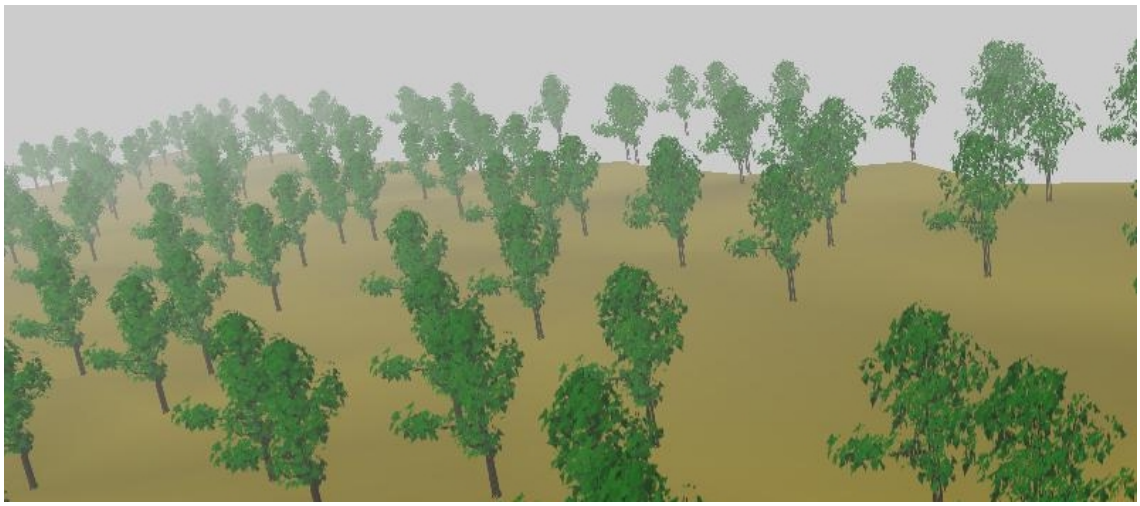
Franck Senegas

DEA Imagerie Vision Robotique 2000/2001

Visualisation haute-qualité de forêts en temps-réel à l'aide de représentations alternatives

RAPPORT DE STAGE

Franck Sénégas



Responsable de Stage : Fabrice Neyret

Laboratoire d'accueil : iMAGIS/GRAVIR -IMAG
INRIA Rhône-Alpes
ZIRST
655, avenue de l'Europe
38330 Montbonnot Saint-Martin
FRANCE

Table des matières

1 Introduction **2**
- 1.1 L'équipe, le domaine, le sujet . . . 2
- 1.2 Présentation du sujet . . . 2
- 1.3 Motivations . . . 3
- 1.4 Plan du rapport . . . 3

2 Texels : une représentation alternative **4**
- 2.1 Les approches non temps-réel : Représentation et rendu . . . 4
  - 2.1.1 L'approche de Kajiya et Kay . . . 4
  - 2.1.2 Les limites du modèle de Kajiya et Kay . . . 7
  - 2.1.3 Généralisation du modèle à des géométries arbitraires . . . 8
- 2.2 Les approches temps-réel . . . 8
  - 2.2.1 Le modèle fondateur des volumes en temps-réel . . . 8
  - 2.2.2 Application aux texels en temps-réel . . . 8
- 2.3 Le shading . . . 10

3 Le rendu de forêts **12**
- 3.1 Les versions temps-réel . . . 12
- 3.2 Le rendu multi-échelles . . . 14
- 3.3 Les approches texels . . . 14
- 3.4 Les problèmes . . . 17

4 Rendu temps réel haute qualité **19**
- 4.1 Le pipeline OpenGL standard . . . 20
  - 4.1.1 Le Geometric Engine . . . 20
  - 4.1.2 Le Raster Engine . . . 20
  - 4.1.3 Attributs divers . . . 21
  - 4.1.4 Rendu complexe . . . 21
- 4.2 Les nouveautés de la nouvelle génération de matériel . . . 21
  - 4.2.1 Le multitexture . . . 22
  - 4.2.2 Les cube maps . . . 22
  - 4.2.3 Les vertex shaders . . . 22
  - 4.2.4 Les pixels shaders . . . 22
  - 4.2.5 Exemple d'implémentation des pixels shaders : les registers combiners . . . 23
- 4.3 A titre d'exemple illustratif : Bump-mapping en Hardware . . . 24
  - 4.3.1 Présentation de l'éclairage . . . 25
- 4.4 Application aux texels . . . 26

5 Texels réalistes temps réel **29**
5.1 Les limitations des modèles actuels . . . 29
5.2 Conditions expérimentales . . . 30
5.3 Notre représentation . . . 30
5.3.1 Notre modèle de texels . . . 30
5.3.2 Le modèle de texture . . . 31
5.4 Modèle de rendu d'un arbre . . . 31
5.4.1 Version simple . . . 32
5.4.2 Version avec brouillard . . . 32
5.5 Modèle amélioré . . . 32
5.6 Filtrage des éléments de texture . . . 33
5.7 Méthode de génération des tranches de volume . . . 34
5.7.1 Les méthodes existantes . . . 35
5.7.2 La méthode employée . . . 35
5.8 Rendu de forêts . . . 36
5.9 Multi-échelle . . . 37

6 Résultats et bilan **38**
6.1 Résultats . . . 38
6.1.1 Qualite visuelle . . . 38
6.1.2 Performances . . . 38
6.2 Améliorations possibles . . . 41
6.2.1 Les déformations . . . 41
6.2.2 Les ombres portées des arbres sur le sol . . . 42
6.2.3 Les ombres des arbres sur les autres . . . 42
6.3 Bilan . . . 42
6.4 limitations du modèle . . . 42

Résumé

Le but de ce stage de recherche est de proposer une nouvelle méthode de représentation et de visualisation de forêts à base de representations alternatives à la géométrie, afin d'obtenir une haute qualité graphique en temps réel de ces scènes très complexes, en s'appuyant sur les cartes graphiques grand public.

Chapitre 1

Introduction

Dans le cadre des jeux vidéos temps réel, la plupart des décors sont en général des endroits fermes. Ceci est du au fait que l'on peut afficher un mur à l'aides de quelques polygones, alors que le moindre arbre requiert au moins 100 polygones. La complexité géométrique devient alors le facteur limitant. En effet, comment maximiser l'interactivité avec le joueur, tout en réussissant à l'immerger dans une forêt ?

1.1 L'équipe, le domaine, le sujet

L'équipe iMAGIS travaille dans le domaine de l'informatique graphique et de la synthèse d'images. iMAGIS développe des outils permettant de concevoir, puis d'utiliser dans le cadre d'applications de taille significative, et en particulier en vue de simulation, des maquettes numériques 3D. Ces maquettes peuvent être purement géométriques ou posséder également des propriétés "physiques" (photométriques ou mécaniques par exemple). Les applications visées se situent dans des domaines très divers (construction automobile ou aéronautique, urbanisme, éclairagisme, bâtiment,téléphonie mobile, chimie, chirurgie assistée, agronomie, environnement, audio-visuel, etc.). Dans bien des cas il s'agit de concevoir les techniques (modélisation et algorithmes graphiques) sur lesquelles reposent les systèmes "de réalité virtuelle" (ou "de réalité augmentée") qui commencent à voir le jour. Le défi à relever est de leur fournir la puissance nécessaire à l'affichage et à l'interaction "temps réel" qui les caractérisent. Les domaines de recherche sont :

- visualisation d'environnements complexes ;
- rendu réaliste, simulation de l'éclairage (utilisation de modèles complexes de réflectance, éclairage global, techniques de radiosité hiérarchiques, environnements dynamiques, etc.) ;
- animation, modélisation d'objets déformables et de leur comportement (modèles à base de surfaces implicites, simulation du mouvement, détection et réponse à des collisions, fortes déformations, manipulation interactive, etc.) ;
- algorithmique de la visibilité ; structures de données efficaces pour le rendu de scènes très complexes (prise en compte de la cohérence, cas dynamique, etc.) ;
- interactivité, réalité augmentée (manipulation et partage d'objets virtuels, intégration image-son, etc.) ;

1.2 Présentation du sujet

Dans le domaine du jeu vidéo, ou même de la simulation temps-réel, il est difficile de pouvoir se promener virtuellement dans une scène naturelle. Autant les scènes d'intérieur sont devenues accessibles au temps-réel, autant les scènes d'extérieur sont beaucoup plus rares. Ceci est dû au fait que la complexité

géométrique d'une scène d'exterieur est immense. Alors que les scènes d'intérieur peuvent utiliser de très grands polygones pour les murs et le sol, et des modèles très simples pour le mobilier, les plantes et les arbres requièrent au moins des dizaines de milliers de polygones pour ressembler à des végétaux. La difficulté est de pouvoir obtenir un rendu de ces scènes en temps-réel. On s'intéresse ici à l'exploration interactive de paysages couverts de forêts.

Comme les détails se comptent par milliards, il n'est pas question de tracer chaque feuille une à une, bien que l'on ne souhaite pas non plus perdre d'information visible. Dans ce domaine, beaucoup de travaux ont été fait dans l'optique de réduire le nombre de polygones en fonction de la distance. Plusieurs représentations récentes offrent une alternative aux polygones ; c'est notamment le cas des textures volumiques, qui encodent dans un volume de faible résolution toute la géométrie qui se situe au voisinage d'une surface. Conformément à l'approche 'texture', le volume de référence (texel) est ensuite dupliqué pour couvrir la surface. Ce modèle a été initialement développé dans le contexte du rendu réaliste (i.e en batch), par contre beaucoup de problèmes restent à résoudre dans le contexte du temps-réel : un volume se représente simplement par une série de tranches transparentes texturées, cependant il n'est pas simple de rendre compte de l'illumination et des ombres.

1.3 Motivations

Des voies ont été ouvertes dans le domaine des texels : visualisation temps-réel de texels, substitution de texels à des modèles polygonaux, applications aux rendus de forêts. Notre but était de réussir à concilier toutes ces directions de recherche en s'appuyant sur la nouvelle génération de cartes graphiques pour y parvenir. Celles-ci permettant des calculs d'éclairage par point, elles peuvent être utilisées pour effectuer de la visualisation de forêts en utilisant des «texels temps-réel réalistes », c'est à dire des texels prenant en compte l'éclairage.

1.4 Plan du rapport

Nous commencerons par un état de l'art, organisé selon 3 axes essentiels, les texels au chapitre 2 le rendu de forêts temps-réel au chapitre 3 le rendu temps-réel haute qualité au chapitre 4

Nous présenterons ensuite au chapitre 5 notre modèle de texels réalistes en temps-réel nous présenterons discuterons résultats et performances au chapitre 6, puis nous conclurons.

Chapitre 2

Texels : une représentation alternative

Originellement introduits par Kajiya et Kay [KK89], les textures volumiques ont été tout d'abord introduites afin de simuler des éléments de fourrure. Le but était de simuler une peau volumique constituée d'un motif de référence placé un grand nombre de fois sur la surface d'un objet. Ce motif de référence (volume cubique) représentait la géométrie des poils sous forme de voxels, et chaque instance du motif était tracée sur la surface comme un pavage. On nomme une instance de ce volume un **texel** pour *texture element*. Le volume englobant pouvant être déformé, la répétition du motif était moins perceptible.
Ce modèle étant reservé à la fourrure, Neyret [Ney96b] étendit le concept et le généralisa à des formes quelconques, tout en introduisant le multi-échelle dans cette représentation afin d'en améliorer les performances. Bien qu'accélérant considérablement le temps de calcul des scènes, cette approche restait très liée au raytracing, car nécessitant le calcul de l'illumination en tout point de l'espace. Le temps-réel semblait donc exclu. Lacroute et Levoy introduisirent le concept de tranches de volume dans le cadre du rendu volumique classique, ce qui permit d'accélerer considérablement le temps de rendu en factorisant les traitements. Cette approche a rapidement engendré les méthodes temps-réel de rendu volumique classiques utilisées aujourd'hui en visualisation. Meyer et Neyret [MN98] adaptèrent l'idée en utilisant le hardware graphique pour afficher des texels en temps réel. Cependant, si la densité en tout point pouvait être prise en compte grâce aux textures, l'illumination en chaque point n'était plus prise en compte avec cette approche. Westermann et Ertl [WE98] ont mis au point une méthode pour obtenir le shading en tout point. Mes travaux visent à réunifier ces diverses idées, de manière à réobtenir la qualité visuelle de Neyret [Ney96b] avec l'approche temps-réel de Meyer [MN98], en introduisant le calcul de l'illuination par point à la manière de Westermann et Ertl [WE98] mais en s'appuyant sur les possibilités des nouvelles générations de cartes graphiques. Je distinguerais dans mon état de l'art, les approches non temps-réel à base de ray-tracing d'une part, et les approches temps-réel d'autre part.

2.1 Les approches non temps-réel : Représentation et rendu

2.1.1 L'approche de Kajiya et Kay

Représentation

Kajiya et Kay [KK89] représentent dans le volume de référence (un cube) le volume (la fourrure) qu'ils souhaitent représenter. Ils stockent l'information géométrique minimale du volume dans une structure de voxels. Chaque cellule de voxel contient :

- Une information de densité (ou «présence »)
- Un ensemble de 3 vecteurs donnant l'orientation locale de la surface (Normale, Tangente, Binormale)
- Une fonction de réflectance, décrivant la façon dont la lumière se réfléchit

Dans leur implémentation dédiée à la fourrure, Kajiya et Kay supposent que les poils sont cylindriques et verticaux dans le cube de référence, la fonction de reflectance est donc la même pour chaque cheveu, ils se contentent alors de stocker l'information de présence.

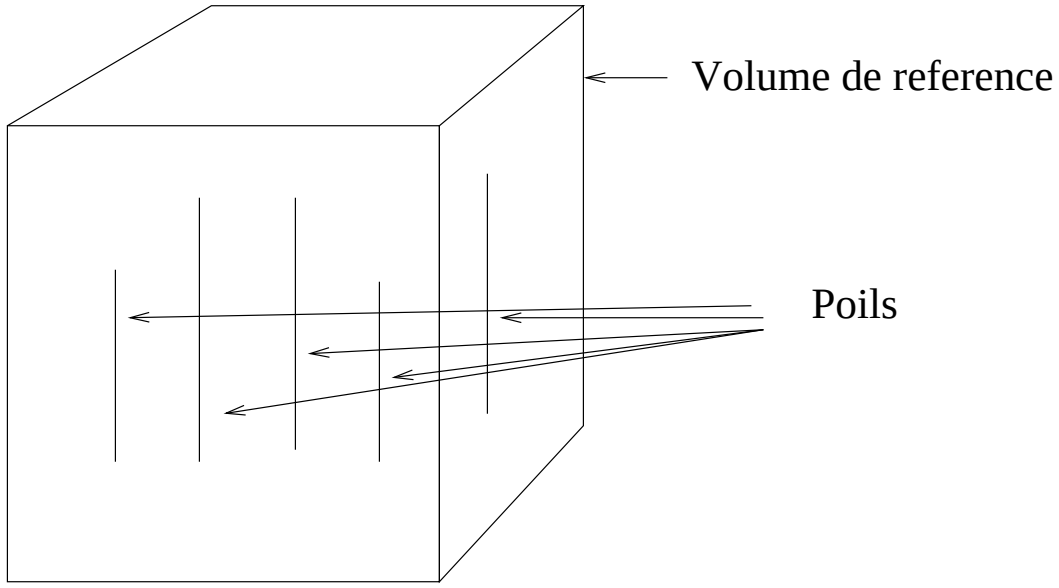


FIG. 2.1 – Le volume de kajiya contient dans le volume de référence uniquement la densité correspondant aux poils

Plaquage des texels

Dans leur cas, ils utilisent une surface uniquement constituée de quadrangles. Les volumes sont posés sur la surface comme sur la figure 2.2 page 6 et collent à la surface, c'est à dire que la base du volume de référence (un quadrilatère) est confondue avec le quadrangle de la surface. Les quatre sommets supérieurs du volume de référence sont initialement positionnés selon les normales aux sommets de la surface. On peut ensuite les orienter différemment pour déformer la texture (la «peigner »). Chaque normale est donc partagée entre 4 texels.

Rendu des texels

La méthode originale de rendu est basé sur une extension du raytracing. Si un rayon intersecte un texel, on détermine l'intersection d'Entrée et celle de Sortie entre le volume de référence et le rayon. Ensuite, le segment [ES] (entrée-sortie) est discrétisé en petits segments S_i. Sur chaque segment S_i, un point P_i est choisi au hasard (échantillonnage stochastique). Pour calculer la quantité de lumière que reçoit P_i, les rayons partant de P_i orientés vers les sources lumineuses sont tirés. Kajiya et Kay négligent les réflexions multiples comme c'est classique en rendu volumique, mais tiennent compte du blocage éventuel par des éléments opaques du voxel. La transparence cumulée T se calcule en la multipliant par la transparence

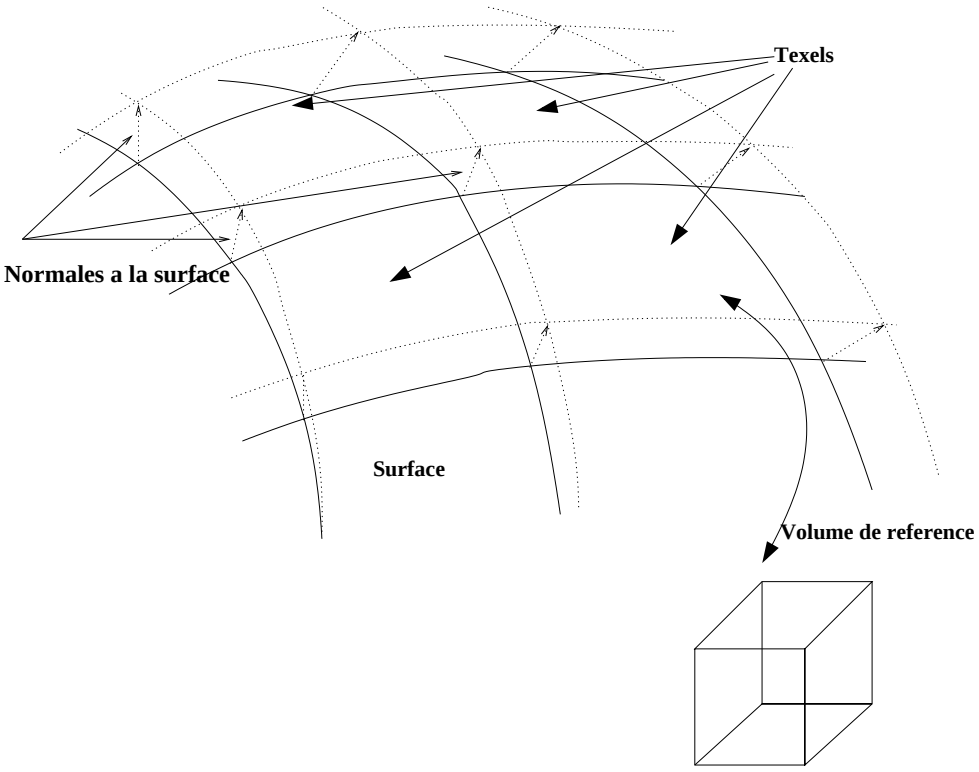


FIG. 2.2 – Le placage des texels sur une surface

T_i du segment S_i. T_i est fonction de la densité, de la longueur et d'une constante de densité.

$$T = T \cdot T_i$$
$$T_i = e^{-C \cdot d \cdot L_i}$$

où C est une constante, d la densité et L_i la longueur du segment S_i. Le calcul de T est effectué en prenant les P_i dans l'ordre de l'entrée vers la sortie. L'intensité I pour le rayon est calculée en fonction de chaque contribution d'intensité I_i en P_i pondérée par la transparence cumulée. L'intensité I_i en P_i est calculée selon le modèle de réflectance en fonction de la lumière L_i et de la transparence T_i.

$$I = I + I_i \cdot T$$

$$I_i = F(L_i) \cdot (1 - T_i)$$

où F est la fonction de réflectance. La transparence T utilisée dans la formule précédente est la transparence cumulée à laquelle on n'a pas encore multiplié la transparence T_i propre au segment S_i. Ce modèle a permis de calculer le fameux Teddy Bear sur un mainframe IBM avec comme processeurs douze 3090 et quatre 3081 en 12 heures de calcul en 1989 (voir figure 2.3 page 7).

FIG. 2.3 – Le *teddy bear* calculé par Kajiya grâce à sa méthode de textures volumiques

2.1.2 Les limites du modèle de Kajiya et Kay

Shinya [Shi92] a proposé quelques améliorations théoriques en 1992 en stockant l'occultation dans les trois directions canoniques afin d'obtenir une occultation anisotrope. Il liste aussi les problèmes non résolus par Kajiya comme la construction de texels à partir de représentation polygonales, la necessité de pouvoir représenter les texels de façon hierarchique, et suggére l'utilisation de données de corrélation entre les cellules. Pour autant, il ne propose pas vraiment de modèle d'implémentation pour résoudre ces problèmes. Noma applique aux arbres les textures volumiques et tire dans ce cas parti :

- une méthode de construction des feuillages
- une représentation multi-échelle

2.1.3 Généralisation du modèle à des géométries arbitraires

Comme le modèle de Kajiya est explicitement simplifié pour représenter des cheveux (pas de différentiation entre les normales ; les fonctions de réflectances sont factorisées en une seule par volume), Neyret [Ney95] introduit comme primitive de réflectance la distribution de normales, représentée de manière compacte par un ellipsoïde. Ces micro-primitives servent à coder la distribution de normales de la cellule. Dans chaque cellule, il stocke donc une distribution de normales sous la forme de coefficients de l'ellipsoïde (6 au total) et peut donc simuler l'effet lumineux de formes différentes.

La méthode originale [KK89] était en outre très lente. En effet, la technique de rendu peut prendre inutilement en compte des voxels dont la taille est inférieure à un pixel de l'image, notamment dans le cas de texels éloignés de l'observateur et l'espace vide est discrétisé autant que le reste. Comme dans le modèle de Neyret [Ney95], chaque cellule contient une micro-primitive, choisies pour leur bonne propriété de s'ajouter et de se moyenner correctement car il s'agit de fonctions de distributions, et que les voxels sont stockés de façon hiérarchique dans un octree, l'affichage des texels est multi-échelle. Ainsi, le parcours total de l'octree ne se fera que si l'observateur est proche de la cellule. Comme l'octree se prête bien au regroupement de volumes voisins, seule la moyenne des micro-primitives sera utilisée au niveau supérieur où une cellule représente 8 voxels voisins.

La méthode originale nécessitait beaucoup de mémoire. Le regroupement en octree permet de compresser les données jusqu'à 95%. [Ney95] obtient des temps de calcul de l'ordre de 10 à 20 minutes en 1995 ce qui est un temps raisonnable pour du ray-tracing. La création de scènes complexes est relativement facile pour l'utilisateur grâce à l'apport du multi-échelle propre aux approches texturelles : la grande échelle d'un côté, les détails d'habillage de l'autre.

2.2 Les approches temps-réel

2.2.1 Le modèle fondateur des volumes en temps-réel

Lacroute et levoy [LL94] introduirent l'idée de découper les volumes en tranches parallèles entre elles. Jusqu'alors, le calcul s'effectuait dans un espace voxelisé. Le rayon traversait le volume et on sommait les contributions de chaque cellule intersectée. L'idée principale de Lacroute et Levoy est d'effectuer le rendu des texels par l'intermédiaire de tranches de volumes que l'on translate puis projette dans l'espace image, de sorte que l'on puisse rendre le texel uniquement avec des opérations 2D (voir figure 2.4 page 9). Pour cela, ils projettent les tranches de volume dans le plan image, tout en les observant dans le bon ordre (de l'observateur vers la tranche la plus éloignée). Par pixel, ils disposent donc de toutes les informations afin de faire la sommation des contributions, mais comme les informations sont dans le plan image, le résultat est simple à calculer. Il ne reste qu'à chercher les pixels opaques puis éventuellement calculer leur éclairage. Comme tous ces calculs s'effectuent dans l'espace images, les opérations sont très rapides.

2.2.2 Application aux texels en temps-réel

Représentation des texels

La grande contrainte de la méthode précédente des textures volumiques est qu'elle n'est pas accélérable en hardware. En effet, toutes les opérations se font à l'échelle du pixel, et requièrent de faire un parcours par pixel de chaque contribution de voxel. Meyer et Neyret [MN98] ont proposé une méthode qui permet d'utiliser efficacement le hardware graphique dans l'esprit de [LL94]afin de rendre des volumes. L'idée est de repartir de la méthode de Lacroute et Levoy de considérer le volume comme une succession de couches. Plutôt que de considérer des tranches de voxels, ils vont considérer des couches d'images. Comme le hardware graphique ne sait pas effectuer du ray-tracing, il faut essayer de trouver une méthode adaptée au rendu projectif avec un algorithme de Z-Buffer. Le hardware graphique permet de tracer rapidement

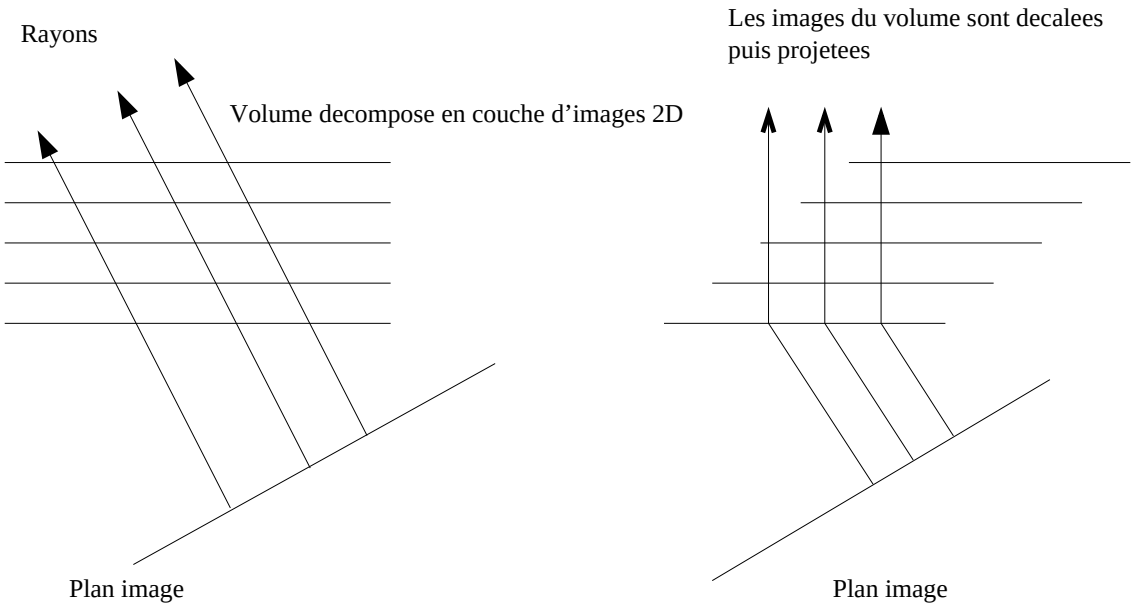


FIG. 2.4 – Afin de permettre le travail dans l'espace image, Lacroute et Levoy translatent les tranches (et les mettent à l'échelle dans le cas d'une transformation perspective) avant de les projeter

des polygones texturés, chaque tranche de volume sera associée à une tranche de texture 2D. Chaque texture comportera en fait deux informations :

- une information d'opacité (correspondant à l'information de densité de la cellule)
- une information de couleur (correspondant au précalcul de la couleur de la cellule : cette couleur est une information statique : si l'éclairage change, il faut recalculer toutes les couleurs des textures)

Le texel devient, dans leur approche, un ensemble de N tranches de textures de taille X x Y parallèlles entre elles, au format RGBA. Il peut être interprété comme N x X x Y voxels différents disposant uniquement d'une densité et d'une couleur. En fait, il y a trois directions de tranches. Par rapport aux approches précédentes, on ne peut donc plus stocker les informations de réflectances, ce qui donne un modèle moins riche, mais temps-réel : comme souvent, il s'agit d'un compromis entre vitesse et qualité. Ces tranches peuvent être fabriquées à partir d'un maillage en utilisant aussi le hardware.

Plaquage des texels sur la surface

Le système de placage de texels sur la surface du modèle de Kajiya se basait sur la géométrie portant la surface. La base du volume de référence coïncidait avec le quadrangle de la surface qui le portait. Le modèle proposé par Neyret [Ney96a, Ney98, MN98] et Meyer se plaque sur la surface sans se raccorder à des sommets de la surface. Afin de pouvoir placer le texel sur n'importe quel type de surface, notamment triangulaires, il propose de déformer le texel pour coller exactement à la surface. Chaque texel va se composer de plusieurs boîtes. Une boîte est une portion du texel, dont le type de tranches dépend uniquement de la nature des facettes de la surface, et dont les coordonées de textures sont calculées en fonction de la déformation locale du texel.

Rendu des texels

Le rendu standard accéléré par le hardware graphique utilise un algorithme de Z-Buffer. Cet algorithme est simple : pour tous les polygones, pour tout les points de ce polygone, si la profondeur de ce point est

supérieure à celui déjà stocké dans le Z-Buffer, on ne l'affiche pas (cf chapitre 4). La technique de rendu consiste à rendre les faces texturées dans l'ordre du fond vers l'observateur, en ne rendant que les points de texture de chaque tranche dont l'opacité est non nulle. Le conflit du test de profondeur (Z-Buffer) avec le calcul de la contribution de chaque cellule nous force à faire ce tri. En effet, il faut que toutes les cellules soient prises en compte. Ensuite le rendu est effectué en mode «mélange »(blend) pour le rendu de la transparence. La couleur de chaque pixel qui a passée le précédent test est calculée en utilisant la formule :

$$C_{ecran} = Opacite_{point} \times Couleur_{point} + (1 - Opacite_{point}) \times Couleur_{ecran}$$

De cette manière, les voxels complètement opaques (probabilité de présence=densité=1), sont remplis par leur couleur.

Afin d'eviter de se retrouver avec la direction de visée parallèle aux tranches de textures, il faut choisir la direction de tranches la plus perpendiculaire à cette direction. Pour cela, ils utilisent trois directions de tranches voir 2.5 page 10).

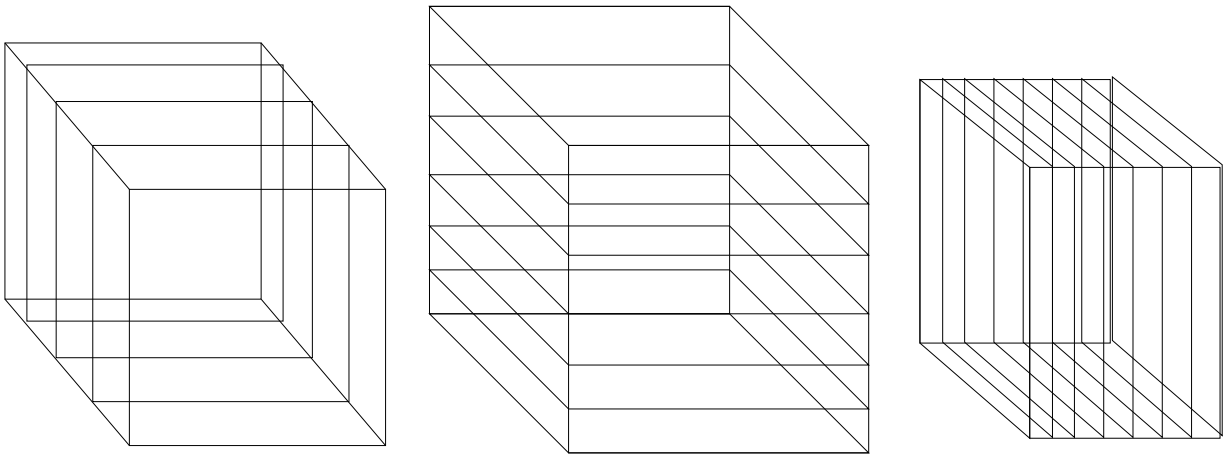


FIG. 2.5 – Meyer et Neyret utilisent 3 directions de tranches de texture afin de pouvoir faire le rendu du texel sous tous les angles de vue

[MN98] propose un critères pour choisir la direction de tranches optimales, à savoir minimiser ce que l'on voit entre deux tranches de textures.

2.3 Le shading

Westermann et Ertl [WE98] présentèrent un modèle susceptible d'introduire l'illumination locale (au moins le diffus). Leur but est d'obtenir l'éclairage diffus sur des images médicales ; la couleur leur importe peu. Leur technique de rendu d'un volume est basée sur les textures 3D prévues par OpenGL [MW]. En fait, il s'agit d'une texture que l'on adresse dans un espace [0..1] × [0..1] × [0..1] (un cube de texture) . On peut voir l'analogie du modèle avec celui de Meyer, qui échantillonait l'information des NxXxY voxels dans N textures : Westermann et Ertl utilisent une texture unique de résolution NxXxY (comme pour compacter les N textures en une seule). Leur technique de rendu consiste à positionner et tourner correctement le cube dans la scène. Il faut ensuite déterminer les polygones formant l'intersection de plans parallèles au plan image avec le cube, de déterminer leurs coordonnées de texture, de les rendre du plus éloigné vers le plus proche. Ils utilisent ensuite le même mode de tracé que Meyer (prise en compte de la transparence). Afin de simuler le shading diffus, ils introduisent une autre texture 3D que celle utilisée par Meyer, dont l'information de couleur correspond à la normale de chaque point de la texture

volumique. Comme une couleur RGB est définie dans l'espace $[0..1] \times [0..1] \times [0..1]$, et qu'une normale est définie dans l'espace $[-1..1] \times [-1..1] \times [-1..1]$, on applique la transformation

$$couleur(normale).R = normale.x/2 + 0.5$$

$$couleur(normale).G = normale.y/2 + 0.5$$

$$couleur(normale).B = normale.z/2 + 0.5$$

L'auteur ne se préoccupe pas de la couleur des cellules. L'image qu'il cherche à calculer est le texel avec une illumination diffuse. La formule de l'éclairage diffus est

$$I = I_a + k_d \vec{L}.\vec{N}$$

où $\vec{L}$ désigne le vecteur issu du point vers la source lumineuse, et $\vec{N}$ désigne la normale au point. Afin d'effectuer le calcul en utilisant les techniques avancées d'OpenGL, ils rendent le texel de normales normalement dans la mémoire d'écran. Ils préparent ensuite une matrice de transformation de couleurs qui s'appliquera sur chaque couleur qui a été tracée dans la mémoire décran. Elle prend en paramètre la couleur RGBA et a cette forme :

$$CM = \begin{pmatrix} L_x & L_y & L_z & 0 \\ L_x & L_y & L_z & 0 \\ L_x & L_y & L_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} M_{rot} \begin{pmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Afin de pouvoir introduire le terme k_d et I_a dans leur équation, OpenGL offre le moyen d'y parvenir : lorsqu'on lui demande de copier le résultat de son rendu sur lui-même, il permet :

- d'appliquer la matrice de couleur sur l'image du rendu
- de mettre à l'échelle (scale) (multiplication du resultat précédent par une valeur RGBA pour tous les pixels de l'image)
- d'additionner à ce résultat une couleur RGBA constante (bias)

le résultat est donc (par pixel) :

$$\begin{bmatrix} I_a \\ I_a \\ I_a \\ 0 \end{bmatrix} + \begin{bmatrix} K_d \\ K_d \\ K_d \\ 1 \end{bmatrix} CM \begin{bmatrix} R \\ G \\ B \\ \alpha \end{bmatrix} = \begin{bmatrix} K_d(\vec{L} \cdot \vec{N}) + I_a \\ K_d(\vec{L} \cdot \vec{N}) + I_a \\ K_d(\vec{L} \cdot \vec{N}) + I_a \\ \alpha \end{bmatrix}$$

Ceci est donc intéressant pour rendre des données médicales, mais ce modèle est insuffisant pour les textures volumiques qui requièrent couleur et fonction de réflection vide.

Au cours de mon stage, une publication [RSEB+00] sur le rendu volumique utilisant les «pixels shaders » de la dernière génération de cartes graphiques est sortie. Elle explique comment passer outre des limitations du modèle précédent en effectuant un calcul d'illumination par point (donc par cellule). Cet article est détaillé dans le chapitre suivant.

Chapitre 3

Le rendu de forêts

Le rendu de scènes naturelles, Grâal de bien des chercheurs, artistes et programmeurs depuis toujours est un exercice difficile. De plus, l'utilisateur terminal (spectateur) connaît en général bien l'aspect de ce genre de scène, et les représentations si elles veulent être fidèles requièrent beaucoup d'information. Cette grande masse d'information est à fortiori inaccessible au temps réel tant les représentations classiques sont inabordables par leur complexité. Ceci justifie qu'une forte activité de recherche s'intéresse à la mise au point d' une recherche de modèles adaptés à ce type de scène, afin de simplifier grandement la complexité géométrique sans perdre en complexité apparente. Du côté du réalisme (donc non temps-réel, des produits phares comme AMAP [Cir, Bio], permettent de réaliser des paysages entiers à partir de mdoèles d'arbres bio-réalistes. La plupart des approches qui ont cherché à représenter en temps-réel utilisent une ou plusieurs méthodes de simplification de complexité. Ainsi, nous détaillerons les versions temps-réel des modèles de forêts. Nous continuerons sur la technique du rendu multi-échelle qui permet de choisir la représentation d'un objet en fonction de sa distance à l'observateur. Ensuite nous parlerons des travaux effectués dans le domaine à l'aide des texels.

3.1 Les versions temps-réel

Pendant longtemps, les arbres étaient représentés par des sprites notamment dans les simulateurs de vol, avant l'apparition des cartes accélératrices 3D. Avec l'arrivée de ces cartes, la technique du billboard s'est imposée, mélange d'image et de géométrie préfigurant l'émergence des approches Image Based Rendering.

Cette technique consiste à utiliser des images semi-transparentes d'arbres tracées toujours face à l'utilisateur. Celle-ci est efficace si les arbres sont distants et si la forme de l'arbre est de symétrie relativement cylindrique. Avec l'apparition des cartes accélératrices, les modèles de billboards ont commencé à se diversifier. Ainsi, on a vu apparaître des représentation d' arbres à base de textures semi-tranparentes. Une des représentations les plus courantes est d'utiliser deux polygones semi-transparents perpendiculaires l'un à l'autre. Cette technique présente l'avantage d'être moins «statique »que le billboard. L'autre gros avantage de cette technique par rapport à un ensemble de polygones est qu'elle ne requiert pas beaucoup plus de calcul pour l'affichage.

Récemment, Jakulin [Jak00] a proposé un modèle hybride entre les texels, la géométrie et polygones en croix. Il sépare l'arbre en deux entités distinctes :

- Le tronc qui est un modèle polygonal
- Le feuillage qui va être représenté par des tranches de textures paralléles entre elles.

Pour créer les tranches de textures associées aux feuilles, il utilise des plans de clipping parallèles entre eux, et ne rend que les primitives inclues dans cet intervalle (voir figure 3.1 page 13). Pour en effectuer

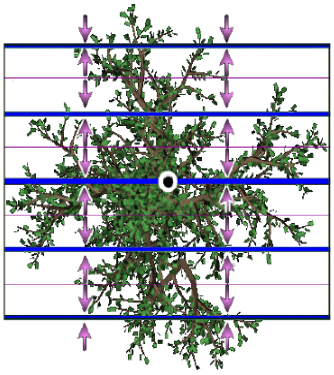


FIG. 3.1 – Chaque tranche de texture correspond à l'image des primitives inclues dans les deux plans de clipping les plus proche

le rendu, il mélange les tranches des deux directions de textures en pondérant les directions par une transparence fonction de l'angle entre la direction de visée et la normale à la tranche. Plus cet angle est proche de 0 et plus le coefficient est important, plus il est proche de $\frac{\pi}{2}$ est plus il est faible (voir résultat 3.2 page 13). Le problème de cette technique apparaît lorsque l'observateur est proche de l'arbre : des «fantômes »de feuilles apparaissent, et disparaissent dès que l'observateur bouge, puisque chaque feuille est représentée dans les deux séries de tranches, et qu'il y a une erreur de parallaxe résiduelle. Par ailleurs,

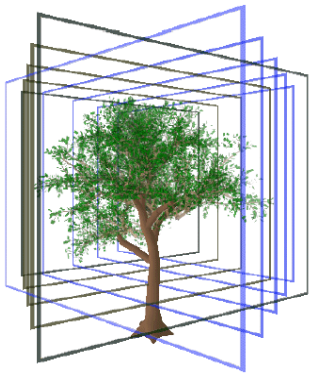


FIG. 3.2 – Méthode de Jakulin : Les directions de textures sont mélangées au cours du rendu du feuillage

le simple fait d'utiliser des tranches perpendiculaires entre elles qui se chevauchent tout en utilisant de la transparence est complètement faux, si on utilise le Z-Buffer, on ne peut plus les trier du fond vers l'avant.

Plusieurs personnes se sont intéressées à la technique des texels pour représenter des forêts notamment Noma [Nom95], Neyret [Ney96a], Chiba [CMDH97]. Nous détaillerons ces approches dans les prochains paragraphes.

D'autres approches sont basées sur des systèmes de particules [RB85], dans lesquels la géométrie avait été remplacé par des hachures ce qui permit de produire les premières images de forêt dense. Autrefois inaccessible pour le temps réel, ces approches sont utilisées maintenant dans certaines applications.

3.2 Le rendu multi-échelles

Le rendu multi-échelle est une méthode qui consiste à considérer différentes représentations pour un même objet, et de choisir sa représentation au moment du rendu en fonction de sa distance à l'observateur. Cette technique est très adaptée au rendu de scènes naturelles. Tout d'abord, un des grands principes des scènes naturelles est qu'elles permettent d'utiliser des niveaux de détails particulièrement différents. En effet, autant à un point de vue proche un arbre est surtout une information géométrique, autant de loin il est plus une silhouette. Afin d'économiser du temps de rendu, il suffit d'utiliser pour un même arbre plusieurs types de représentations de complexité d'affichage et de qualité différentes. On affichera donc les versions les plus détaillées (et donc les plus chères en temps de rendu) lorsque le modèle est proche de l'observateur, et des versions de plus en plus dégradées en fonction de l'éloignement. Néanmoins, ce type de rendu requièrent que :

- si on souhaite utiliser un seul système de représentation (par exemple que des polygones ou que des texels), il faut avoir un moyen permettant de stocker ou de calculer à partir d'une forme détaillée une forme dégradée
- si on souhaite utiliser plusieurs systèmes de représentations, il faut une façon d'effectuer des transitions douces entre les niveaux de détails (éviter les effets de popping désagréables à l'oeil)

Ce principe de rendu est essentiel quelques soient les systèmes de représentations utilisé pour les arbres, car la distance aidant, on peut utiliser des modèles très simples.

3.3 Les approches texels

Les deux approches présentées dans cette partie sont basées sur les texels dans un contexte de ray-tracing. Nous avons déjà détaillé la technique qu'utilisait Neyret [Ney96b] pour faire du multi-échelle, afin de pouvoir faire du rendu de forêts. L'approche de Noma [Nom95] permet de gérer le multi-échelle en utilisant une représentation du texel de type octree. Les informations qu'il va stocker dans son octree sont une densité et une BRDF.

Il part du principe que la majorité de l'apparence de l'arbre ne dépend presque que de ses feuilles. Noma va utiliser des informations différentes pour chaque direction. Pour chaque direction pour laquelle il veut calculer le texel, il échantillonne le volume à un niveau très fin. Pour se faire, il calcule en fonction des feuilles la densité de chaque cellule dans le niveau de détail le plus fin, . Ensuite les opacités des niveaux supérieurs de l'octree sont calculées en considérant que la densité de la cellule (englobant les 8 précédentes). Il les numérote comme détaillé sur la figure 3.3 page 14.

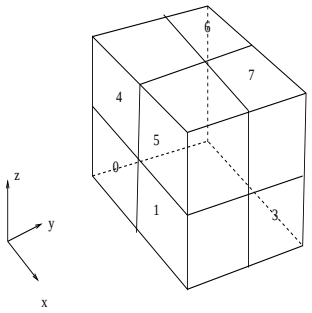


FIG. 3.3 – Le modèle de cube utilisé dans les formule de Noma

Pour calculer la densité $\rho(x)$ du noeud père dans l'octree pour la direction selon l'axe des x (i.e.

correspondant à la réunion de tous les voxels de la figure 3.3), il applique la formule suivante :

$$\rho^x = \frac{\rho_0^x + \rho_1^x + \ldots + \rho_7^x}{4} - \frac{\rho_0^x\rho_1^x + \rho_2^x\rho_3^x + \rho_4^x\rho_5^x + \rho_6^x\rho_7^x}{4}$$

Si on rapporte la notion de densité à une probabilité de présence, le calcul se comprend comme la somme des densités de toutes les cellules, donc la probabilité de toucher un élément de la cellule avec un rayon, et le deuxieme terme correspond à la probabilité de toucher une surface en traversant 2 cellules. Pour stocker les informations de réflectance, il échantillonne une BRDF par cellule. Cette BRDF est stockée sous la forme de la somme de cosinus d'angles entre chaque normale de chaque feuille de la cellule et la direction de la lumière. Si la position de la lumière varie, il doit tout recalculer. Il effectue ce calcul à tous les niveaux de détail. Ceci lui permet en fait d'effectuer le calcul à chaque niveau de détail. Le rendu qu'il obtient est visible sur la figure 3.4 page 15.

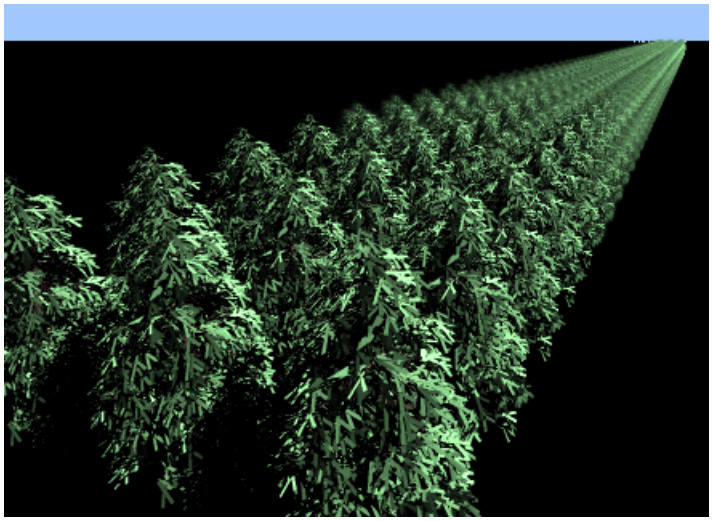


FIG. 3.4 – L'image obtenue par la méthode multi-échelle de Noma

L'approche de Chiba [CMDH97] permet de faire des arbres dont deux texels peuvent avoir une intersection non vide. Il génère ses textures 3D en découpant l'espace en voxels. Chaque voxel contient une densité, une couleur, et une normale obtenue en intersectant l'arbre avec chaque voxel (finalement il ne s'agit rien de plus qu'un échantillonnage volumique de l'arbre). Il calcule ensuite un modèle dégradé en moyennant les données du voxel calculé précédemment[1]. La normale est calculée à l'aide d'un opérateur de type Sobel. Le rendu dans le cas d'un seul texel à rendre est en fait l'algorithme utilisé par Kajiya et Kay [KK89], sauf pour l'illumination qui est un modèle déclairage diffus. On tire un rayon à travers le texel. On connait donc la cellule d'entrée et celle de sortie, on somme les contributions de chaque cellule en effectuant la somme de la contribution des opacités modulées par l'éclairage de la cellule. La clarté B d'un point de vue s'exprime comme :

$$\sum_{t=t_{near}}^{t_{far}} I_{r(t)} \cdot \exp(-\gamma \sum_{s=t_{near}}^{t} \rho(s)) \cdot$$

Le shéma 3.5 page 16 permet de comprendre cette équation. La formule d'éclairage est un modèle de

[1] Il calcule dans son papier un texel 256x256x256 dont il dérive un texel de résolution 32x32x32

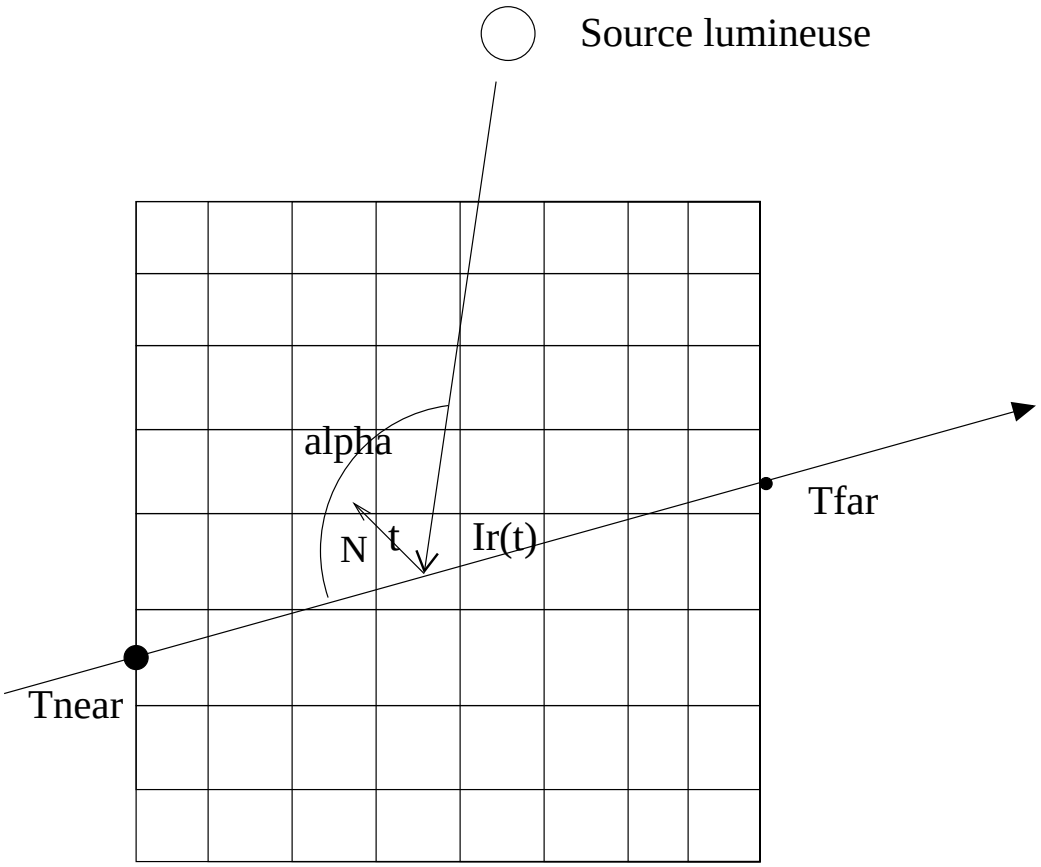


FIG. 3.5 – Le principe de rendu du texel de Chiba, pour la cellule correspondant à t, on évalue la contribution de l'éclairage de la cellule Ir(t)

Phong standard :

$$Ir(t) = \rho(t)I(t) \cdot (K_d \cdot cos\theta + K_s \cdot cos^n\alpha) + I_a \cdot K_d$$

où

- α désigne l'angle entre le rayon provenant de la source réfléchie par la normale en t et la direction de l'oeil partant de la cellule t
- θ désigne l'angle entre la normale de la cellule t et le vecteur issu de la source lumineuse vers la cellule t

Pour rendre plusieurs texels s'interpénétrant, il propose une méthode de sommation. En tout point x de l'espace, il faut déterminer un ensemble J, représentant toutes les pixels de textures 3D ayant une intersection avec ce point. Ensuite, la densité de cette cellule se calcule comme suit :

$$\rho(x) = \sum_{j \in J} \rho_j(x)$$

La densité de la somme de cellules de texels s'exprime comme la somme des densités des cellules de texels.

La normale se calcule comme :

$$N(x) = \sum_{j \in J} N_j(x)\rho_j(x)$$

La normale de la cellule est la moyenne pondérée par la densité de chaque normale.

La Couleur se calcule comme suit :

$$C(x) = (\sum_{j \in J} C_j(x)\rho_j(x))/\rho(x)$$

La couleur est la moyenne des couleurs de chaque cellule pondérée par leur densité. Pour gérer le niveau de détail, il choisit la version du texel détaillée dans le cas de point de vue proche, l'autre dans tous les autres cas. Les résultats qu'il a obtenus sont visibles sur la figure 3.6 page 18.

3.4 Les problèmes

Comme Neyret [Ney96b], les deux techniques précédentes sont reservées à un rendu de type raytracing, or cette approche n'est pas encore envisageable en temps-réel. Cependant, Meyer et Neyret [MN98] ont démontré que l'on pouvait adapter des texels au temps-réel en développant de nouvelles techniques, permettant potentiellement d'afficher des forêts en temps-réel. Notre stage a consisté à étendre cette technique, et à l'appliquer effectivement aux forêts.

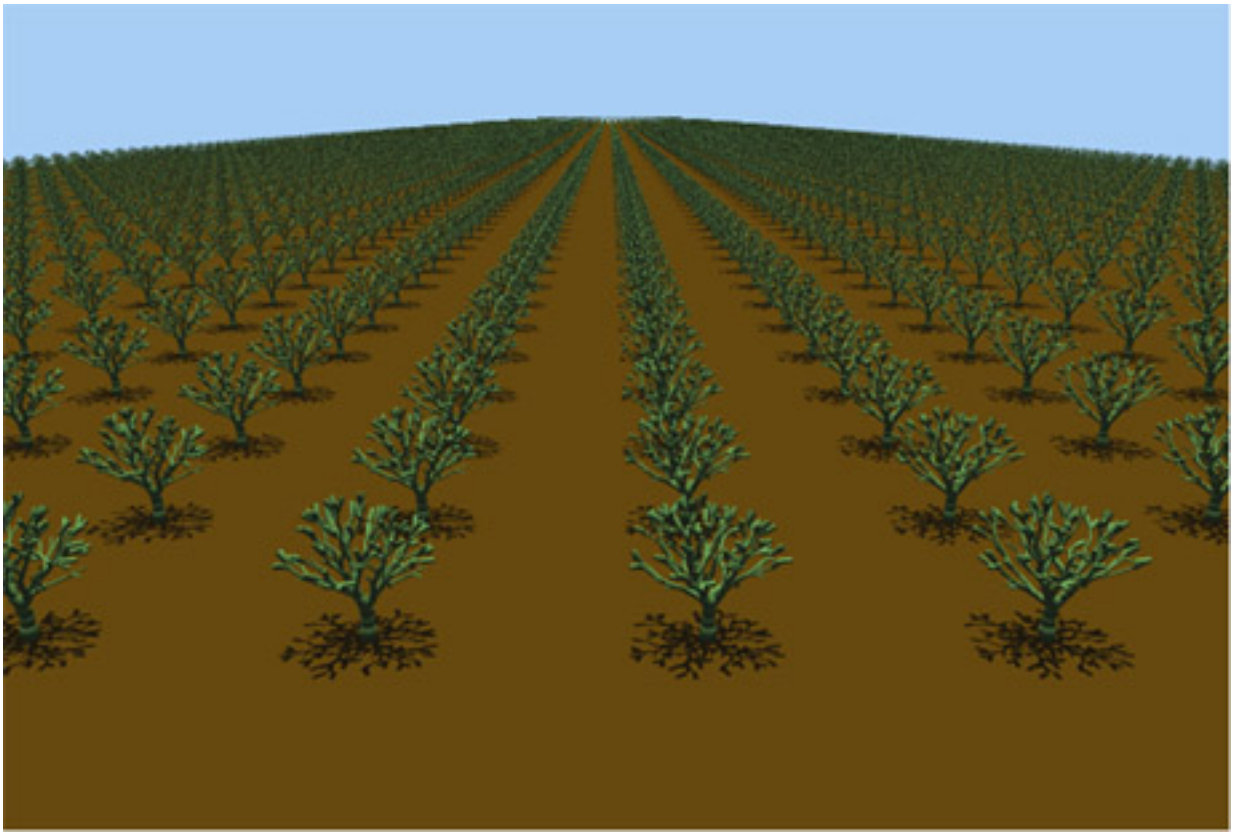


FIG. 3.6 – Les images calculées par le modèle de Chiba

Chapitre 4

Rendu temps réel haute qualité

Pour représenter des scènes complexes à l'aide de géométrie [FvDFH90], deux types d'approches existent :

- Les méthodes orientées objet (e.g rendus projectifs) qui résolvent la problématique : Pour une primitive géométrique donnée, quels sont les pixels de l'image à colorier et quelles sont leurs couleurs ?
- Les méthodes orientées image (e.g lancer de rayons (raytracing)) qui résolvent la problématique : Quelles sont les objets de la scène à prendre en compte pour le calcul de la couleur d'un pixel donné ?

Avec l'évolution de la puissance des machines, il est possible depuis une dizaine d'années de faire de la synthèse d'images 3D temps réel. La méthode employée utilise une méthode orientée objet : le rendu projectif par Z-Buffer. Le principe en est simple : En simplifiant la géométrie en ne travaillant plus qu'avec des surfaces discrétisées (triangles), il suffit de projeter chaque triangle à l'écran puis de déterminer quels pixels de l'images vont être allumés. Pour ceci, il suffit d'utiliser un algorithme d'occultation (par exemple un Z-Buffer qui autorise un pixel d'une primitive à être tracé uniquement si le pixel déjà tracé dans l'écran est plus proche de l'observateur). Les algorithmes de rendu à l'aide du Z-Buffer utilisent une méthode de remplissage des polygones de type scanline. Pour tracer un polygone, il faut déterminer les coordonnées de ses sommets projetées dans l'image, puis allumer chaque pixel de chaque ligne horizontale inclue dans le polygone. Le candidat pixel en cours de traitement s'appelle un **fragment**.

Pour accroître la qualité des images, il faut pouvoir «habiller »les primitives (triangles). Une technique courante pour peindre les surfaces est d'utiliser des textures. Une texture au sens général constitue un outil fondamental et pourrait se définir comme un réservoir d'informations potentiel par pixel. L'utilisation la plus banalisée de la texture consiste à plaquer une image (Texture2D) sur une surface. Afin de simuler à l'observateur l'éclairage des objets, la technique utilisée par OpenGL consiste à calculer en chaque sommet l'illumination locale, puis d'interpoler la couleur de cet éclairage sur le polygone.

Pendant longtemps, toutes ces opérations étaient calculées par le processeur central. Le problème est que le rendu d'une scène n'est que très rarement une fin en soi. Il est intégré dans un processus de calcul (d'animation par exemple). Afin de soulager le processeur de cette tâche, des constructeurs, SGI principalement, ont donc décidé de créer des matériels spécifiques pour effectuer des rendus de scènes. Cependant ces matériels étaient très chers car complexes, donc réservées aux professionnels. Sous l'impulsion du succès du jeu vidéo, ce genre de matériels a commencé à être développés sur des machines personnelles de type PC. Mais les coûts de production étant complètement différents pour toucher le marché des utilisateurs, les cartes PC sorties au début ne géraient pas la partie géométrique (projection), mais uniquement la phase de remplissage des polygones. Une partie très importante restait donc à la charge du processeur.

Depuis 1999, une nouvelle génération de cartes graphiques commencent à faire leur apparition. Ces cartes, introduites par NVidia, permettent entre autres :

- de faire prendre en charge par le processeur graphique toutes les opérations de transformations et projections des primitives comme dans les stations graphiques haut de gamme de type SGI (une première dans le marché des cartes grand public). Ceci est une grande nouveauté dans le monde PC. (NVidia s'est formée à partir de l'équipe de recherche et développement qui a conçu le graphique des SGI ce qui explique son savoir-faire)
- d'utiliser simultanément plusieurs unités de texture pour un même fragment ;
- de redéfinir la fonction de calcul de la couleur du fragment par pixel. Cette fonctionnalité est utilisée par exemple pour effectuer du bump mapping de façon correcte par point .

Dans ce chapitre, nous expliquerons tout d'abord en quoi consiste un pipeline opengl standard. Ensuite, je présenterais la nouveauté de ces cartes. Nous en présenterons plusieurs applications, notamment la simulation d'aspérités par des textures (bump mapping). Puis nous en présenterons l'application aux texels.

4.1 Le pipeline OpenGL standard

OpenGL est une librairie graphique standard pour faire du rendu 3d. Ce standard a été défini par SGI pour ses stations graphiques. De plus en plus, les cartes grand public supportent cette librairie, et permettent donc de l'utiliser sur PC. Cette librairie comporte en fait deux moteurs principaux, correspondant aux processeurs dédiés sur les SGI.

4.1.1 Le Geometric Engine

Ce moteur permet de spécifier des primitives (triangles, quads, polygones) dans l'espace 3D. On spécifie donc à OpenGL les coordonnées de chaque point ainsi que d'autres attributs (couleur, coordonnées de texture, normales ...). Le rôle de ce moteur est :

- d'effectuer les transformations géométriques (translation, rotation, mise à l'échelle)
- d'effectuer les projections de perspective
- d'écarter les polygones qui sont hors de l'écran ou derrière la caméra
- de tailler ceux qui sont à cheval (à la fois devant et derrière la caméra, à moitié en dehors de l'écran)

On obtient donc toutes les primitives dans le repère écran. Maintenant que toutes les opérations purement 3D ont été effectuées, il ne reste plus qu'à donner la main au Raster Engine qui va s'occuper de remplir les pixels de l'écran concernés par chaque primitive et du calcul du shading (l'illumination locale).

4.1.2 Le Raster Engine

Ce moteur ne fait que des opérations 2D. Il prend en entrée tous les polygones calculés précédemment, pour les rasteriser. Ceci revient à produire des pixels en interpolant les valeurs des différents attributs aux sommets et l'information de profondeur. Chacune de ces valeurs de profondeur (par pixel) est comparée avec la valeur présente dans le Z-Buffer : si elle est plus proche de l'oeil que celle du Z-Buffer, elle est affichée à l'écran, sinon elle est rejetée. Tant que la valeur du pixel n'est pas écrite à l'écran, elle s'appelle un *fragment*. OpenGL permet d'utiliser d'autres critères que le z pour refuser à un pixel d'être tracé. L'**alpha test** permet ainsi de définir une valeur d'opacité seuil (borne sup ou borne inf) permettant de rejeter tous les pixels dépassant cette valeur de seuil. Ce test est utilisé classiquement dans le tracé de billboards. Les billboards sont des images tracées toujours face à l'observateur ayant une opacité non nulle uniquement sur les pixels représentant les feuilles et le tronc. Ainsi, seuls les pixels représentant l'arbre sont tracés. Meyer et Neyret [MN98] l'utilisent afin de ne tracer que les pixels ayant une information de forme (une valeur d'opacité non nulle) dans le cadre de ses texels temps réel.

4.1.3 Attributs divers

Les textures

OpenGL en standard permet à l'utilisateur de définir par polygone une texture permettant d'habiller ce polygone. Pour cela, l'utilisateur donne au choix à OpenGL :

- un signal 1D
- une image (texture 2D),
- un volume (texture 3D) (NB : cette fonctionnalité n'est pas implémentée en hardware sur beaucoup de matériels ce qui revient à faire effectuer tous les calculs par le processeur central)

Ces textures peuvent servir à habiller le polygone, ou bien faire des lignes de niveau (cas du signal 1D), ou bien du rendu volumique. Pour s'en servir, l'utilisateur doit définir par sommet des coordonnées de texture.

La transparence

Les attributs qu'OpenGL permet de spécifier concernant la transparence sont l'opacité. Chaque valeur d'opacité est spécifiée par sommet. Si cette valeur vaut 1, on a un matériau opaque à cet endroit. On peut aussi utiliser des textures avec des informations de opacité (textures RGBA ou textures Alpha). Pour effectuer des effets de transparence, il faut utiliser un mode de tracé spécial appelé Blend. En fait la couleur d'un pixel est fonction de la couleur du fragment et de la couleur (et éventuellement de l'opacité) présente dans la mémoire écran. La fonction de mélange employée dans ce cas est :

$$Couleur_{ecran} = Opacite_{fragment} \cdot Couleur_{fragment} + (1 - Opacite_{fragment}) \cdot Couleur_{ecran}$$

4.1.4 Rendu complexe

Les limitations d'OpenGL (une seule texture par face, opérations par pixel limitées) requièrent souvent d'utiliser des techniques de rendu en plusieurs passes (appelé couramment **rendu multipasse**). Généralement, la technique consiste à effectuer un rendu de tous les objets sur lesquels on souhaite effectuer ces effets, puis de rendre une nouvelle fois tous ces objets avec d'autres attributs et une fonction de mélange d'image entre les deux rendus afin d'effectuer les opérations souhaitées. En effet, il est courant de précalculer les éclairages statiques (par exemple dans le cas de sc'enes d'intérieur) et dynamiques (e.g pour des roquettes proches d'un mur) puis de les utiliser sous forme de textures et de les mélanger avec la texture d'habillage. Le mélange des textures ne peut pas se faire avant de lancer le rendu. De même, afin de soulager le processeur pour les opérations de transformation et de projection, pour simuler des micro-aspérités de la surface, il peut être intéressant de disposer d'une information de perturbation de la surface

4.2 Les nouveautés de la nouvelle génération de matériel

La quête de réalisme d'une image en minimisant de plus en plus les temps de calcul est le Grâal de beaucoup de chercheurs et de l'industrie du jeu vidéo. Les limitations du modèle de calcul présenté précédemment se font radicalement sentir, dès qu'il s'agit d'accroître la qualité du rendu des surfaces. . On se rend donc bien vite compte des limitations imposées par en standard :

- On voudrait pouvoir disposer de plusieurs informations simultanément afin de moins utiliser le rendu multipasse. En utilisant le rendu multipasse, toute la géométrie doit être à nouveau retransformée par le geometric engine ce qui finit par coûter cher. En effet, on aimerait disposer de plusieurs textures et d'effectuer des opérations plus libres sur la façon de combiner leurs informations.

- On souhaiterais pouvoir utiliser d'une illumination moins rudimentaire (avec l'interpolation des valeurs des couleurs d'éclairage entre les sommets, les reflets spéculaires de type miroir nécessitent pour être vus une géomtrie très détaillée).
- On souhaiterais pouvoir disposer de plus de liberté sur la façon de calculer le mélange des couleurs d'un fragment

4.2.1 Le multitexture

OpenGL en standard 1.1 ne permet l'utilisation que d'une seule texture par face (triangle, quad ou autre). Comme les besoins pour le temps réel nécessitent de plus en plus d'information (exemple : une texture de couleur, une texture de relief, une texture de lumière) par fragment, la norme 1.2 intègre maintenant le multitexture. Cette fonctionnalité consiste à pouvoir utiliser plusieurs textures en même temps pour une même face, en définissant en chaque sommet une coordonnée de texture par textures comme une seule passe avec une seule fois l'information géométrique à transférer. En OpenGL standard 1.1, lorsque l'on veut utiliser plusieurs textures par polygones, on effectue autant de fois le rendu des polygones, en chargeant à chaque fois une nouvelle texture, tout en spécifiant comment on souhaite mélanger les images des rendus. La géométrie était donc transférée et transformée autant de fois qu'il y a de passes.

4.2.2 Les cube maps

Comme il est de plus en plus fréquent d'utiliser les textures comme des tables d'indirections, ce dispositif permet d'associer facilement une valeur (couleur, intensité...) à un vecteur (normale, direction de vue ...) : une cube map contient comme son nom l'indique 6 textures planes et la valeur à utiliser est celle pointée par le vecteur à partir du centre du cube. A noter que la norme du vecteur n'intervenant pas, on peut sans problèmes utiliser un vecteur provenant d'une interpolation. On peut appliquer une fonction de reflectance d'un matériau, le facteur de Fresnel, la couleur du ciel dans les diverses directions, etc .... Ce dernier point appelé «Environment mapping », consiste à utiliser une «texture de reflets ». En OpenGL 1.1, seule une implémentation limitée était disponible, interdisant de changer le point de vue. D'autres approches comme celles d'Heidrich et Seidel [HS99] permettaient de passer outre la limitation du point de vue inhérente à la technique utilisée par OpenGL 1.1. Cependant, cette technique n'offrant pas autant de liberté que le cube map, elle n'a pas été retenue.

4.2.3 Les vertex shaders

L'idée des vertex shaders est d'enrichir le géométric engine d'OpenGL en permettant des effets sur la géométrie et les attributs par sommets (vue fish-eye, rendu non photo-réaliste) tout en continuant d'utiliser l'accélération matérielle. OpenGL fournit par l'intermédiaire d'extensions accès à l'écriture de programmes. Ces vertex shaders ne génèrent pas de géométrie, ils laissent libre choix à l'utilisateur de faire les transformations géométriques (affines,projectives ou non) sur le point courant, de générer les coordonnées de texture, les couleurs, dynamiquement. Une fois le micro-programme executé, le géométric engine reprend la main afin d'effectuer le clipping.

4.2.4 Les pixels shaders

Le grand problème d'OpenGL par rapport au raytracing est de ne pas permettre de contrôle sur les fonctions de calcul de couleur au pixel près. Les fonctions de shading sont prédéfinies, les fonctions de mélange des couleurs aussi. Pire : le shading n'est évalué qu'aux sommets du maillage et c'est la couleur résultante qui est interpolée le long des polygones. Le concept des pixels shaders est de donner la main au programmeur dans le Raster Engine, une fois toutes les informations du fragment sont calculées :

couleur, normales, couleurs de textures, etc...L'idée de langages type Renderman (orienté ray-tracing) est de laisser à l'utilisateur la liberté de définir sa fonction de mélange des couleurs ainsi que sa fonction de shading. Jusqu' à la sortie de la NVidia GeForce 256, le rendu de tels effets en rendu projectif nécessitait plusieurs passes de rendu, ce qui conduit à effectuer plusieurs fois la même transformation de la scène puis à mélanger les deux images produites de façon appropriée. La librairie OpenGL shader de SGI [SGIb] permet d'automatiser cette traduction de shader complexe type Renderman en rendu multipasse, mais comme on l'a vu, le coût peut être très cher. L'idée du pixel shader est de permettre de définir une fonction de mélange de couleurs et de shading en une seule passe. NVidia a été le premier industriel à implémenter le concept dans ses processeurs GeForce. Le reste de l'industrie est en train de le suivre, comme ATI qui l'implémente dans son Radeon2. Microsoft a développé une couche d'abstraction dans DirectX 8 pour permettre de programmer des shaders indépendamment du processeur graphique employé. OpenGL permet la programmation de shaders par le biais d'extensions (fonctions supplementaires à la librairie OpenGL) dépendantes du processeur employé. Le concept des pixels shaders est d'offrir au programmeur un contrôle de la couleur finale à partir des diverses données interpolées par le raster engine. L'utilisateur écrit un micro-programme, avec des variables et des opérations spécifiques aux pixels shaders. Les données accessibles, sont :

- la couleur et l'opacité du fragment (on distingue une couleur primaire et une couleur secondaire)
- la couleur et l'opacité obtenue dans chaque texture active du multitexture
- deux constantes définies par l'utilisateur
- le brouillard (couleur + opacité)
- la couleur et l'opacité diffuse (obtenue par interpolation des valeurs fournies aux sommets)
- la couleur et l'opacité ambiante (idem)
- la couleur et l'opacité spéculaire (idem)

Ces *pixels shaders* offrent aussi des variables temporaires afin de pouvoir effectuer des calculs à ce niveau. Les opérations fournies sont :

- le produit scalaire
- le produit composante à composante
- l'addition

Une des particularités de ces shaders est que chaque couleur peut être interprétée comme un vecteur 3D. On peut donc utiliser les couleurs comme des vecteurs, afin de fournir des paramètres vectoriels au shder chargé dans le processeur graphique. Des convertisseurs couleurs/vecteurs sont disponibles dans le processeur, ce qui nous permet d'utiliser des couleurs comme des vecteurs.

4.2.5 Exemple d'implémentation des pixels shaders : les registers combiners

Sur la carte NVidia GeForce, les pixels shaders sont implémentés sous la forme d'une extension OpenGL NV_register_combiners [SGIa]. En fait, à un haut niveau, un pixel shader utilise une algèbre (ensemble + opérations) signée par fragment. Les opérations de cette algèbre sont le produit scalaire, la sélection, la multiplication terme à terme. Chaque opérande peut être soit une couleur, soit un vecteur (provenant d'une conversion de couleur), soit un scalaire (transparence). Les valeurs initiales (par fragment) de ces opérandes sont les données calculées par le raster engine pour ce fragment (couleurs, constantes définies par l'utilisateur, valeur des couleurs de texture pour ce fragment, valeurs de brouillard). Le pixel shader est écrit comme une suite d'opérations. Pour permettre l'execution du pixel shader, NVidia a construit un pipeline d'opérations par fragment. Ce pipeline est divisé en étages. Au sein d'un étage, un nombre défini d'opérations est autorisé. Le nombre d'étages standard est de deux sur une NVidia GeForce 2. Ensuite un étage particulier est situé en fin de pipeline pour calculer la couleur et l'opacité du fragment. Au sein d'un étage, les opérations s'effectuent d'une part sur des vecteurs ou des couleurs (canal RGB), et sur des valeurs d'opacité d'autre part (canal ALPHA). Les opérations effectuées sur le canal RGB sont indépendantes du canal ALPHA. On peut au plus par canal et par étage effectuer deux multiplications (ou produit scalaire) et une addition. Ensuite, on peut propager les résultats dans l'étage

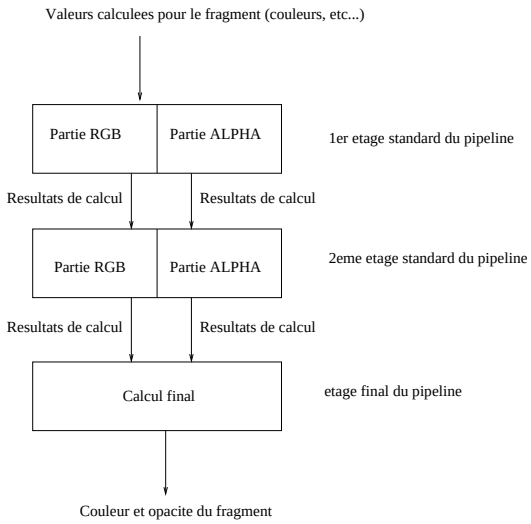


FIG. 4.1 – Le mécanisme des registers combiners : un pipeline d'opérations pour calculer le pixel shader entre les données du raster engine et la couleur finale du fragment

suivant du pipeline. Le résumé de ce principe est visible sur la figure 4.1 page 24.

Contraintes

On peut choisir de n'utiliser qu'un seul étage pour l'execution du pixel shader. Lorsque l'on utilise deux étages au lieu d'un seul, on divise le fillrate (le taux de remplissage de polygones) par 2. Ceci impose de trouver des astuces pour pouvoir tout faire rentrer dans une seule passe de rendu et deux étages.

4.3 A titre d'exemple illustratif : Bump-mapping en Hardware

Afin de simuler des aspérités, il existe plusieurs méthodes (classées par ordre de coût en temps de rendu) :

- créer explicitement la géométrie, cette opération est alors très chère pour le rendu, car tous les points de relief doivent être construits puis transférés et projetés par le Geometric Engine. Cependant, cette technique permet d'obtenir un très bon éclairage
- stocker la géométrie sous la forme d'une texture de hauteur, (appelée alors displacement map). C'est un bon compromis car l'utilisation de la texture ne nécessite que de projeter les sommets principaux de la surface. Kautz et Seidel [KS01] ont présenté une technique basée sur le principe des texels pour en faire en temps réel. Cette technique est intéressante car elle permet de pouvoir voir les aspérités sous un angle rasant. Pour le rendre, ils génèrent des tranches de hauteur.
- utiliser la technique du bump-mapping de Blinn [Bli78]. La technique consiste à encoder dans une texture (appelée bump map) la déformation (sous forme de normales, carte de hauteur, ou bien de différences de normales) de la surface, et de calculer l'équation de la lumière par point afin de simuler le comportement à la lumière de la surface. Le comportement lumineux simule à l'observateur la surface déformée. Le problème de cette technique est qu'elle ne crée pas de géométrie, ce qui se voit sous un point de vue ou d'éclairage rasant, car l'observateur voit que la surface est plate.

Nous détaillerons uniquement la technique de Kilgard [Kil00] utilisant les pixels shaders car elle offre un bon compromis entre performance et qualité visuelle. L'idée de la méthode est d'utiliser une carte de

normales perturbées sur la surface, dans le but de calculer la formule de shading par point afin de simuler l'effet.

4.3.1 Présentation de l'éclairage

Selon le modèle de Blinn-Phong [FvDFH90], la couleur d'un fragment s'exprime comme suit :
$I = I_{ambiant} + K_{diffuse} \cdot max(0, L.N) + K_{specular} \cdot max(0, H.N)$
où

- N désigne le vecteur normal à la normale locale au point courant
- L désigne le vecteur lumière (issu du point pointant vers la source de lumiere ponctuelle)
- H désigne le vecteur L+V (V désigne le vecteur issu du point pointant vers l'oeil)

Afin que les calculs soient corrects, chaque vecteur doit bien sûr normé (de norme 1). Dans le cadre du calcul effectué par OpenGL, ce calcul d'illumination est fait au sommet, puis les valeurs de ce calcul sont interpolées entre les sommets. Le calcul est donc dégradé. L'idée du papier est d'utiliser les nouveautés du hardware, pour effectuer le calcul de façon correcte. De façon habituelle, tous ces vecteurs sont exprimés dans le repère monde. L'idée du papier est d'effectuer le rendu en 3 passes :

- Un rendu pour l'intensité de l'illumination diffuse
- Un rendu pour la couleur de l'objet
- Un rendu pour le spéculaire

La formule de rendu appliquée appliquée pour la première passe de rendu est :

$$I = I_{ambiant} + K_{diffuse} \cdot S_{self} \cdot max(0, \vec{L} \cdot \vec{N'}))$$

où

$$S_{self} = max(0, \vec{L} \cdot \vec{N})$$

où N désigne la normale originale de la surface (le point sur lequel est calculé l'éclairage). Le terme S_{self} désigne une composante d'auto-ombrage. La normale N correspond à la normale au niveau de détail le plus gros, et N' désigne la normale au niveau de détail le plus fin. Ceci permet de pénaliser les éléments qui apparaîtraient lumineux alors que vraisemblablement, la lumière ne peut atteindre (voir fig 4.2 page 25) Pour effectuer son calcul, les normales perturbées vont être exprimées dans le repère tangent, défini

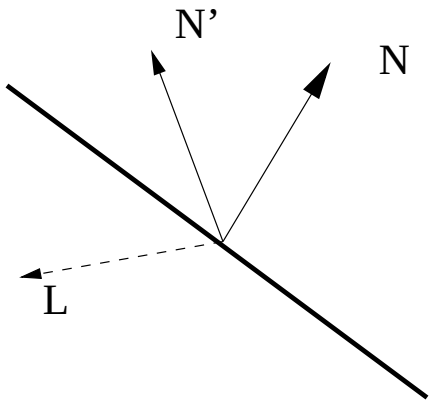


FIG. 4.2 – Le cas où le point ne doit pas être éclairé, la surface principale est opposée à la lumière, alors que le micro-relief est orienté vers la lumière

par la base (Tangent,BiNormal,Normal). Ce qui donne un repère par sommet de facette. Tous les vecteurs (normales et lumière) seront exprimées dans ce repère. Toutes les normales exprimées dans la texture sont

de norme 1. Pour pouvoir renormer le vecteur lumière par point, il utilise une cube map, qui comme nous l'avons indiqué précédemment permet d'associer une valeur à toute direction de l'espace. Nous réglons la cube map afin qu'elle renvoie le vecteur unitaire associée à la direction passée en paramètre. Ce mécanisme est expliqué sur la figure 4.3 page 26 Si dans les faces de la cubemap, on crée des textures qui pour

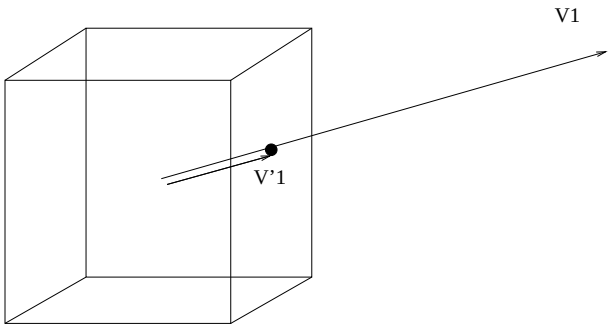


FIG. 4.3 – Étant donné le vecteur V1, le cube map se sert du vecteur colinéaire V'1 intersectant le cube

tout V'1 associe V"1 correspondant à V'1 de norme 1, on obtient un système efficace de renormalisation. En effet, si on donne aux sommets la direction du vecteur lumière à la cube map, le raster engine va remplir la facette triangulaire à l'aide du triangle de texture délimité par les directions aux sommets, on obtiendra bien une normale de norme 1 pour chaque direction du vecteur lumière. Ensuite, le calcul du diffus, comme celui du spéculaire est fait dans le *pixel shader*.

4.4 Application aux texels

Les pixels shaders, par leur liberté de programmation permettent beaucoup de choses. La solution suggérée par Rezk-Salama, Engel, Bauer, Greiner, Ertl,[RSEB+00] pour effectuer une interpolation trilinéaire des données du texel est de rajouter des tranches intermédiaires lors du rendu. L'idée est de considérer que le texel dispose certes d'un nombre de tranches de textures 2D fini en mémoire, mais qu'on peut simuler l'interpolation trilinéaire des données en calculant à la volée des textures 2D entre deux texture 2D du texel, et en rendant un polygone situé entre les deux tranches de texture avec cette texture. Cette texture est créée en interpolant linéairement les valeurs entre les deux tranches de référence. Cependant le calcul de la texture avec le paramètre d'interpolation nécessiterait de créer une texture en mémoire, la charger dans la carte graphique, on perdrait donc le temps-réel. L'idée est de charger lors du rendu de ce polygone les deux textures les plus proches de ce polygone en mémoire, puis d'effectuer le calcul d'interpolation dans le *pixel shader*. Le calcul pour la tranche $S_{i+\alpha}$ est fait de cette façon :

$$S_{i+\alpha} = (1-\alpha) \cdot S_i + \alpha \cdot S_{i+1}$$

De cette façon, on peut supprimer les artefacts dûs au fait que le texel a un nombre de tranches trop faible (on voit le bord des tranches).

Afin de permettre aussi de réduire la complexité de rendu lorsque le point de vue est trop éloigné, cet article propose une méthode de fusion des tranches (rendu de chaque contribution dans un seul polygone). Pour se faire, il choisit de fusionner deux tranches successives en même temps. Il détermine les coordonnées du plan portant le polygone le plus éloigné, puis il détermine les coordonnées de textures pour ce plan pour la texture la plus éloignée, puis pour la plus proche de l'observateur (voir figure 4.4 page 27). Ensuite il règle le *pixel shader* afin de pouvoir effectuer le calcul de la contribution de la tranche la plus proche sur la plus éloignée. Le rendu de ces méta-tranches s'effectue dans l'ordre éloigné-proche

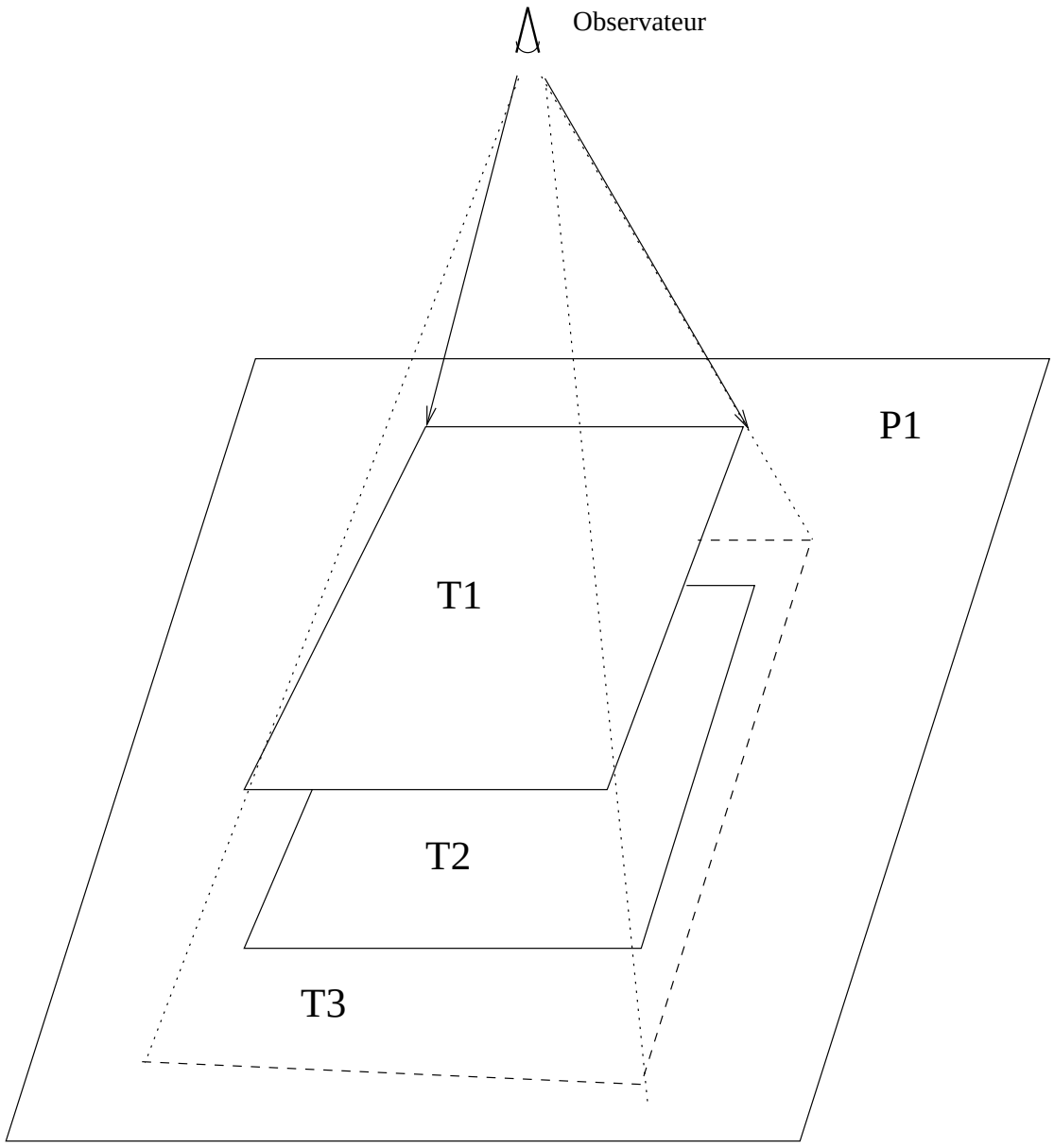


FIG. 4.4 – La tranche de texture T1 est reprojetée sur le polygone P1, en fonction de cette projection, les coordonnées de texture au sommet de P1 sont calculées, de même que celle de T2 dans P1

avec une fonction de mélange différente celle utilisée d'habitude :

$$C' = C_s * 1 + C_d \cdot A_s$$

où C_s désigne la couleur du fragment en cours et C_d désigne la couleur présente dans l'image. De cette façon, le calcul reste identique à celui de la méthode de Meyer.

Afin d'étendre le réalisme de cette méthode, les auteurs proposent une technique pour effectuer le calcul de l'éclairage des texels en une seule passe utilisant les pixels shaders et la méthode d' «interpolation trilinéaire »des tranches. La méthode consiste à utiliser un *pixel shader* un peu plus complexe que le précédent. Il utilise le même principe de chargement de deux tranches de textures adjacentes, en interpolant les données à l'aide du paramètre d'interpolation. Ces textures contiennent les normales à chaque cellule de voxel. Il interpole donc les normales. En passant la direction de la lumière, il s'inspire de la technique de Kilgard pour le bump mapping. Ensuite, dans le *pixel shader* il calcule :

$$C_f = C_a + C_d((\alpha \cdot G_i + (1 - \alpha) \cdot G_{i+1}) \cdot \vec{L})$$

Où

- C_f désigne la couleur du fragment calculé
- G_i désigne le gradient (normale) calculée dans la tranche i
- G_{i+1} le gradient dans la tranche i+1
- C_d la couleur de la lumière diffuse
- C_a la couleur de la lumière ambiante

Le principal avantage de cette méthode est qu'elle est très rapide à utiliser et qu'elle permet d'obtenir un calcul d'éclairage. Cependant, ce calcul est assez peu rigoureux. En effet, le vecteur lumière n'est pas renormalisé car interpolé puisque donné seulement en chaque sommet. D'autre part le calcul des normales des tranches intermédiaires est correct pour la direction, mais pas pour la norme. En effet, l'interpolation de deux vecteurs donne la bonne direction, mais pas la même norme (voir figure 4.5 page 28). Comme

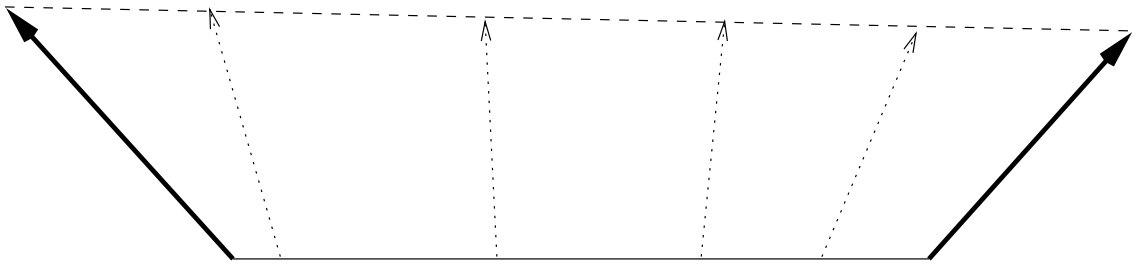


FIG. 4.5 – En gras les deux vecteurs de norme 1, en pointillés long l'interpolation, en pointillés courts quelques vecteurs résultants de cette interpolation

le calcul suppose que les vecteurs lumière et normaux soient de norme 1 pour le calcul de l'illumination diffuse, le calcul est une solution intermédiaire entre qualité et rigueur. En effet, la difficulté est lorsque deux normales sont interpolées dans un texel, elles peuvent être très différentes d'une tranche à l'autre, le calcul est donc très faussé par cette méthode.

Les auteurs présentent une technique de rendre des texels contenant à la fois une couleur, une densité, et une normale par cellule. Il s'agit d'utiliser deux textures par tranches, puis de régler le *pixel shader* pour effectuer le calcul de l'illumination. Comme cette technique a fait partie de mon stage, je la détaillerais dans la partie personnelle.

Chapitre 5

Texels réalistes temps réel

Le but de ce stage était de pouvoir étendre la technique de Meyer et Neyret [MN98] de manière à rendre des texels réalistes en temps réel en exploitant les capacités des cartes graphiques. Neyret, Meyer, Noma, Chiba avaient montré que le modèle des texels était bien adapté au rendu de forêts. Cependant les approches utilisées jusqu'alors présentaient des limitations :

- les modèles les plus réalistes sont basés sur du ray-tracing, donc très loin du temps-réel
- les modèles temps-réel n'ont pas de shading , et sont donc peu réalistes

Nous avons essayé de faire reculer ces limites en unifiant temps-réel et réalisme. Nous détaillerons donc les limitations des modèles actuels d'une part, nous présenterons ensuite les hypothèses utilisées pour mettre au point notre modèle. Nous rentrerons enfin dans le vif du sujet, à savoir notre modèle de texel, dont nous étudierons les limites et les performances. Enfin nous suggérerons quelques améliorations à court et à long terme.

5.1 Les limitations des modèles actuels

Habituellement, pour représenter des forêts, un système de rendu hiérarchique est utilisé. Selon la distance à laquelle nous sommes d'un arbre, on utilise un modèle d'arbre différent :

- de près une représentation géométrique détaillée ;
- à une distance intermédiaire, des modèles simplifiés ou des représentations alternatives (comme par exemple des objets multi-texture dans le cas de Chaudy [Cha97]) ;
- à une grande distance, des approches orientées texture comme des billboards, qui correspondent à des images semi-transparentes toujours perpendiculaires à la direction de visée.

Afin de montrer que les texels sont utilisables en temps réel dans les scènes naturelles, nous avons essayé de réaliser des scènes complexes à l'aide des texels. Jusqu'alors, aucun modèle de texel temps-réel ne permettait de prendre en compte une orientation variable de la lumière avec un éclairage variable[1], même si la possibilité de le faire découle implicitement [RSEB+00] qui s'intéresse à un contexte différent, personne ne l'a employé pour faire des scènes naturelles. nous avons donc tenté de démontrer pendant ce stage que cette approche était appropriée et même particulièrement performante tant en qualité qu'en coût.

[1] par exemple le modèle de Meyer et Neyret [MN98] nécessitait de regénérer les texels dès que la lumière bougeait

5.2 Conditions expérimentales

Nous nous sommes placés dans le cas où les forêts sont éclairées par le soleil (lumière à l'infini). Nous avons négligé dans l'éclairage de nos texels la composante spéculaire, et nous n'avons considéré que les composantes diffuses de la couleur, et ce pour plusieurs raisons :

1. Les reflets spéculaires d'un arbre interviennent au niveau des feuilles. Notre modèle de texels n'est pas très adapté pour simuler cet effet :
   - le tronc n'étant pas spéculaire, il faudrait traiter séparément le tronc et les feuilles, multipliant ainsi le coût de stockage et de traitement
   - Contrairement au modèle de Neyret [Ney96b], nos reflets sont pilotés par une seul normale. Or les reflets sont extrêmement sensibles aux variations des normales, aussi il faudrait stocker les normales à une résolution élevée
2. La taille mémoire pour les textures étant limité par la carte graphique, la résolution de tranches de textures doit être raisonnable (en haute définition, chaque tranche a une résolution de 128x128), ce qui n'est pas un niveau de détail suffisant pour permettre de calculer ce genre d'effets car les reflets spéculaires seraient de la taille de la feuille représentée.
3. Le *pixel shader* étant un micro-programme de taille limitée, et vu la faible longueur des micro-programmes autorisés par le hardware graphique actuel, le *pixel shader* correspondant à cette formule d'éclairage ne «tiendrait » pas dans le nombre d'opérations réalisable en 1 seule passe par la carte graphique de référence. Ceci nécessiterait un rendu en plusieurs passes, ce qui multiplierait le coût par 2.
4. A l'opposé du plastique et du métal, les reflets des végétaux sont très étalés, et donc moins marqués

5.3 Notre représentation

Nous sommes partis du modèle de Meyer et Neyret [MN98] qui consistait à représenter le volume comme des tranches de textures parallèles entre elles. Nous avons à stocker les tranches de texture dans 3 directions, afin de pouvoir toujours disposer d' une direction de tranches orthogonale à celle de la visée. Ceci permet une occultation anisotrope essentielle pour les feuilles (et que ne permettrait pas une représentation en volumes) : vue sur la tranche, une feuille est quasi-invisible. Afin d'étendre ce modèle, nous stockons un système de textures similaire à celui présenté dans [RSEB+00] : nous représentons le volume par un système de tranches de textures au format de couleurs RGBA (couleur RougeVertBleu, A pour alpha le canal d'opacité). Pour chaque tranche nous stockons ainsi la couleur et la normale.

5.3.1 Notre modèle de texels

Alors que Meyer n'utilisait qu'une seule tranche de texture par tranche de volume, nous en utilisons deux :

- Une tranche codant simultanément la couleur (au format RGB) et la densité dans le canal alpha de la texture (i.e. une opacité)
- Une tranche codant les normales

L'idée est de repartir de la technique de bump-mapping pour simuler des aspérités sur une surface. Comme cette technique consiste à simuler du relief en calculant en chaque point l'éclairage de la surface en se référant à la normale locale perturbée, nous avons décidé de s'inspirer de la représentation des tranches de volume. En effet, ceci nous permet de simuler la réflexion de la lumière en chaque voxel du texel. Pour pouvoir effectuer le calcul du *pixel shader*, il faut se plier à quelques obligations : notamment à celle que toutes les normales d'une tranche de textures soient exprimées dans le repère (tangent,binormal,normale) à la surface. Comme les tranches de textures sont toutes de même taille et parallèles entre elles, les repères

(tangent, binormal, normal) sont tous identiques en chaque sommet. La direction de la lumière exprimée dans chacun de ces repères sera toujours le même vecteur car positionnée à l'infini.

5.3.2 Le modèle de texture

Représentation des normales sous forme de textures

La texture de normales doit être encodée dans une texture RGBA. La composante alpha devait nous servir à pouvoir stocker des informations servant à moduler l'influence de l'éclairage à savoir une forte ou faible réflexion du matériau. Les *pixel shaders* ayant une taille trop limitée, nous n'avons pas pu utiliser cette information. Comme les couleurs sont dans un espace [0..1]x[0..1]x[0..1] et que les normales sont dans un espace [-1..1]x[-1..1]x[-1..1]. On applique la même technique que décrite dans l'état de l'art. La technique de conversion est :

$$Composante_{rouge} = \frac{Normale.x}{2} + \frac{1}{2}$$

$$Composante_{vert} = \frac{Normale.y}{2} + \frac{1}{2}$$

$$Composante_{bleu} = \frac{Normale.z}{2} + \frac{1}{2}$$

A chaque pixel de la texture de couleur doit correspondre à un vecteur de normale de norme 1. Les *pixels shaders* offrent la possibilité au programmeur de faire la conversion inverse avant de faire les calculs vectoriels d'éclairage. Ce système de stockage est donc particulièrement adapté.

Représentation de l'information de densité

L'information de opacité est stockée dans le canal α (le A de RGBA). Pour une densité maximale, $\alpha = 1$ et pour une cellule de voxel vide $\alpha = 0$.

Représentation de l'information de couleur

L'information de couleur correspond à la composante diffuse du matériau contenu dans la cellule.

5.4 Modèle de rendu d'un arbre

L'idée générale est d'utiliser le pixel shader le plus court en nombre d'instructions, afin de maximiser la performance tout en calculant une fonction d'éclairage de qualité suffisante. Le problème des pixel shaders disponibles dans les cartes actuelles est qu'ils sont de taille très limitée et disposant d'un jeu d'instructions limité. Une première difficulté est donc de réussir à écrire un shader tenant dans la taille maximale imposée par le matériel. Ceci nous contraint donc à utiliser des modèles d'éclairages simples de type modèle d'illumination diffuse ou Phong [FvDFH90], plutôt que des modèles comme ceux utilisés par les approches raytracing à base de BRDF (Bidirectional Reflectance Distribution Function). La prochaine génération de cartes devrait disposer de shaders plus compliqués dans lequel nous pourrons peut-être utiliser ce genre de modèles à moindre coût. A l'heure où ce rapport est écrit, les pixels shaders ne sont capables d'effectuer que des produits scalaires, des additions, et des produits composante à composante. Ceci rend difficile l'écriture de shaders de fonctions compliquées en une seule passe. Cependant, des solutions (développement limités, tables d'indirections ...) commencent à apparaître dans la communauté des développeurs de shaders temps-réel [OGL]. De plus, il existe aussi une tendance à utiliser le matériel à utiliser le matériel pour faire du rendu de qualité sans contrainte de temps-réel, passant par l'écriture de shaders très complexes nécessitant de nombreuses passes.

5.4.1 Version simple

Afin de commencer notre système de rendu, nous avons essayé de créer un shader capable de faire le rendu par pixel de l'éclairage et le rendu. La formule utilisée est :

$$C_{fragment} = (C_{lampe} \times (min(max(\vec{N} \cdot \vec{L}, 0), 1) + I_{ambiante}) * C_{textureCouleur}$$

$$Opacité_{fragment} = Opacité_{textureCouleur}$$

La couleur de la texture de couleur est donc modulée par le résultat de l'éclairage. On considère que la lumière est à l'infini. Le vecteur lumière est normé en tout pixel car constant puisque par hypothèse la source lumineuse est à l'infini. Tous les vecteurs stockés sous forme de couleurs dans la texture de normales sont renormés eux aussi afin d'obtenir un produit scalaire correct. Lors de la génération des tranches de normales, nous stockons sous forme de couleurs les normales normées en chaque cellule. Nous n'avons donc pas de calcul de renormalisation particulier à effectuer au moment du calcul de l'éclairage. Grâce à l'utilisation du multitexture, et des pixels shaders, tout ceci tient en un étage de calcul des registers combiners sur une NVidia GeForce 2, ce qui permet de faire l'affichage du texel aussi rapidement qu'un texel dans [MN98] sur la même carte graphique.

5.4.2 Version avec brouillard

Le brouillard, même très léger, joue un rôle essentiel dans les scènes naturelles type paysage : à cette échelle, l'air n'est jamais totalement transparent (Bleuissement dû à la diffusion de Rayleigh, blanchissement du à l'humidité). Cet effet, bien compris des peintres, participe largement à l'impression de profondeur. A contrario, les images de synthèse ne s'en servent pas toujours, ce qui fait que l'on a du mal à distinguer la distance, à cause de la saturation du champ visuel d'éléments ayant tous le même aspect. Nous avons donc choisi de prendre en compte cet effet dans notre implémentation.

Les cartes graphiques utilisent une technique particulière pour rendre cet effet : le calcul du fragment est modifié en fonction de la distance à la caméra. Après avoir réalisé le calcul de la couleur du fragment, la formule de calcul est :

$$C'_r = f \cdot C_r + (1 - f)C_f$$

où

- C'_r désigne la couleur du fragment sous l'influence du brouillard
- f désigne l'opacité du brouillard ($= \exp^{-\tau \cdot \rho \cdot z}$)
- C_f désigne la couleur du brouillard

Nous avons pu construire le pixel shader en appliquant cette formule à notre formule d'illumination précédente. La formule de calcul est :

$$C_{fragment} = f \times ((Clampe \times (min(max(\vec{N} \cdot \vec{L}, 0), 1)) + I_{ambiante}) \times C_{textureCouleur}) + (1 - \alpha_{fog}) \times C_{fog}$$

$$Opacité_{fragment} = Opacité_{textureCouleur}$$

5.5 Modèle amélioré

L'idéal dans le cas des feuilles serait de disposer de deux normales opposées. En effet, comme dans notre modèle actuel, on a une seule normale par feuille. Le problème vient du fait que si on regarde une feuille par dessous et que la normale est orientée vers le ciel, si la source lumineuse est positionnée du sol vers le le ciel, le calcul de l'éclairage donnera toujours une intensité nulle. Cependant, comme nous avons affirmé dans nos hypothèses que la lumière était positionnée à l'infini,et que nous tentions de simuler une lumière de type soleil, donc cette configuration (soleil éclairant par dessous) ne peux pas arriver. D'autre part, la géométrie de l'arbre étant plutôt sphérique, il suffit de choisir de stocker dans le texel pour chaque

feuille la direction de normales tournée vers l'extérieur. Ceci permet d'obtenir de bons résultats. Dans l'absolu, il faudrait modifier le shader pour calculer l'éclairage de cette façon :

$$C_{fragment} = (C_{lampe} \times (max((\vec{N} \cdot \vec{L}), -(\vec{N} \cdot \vec{L}))) + I_{ambiante}) * C_{textureCouleur}$$

Le problème actuel des shaders est qu'il ne permettent pas d'opérations max en standard, ni d'opérateurs simples de sélection. Pour le moment, ce n'est pas calculable seulement par un calcul de shader, il faut rajouter plusieurs passes de rendu. Ceci serait donc trop contraignant pour notre objectif. Nous conservons donc la précédente méthode.

5.6 Filtrage des éléments de texture

Comme par défaut, l'utilisation de textures de résolution finie fait que à un espace continu d'adressage de coordonnées de textures on associe des données en créneau, on assiste à des sauts de valeurs. Dans le cas d'une surface regardée de près, on va voir des zones uniformes de couleurs carrés de texture sur la surface. Pour vaincre ce problème, on utilise un système de filtrage des données en interpolant les valeurs de la texture. Le filtrage de la texture est le filtrage bilinéaire standard, c'est à dire que la couleur du pixel est obtenu en interpolant la couleur des pixels de texture les plus proches, lorsque la taille d'un pixel de l'écran est inférieure à celle d'un pixel de texture. Ceci est une technique d'antialiasing courante en image de synthèse. Afin d'éviter que l'observateur puisse distinguer la résolution des pixels de texture utilisés lorsqu'il est près du texel, nous avons autorisé ce filtrage. Cependant, ceci pose plusieurs problèmes :

- des pixels vont «baver »sur la tranche de derrière
- les deux textures par tranches doivent utiliser le même type de filtrage. Si on n'utilise pas le filtrage pour les normales, on voit sur chaque tranche des zones plus sombres carrées autour de formes douces, ou dans le cas où seules la texture de normale est filtrée, la texture de couleur donne des zones tranchées de couleur lors du rendu, ce qui est visible à l'oeil.

Lorsque l'on utilise des normales sous forme de textures et qu'on applique un filtrage bilinéaire, on effectue un calcul d'éclairage car les vecteurs résultants ne sont pas renormés. Ce problème n'est jamais évoqué dans la littérature, et a fortiori non résolu [2]. En effet, les données d'un texel sont faiblement échantillonnées. On peut donc avoir des normales variant de façon brutale d'un pixel de texture à l'autre. Ceci signifie que puisque les normales sont interpolées entre deux pixels, elles ne sont plus renormalisées. Ce qui implique que le calcul devient faux car il faudrait diviser par la norme de cette normale. Ceci est un problème très classique en image de synthèse mais personne ne l'a réellement résolu pour le moment. Une solution serait d'augmenter la résolution des textures afin de minimiser l'erreur pour un point de vue proche, mais la limite de taille mémoire se fait sentir et le cas des points devenus lointains ne serait pas amélioré. Pour des texels de résolution 64x128x128x2x4x3 (64 tranches de textures de résolution 128*128, chaque tranche contenant une texture de normale et une de couleur, au format rgba chaque composante sur 8 bits, le tout dans 3 directions principales) requiert déjà 24 Mo de mémoire texture, ce qui est beaucoup puisque les cartes actuelles ne dépassent que très rarement 32 Mo. La contrainte sur la mémoire serait donc trop forte pour ce genre de matériel.

Il existe cependant une alternative à ceci : il s'agit de la compression de textures. OpenGL 1.2 vient d'introduire une extension importante : GL_ARB_Texture_Compression [SGIa]. Cette extension permet de bénéficier d'une compression en mémoire texture pour chaque tranche, le texturage s'effectue de façon

[2] Dans Rezk-Salama et al [RSEB+00], les auteurs ne considèrent pas que leur lumière est à l'infini ce qui entraîne aussi que leur produit scalaire $\vec{N} \cdot \vec{L}$ n'a que très peu de sens car ni L, ni N ne sont normés, puisque N est résultat de l'interpolation bilinéaire de la texture, et L le résultat de l'interpolation linéaire de quatre vecteurs lumière normés passés en paramètres aux sommets de la tranche. Or le terme calculé par cette équation est censé être $cos(N, L)$. Comme on a vu précédemment (voir figure 4.3 page 26), des vecteurs interpolés ne sont pas normés. Cependant, ils ne parlent pas de solution pratique pour pallier à cette difficulté. On peut tout de même supposer que si l'angle entre les deux vecteurs lumière est faible, le problème de la renormalisation est peu visible

transparente. Le facteur de compression est de 1 :8 dans le meilleur des cas. Ceci nous permettrait d'augmenter en 256x256. Les problèmes de cette technique sont :

- L'algorithme employé pour la compression est une technique destructive. Toutefois, si sur des couleurs le résultat est moins choquant, le résultat sur les normales est visible, introduisant une forte détérioration de la qualité due à l'insertion de bruit. Il serait intéressant de disposer de compressions non destructives style RLE ou LZH.
- Les textures étant de faible résolution, chaque détail compte : le gain en résolution améliorée est perdu par les détériations de la compression
- Le coût de la décompression à la volée n'est pour l'instant pas nul. Dans le cas de tracé de plusieurs milliers d'instances, le coût n'est pas négligeable

Après quelques essais, nous avons abandonné cette possibilité, mais cette approche sera à reconsidérer dans les futures cartes.

Le premier problème que l'on voit apparaître avec ce mécanisme d'interpolation est que des contours noirs semi-opaques apparaissent autour des pixels de texture ayant une information de densité non nulle. Ce problème n'est jamais traité dans la littérature non plus. Pour résoudre le problème du bavage occasionné par l'utilisation du filtrage bilinéaire, nous avons utilisé un algorithme de complétage de couleur. Pourquoi a-t-on un problème de bavage ? En fait, comme l'information d'opacité est stockée dans la texture de couleur, elle est interpolée au même titre que les autres composantes de couleur. Or, comme les valeurs de références sont le centre de chaque pixel de texture, si deux pixels adjacents A et B ont l'un 0 comme valeur de densité, l'autre 1, et comme notre test d'élagage des alpha est que l'information de densité (ie alpha) soit non nulle, des points proches de A sont encore visibles et réciproquement, la couleur 0 associée à A intervient dans l'interpolation des couleurs jusque pour les points proches de B. Donc on voit apparaître ce dégradé grisâtre sur les bords de la forme.(voir figure 5.1 page 34) L'algorithme

FIG. 5.1 – A gauche, un seul pixel a une opacité nulle et sa couleur, non définie, est nulle. En foncé, à gauche, la zone rendue lors de l'affichage du texel . Le résultat de l'interpolation est que le mélange des couleurs va se faire tout de même avec les pixels de texture dont la couleur n'est pas définie. La solution consiste à corriger la couleur de cette cellule afin de minimiser l'erreur

utilisé pour passer cette difficulté consiste remplacer la couleur de tout pixel de densité nulle par celle de la moyenne de chacun de ses voisins, i.e. à donner une couleur dans toute la frange théoriquement transparente autour d'une forme opaque (voir algorithmes 5.2 page 35).

5.7 Méthode de génération des tranches de volume

Pour générer des texels, certaines techniques sont spécifiques pour les arbres, d'autres plus génériques appliquées à des volumes arbitraires.

```
Pour tous les points de la texture
       si alpha=0 //densite nulle
       faire
            determiner la couleur de ses 8 voisins
            faire la moyenne de ces couleurs
            remplacer la couleur courante par la moyenne precedente
       finfaire
```

Version de l'algorithme pour les couleurs

```
Pour tous les points de la texture
        si alpha=0 //densite nulle
        faire
                determiner la couleur de ses 8 voisins
                les convertir chacune en vecteur
                faire la somme vectorielle
                renormer le vecteur obtenu
                le convertir en couleur
                remplacer la couleur courante par la couleur calculee
        finfaire
```

Version de l'algorithme pour les normales

FIG. 5.2 – Algorithmes de filtrage des textures

5.7.1 Les méthodes existantes

Dans la méthode de Meyer [MN98], le principe de génération des tranches de textures de couleur à partir d'un objet 3D facettisé reposait sur l'utilisation des plans de clipping successifs. En fait, il regardait l'objet selon une direction principale, puis rendait toute la partie de l'objet présente entre les deux plans de clipping de manière à obtenir l'image d'une tranche, comme dans la figure 5.3 page 36. L'image obtenue était ensuite traitée de façon à former et remplir les contours obtenus. Tous les pixels de l'image qui étaient de la couleur du fond obtenaient une densité nulle (ie pas d'information dans ce pixel), et ceux qui avaient une couleur différente obtenaient une densité maximale. Enfin, elle était sauvée comme une tranche. Puis il déplaçait ses plans de clipping pour générer les autres tranches. Cette méthode est adaptée à de gros objets, mais pas aux petits : une feuille peut se retrouvée coupée et séparée dans deux tranches de textures différentes, ce qui donne vu de près une impression d'écho et de trous.

5.7.2 La méthode employée

Nous avons développé une technique différente, plus adaptée aux arbres. Notre but était d'éviter la coupure des éléments de type feuille. Comme en général, les modèles d'arbres que nous avons utilisés proviennent d'AMAP [dREF+88], un logiciel de génération d'arbres, que ces modèles sont polygonaux, et que ces arbres ont très souvent des feuilles qui ne sont constituées que d'un seul et petit polygone, l'idée est d'adapter la notion de tranche en plans. La base des plans de clipping : plutôt que rendre la portion de l'objet comprise entre les plans de clipping, nous allons rendre les polygones dont le barycentre est compris entre les plans de clipping. Ainsi, les feuilles restent entières et uniques.

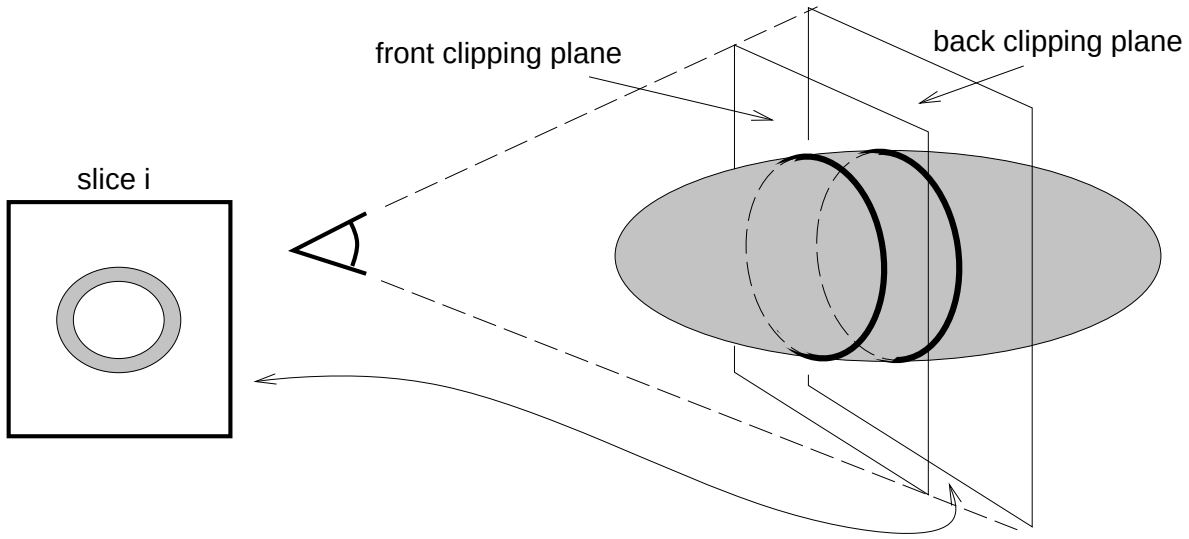


FIG. 5.3 – Construction d'une tranche en utiisant les outils de rendu classique

5.8 Rendu de forêts

Notre modèle de forêts est constitué d'arbres que nous avons disposé sur un modèle de terrain. Ce terrain est une surface constituée de quadrangles qui peut être vue comme une grille de positionnement : dans chaque quadrangle est positionné une instance d'arbre. Nous avons introduit une perturbation aléatoire dans le positionnement d'un arbre au sein d'un quadrangle afin que les texels ne soient pas tous situés au centre du quadrangle. Comme la grille est relativement régulière, nous nous serions retrouvé avec une forêt trop alignée.

Chaque arbre est un texel. Pour chaque instance du volume de référence , nous fournissons un repère lié à l'arbre aux 4 sommets des N tranches. Nous exprimons le vecteur lumière dans chaque repère de chaque arbre. Le texel est une représentation équivalente à la géométrie du point de vue du rendu : il suffit de dupliquer, tourner, scaler les objets, tout marche et ça ne prend pas plus de mémoire (seule limitation : on ne peut pas utiliser des scaling anisotropes avec des facteurs différents dans les trois directions) Dans le cas général, ceci nous permet de pouvoir tourner, translater et mettre à l'échelle n'importe quel texel. Cependant, si nous voulons maximiser la performance, nous avons avantage à utiliser des *display lists*. Ce mécanisme permet d'accélérer le rendu de chaque texel. Si on duplique, translate, scale les instances sans les tourner alors on peut utiliser ce mécanisme. Ceci est contraignant de ne plus pouvoir utiliser la rotation, mais ce mécanisme permet un gain de temps substantiel (de l'ordre de 30%). Le choix dépend du compromis que l'on recherche. On peut aussi se limiter à quelques valeurs de rotations et stocker autant de display lists. Les rotations modifient la direction relative de la lumière, que l'on encode dans la couleur, on ne peut plus utiliser ce degré de liberté, cependant en sonservant la géométrie (i.e. les tranches) dans la mémoire de la carte, évitant ainsi les transferts redondants.

Pour le rendu de toute la forêt, il faut que les texels soient tracés du fond vers l'observateur, comme pour tout rendu de scène en Z-Buffer ayant des objets semi-transparents. En effet, si on ne respecte pas cette contrainte, on obtient des effets indésirables, comme le fait d'avoir des pixels du fond tracés par dessus un arbre situé au premier plan de l'image. De plus, pour une scène très étendue comme le sont les paysages, beaucoup d'éléments sont hors du champ de vision. Pour ceci, à chaque rendu, on construit une liste d'éléments potentiellement traçables, c'est à dire les entités dont le barycentre est dans la pyramide de vue, puis on les trie, et enfin on les affiche.

5.9 Multi-échelle

Afin de diminuer la complexité d'affichage pour des entités de second plan, nous avons utilisé deux types de texels : une version détaillée de texel pour les plans proches de résolution 128x128x64 et une version dégradée de volume de résolution 128x128x8. La version 128x128x64 est utilisée au premier plan alors que la version 128x128x8 est utilisée pour les arbres éloignés. Ceci nous permet d'économiser du temps de rendu pour les arbres de second plan. Nous avons choisi un critère de distance par rapport à l'observateur de telle façon que l'observateur ne perçoivent pas la différence visuelle entre le texel détaillé et non détaillé.

Chapitre 6

Résultats et bilan

6.1 Résultats

Nous avons utilisé comme machine de test un PC BiProcesseur Pentium 3 fréquencés à 850 MHz, 768 Mo de RAM, équipés d'une NVidia Quaddro 2 (version professionnelle de la GeForce).

6.1.1 Qualite visuelle

Affichage d'un texel

Au niveau de la qualité visuelle, nous avons remarqué que notre méthode était suffisamment générale pour rendre autre chose que des arbres. Par exemple, la gargouille présentée en figure figure 6.1 permet mieux de visualiser les effets de l'éclairage. L'éclairage diffus est donc bien calculé et on obtient le résultat escompté. Sa version en polygones comporte 40348 faces.

affichage d'un arbre en texel

Lorsque nous affichons un peuplier généré par AMAP constitué de 98286 faces triangulaires, nous obtenons le résultat suivant (figure 6.2 page 39).

Problèmes d'affichage avec le texel

Nous avons orienté la gargouille dans une direction perpendiculaire à une des directions principales. Si on choisit un angle défavorable, on voit des trous dans le texel (figure 6.3 page 40). Ceci est dû au faible nombre de tranches de texture. Ce problème est inhérent à la représentation. Une autre façon d'éviter cet artefact serait d'utiliser plus de tranches. Pour des texels de type arbre, l'erreur est moins visible car la forme est moins continue. Le résultat semble moins perturbé.

6.1.2 Performances

Nous avons cherché à obtenir un taux de rafraîchissement de l'écran de l'ordre de 20 à 25 frames par secondes. Nous avons réussi à afficher 100 arbres à ce taux de rafraîchissement (voir figure 6.4 page 40). Nous avons effectué les tests de performances sur une forêt de peupliers en faisant varier les paramètres suivants :

- le nombre de texels affichés à l'écran
- le type du shader

FIG. 6.1 – La gargouille éclairée par une lampe dont la position varie, la résolution du texel est 128x128x64

FIG. 6.2 – L'affichage d'un arbre sous différentes positions de la lumière

FIG. 6.3 – Le résultat visuel lorsqu'on est trop près du texel et que l'on regarde dans une direction trop oblique

FIG. 6.4 – Les texels permettent de faire un rendu de forêt dans une résolution 800x600 en temps réel (20 fps) avec 100 arbres à l'écran avec un niveau de détail intermédiaire

– le niveau de détail : Fort signifie que la résolution est haute (ie tous les texels de la scène sont dans la résolution 128x128x64), faible signifie que la résolution est basse (tous les texels de la scène sont dans la résolution 128x128x8), intermédiaire signifie que les texels proches sont en haute résolution et les lointains sont en basse résolution

La version polygonale pouvait s'afficher à 27 images par seconde pour une seule instance. Pour 200 instances, le taux de rafraîchissement est de 0.135, et ainsi de suite...

Nombre d'arbres rendus par la carte	Taux de rafraîchissement	Shader	Niveau de détail
200	16.84	Diffus	Faible
200	16.9	Diffus+fog	Faible
200	8.66	Diffus	Intermédiaire
200	7.51	Diffus+fog	Intermédiaire
200	2.24	Diffus	Fort
200	2.28	Diffus+fog	Fort
500	8.22	Diffus	Faible
500	8.16	Diffus+fog	Faible
500	4.24	Diffus	Intermédiaire
500	3.98	Diffus+fog	Intermédiaire
500	1.04	Diffus	Fort
500	1.03	Diffus	Fort
1000	5.31	Diffus	Faible
1000	4.46	Diffus+fog	Faible
1000	2.51	Diffus	Intermédiaire
1000	2.48	Diffus+fog	Intermédiaire
1000	0.64	Diffus	Fort
1000	0.63	Diffus+fog	Fort

Ces résultats mettent en évidence plusieurs choses. Alors que le taux de remplissage est divisé par deux lorsque l'on rajoute le fog (d'après le constructeur, le fait d'utiliser un shader utilisant deux étages de register combiners plutôt qu'un seul requiert deux fois plus de taux de remplissage), on ne constate pas de baisse de taux de rafraîchissement. Ce qui prouve donc que le taux de remplissage n'est pas le facteur limitant, lors du rendu de forêts. Si lorsque l'on active le fog, le taux de rafraîchissement est en général équivalent, il faut bien préciser que ce taux de rafraîchissement est échantillonné toutes les 10 images ce qui donne un calcul variable entre deux mesures. Les facteurs potentiels limitants sont donc :

– la quantité de géométrie à transmettre à la carte graphique pour chaque image
– le temps passé à changer de texture entre chaque tranche de texel.

Lorsque l'on utilise deux niveaux de détails simultanément, on économise un temps de calcul tout en conservant une qualité acceptable visuellement parlant. Pour un même nombre d'arbres, on multiplie le taux de rafraîchissement par 4 en utilisant ce genre de technique par rapport au modèle fort de détail, et on multiplie par 2 par rapport au niveau de détail intermédiaire en utilisant le mode peu détaillé. Le problème devient alors la qualité visuelle qui est trop dégradée. L'utilisation de niveaux de détails consiste à minimiser l'influence des facteurs potentiels limitant présentés précédemment.

6.2 Améliorations possibles

6.2.1 Les déformations

Nous avons vu que nous ne pouvons pas simplement utiliser l'étirement des tranches dans une direction car alors, les normales ne seraient pas correctes. Pour résoudre ce problème, nous avons pensé utiliser les vertex shaders, que nous aurions utilisé pour déformer dynamiquement les repères liés à chaque sommet

de chaque tranche. Nous pourrions ensuite les déformer facilement tout en gardant un calcul d'éclairement correct.

6.2.2 Les ombres portées des arbres sur le sol

Nous avons pensé à un système de textures d'ombres sur le sol, mises à jour dès que la lumière bouge. A chaque mouvement de la lampe, il suffirait de faire un rendu du point de vue de la lampe puis de plaquer le résultat sur le sol.

6.2.3 Les ombres des arbres sur les autres

Meyer,Neyret et Poulin [MNP01] ont proposé un système d'ombres à partir de cube maps. Le problème de cette approche est qu'elle requiert une cube map par instance.

6.3 Bilan

Nous avons réussi à utiliser le hardware graphique de nouvelle génération (disponible dans le grand public) pour étendre le rendu de forêts en trouvant un équilibre entre qualité et performance. Il est maintenant possible d'utiliser des sources de lumières qui bougent lors du rendu de forêts tout en conservant le temps-réel ce qui n'était pas possible de faire dans les modèles antérieures. Notre technique consiste à calculer l'éclairage en chaque pixel de chaque tranche de texture du texel, ce qui nous permet de simuler l'éclairage de l'objet. Nous avons tenté de minimiser la complexité en performance en utilisant au mieux les pixels shaders. Nous avons essayé d'explorer la nouvelle orientation des cartes graphiques ainsi que les modèles existants afin de proposer une amélioration au rendu de forêts.

Comme le hardware graphique tend à s'uniformiser (API indépendantes de la plate-forme, extensions communes), notre modèle est général, et appliquable sur les dernières cartes produites.

De par nos statistiques, nous avons montré que ce modèle est applicable à des forêts de tailles moyennes, ce qui peut convenir à des scènes d'exterieur dans un jeu vidéo par exemple. Dans nos résultats sur le plan qualitatif, nous avons montré que l'utilisation des texels n'était pas forcément limitée à des modèles d'arbres, mais aussi à des modèles géométriques quelconques tels la gargouille. L'approche texel temps-réel est applicable à des objets censés être à une distance intermédiaire. Leurs imperfections sont faiblement perceptibles et ils ne requièrent que très peu de polygones à remplir. Nos texels sont donc une représentation à part entière, qui pourrait être promise à un grand avenir.

6.4 limitations du modèle

La technique présente des défauts visuels, liés à l'utilisation de la technique de tranches de textures. Comme beaucoup de techniques d'image based-rendering, elle présente des problèmes de continuité de surface. Ceci n'est valable que dans le cas du rendu temps-réel de texels. Les modèles en ray-tracing passent outre de ces problèmes par le biais de l'interpolation trilinéaire. Une solution existe pour réaliser l'interpolation trilinéaire mais elle requiert, pour le hardware graphique actuel, d'utiliser les deux unités de textures simultanément pour une seule information (couleur ou normale) ce qui reviendrait à faire deux passes de rendu, donc diviser le taux de rafraichissement par 2. La contrainte du temps-réel est donc difficile à tenir dans ces conditions, d'autant que le fait d'utiliser les niveaux de détails en fonction de la distance à l'observateur permet de limiter ces détails visuel, car moins perceptibles (puisque plus petites à l'écran).

Le modèle de texel étant gourmand en mémoire, il faudrait pour être utilisable pleinement que les bus mémoire voient leurs débits augmenter à l'avenir, afin de pouvoir utiliser la mémoire centrale conjointement au processeur sans que ce soit perceptible à l'utilisateur. Les expérimentations que nous avons

effectuées avec les systèmes de compression de texture ne permettent pas d'obtenir des résultats d'une qualité suffisante. Dans les approches de type ray-tracing, la place était économisée en utilisant des structures de données adaptées (octrees) ce qui réduisait la taille en mémoire de l'ordre de 90%.

Le système de découpage du volume en tranche requiert que le modèle soit très détaillé car chaque polygone ne se projetant qu'une seule fois, les polygones en zone médiane peuvent former des trous dans le modèle du texel. Ceci est visible lorsque les polygones du tronc sont très allongés et que l'observateur regarde le texel sous un angle intermédiaire entre deux directions de texels. Le tronc se rapporte à un cercle flottant dans l'espace.

Toutefois la tendance est à l'augmentation de la mémoire sur carte, de la puissance d'expression des shaders par point, et à l'accroissement de la taille et de la complexité des scènes. Donc notre modèle va se voir sûrement étendu par les capacités du hardware à venir.

Bibliographie

[Bio] Bionatics. Amap.
`http ://www.jmg-graphics.com/amap.htm`.

[Bli78] J. F. Blinn. Simulation of wrinkled surfaces. volume 12, pages 286–292, August 1978.

[Cha97] Christophe Chaudy. *Modélisation et rendu d'images réalistes de paysages naturels.* PhD thesis, Université Joseph Fourier - Grenoble 1, 1997.

[Cir] Cirad. Produits du cirad.
`http ://www.cirad.fr/produits/amap/amap.html`
`http ://www.cirad.fr/produits/amap/pepinieres/pepinieres.shtml`.

[CMDH97] Norishige Chiba, Kazunobu Muraoka, Akio Doi, and Junya Hosokawa. Rendering of forest scenery using 3D textures. *The Journal of Visualization and Computer Animation*, 8(4) :191–199, 1997. ISSN 1049-9807.

[dREF+88] Phillippe de Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22(4), pages 151–158, August 1988.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics : Principles and Practices (2nd Edition).* Addison Wesley, 1990.

[HS99] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In Alyn Rockwood, editor, *Siggraph 1999, Annual Conference Proceedings*, Annual Conference Series, pages 171–178, Los Angeles, 1999. ACM Siggraph, Addison Wesley Longman.

[Jak00] Aleks Jakulin. Interactive vegetation rendering with slicing and blending. In A. de Sousa and J.C. Torres, editors, *Proc. Eurographics 2000 (Short Presentations).* Eurographics, August 2000.

[Kil00] Mark J. Kilgard. A practical and robust bump-mapping technique for today's gpus. Technical report, GDC2000, 2000.

[KK89] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 271–280, July 1989.

[KS01] Jan Kautz and Hans-Peter Seidel. Hardware accelerated displacement mapping for image based rendering. Graphics Interface, 2001.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 451–458. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[MN98] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pages 157–168, New York City, NY, Jul 1998. Eurographics, Springer Wein. http ://www-imagis.imag.fr/Publications/1998/MN98b.

[MNP01] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. Interactive rendering of trees with shading and shadows. In *Eurographics Workshop on Rendering*, Jul 2001.

[MW] Tom Davis Mason Woo, Jackie Neider. *OpenGL programming Guide - The Official Guide to learning Opengl (Red Book)*. Addison Westley developers press.

[Ney95] Fabrice Neyret. A general and multiscale method for volumetric textures. In *Graphics Interface'95 Proceedings*, pages 83–91, May 1995.

[Ney96a] Fabrice Neyret. Synthesizing verdant landscapes using volumetric textures. In *Eurographics Workshop on Rendering'96*, pages 215–224, June 1996.

[Ney96b] Fabrice Neyret. *Textures Volumiques pour la Synthèse d'images*. PhD thesis, Université Paris-XI - INRIA, 1996. http ://www-imagis.imag.fr/Membres/Fabrice.Neyret/publis/thesefabrice-fra.html.

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

[Nom95] Tsukasa Noma. Bridging between surface rendering and volume rendering for multi-resolution display. In *6th Eurographics Workshop on Rendering*, pages 31–40, June 1995.

[OGL] Opengl advanced coders forum.
http ://www.opengl.org/.

[RB85] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 313–322, July 1985.

[RSEB+00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 109–118, August 2000.

[SGIa] SGI. Opengl extension registry.
http ://oss.sgi.com/projects/ogl-sample/registry/index.html.

[SGIb] SGI. Opengl shader.
http ://www.sgi.com/software/shader/.

[Shi92] Mikio Shinya. Hierarchical 3D texture. In *Graphics Interface '92 Workshop on Local Illumination*, pages 61–67, May 1992.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 169–178. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.