N° d'ordre : 02/2010-M/EL

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE
FACULTE D'ELECTRONIQUE ET D'INFORMATIQUE

Mémoire

Présenté pour l'obtention du diplôme de Magister

En Electronique

Spécialité : **Traitement du Signal et d'Images**

Par : Benmeziane Samir

THEME

RENDU AVANCE PAR LA METHODE RAY-CASTING BASE GPU: APPLICATION A L'IMAGERIE MEDICALE

Soutenu publiquement, le 23 Novembre 2010, devant le jury composé de :

M.	M. ATTARI	Professeur	à l'USTHB	Président
Mme.	N. ACHOUR	Maitre de conférence A	à l'USTHB	Examinatrice
M.	K. BOUATOUCH	Professeur	à l'IRISA/Rennes	Examinateur
Mme.	F. BOUMGHAR	Professeur	à l'USTHB	Directrice de mémoire
Mme.	H. BENTOUMI	Maitre Assistante A	à l'USTHB	Examinatrice

RENDU AVANCE PAR LA METHODE DE RAY-CASTING BASE GPU: APPLICATION A L'IMAGERIE MEDICALE

A mes parents, ma femme,

ma sœur, mes frères, ma belle sœur,

mes nièces et mon neveu

Je dédie ce mémoire

REMERCIEMENTS

Je remercie vivement en premier lieu Madame H. Bentoumi, Monsieur K. Bouatouch et Madame F.O. Boumghar pour m'avoir proposé un sujet aussi passionnant, pour leur conseil, leur aide, leur disponibilité, ainsi que pour leurs encouragements tout au long de ce travail de recherche.

J'exprime ma gratitude aux membres de l'équipe ParIMèd du laboratoire LRPE et de l'équipe BUNRAKU du laboratoire de l'IRISA de Rennes pour leur aide. En particulier, Kevin Boulanger et Mathias Schoot, pour leurs conseils, leur disponibilité et avec qui j'ai pu faire un grand pas dans la programmation.

Un grand merci à mes parents pour leur soutien moral et financier et surtout à ma femme qui m'a soutenu tout au long de ce travail.

J'exprime ma sincère reconnaissance aux membres du jury.

Sans oublier, ceux qui ont participé de loin ou de près, à ce travail.

TABLE DES MATIERES

LISTE DES FIGURES

Figures Chapitre I

Figures Chapitre II

Figures Chapitre III

Figures Chapitre IV

INTRODUCTION

A l'heure actuelle, la visualisation des données volumiques est l'un des axes majeurs de la recherche. Ce domaine très vaste permet des applications variées et offre des solutions de représentations visuelles de données parfois complexes, de grandes tailles et provenant de diverses sources, telles que les données médicales prélevées par TDM (TomoDensitoMétrie), IRM (Imagerie par Résonance Magnétique), les données sismiques de l'exploration dans le domaine du pétrole ou du gaz, les données du modèle de calcul d'éléments finis, etc...

Parmi toutes ces données à visualiser, on peut retenir les données médicales qui sont issues de l'acquisition de différentes modalités qui servent à reproduire l'ensemble des organes du corps et des matériaux scannés. Dans ce domaine, la qualité et le temps du rendu graphique sont primordiaux afin que le diagnostic du médecin soit immédiat et précis.

Dans cette optique, de très nombreuses contributions en rendu volumique ont été apportées afin de satisfaire les besoins d'exploration et d'analyse des données pour obtenir des qualités de rendu que l'on peut qualifier de réaliste. Cependant, bon nombre de ces techniques permettent un rendu de qualité, mais au prix de lourds calculs, impossibles à réaliser en temps réel ou même en interactif. Ces approches étant au départ purement logicielles et réservées aux matériels spécialisés de haute gamme.

L'avancée considérable de l'architecture des processeurs des cartes graphiques, accessible aux programmeurs et aux développeurs permet une gestion plus facile des masses de données numériques et il est possible d'envisager une visualisation de ces données sur un PC standard, donnant ainsi lieu à une nouvelle famille d'algorithmes tirant profit des possibilités offertes par ce matériel.

Les cartes graphiques effectuent des traitements sur les données qui constituent la scène. L'ensemble de ces traitements définissent le pipeline graphique lequel est composé de trois phases de traitement : les opérations sur les sommets, les opérations sur les pixels et au final, la composition de la couleur. Au niveau des deux premières phases de traitement se trouvent deux processeurs programmables, le *vertex processeur* et le *fragment processeur*, conçus pour exécuter des microprogrammes, respectivement le *vertex shader* et le *fragment shader*. Ces shaders sont écrits à l'aide d'un langage de programmation dédié tel que : *OpenGL Shading Language* (GLSL).

C'est dans ce contexte que s'inscrit notre travail. La technique de rendu volumique à laquelle on s'est intéressée est le ray-casting, appartenant à la classe ***image-order,*** ou ***Backward Mapping***. Le ray-casting permet d'obtenir des images de grande qualité, mais il est très exigeant en mémoire et en temps de calcul. Il existe deux approches de ray-casting se basant sur l'accélération par carte graphique, la première est le rendu volumique basé texture, ce que propose CULLIP et NEUMANN et al. ; cet algorithme utilise directement les capacités de la texture mapping. La seconde approche est le rendu volumique basé sur GPU utilisant les unités du processeur graphique et proposé à l'origine par KRÜGER et WESTERMANN ; dans cette même optique STEGMAIER et al. représentent un ray-casting, dans lequel l'ensemble des rayons est décrit analytiquement par deux textures 3D, l'une pour les positions de départ et l'autre pour les positions finales. Cette approche est intéressante, puisque l'ensemble des données est stocké comme texture 3D pour tirer profit de l'interpolation tri-linéaire intégrée dans le matériel graphique, à la différence des méthodes classiques qui

utilisent des tranches de texture (surfaces de substitution ou surfaces de proxy) pour lesquelles le matériel graphique effectue le rééchantillonnage.

Dans notre travail nous nous plaçons dans cette seconde approche et proposons de tirer profit de la capacité de calcul des *shaders* pour réaliser les différents calculs concernant la géométrie analytique ce qui nous évite de faire deux passes de calcul. L'algorithme de ray-casting que nous avons mis en œuvre se scinde en deux phases : Une phase dans laquelle un certain nombre de calculs s'exécute au niveau de l'application, c'est-à-dire le CPU (lecture des données volumiques à partir d'un fichier, définition des positions de la lumière, de l'observateur et du volume de données et calcul de la valeur du gradient local) et une autre phase dans laquelle l'ensemble des calculs s'effectue au niveau des shaders, c'est-à-dire le GPU (calculs de la géométrie analytique, définition des directions de chaque vecteurs de visualisation et de lumière et calcul d'intersection de ces vecteurs avec le volume de données et l'application du modèle d'illumination de Blinn-Phong)

Pour ce qui est de l'illumination, procédé incontournable dans la visualisation graphique, on trouve deux types d'illumination : l'une dite locale et l'autre dite globale. L'illumination locale est basée sur la contribution directe de la source de lumière dans la couleur des objets. L'illumination globale simule la contribution de toutes les sources de lumière dans une scène, sur la couleur des objets. L'exemple typique de l'illumination locale est le modèle d'illumination de Phong. Dans notre rendu volumique, nous adoptons le modèle d'illumination de Blinn-Phong qui est une autre façon d'interpréter l'équation de Phong, dans laquelle on suppose que les matériaux possèdent des propriétés spatiales pour réfléchir la lumière, qui sont la réflexion spéculaire, la réflexion diffuse, la réflexion ambiante et la constante de brillance (*shininess*). Chacune de ces composantes définit la direction spatiale vers laquelle la lumière est réfléchie.

Notre travail s'inscrit dans l'axe de recherche en imagerie médicale développé par l'équipe ParIMéd (Parallélisme et Imagerie Médicale) du laboratoire LPRE (Laboratoire du Parallélisme, Robotique), de la Faculté d'Electronique et d'informatique (FEI) à l'USTHB avec la collaboration de l'équipe BUNRAKU du laboratoire de recherche de l'IRISA de Rennes. L'étude est orientée essentiellement vers la visualisation tridimensionnelle d'images médicales exploitant les performances des cartes graphique GPU. Le choix de développer l'algorithme de ray-casting sur GPU a été adopté car cet axe de recherche est très prometteur et plusieurs modèles de cartes graphiques intègrent cet algorithme.

Ce document se présente sous forme de quatre chapitres. Le premier chapitre est un bréve état de l'art consacré à la description des types de données utilisés en imagerie médicale. La représentation de ces données numériquement et les différents systèmes de coordonnées utilisés pour pouvoir visualiser correctement ces données y sont définis ainsi que les notions de base du rendu volumique direct, l'équation de rendu et, la discrétisation de cette équation d'un point de vue *image-order* spécifique au ray-casting. L'algorithme du ray-casting consiste à émettre des rayons à partir de la position de vision vers chaque pixel du plan image ou écran, en parcourant le volume, la couleur finale du pixel à afficher correspondant à la contribution de l'ensemble des couleurs des voxels le long de ce rayon,

auquel nous ajoutons la contribution de la lumière sur chacun de ces voxels. Ce chapitre comprend aussi la description des différentes étapes du pipeline de rendu et illustre de façon non-exhaustive l'algorithme du ray-casting avec une description des méthodes de ray casting basées sur le GPU ainsi que l'introduction de notre approche.

Le second chapitre relate les différents étages du pipeline graphique et situe les unités programmables, le vertex processeur et le fragment processeur. Il donne une description assez bref du pipeline OpenGL, et met en avant les différents flux de données qui transitent entre l'application OpenGL et les différentes unités du pipeline, le vertex processeur et le fragment processeur.

Le troisième chapitre décrit de façon exhaustive les différentes étapes de notre implémentation basée GPU. On détermine les différents calculs effectués, sur le CPU et les calculs effectués au niveau du GPU. Nous détaillons le calcul du gradient, la manière avec laquelle la fonction de transfert est appliquée et l'utilisation efficace de la mémoire de texture pour la sauvegarde des calculs intermédiaires. Le dernier chapitre illustre et interprète les différents résultats obtenus avec différentes configurations. Il illustre les résultats obtenus dans le cas du calcul d'ombrage et finalement, des perspectives à notre application sont proposées.

CHAPITR I

LE RENDU VOLUMIQUE :

ETAT DE L'ART

Le rendu volumique est devenu une technique de visualisation incontournable pour une large variété d'applications ; la médecine, la biotechnologie, l'astrophysique, et d'autres domaines. Par exemple la visualisation des données médicales prélevées par Tomographie (CT) et Imagerie à Résonance Magnétique (MRI), des données sismiques de l'exploration de pétrole et de gaz, et des données de calcul de modèle d'élément fini. Ces données numériques de plusieurs Méga-octets sont stockées dans des tableaux tridimensionnels représentant ainsi une grille d'espace à trois dimensions. Ces données ont longtemps été une contrainte pour les techniques de rendu. De nos jours, la grande avancée dans la conception du matériel graphique avec GPU : Graphic Process Unit, a depuis peu, permis une gestion plus facile du volume de données.

Le rendu volumique a pour effet de visualiser les données de la grille de l'espace 3D sur un plan 2D avec la projection du volume 3D sur l'écran de visualisation. Les techniques de rendu ont pour but d'obtenir des images plus réalistes de la représentation numérique des données en se basant sur les phénomènes d'optique et les modèle d'illumination qui les représente.

Les techniques de rendu volumique sont classées en deux grandes catégories, le rendu volumique direct et le rendu volumique indirect. Ce travail s'inscrit dans la première catégorie.

I.1. Types et nature des Données Volumiques

Les données d'un rendu volumique peuvent provenir de plusieurs domaines d'application. Le domaine qui nous intéresse est la visualisation scientifique, plus spécialement, la formation d'image médicale 3D. En imagerie médicale, ces données sont généralement acquises à l'aide d'un dispositif de balayage, ou modalité.

- L'Estimation Tomographique (*Computed Tomography*) est une méthode fréquemment utilisée pour obtenir les données médicales en 3D. Le procédé physique du scanner est basé sur le calcul de l'atténuation des rayons X émis. [14].
- L'Imagerie à Résonance Magnétique (*Magnetic Resonnance Imaging*) peut également produire des ensembles des données d'image 3D. Les images produites sont un peu bruitées, cette technique a la capacité de séparer toute structure de tissu mou.
- D'autres modalités sont également employées, par exemple : l'ultrason ou la Tomodensitométrie (TDM) pouvant être appliqué pour la reconstruction 3D.

Toutes ces modalités ont en commun le fait que l'ensemble des données discrétisées sont reconstruites à partir de la rétroaction, ou le feedback, des rayonnements détectés. Ceci signifie qu'un certain genre de processus de transformation a lieu et que nous n'employons pas des données d'origines pour le rendu de volume [14].

Les valeurs des données (supposées discrétisées) sont généralement converties en entiers ou en flottants codés sur 1, 2, 4, voire 8 octets.

Afin de bénéficier des performances des cartes graphiques, il est nécessaire de stocker les données sous la forme de texture 3D, un tableau tridimensionnel de valeurs discrètes, qui constitue un maillage structuré de l'espace. On affecte ainsi à tout voxel, l'unité de volume qui correspond au plus petit élément d'une texture 3D, une valeur calculée en fonction des données initiales, ce qui peut engendrer dans certains cas un rééchantillonnage.

Comme la texture 3D (*Figure I.1*) réside dans une mémoire dédiée à la carte graphique, la rapidité de traitement et du rendu en est grandement accéléré. Des fonctions internes, notamment d'interpolation tri-linéaire dans l'espace des voxels, sont également disponibles.

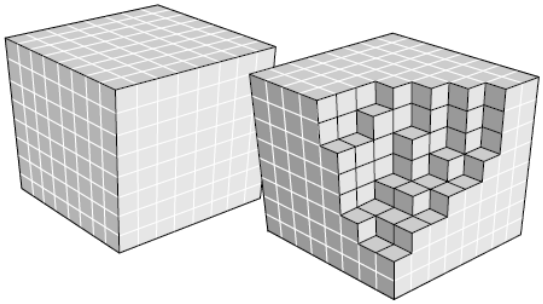


Figure I.1 : *Représentation des données volumique discrétisées comme texture 3D.*

I.2. Principe du Rendu Volumique

Le rendu volumique direct [12] est introduit vers la fin des années 1980 notamment par les travaux de Marc Levoy [21] et ceux des Studios *Pixar*. Contrairement au rendu de surface où seulement des iso-surfaces représentant des régions d'intérêt extraites du volume sont présentées, le rendu de volume permet de visualiser une projection bidimensionnelle de celui-ci. Ces algorithmes consistent généralement en l'association d'une couleur et d'une opacité à chaque voxel et au calcul de la contribution de ces voxels sur un plan de projection bidimensionnel (plan image). A l'origine, ces algorithmes ont principalement été développés pour visualiser des données médicales. Ils peuvent également être utilisés pour produire des représentations graphiques de tout autre ensemble de données tridimensionnelles ré-échantillonnées sur une grille de points régulière.

Les algorithmes de rendu volumique sont regroupés en deux grandes classes, le *Backward Mapping* [10] [30] et le *Forward Mapping* [39] [17], respectivement connus sous les noms d'*Image-Order* et *Object-Order*. Les algorithmes utilisant le *Backward Mapping* projettent le plan image sur l'ensemble de données tandis que les algorithmes utilisant le *Forward Mapping* projettent le volume sur le plan image.

I.3. Equation du rendu volumique : du continu au discret

Les méthodes de rendu volumique sont fondées sur les propriétés optiques du milieu : l'émission, l'absorption et la diffusion, décrivant l'interaction de la lumière avec les différents milieux rencontrées lors de son parcours. L'évaluation de l'équation du rendu volumique le long d'un rayon de lumière se fait par approximation en considérant un faible albédo [4] [16],

c'est-à-dire, qu'un rayon de lumière subit une réflexion parfaite (se concentre dans une seule direction). Les effets de l'interaction de la lumière sont intégrés le long du rayon de visualisation dans l'espace-rayon.

Soit $\vec{r}(t)$ un rayon lancé dans le volume la zone de l'espace délimitant le champ scalaire, ce dernier étant paramétré par la distance à l'observateur t (*Figure I.2*). Le scalaire correspondant à la position courante dans l'espace $\vec{r}(t)$ est donné par $s(\vec{r}(t))$. Etant donné que l'on se place dans un modèle optique d'absorption, l'équation propre au Rendu Volumique va intégrer l'opacité $\tau\left(s(\vec{r}(t))\right)$ainsi que la couleur $c\left(s(\vec{r}(t))\right)$ le long du rayon, nous donnant la couleur finale de notre pixel *C* selon l'équation *(I.1)* :

$$C = \int_O^D c\left(s(\vec{r}(t))\right) e^{-\int_0^{t'} \tau\left(s(\vec{r}(t'))\right) dt'} \, dt. \qquad \text{(I.1)}$$

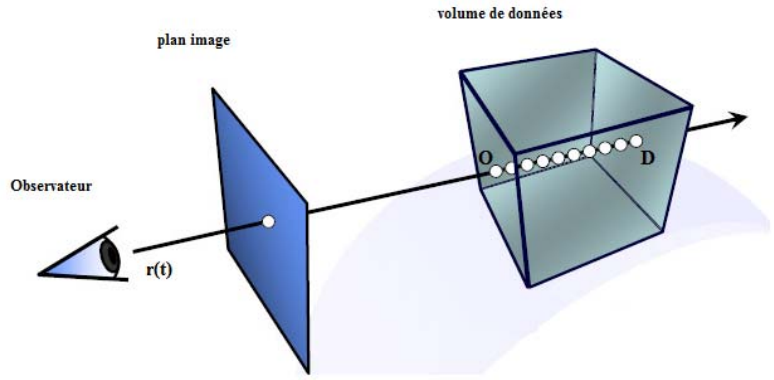


Figure I.2 : *Intégration du Volume le long d'un rayon.*

Cette forme courante correspond à la théorie physique de transport de lumière en milieu absorbant, mais non diffusant.

Pour une position de l'observateur dans l'espace et un vecteur de vue, la résolution de l'intégrale pour chacun des rayons traversant le volume du point d'entrée O au point de sortie D nous permet d'obtenir l'image finale.

En subdivisant le rayon allant de O à D (*figure I.2)* en une succession finie d'échantillons rapprochés où nous considérons que nous avons (n+1) échantillons et en substituant l'intégrale continue par une somme de Riemann, nous obtenons alors pour l'échantillon *d* du *i*ème rayon lancé :

$$C \approx \sum_{i=0}^{n} C_i \, \alpha_i \left(e^{-\sum_{j=0}^{i-1} \alpha(j)} \right). \qquad \text{(I.2)}$$

Avec :

- $C_i = c\left(s(\vec{r}(d_i))\right)$ la couleur à l'émission du rayon incident.
- $\alpha_i = \tau\left(s(\vec{r}(d_i))\right)$ la transparence, soit le coefficient d'absorption.

De par la propriété des exponentielles ($e^{x+y} = e^x e^y$), nous obtenons :

$$C \approx \sum_{i=0}^{n} C_i \, \alpha_i \prod_{j=0}^{i-1} e^{-\alpha(j)}. \quad \text{(I.3)}$$

En considérant le développement limité de la fonction exponentielle e^x au voisinage de 0, soit $(1 + x + \epsilon(x))$ et en négligeant $\epsilon(x)$, nous obtenons au final :

$$C \approx \sum_{i=0}^{n} C_i \, \alpha_i \prod_{j=0}^{i-1} (1 - \alpha_j). \quad \text{(I.4)}$$

Où C_i est la couleur de l'i^{ième} échantillon du rayon (à trois composantes RVB), α_i étant l'opacité. Les échantillons sont strictement ordonnés selon la profondeur afin d'obtenir un rendu volumique correct en respectant la transparence.

I.4. Pipeline du Rendu Volumique

Le rendu volumique peut être vu comme un ensemble d'étapes de traitement en pipeline, qui s'adapte à l'architecture hardware. Dans cette section, nous examinons plus en détail, les étapes du pipeline sur lesquels se basent les algorithmes de rendu volumique.

I.4.1. Parcours des Données

Tout algorithme de rendu de volume affecte des adresses à chaque position de ré-échantillonnage dans tout le volume. Le ré-échantillonnage des positions dans l'espace objet sont les plus susceptibles de ne pas correspondre aux positions des voxels, ce qui exige une interpolation des voxels entourant l'échantillon, pour estimer la valeur à la position non-intégrée [17].

I.4.2. Interpolation

Les positions d'échantillonnage sont habituellement différentes des points de la grille. Donc, l'espace 3D continu est reconstruit à partir d'une grille discrète afin d'obtenir les valeurs des données aux positions de rééchantillonnage [14]. L'interpolation tri-linéaire est l'interpolation la plus utilisée pour une grille uniforme et pour les méthodes de rendu notamment le ray-casting [2] [25], *(Figure I.3)*.

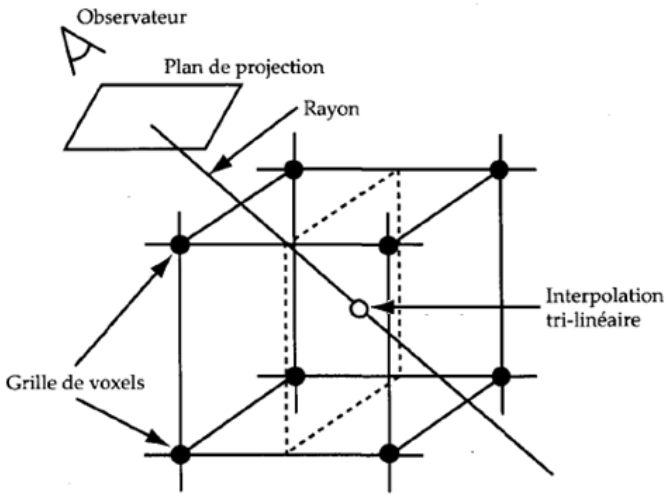


Figure I.3 : *Rééchantillonnage par interpolation tri-linéaire.*

I.4.3. Calcul du Gradient

Le calcul du gradient permet l'approximation des normales des surfaces, utilisé dans le calcul du modèle d'illumination où bien le shading et dans le processus de classification. Pour une fonction continue donnée f(x), le gradient ∇f est défini comme la dérivée partielle selon les trois directions *x*, *y* et *z* [14].

$$\nabla f(\mathrm{x}) = \left(\frac{\partial f(\mathrm{x})}{\partial x}, \frac{\partial f(\mathrm{x})}{\partial y}, \frac{\partial f(\mathrm{x})}{\partial z}\right)^T \tag{I.8}$$

Le vecteur normal utilisé pour le modèle d'illumination local doit être unitaire, ce qui est garanti en normalisant le gradient :

$$\mathrm{n}(\mathrm{x}) = \frac{\nabla f(\mathrm{x})}{\|\nabla f(\mathrm{x})\|}, \qquad \text{if} \|\nabla f(\mathrm{x})\| \neq 0 \tag{I.9}$$

Le gradient des données volumiques discrétisées, est approximé par la méthode de la différence centrée, qui est calculé par les différences locales sur le voisinage de 6 voxels [14] [17].

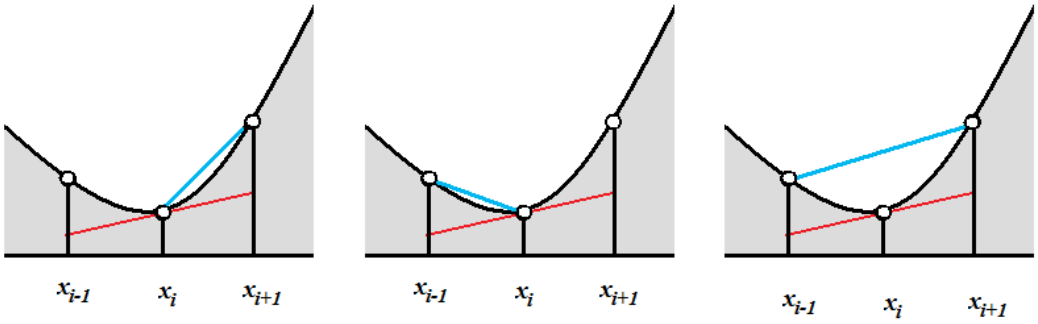


Figure I.4 : *Calcul de l'intégrale par la différence centrée.*

Comme on peut le voir sur la figure *(I.4)*. La différence centrée approxime le mieux la pente de la tangente au point x_i avec la pente de la droite en considérant les points x_{i-1} et x_{i+1}.

L'estimation du gradient d'une fonction f par la différence centrée est:

$$\nabla f(x,y,z) \approx \frac{1}{2h}\begin{pmatrix} f(x+h,y,z) - f(x-h,y,z) \\ f(x,y+h,z) - f(x,y-h,z) \\ f(x,y,z+h) - f(x,y,z-h) \end{pmatrix} \tag{I.10}$$

En posant $\nabla f(\mathrm{x}) = \nabla f(x,y,z)$.

Comme on peut le voir, la différence se fait pour un voisinage de six échantillons avec une distance *h* autour de la position où le gradient est estimé. En pratique *h* correspond à la taille de la grille [22] [24].

I.4.4. Classification

La classification est le processus de mise en relation, ou de *mapping,* des propriétés physiques du volume et des propriétés optiques, par l'intégrale du rendu volumique, tel que l'émission (la couleur, RGB) et l'absorption (l'opacité, α). Ceci s'effectue en appliquant une fonction de transfert qui prend le domaine des données d'entrée et le transforme en une gamme de rouge, vert, bleu et alpha. Cependant, les fonctions de transfert sont employées pour assigner des couleurs spécifiques aux intervalles des valeurs de la source de donnée correspondant à des zones d'intérêt. L'os par exemple, est bien contrasté à l'aide de la Tomographie X, alors que les tissus mous le sont par l'IRM.

La classification revient à définir une fonction allant de $\mathbb{R}$ dans $\mathbb{R}^4$ appelé *fonction de transfert*, cette dernière est généralement transcrite sous forme d'une texture unidimensionnelle. Ces fonctions assignent des couleurs spécifiques aux intervalles des valeurs de la source de données, correspondent à des zones d'intérêt, par exemple l'os et les tissus mous des données de CT, dont la qualité visuelle dans l'image finale est employée pour identifier ces régions. Pour une valeur scalaire s le long d'un rayon $g(\mathrm{x}) = T_0(s(\mathrm{x}))$, on lui introduit souvent la grandeur $|\nabla_s|$ du gradient comme paramètre additionnel de classification. Cette approche est généralement utilisée dans la visualisation des os et des tissus pour des ensembles de données médicales ou dans le cas de la visualisation des iso-surfaces d'images de densité.

Le terme d'émission $c(\mathrm{x})$ peut être utilisé comme fonction de transfert pour des valeurs scalaires s; $c(\mathrm{x}) = T_c(s(\mathrm{x}))$, qui ne dépend pas de la direction.

On distingue deux types de classifications, une pré-classification et une post-classification, selon la manière avec laquelle les valeurs des voxels du volume sont classifiées soit avant interpolation ou après interpolation [14] [18].

a) pré-classification : La fonction de transfert est appliquée pour chaque échantillon avant l'interpolation tri-linéaire. On dit alors que l'on travaille dans l'espace des couleurs, celle-ci étant interpolées. Les hautes fréquences de la fonction de transfert, c'est-à-dire les zones à forte variation d'intensité de couleur et de transparence pour deux scalaires rapprochés, peuvent ne pas être reproduites, ce qui se traduit par l'introduction de défauts visuels [34] [41].

b) post-classification : La fonction de transfert est appliquée après interpolation, donc au niveau du champ de données. La visualisation en découlant, permet de rendre tous les détails de la fonction de transfert. Cependant cette approche est moins performante que la précédente de par l'application de la fonction de transfert à toutes les valeurs issues de l'interpolation linéaire inter-échantillons [2] [13].

I.4.5. Illumination (*Shading*)

L'illumination est susceptible de rajouter du réalisme et permet une meilleure interprétation des données volumiques. Différents modèles d'illumination permettent d'estimer les différentes interactions de la lumière, on distingue deux classes, des modèles dit locaux et d'autres dits globaux. Le modèle d'illumination local, auquel on s'intéresse, donne l'approximation de l'intensité de la lumière réfléchie en un point de la surface de l'objet. Cette intensité dépend de l'orientation de la surface éclairée par rapport à la position de la source de lumière et de quelques propriétés des matériaux. Contrairement, au modèle d'illumination globale, l'éclairage indirect, les ombres et les caustiques ne sont pas pris en compte.

Dans notre cas, l'illumination du volume est calculée par le modèle d'illumination de Blinn-Phong [6] [27]. La couleur résultante est une fonction du gradient, de la lumière et de la direction de visualisation, aussi bien que les paramètres ambiant, diffus, et spéculaire, dont on rajoute la couleur du résultat de la classification.

Pour une lumière dont la réflexion est parfaitement spéculaire nous pouvons écrire :

$$I_{Phong} = I_{ambiant} + I_{diffuse} + I_{speculaire} \tag{I.11}$$

Le calcul des différents termes de cette équation sont données dans l'annexe C.

L'illumination est généralement calculée en utilisant le vecteur de visualisation et le vecteur de lumière dans l'espace-monde [26].

Alternativement, Wu et al [42] décrivent une solution élégante et efficace pour la correction de gradient en utilisant un espace intermédiaire de l'illumination [18].

I.4.6. Composition (*Compositing*)

La Composition est l'évaluation récursive de l'intégration numérique de l'équation *(I.4)* [16]. On peut faire de la composition de deux manières, suivant l'ordre du parcours du volume ; la composition **front-to-back**, correspond à l'accumulation de la couleur dans le sens du parcours du rayon de visualisation, tracé à partir de la position de l'observateur vers le volume, la composition **back-to-front**, correspond à l'accumulation de la couleur dans le sens inverse du parcours du rayon de visualisation.

La formulation récursive de la composition front-to-back est donné par:

$$\begin{aligned} &\hat{t}_0 = (1 - \alpha_0);\ \hat{C}_0 = \tilde{C}_0 \\ &\hat{C}_i = \hat{C}_{i-1} + \hat{t}_{i-1}\,\tilde{C}_i \\ &\hat{t}_i = \hat{t}_{i-1}\,(1 - \alpha_i) \end{aligned} \tag{I.12}$$

Tel que ;

$\hat{t}_i$ et $\hat{C}_i$, sont respectivement les résultats de l'itération courante de la transparence et de la couleur,

$\hat{t}_{i-1}$ et $\hat{C}_{i-1}$, sont les résultats de l'accumulation de l'itération précédente,$\hat{C}_i$ et α_i, sont respectivement l'échantillon de la couleur-associée et la valeur de l'opacité à la position courante de ré-échantillonnage.

Dans la composition front-to-back, les transparences accumulées $\hat{t}_i$ (*équation I.12*) n'ont pas besoin d'être mis à jour. Cependant, elles sont utiles si l'image finale est composée sur un autre arrière-fond ou pour des volumes et polygones mélangés.

La formulation de la récursivité back-to-front pour n points d'échantillons est donnée par [14] :

$$\begin{aligned} &\hat{t}_n = (1-\alpha_n);\ \hat{C}_n = \tilde{C}_n \\ &\hat{C}_i = \hat{C}_{i-1}(1-\alpha_i) + \tilde{C}_i \\ &\hat{t}_i = \hat{t}_{i-1}(1-\alpha_i) \end{aligned} \tag{I.13}$$

Comme le coefficient d'extinction mesure l'absorption de la lumière à travers le volume par unité de longueur, la valeur de l'opacité α doit être adaptée à la distance entre les échantillons interpolés. Cette graduation est appelé correction de l'opacité. Pour une opacité définie pour une position d_{old}, et les échantillons sont espacés d'une distance d_{new}, la graduation devient [14] :

$$\alpha_{corrigé} = 1 - \left(1 - \alpha_{enregistré}\right)^{\frac{d_{old}}{d_{new}}} \tag{I.14}$$

Celle-ci peut être implémenté efficacement en utilisant une lookup-table pré-calculé qui stocke les $\alpha_{corrigé}$ en fonction des $\alpha_{enregistré}$ et des d_{new}.

Analogiquement, la correction de la couleur associée correspond à [14]:

$$\tilde{C}_{corrigé} = \tilde{C}_{enregistré}\left(\frac{\alpha_{corrigé}}{\alpha_{enregistré}}\right) \tag{I.15}$$

Les différentes étapes d'interpolation, de calcul du gradient, d'ombrage, et de classification s'effectuent sur une base locale, c'est-à-dire exécutées dans le voisinage, ou directement, au point d'échantillonnage. Ces étapes sont indépendantes de la méthode de rendu et peuvent être réutilisés dans différentes méthodes. Nous utilisons généralement des textures 2D ou 3D pour la sauvegarde des résultats.

I.5. Méthodes de rendu volumique

Dans la littérature scientifique, il existe un grand nombre de techniques utilisées en rendu volumique, les approches qui sont favorable à l'accélération matériel (GPU) sont, le ray-casting [21], le splatting [39], le shear-warp [20] et le shear-warp-splatting [5]. Dans notre travail nous avons choisi de travailler sur le ray-casting, pour la qualité d'image assez élevée qu'on peut obtenir et c'est un algorithme utilisé dans de nombreuses applications. C'est un algorithme qui nécessite beaucoup de mémoire et de temps de calcul, ce qui ne permet pas d'obtenir des temps de frame interactifs, sans l'utilisation du matériel graphique.

L'algorithme de ray-casting appartient à la classe Image-Order. Il consiste à émettre des rayons à partir du point de vision vers chaque pixel du plan image en parcourant le volume (*Figure I.5*), sur lesquels est prélevé un ensemble d'échantillons de voxel. La valeur à chaque point de prélèvement est obtenue par une interpolation tri-linéaire sur le voisinage. La contribution de ces échantillons est accumulée jusqu'à ce que le rayon ait quitté le volume. La valeur finale, calculée avec l'une des formules de composition *front-to-back* ou *back-to-front* respectivement selon les équations (*I.12*) et (*I.13*), est placée sur le pixel que le rayon atteint [20], l'ensemble des rayons émis reconstitue l'image d'origine.

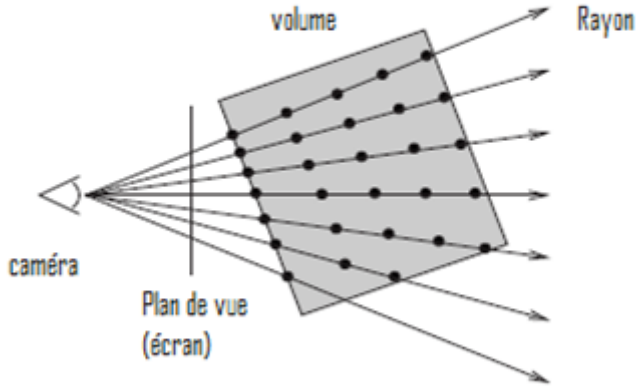


Figure I.5 : *L'algorithme de ray casting.*

De nombreuses techniques sont utilisées pour accélérer le ray casting de manière 'brute'. L'une de ces techniques d'accélération est basée sur le matériel graphique.

Typiquement on distingue deux approches se basant sur les cartes graphiques [43], celle utilisant la texture mapping, rendu volumique basé texture, et celle utilisant les unités du processeur graphique, rendu volumique basé GPU. La première approche a été à l'origine, proposé par CULLIP et NEUMANN [10] et développé par CABRAL et al. [8]. L'algorithme utilise directement les fonctionnalités de mappage de texture du matériel graphique par substitution de plan de ré-échantillonnage, qui peut être soit à axe-aligné [28] selon trois axes de la pile de texture 2D ou à vision-alignée [9] avec la texture 3D. Il peut atteindre des fréquences d'images interactives, mais il produit une qualité d'image relativement faible, surtout dans les cas de vues proches. La seconde approche, basé sur les unités du GPU est devenu une technique de choix pour l'implémentation standard du ray-casting. KRÜGER et WESTERMANN [19] ont proposé le ray-casting basé GPU. BENTOUMI et al. [3] ont proposé l'algorithme de shear-warp basé GPU. Les algorithmes de rendu volumique basé GPU peuvent générer des rendus de haute qualité à des fréquences d'images interactives.

I.5.1. Ray-casting basé GPU traditionnelle

L'algorithme présenté par STEGMAIER et al. [33] représente le ray-casting classique, dans lequel l'ensemble des données sont stockés comme une texture 3D pour tirer parti de l'interpolation tri-linéaire intégré dans le matériel graphique. Dans cet algorithme, une boîte

de contours est créée pour l'ensemble des données, généralement on utilise un cube unitaire, dont les coordonnées de début et de fin de chaque rayon sont encodées dans le canal de couleur des faces de la boîte.

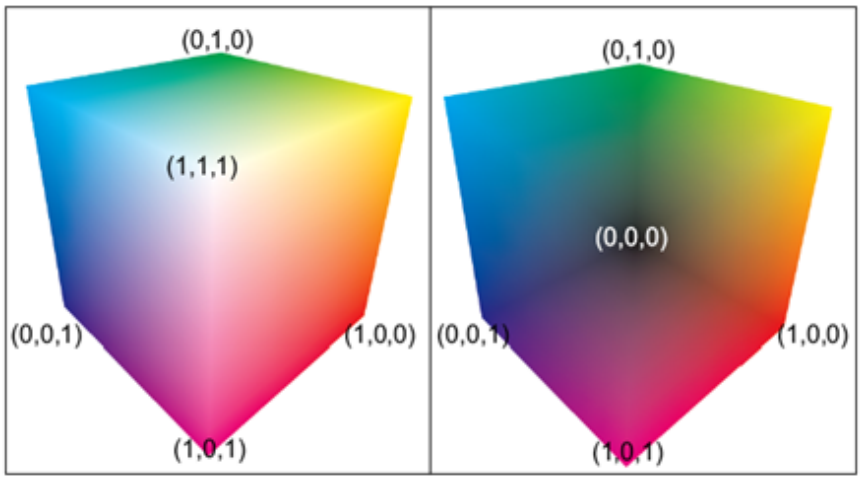


Figure I.6 : *Texture 3D les couleurs des faces avant (à gauche) et arrière (à droite) correspond respectivement aux positions de départ et aux positions finales.*

La couleur des faces avant, est considérée comme la position de départ du processus de ray-casting. Et la couleur des faces arrière, est considérée comme la position finale. La direction du rayon à un pixel donné peut être calculée en soustrayant la couleur de la face avant à la couleur de la face arrière dans la même position (*Figure I.6*). Deux passes de rendu sont effectuées [48], c'est-à-dire une passe pour la face avant et une autre pour la face arrière.

I.5.2. Ray-casting basé GPU proposé:

Dans l'algorithme ray-casting basé GPU proposé à l'aide des shaders, dans le cadre de ce travail, et qui diffère des procédés traditionnels de ray-casting basés sur le GPU, nous n'utilisons pas de texture 3D pour le calcul des équations des rayons et des intersections rayon-plan. L'ensemble de ces calculs se fait par le GPU, en décrivant tous les calculs de la géométrie analytique dans celui-ci, au lieu d'effectuer deux passes de rendu. Ceci est réalisé en transmettant la position de visualisation et la position de la source lumineuse au fragment processeur où nous calculons les différentes intersections de chaque rayon tracé avec le volume.

Conclusion :

Comme nous avons pu le voir dans ce chapitre, le rendu volumique direct est une technique qui recherche à travers la discrétisation de l'équation de rendu volumique à créer une visualisation réaliste. La manière de discrétiser cette équation donne lieu à de nombreuses techniques. La programmation de ces techniques d'accélération du matériel graphique donne lieu à plus de performance et de réalisme. Il y a deux approches de ray-casting basé sur l'accélération du matériel graphique, celle qui utilise la texture mapping et celle qui utilise l'unité de processeurs graphique (GPU).

Après avoir vu les différentes étapes à suivre pour faire du rendu volumique et situé les différentes approches de l'algorithme de ray-casting basé sur l'accélération du matériel graphique ainsi que l'approche que nous proposons, dont l'implémentation est détaillée au *chapitre III*, nous allons introduire dans le chapitre suivant les bases de la programmation des GPU ainsi que la notion des shaders et du langage de programmation dédié GLSL ce qui nous permettra de créer un programme shader et de mieux situer les étapes d'implémentation du ray-casting sur GPU.

CHAPITRE II
GPU PROGRAMMABLE ET API DE PROGRAMMATION GLSL

La connaissance des caractéristiques du matériel et l'API utilisée pour exploiter ce matériel afin d'afficher une scène, est l'une des premières étapes à effectuer avant de passer à l'implémentation de n'importe quel algorithme. Cette étape est dans notre travail très importante et qui nous à pris le plus de temps. Les cartes graphiques disposent de processeurs très performants. De nombreuses fonctionnalités ont été incorporées afin d'effectuer un certain nombre de calculs, la recherche des parties visibles, le colorie des facettes, la création des effets de transparences et des effets de brouillards. L'ensemble des fonctions qu'exécute un GPU est assuré par le pipeline graphique. Des unités de ce pipeline sont accessibles à la programmation par des langages spécialisées dont l'utilisation nécessite des connaissances spécifiques. Il m'a été possible d'effectuer un stage dans ce cadre au niveau du Laboratoire de recherche de l'IRISA de Rennes, avec l'équipe BUNRAKU que dirige le P^{r} Kadi Bouatouch.

Les cartes graphiques sont entrées dans une nouvelle ère avec l'accroissement de la taille mémoire, de la fréquence et l'incorporation de processeurs programmables introduisant ainsi la notion de Shader. Ceci permet aux programmeurs d'exécuter plus rapidement et efficacement leurs programmes.

Dans ce chapitre, nous donnons une vue générale sur le pipeline graphique moderne d'un point de vue programmeur. Nous décrirons également le langage de programmation des *shaders OpenGL Shading Language*, ou GLSL, que nous avons utilisé pour implémenter notre algorithme.

II.1. Pipeline graphique

Dans une carte graphique, le flux des données décrivant une scène virtuelle est un ensemble de polygones planaires. Ces polygones sont décrits par un triplé de vertex. Le GPU est conçu pour produire des images rasters à partir de cet ensemble de polygones. Ce procédé est généralement constitué d'une succession d'opérations sous forme de pipeline, qui définit le pipeline graphique. Les deux unités programmables, le vertex shader et le fragment shader sont situées respectivement au niveau de l'étage de géométrie et de l'unité de rastérisation [14]. Le pipeline se décompose en trois grands étages *(Figure II.1) :*

- **Traitement des sommets** (*vertex processing*) cette phase de traitement réalise ce qu'on appelle les **opérations par *sommet***, elle prend en charge les opérations qui se font au niveau de chaque sommet, et plus particulièrement les transformations géométriques (rotation, translation,…) et d'éclairage [29] [44] dans l'espace 3D. Cette étape comprend également les transformations de passage d'un système de coordonnées à un autre. Le vertex processeur reçoit l'ensemble des attributs des vertex de l'application, niveau CPU, exécute le vertex program, et émet au final un ensemble d'attributs pour un vertex [14] [29]. L'unité d'assemblage des primitives forme les primitives géométriques, selon les attributs de chaque sommet, qui sont transmises au *fragment processeur* après avoir effectué les opérations de surfenêtrage (*clipping*), d'élimination (*culling*) et de recadrage (*viewport*), (*annexe B*).
- **Les opérations sur les pixels** (*fragment processing*) : cette phase de traitement effectue ce qu'on appelle les **opérations par fragments**, elle calcule la composition

de couleur et affecte les valeurs de la texture à chaque fragment. La première étape est la *rasterization* qui décompose les primitives géométriques en un ensemble de fragments. Chaque fragment correspond à un seul pixel de l'écran. Le calcul de chaque attribut d'un fragment se fait par interpolation des attributs des sommets formant la primitive. Le fragment processeur calcule la couleur finale du fragment depuis l'interpolation des attributs des sommets [14] [29] [44].

- **Accumulation** (Compositing) elle est l'étape finale avant que les fragments soient écrits dans le *frame buffer*. Plusieurs tests sont effectués à ce niveau, qui détermine au final la façon avec laquelle les fragments reçus sont éliminés ou affichés sur l'écran. Les opérations du frame buffer décident aussi quelle couleur des fragments reçus est combinée avec la couleur stockée dans le frame buffer à la position raster du pixel correspondante [29].

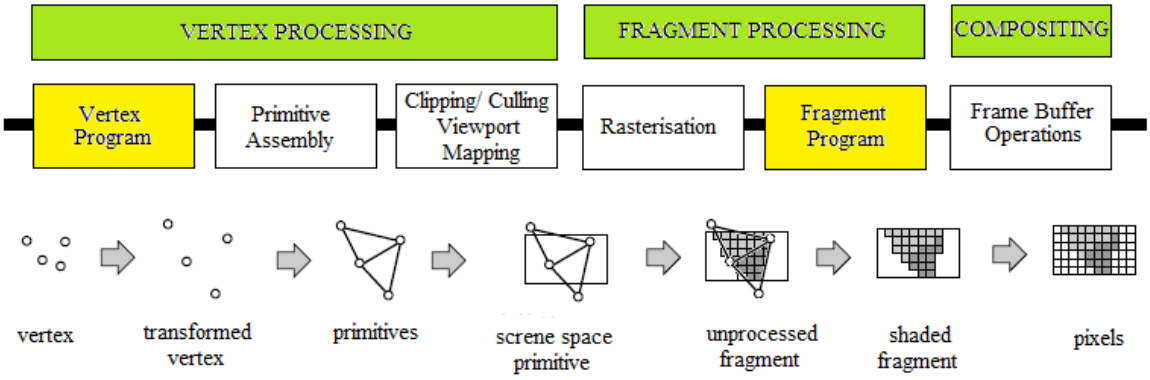


Figure II.1 : *Le Pipeline Graphique.*

L'illumination locale peut être calculée par le *vertex processeur* pour chaque sommet durant l'étape de traitement géométrique, les termes d'illumination sont interpolés pour chaque pixel (Gouraud ou smooth shading) et peut également être calculée par le *fragment processeur* (Phong shading) [14].

Après que la scène ait complètement traversé le pipeline graphique, l'image raster qui en résulte, contenu dans le frame buffer, peut être affichée sur l'écran ou écrite dans un fichier.

II.2. Introduction à OpenGL Shading Language (GLSL)

OpenGL Shading Langage est un langage permettant la programmation GPU de scènes OpenGL. La programmation GPU se pratique au moyen de deux éléments types : le *vertex shader* et le *fragment shader*. Un *vertex shader* réalise des opérations sur un sommet alors qu'un *fragment shader* réalise des opérations sur un *pixel* ou *fragment*.

Il est possible d'envoyer des informations du programme C/C++ vers le programme GLSL mais pas dans le sens inverse, mis à part le résultat final visualisé à l'écran. Les *shaders* sont généralement écrits dans des scripts, en dehors du code. Les *shaders* ne sont donc rien de plus que des programmes, c'est-à-dire, un code source qui est compilé et linké. Ils se différencient toutefois des programmes écrits en C, C++ ou autre langage réservé à une

exécution CPU, de par leur compilation et leur exécution. La compilation d'un shader est effectuée lors du lancement de l'application, et l'exécution se passe au niveau du GPU, contrairement aux programmes habituels qui eux sont traités par le CPU, d'où la puissance et la flexibilité des shaders dans le rendu 3D.

La programmation des shaders nécessite des APIs dédiés, parmi les nombreux langages de programmation. OpenGL Shading Language (GLSL) est défini par ARBO (*Architectural Review Board of OpenGL*) sous la norme *OpenGL* 2.0 qui inclut les calculs d'ombre par GPU, (*annexe A et B*). GLSL est un langage qui ne dépend pas du matériel et de la plateforme utilisée et est basé sur une syntaxe et un contrôle de flux emprunté au langage C et C++. Il supporte nativement les opérations vectorielles et matricielles puisque celles-ci sont inhérentes à de nombreux algorithmes graphiques 3D [28].

Voici en résumé ce qu'il possible de réaliser avec OpenGL Shading Language [29]:

- de la matérialisation de plus en plus réaliste, des structures telles que la pierre, le bois, et les peintures,
- de l'éclairage de plus en plus réaliste des surfaces, des ombres douces,
- de modéliser les phénomènes naturelles comme le feu, la fumée, l'eau, les nuages,
- de faire du rendu avancé avec les effets globaux d'illumination, le lancer de rayons,
- des matériaux non réalistes,
- une utilisation efficace de la mémoire de texture en stockant par exemple : la normale, les valeurs de brillance, les coefficients polynomiaux,
- une utilisation procédurale des textures générant dynamiquement des textures 2D et 3D,
- du traitement de convolution, du masquage flou, du mélange complexe,
- de l'animation par interpolation du frame,
- une utilisation des méthodes *d'antialiasing*,
- une génération des calculs de tri, de la modélisation mathématique, de la dynamique des fluides.

II.2.1. Pipeline Graphique OpenGL

Avant d'écrire des *programmes shaders*, il est nécessaire de comprendre le fonctionnement du pipeline graphique qui lui est approprié et de connaitre le type de *shader* disponible, ses performances et ses limites.

Le pipeline graphique OpenGL 2.0 (*Figure II.2*), exprime la nature des traitements que subissent les flux de données à travers le pipeline. Les flux de données vont de l'application au *vertex processeur* vers le *fragment processeur*, pour finalement arriver au *frame buffer*. Les flux de données sont unidirectionnels, ce qui implique que le *vertex processeur* n'as pas l'accès aux données du *fragment processeur*. De plus, la mémoire de texture est accessible à la fois par le *vertex processeur* et le *fragment processeur*. Le contenu du *frame buffer* est accessible par l'application (niveau CPU) et peut aussi être transféré vers la mémoire de texture. L'application peut transférer des données au *vertex processeur* et au *fragment processeur*.

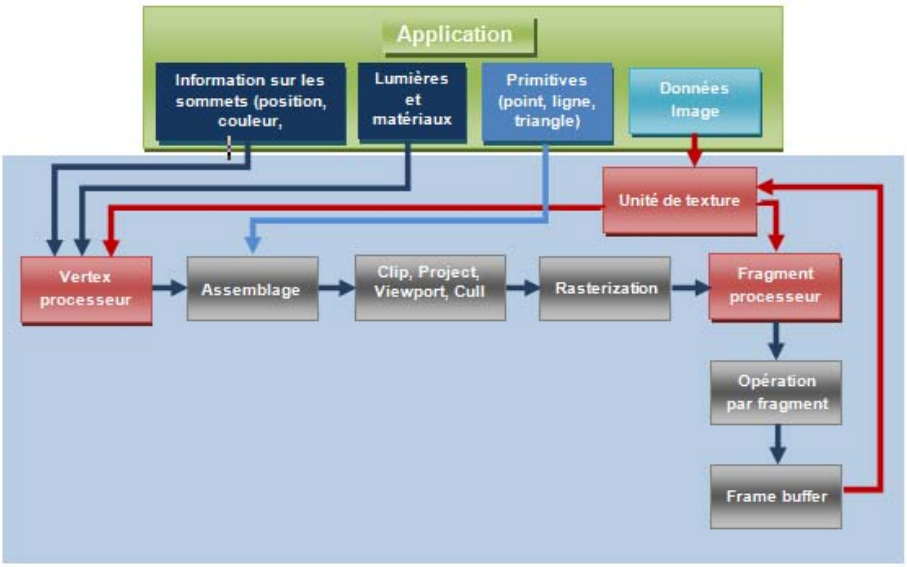


Figure II.2 : *Le Pipeline OpenGL.2.0.*

GLSL a été soigneusement conçu pour permettre l'implémentation matérielle et pour un traitement parallèle des sommets et des pixels, permettant d'augmenter considérablement la rapidité des cartes graphiques avec la mise en parallèle des processeurs. Le contrôle des flux de données qui transite entre l'application OpenGL et les différents étages programmables du pipeline graphique est possible grâce aux différentes variables et qualificateurs qu'OpenGL définit, en plus de l'accessibilité des états d'OpenGL [29] [44].

II.2.2. Les shaders

Les shaders servent à programmer le pipeline de rendu par défaut dans la carte graphique. Les shaders ne sont donc rien de plus que des programmes, c'est-à-dire un code source, une compilation et une exécution. Ils se différencient toutefois des programmes écrits en C, C++ ou autre langage réservé à une exécution CPU, de par leur compilation et leur exécution. La compilation d'un shader est effectuée lors du lancement de l'application, et l'exécution se passe au niveau du GPU, contrairement aux programmes habituels (C, C++, ...) qui eux sont traités par le CPU, d'où la puissance et la flexibilité des shaders dans le rendu 3D. Il y a deux types de *shaders* : le *vertex shader* qui remplace les calculs pour chaque sommet avant leur assemblage par primitive, et le *fragment shader* qui remplace les calculs propres à chaque pixel. Sur certaines cartes se développe un troisième shader, la *géométrie shader* située juste après l'assemblage par primitives et qui peut générer des nouvelles primitives, il est encore très peu répandu. Comme les opérations par sommet ou par pixel sont totalement indépendantes, la carte graphique a plusieurs *vertex processeurs* et *fragment processeurs* qui exécutent ces tâches en parallèle. Ce sont des processeurs à jeu d'instructions spécifiques de type *SIMD* (*Simple Instruction Multiple Data*), et la tendance se porte sur des processeurs capables d'exécuter aussi bien un *fragment shader* qu'un *vertex shader*, voir (*annexe B*).

II.2.2.1. Le vertex processeur

Le ***vertex processeur*** prend en charge l'exécution du contenu du *vertex shader*, les valeurs d'entrée du vertex shader sont des attributs de sommet ; la position, la couleur, la normale et les coordonnées de texture, selon ce que l'application transmet. Le vertex processeur opère sur un sommet à la fois [29] [44].

Un ***vertex shader***, ou vertex program, est un programme écrit pour remplacer la majorité des calculs effectués par l'unité de traitement géométrique du pipeline traditionnel. Il est utilisé pour optimiser les transformations sur les sommets et modifier les attributs des sommets. Le *vertex shader* est exécuté une fois par sommet [29] [44]. Un *vertex shader* peut effectuer les tâches suivantes :

- Transformation de position sur les sommets,
- Transformation sur les normales,
- Génération et transformation des coordonnées de texture,
- Illumination par sommet ou bien calcul des valeurs pour l'illumination par pixel,
- Calcul de couleur.

Ces tâches peuvent ne pas être toutes exécutées et seules les tâches contenues dans le bout de code du *vertex shader* sont exécutées. Un *vertex shader* devra toujours contenir la variable de sortie *gl_Position*, et la transformation des sommets par les matrices de *modelview* et de *projection* qui doit être transmise au fragment processeur.

Conceptuellement, le *vertex processeur* traite un vertex à la fois, mais une exécution peut avoir des *vertex processeurs* multiples qui fonctionnent en parallèle. La conception du *vertex processeur* est concentrée sur la fonctionnalité requise pour transformer et éclairer un simple sommet [29] [44].

II.2.2.2. Fragment processeur

Le ***fragment processeur*** se charge de l'exécution du *fragment shader*. Les données d'entrées du *fragment processeur* sont des valeurs interpolées résultantes de l'étape précédente du pipeline (positions, couleurs et normales des sommets).

Un ***fragment shader***, ou *fragment program*, est un programme écrit pour remplacer la majorité des calculs effectués par de l'unité de *rasterization* du pipeline traditionnel. Il est utilisé pour calculer la couleur finale de chaque pixel et peut également calculer la valeur de profondeur des pixels. Le *fragment program* est exécuté une seule fois par *fragment* (pixel) [29] [44]. Il peut réaliser les opérations suivantes :

- Opération sur les valeurs interpolées,
- Accès et Application de la texture,
- composition des couleurs.

A ce niveau, le traitement des fragments se fait à l'intérieur des primitives, d'où le besoin de valeurs interpolées. Le *fragment processeur* ne traite qu'un seul fragment à la fois,

il ne possède aucune information sur le voisinage du fragment traité. Il est important de souligner qu'un fragment shader ne peut pas changer la coordonnée d'un pixel, comme le fait un *vertex shader* grâce aux matrices de modélisation, de visualisation et de projection [29] [44]. Un *fragment shader* pourra donc soit :

- rejeter le pixel et par conséquent ne produira rien.
- calculer la couleur finale du pixel, rendre une valeur arbitraire ou bien calculer la profondeur du fragment bien qu'il n'est pas exigé, car l'étape précédente l'a déjà calculé.

Le *fragment shader* n'a aucun accès au *frame buffer*. Ce qui implique que les opérations telles que le mélange de couleur (*compositing)* se produisent seulement après l'exécution du *fragment shader* [29].

II.2.3. Flux de données

Le *vertex shader* et le *fragment shader* utilisent des variables d'entrée et des variables de sortie spécifiques qui leurs permettent de recevoir et d'envoyer des données. Ces données reçues seront traitées par les instructions contenues dans le code source du *shader*, pour obtenir un résultat qui sera ensuite renvoyé à l'étage suivant. On distingue trois types de variables (*annexe B*):

Les variables *attributs* : représentent les données transmises de l'application au *vertex processeur*, et elles définissent les attributs de chaque sommet transmis pour être traiter dans le *vertex processeur* ; position, couleur, normale et coordonnées de texture. Les variables attributs sont des variables qui sont définies en lecture seulement et qui changent fréquemment [29] [44].

Les variables *uniformes* : Elles ne sont pas utilisées pour changer un attribut de sommet. Les variables uniformes sont adaptées pour des valeurs qui restent constantes le long d'une primitive et peuvent être lus dans le *vertex shader* et dans le *fragment shader*. Utilisé pour transmettre des valeurs de l'environnement ; la position de la source de lumière et de l'observateur, des valeurs de contrôle et de changement de repère, et transmettre les textures. L'utilisation du type attribut et uniforme nécessite la récupération de leurs positions en mémoire [29] [44].

Les variables *varying* : elles définissent les données qui sont transmises du *vertex processeur* au *fragment processeur* uniquement, ce sont des valeurs interpolées de couleur, de coordonnées de texture et autre. Utilisé pour réaliser des calculs par fragment comme par exemple faire de l'illumination par fragment, il est nécessaire de transmettre les normales de chaque sommet de *l'application* au *vertex shader*, qui lui transmettra ces valeurs au *fragment shader* sous forme de *varying*. Elles sont dites varying car les valeurs de chaque sommet peuvent être différentes après interpolation. Ce type de variable doit être écrit dans le *vertex shader*, où l'on calcule la valeur de la variable pour chaque sommet, elles sont transmises par la suite au *fragment processeur*. Elles peuvent être lues dans le *fragment shader* [29] [44].

II.2.3.1. Entrées et sorties du vertex processeur

Les *vertex shader* exprime l'algorithme exécuté dans le *vertex processeur* pour produire des résultats fondés sur les données fournies en entrée par l'application (niveau CPU). Les variables d'entrées du *vertex processeur* sont des variables ***attributs*** et ***uniformes,*** incorporées ou bien définies manuellement, et des textures sous forme de ***sampler***. Les valeurs de sortie produite par le vertex processeur sont transmises à l'étage suivant, pour être interpolées (*Figure II.3*). Ces variables de sortie sont de type ***varying*** et de type ***spéciale*** [29] [44] (*annexe B*)**.**

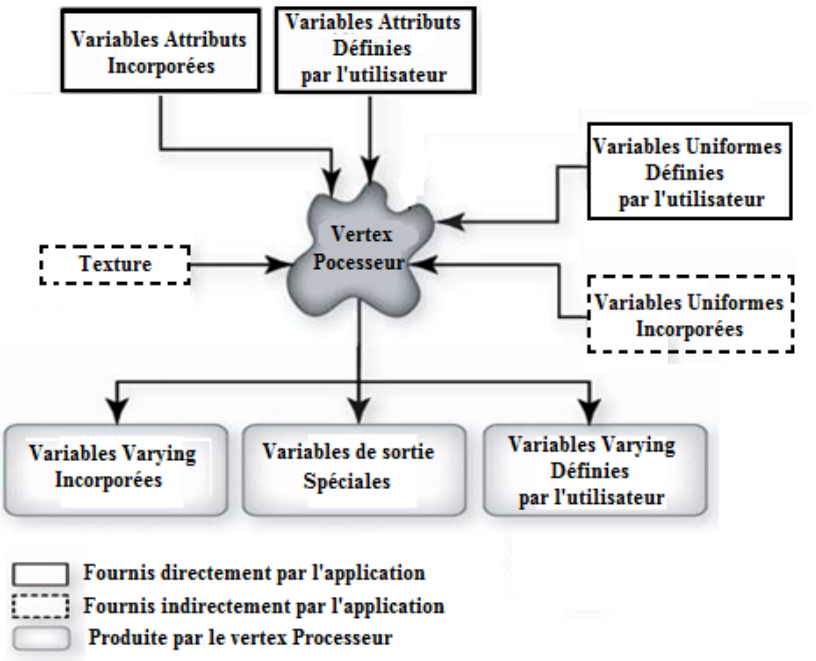


Figure II.3 : *Entrées et sorties du Vertex Processeur.*

II.2.3.2. Entrées et sorties d'un fragment processeur

Le *fragment shader* exprime l'algorithme exécuté dans le *fragment processeur* pour produire des résultats fondés sur les données interpolées fournies en entrée par l'étage de *rasterization* ou de l'application. Les variables d'entrées définies dans un *fragment shader* sont des variables ***uniformes, varying*** et des variables ***spéciales***. En sortie, le *fragment processeur*, après avoir traité les données reçues, transmet le résultat au *frame buffer* sous forme de variables ***spéciales*** qui sont des couleurs, la profondeur et autre (*Figure II.4*) [29]. Comme le *fragment processeur* a accès à la mémoire de texture il pourra donc recevoir en entrée des textures sous forme de *variable sampler* [28] [44] (*annexe B*).

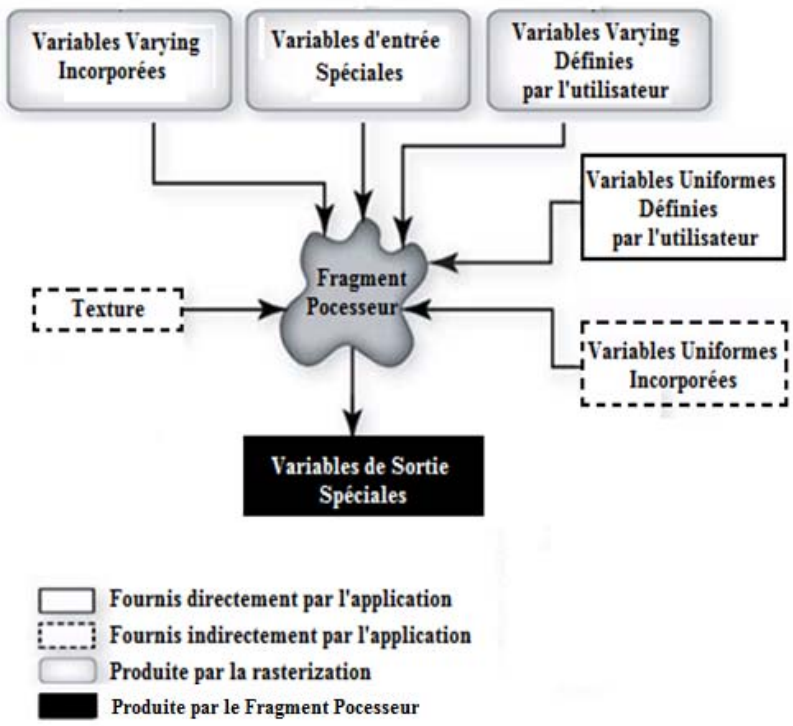


Figure II.4 : *Entrées et Sorties du Fragment Processeur.*

II.2.4. Création d'un Programme en OpenGL Shading Language

Après avoir vu l'architecture du pipeline de rendu, nous allons maintenant voir comment GLSL gère l'ensemble de ce pipeline et en particulier les unités programmable du point de vue software. Ce qui revient à créer un *programme shader* en GLSL, qui s'effectue en créant un *shader objet* et un *programme objet* (*annexe B*).

En résumé, on aura donc d'une part, à écrire le *shader* lui-même, c'est-à-dire le codage de la fonction qu'il remplira et l'effet graphique qu'il produira, et d'autre part, à définir son utilisation, quand et comment l'utiliser dans notre application OpenGL [29] [44].

Voici en générale les étapes et les fonctions pour créer un *programme shader* :

1. Créer un ou plusieurs *shaders objets* de type vertex ou fragment avec *glCreateShader*.
2. Fournir au *programme shader* le code source du shader avec *glShaderSource*.
3. Compiler chaque shader avec *glCompileShader*.
4. Créer un programme objet relatif au deux type avec *glCreateProgram*.
5. Attacher tous les shaders objets au programme objet avec *glAttachShader*.
6. Linker le programme objet pour vérifier son état avec *glLinkProgram*.
7. Utiliser le programme exécutable comme état d'OpenGL avec *glUseProgram*.
8. Libérer les ressources.

Le schéma qui suit (*Figure II.5*), illustre les étapes de création d'un programme shader.

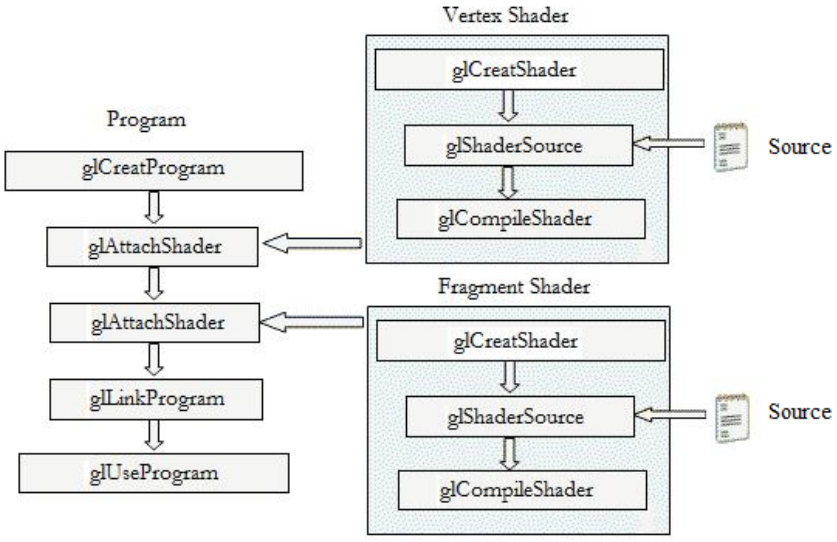


Figure II.5 : *Etapes de Création d'un programme en GLSL.*

II.2.4.1. Création d'un shader

La première étape pour créer un *shader* est de créer un *shader objet*. Un *Shader objet* peut être un *code objet* d'un *vertex shader* ou d'un *fragment shader*. Puis, on lui spécifie le code source du shader. Le code source créé doit alors être compilé. Un code source n'est autre qu'une chaîne de caractère. Quand un objet est créé, OpenGL renvoie un identifiant unique pour celui-ci. Cet identifiant peut être utilisé pour manipuler l'objet et pour assigner ou faire des requêtes sur les paramètres de l'objet.

II.2.4.2. Création et utilisation d'un programme

Pour créer un *programme* exécutable par la carte graphique, l'application a besoin d'un mécanisme permettant de spécifier une liste de *shader objet* à associé. Ce qui revient donc, à créer un *programme objet* auquel on attache les *shaders objets*, *vertex* et *fragment*. Pour créer un *program* valide, tous les *shaders objets* attachés au *programme objet* doivent être compilés et le *programme objet* lui-même doit être lié. Cette opération permet de lier le *vertex shader* et le *fragment shader*. Après l'opération de lien, la source des shaders peut être modifiée et les shaders recompilés sans affecter le programme. Quand l'opération de lien est accomplie avec succès, le programme qu'elle contient peut être installé en tant qu'élément de l'état courant du rendu.

Déboguer un shader est très dur. Comme toute programmation, il est nécessaire de vérifier le succès de chaque compilation et de chaque opération de lien (linkage) et dans le cas d'erreurs de pouvoir les localiser afin de les corriger. A cet effet, des fonctions sont prévues pour vérifier si le code est compilé et lié avec succès.

Chaque objet (*shader* et *programme*) créé doit être supprimé après l'avoir utilisé car il occupe de la ressource mémoire. Nous pouvons également aussi détacher un shader d'un programme quand celui-ci n'est plus utilisé.

II.3. La lumière et la texture en GLSL

II.3.1. La lumière

OpenGL utilise trois types de source de lumière : source ponctuelle, source directionnelle et un spot de lumière. Il offre la possibilité d'implémenter les différents modèles d'illumination entre autres le modèle de Gouraud, le modèle de Phong et le modèle de Blinn-Phong. Vu que les sommets et les pixels des facettes sont aisément accessibles avec GLSL, on a la possibilité de faire de l'illumination par sommet et de l'illumination par pixel (fragment). Comme la lumière peut être composée d'une couleur ou plus, elle est aussi définie par un triplé RVB.

GLSL définit la source de lumière et les matériaux avec un certain nombre de paramètres (ambiant, spéculaire, diffus,…) qui sont accessibles et calculables à tout moment par l'utilisateur et qui permettent de varier à la fois la nature de la source et son aspect ainsi que celle des matériaux [29].

Dans notre cas nous utilisons le modèle d'illumination de Blinn-Phong déjà cité Il est important de noter qu'avec OpenGL toute interpolation doit être normalisée et que chaque opération et transformation sur les normales doit être soigneusement faite. Puisque différents systèmes de coordonnées sont utilisés, dès lors qu'une normale subit une transformation celle-ci peut perdre le sens de normale.

II.3.2. La Texture

L'utilisation des textures est un procédé incontournable dans le processus de rendu que ce soit en visualisation scientifique ou autre. Elle permet de créer plusieurs effets visuels (éclairement, ombre, reliefs) plus proches du réalisme que la lumière et la couleur ne peuvent à elles seules créer. Ce qui rend leur utilisation plus efficace est la présence de zones mémoires dédiées aux textures incorporées dans les cartes graphiques, permettant ainsi d'utiliser les textures non seulement pour faire du placage, mais aussi pour stocker des résultats intermédiaires de rendu telles que les normales, les facteurs de perturbation normaux, les valeurs de brillances, l'information sur la visibilité, et les coefficients polynômiaux. OpenGL offre une flexibilité qui permet d'utiliser la mémoire de texture pour divers objectifs.

GLSL donne la possibilité d'accès simultanée à 8 unités de texture, et permet ainsi l'accès aux coordonnées de texture par sommet. Un certain type de qualificateur (*sampler*) est utilisé pour manipuler les textures. L'utilisation d'une texture se fait en exécutant les étapes suivantes :

- Réserver un descripteur de texture,
- Spécifier cette texture comme texture courante,
- Ajuster certains paramètres spécifiant le mode de filtrage et le mode de placage de texture,
- Transmettre l'image de texture dans la mémoire vidéo,
- Activer la texture,
- Fixer les coordonnées de texture.

Conclusion :

Dans ce chapitre, nous avons pu voir que trois phases de traitement au niveau de la carte graphique définissent le pipeline graphique, les opérations sur les sommets, les opérations sur les fragments et la composition. Le pipeline graphique est accessible à la programmation via le vertex processeur et le fragment processeur qui se situent respectivement au niveau des deux premières phases de traitement. Ces processeurs prennent en charge l'exécution des shaders qui leurs est spécifique, le vertex shader et le fragment shader. La création d'un programme shader s'effectue en créant des shaders objets et d'un programme objet auquel ils sont liés, sous un langage de programmation de shader.

Après avoir pris connaissance des différents étages du pipeline graphique côté hardware et du pipeline OpenGL côté software, ceci afin cerner les a particularités de la programmation en GLSL et la façon de créer et de programmer un shader, dans le chapitre suivant nous allons voir les différentes étapes de notre implémentation

CHAPITRE III

CONCEPTION ET MISE EN ŒUVRE

Comme nous l'avons vu au chapitre II, les techniques de rendu volumique cherchent à mieux approcher numériquement l'équation de rendu, en mettant en avant soit la qualité du rendu, soit le temps de rendu. Les approches qui tirent profit de l'accélération du matériel graphique sont les plus sollicitées dans la visualisation médicale, car les cartes graphiques offrent un nouveau souffle dans la manière d'implémenter les différents algorithmes de rendu.

Cette nouvelle génération de cartes graphiques intègre des unités programmables (shaders) qui permettent d'implémenter des algorithmes dédiés au calcul de l'illumination d'un rendu quelconque et pouvant effectuer différents calculs, donnant ainsi plus de liberté au CPU pour effectuer d'autres tâches. Dans le ray-casting basé GPU classique, les différents calculs de la géométrie analytique se font au niveau du CPU. Dans notre approche nous réalisons ces différents calculs au niveau du fragment shader. Comme le ray-casting ne prend pas en considération les occlusions indirectes, nous rajoutons le calcul d'ombre pour avoir un résultat plus réaliste. C'est ce que nous allons illustrer dans ce chapitre.

III.1. Description des différentes étapes

Avant tout rappelons le principe de base du ray casting Le ray-casting représente une évaluation numérique directe de l'équation de rendu volumique. Il utilise les lois de l'optique géométrique. Il consiste à suivre le trajet inverse des rayons lumineux afin de calculer les propriétés lumineuses du volume à visualiser. Le volume est prélevé le long du rayon, à intervalles réguliers et le calcul de la contribution des échantillons déterminent la valeur du pixel final à afficher (*Figure III.1*).

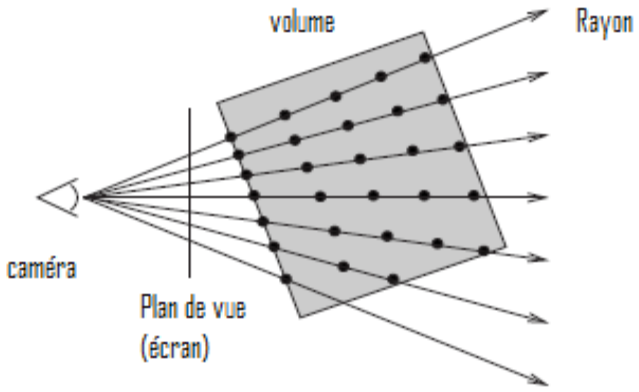


***Figure III.1**: Principe de base du ray casting.*

Après ce bref rappel, nous décrivons les différentes étapes de notre mise en œuvre.
A partir de la position de visualisation nous émettons un rayon vers chaque pixel du plan image (écran) en parcourant le volume. A ce stade il est utile de considérer que le volume est contenu dans une primitive géométrique, généralement un cube unitaire, qui est utilisé pour déterminer les points d'intersection (entrée/sortie) du rayon avec le volume. Le long de la portion du rayon contenu dans le volume, nous prélevons un nombre d'échantillons, de façon équidistante, dont les valeurs à chaque position sont interpolées tri-linéairement, car dans la plupart des cas, l'échantillon se trouve entre deux voxels [29] [34].

Après l'échantillonnage, nous calculons l'illumination reçue à chaque position, en évaluant la valeur du gradient ce qui permet de déterminer l'orientation des surfaces locales dans le volume. Préalablement, nous appliquons une fonction de transfert qui permet d'associer une couleur (RVBA) selon les zones d'intérêt par l'intermédiaire d'une table de consultation pour chaque voxel [13] [22]. Par la suite, nous accumulons la contribution de la couleur de ces voxels jusqu'à ce que le rayon quitte le volume, ce qui est calculé suivant la formulation de la composition décrite en section *(§I.2.2.7.)*. Au final, nous obtenons une visualisation à trois dimensions, du volume de données.

Dans cette description, nous voyons que l'on ne prend pas en considération l'occlusion due aux voxels se trouvant sur la trajectoire de la source lumineuse, ce qui ne permet pas d'avoir les ombres dues à cette occlusion. Pour cela, nous avons ajouté cette contribution en calculant l'atténuation que subit le rayon de lumière en pénétrant dans le volume, le calcul d'***ombrage*** ou le ***shadow***.

III.2. Calcul d'ombrage

Le procédé est identique dans la façon de tracer un rayon de lumière et dans l'étape de prélèvement des échantillons. Partant d'une position d'un voxel nous traçons un rayon vers la source de lumière, que nous appelons rayon d'ombrage. Puis suivant cette direction nous déterminons le point d'intersection de ce rayon avec le volume, ce qui permet de déterminer la portion contenue dans le volume. Sur cette portion nous prélevons un certain nombre d'échantillons, (*Figure III.2*), de manière équidistante : à chaque position, les valeurs sont rééchantillonnées en interpolant tri-linéairement par rapport au voisinage. Au final, pour ces échantillons nous calculons l'atténuation que subit la lumière par la loi exponentielle de l'absorption, (*équation III.1*).

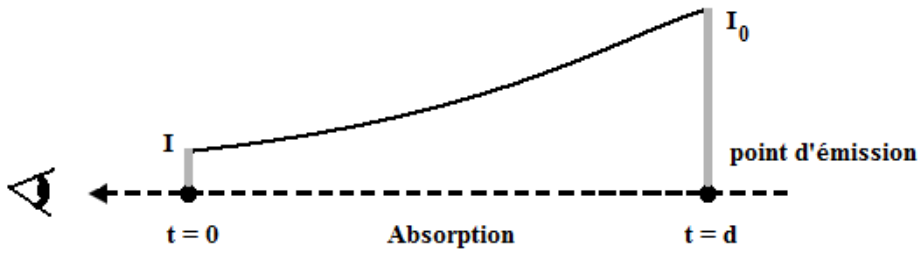


Figure III.2*: L'absorption de la lumière.*

L'atténuation que subit un rayon de lumière lorsqu'il traverse un milieu dépend des propriétés optiques de ce milieu [13]. L'absorption est estimée par la loi de Béer-Lambert, pour une émission radiante I_0 située à une distance D, l'énergie radiante I reçue est telle que :

$$I = I_0 \exp\left(-\int_D \kappa(s)\, ds\right) \qquad \text{(III.1)}$$

Où : $\kappa(s)$ est le coefficient d'absorption.

Le terme de l'intégrale, correspond à la profondeur optique qui caractérise physiquement la longueur à partir de laquelle la lumière est absorbée. Elle indique aussi la distance de propagation pour laquelle le rayon de lumière est diffusé. De grandes valeurs de la profondeur optique correspondent à un milieu opaque et de petites valeurs à un milieu transparent [13].

III.3. Conception et mise en œuvre

Nous avons développé un outil de visualisation sur une plateforme PC-Windows, permettant une visualisation volumique d'images médicales dans un but de diagnostic complémentaire mais aussi d'enseignement.

Avant toute programmation sur GPU il faut définir le type de matériel graphique et la version des shaders supportée. Nous avons utilisé deux modèles de cartes graphiques : une Geforce 9600(M) GT avec une mémoire de 512Mo d'une fréquence de 400 MHz.et une Geforce 9800 GT de mémoire de 512Mo d'une fréquence de 400MHz. Les programmes shaders sont conçus sous une plateforme Visual studio 2005.

Pour la mise en œuvre nous faisons les suppositions suivantes : nous créons une scène graphique composée d'un quad unitaire, d'une source de lumière et d'un observateur. Le modèle d'illumination que nous utilisons est celui de Blinn-Phong avec une source de lumière blanche, directionnelle et homogène. Nous rendrons les données à visualiser à l'intérieur du quad et nous positionnons la source de lumière et notre observateur à une certaine distance.

L'ensemble des opérations que nous effectuons au niveau du CPU et du GPU sont décrites en figure *III.3*.

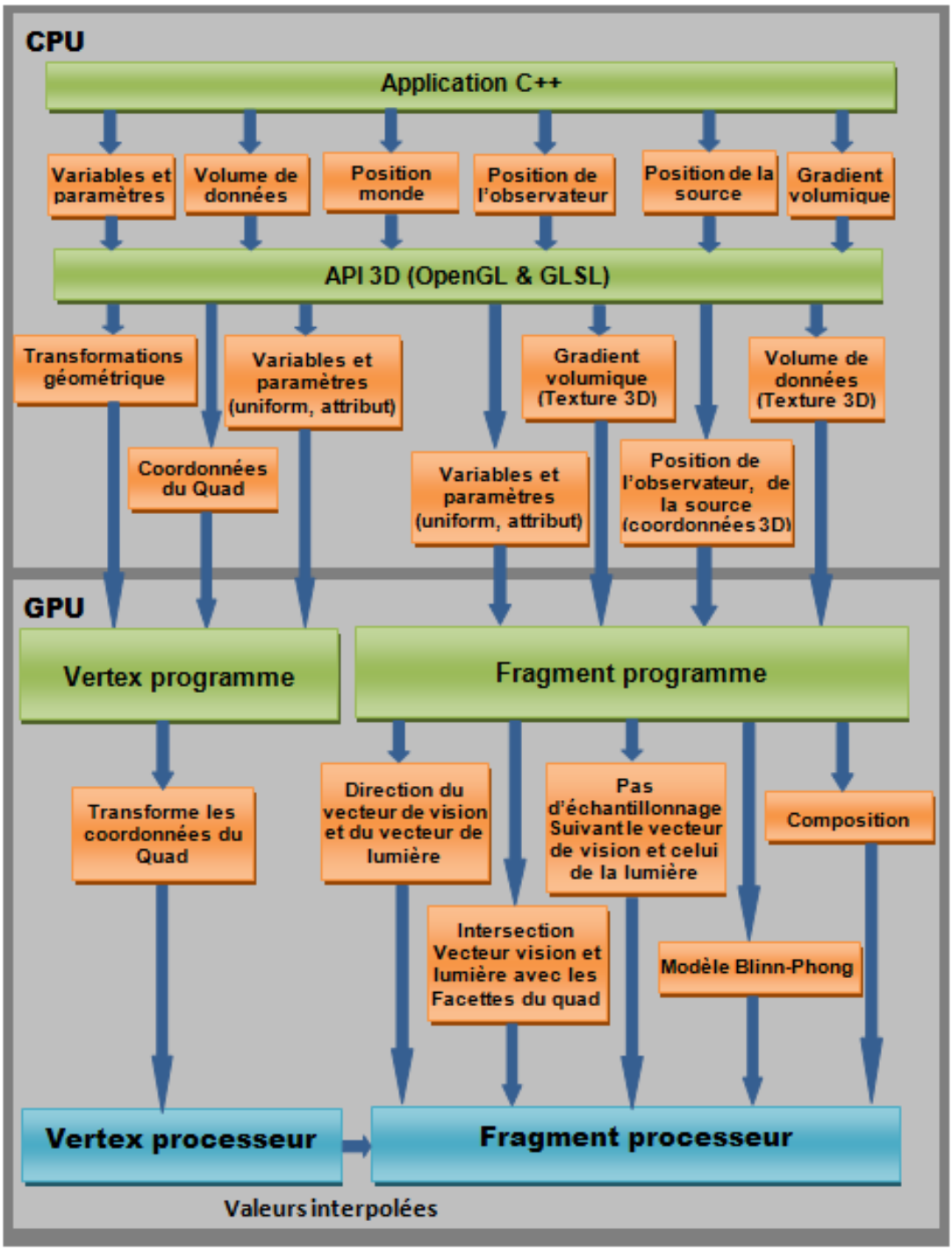


Figure III.3 : *Etapes de la mise en œuvre.*

III.4. Les étapes de l'implémentation

Nous exploitons GLSL pour écrire nos shaders sous une plate-forme *Visual Studio* 2005/ *VC++*. Afin d'utiliser les ressources de GLSL sous C++ nous installons les librairies GLEW et GLUT. GLEW comprend les ressources nécessaires pour écrire et communiquer avec un shader, qui devra être initialisé, et GLUT comprend les ressources d'affichage et de gestion des périphériques. En plus de ces librairies nous utilisons la librairie Devil pour manipuler plus aisément les différents formats d'image sous C++. La syntaxe de GLSL,

comme son nom l'indique, constitue une extension de l'API OpenGL dédié à la programmation des shaders. Nous retrouvons les matrices d'état d'OpenGL (*annexe A*) pour aborder les différentes étapes de rendu d'une scène : initialisation, fenêtrage, couleurs, éclairage, application de texture, gestion des ressources et des périphériques. Nous écrivons les codes sources définissant le vertex shader et le fragment shader comprenant les opérations à effectuer respectivement au niveau du vertex processeur et du fragment processeur, puis le code qui permet de transférer des données de l'application vers ces shaders. Ces données sont la position de l'observateur, la position de la source de lumière et les coordonnées de texture du volume de données. Ces données sont nécessaires pour les calculs d'intersection, l'application du modèle d'illumination et le calcul de l'accumulation.

III.4.1. Etapes réalisées au niveau de l'application (CPU)

Les étapes réalisées au niveau de l'application correspondent à l'ensemble du code écrit en C++ et celles réalisées au niveau du vertex et fragment shader correspondent à l'ensemble du code écrit en GLSL. Les étapes sont décrites comme suit :

a) Lecture des données du volume, contenues dans un fichier image de format RAW que nous sauvegardons dans une texture 3D pour être exploitées et être transmises au fragment shader.
b) Calcul du gradient du volume, selon l'équation (*I.10*), et nous sauvegardons le résultat dans une texture 3D pour être utilisé dans des calculs dans l'application et pour être transmis également au fragment shader pour le calcul du modèle d'illumination.
c) Récupération des données de la fonction de transfert sauvegardée dans un fichier image 2D, pour être transmise au fragment shader sous forme de texture 2D.
d) Initialisation de la position de la source de lumière O_l et la position de l'observateur O_v dans la scène, que nous transmettons au fragment shader.
e) Positionnement de manière correcte le cube unitaire dans la scène à l'aide d'une transformation rigide décrite par OpenGL (*annexe A*). Ce cube recevra la texture 3D correspondant au volume de données. Les coordonnées de texture sont transmises au vertex processeur pour être par la suite transmises au fragment shader.
f) Ecriture des codes sources du vertex et fragment shader en suivant les étapes décrites à la section (*§II.2.4*), puis nous libérons les ressources une fois utilisées.
g) Transmettre à chaque shader les différentes données dont il a besoin, sous forme de variable uniforme.

l'implémentation au niveau de l'application est la suivante :

- Initialisation du GLEW
- Lecture du volume de données
- Clacul du gradient volumique
- Filtrage du gradient volumique
- Calcul de la fonction de transfert
- Sauvgarde du volume dans une texture 3D
- Sauvegarde du gradient volumique dans une texture 3D

- Sauvegarde de la fonction de transfert dans une texture 2D
- Libére la resource mémoire
- Création et compilation du programme shader
- Activation des textures
- Utilisation du programme shader
- Declaration des variables uniformes
- Envoie des variables uniformes aux fragment shader
- Positionnement de caméra et de la source de lumière
- Création et positionnement d'un quad dans la scène à rendre
- Desactivation du programme shader
- Affichage du rendu sur l'écran.

III.4.2. Utilisation des textures

Nous utilisons les textures comme des tableaux de données pour stocker des valeurs qui peuvent être des résultats de calculs intermédiaires, calcul du gradient, calcul de la fonction de transfert et autres. Ces valeurs sont alors transmises au vertex et au fragment processeur sous la forme de variable de type *uniform sampler, (voir annexe B).*

Les données que nous utilisons pour notre rendu sont des données 3D d'imagerie médicale de type CT ou IRM que nous stockons dans une texture 3D avec une composante d'intensité dans chaque voxel. Pour visualiser ces données nous appliquons cette texture 3D sur un cube unitaire, ce qui facilitera les calculs sur ces données puisque les coordonnées de texture sont comprises entre $[0, 1]^3$. Lors de l'application de la texture, OpenGL nous donne la possibilité de définir le types de filtre à appliquer. Dans notre cas, nous utilisons un filtrage linéaire et le mode de texture est en mode répété.

Les étapes pour la création de la texture sont :

- Reserver un espace mémoire pour la texture (2D ou 3D)
- Definir le type de filtrage de texture
- Définire le type de placage de texture
- Création de la texture

Le calcul du gradient du volume est effectué au niveau du CPU, dont les résultats sont sauvegardés dans une texture 3D (RVBA) pour pouvoir le transmettre au fragment shader et être utiliser dans le modèle d'illumination de Blinn-Phong. Dans les trois premières composantes nous sauvegardons les valeurs du gradient et dans la dernière nous sauvegardons l'amplitude du gradient. Avant de pouvoir utiliser cette texture nous devons redéfinir les valeurs à sauvegarder entre 0 et 1.

Nous calculons le gradient à différence centrée à partir de l'interpolation des voxels voisins sur les trois directions x, y et z. La valeur du gradient suivant une direction est calculée en interpolant la valeur du voxel précédent et du voxel suivant de la position courante, *(équation I.10)*. Trois cas de figures peuvent se présenter selon la position du voxel considéré. L'algorithme suivant décrit ce calcul :

Pour tout voxel $V(i)$ du volume faire ;
Si le voxel $V(i-1)$et le voxel $V(i+1)$ font partir du volume faire ;

$$Grad(i) = \frac{V(i+1) - V(i-1)}{2}$$

Si le voxel $V(i-1)$ ne fait pas partir du volume faire ;

$$Grad(i) = \frac{V(i+1) - V(i)}{2}$$

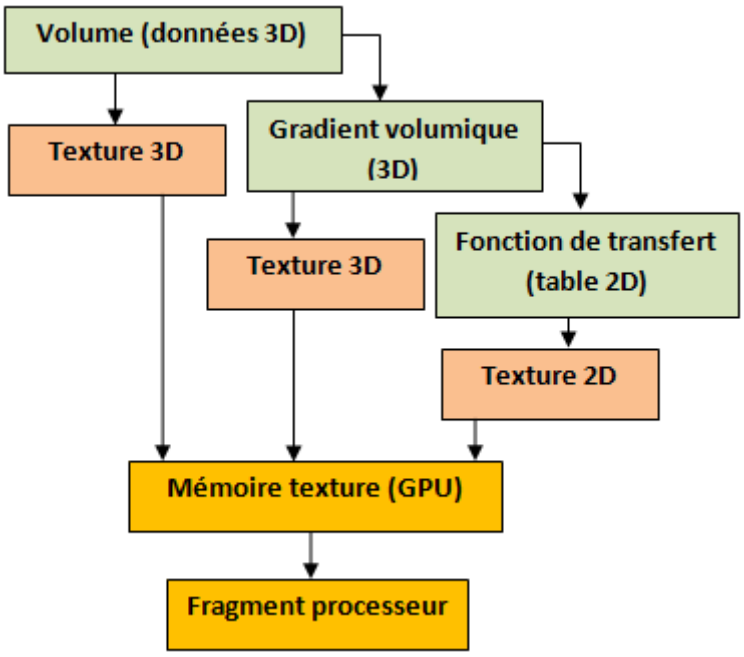
Si le voxel $V(i+1)$ ne fait pas partir du volume faire ;

$$Grad(i) = \frac{V(i) - V(i-1)}{2}$$

Pour pouvoir utiliser la normale il faut normaliser le gradient. Les valeurs calculées du gradient peuvent être grandes ce qui signifie l'existence de transitions brusques. Pour éviter cela nous appliquons un filtrage du gradient pour adoucir les transitions.

Le gradient est utilisé pour la détermination de la fonction de transfert, calculé à partir d'une table de couleur. Cette fonction de transfert est sauvegardée dans une texture 2D (RVBA).

Les données de notre volume, du résultat du calcul du gradient et de la fonction de transfert sont sauvegardées sous forme de texture, ce qui peut se schématiser comme suit :



III.4.3. Etapes réalisées au niveau du vertex et du fragment shader (GPU)

Nous programmons le *vertex shader* de telle sorte qu'il renvoie les coordonnées de chaque vertex par rapport au repère monde, après les avoir multipliées par la matrice de modélisation et de visualisation. Ces coordonnées sont transmises au *fragment shader* en utilisant des variables de type *varying*.

Puisque le plaquage de texture 3D se fait sur un cube unitaire, les coordonnées de texture et les coordonnées monde du cube sont identiques. Ceci nous permet de raisonner en coordonnées de texture.

Les calculs analytiques nécessaires pour générer les différents vecteurs et calculs d'intersection entre le volume de données, sont réalisés au niveau du *fragment shader* à partir des coordonnées transmises de l'application sous forme de *variables uniforme* et *varying*. Nous programmons le *fragment shader* de sorte qu'il récupère les différentes données envoyées par l'application et le *vertex shader*.

La lecture des données contenues dans les textures transmises par l'application au *fragment shader* s'effectue comme suit :

- Récupération de la position courante du texel
- Récupération de la texture 3D stockée en mémoire
- Lecture de la valeur du texel

L'implémentation au niveau du *fragment shader* s'effectue en plusieurs étapes. Pour chaque rayon lancé nous :

a) traçons le rayon de visualisation partant de la position de l'observateur O_v vers le volume de données, (*figure III.4*).

b) déterminons la position d'entrée O_e du rayon de visualisation dans le volume ; dans notre cas, les positions d'entrées correspondent aux premières valeurs de l'itération des coordonnées de texture transmises au fragment shader.

c) déterminons la position de sorti*e* O_s du rayon de visualisation du volume, suivant la trajectoire décrite par le vecteur directeur $\vec{V}_v$.

Le segment $\overline{O_e O_s}$ contenu dans le volume permet de fixer le pas d'échantillonnage, selon le nombre d'échantillons désiré. Nous verrons ci-dessous comment l'on détermine les points d'intersection sur les six faces du cube, (*figure III.4*).

d) rééchantillonnons les positions des voxels le long du segment $\overline{O_e O_s}$, avec une interpolation tri-linéaire pour obtenir la valeur adéquate à cette position. Cette étape est réalisée automatiquement par la carte graphique.

e) appliquons la fonction de transfert au volume de données, en chargeant la texture 2D contenant les classes de couleur à appliquer au volume.

f) traçons un rayon de lumière à partir de chaque échantillon O_i du segment $\overline{O_e O_s}$ vers la position de la source de lumière O_l, (*figure III.4).*

g) Suivant le vecteur directeur $\vec{V}_l$, nous déterminons la position d'entrée O_{le} par laquelle la lumière pénètre dans le volume. Le segment $\overline{O_{le} O_i}$ permet de calculer le pas d'échantillonnage pour un nombre d'échantillons choisi, (*figure III.4*).

h) Nous calculons l'atténuation que subit le rayon de lumière, en parcourant le volume, le long du segment $\overline{O_{le} O_i}$.

i) Nous appliquons le modèle de Blinn-Phong à chaque position O_i, après avoir récupéré les valeurs du gradient transmises par l'application, au fragment shader sous forme de

texture 3D. On tient compte dans les calculs de l'atténuation de la source de lumière, (*figure III.7*).

j) Nous calculons la composition de couleur selon le front-to-back le long du segment $\overline{O_e O_s}$.

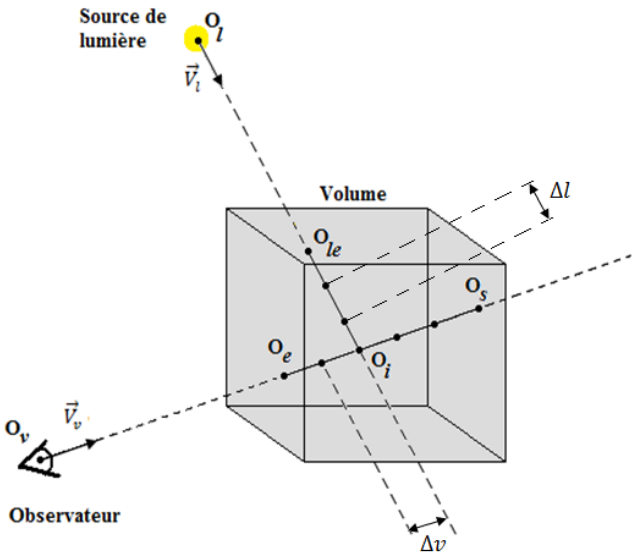


***Figure III.4**: Détermination du pas d'échantillonnage selon la direction de visualisation*

III.5. Représentation et calcul géométrique

Quand nous parlons de géométrie, la première chose à faire est de choisir un référentiel dans lequel tous les calculs se reporteront. En OpenGL plusieurs référentiels peuvent être définis, repère monde, repère de la scène, repère camera, repère objet et également repère de texture. Dans tout ce qui suit, nous reportons les différents calculs au repère monde, ce qui se fait en multipliant les différents sommets par la matrice de modélisation et de visualisation.

Comme le volume de données est affiché sur un cube unitaire dont les coordonnées sont connues, nous nous contentons de tracer les rayons à partir du point de visualisation vers le volume, délimité par ce cube uniquement. En calculant les points d'intersection d'entrée/sortie de chaque rayon nous réduisons l'échantillonnage à la portion du volume délimité par ces points, évitant ainsi d'effectuer des calculs sur toute la fenêtre d'affichage. Dans notre cas, les coordonnées du cube et les coordonnées de la texture 3D (données volumique) sont identiques, ce qui simplifie le calcul des premiers points d'intersection du rayon de visualisation avec le volume. Elles correspondent aux premières coordonnées de texture des premières itérations reçues par le f*ragment shader*.

III.5.1. Représentation des vecteurs de visualisation et de lumière

Nous définissons la direction de visualisation $\vec{V}_v$ comme étant la direction du vecteur partant de la position de l'observateur O_v vers un voxel de la face avant du volume, la position d'entrée du rayon dans le volume correspond à O_e, (*figure III.5*), tel que :

$$\vec{V_v} = \overrightarrow{O_m O_e} - \overrightarrow{O_m O_v} / \left\| \overrightarrow{O_m O_e} - \overrightarrow{O_m O_v} \right\| \text{ ou } \vec{V_v} = \overrightarrow{O_v O_e} / \left\| \overrightarrow{O_v O_e} \right\| \quad \text{(III.2)}$$

O_m, définit l'origine du repère monde.

Nous définissons la direction de la source lumineuse $\vec{V_l}$ comme étant la direction du vecteur partant de la position de la source lumineuse O_l vers une position quelconque O_i du volume, (*figure III.5*), tel que :

$$\vec{V_l} = \overrightarrow{O_m O_i} - \overrightarrow{O_m O_l} / \left\| \overrightarrow{O_m O_i} - \overrightarrow{O_m O_l} \right\| \; ou \; \vec{V_l} = \overrightarrow{O_l O_i} / \left\| \overrightarrow{O_l O_i} \right\| \quad \text{(III.3)}$$

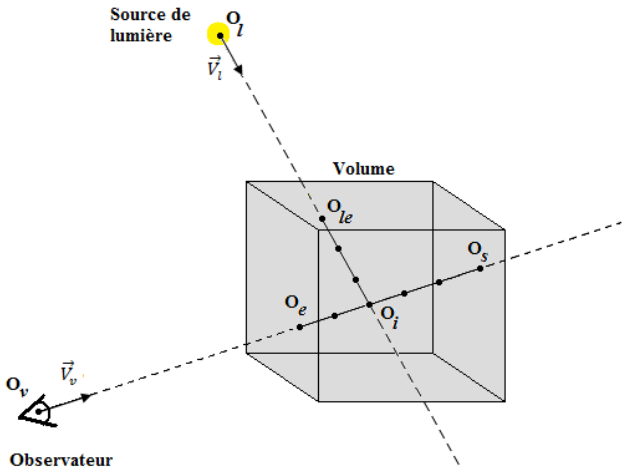


***Figure III.5**: Représentation du vecteur de visualisation ($\vec{V_v}$) et de lumière ($\vec{V_l}$)*

L'implémentation les étapes au niveau au niveau du *fragment shader* est :

- Récupération de la position courante du voxel
- Récupération de la position de la caméra
- Récupération de la position de la source de lumière
- Calcul du vecteur de vision
- Calcul du vecteur de lumière

III.5.2. Calcul d'intersection entre $\vec{V_v}$ et $\vec{V_l}$ et le Volume

Comme nous l'avons défini précédemment les vecteurs $\vec{V_v}$ et $\vec{V_l}$ sont respectivement les vecteurs directeurs de visualisation et de lumière. Les équations des droites portées par ces vecteurs sont faciles à déterminer. Soit un point quelconque $O_x(t)$ de la droite suivant la direction du vecteur de visualisation $\vec{V_v}$ et qui passe par le point O_e , nous avons :

$$\overrightarrow{O_m O_x}(t) = \overrightarrow{O_m O_e} + t \cdot \vec{V_v} \quad ; \; avec \; t \geq 0 \quad \text{(III.4)}$$

De manière simplifiée nous avons :

$$\vec{O_x}(t) = \vec{O_e} + t \cdot \vec{V_v} \quad ; \; avec \; t \geq 0 \quad \text{(III.5)}$$

Soit un point quelconque $O_y(t)$ de la droite suivant la direction du vecteur de lumière $\vec{V_l}$ et qui passe par le pointO_i, nous avons :

$$\overrightarrow{O_m O_y}(t) = \overrightarrow{O_m O_i} + t \cdot \vec{V_l} \quad ; \; avec\; t \geq 0 \tag{III.6}$$

De manière simplifiée nous avons :

$$\vec{O_y}(t) = \vec{O_i} + t \cdot \vec{V_l} \quad ; \; avec\; t \geq 0 \tag{III.7}$$

a) Equation des faces du cube :

Un plan (P) peut être défini par un point O_p de ce plan et de sa normale $\vec{N}$. Donc pour tout point $O_x(t)$ appartenant au plan (P) il vérifie l'équation suivante :

$$(\vec{O_x}(t) - \vec{O_p}) \cdot \vec{N} = 0 \tag{III.8}$$

Comme notre volume n'est autre qu'un cube unitaire dont les coordonnées et ses normales sont connues, il est facile d'en déduire les équations de chaque plan, *figure* (*III.6).*

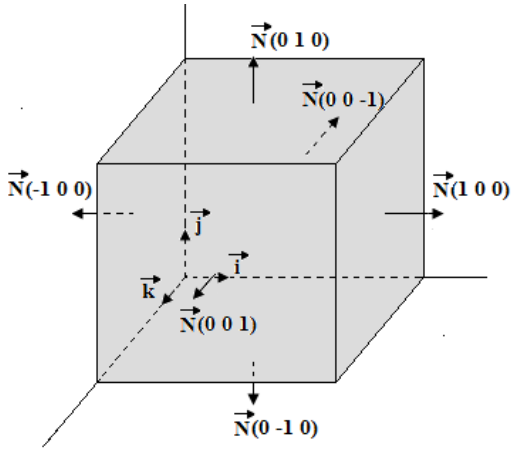


Figure III.6*: Représentation des normales sur les six faces du cube.*

Les équations de chaque plan sont :

$$\left.\begin{array}{ll}
\text{Pour } \vec{N}(1 \quad 0 \quad 0), \text{ est défini par :} & x - 1 = 0 \\
\text{Pour } \vec{N}(0 \quad 1 \quad 0), \text{ est défini par :} & y - 1 = 0 \\
\text{Pour } \vec{N}(0 \quad 0 \quad 1), \text{ est défini par :} & z - 1 = 0 \\
\text{Pour } \vec{N}(-1 \quad 0 \quad 0), \text{ est défini par :} & x = 0 \\
\text{Pour } \vec{N}(0 \quad -1 \quad 0), \text{ est défini par :} & y = 0 \\
\text{Pour } \vec{N}(0 \quad 0 \quad -1), \text{ est défini par :} & z = 0
\end{array}\right\} \tag{III.9}$$

b) Détermination des positions d'intersection :

1. Entre le rayon de visualisation et le volume

Pour un rayon tracé vers le volume de données, les positions d'entrée O_e du rayon de visualisation dans le volume sont simples à récupérer dans le fragment shader. Elles correspondent aux premières coordonnées de texture traitées de la face avant du volume. La seconde intersection O_s, position par laquelle le rayon quitte le volume, est déterminée analytiquement en trouvant la valeur scalaire de *t*, solution du système :

$$\vec{O}_s(t) = \vec{O}_e + t \cdot \vec{V}_v \quad ; \ avec\ t \geq 0 \tag{III.10}$$

$$(\vec{O}_s(t) - \vec{O}_p) \cdot \vec{N} = 0 \tag{III.11}$$

D'où : $$t = (\vec{O}_p - \vec{O}_e) \cdot \vec{N} / \vec{V}_v \cdot \vec{N} \tag{III.12}$$

t peut avoir six valeurs, puisque un plan par définition est infini donc les six plans formant le volume serons intersectés par le rayon de visualisation à des points différents tels que :

$$\left.\begin{array}{ll} \text{Plan de normale } \vec{N}(1 \quad 0 \quad 0): & t_1 = (1-x)/\ V_v \\ \text{Plan de normale } \vec{N}(0 \quad 1 \quad 0): & t_2 = (1-y)/\ V_v \\ \text{Plan de normale } \vec{N}(0 \quad 0 \quad 1): & t_3 = (1-z)/\ V_v \\ \text{Plan de normale } \vec{N}(-1 \quad 0 \quad 0): & t_4 = (-x)/\ V_v \\ \text{Plan de normale } \vec{N}(0 \quad -1 \quad 0): & t_5 = (-y)/\ V_v \\ \text{Plan de normale } \vec{N}(0 \quad 0 \quad -1): & t_6 = (-z)/\ V_v \end{array}\right\} \tag{III.13}$$

V_v , est la valeur scalaire du vecteur $\vec{V}_v$.

Nous constatons que pour $V_v = 1$, t_i correspond à la distance entre la position O_e de la face avant du volume et du point d'intersection de l'un des plans. Afin que ce point corresponde au point de sortie O_s du volume recherché nous choisissons le minimum positif des valeurs scalaire de t_i. Puis nous remplaçons cette valeur t_i dans l'équation (*III.10*) du rayon de visualisation pour déterminer la position de sortie O_s.

2. Entre le rayon de lumière et le volume (ombre/volume)

Pour le rayon de lumière défini par le vecteur $\vec{V}_l$ nous avons besoin uniquement de déterminer la position d'entrée O_{le} de ce rayon dans le volume. Pour cela, on procède avec la même logique que pour le rayon de visualisation, ce qui revient à déterminer *t* à partir de l'équation:

$$\vec{O}_{le}(t) = \vec{O}_l + t \cdot \vec{V}_l \quad ; \ avec\ t \geq 0 \tag{III.14}$$

$$(\vec{O}_{le}(t) - \vec{O}_p) \cdot \vec{N} = 0 \tag{III.15}$$

D'où : $$t = (\vec{O}_p - \vec{O}_l) \cdot \vec{N} / \vec{V}_l \cdot \vec{N} \quad \text{(III.16)}$$

D'où les valeurs de *t* sont les suivantes :

$$\left.\begin{array}{ll} \text{Plan de normale } \vec{N}(1 \quad 0 \quad 0) \text{ est :} & t_1 = (1-x)/\,V_l \\ \text{Plan de normale } \vec{N}(0 \quad 1 \quad 0) \text{ est :} & t_2 = (1-y)/\,V_l \\ \text{Plan de normale } \vec{N}(0 \quad 0 \quad 1) \text{ est :} & t_3 = (1-z)/\,V_l \\ \text{Plan de normale } \vec{N}(-1 \quad 0 \quad 0) \text{ est :} & t_4 = (-x)/\,V_l \\ \text{Plan de normale } \vec{N}(0 \quad -1 \quad 0) \text{ est :} & t_5 = (-y)/\,V_l \\ \text{Plan de normale } \vec{N}(0 \quad 0 \quad -1) \text{ est :} & t_6 = (-z)/\,V_l \end{array}\right\} \quad \text{(III.17)}$$

V_l , est la valeur scalaire du vecteur $\vec{V}_l$.

Il suffit de remplacer la valeur minimale de t_i dans l'équation (III.14) du rayon de lumière pour déterminer la position d'entrée O_{le}.

Puis, le calcul des intersections permet de déterminer la portion du segment contenu dans le volume pour effectuer les prélèvements des échantillons entrant dans la composition de couleur le long du segment, ce qui permet de calculer le pas d'échantillonnage.

Les étapes d'implémentation pour le calcul d'intersection est le suivant :

- Récuperation de la position courante du voxel
- Calcul de la direction de visualisation
- Calcul de la direction de la lumière
- Prendre la position courante comme le première point dintersection du vecteur de visualisation avec le volume
- Calcul le second point d'intersection avec les faccettes du quad
- Calcul du premier point d'intersection du vecteur de lumière avec le volume
- Calcul du segment du vecteur de visualisation contenue dans le volume
- Calcul du segment du vecteur de lumière contenue dans le volume

III.6. Application du modèle de Blinn-Phong

Nous considérons une source de lumière ponctuelle homogène fixée à une certaine distance. Les faisceaux de lumière émis sont assez fins pour être assimilés à une droite. Pour chaque voxel du volume nous appliquons le modèle de Blinn-Phong pour déterminer la couleur de celui-ci. Nous ne considérons que la lumière réfléchie, réflexion diffuse et réflexion spéculaire, sans transmission indirecte et sans émission propre (*Figure III.7*).

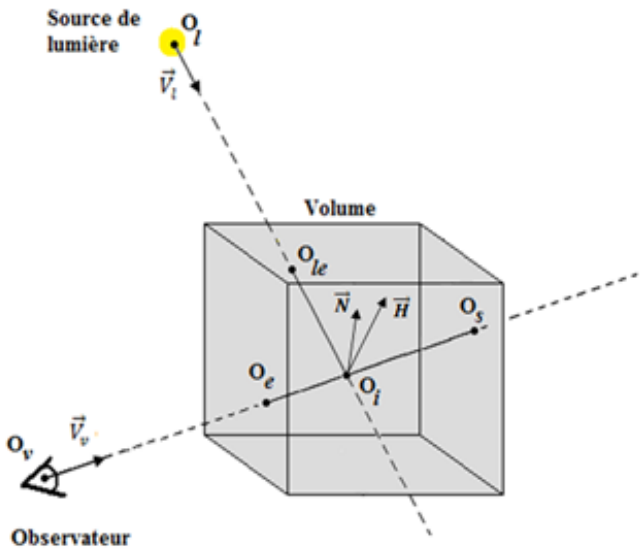


***Figure III.7**: Application du modèle de Blinn-Phong*

Nous rappelons que le modèle de Phong est la combinaison linéaire de trois termes, l'intensité ambiante, l'intensité diffuse et l'intensité spéculaire.

$$I_{Phong} = I_{ambient} + I_{diffuse} + I_{specular} \qquad \text{(III.18)}$$

Avec : $I_{ambient} = k_a = const.$

$I_{diffuse} = I_\rho \cdot k_d \cdot \cos\varphi = I_\rho \cdot k_d \cdot (\vec{l} \cdot \vec{n})$

$I_{specular} = I_\rho \cdot k_s \cdot cos^n\alpha = I_\rho \cdot k_s \cdot (\vec{h} \cdot \vec{n})^n$

Où : I_ρ , est l'intensité émise par la source.

k_a, k_d, k_s , caractérisent les propriétés ambiant, diffus, et spéculaire du matériau.

$\vec{l}$, est la direction de la source.

$\vec{n}$, est la normal au point de la surface considéré.

$\vec{h}$, est la bissectrice entre la direction de la source $\vec{l}$ et de direction de la lumière réfléchie $\vec{r}$.

φ , est l'angle d'incidence entre la source de lumière $\vec{l}$ et la normal $\vec{n}$.

α , est l'angle entre la direction de la source $\vec{l}$ et de la bissectrice $\vec{h}$.

n, est la brillance (shininess).

L'intensité spéculaire déterminée par le modèle de Blinn-Phong s'effectue sur les trois canaux de couleur RVB, d'où l'utilisation des trois composantes RVB pour les intensités $I_{ambiant}$, $I_{diffuse}$ et $I_{spéculaire}$.

Comme le calcul du modèle d'illumination s'effectue au niveau du fragment shader en récupérant les valeurs de la normale locale à partir de la texture transmise de l'application, cette normale doit être corrigée et ramener entre 0 et 1, car les valeurs de la texture sont comprise entre -1 et +1.

L'implémentation du modèle d'illumination Blinn-Phong, au niveau du fragment shader est le suivant :

- Récupération de la position courante du voxel
- Récupération de l'orientation du vecteur de lumière
- Récupération de l'orientation du vecteur de visualisation
- Récupération de la normale à la position courante duvoxel
- Normalisation des vecteurs
- Calcul du vecteur bisecteur
- Calcul de la composante diffuse du modèle de Blinn-Phong
- Calcul de la composante spéculaire du modèle de Blinn-Phong

III.7. Calcul de la Composition

Après avoir déterminé les positions O_e et O_s d'intersection du rayon de visualisation $\vec{V}_{View}$ avec le volume de données, nous déterminons le pas d'échantillonnage avec lequel nous effectuons le calcul de composition le long du segment $\overline{O_e O_s}$.

Dans un premier temps, nous calculons pour chaque position d'échantillonnage considéré O_i l'atténuation que subit la source de lumière le long du segment t $\overline{O_{le} O_l}$, qui est dû uniquement à la densité du volume, selon l'équation *(III.1)*. Nous appliquons le modèle d'illumination de Blinn-Phong, équation *(III.18)*, en tenant compte de l'atténuation de la lumière et nous effectuons la composition de couleur front-to-back selon l'équation *(I.12)*. L'implémentation de la composition de couleur le long du segment $\overline{O_e O_s}$ est réalisée ainsi :

- Récupération du segment du vecteur de visualisation contenu dans le volume
- Détermination du pas d'échantillonnage suivant le vecteur de visualisation
- Récupération la valeur de la couleur et de l'opacité du volume de chaque échantillon
- Appliquation à chaque échantillon la fonction de tranfert
- Calcul de l'atténuation de la lumière
- Appliquation à chaque échantillon le modèle de Blinn-Phon
- Accumulation de la couleur et de l'opacité selon la formule du front-to back
- Transmettre le résultat au frame buffer

L'implémentation de l'atténuation de la source de lumière le long du segment $\overline{O_{le} O_l}$ est le suivant :

- Récupération de la position courante du voxel
- Récupération de la direction du vecteur de lumière
- Récupération du segment du vecteur de lumière contenue dans le volume
- Calcul du pas d'échantillonnage
- Récupération de la valeur de l'opacité
- Accumulation de l'opacité

A ce stade, la couleur calculé par le fragment processeur est la combinaison de la couleur issue de la lumière et des voxels le long d'un seul rayon. Cette couleur correspond à

la couleur réel du pixel perçu par l'observateur. L'ensemble de ces étapes sont réalisé sur chaque rayon lancé. Au final le fragment processeur transmet les couleurs de chaque pixel (fragment) au frame buffer.

Conclusion :

Il apparait à travers notre mise en œuvre que le processus est assez complexe et long à implémenter. Différentes hypothèses sont à prendre en considérations, différentes notions à maitriser et plusieurs étapes à implémenter aussi bien au niveau du CPU que du GPU. Notre approche permet de réaliser les calculs les plus complexes au niveau du GPU.

Le CPU se charge de lire les données volumique, de calculer le gradient et la fonction de transfert. Toutes ces données sont transmises au GPU sous forme de texture 3D ou 2D, en plus de la position de l'observateur et de la source de lumière.

Au niveau du GPU, le vertex processeur transmet les positions des sommets au fragment processeur. Le fragment processeur quand à lui se charge d'effectuer les différents calculs de la géométrie analytique, d'appliquer le modèle d'illumination avec le calcul d'ombre et de calculer la contribution des couleurs des voxels, le long du rayon de visualisation, sur la couleur du fragment final à afficher. La couleur du fragment est ensuite envoyée au frame buffer pour être affichée.

Dans le chapitre suivant nous donnons les différents résultats de notre implémentions basée GPU.

CHAPITRE IV
RESULTATS ET INTERPRETATIONS

Ce chapitre regroupe les différents résultats obtenus pour différentes données volumique (3D) en considérant plusieurs paramètres. Il est composé de deux parties, la première regroupe les résultats obtenus avec l'implémentation de l'algorithme du ray casting sans tenir compte du calcul d'ombre (le shadow) et la seconde regroupe les résultats obtenus en tenant compte du calcul d'ombre. Nous comparons également nos résultats avec ceux obtenus avec le ray casting basé CPU que nous avons implémenté dans le cadre d'un co-encadrement de mémoire de fin d'études.

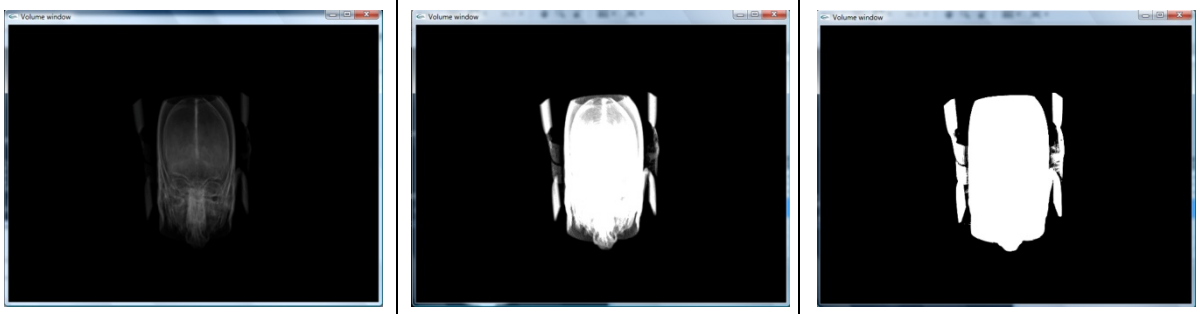
Les données 3D qu'on utilise sont des données issues de scan CT ou MRI. Ce sont des données binaires dont chaque voxel est codé sur 8 bits à une échelle de 1. Ces données sont stockées sous le format *RAW*. Les résultats de notre implémentation sont obtenus à l'aide d'une carte graphique Geforce 9600(M) GT avec une mémoire de 512Mo et une fréquence de 400 MHz.

IV.1. Résultats sans calcul d'ombrage

Dans cette partie nous allons mettre en évidence l'apport visuel lors de l'utilisation d'un modèle d'illumination, modèle de Blinn-Phong, *(équation III.18)* ; Nous supposons dans ce cas, que le volume diffuse de la lumière dans toutes les directions et de façon homogène, et on évalue la composition de couleur d'avant en arrière (front-to-back), *(équation I.12)*. Nous calculons le nombre de frames par seconde (FPS) qui correspond au temps d'affichage de la carte graphique dans chaque cas. Nous considérerons dans cette partie que la source de lumière ne subit aucune atténuation lorsqu'elle traverse le volume.

a) Sans illumination :

Dans ce cas (figure *IV.1*), nous varions le facteur de densité « *d* » et nous fixons les pas d'échantillonnage « ΔR_{view} » le long du rayon de visualisation, le calcul du pas d'échantillonnage étant décrit au paragraphe (*III.4.*).

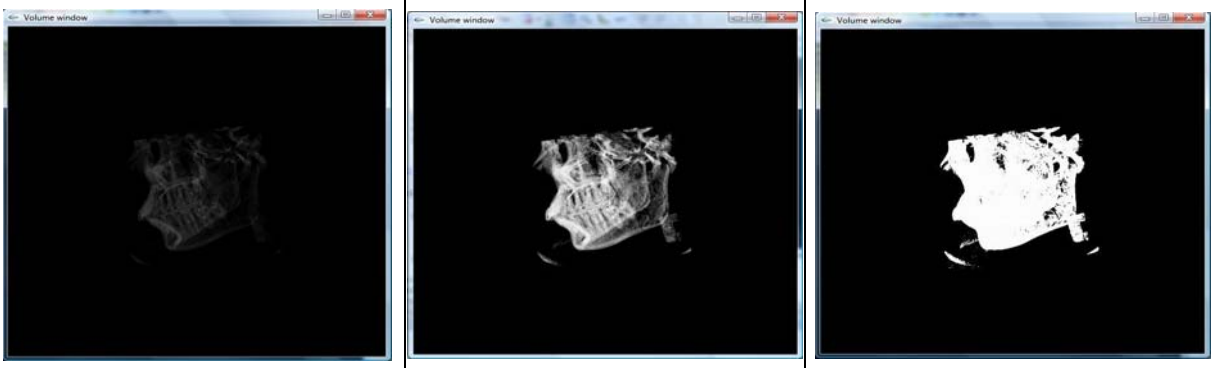


(a) *d=0.01, ΔR_{view}= 0.01, FPS =39.*

(b) *d=0.1, ΔR_{view}= 0.01, FPS =42.*

(c) *d =1, ΔR_{view}= 0.01, FPS =61.*

Figure IV.1 : *Rendu sans illumination pour différents facteur de densité d, (CT de la taille 128×128×112).*



(d) *d=0.01, ΔR_{view}= 0.01, FPS =17.3.*

(e) *d=0.1, ΔR_{view}= 0.01, FPS =28.*

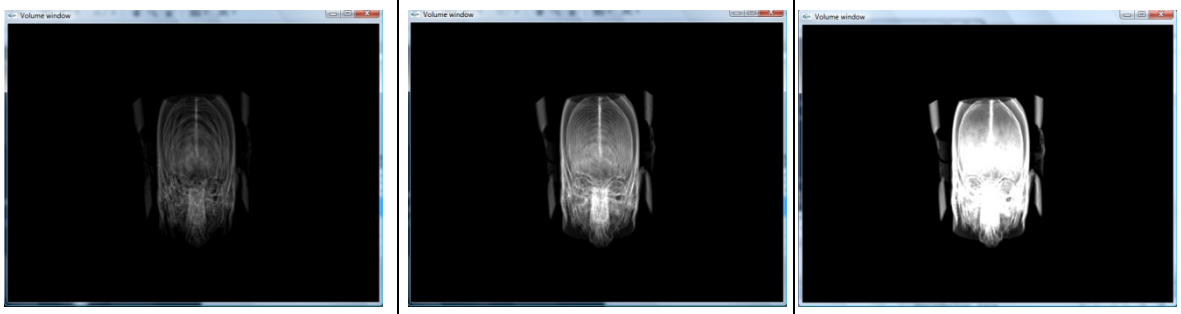
(f) *d =1, ΔR_{view}= 0.01, FPS =30.8.*

Figure IV. 2: *Rendu sans illumination pour différents facteur de densité d, (CT de la taille 256×256×256).*

Dans le cas de l'image *IV.1(a)* et l'image *IV.2(a)* avec un facteur de densité *d =0.01* le volume est transparent, ce qui permet de voir l'intérieur du volume. Dans l'image *IV.1(b)* et l'image *IV.2(b)* où *d=0.1* le volume est plus au moins transparent et pour l'image *IV.1(c)* et l'image *IV.2(c)* avec un facteur *d=1* le volume est totalement opaque et ne permet pas de voir le contenu du volume.

A travers ces premiers tests, nous voyons, comment pour un pas d'échantillonnage ΔR_{view} fixe (*0.01*), le facteur de densité permet de faire varier la transparence du volume.

Nous allons à présent fixer la densité d, correspondant à un milieu translucide et varier le pas d'échantillonnage « ΔR_{view} » le long du rayon de visualisation (*Figure IV.3* et *Figure IV 4*).

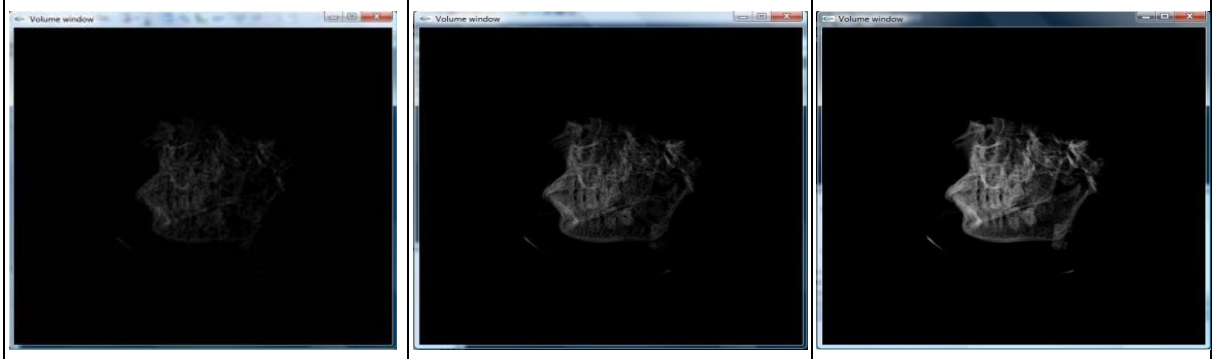


(a) *d=0.05, ΔR_{view}= 0.045, FPS =61.*

(b) *d=0.05, ΔR_{view} = 0.02, FPS =61.*

(c) *d=0.05, ΔR_{view} = 0.01, FPS =40.*

Figure IV.3 : *Rendu sans illumination pour différents pas d'échantillonnage le long du rayon de visualisation, (CT de la taille 128×128×112).*



(a) *d=0.05, ΔR_{view}= 0.045, FPS =58.*

(b) *d=0.05, ΔR_{view} = 0.02, FPS =30.*

(c) *d=0.05, ΔR_{view} = 0.01, FPS =17.*

Figure IV.4 : *Rendu sans illumination pour différents pas d'échantillonnage le long du rayon de visualisation, (CT de la taille 256×256×256).*

Dans le cas de l'image *IV.3(a)* et de l'image *IV.4(a)* avec un pas d'échantillonnage ΔR_{view}*=0.045* le volume est transparent mais des artéfacts apparaissent qui sont dus au nombre insuffisant d'échantillons entrant dans l'accumulation de couleur. Pour le cas de l'image *IV.3(b)* et de l'image *IV.4(b),* avec ΔR_{view}*= 0.02,* le volume est transparent et ne contient pas d'artéfact. Dans ce cas, le nombre d'échantillons est bien adapté pour cette densité, Pour l'image *IV.3(c)* et l'image *IV.4(c)* avec ΔR_{view}*= 0.01* le volume est plus au moins opaque, ceci est dû à un nombre trop élevé d'échantillons entrant dans l'accumulation de couleur.

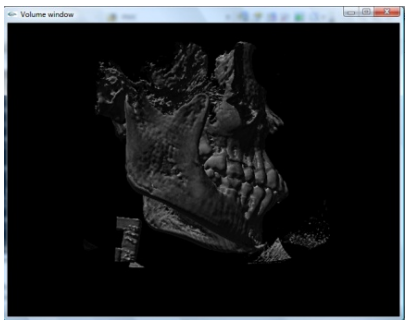
Pour un facteur de densité fixe correspondant à un milieu transparent *d= 0.05* et suivant le pas d'échantillonnage ΔR_{view}, nous obtenons un rendu plus ou moins opaque, ce qui fausse la visualisation des données.

Pour visualiser correctement les données il faut donc choisir un pas d'échantillonnage adapté à la densité utilisée.

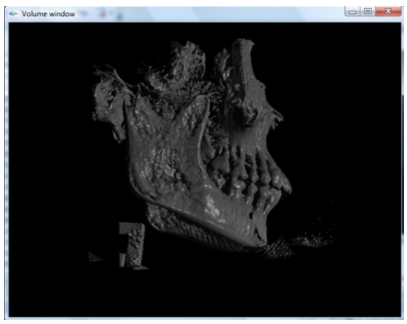
b) Avec illumination :

Nous utilisons une source de lumière blanche avec une intensité moyenne située à une position Pos_{Linght}. Nous évaluons la quantité de lumière émise par notre objet et que recevra l'observateur selon l'équation *(III.18)*. Le modèle de Blinn-Phong est réduit aux deux termes diffus I_{difuse}et spéculaire $I_{spéculaire}$, dont l'algorithme implanté dans le *fragment shader* est détaillé au *paragraphe III.5*. Nous calculons la composition de la couleur toujours d'avant en arrière.

Nous fixons l'orientation de la source dans la même direction de visualisation (*Figure IV.5*) et nous fixons le pas d'échantillonnage (ΔR_{view}*= 0.02*) et le facteur de densité (*d=1*). L'image *IV.5(a)* est donnée en tenant compte uniquement du terme diffus et l'image *IV.5(b)* est donnée en tenant compte aussi bien du terme diffus que du terme spéculaire.

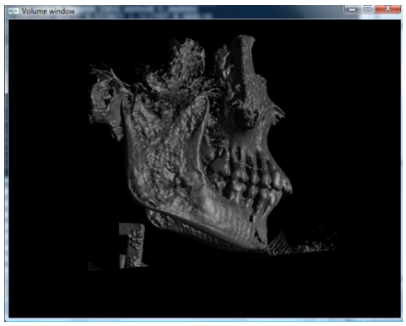


(a) *Avec uniquement terme diffuse*

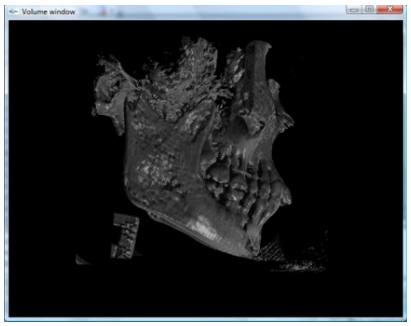


(b) *Avec le terme diffus et spéculaire*

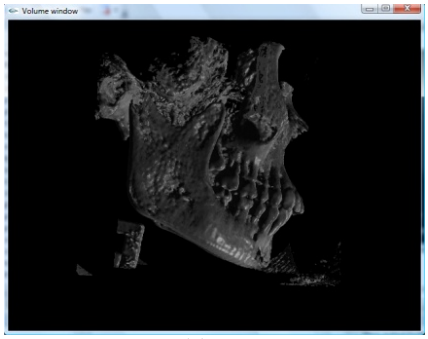
Figure IV.5 : *Résultats avec application du modèle de Blinn-Phong, (CT de la taille 256×256×256).*



(a) n=24.



(b) n=64.



(c) n=82.

Figure IV.6 : *Résultats pour différente valeur la brillance n. (CT de la taille 256×256×256).*

Nous constatons en comparant les images *IV.5(a)* et *IV.5(b)* que le volume est bien visible, que l'on distingue bien les caractéristiques des surfaces et que l'ajout du terme spéculaire donne un aspect plus net et lisse à la surface.

Les résultats de la figure *(IV.6)* sont donnés avec les mêmes considérations que pour l'image *IV.5(b)*, Le modèle de Blinn-Phong est calculé en tenant compte des deux termes diffus et spéculaire et en faisant varier le facteur de brillance *n* (*shininess*) du terme spéculaire *(paragraphe III.5)*.

Dans le cas de l'image *IV.6(a)* on constate que la surface brillante occupe une plus grande surface par rapport à l'image *IV.6(b)* et *IV.6(c)*. Dans ce dernier cas ou n=82 la surface de brillance est très concentrée.

Nous varions la source de lumière :

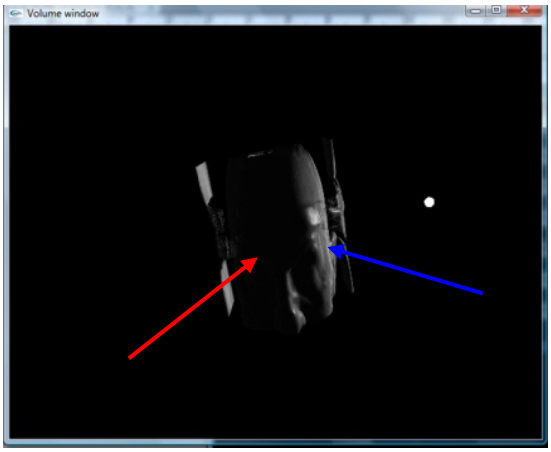


Figure IV.7 : *Résultat avec illumination droite.*

Nous voyons (*Figure IV.7*) qu'en, variant uniquement la position de la source de lumière, les surfaces orientées directement à la source de lumière sont correctement éclairées (*indiqué par la flèche bleue*) et que les surfaces opposées sont ombragées (*indiqué par la flèche rouge*).

Nous varions l'angle de vue :

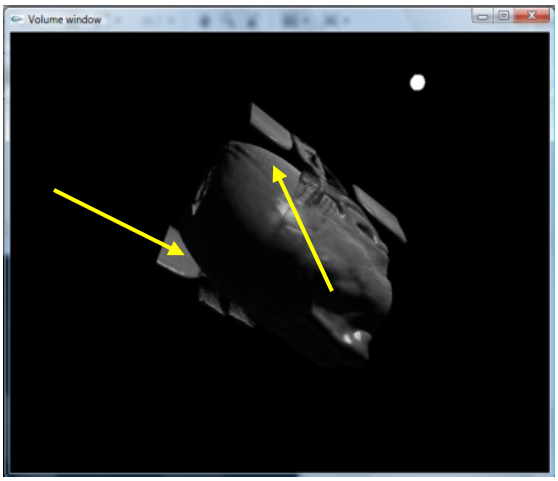
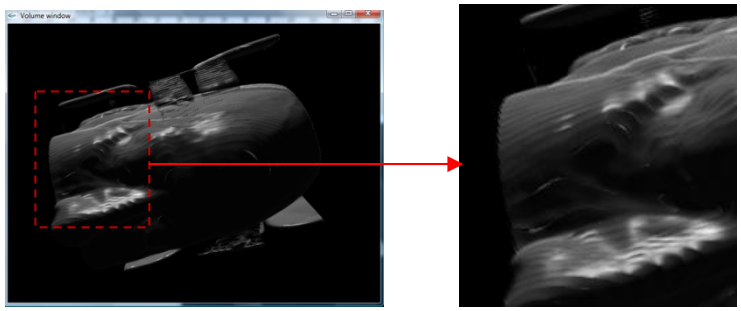


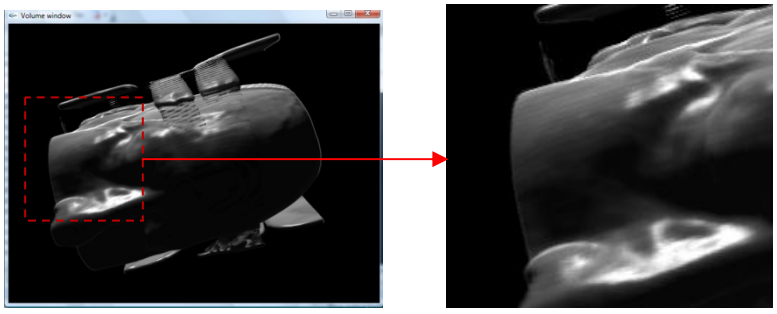
Figure IV.8 : *Résultats obtenus en variant l'angle de vue.*

En variant l'angle de vue, (*Figure IV.8*) nous constatons que les surfaces orientées vers la source de lumière mais qui se trouvent derrière des surface qui font obstacle à la lumière sont également éclairées (indiqué avec les flèches jaunes). Donc nous avons les ombres directs mais pas les ombres portées.

Nous varions le pas d'échantillonnage :



(a) $\Delta R_{view} = 0.0122$, *FPS =31.*



(b) $\Delta R_{view} = 0{,}0038$, *FPS=12.*

Figure IV.9 : *Résultats pour différents pas d'échantillonnage* ΔR_{view}.

Les résultats pour différentes valeurs de ΔR_{view} (*Figure IV.9*), nous montrent clairement en image *IV.9(a)* les artéfacts qui sont dus à un nombre insuffisant d'échantillons dans le calcul de la composition de couleur et un temps de rendu de 31 FPS. Alors que dans l'image *IV.9(b)* la qualité de l'image est nettement meilleure et la surface est bien lisse mais avec un temps de rendu plus élevé (12 FPS).

Nous avons vu à ce stade l'effet de l'utilisation d'une source lumineuse, qui donne une meilleure appréciation des détails des surfaces. Le temps de rendu est fortement lié au pas d'échantillonnage considéré. Toutefois pour avoir une visualisation de meilleure qualité et, dans le cas de zones cachées par d'autres, il faut prendre en compte les ombres portées.

IV.2. Résultats avec calcul d'ombrage

Nous considérons dans cette partie que la source de lumière subit une atténuation lorsqu'elle traverse le volume qui est calculé pars l'équation *(III.1)*. Ce qui revient donc à faire une sommation de l'opacité le long du vecteur de lumière pour un certain nombre d'échantillons.

A présent nous varions à la fois l'angle de vue et la disposition de la source de lumière (*Figure IV.10*). Les images *IV.10(a)* et *IV.10(b)* sont les résultats obtenus sans calcul d'ombrage. Les images *IV.10 (a.1)* et *IV.10 (b.1)* sont les résultats obtenus en considérant le calcul d'ombrage. Les images *IV.10 (a.2)* et *IV.10 (b.2)* sont un zoom de la zone encadrée en rouge.

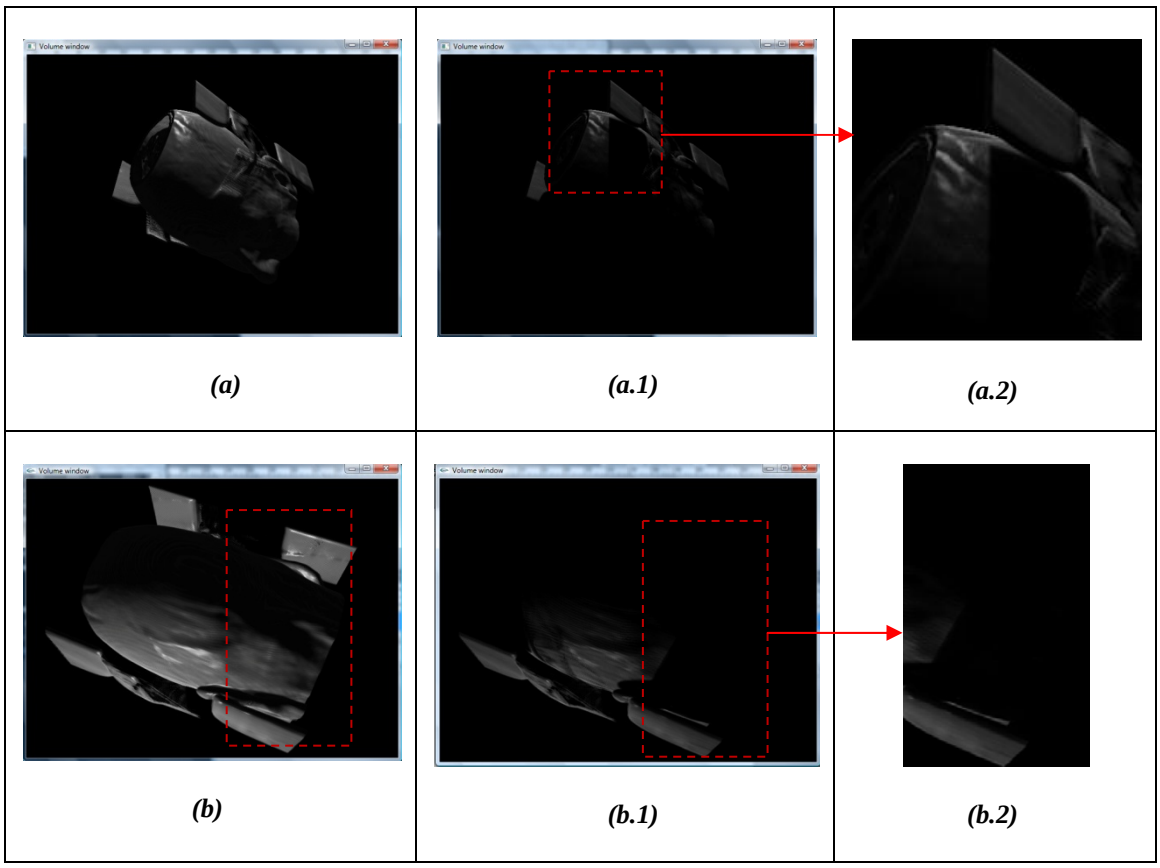
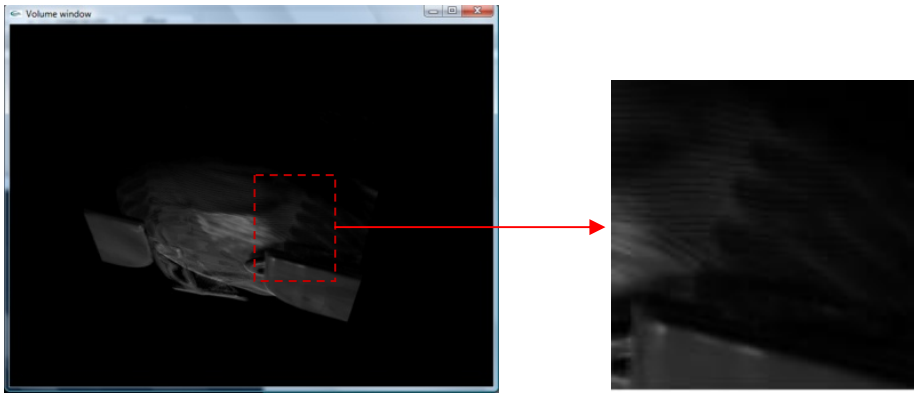


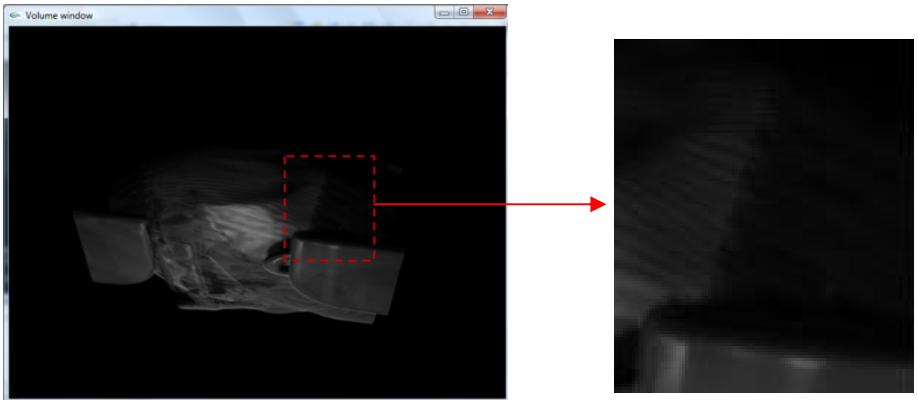
Figure IV.10 : *Résultats avec modèle de Blinn-Phong et calcul d'ombrage avec une image de zoom de la zone d'ombre (CT de la taille 128×128×112).*

Comme nous le constatons à travers les images *IV.10 (a.1)* et *IV.10 (b.1)*, les ombres portées sont bien prises en compte avec l'ajout de l'atténuation de la source de lumière en pénétrant le volume. Nous pouvons voir sur les images de zoom *IV.10 (a.2)* et *IV.10 (b.2)* que l'ombre s'étale correctement selon l'angle d'incidence de la source de lumière.

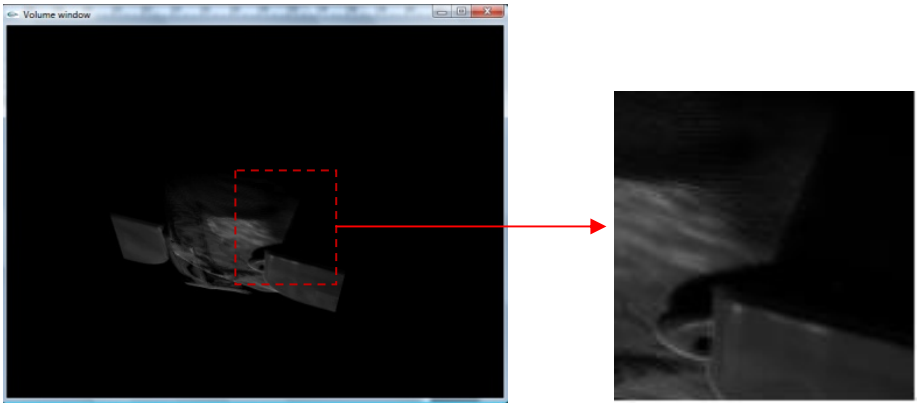
Nous varions le pas d'échantillonnage le long du rayon de lumière. Les résultats sont donnés pour différents pas d'échantillonnages ΔR_{shadow} (*Figure IV.11*).



(a) $\Delta R_{shadow} = 0.034$, *FPS=3.*



(b) $\Delta R_{shadow} = 0.023$, *FPS=2.*



(c) $\Delta R_{shadow} = 0.01$, *FPS=2.*

Figure IV.11 : *Influence du pas d'échantillonnage le long du rayon d'ombre (CT de la taille 128×128×112).*

Dans l'image *IV.11(a)* pour un pas d'échantillonnage ΔR_{shadow}=0.034, des artéfacts se voient clairement sur la zone d'ombre.

Dans l'image *IV.11(b)* pour un ΔR_{shadow}=0.023 l'ombre devient de plus en plus homogène alors que dans l'image *IV.11(c)* pour un ΔR_{shadow}=0.01 l'ombre est plus douce avec des contours bien lisses.

Nous constatons que l'ombre devient de plus en plus douce quand le pas d'échantillonnage est élevé (ΔR_{shadow} petit) et que pour des pas d'échantillonnage faible les artéfacts dus à cet échantillonnage deviennent de plus en plus visibles et réduisent la qualité du rendu.

IV.3. Temps d'exécution du ray- casting basé CPU :

Dans le cadre du co-encadrement d'un mémoire de fin d'études, nous avons pu implémenter une version de base du ray casting sans modèle d'illumination, sur CPU. Les résultats de cette implémentation, en termes de temps, sont donnés dans le tableau *IV.01*, en fonction du nombre de coupes utilisé. Notons que nous donnons ces résultats pour avoir une approximation sans l'utilisation des cartes graphiques programmables.

Nombre de coupes	20	30	60
Temps d'exécution (ms)	4025	4477	5351

Tableau IV.01 : *Temps de calcul du ray-casting basé CPU en fonction de nombre de coupes.*

Conclusion :

A travers l'illustration de ces résultats, nous constatons que les temps de rendu de nos différentes implémentations se situent entre 30 FPS à 10 FPS sans calcul d'ombre et que ces temps diminuent fortement dans le cas où le calcul d'ombre est prit en considération. Cette diminution est due au nombre important d'itérations qui se fait lors du calcul de la composition de couleur auquel s'ajoute le nombre d'itérations lors du calcul de l'atténuation de la lumière. Toutefois nous obtenons une interactivité assez bonne mais au détriment de la qualité du rendu. Certes ces temps sont plus intéressants que si l'algorithme était pris en charge par le CPU uniquement, mais il nous faut chercher une approche qui nous permette de diminuer ces temps surtout dans le cas ou l'on tient compte des ombres portées. L'ajout du modèle d'illumination local au ray-casting, précisément le modèle de Blinn-Phong, donne un aspect visuel réaliste ce qui permet de distinguer les reliefs et les points d'ombre et reproduit correctement les phénomènes de réflexion et de diffusion de la lumière. De plus, pour garantir une interactivité et une qualité de rendu des plus ou moins réaliste, il faut définir le nombre d'échantillons à utiliser aussi bien dans le calcul de la composition que dans le calcul d'ombre.

CONCLUSION ET PERSPECTIVES

Dans ce travail nous avons développé un outil de visualisation sur une plateforme PC-Windows, permettant la visualisation volumique d'images médicales dans un but de diagnostic complémentaire mais aussi d'enseignement. Notre application permet d'obtenir des rendus volumiques de très bonne qualité, testés sur plusieurs images correspondants à diverses régions du corps humain, grâce à l'utilisation et la programmation des GPU. Bien que la programmation sur GPU soit assez particulière à appréhender car différente de la programmation classique, les résultats obtenus en terme de qualité de rendu et de temps sont très satisfaisants.

Après une étude de l'état de l'art sur le rendu volumique, nous avons dégagé une approche de rendu volumique basée sur le ray-casting. Dans un premier temps, nous avons défini les avantages et les particularités de l'implémentation du ray-casting sur GPU. Nous avons pu voir que les phases de traitement au niveau de la carte graphique s'effectuent en plusieurs étapes définissant le pipeline graphique. Trois grandes phases de traitement se distinguent clairement : les opérations sur les sommets, les opérations sur les fragments et la composition de la couleur. L'accessibilité du pipeline graphique à la programmation est réalisée grâce au vertex processeur et au fragment processeur qui se situent respectivement au niveau des deux premières phases de traitement. Ces processeurs prennent en charge l'exécution des shaders qui leur sont spécifiques, le vertex shader et le fragment shader.

Il nous a donc fallu apprendre à programmer les cartes graphiques avec un langage de programmation dédié au shader, nous avons choisi le GLSL. Nous avons alors proposé une approche de ray-casting où les différents calculs de la géométrie analytique sont réalisés au niveau des *shader*, c'est-à-dire que la représentation des vecteurs et des plans ainsi que leurs intersections sont calculés directement par le *fragment shader*.

Nous avons implémenté notre algorithme en deux phases : une phase dans laquelle un certain nombre de calcul s'exécute au niveau de l'application par le CPU et une autre phase dans laquelle l'ensemble des calculs s'effectue au niveau des shaders par le GPU. Les calculs sur le CPU sont la lecture des données volumiques à partir d'un fichier, la définition des positions de la lumière et de l'observateur et le calcul de la valeur du gradient local. Les calculs sur le GPU sont les calculs de la géométrie analytique, la définition des directions de chaque vecteur de visualisation et de lumière ainsi que le calcul d'intersection de ces vecteurs avec le volume de données, l'application du modèle d'illumination de Blinn-Phong et la composition de couleur.

La considération du ray-casting avec application d'un modèle d'illumination reproduit correctement les phénomènes de réflexion et de diffusion de la lumière. Cet algorithme permet d'éclairer les surfaces directement exposées à la source de lumière et d'ombrager les autres surfaces, mais ne tient pas compte des ombres portées des zones d'occlusion. Le ray casting à lui seul est très avantageux pour des volumes ne comprenant pas des surfaces d'occlusion, mais nécessite d'introduire le calcul d'ombre pour des volumes avec des surfaces d'occlusion.

Notre approche avec la considération du calcul d'ombre portée permet de corriger ce manque. Le pas d'échantillonnage dans les deux directions de visualisation et de la source de lumière influe de manière considérable sur la qualité et la vitesse du rendu, d'où la nécessité d'effectuer un compromis entre le pas d'échantillonnage et le temps de rendu pour avoir une visualisation acceptable.

Notre ray-casting sans le calcul d'ombrage donne des temps de rendu interactifs qui peuvent atteindre 30 FPS (frame par second) pour des volumes de données de 8 bits de la taille de 256x256x256,et des temps de rendu interactifs qui peuvent atteindre 10 FPS pour des volumes de données de 8 bits de la taille 256x256x256 en considérant le calcul d'ombrage. Ce qui donne à notre approche une interactivité et une qualité de rendu assez bonne, bien que cette interactivité soit réduite lorsque nous tenons compte du calcul d'ombrage.

Les temps et la qualité du rendu sont étroitement liés au nombre de pas d'échantillonnage qui entre dans le calcul de la composition de la couleur. Plus le nombre est élevé plus la qualité est meilleure et plus le temps de rendu est lent. Inversement plus le pas d'échantillonnage est faible plus le rendu est élevé et moins la qualité de rendu est bonne. Dans le calcul d'ombrage, le nombre d'échantillons entrant dans le calcul de l'atténuation de la source de lumière affecte grandement le temps de rendu et la qualité visuelle du rendu d'où la nécessité de prendre un compromis entre les pas d'échantillonnage et la qualité du rendu pour obtenir une meilleure interactivité.

Comme perspectives, ce travail peut se généraliser à d'autres types de données médicales, mais également à d'autres domaines, voire à des scènes animées : il existe de nombreux axes pour augmenter la qualité et l'efficacité de la méthode proposée.

Il serait aussi intéressant de reprendre cet algorithme sur une version plus récente de shader (version 3.0) qui intègre la *geometry shader* et avec la nouvelle API CUDA qui promet beaucoup de performance avec le contrôle de la répartition des calculs.

De plus, il faudrait s'inspirer des résultats obtenus dans ce domaine par l'équipe BUNRAKU du laboratoire de recherche de l'Irisa de Rennes où j'ai effectué une grande partie de mes travaux grâce a la collaboration qui existe depuis des années avec l'équipe ParIMéd du laboratoire LRPE. Par exemple voir les diverses évolutions de l'illumination de scènes tridimensionnelles en synthèse d'image qui ont permis d'augmenter le sentiment d'immersion dans les scènes virtuelles grâce à un plus grand réalisme de celles-ci. Afin de calculer cette illumination, plusieurs algorithmes ont été proposés parmi lesquels la radiosité.

Finalement, afin de créer une plateforme médicale interactive qui regroupera l'ensemble des logiciels de rendu volumique réalisés au sein de l'équipe ParIMéd à savoir le Ray-Casting, le Shear-Warp, le MIP, le MPR, le Splatting, le ShearWarp-Splatting, il faudrait étudier la faisabilité de traitements en parallèle avec des cartes graphiques hybride SLI.

ANNEXE A
« Généralités sur OpenGL »

Le monopole des bibliothèques de développement 3D est détenu par Direct3D et OpenGL. Direct3D est une API (Application Programming Interface) développée par Microsoft pour son OS Windows. Propriétaire et non portable, Direct3D fait partie de l'ensemble DirectX et est relativement orienté "Jeux vidéo". Son principal avantage est d'être bien supporté par les constructeurs de cartes graphiques grand public. Pour sa part, OpenGL (Open Graphic Library), ce qui se traduit en bon français par Bibliothèque Graphique Ouverte, est une API développée depuis 1992 par Silicon Graphics (http://www.sgi.com), un des grands ténors de l'informatique graphique professionnelle.

Originellement développé pour les stations graphiques haut de gamme Silicon Graphics, l'aspect ouvert d'OpenGL lui a valu d'être porté sur la plupart des plateformes modernes, et sur la plupart des systèmes d'exploitation. Aujourd'hui, OpenGL est reconnu par les professionnels comme un standard, et est utilisé par tous les logiciels professionnels de synthèse d'images (3DSMax, Maya, Softimage...), de CAO (ProEngineer, Catia...), mais aussi dans le domaine des jeux vidéos 3D (Quake (1,2,3), Unreal...).

1. Fonctionnalité d'OpenGL

OpenGL permet de représenter à l'écran des scènes tridimensionnelles réalistes. Le terme "réaliste" est ici tout à fait relatif : on peut considérer comme réaliste une image de synthèse où tous les objets représentés sont correctement placés au niveau de la perspective sans que les couleurs correspondent à ce qu'on observe dans la nature. Le réalisme ultime peut être atteint lorsqu'il est impossible, à première vue, de savoir si une image est issue d'un calcul ou si elle est simplement une photo numérisée, on parle alors de photoréalisme. Un programme conçu avec OpenGL permet en général à l'utilisateur d'agir par l'intermédiaire de périphériques d'entrée (clavier, souris, trackball...) sur la scène (déplacement de caméra, changement de focale, déplacement des objets....), et d'en visualiser immédiatement l'effet.

Bien que non orienté photoréalisme, OpenGL permet d'obtenir des rendus tout à fait satisfaisants. Voici une liste non exhaustive des principales fonctionnalités d'OpenGL.

– Affichage de maillages polygonaux.
– Rendu par projection orthogonale ou perspective.
– Placage de texture.
– Simulation d'éclairages réalistes.
– Ombre portées.
– Insertion d'objets bitmaps en avant plan (images, polices).
– Effet de brouillard, de flou.
– Gestion des courbes et surfaces gauches (Splines, Nurbs...)

1.2. Cartes graphiques et OpenGL

Afin d'accélérer les temps de calcul, on fait souvent appel à une accélération matérielle. Sur les ordinateurs actuels, cette accélération matérielle est prise en charge par les cartes graphique. Les principales opérations de calcul sont prises en charges par des circuits intégrés situés sur la carte, qui dispose également d'espaces de stockage dédiés au calcul et au stockage des données nécessaires à la création de l'image. Ainsi, on "soulage" le processeur de la machine "CPU". Ce dernier n'a plus qu'à transmettre les informations à la carte 3D qui s'occupe de générer l'image et de l'afficher.

Autre avantage, le calcul se fait bien plus rapidement sur la carte en "hardware", que par le processeur "en software", puisque la carte est architecturée spécifiquement pour ces calculs. Pour pouvoir bénéficier de l'accélération matérielle, il faut disposer de pilotes "dirvers" c'est-à-dire une implémentation OpenGL.

1.3. Glut

Pour créer un programme OpenGL dans un environnement fenêtré comme Windows ou XWindow, il faut être en mesure de créer une fenêtre pour y afficher nos images, de la détruire, et de gérer les interactions avec l'utilisateur par le clavier, la souris et tous les autres types de périphériques d'entrée. OpenGL se voulant indépendant de toute plate-forme matérielle, l'API ne fournit pas de telles fonctions. En revanche , une bibliothèque annexe a été créée, nommée Glut (GL Utility Toolkit) qui fournit ces fonctions. Elle est développée par Mark Kilgard et est disponible sur le site http://reality.sgi.com/mjk/glut3.

Voici une liste de toutes les fonctionnalités proposées par Glut :

– Gestion de fenêtres.
– Gestion des évènements par fonctions de rappel.
– Gestion de périphériques d'entrée exotiques (spaceballs...).
– Gestion des polices de caractères.
– Fonctions de création de menus.

1.4. Notions en OpenGL

1.4.1. "caméra virtuel"

Tel un modeleur 3D, nous utiliserons en OpenGL le concept de caméra virtuelle, n'est pas un objet de la scène 3D, permet d'indiquer le point de vue de l'observateur, la direction du regard et l'inclinaison de la caméra. le point de vue par défaut d'OpenGL est : la caméra est située sur l'axe Z, pointé vers l'origine du repère (le point O), et la verticale de la caméra correspond à l'axe Y du repère *(figure 01)*.

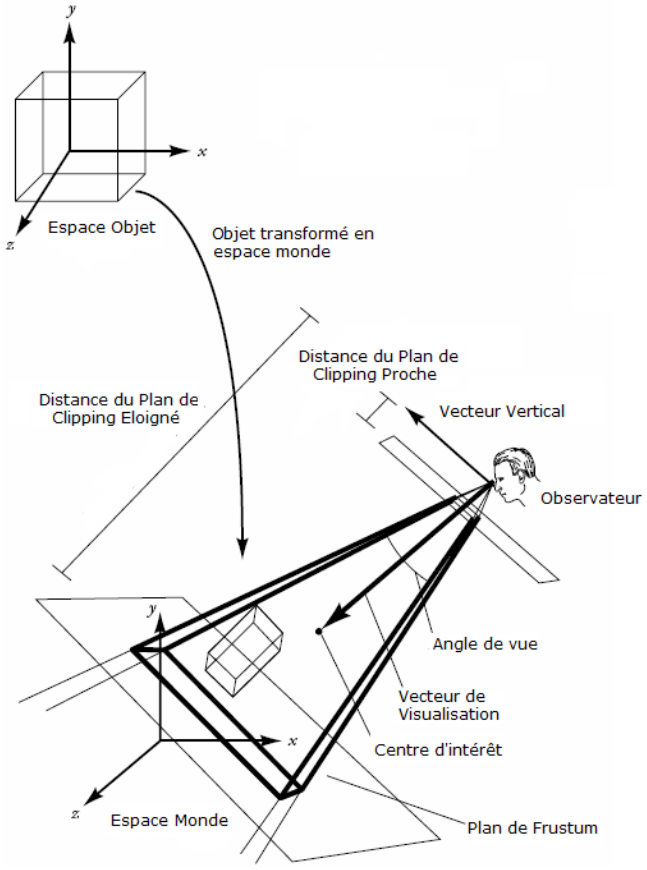


Figure 01 : *Représentation d'une scène en OpenGL.*

1.4.2. Conventions de noms de fonctions

Les fonctions OpenGL sont régies par un certain nombre de règles :

- Tous les noms de fonctions OpenGL sont préfixés : les noms de fonctions OpenGL commencent par "gl". Pour les bibliothèques annexes telles que GLU, GLX, GLAUX, GLUT les préfixes respectifs sont "glu", "glX", "aux", "glut".
- Certaines commandes OpenGL existent en plusieurs variantes, suivant le nombre et le type de paramètres qu'admet la variante. Ainsi, la fonction "glVertex3f ()"prend trois paramètres de type Float, alors que "glVertex2i ()" fonctionne avec 2 entiers. Certaines fonctions possèdent une version suffixée par un "v". Les paramètres sont passés à ces fonctions par l'intermédiaire d'un pointeur sur un tableau.
- OpenGL redéfinit des types de données numériques : GLint correspond aux entiers, GLfloat aux flottants, GLbyte aux caractères non signés... . Il est préférable d'utiliser ces types de données plutôt que les types standards du C.

Suffixe	Type de donnée	C++	OpenGL
b	entier 8 bits	char	GLbyte
s	entier 16 bits	short	GLshort
i	entier 32 bits	int ou long	GLint GLsizei
f	réel 32 bits	float	GLfloat GLclampf
d	réel 64 bits	double	GLdouble GLclampd
ub	entier non signé 8 bits	unsigned char	GLubyte GLboolean
us	entier non signé 16 bits	unsigned short	GLushort
ui	entier non signé 32 bits	unsigned int ou long	GLuint GLenum GLbitfield

Tableau 01 : *Types de données.*

1.5. Transformations géométriques

OpenGL propose un certain nombre de transformations de base : les translations, les rotations et les homothéties centrées en l'origine.

1.5.1. Rappel sur calcul matriciel

Les algorithmes 3D utilisent le calcul matriciel, car il permet de rassembler tous les calculs nécessaires sous une forme compacte. OpenGL travaille avec des matrices 4x4, dans un système de coordonnées dit homogène. L'avantage de ce système est qu'il permet de représenter les transformations par des matrices et de composer toutes ces transformations par un simple produit matriciel.

Les points de l'espace sont stockés dans des vecteurs de 4 lignes $P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$. Les 3 premières lignes contiennent respectivement les coordonnées x, y et z du point. La quatrième ligne contient un coefficient noté w, appelé facteur d'échelle, qui vaut souvent 1. Les coordonnées homogènes permettent de définir des points situés à l'infini, en choisissant w = 0. Les formes des matrices de translation, de rotation, et d'homothétie. Une homothétie est un changement d'´echelle de l'objet.

Dans la matrice d'homothétie on trouve 3 facteurs s_x, s_y et s_z. Si ces trois coefficients sont égaux, on est dans le cas d'une homothétie uniforme. Si les facteurs ne sont pas identiques, on a une homothétie non uniforme, qui vous donne la possibilité d'étirer et de rétrécir différemment suivant les axes x, y et z.

- La matrice de translation est : $M_t = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- La matrice de rotation au tour de l'axe x avec un angle θ est : $M_r = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- La matrice d'homothétie est : $M_s = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

1.5.2. Les transformations en OpenGL

La bibliothèque OpenGL dispose de trois matrices : une matrice de transformation-visualisation, une matrice de projection, et une matrice de texture. Ces trois matrices sont stockées dans des structures de données internes à la bibliothèque.

Afin de travailler sur ces matrices, OpenGL définit une matrice active, **glMatrixMode**(GLenum mode). Le paramètre "mode" désigne la matrice que l'on souhaite activer. Il peut prendre comme valeur GL_MODELVIEW, GL_PROJECTION, et GL_TEXTURE.

Définir une transformation avec OpenGL consiste à placer la matrice correspondant à cette transformation dans la matrice de modélisation-visualisation. Bien qu'il soit possible de spécifier directement les 16 coefficients de la matrice active grâce à la fonction **glLoadMatrix**(), on procède en général autrement, en utilisant les fonctions **glLoadIdentity**(), **glRotate**(), **glTranslate**() et **glScale**().

- **glLoadIdentity**() a pour effet de placer dans la matrice active la matrice dite d'identité, $I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.
- **glTranslatef**(float x,float y,float z) multiplie (à droite) la matrice active par une matrice de translation de vecteur (x;y;z).
- **glRotatef**(float theta,float x,float y,float z) multiplie la matrice active par une matrice de rotation d'angle thˆeta autour de l'axe passant par l'origine et port´e par le vecteur (x;y;z).
- **glScalef**(float s_x, float s_y, float s_z) multiplie la matrice active par une matrice de d'homothétie dont les facteurs suivant les axes x, y et z sont respectivement s_x, s_y et s_z.

1.6. Les transformations de visualisation

Le positionnement de l'observateur est primordial car il conditionne notre affichage. Il existe différentes manières d'aborder la question des transformations de visualisation. Par exemple : un déplacement de l'observateur de *x* dans une direction donnée équivaut à un

déplacement de tous les objets de la scène de -*x* dans cette direction. De la même manière, faire tourner la caméra d'un angle θ produit le même résultat qu'une rotation d'angle -θ appliquée à tous les objets de la scène. Finalement, les transformations de visualisation ne sont rien d'autre que des transformations de modélisation.

Par défaut, la caméra est positionnée en l'origine, qu'elle regarde en direction de l'axe des *z* négatifs, et que l'axe *y* correspond à la verticale. En combinant rotations et translations, il est possible de faire prendre n'importe quelle position à notre caméra virtuelle.

La construire d'une scène 3D se fait comme suit : tout d'abord, construire la scène en faisant abstraction de la position de l'observateur. Une fois la scène construite, positionné l'observateur en appliquant à tous les objets une translation et des rotations. Compte tenu du fait qu'en OpenGL, les opérations se décrivent dans l'ordre inverse de leur application, les transformations de visualisation sont les premières à être spécifier.

Exemple :

```
void display()
    {
     glLoadIdentity();
     glTranslated(0,0,-5);
     glTranslated(1,0,0);
     glBegin(GL_POLYGON);
     ...
     glEnd();
    }
```

La bibliothèque GLU d'OpenGL, propose une fonction annexe qui rend le positionnement de la caméra plus facile qui est, **gluLookAt**(), dont le prototype est le suivant :

```
void gluLookAt(GLdouble camx, GLdouble camy, GLdouble camz, GLdouble centrex,
               GLdouble centrey, GLdouble centrez, GLdouble hautx,GLdouble hauty,
               GLdouble hautz);
```

- **gluLookAt**() simplifie la spécification de transformations de visualisation.
- La caméra est positionnée en (camx, camy, camz),
- elle est dirigée vers le point (centrex, centrey, centrez),
- et l'axe (hautx, hauty, hautz) correspond à la verticale de la caméra.

gluLookAt() calcule les rotations et translations nécessaires pour positionner la caméra correctement, et multiplie le résultat à droite de la matrice active. En réécrivant l'exemple précédant cela donne :

Exemple :

```
void display()
    {
     glLoadIdentity();
     gluLookAt(-1,0,5,0,0,0,0,1,0);
     glBegin(GL_POLYGON);
     ...
     glEnd();
    }
```

1.7. Les transformations de projection

Les transformations de projection correspondent au passage de la scène 3D à une image 2D affichable à l'écran. La matrice pour stocker les transformations de projection est activée en appelant **glMatrixMode**(GL_PROJECTION) et la fonction qui permet de spécifier une matrice en appelant la fonction **glLoadMatrix**(). Plusieurs projections sont possibles mais OpenGL fournit des fonctions permettant de spécifier simplement deux types de projection, la projection orthogonale et la projection perspective, tout en gérant le problème du fenêtrage.

1.7.1. Le surfenêtrage

Lorsqu'on observe une scène 3D, il est possible qu'un certain nombre d'objets se situent hors du champ de vision. Le surfenêtrage, ou clipping en anglais, consiste à déterminer à l'avance les objets de la scène qui se situent hors du champ de la caméra. Pour cela, on définit un volume de vue : tous les objets se situant à l'intérieur du volume seront affichés à l'écran, les objets situés à l'extérieur du volume seront ignorés. Si un objet est partiellement inclus dans le volume, il sera tronqué, et seule la partie située à l'intérieur du volume sera affichée.

1.7.2. La projection perspective

OpenGL fournit deux fonctions pour spécifier une projection perspective. La première se nomme **gluPerspective**, et a le prototype suivant :

void **gluPerspective**(GLdouble fovy, GLdouble Aspect, GLdouble near,
GLdouble far);

Les paramètres de **gluPerspective**() permettent de définir le volume de vue associé à la projection perspective.

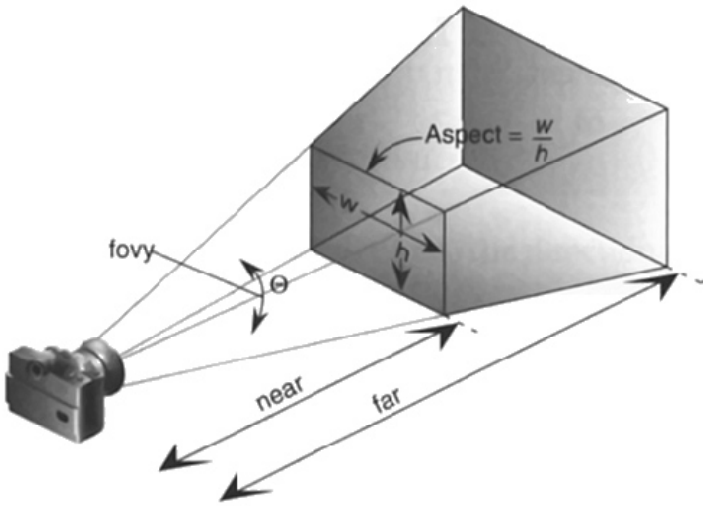


Figure 2: *projection perspective.*

On peut également utiliser la fonction :

void **glFrustum**(GLdouble left,GLdouble right, GLdouble bottom, GLdouble top,
GLdouble Near, GLdouble far);

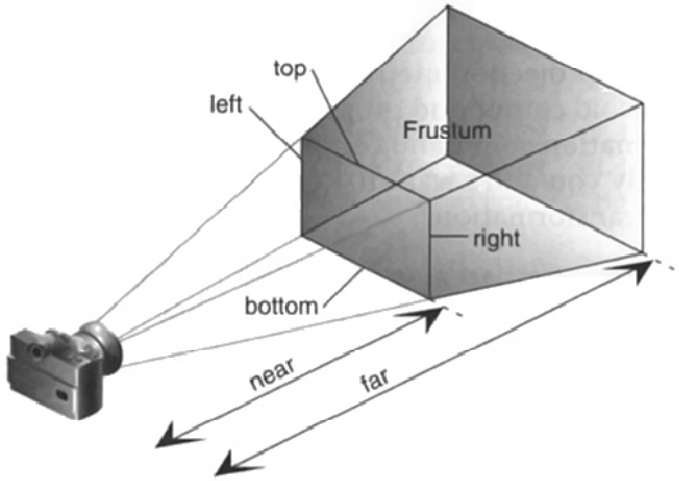


Figure 3: *projection frustum.*

1.7.3. La projection orthogonale

Pour la projection orthogonale le volume de vue est un parallélépipède rectangle. Ainsi la taille des objets n'est pas influencée par la distance au point de vue. Ce type de projection est utilisé en CAO où il est important de maintenir la taille des objets et leurs positions relatives. Voici le prototype de la fonction **glOrtho** :

void **glOrtho**(GLdouble left,GLdouble right, GLdouble bottom, GLdouble top,
GLdouble Near, GLdouble far) ;

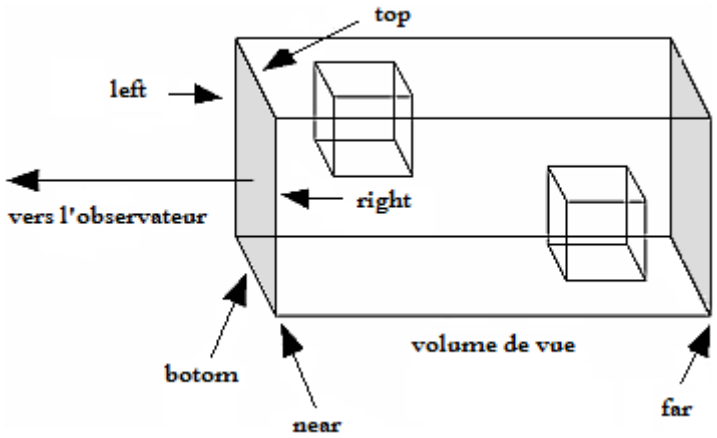


Figure 4: *projection orthogonale.*

1.8. Transformation des normales :

En OpenGL les normales sont multipliées par une matrice particulière pour conserver leur caractéristique qui les lie aux surfaces.

Soit un plan (P) dont la normal $\vec{n}$ est défini par le quadruplet (a, b, c, d) dont l'équation du plan est :

$$[a \quad b \quad c \quad d] \cdot [x \quad y \quad z \quad 1]^T = 0$$

Soit une transformation affine M qu'on applique au vecteur $\vec{n}$ et soit $\vec{n}'$ le résultat. Le vecteur $\vec{n}'$ doit vérifier la condition suivant pour qu'il soit normal au plan (P) :

$$N' \cdot (M[x \quad y \quad z \quad 1])^T = 0$$

Ce qui signifie que : $N' = (M^T)^{-1} \cdot N$

Donc les normales sont transformées par la transposé inverse de la matrice M. En GLSL la matrice **glNormalMatrix** se charge de ce calcul.

2. Eclairage et matériaux réalistes

Un sommet dont la couleur est définie comme rouge apparaît toujours à l'écran avec le même rouge, quelque soit le point de vue depuis lequel on l'observe. Dans la réalité, un point de couleur rouge apparaîtra différemment suivant l'éclairage auquel il est soumis et suivant l'angle observation. OpenGL permet d'effectuer un rendu prenant en compte l'illumination des objets. Pour simuler de manière réaliste une scène tridimensionnelle il est nécessaire de reproduire les lois physiques qui régissent la lumière. Ces lois sont complexes, et il est impossible de les reproduire exactement. On se contente donc de choisir un modèle mathématique aussi proche que possible de la réalité. En fonction de l'application à laquelle il est destiné, le modèle d'illumination choisi permettra d'obtenir des résultats plus ou moins réalistes et rapides.

Le modèle d'illumination utilisé par OpenGL, se base sur 3 points :

– Les sources lumineuses

– Les matériaux

– L'algorithme de remplissage

2.1. Sources lumineuses

Avec OpenGL, une scène peut contenir jusqu'à huit sources de lumière. Ces sources peuvent être de type "directionnel", ou "spot".

Une source de lumière est caractérisée par 10 paramètres qu'on peut classer en 4 catégories :

a) Les paramètres de lumière

La lumière émise par une source est formée de trois composantes : la plus importante, la composante diffuse, est réfléchie par un objet dans toutes les directions. La composante spéculaire correspond à la lumière qui est réfléchie dans une direction privilégiée (et qui est donc à l'origine de l'effet de brillance). La composante ambiante est la plus difficile à appréhender. La lumière ambiante d'une scène est une lumière non directionnelle, que l'on peut considérer comme issue des multiples réflexions de rayons lumineux. OpenGL permet d'affecter une lumière ambiante pour toute la scène. La composante ambiante d'une source ajoute une contribution à cette lumière ambiante globale.

b) Le paramètre de position

Comme son nom l'indique, il permet de définir la position de la source de lumière dans la scène.

c) Les paramètres de spot

L'angle de coupure, permet de définir le type de source utilisée : si l'angle vaut 180°, la source de lumière omnidirectionnelle. Si l'angle est compris entre 0° et 90°, la source est un spot, et les deux autres paramètres de spot permet de modifier : la direction dans laquelle il pointe et l'exposant du spot. Ce dernier paramètre permet de faire varier la concentration de la lumière à l'intérieur du cône définissant le spot.

d) Les paramètres d'atténuation

Permettent de prendre en compte le phénomène physique suivant : plus un objet est éloigné d'une source de lumière, moins il est éclairé par cette dernière. Les paramètres

d'atténuation sont au nombre de 3 : le facteur d'atténuation constante, le facteur d'atténuation linéaire et le facteur d'atténuation quadratique, noté respectivement A_c, A_l et A_q, l'intensité reçue par un point P situé à une distance d de la source lumineuse est divisée par :

$$A_c + A_l d + A_q d^2$$

2.2. Matériau

La spécification d'un matériau pour un objet se fait par l'intermédiaire de 5 paramètres :

– La couleur diffuse
– La couleur spéculaire
– La couleur ambiante
– La couleur émise
– Le coefficient de brillance

Dans la réalité, un rayon lumineux est constitué d'un ensemble d'ondes de longueurs différentes. A chaque longueur d'onde correspond une couleur (visible ou non). Un objet frappé par un rayon lumineux va absorber certaines longueurs d'onde et réfléchir les autres. En informatique, une couleur est représentée par un triplet de composantes rouge, verte, et bleue. Par synthèse additive, on arrive à recréer à partir de ces trois composantes la plupart des couleurs visibles. Pour définir le comportement d'un objet vis-à-vis d'une couleur RVB, il suffit de dire quel pourcentage de chaque composante est réfléchi.

La lumière émise par une source est décomposé en lumière diffuse, spéculaire et ambiante, on peut spécifier le comportement du matériau pour chacune de ces composantes, ce qui explique les trois premiers paramètres du matériau : couleur diffuse, couleur spéculaire et couleur ambiante.

Le paramètre de couleur émise correspond à une composante de lumière supplémentaire, qui permet de prendre en compte le fait que certains objets peuvent émettre eux-mêmes de la lumière.

2.3. L'algorithme d'´eclairage

Pour des raisons d'efficacité, OpenGL ne calcule pas la couleur de chaque pixel d'un polygone par contre, soit il remplit chaque polygone avec un couleur unie ce mode de remplissage est dit le "Flat", soit il utilise un algorithme de Gouraud ce mode est dit le "Smooth", dont le principe est le suivant : pour un polygone donné, la couleur de chacun des sommets est calculée, et le polygone est rempli avec un dégradé entre ces différentes couleurs.

Pour calculer correctement la réflexion des rayons lumineux en un point, OpenGL a besoin de connaître la perpendiculaire à la surface de l'objet au point considéré. On appelle cette donnée une normale.

2.4. Initialisation des paramètres

a) Paramètres d'´éclairage

La phase d'initialisation de l'éclairage commence par la spécification du mode de remplissage des polygones avec : **glShadeModel**(GL_SMOOTH);

Ensuite on indique à OpenGL qu'on souhaite utiliser le calcul d'éclairage, en activant la variable d'état GL_LIGHTING : glEnable(GL_LIGHTING);

OpenGL permet d'utiliser jusqu'à huit sources de lumière. Ces huit lampes sont indexées par les constantes GL_LIGHT0 à GL_LIGHT7. Il faut activer chacune des sources qu'on souhaite utiliser, par exemple dans le cas de deux lampes :

glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);

Vient ensuite le paramétrage des lampes. Il se fait avec une unique fonction, **glLightfv**(), dont le prototype est le suivant :

void **glLightfv**(GLenum lampe,GLenum nomparam, GLType param);

Où lampe désigne la lampe dont on veut modifier une propriété, nomparam est le nom du paramètre à modifier. Il s'agit d'une des dix propriétés de source lumineuse qui sont :

– GL_DIFFUSE
– GL_AMBIENT
– GL_SPECULAR
– GL_POSITION
– GL_SPOT_CUTOFF
– GL_SPOT_DIRECTION
– GL_SPOT_EXPONENT
– GL_CONSTANT_ATTENUATION
– GL_LINEAR_ATTENUATION
– GL_QUADRATIC_ATTENUATION

"param" désigne la valeur à affecter au paramètre choisi.

La définition de la position des sources de lumière se trouve dans la fonction d'affichage. En effet, tout comme les sommets des polygones, les paramètres de position et de direction d'une source subissent les transformations contenues dans la matrice de modélisation-visualisation. Il faut donc placer judicieusement la déclaration de ces deux paramètres.

b) Paramètres de matériaux

Le système d'affectation des propriétés de matériau utilise le principe de machine à états.

OpenGL gère un matériau courant. Lorsqu'un polygone est décrit, il se voit affecter le matériau courant.

La modification du matériau courant se fait avec la fonction **GlMaterialfv**() :

void **glMaterialfv**(GLenum face, GLenum nomparam, Gltype param);

où face indique la face avant ou arrière dont on souhaite modifier les paramètres. Comme pour **glLightfv**(), nomparam désigne la propriété qu'on souhaite changer, et param est un tableau contenant la nouvelle valeur à affecter à nomparam. Les valeurs de nomparam possibles sont :

– GL_AMBIENT
– GL_DIFFUSE
– GL_SPECULAR
– GL_EMISSION
– GL_SHININESS (coefficient de brillance)

3. Placage de texture

Le placage de texture est une technique permettant d'accroître le réalisme d'un rendu 3D. Cela consiste à coller une image sur un objet 3D à la manière d'une tapisserie.

3.1.Utilisation de la texture

La première étape nécessaire au placage de la texture est évidemment le chargement de l'image avec une fonction au format de l'image par exemple **LoadBMPImage**(). Ensuite paramétrer la manière dont OpenGL devra filtrer la texture lors de l'application de celle-ci sur un objet. Lors du rendu, la texture va être déformée par la perspective. A certains endroits elle va être étirée, à d'autre elle va être rétrécie. Le filtrage définit la méthode de calcul final de la texture déformée. OpenGL propose les méthodes du plus proche voisin "nearest" et linèaire "linear".

Le paramétrage de la méthode de placage de texture se fait grâce à la fonction :

void **glTexParameteri**(Glenum cible,GLenum nomparam,GL valeur)

"cible" définit le type de texture que l'on veut paramétrer GL_TEXTURE_1D, GL_TEXTURE_2D ou GL_TEXTURE_3D.
"nomparam" désigne le paramètre que l'on souhaite modifier, soit la méthode de filtrage lors d'un étirement GL_TEXTURE_MAG_FILTER ou d'un rétrécissement GL_TEXTURE_MIN_FILTER.
"valeur" correspond à la nouvelle valeur à affecter au paramètre, soit GL_LINEAR ou GL_NEAREST.

L'étape suivante consiste à définir la texture a utilisée, par exemple une texture 2D:

void **glTexImage2D**(GLenum cible,Glint niveau,Glint formatinterne,GLsizei largeur,
GLsizei hauteur,GLint bord,GLenum format,GLenum type,
const Glvoid texture) ;

Enfin la dernière étape consiste à positionner la texture sur le polygone. En anglais, on parle mapping. Le principe est simple : on affecte un système de coordonnées (u,v) à la texture suivant l'illustration de la figure ci dessous. Positionner la texture consiste à affecter à chaque sommet d'un polygone la coordonnée de la texture en ce point. Pour des polygones carrés, la tâche est simple, les coordonnées de textures aux sommets des points des faces sont simplement (0;0), (1;0), (1;1) et (0;1). A l'instar de la couleur des sommets, les coordonnées de texture se spécifient lors de la déclaration des polygones entre un **glBegin**() et un **glEnd**(). Le principe est toujours le même : lorsqu'un sommet est déclaré avec **glVertex**(), la coordonnée de texture courante lui est affectée. La modification de la coordonnée de texture 2D courante se faite par un appel à : **glTexCoord2f**(GLfloat u,Glfloat v)

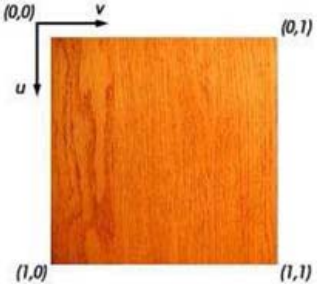


Figure 5 : *Coordonnées de texture.*

ANNEXE B

« Généralités sur OpenGL Shading Language »

Depuis le C++, GLSL a repris une syntaxe proche du C, le point d'entrée de fonction "main", le principe de surcharge des fonctions, le type bool, la façon de déclarer les variables, le principe des constructeurs et etc. Le langage GLSL fonctionne avec la même logique que le C et le C++ au niveau de la compilation et du *link*. Tout programme doit subir ces opérations pour être exécuté. Cependant, qu'il n'est pas possible de récupérer un fichier issu de la compilation et du *link*. De plus, le compilateur est intégré dans les drivers de la carte graphique et son utilisation se fait au travers de l'API OpenGL.

L'utilisation d'un langage de programmation GPU a de fortes implications sur le déroulement d'un programme OpenGL. Le rendu du brouillard, la transparence ou encore l'éclairage ne sont plus pris en compte. Il faudra reprogrammer le rendu de ces différentes fonctionnalités dans le vertex shader voir le fragment shader.

1. Type de données et variables en GLSL

1.1. Types Simples

Le langage GLSL a beaucoup de points communs avec le langage C. Syntaxiquement, les deux langages sont quasiment identiques. GLSL permet de déclarer des variables de type simple, entier, flottant, et booléen dont la syntaxe est identique à celle du C. il permet aussi de déclarer des variables vectorielles de 2, 3 ou 4 composants (vec2, vec3 et vec4) et matricielles de 2x2, 3x3 ou 4x4 composants (mat2, mat3 et mat4),voir le *tableau 1*.

Type de données	Description
float	flottant
int	entier
vec2	vecteur à 2 composantes flottantes
vec3	vecteur à 3 composantes flottantes
vec4	vecteur à 4 composantes flottantes
mat2	matrice 2x2 de flottants
mat3	matrice 3x3 de flottants
mat4	matrice 4x4 de flottants
ivec2	vecteur à 2 composantes entières
ivec3	vecteur à 3 composantes entières
ivec4	vecteur à 4 composantes entières
bvec2	vecteur à 2 composantes booléennes
bvec3	vecteur à 3 composantes booléennes
bvec4	vecteur à 4 composantes booléennes

Tableau 1 : *type de variables*

GLSL défini un ensemble de type spécial qui est propre à la manipulation des textures qui est le type **sampler.** Les samplers permettent l'accès aux valeurs de texture (texel) elles sont définie comme suit :

Type de données	Description
sampler1D	pour une texture 1D
sampler2D	pour une texture 2D
sampler3D	pour une texture 3D
samplerCube	pour une texture cube
sampler1DShadow	pour shadow map 1D
sampler2DShadow	pour shadow map 2D

Tableau 2 : *Le type sampler.*

Remarque :

- les types de données **sampler** n'indique pas un identifiant d'objet de textures mais simplement l'unité de texture utilisé. C'est l'équivalent de **GL_TEXTURE0** dans cette instruction : **glMultiTexCoord2f (GL_TEXTURE0, 0, 0);**

- Les extensions OpenGL peuvent étendre le langage de nouveaux types de données. Par exemple, l'extension GL_texture_rectangle d'OpenGL 2.0 ajoute les types **samplerRect** et **samplerRectShadow**.

- Le type **void** n'est utilisé que pour les fonctions qui ne retournent rien.

1.2. Type de données complexes

GLSL offre la possibilité de créer des structures de données à la manière du langage C :

```
struct SVertex
{
vec3 Normal ;
vec2 TexCoord ;
vec3 Position ;
};
```

Remarque : la définition de variables de types structures se réalise ainsi : SVertex UnVertex;

GLSL permet la création de tableaux qui fonctionnent globalement comme les tableaux C mais avec une souplesse sur sa taille beaucoup plus grande. Il est ainsi possible de déclarer un tableau sans indiquer sa taille puis on adapte sa taille en fonction de nos besoins.

1.3. Variables qualificatifs

Les variables GLSL peuvent être déclarées avec quatre qualificatifs différents indiquant la façon de communiquer des variables avec d'autres programmes.

- Le qualificatif « **const** » fonctionne de la même manière qu'en C. La valeur de la variable n'est pas modifiable lors de l'exécution. Cette variable est accessible seulement par le programme qui la déclare.
- Le qualificatif « **varying** » offre une solution pour envoyer des informations du vertex program au fragment program. Si la valeur GL_SMOOTH a été passée à glShadeModel () dans le programme C/C++ alors la valeur de la variable dans le fragment program sera interpolé suivant la valeur qu'elle prend pour chaque vertex qui génère la surface dont le fragment est issu.
- Le qualificatif « **uniform** » indique que la valeur de la variable est valable et identique pour tous les vertrices et tous les fragments. La valeur de la variable est transmise par le programme C/C++.
- Le qualificatif « **attribute** » indique que la valeur de la variable est valable pour un vertex uniquement. La valeur de la variable est transmise par le programme C/C++.

Il existe un grand nombre de variables prédéfinis. En voici une liste :

- Les variables d'entrée d'un vertex shader permet de récupérer des valeurs des données vertex qui seront traité dans le vertex shader. Elles sont énumérées ci dessous :

Nom de la variable GLSL	Type	Fonction OpenGL	Description
gl_Vertex	attribute vec4	glVertex*()	Position du sommet
gl_Color	attribute vec4	glColor*()	Couleur du sommet
gl_Normal	attribute vec3	glNormal*()	Normale du sommet.
gl_MultiTexCoordn	attribute vec4	glMultiTexCoord*() ou glTexCoord*()	Coordonnées de l'unité de texture n
gl_SecondaryColor	attribute vec4	glSecondaryColor*()	Couleur secondaire du sommet
gl_FogCoord	attribute float	glFogCoord*()	Coordonnées de brouillard

Tableau 3 : *Variables d'entée du vertex shader.*

Comme un vertex shader manipule des attributs de vertex, les variables d'entrée du vertex sont précédées du qualifier **attribute**.

- Les variables de sortie elles sont destinées à recevoir les résultats du traitement. Elles sont énumérées ci dessous :

Nom de la variable GLSL	Type	Description
gl_Position	vec4	Position en coordonnées écran du sommet
gl_FrontColor	varying vec4	Couleur du côté "avant" de la face à laquelle est rattaché le sommet
gl_BackColor	varying vec4	Couleur du côté "arrière" de la face à laquelle est rattaché le sommet
gl_FrontSecondaryColor	varying vec4	Couleur secondaire du côté "avant" de la face à laquelle est rattaché le sommet
gl_BackSecondaryColor	varying vec4	Couleur secondaire du côté "arrière" de la face à laquelle est rattaché le sommet
gl_TexCoord[n]	varying vec4 (tableau)	Coordonnées de l'unité de texture n
gl_FogFragCoord	varying float	Coordonnée de fog
gl_PointSize	float	Taille du point du sommet
gl_ClipVertex	vec4	Vecteur utilisé pour les plans de clipping

Tableau 4 : *Variables de sortie du vertex shader.*

Comme on peut le voir sur le tableau ci-dessus que certaine variables sont qualifier de **varying** car on a vu plus haut que le vertex shader communique avec le fragment shader avec ce type de variable uniquement.

- Les variables d'entrée d'un fragment shader permet de récupérer des valeurs des données des primitives qui seront traité dans le fragment shader. Elles sont énumérées ci dessous :

Nom de la variable GLSL	Type	Description
gl_Color	varying vec4	Couleur du pixel
gl_FragCoord	vec2	Coordonnées écran du pixel
gl_SecondaryColor	varying vec4	Couleur secondaire
gl_FrontFacing	bool	Test si le fragment appartient à la face avant
gl_TexCoord[n]	varying vec4 (tabeau)	Coordonnées de l'unité de texturage n
gl_FogFragCoord	varying float	Coordonnée de fog

Tableau 5 : *Variables d'entrées du fragment shader.*

Le qualifier **varying** spécifie que les variables sont reçues du vertex shader.

- Les variables de sortie d'un fragment shader sont destiner à recevoir les valeurs des primitives traitées et de les transmettre au frame buffer pour être affichées. Elles sont énumérées ci-dessous :

Nom de la variable	Type	Description
gl_FragColor	vec4	Couleur finale du pixel
gl_FragDepth	float	Profondeur du pixel dans le depth buffer
gl_FragData[n]	tableau de vec4	En rapport avec glDrawBuffers()

Tableau 6 : *Variables de sortie du fragment shader.*

1.4. Variables uniformes prédéfinies

GLSL fournit un ensemble de variables qui convient pour connaitre les états courant de rendu pendant l'exécution du programme à partir d'un shader. Voir (livre orange) pour une liste complète.

Par exemple :

uniform mat4 **gl_ModelViewProjetionMatrix** ;
uniform mat4 **gl_ModelViewMatrixIverseTranspose**;

Variables attribute et varying définies par l'utilisateur

L'utilisateur peut définir ces propres variables attributs et variables varying qui gérera lui-même.

Par exemple :

attribute vec3 **myTangent** ;
varying vec4 **myNormalPrime** ;

1.5. Fonctions intégrées

GLSL fournit un ensemble de fonction intégrée tel que :

- Les fonctions trigonométriques (sin, cos, tan,…),
- Les fonctions exponentielles (pow, exp, log,…),
- Les fonctions usuelles (abs, sign, floor, ceil…),
- Les fonctions pour la géométrie (length, distance, dot, normalize,…),

- Les fonctions de comparaisons (lessThan, greaterThan, …),
- Les fonctions pour les textures (texture1D, texture2D, textureCube,…),
- Les fonctions sur les fragments (dFdx, dFdy,…),
- Les fonctions sur les nombres aléatoires (noise1, noise2,…).

2. L'API GLSL

2.1. Les extensions, le langage et l'API

Pour utiliser GLSL, les drivers de la carte graphique doivent supporter 4 extensions. La première extension est la version du shader "GLEW_VERSION" la carte doit au moins supporter "OpenGL 1.5". Celle-ci n'est pas suffisante car nous avons également besoin de "GL_shader_objects" qui définit un nouvel object OpenGL à la manière des objets de textures, "GL_vertex_shader" pour la programmation de vertex shader et "GL_fragment_shader" pour la programmation de fragment shader. Avec "OpenGL 2.0" les extensions ont étés intégrer au corps de la spécification. Notons que de nouvelle version d'OpenGL ont été créée telle la "version 3.0" et la "version 4.0."

La création d'un programme shader se fait en suivant ces étapes :

- Créer un ou plusieurs shaders objets (vides) avec **glCreateShader**.
- Fournir au shader le code source avec **glShaderSource**.
- Compiler chaque shaders objets avec **glCompileShader**.
- Créer un programme objet avec **glCreateProgram**.
- Attacher tous les shaders objets au programme objet avec **glAttachShader**.
- Lier le programme objet avec **glLinkProgram**.
- Utiliser le programme exécutable comme état d'OpenGL avec **glUseProgram**.

2.2. Création d'un shader objet

Un Shader objet peut être un code objet compilé d'un vertex shader ou d'un fragment shader. Le shader objet pourra être compilé après lui avoir spécifié le code source du shader. Le code source créer doit alors être compilé, les divers modules compilés doivent être liés, et finalement le code en résultant peut être exécuté par le processeur adéquat. Un code source n'est autre qu'une chaîne de caractère. Quand un objet est créé, OpenGL renvoi un identifiant unique pour celui-ci. Cet identifiant peut être utilisé pour manipuler l'objet et pour renseigner ou faire des requêtes sur les paramètres de l'objet. La première étape pour créer un programme shader est de créer un shader objet, la commande qui nous permet de créer un shader objet est :

GLuint **glCreateShader**(GLenum shaderType)

shaderType spécifie le type de shader à créer. Il prend comme valeur l'un des constantes "GL_VERTEX_SHADER" ou "GL_FRAGMENT_SHADER". Chacune de ces constantes représente respectivement un shader de sommet et un shader de pixel. Après avoir créé un shader objet, les chaînes qui définissent le code source du shader doit être fournie. Le code source d'un shader est fourni dans un tableau de chaînes de caractères. La commande qui permet de spécifier le code source est :

Void **glShaderSource**(GLuint shader, GLsizei count, const GLchar string, const GLint length)

"Shader", est l'identifiant de notre shader, afin que la fonction adresse notre code source à ce shader et pas un autre.
"Count", est le nombre de chaînes contenues dans string.
"String", est notre code source, décomposé en plusieurs chaînes.
"Length", est la longueur de la chaîne caractère du code source, que nous laisserons à NULL.

Cela permet de renvoyer le code source à OpenGL est de pouvoir par la suite le compiler au moment voulu.

Après que le code source soit chargé dans un shader objet, le code source doit être compilé pour vérifier sa validité. La commande pour compiler un shader objet est :

void **glCompileShader**(GLuint shader)

"shader", est l'identifiant de notre shader.

2.3. Création d'un programme objet

Chaque objet shader est compilé de façon indépendante. Pour créer un programme exécutable par la carte graphique, les applications ont besoin d'un mécanisme permettant de spécifier une liste d'objets de shader à liés. Ce qui revient donc, à créer un programme objet au quel nous attacherons les shaders objets. La création d'un programme objet se fait par la commande qui suit :

GLuint **glCreateProgram**(void)

Cette commande ne prend aucun paramètre elle renvoie uniquement l'identifiant du programme créé. Par la suit nous devons attacher les shaders objets à ce programme, la commande qui permet d'attacher un shader objet à un programme est :

void **glAttachShader**(GLuint program, GLuint shader)

"Program", est l'identifiant du programme qui recevra le shader.
"Shader", est le shader à attacher.

Pour créer un programme valide, tous les shaders objets attachés à au programme objet doivent être compilés et le programme objet lui-même doit être lié (linké). Cette opération permet de lier le vertex shader et le fragment shader. La commande qui permet de lier un programme est :

void **glLinkProgram(**GLuint program)

"Program", identifiant du programme à lier.

Après l'opération de lien la source des shaders peut être modifiée, et les shaders recompilé sans affecter le programme. Quand l'opération de lien est accomplie avec succès, le programme qu'elle contient peut être installé en tant qu'élément de l'état courant du rendu. La commande qui permet d'utiliser le program en tant qu'élément de rendu est :

void **glUseProgram**(GLuint program)

"Program", identifiant du programme à utiliser.

Déboguer un shader est très dure. Et comme toute programmation il est nécessaire de vérifier le sucé de chaque compilation et de chaque opération de lien (link) et dans le cas d'erreurs de pouvoir les localisées afin de les corriger. A cet effet des fonctions sont prévues pour vérifier si le code est compilé et est lié avec succès.

2.4. Vérification du succès de la compilation et du linkage

Comme pour toute compilation, des erreurs sont possibles et leur nature doit être connue. Pour cela, il nous faut savoir s'il y a eu une erreur, dans ce cas, on récupère le message qu'elle contient qui pourra être affichée à l'écran. Il existe une fonction pour récupérer l'état de la compilation. Plus globalement, il s'agit d'une fonction permettant de récupérer un entier relatif à une information spécifique d'un shader ou d'un programme.

void **glGetShader**iv(GLuint shader, GLenum pname, GLint *params)

"shader", est l'identifiant du shader.
"pname", est le type d'état demandé qui est le statu de compilation dont la valeur sera GL_COMPILE_STATUS.
"Params", est le pointeur dans lequel OpenGL écrira la valeur de l'état demandé.qui sera soit GL_TRUE ou GL_FALSE.

void **glGetProgram**iv(GLuint program, GLenum pname, GLint *params)

"Program", est l'identifiant du programme.
"pname", est le type d'état demandé qui est le statu du linkage dont la valeur sera GL_LINK_STATUS.
"Params", est le pointeur dans lequel OpenGL écrira la valeur de l'état demandé, qui sera soit GL_TRUE ou GL_FALSE.

2.5. Récupération du message d'erreur InfoLog

Les informations sur l'opération de compilation d'un shader objet, les informations sur le *link* et les opérations d'un programme objet sont stockées dans un InfoLog. L'InfoLog une chaîne de caractère qui contient les messages de diagnostic et les *warnings*. L'InfoLog peut contenir une information utile pendant le développement d'applications même si la compilation ou l'opération de link a réussi. La commande qui permet de récupérer le message d'erreur Infolog d'une shader objet est :

void **glGetShaderInfoLog**(GLuint shader,GLsizei maxLength,GLsizei length, GLchar infoLog)

Et la commande qui permet de récupérer le message d'erreur Infolog d'un programme objet est :

void **glGetProgramInfoLog**(GLuint program, GLsizei maxLength, GLsizei length, GLchar infoLog)

"shader" et "program", sont respectivement les identifiants du shader et du programme objet.
"maxLength", est le nombre maximum de caractère à extraire de InfoLog.
"Length", retourne la longueur réelle de l'InfoLog.
"infiLog", est l'InfoLog le message d'erreur.

Comme les spécifications de GLSL ne permettent pas de retrouver la longueur du message d'erreur il nous faut récupérer cette information avec la commande glGetSahder pour le shader objet et glGetProgram pour le programme objet avec comme second paramètre GL_INFO_LOG_LENGTH.

2.6. Libération de la mémoire

Chaque objet créé doit être supprimé après l'avoir utilisé car il occupe de la ressource mémoire. Et nous pouvons aussi détacher un shader d'un programme quand celui-ci quand il n'est plus utilisé.

La commande qui permet d'effacer un shader est :

void **glDeleteShader**(GLuint shader)

La commande qui permet d'effacer un programme objet est :

void **glDeleteProgram**(GLuint program)

Et la commande qui permet de détacher un shader d'un programme est :

void **glDetachShader(**GLuint program, GLuint shader)

Dans le cas d'un shader qui est encore attaché à quelques programmes (d'un ou plusieurs), le shader n'est pas supprimé réellement, mais il est seulement marqué pour la suppression. Il sera supprimer dés n'est plus attaché à n'importe quel programme.

ANNEXE C

« Modèle d'illumination local »

La lumière constitue un atout majeur dans la distinction des formes et des reliefs des objets. Cela grâce aux interactions des différents matériaux avec la lumière. La lumière connue pour être à la fois de nature corpusculaire et ondulatoire interagit avec les différents milieux quelle trouve sur son parcours suivant les propriétés de ces derniers. La nature corpusculaire correspond le mieux pour étudier ces interactions avec les hypothèses qu'un faisceau de lumière infiniment fin se propage sur une ligne droite.

1. La lumière

Lorsqu'un rayon de lumière heurte un objet trois phénomènes peuvent se produire conjointement: la réflexion, la réfraction et l'absorption. Le résultat de cette interaction est rarement un rayon unique: le rayon se brise en une infinité de rayons projetés dans une infinité de directions (ce qui ne signifie pas toutes les directions). C'est la façon dont se distribue l'énergie du rayon incident sur tous ces rayons qui caractérise la matière constituant l'objet éclairé. Dans notre cas, on considérera une réflexion spéculaire et une réflexion diffuse uniquement.

Une source lumineuse peut être soit :

- **une source ponctuelle :** ce point diffuse des rayons de lumière dans toutes les directions.
- **une source parallèle :** les rayons de lumière proviennent d'une source ponctuelle infiniment lointaine, considérant ainsi les rayons de lumière parallèle les un les autre (le soleil).
- **une source distribuée :** les rayons de lumière proviennent de différent point d'une surface lumineuse infinie (un néon).

2. Modèle d'illumination local

Les modèles d'illumination sont des méthodes de calcules utiliser dans le rendu, pour la détermination de l'intensité réfléchi par les surfaces (modèle de réflexion) pour déterminer la couleur finale perçu par l'œil. Elles se basent sur les propriétés optiques des surfaces et des conditions d'éclairement. Ces modèles considèrent une scène de rendu se composant d'une source de lumière ou de plusieurs, de l'objet à visualisé et d'un observateur (carmera).

Les modèles locaux d'illumination permettent d'approximer le calcule de l'intensité de la lumière reflétée d'un point de la surface d'un volume. Cette intensité est évaluée en fonction de l'orientation (locale) de la surface par rapport à la position d'une source lumineuse ponctuelle et des propriétés des matériaux.

L'illumination locale se distingue de l'illumination globale par le fait qu'elle ne considère que des critères locaux (réduits) pour calculer l'illumination. En ne considérant que:

- les paramètres du point ou de la face à éclairer (normale, couleur, absorption, émission)
- L'emplacement des sources lumineuses

Ce modèle est aussi appeler modèle d'ombrage, car il permet de calculer les ombres propres d'un objet et non des ombres projetées d'un volume sur l'autre. Dans la littérature, on trouve les algorithmes les plus courants pour calculer cette illumination :

- Algorithme de Lambert ;
- Algorithme de Gouraud ;
- Algorithme de Phong.

D'autres algorithmes, parfois plus réalistes mais plus coûteux, existent tels que l'algorithme de Cook-Torrance, l'algorithme de Torrance-Sparrow, basés sur des fondements physiques, ou l'ombrage par micro-facettes souvent combiné à l'algorithme de Phong pour donner l'algorithme de Blinn-Phong.

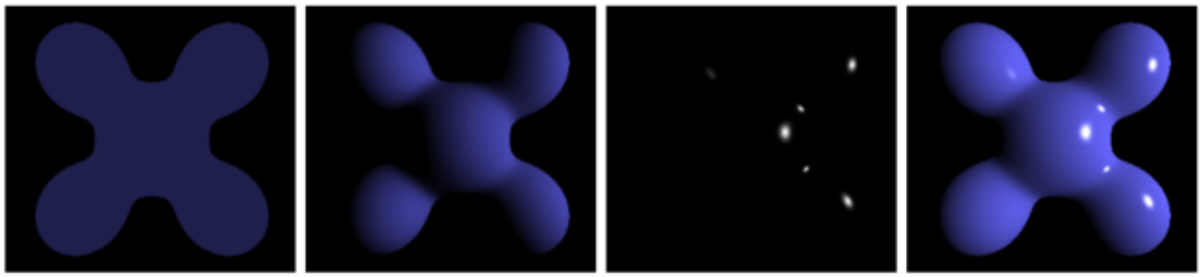
Des algorithmes encore plus récents tels que l'algorithme de Kajiya ne sont pour ainsi dire jamais utilisés en pratique. D'autres finalement sont plus spécialisés, portant par exemple sur les sources lumineuses non ponctuelles.

3. Modèle de Phong et Blinn-Phong

Nous focaliserons notre itérer sur le modèle de Phong et de Blinn-Phong, car c'est ces deux modèles qui sont utilisés comme modèle de rendu en OpenGL.

Parmi ces modèles, le modèle de Phong est le plus populaire il calcule l'intensité lumineuse comme étant une combinaison linéaire des différents termes ambiant, diffus et spéculaire. En supposant que la lumière réflexion est parfaitement spéculaire on peut écrire :

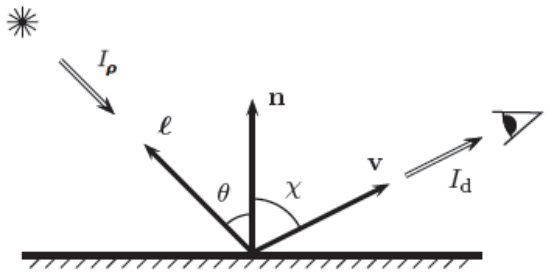
$$I_{Phong} = I_{ambiant} + I_{diffuse} + I_{speculaire}$$



3.1. La lumière ambiante

La lumière ambiante est un terme constant tel que ; $I_{ambient} = k_a = const.$

Elle permet de distinguer les objets qui ne sont pas directement éclairé. Elle correspond généralement à la réflexion des autres surfaces.



réflexion diffuse

3.2. La réflexion diffuse

La réflexion diffuse se rapporte à la lumière qui est réfléchie avec l'intensité égale dans toutes les directions (réflexion Lambertienne). La luminance d'une surface mate est indépendante de la direction de visionnement χ et dépend seulement de l'angle d'incidence θ entre la direction de la source lumineuse $\vec{l}$ et la normale de la surface d'incidence $\vec{n}$. Le terme de l'illumination diffuse, $I_{diffuse}$ est donnée par :

$$I_{diffuse} = I_\rho K_d \cos\theta = I_\rho K_d(\vec{l} \cdot \vec{n}), \quad 0 \leq \theta \leq 2\pi$$

En normalisant :

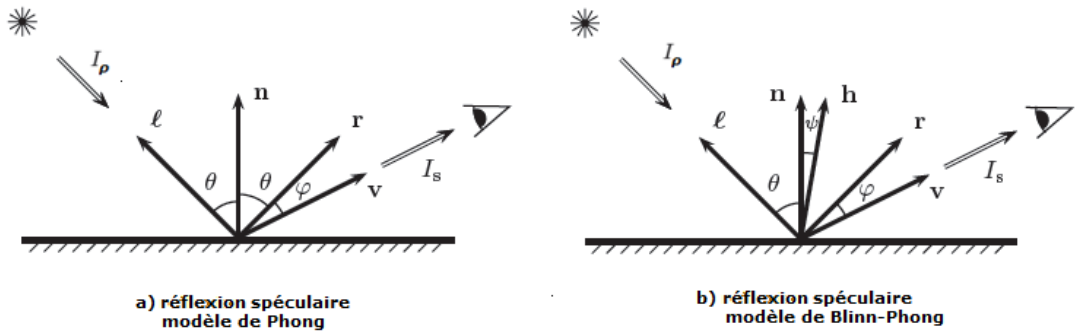
$$I_{diffuse} = I_\rho K_d(L \cdot N), \quad 0 \leq \theta \leq 2\pi$$

Avec I_ρ : l'intensité de la lumière,
θ : l'angle entre la normal de la surface et la source de lumière,
K_d : coefficient de réflexion diffuse.
$\vec{n}$: la normal de la surface au point d'incidence.
$\vec{l}$: le vecteur direction de la source.

En pratique nous implémentons le produit $(\vec{l} \cdot \vec{n})$ par $\max(0, (L \cdot N))$, L et N les vecteurs normalisés respectivement de $\vec{l}$ et $\vec{n}$.

3.3. La réflexion spéculaire

La réflexion spéculaire se manifeste par chaque surface brillante et cause le soi-disant point culminant. Le terme de la lumière spéculaire se calcule en fonction du vecteur de visualisation $\vec{v}$ qui est orienté de l'objet vers l'œil de l'observateur. La lumière est réfléchie dans la direction de la réflexion $\vec{r}$ avec le même angle d'incidence θ par rapport à la normal de la surface $\vec{n}$.



3.4. Modèle de Phong

Le terme spéculaire $I_{specular}$ est donnée par :

$$I_{specular} = I_\rho K_s \cos^n \varphi = I_\rho K_s (\vec{r} \cdot \vec{v})^n$$

En normalisant :

$$I_{specular} = I_\rho K_s (R \cdot V)^n$$

Avec : Avec I_ρ : l'intensité de la lumière,
φ : l'angle entre la normal de la surface et direction de réflexion,
K_s : coefficient de réflexion spéculaire,
$\vec{v}$: le vecteur direction de vue vers le point d'incidence,
$\vec{r}$: le vecteur direction de la réflexion,
n : coefficient de brillance (shininess).

En pratique nous implémentons le produit $(\vec{r} \cdot \vec{n})$ par $\max(0, (R \cdot V))$, R et V les vecteurs normalisés respectivement de $\vec{r}$ et $\vec{v}$.

On notera que ce modèle est purement empirique, et qui ne tiens pas compte de l'atténuation de la lumière.

3.5. Modèle de Blinn-Phong

Ce modèle est très intéressant car il se place à une échelle macroscopique pour décrire le la réflexion de la lumière, considérant ainsi les surfaces formées de micro-facettes qui ont le comportement d'un miroir.

Blinn effectue les calculs du spéculaire en remplaçant le vecteur $\vec{r}$ par la bissectrice $\overrightarrow{h}$ entre le deux vecteurs $\overrightarrow{l}$ et $\vec{v}$, tel que :

$$R = 2N(L \cdot N) - L \; et \; H = L + V$$

Se qui permet d'écrire :

$$I_{specular} = I_\rho K_s (N \cdot H)^n$$

Avec : H : est le vecteur bissecteur normalisé du vecteur de la source et vue.

Notons que la lumière est aussi définie avec les trois canaux de couleur RVB, ce qui fait que chaque calcul des intensités $I_{ambiant}$, $I_{diffuse}$ et $I_{spéculaire}$ sont calculées pour chaque canaux.

BIBLIOGRAPHIE

[1] T. Akenine-Möller et E. Haines, *Real-Time Rendering*, 2nd ed. A. K. Peters Ltd., 2002.

[2] M. Bentum, *Interactive Visualization of Volume Data*, PhD thesis University of Twente Enschede, The Netherlands, 1995.

[3] H. Bentoumi, P. Gautron, K. Bouatouch, *GPU-based volume rendering for medical imagery*, International journal of computer systems science and engineering **1**(1), 2007, pp. 36–42.

[4] J. Blinn, *Compositing: theory. Computer Graphics & Applications*, pages 83–87, 1994.

[5] M. Brady, K. Jung, H.T. Nguyen, *Two-phase perspective ray casting for interactive volume navigation*, Proceedings of the IEEE Visualization Conference, Phoenix, Arizona, pages 183-189, 1997.

[6] F. O. Boumghar, Y. Touat, D. Sarrut et S. Miguet, *Génération de DRR à l'aide d'un système hybride ShearWarp-Splatting pour la radiothérapie Conformationnelle*, Surgetica'2005, Centre des Congrès, Le Manège, Chambéry, Savoie, 19-21 Janvier 2005.

[7] B. Cabral, N. Cam, J. Foran, *Accelerated volume rendering and tomographic reconstruction using texture mapping hardware*, Proceedings of IEEE Symposium on Volume Visualization, pages 91- 98, 1994.

[8] R. L. Cook, K. E. Torrance, *A reflectance model for computer graphics*, ACM Transactions on Graphics, pages 7–24, 1982.

[9] T. Cullip et U. Neumann, *Accelerating volume reconstruction with 3D texture mapping hardware*, Technical Report TR93-027, Department of Computer Science, University of North Carolina, Chapel Hill, 1993.

[10] J. Danskin et P. Hanrahan, *Fast algorithms for volume ray tracing*, Workshop on Volume Visualization, édition A. Kaufman and W. L. Lorensen, pages 91–98, 1992.

[11] J. Dengler, S. Warfield, J. Zaers, C. Guttmann, W. Wells, G. Ettinger, J. Hiller, et R.Kikinis. *Automatic identification of grey matter structures from MRI to improve the segmentation of white matter lesions. In Proceedings of Medical Robotics and Computer Assisted Surgery*, pages 140–147, 1995.

[12] R. A. Drebin, L. Carpenter et P. Hanrahan, *Volume rendering, Computer Graphics*, pages 65–74, 1988.

[13] K. Engel, M. Kraus, et T. Ertl, *High quality pre-integrated volume rendering using hardware-accelerated pixel shading. In Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 9–16, 2001.

[14] K. Engel, M. Hadwinger, J. M. Kniss, C. Rezk-Salama et D. Weiskopf, *Real time volume graphics*, A. K. Peters Ltd., 2006.

[15] M. E. Goss, *An adjustable gradient filter for volume visualization image enhancement. In Graphics Interface* ’94, pages 67–74, 1994.

[16] J. T. Kajiya et B. P. V. Herzen, *Ray tracing volume densities, In Computer Graphics*, SIGGRAPH ’84 Proceedings, pages 165–174, 1984.

[17] A. Kaufman et K. Mueller, *Overview of Volume Rendering*, The Visualization Handbook, C. D. Hansen, C. R. Johnson, Elsevier Butterworth–Heinemann, page 127-174, 2005.

[18] K. Kreeger et A. Kaufman, *Mixing translucent polygons with volumes*, In Proceedings of IEEE Visualization ’99, pages 191–198, 1999.

[19] J. Krüger, R. Westermann, *Acceleration techniques for GPU-based volume rendering*, IEEE Visualization, pages 287–292, 2003.

[20] P. Lacroute, Fast *Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform*, PhD thesis, Stanford University, Departments of Electrical Engineering and Computer Science, 1995.

[21] M. Levoy, *Display of Surfaces from Volume Data*, Ph.D. dissertation, University of North Carolina at Chapel Hill, 1989.

[22] B. Lichtenbelt, R. Crane et S. Naqvi, *Introduction to Volume Rendering*, Los Angeles, Prentice Hall, 1998.

[23] N. Max, *Optical models for direct volume rendering*, IEEE Transactions on Visualization and Computer Graphics, pages 99–108, 1995.

[24] T. Möller, R. Machiraju, K. Mueller, et R. Yagel, *A comparison of normal estimation schemes*, In Proceedings of IEEE Visualization ’97, pages 19–26, 1997.

[25] U. Neumann, *Volume Reconstruction and Parallel Rendering Algorithms, A Comparative Analysis*, PhD thesis, University of North Carolina at Chapel Hill, 1993.

[26] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, et L. Seiler, *The Volume Pro real-time ray casting system*, In Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’99), pages 251–260, 1999.

[27] B. T. Phong, *Illumination for computer generated pictures,* Communications of the ACM, pages 311–317, 1975.

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

[29] R. J. Rost, *OpenGL Shading Language*, Addison Wesley Professional, January 25, 2006.

[30] P. Sabella, *A rendering algorithm for visualizing 3D scalar fields*, Computer Graphics (SIGGRAPH ’88 Proceedings), Vol. 22, Atlanta, pages 51–58, 1988.

[31] J. Schulze, M. Kraus, U. Lang et T. Ertl, *Integrating pre-integration into the shear-warp algorithm,* In Proceedings of the Third International Workshop on Volume Graphics, pages 109–118, 2003.

[32] M. Sharghi, I.W. Ricketts, *A novel method for accelerating the visualization process used in virtual colonoscopy*, Information Visualization 2001, San Diego, California, pages 167–172, 2001.

[33] S. Stegmaier, M. Strengert, T. Klein et T. Ertl, *A simple and flexible volume rendering framework for graphics-hardware based ray casting*, Fourth International Workshop on Volume Graphics, pages 187–241, 2005.

[34] U. Tiede, T. Schiemann, et K. Höhne, *High quality rendering of attributed volume data,* In Proceedings of IEEE Visualization, pages 255–262, 1998.

[35] C. Upson, Jr. T. Faulhaber, D. Kamins, D. Laidlaw,D. Schlegel, J. Vroom, R. Gurwitz et V. A. Dam, *The Application Visualization System: A computational environment for scientific visualization*, IEEE Computer Graphics & Applications, pages 30–42, 1989.

[36] A. Vilanova, E. Groller, A. Kong, *Cylindrical approximation of tubular organs for virtual endoscopy*, Technical Report TR-186-2-00-02, 2000.

[37] W. Wells, P. Viola, H. Atsumi, S. Nakajima et R. Kikinis, *Multi-modal volume registration by maximization of mutual information. Medical Image Analysis*, pages 35–51, 1996.

[38] R. Westermann et T. Ertl, *Efficiently using graphics hardware in volume rendering applications*, SIGGRAPH'98, pages 169–178, 1998.

[39] L. Westover, *Interactive volume rendering, Chapel Hill Workshop on Volume Visualization*, Chapel Hill, NC, pages 9–16, 1989.

[40] P. L. Williams et N. Max, *A volume density optical model. Workshop on Volume Visualization*, pages 61–68, 1992.

[41] C. Wittenbrink, T. Malzbender et M. Goss, *Opacity-weighted color interpolation for volume sampling. In Symposium on Volume Visualization*, pages 135–142, 1998.

[42] Y. Wu, V. Bhatia, H. C. Lauer, et L. Seiler, *Shear-image order ray-casting volume rendering, In Symposium on Interactive 3D Graphics*, pages 152–162, 2003.

[43] F. Yuan, *Real time multiple planar volume clipping based on programmable graphics process unit*, School of information technology, Jiangxi University of Finance and Economics, Nanchang 330013, Jiangxi, China, 2009.

[44] http://www.lighthouse3d.com/opengl/glsl/

[45] http://nehe.gamedev.net/data/articles/article.asp?article=21

[46] http://www.opengl.org/documentation/glsl/

[47] http://www.siteduzero.com/tutoriel-3-8894-les-shaders-en-glsl.html

[48] P.Triers, *GPU ray casting tutorial*, http://www.daimi.au.dk/~trier/?page_id=98.