Un environnement pour le tracé de rayons utilisant une modélisation par arbre de construction

Marc Roelens

22 avril 1993

Chapitre 1

Introduction

1.1 Généralités

Le but de la synthèse d'images est de convertir une description symbolique d'un certain environnement, ce que l'on appelle une *scène* en une image, c'est-à-dire un ensemble de valeurs numériques susceptibles d'être affichées sur un moniteur vidéo ou d'être imprimées sur une imprimante couleur.

On peut décomposer cette conversion en trois étapes :

- la *modélisation*, qui concerne donc les méthodes de description d'une scène. Cette description doit faire connaître tous les éléments de la scène (géométrie et propriétés colorimétriques des objets, sources lumineuses, points de vue, direction de visée....).
- l'*élimination des parties cachées*, qui détermine en un point de l'image l'objet (ou la liste d'objets dans le cas d'objets transparents) vu en ce point.
- le *rendu*, qui permet de calculer la "couleur" de l'objet vu en un point de l'image.

Les travaux que nous allons présenter dans cette thèse recouvrent ces trois domaines de la synthèse d'image, aussi nous allons rapidement les présenter.

1.2 Modélisation

Le domaine de la modélisation peut recouvrir deux domaines encore bien distincts :

- la modélisation géométrique, qui permet de définir la forme des objets à représenter, ainsi que des données comme la position du point de vue, du point visé,
- la modélisation des couleurs des objets, qui définit les propriétés colorimétriques des objets.

Présentons brièvement chacun de ces deux domaines.

1.2.1 Modélisation géométrique

Il convient de prononcer ici une évidence : les objets réels sont extrêmement compliqués, et il est quasiment impossible d'en donner une description exacte. Le rôle de la modélisation géométrique est donc de permettre une approximation "aussi bonne que possible" desdits objets réels.

1.2.1.1 Modélisation à facettes

On peut considérer que les premières modélisations en synthèse d'images ont été faites à l'aide de facettes planes. Un objet est donc approximé par un ensemble de morceaux de surfaces planes, ces morceaux ayant parfois des contraintes sur leur nombre de sommets (triangles ou quadrilatères, par exemple).

On sait que, quitte à augmenter le nombre de facettes, on peut donner une approximation aussi fine que voulue par ce type de modèle. Cependant, pour représenter certains objets simples, comme une sphère par exemple, le nombre de facettes doit être important si l'on ne veut pas voir apparaître sur l'image finale des segments bien visibles sur la frontière apparente de la sphère.

De plus, on a parfois bien du mal à pouvoir reconnaître l'intérieur et l'extérieur de l'objet. Ceci peut poser des problèmes de cohérence des objets.

On a donc amélioré ce système en introduisant la notion de *B-rep* (pour *Boundary REPresentation*), c'est-à-dire la description par les contours. On associe alors à une description par facettes des notions topologiques comme l'adjacence de deux facettes en un sommet, en une arête de l'objet.

Au prix d'une complexification des structures de données décrivant la scène, on obtient une bien meilleure cohérence des objets. On est alors capable de résoudre les problèmes d'intérieur et d'extérieur des objets, ce qui permet par la suite de déterminer les caractéristiques volumiques des objets.

Enfin, afin de permettre des formes d'objets plus variées, on a étendu la notion de facette plane en notion de morceau de surface. Les surfaces peuvent être des morceaux de surfaces gauches, et on maintient dans la description de l'objet cette définition.

De nombreuses espèces de surfaces sont utilisées parmi lesquelles on peut citer les surfaces paramétriques dont chacune des composantes est définie par un polynôme de degré inférieur ou égal à 3 (on parle alors de surfaces cubiques). Ces surfaces peuvent être définies à l'aide de points particuliers appelés *points de contrôle*. Selon le type de surface, ces points appartiennent à la surface ou permettent simplement d'en contrôler la forme.

Notons que le problème de raccordement de ces morceaux de surface afin d'obtenir un volume (c'est-à-dire une surface fermée) est un problème non trivial.

Lorsque l'on a besoin de visualiser l'objet, il est alors possible de générer une description à facettes "au vol", qui est alors adaptée en fonction des besoins de précision, ou bien d'utiliser dans certains cas la définition paramétrique de ces surfaces, dans le cas du lancer de rayons en particulier.

1.2.1.2 Modélisation par construction

Dans le principe de modélisation que nous venons de décrire, la modélisation est explicite : toutes les données des facettes sont stockées dans la définition de l'objet. Le principe de la modélisation par construction consiste quant à lui à rendre une partie de l'information implicite.

Plus précisément, on utilise dans ce modèle des objets de base mais également des opérations de constructions sur ces objets. Ces opérations peuvent être des transformations affines ou non de ces objets, des opérations de combinaison booléenne entre ces objets. On a alors un principe de modélisation qui se rapproche de l'usinage des pièces mécaniques, où, à partir de diverses pièces obtenues par moulage ou forgeage, on construit de nouvelles pièces par perçage, fraisage, tournage ou soudure.

Ce principe permet aussi de respecter le caractère volumique des objets créés, dans la mesure où les objets de base sont eux-mêmes volumiques.

Les objets de base peuvent alors être des objets à facettes comme ceux définis précédemment, ou bien des entités plus implicites encore : on peut ainsi utiliser des cubes, des cônes, des sphères ou des cylindres en ne retenant de ces objets que leur description symbolique.

On appelle de tels modèles des modèles CSG (pour *Constructive Solid Geometry*).

1.2.2 Modélisation des couleurs

Si le domaine précédent a été extrêmement développé, le domaine de la modélisation des couleurs est en pleine extension actuellement. On voit apparaître régulièrement de nouvelles méthodes pour représenter les "couleurs" des objets. La couleur d'un objet est le plus souvent prise comme l'ensemble des propriétés de cet objet à interagir avec la lumière.

Nous ne ferons pas ici de liste exhaustive de tous les modèles existants, nous insistons simplement sur le fait qu'ils sont nombreux et extrêmement variés. Les systèmes de modélisation de couleurs doivent donc être les plus évolutifs possible.

1.3 Elimination des parties cachées

Nous allons présenter dans cette section quatre grandes familles d'algorithmes utilisés pour réaliser l'élimination des parties cachées. Encore une fois, cette liste n'est pas exhaustive et nous nous sommes restreints aux algorithmes les plus utilisés.

1.3.1 Algorithmes à affichage ordonné

Nous regroupons sous ce terme tout une famille d'algorithmes reposant sur des principes très voisins :

- on utilise une modélisation de la scène par facettes polygonales,
- on affiche les polygônes sur l'écran en respectant leur ordre d'affichage (les polygônes les plus lointains sont affichés les premiers)
- lorsque tous les polygônes ont été traités, l'image est terminée.

Cet algorithme très simple est celui de l'algorithme *du peintre*. Le problème est dû à l'ordre des polygônes à trouver. De plus, pour une même scène, si l'oeil change de place ou si le point visé change de place, l'ordre doit être entièrement recalculé. Des améliorations ont conduit à des algorithmes où l'ordre est calculé "au vol" en parcourant une structure de données qui partitionne les polygônes en zones disjointes : il suffit alors de parcourir les zones dans leur ordre de distance par rapport à un point de vue et un point visé donné. Ces algorithmes sont en particulier utilisés pour les simulateurs de vols, et ils permettent un affichage en temps réel (25 images par seconde) de plusieurs milliers de polygônes.

1.3.2 Algorithme à tampon de profondeur

Dans cet algorithme, la modélisation géométrique doit permettre la génération d'une description sous forme de facettes planes. On utilise un tableau de la taille de l'image, chaque case du tableau contenant la couleur courante de cet élément de l'image (un *pixel*, pour *PICTure ELement*), et la profondeur courante, c'est-à-dire la distance de l'oeil à l'objet le plus proche connu actuellement dans la direction considérée.

L'algorithme est alors le suivant :

```
initialiser le tableau avec la couleur du fond et une profondeur infinie
pour chaque facette de l'objet
        déterminer sur quels pixels de l'image se projette la facette
        pour chaque pixel de projection
                calculer la distance de l'objet à l'oeil en ce point
                si cette distance est inférieur à la distance courante
                        actualiser la distance
                        affecter au pixel la couleur calculée pour cet objet
                finsi
        finpour
finpour
```

Cet algorithme extrêmement simple possède l'avantage de pouvoir être réalisé par des processeurs spécialisés de conception assez simple. Il peut de plus être adapté pour permettre l'ajout d'objets transparents. Il se prête toutefois assez mal aux méthodes de construction même si des extensions ont été décrites.

1.3.3 Algorithmes à balayage de ligne

Les algorithmes de cette famille considèrent également une scène décrite par des facettes. Pour une ligne de l'écran, on intersecte les facettes avec le plan passant par l'oeil et cette ligne de l'écran. On obtient ainsi un ensemble de segments, correspondant aux facettes actives, c'est-à-dire se projettant sur cette ligne. On ordonne ensuite ces segments de façon à déterminer ceux qui sont les plus proches de l'écran pour un pixel donné. Si cette classification est impossible, on peut subdiviser les segments en sous-segments de façon à ce qu'ils soient ordonnables.

Cet algorithme est lui aussi assez simple, et permet l'extension à des modèles représentés par un arbre de construction. On peut également traiter les objets transparents.

De plus, il est facile de concevoir un algorithme parallèle basé sur ce principe puisque chaque ligne est indépendante de la précédente. D'autres extensions utilisent au contraire la cohérence entre deux lignes successives pour accélérer les calculs.

1.3.4 Tracé de rayon

Cet algorithme étant celui qui nous intéresse particulièrement dans ce rapport, nous le détaillons un peu plus précisément.

Le principe en est très simple puisqu'il essaie de reproduire le trajet de la lumière dans une scène, dans le sens inverse de parcours. On utilise donc un rayon lumineux partant de l'oeil et passant par un point de l'écran. On détermine alors quel est l'objet vu dans cette direction en réalisant l'intersection entre les objets de la scène et ce rayon, puis en les ordonnant par distance croissante à l'oeil.

Une fois ce premier point déterminé, on en calcule son éclairement en relancant des rayons (appelés *rayons d'ombre*) vers les sources lumineuses. On peut ainsi détecter les ombres portées sur les objets par d'autres objets.

Il est également possible que la surface de l'objet rencontré soit réfléchissante ou réfractante. L'algorithme relance alors d'autres rayons dans des directions déterminées par les lois physiques de réflexion ou de réfraction : ces rayons sont appelés *rayons secondaires*. Ces rayons secondaires peuvent à leur tour générer des rayons d'ombre ou d'autres rayons secondaires.

L'ensemble des rayons générés pour un seul point de l'image a donc une structure arborescente, et on parle alors d'*arbre des rayons*. Cet arbre est en principe infini et il convient de le tronquer, ce que l'on peut réaliser en limitant a priori la profondeur de cet arbre, ou en ne retenant que les rayons dans la contribution est supérieure à un certain seuil.

La première description d'un tel algorithme a été faite par Appel en 1968 [App68], avec une implantation dans un système de CAO-CFAO dès 1971 [GN71]. Cette première réalisation ne comporte aucun rayon d'ombre ou rayon secondaire, et on lui a donné le nom de *brute force algorithm* [SSS74]. Cet algorithme très simple nécessite d'énormes temps de calcul, aussi est-il pratiquement oublié jusqu'en 1980, date à laquelle Whitted présente des images d'un grand réalisme réalisée grâce à cette technique, incluant les phénomènes de réflexions multiples, de réfractions, d'ombres [Whi80]. Whitted est aussi l'un des premiers à utiliser des techniques d'accélération des calculs basées sur l'utilisation de hiérarchies de volumes englobant les objets de la scène.

Roth [Rot82] étend ensuite ce modèle aux scènes modélisées par des arbres de construction. On assiste ensuite à une véritable explosion de cette technique, avec des temps de calculs toujours importants, mais des effets permis de plus en plus nombreux. Citons parmi ceux-ci le tracé de rayons inverse [Arv86], qui applique le principe du tracé de rayons à partir des sources lumineuses et permet ainsi de "répartir" la lumière dans une scène. Ceci autorise ainsi les éclairages indirects par l'intermédiaire d'un miroir ou d'une lentille. Citons enfin la méthode du tracé de rayons distribué [CPC84], qui permet de réaliser des effets de flou, de surfaces non parfaitement réfléchissantes, de profondeur de champ, et résout une partie des problèmes d'aliassage.

1.4 Rendu et modèles d'éclairement

Une fois le problème de l'élimination des parties cachées résolu, il reste à calculer la couleur de l'objet vu. Cette couleur dépend de nombreux paramètres :

- propriétés de l'objet,
- éclairement direct par des sources lumineuses,
- éclairement indirect par réflexions lumineuses sur les autres objets de la scène.

Ces phénomènes sont extrêmement complexes, aussi est-on conduit la plupart du temps à des simplifications. Il convient de noter que le problème du rendu possède deux aspects pratiquement indépendants :

- modéliser le comportement de la lumière arrivant sur la surface d'un objet,
- modéliser les effets de lumière entre les sources de lumière et les différents objets,

On parlera dans le premier cas de modèle local du comportement de la lumière, et dans le second cas de modèle global.

1.4.1 Modèles locaux

Ici encore, nous allons introduire une distinction entre deux types de modèles locaux : les modèles empiriques et les modèles physiques.

1.4.1.1 Les modèles empiriques

Le modèle de Lambert Historiquement, le premier algorithme de rendu est celui du lissage plat (traduction de l'anglais *flat-shading*), qui utilise le modèle de réflexion de la lumière de Lambert. La couleur d'une facette plane est alors calculée grâce au calcul du cosinus de l'angle entre la normale à la surface et la direction d'illumination. Cette méthode produit toutefois des effets indésirables, en particulier le fait de marquer les passages entre facettes pour les objets obtenus par facétisation d'un objet à frontière gauche.

Le modèle de Gouraud Pour pallier cet inconvénient, Gouraud fut le premier à apporter une solution [Gou71]. Il convient de noter que ses travaux couvrent une partie de modélisation du rendu et une partie purement algorithmique. On associe à chaque sommet de chaque facette une normale, ce qui permet de calculer la couleur en ce point. Puis la couleur d'un point quelconque de la facette est obtenue par interpolation des couleurs des sommets de la facette. Cette technique permet alors d'obtenir des dégradés de couleurs. Un autre avantage est le fait que cet algorithme peut être implanté matériellement grâce à des structures très simples (additionneurs) : il existe donc de nombreux processeurs spécialisés effectuant ce lissage.

Le modèle de Phong Phong [Pho75] améliore encore la technique en interpolant directement les normales des sommets de la facette : ceci permet alors de représenter les reflets spéculaires, c'est-à-dire des endroits où l'intensité est très forte du fait de la symétrie de la direction de vue par rapport à la direction d'illumination.

Ce modèle permet de plus de moduler les reflets spéculaires, en introduisant un coefficient de spécularité qui précise la part de comportement spéculaire de la surface, et un indice spéculaire qui précise la taille des reflets spéculaires.

Le problème de l'interpolation des normales est un problème bien moins simple à implanter matériellement que l'interpolation des couleurs : un grand nombre de techniques d'approximation du modèle de Phong ont donc été proposés.

Le modèle de Whitted Whitted [Whi80] améliore encore le modèle de Phong en introduisant une composante supplémentaire qui est la composante de réfraction : Whitted est en effet le premier à pouvoir modéliser cet aspect particulier du comportement de la lumière grâce à l'introduction du tracé de rayons.

1.4.1.2 Les modèles physiques

Le modèle de Torrance-Sparrow Ce modèle fut d'abord introduit dans le domaine du calcul des transferts thermiques et des rayonnements [TS67]. C'est un modéle que l'on peut qualifier de *géométrique*. Les hypothèses de ce modèle sont en effet les suivantes :

- une surface rugueuse est constituée de micro-facettes;
- les normales à ces micro-facettes suivent une loi de probabilité uniforme en ce qui concerne l'angle de rotation autour de la normale moyenne à la surface, et une loi de probabilité à préciser pour l'angle d'écartement par rapport à cette normale moyenne (on utilise le plus souvent une loi gaussienne);
- chaque micro-facette est un miroir parfait;

Alors, on peut montrer que le coefficient de réflectance, déterminant la proportion d'énergie réémise par la surface après réflexion, est le produit de trois termes indépendants, le premier étant le coefficient de Fresnel du miroir parfait, le deuxième étant un coefficient lié à la distribution des micro-facettes et le dernier étant lié aux phénomènes de masquage et d'ombrage d'une micro-facette par les autres micro-facettes.

Le premier terme ne pose pas de problème car il peut être déterminé en connaissant les caractéristiques du matériau réfléchissant. Le deuxième est également simple à calculer grâce à la fonction de distribution des micro-facettes. Enfin. le troisième terme est celui qui pose le plus de problèmes. En effet, des hypothèses supplémentaires doivent être faites pour avoir la valeur de ce coefficient, et le calcul exposé dans [TS67] n'est pas des plus rigoureux.

Ce modèle fut pour la première fois utilisé en synthèse d'image par Blinn [Bli77] puis popularisé par la suite par Cook et Torrance [CT82]. Des méthodes sont également proposées pour déterminer le coefficient de Fresnel en ne connaissant sa valeur que pour une incidence normale. Dans [Str90], on propose un modèle plus “calculable”, qui est contrôlé par des paramètres plus intuitifs que dans le modèle initial.

Le modèle de Beckmann-Spizzichino A l'inverse du modèle précédent, qui est purement géométrique (distribution de facettes planes, coefficient de masquage entre facettes), les travaux de Beckmann et Spizzichino [BS63], poursuivis par Smith [Smi67] s'appuient sur l'optique physique, qui utilise la définition électromagnétique de la lumière. La surface réfléchissante est cette fois supposée obéir à une distribution de hauteurs, cette distribution étant de loi normale, de moyenne nulle, et définie par sa variance et son coefficient de corrélation.

Alors le champ électromagnétique après réflexion sur la surface est connu comme étant une solution d'une équation intégrale appelée intégrale de Kirschoff. En faisant de bonnes hypothèses sur la surface réfléchissante (irrégularités ayant un rayon de courbure important par rapport à la longueur d'onde) et sur l'onde incidente (polarisation parallèle ou normale), on peut exprimer la puissance réfléchie dans n'importe quelle direction.

Le modèle de He Dans [HTSG91], on propose un modèle unifiant pratiquement tous les modèles présentés. Ce modèle s'appuie sur des hypothèses assez simples, qui sont les suivantes :

- en tout point, la surface peut être remplacée par son plan tangent, ce qui signifie que les irrégularités sont de taille supérieures à la longueur d'onde,
- on ne considère que la lumière n'est réfléchie qu'une fois sur la surface (pas de réflexions multiples entre les facettes).

Alors, on observe que l'énergie réfléchie par la surface se décompose en trois phénomènes :

- un lobe diffus, représentatif des phénomènes de réflexion et de réfraction à l'intérieur de la surface. Ce lobe peut être modélisé par la loi de Lambert.
- un lobe spéculaire, représentatif de l'effet de réflexion unique. Ce lobe spéculaire est modélisé à l'aide du modèle de Torrance-Sparrow.
- un pic spéculaire, représentatif du comportement en miroir parfait de la surface. Il est modélisé grâce au modèle de Beckmann-Spizzichino.

Des coefficients de proportion permettent alors de varier les effets entre une surface parfaitement diffuse, pour laquelle seul le lobe diffus sera pris en compte, à un miroir parfait, pour lequel seul le pic spéculaire sera pris en compte.

1.4.2 Modèles globaux

Dans cette partie du modèle de rendu, il s'agit de déterminer quelles sont les énergies arrivant sur une surface. Les modèles locaux permettent alors de calculer l'énergie réémise dans une direction déterminée après réflexion sur cette surface.

1.4.2.1 Modèle simple

Les premiers algorithmes de rendu utilisaient simplement la définition des sources lumineuses pour calculer les énergies incidentes. On ne tenait alors aucun compte des phénomènes d'ombres portées, c'est-à-dire des masquages de la lumière par d'autres objets. Seule l'ombre propre était prise en compte, c'est-à-dire la position et l'orientation de la surface par rapport à la source.

Parmi les sources utilisées, on peut citer les sources de type soleil, caractérisées par une direction d'éclairement, ne subissant aucune atténuation, les sources ponctuelles émettant dans toutes les directions ou dans un nombre de directions privilégiées. Les phénomènes d'atténuation par rapport à la distance à la source ainsi que les phénomènes de sources colorées pouvaient également être pris en compte [War83].

1.4.2.2 Ombres et pénombre

Afin d'ajouter du réalisme aux images, on a cherché à rendre compte des phénomènes d'ombres portées. Différentes techniques ont été utilisées, parmi lesquelles on peut citer le double tampon de profondeur, la méthode des volumes d'ombres ou le tracé de rayons.

Ces méthodes avaient toutefois l'inconvénient de donner des ombres fortement marquées, dont les frontières étaient parfois trop apparentes. On a donc introduit la notion de pénombre, ou d'ombre partielle, en donnant un caractère surfacique aux sources de lumière ce qui permet de mieux représenter les ombres. Diverses techniques sont utilisées là encore comme un échantillonnage ponctuelle de la surface des sources lumineuses ou un calcul de volume de pénombre.

1.4.2.3 Interréflexions

Le phénomène le plus difficile à prendre en compte est celui des interréflexions, c'est-à-dire la possibilité pour tout élément de surface d'agir comme une source lumineuse vis-à-vis des autres éléments de surface.

Résoudre ce problème des interréflexions n'est pas du tout trivial, car déterminer en tout point l'énergie émise dans une direction particulière aboutit à la résolution d'une équation intégrale dont on ne connaît pas de solution analytique [BB89]. Dans les premiers algorithmes de rendu, on se contentait d'ajouter à l'éclairement en tout point, une composante dite *ambiante* qui représentait la quantité moyenne (et donc uniforme) d'énergie apportées par ces interrflexions.

On dispose à présent de meilleures solutions, bien que souvent partielles, et nous allons en présenter quelques unes.

Radiosité Cette technique, apparue en 1984 [GTGB84], consiste à calculer les interréflexions de lumière entre toutes les surfaces de la scène en décomposant ces surfaces en petits éléments pour lesquels on peut assurer que l'énergie émise est uniforme pour tous les points de la surface. On suppose de plus que les surfaces agissent comme des réflecteurs diffus parfaits, c'est-à-dire suivant la loi de Lambert pour la réflexion.

L'équation de conservation de l'énergie peut alors s'exprimer comme un système linéaire de taille N où N est le nombre de polygônes. Les coefficients de la matrice sont appelés *facteurs de forme*, et comme leur nom l'indique, ils ne dépendent que de la géométrie et de la position des polygônes les uns par rapport aux autres. Une fois ces coefficients calculés, pour tout éclairage et tout point de vue, il suffit de résoudre le système linéaire et d'effectuer un classique tampon de profondeur pour visualiser la scène. Le problème des temps de calcul se trouve alors reporté sur le calcul des facteurs de forme.

De nombreuses améliorations de cet algorithme ont été apportées. On peut citer parmi celles-ci l'introduction de calcul des facteurs de forme dit "de l'hémicube" [CG85], l'introduction de surfaces non parfaitement rugueuses [ICG86], une méthode de raffinements successifs qui évite de calculer la matrice complète des facteurs de forme [CCWG88].

Des extensions qui nous intéresse plus particulièrement ont introduit l'utilisation de facteurs de forme étendus (ou spéculaires) qui prennent en compte des lois de réflexions plus complexes que la loi de Lambert, et sont alors adaptées aux surfaces non parfaitement rugueuses. Un tracé de rayons est alors utilisé non pour la visualisation mais pour le calcul de ces facteurs de formes étendus. On aboutit alors aux méthodes *en deux passes*, la première passe étant un calcul de facteurs de formes étendus, la seconde étant un algorithme classique de visualisation (tampon de profondeur par exemple) [WCG87] [SP89].

Nous avons dans l'esprit l'idée de pouvoir calculer des facteurs de forme étendus avec le même tracé de rayons que celui utilisé pour la visualisation.

Tracé de rayons arrière Comme son nom l'indique, cette méthode [Arv86] consiste à utiliser un tracé de rayons non pas à partir de l'oeil, mais à partir des sources de lumière. On détermine ainsi les objets qui peuvent être atteints directements à partir d'une source lumineuse. La source peut d'ailleurs avoir des formes très variée : source ponctuelle, source surfacique ou soleil. De l'énergie est alors émise à partir de ces sources, puis les modèles locaux déterminent le comportement de cette énergie après avoir atteint le premier objet.

Si ce premier objet est de type diffus, on peut relancer un certain nombre de rayons dans toutes les directions à partir de ce point, ce qui revient à considérer ce point comme une source lumineuse ponctuelle secondaire. Si la surface est de type spéculaire, on ne relancera des rayons que dans un nombre de directions limitées. Des techniques d'échantillonnage stochastique des directions de réflexions peuvent être utilisées.

Ce processus de relancement des rayons s'arrête lorsque l'énergie véhiculée par un rayon tombe en dessous d'un certain seuil, ou lorsqu'un nombre maximal de réflexions sera atteint.

Il faut alors stocker pour chaque point (ou plutôt pour chaque surface) toutes les contributions obtenues de cette façon. Lors de la phase de visualisation (le plus souvent effectuée par un tracé de rayons conventionnel), on tient compte de l'éclairement précédemment calculé. On peut même éviter le relancement de rayons d'ombres, puisque les ombres sont déjà prises en compte par la première phase.

L'avantage de cette technique est la possibilité de représenter les éclairements indirects spéculaires, comme ceux obtenus après réflexion sur un miroir ou à travers un milieu réfringent (loupe ou épaisseur d'eau par exemple).

La technique de Ward Un inconvénient de la technique précédente provient du fait que l'on calcule souvent des contributions pour des surfaces qui n'interviendront pas dans la visualisation car elles seront cachées. La première phase est ainsi parfois inefficace.

Ward [WRC88] a proposé une solution basée sur le principe suivant :

- au début de l'algorithme, on ne précalcule aucune contribution et on utilise un tracé de rayons classique.
- lorsque l'on a déterminé un point d'intersection, on regarde si dans le voisinage de ce point, on a déjà calculé un éclairement dû aux interréflexions.
  - si oui, on interpole les valeurs d'éclairement calculés pour les points voisins.
  - si non, on lance un calcul de valeur d'éclairement en relançant dans toutes les directions des rayons.

On utilise ainsi le fait que sur deux points voisins d'une même surface, les éclairements sont voisins. Notons que les ombres ne sont pas prises en compte dans ce calcul d'éclairement indirect, mais dans la phase classique de rendu du tracé de rayons.

L'un des problèmes est de définir une bonne métrique pour déterminer les points voisins, et Ward propose à ce sujet une méthode tenant compte de la distance entre les deux points, du rayon de courbure de la surface en ces points, ainsi que la distance moyenne aux autres surfaces de la scène.

1.5 Principes généraux de notre modèle

Le point essentiel des travaux développés est le suivant : nous nous sommes efforcés de ne pas considérer l'algorithme de tracé de rayons comme un simple algorithme de visualisation de scènes. Nous avons essayé de l'envisager comme un outil, une méthode de simulation de phénomènes les plus variés possibles.

Ceci a alors deux conséquences très importantes :

- il est nécessaire de modéliser les phénomènes qu'un algorithme de type tracé de rayons est susceptible de simuler.

- un grand soin doit être apporté dans la phase de développement afin de pouvoir permettre un passage facile entre les différents phénomènes à simuler.

Le premier point concerne surtout une réflexion sur la modélisation, c'est-à-dire des travaux d'ordre théorique, alors que le second point est bien plus technique, car il est directement lié à l'implantation.

Nous avons d'ailleurs poussé ce second point à l'extrême puisqu'il est possible avec le système développé de simuler des phénomènes différents sur les mêmes scènes, en changeant simplement (et dynamiquement) certains paramètres de l'algorithme.

Ce second point peut être d'autre part lié à un souci permanent pour le développement de code, que l'on retrouve aussi bien dans les concepts des ateliers de génie logiciel que dans les langages à objets, à savoir :

- permettre une maintenance aisée du logiciel,
- permettre une évolution du logiciel,
- permettre la réutilisation du logiciel.

C'est par la généricité et la modularité que nous avons choisi de parvenir à ces buts.

Précisons maintenant à quels développements nous a conduit le premier point.

1.5.1 A quoi peut servir un tracé de rayons?

Cette question a été l'une des premières que nous nous sommes posée. Elle reprend la nécessité de modéliser les phénomènes simulables par un tracé de rayons. Il faut ainsi essayer de faire abstraction de l'utilisation usuelle du tracé de rayons, qui est la visualisation en images de synthèse.

Cette démarche, qui nous semble assez originale, nous a conduits à formuler la réponse suivante :

> Un algorithme de tracé de rayons permet de simuler tout phénomène de transfert d'une grandeur extensive au sein d'un environnement, ce transfert respectant les propriétés suivantes :
>
> - les transferts s'effectuent de point à point.
> - les transferts s'effectuent en ligne droite au sein d'un milieu homogène.
> - les transferts concernent des échanges de quantités finies (notion de quantum de transfert).
> - les transferts peuvent changer de direction uniquement aux frontières entre deux milieux homogènes, la quantité transportée pouvant quant à elle varier de plus au sein d'un milieu homogène.

Le calcul d'une image de synthèse remplit évidemment toutes les conditions énoncées ci-dessus, mais on peut également penser à des calculs d'échanges thermiques, ou bien des calculs de masse, de moment d'inertie.

1.5.2 Modélisation de l'environnement

Il convient alors de préciser un peu plus ce qu'est l'environnement au sein duquel nous allons simuler ces transferts.

1.5.3 Volume et surface

Nous avons déjà parlé de milieux homogènes. Il semble naturel que ces milieux homogènes soient représentés par des volumes, ayant si possible une certaine régularité topologique. Or, le laboratoire de l'Ecole des Mines de Saint-Etienne a développé depuis quelques années un modeleur de type arbre de construction. Cette méthode nous a semblé la plus adaptée pour représenter les volumes.

A chacun de ces volumes, nous pouvons associer les propriétés du milieu correspondant. Afin d'élargir la gammes des objets représentables, nous avons introduit la notion de texture, qui permet de modifier les propriétés d'un volume en fonction de la position. On parlera alors plutôt de volume homogène en loi. Nous avons aussi introduit la notion d'objet neutre, qui est susceptible de laisser passer l'énergie en en modifiant simplement l'intensité.

Cependant, en étudiant certains phénomènes physiques, nous nous sommes rendu compte que ces phénomènes faisaient intervenir des volumes extrêmement faibles, ce qui est le cas par exemple des comportements particuliers des zones frontières des volumes. On peut penser à des phénomènes de rugosité de surface, ou encore à l'effet de peau. Il est alors très difficile pour des raisons purement informatiques (précision de calcul d'un ordinateur) de représenter ces phénomènes par une description d'un volume particulier.

Nous avons alors introduit la notion de propriété de surface, qui permet de mieux intégrer ces effets particuliers sans introduire des objets de taille trop petite. Ces caractéristiques de surface sont bien sûrs attachées à un objet de type volume.

1.5.3.1 Géométrie et propriétés

Comme nous l'avons vu, un modèle comporte la description des formes des objets (ce que nous appelons la géométrie de la scène) ainsi que la description des propriétés de ces objets.

Il nous a paru essentiel de séparer autant que possible ces deux descriptions, toujours afin de permettre la plus grande évolutivité de notre modèle. Ainsi, toute propriété sera associée à un objet de la façon la plus neutre possible, par l'intermédiaire d'une clé (numéro ou chaîne de caractères par exemple) mais cette clé ne détermine en rien le type de la propriété, qui est connue de façon implicite.

Cette séparation facilite grandement la modélisation et le développement du code correspondant puisque les problèmes sont résolus séparément.

Nous constatons donc que les propriétés peuvent concerner n'importe quel objet, y compris des objets composites. Des problèmes de priorité apparaissent alors et nous avons trouvé une solution particulière, qui nous semble assez originale.

Notons également que nous avons introduit la notion d'objet non-colorable, qui est un objet dont on ne peut changer ultérieurement les propriétés.

1.5.3.2 Sources d'énergie

Nous avons également cherché à intégrer dans notre modèle deux phénomènes distincts de génération d'énergie :

- l'énergie générée par les objets du modèle (ce que nous appelons l'énergie propre),
- l'énergie générée par des objets extérieurs au modèle (ce que nous appelons les sources).

Si les objets de la seconde catégorie sont bien étudiés, il est bien plus rare de trouver dans les modèles actuels des objets de la première catégorie. Notons dès à présent que nous pensons utiliser cette notion pour des algorithmes basés sur les techniques de radiosité, où précisément la radiosité serait une des composantes à prendre en compte sous forme d'énergie propre.

Enfin, nous avons introduit une source particulière, nommée le *fond*, qui permet de représenter des phénomènes de ciel, ou de placage d'environnement.

1.5.4 Algorithme générique de rendu

Ainsi que nous l'avons déjà précisé, nous avons essayé de modéliser les phénomènes simulables par un tracé de rayons. Il est ainsi naturel de retrouver cette modélisation au niveau de l'algorithme de rendu.

Les rares exemples d'algorithmes génériques sont encore trop marqués par leur histoire, à savoir qu'ils sont issus directement de la synthèse d'image. Notre souci a réellement été d'essayer d'oublier cette origine.

Nous avons ainsi développé les notions suivantes :

- sources d'énergie, et générateur de rayons d'ombre associé,
- générateur de rayons primaires,
- générateur de rayons secondaires,
- fonction génériques de réflexion et de réfraction,
- intégration de l'énergie propre,
- gestion de l'énergie de fond,
- la possibilité de mixer certains objets (cas des primitives de lumière par exemple).

Là encore, la généricité simplifie le développement des algorithmes, et l'intégration aisée de phénomènes comme la réflexion imparfaite, la translucidité, la pénombre.

1.5.5 L'intersecteur

L'algorithme utilisé pour réaliser les intersections n'est pas le point le plus original de nos travaux, si ce n'est en ce qui concerne les deux points suivants :

- la prise en compte du caractère volumique des objets et la notion d'intervalle d'intersection,
- la prise en compte un peu particulière des objets transparents.

La technique de boîtes englobantes utilisée est elle tout à fait originale, et elle résout certains problèmes tout en en posant d'autres.

1.6 Plan

Nos travaux ont portés sur la chaîne complète de fabrication d'une image de synthèse, et nous allons donc détailler chacun des éléments de cette chaîne. Cette thèse est ainsi organisée en sept chapitres.

- le chapitre 1 est la présente introduction.
- le chapitre 2 présente le système que nous avons développé pour la modélisation géométrique.
- le chapitre 3 présente notre algorithme de tracé de rayons, ou plus exactement l'algorithme d'intersection entre un rayon et une scène.
- le chapitre 4 s'intéresse au problème de l'accélération des calculs d'intersection en utilisant une méthode à englobants originale.
- le chapitre 5 essaie d'établir un algorithme générique de rendu, et quelques implantations que nous avons réalisées en suivant ce modèle.

- le chapitre 6 présente quelques applications intéressantes permises par notre modèle. On y trouve l'utilisation de perspectives non traditionnelles, ainsi que l'intégration dans notre modèle de primitives particulières appelées *primitives de lumière*, et la visualisation de densités volumiques. Une technique de parallélisation originale est également présentée.
- enfin, le chapitre 7 décrit quelques extensions qui pourraient être apportées à notre modèle. Une description plus précise de l'implantation de nos travaux y est présentée, ainsi qu'une conclusion de ce rapport.

Chapitre 2

Un modèle CSG étendu

2.1 Les principes de base

2.1.1 Séparation géométrie/caractéristiques de rendu

Il apparaît de plus en plus nécessaire lorsque l'on écrit des algorithmes de rendu en synthèse d'images de pouvoir séparer les deux composantes que sont la géométrie du modèle d'une part, et les informations de rendu qui lui sont associées d'autre part.

Les avantages que l'on peut tirer d'une telle séparation sont les suivants :

– le processus de modélisation se trouve simplifié puisque les deux composantes de modélisation sont indépendantes l'une de l'autre : la géométrie n'a aucune connaissance du rendu et réciproquement.

– dans le cadre d'un algorithme de visualisation par lancer de rayons, cette séparation permet d'utiliser le même intersecteur géométrique pour des calculs de rendu différents. Ceci peut être très intéressant dans les phases de mise au point de scènes : on peut se contenter d'un modèle de rendu très simple pour générer les premières images, puis utiliser des modèles plus perfectionnés lorsque le debogage de la géométrie est réalisé.

– dans le cadre d'une parallélisation d'un algorithme de tracé de rayons, on peut ne paralléliser que la partie géométrique qui est de loin la plus gourmande en temps de calcul sur des scènes complexes. Ceci diminue donc la taille du code et également les données à répartir pour l'algorithme parallèle.

2.1.2 Graphe CSG

Une particularité importante des modèles CSG que nous utilisons est le fait que les objets peuvent être utilisés plusieurs fois quand ils ont été définis. Ceci simplifie grandement le processus de modélisation et permet par exemple la création de bibliothèques d'objets. Ceci entraine que le modèle n'est pas à proprement parler un arbre mais plutôt un graphe sans circuits. On parlera donc de **graphe CSG** plutôt que d'arbre CSG.

Ainsi, un nœud du graphe CSG ne représente pas forcément un objet unique de la scène. Il n'y a donc pas de position absolue de cet objet dans la scène puisque les positions peuvent être multiples et dépendent du chemin d'accès à ce nœud depuis la racine. Il y a donc un aspect local pour toutes les caractéristiques de cet objet.

Si l'on voulait utiliser le langage de l'approche objet, nous dirions que le nœud du graphe représente une classe, et que chaque chemin d'accès à ce nœud dans le graphe représente une instance de cette classe.

2.1.3 Volume et surface

Les objets que nous modélisons sont des objets volumiques. Ceci impose pour tout objet de pouvoir connaître l'intérieur et l'extérieur.

La surface d'un objet est simplement sa frontière (au sens topologique classique de $\mathbb{R}^3$). En général, la frontière sépare l'espace en deux composantes connexes (sauf pour quelques objets bizarres du type bouteille de Klein), et nous utilisons des conventions pour chaque objet qui déterminent l'intérieur de l'objet (et donc son extérieur).

Ceci n'interdit absolument pas l'utilisation d'objets purement surfaciques (polygône ou polyèdre). Ceux-ci peuvent en effet être considérés commes des objets volumiques à volume nul. Il n'est d'ailleurs nul besoin d'exiger que cet objet soit alors fermé. On verra que ceci à des influences sur les algorithmes de résolution des opérations booléennes.

2.1.4 Attributs

On appelle attribut toute propriété d'un objet qui n'est pas une propriété géométrique. On peut ainsi trouver sous ce terme général des informations aussi variées que la couleur, la densité, le spectre de réflectance,. . . .

Ces attributs sont utilisés par le processus de rendu. Il faut insister encore sur le fait que le terme de *rendu* est ici pris dans un sens très large : ce peut être un calcul d'image, un calcul de masse ou de moments d'inertie, un calcul d'éclairement ambiant ou de facteurs de forme spéculaires pour un algorithme de radiosité.

Tout attribut peut être affecté à n'importe quel nœud du graphe CSG, et non pas aux seules primitives de modélisation.

2.1.5 Objets neutres et actifs

Nous avons essayé d'intégrer dans notre modèle des objets d'un type particulier que nous appelons *objets neutres*. De tels objets ont des propriétés caractéristiques qui sont d'une part le fait que la lumière peut les traverser sans être déviée et d'autre part le fait que ces objets peuvent se superposer en un point de l'espace (leurs propriétés (attributs) sont alors combinées).

De tels objets permettent par exemple de représenter des objets transparents pour lesquels on veut négliger la réfraction, ou des primitives de lumière. Ces primitives de lumière permettent de modéliser des objets volumiques tels que les faisceaux lumineux générés par des sources lumineuses en atmosphère poussiéreuse ou dans le brouillard.

On peut noter que le fait d'être un objet neutre est en fait un attribut volumique et non une caractéristique géométrique. Il est cependant utile que l'intersecteur puisse reconnaître de tels objets car les algorithmes d'intersection sont alors différents.

2.2 Le modèle géométrique

Nous allons détailler ici les éléments constitutifs du modèle géométrique. Ceux-ci sont classiques en modélisation CSG, et sont constitués par les primitives, les transformations et les opérations booléennes.

2.2.1 Les primitives

Il faut ici distinguer deux choses :

- la primitive géométrique. Comme son nom l'indique, c'est une entité purement géométrique. Sa définition contient les éléments suivants : description de la géometrie, description de la structure englobante, type de la primitive et fonction spécifique d'intersection.

– le nœud-primitive. C'est en fait un nœud pendant du graphe CSG, qui contient en particulier une primitive géométrique. C'est sur ce nœud-primitive que l'on peut attacher des attributs et non sur la primitive géométrique.

La description de la géométrie peut être de type variable en fonction du type de la primitive : nous avons donc choisi de réserver un champ dans la structure de données représentant la primitive permettant de pointer sur une autre structure de données spécifique du type de la primitive.

2.2.1.1 Canonisation des primitives

Comme souvent en modélisation CSG, nous utilisons de nombreuses primitives par l'intermédiaire d'une définition canonique. Ainsi, un cube est toujours le volume de $\mathbb{R}^3$ constitué des points (x, y, z) tels que $0 \leq x \leq 1$, $0 \leq y \leq 1$, $0 \leq z \leq 1$. Ceci permet de n'avoir qu'une définition de primitive géométrique. Les nœud-primitives sont par contre définis pour chaque instance de cette primitive. Lorsque cette canonisation est impossible (cas du tore par exemple), une primitive géométrique est définie pour chaque instance.

2.2.1.2 Les primitives utilisées

Les primitives que nous utilisons sont actuellement les suivantes :

– cône unité, de sommet $(0, 0, 1)$, ayant comme base le cercle du plan $0xy$ centré en $(0, 0)$, de rayon 1, limité à $0 \leq z \leq 1$.

– cube unité, de côté 1, c'est-à-dire le volume de $\mathbb{R}^3$ $[0, 1]^3$

– cylindre unité, d'axe $0z$, de rayon 1, limité à $0 \leq z \leq 1$.

– cylindre hyperbolique d'axe $0z$, d'équation $x^2 - y^2 - 1 \leq 0$

– cylindre parabolique d'axe $0z$, d'équation $x^2 - y \leq 0$

– hyperboloïde de révolution à une nappe, d'équation $x^2 + y^2 - z^2 - 1 \leq 0$

– hyperboloïde de révolution à deux nappes, d'équation $x^2 + y^2 - z^2 + 1 \leq 0$

– objet nul

– paraboloïde de révolution d'axe $0z$ d'équation $x^2 + y^2 - z \leq 0$

– paraboloïde hyperbolique d'équation $x^2 - y^2 - z \leq 0$

– sphère de centre $(0, 0, 0)$ de rayon 1

– tore, généré par révolution autour de $0z$ d'un cercle dans le plan $0xz$ centré en $(1, 0)$ et de rayon r. On peut noter que si r est supérieur à 1, on retire avant révolution la partie du cercle comprise dans le demi-plan $x \leq 0$. Un tel tore n'a pas de surface intérieure.

Cette liste n'est pas limitative, mais elle représente ce qui est implanté actuellement. On peut noter que dans cette liste ne figurent actuellement pas les polygônes et les polyèdres.

2.2.2 Les transformations affines

Celles-ci sont des nœuds unaires du graphe CSG. Ici encore nous devons insister sur le fait que les transformations sont toujours définies dans un repère local.

Les transformations que nous utilisons sont :

- l'homothétie de rapport k
- l'affinité de rapports k_x, k_y, k_z
- la translation de vecteur (t_x, t_y, t_z)
- la rotation d'angle θ autour d'un axe (v_x, v_y, v_z) (l'axe de rotation passe par l'origine)
- la transformation quelconque de matrice $M \in \mathbb{M}_4(\mathbb{R})$

En fait, toutes les transformations sont stockées sous forme d'une matrice carrée 4×4, ce qui permet d'avoir une représentation uniforme pour toutes les transformations d'une part, pour les points et les vecteurs d'autre part. Comme ceci sera expliqué au chapitre 3, c'est la matrice de transformation inverse qui est stockée.

Notons cependant que ces matrices ont la particularité d'avoir la quatrième ligne dont les trois premiers termes sont nuls et dont le terme diagonal est égal à 1. Cette propriété doit aussi être vraie pour la matrice de transformation quelconque.

Notre modèle permet l'utilisation d'une transformation un peu particulière puisque cette transformation ne modifie pas l'objet auquel elle s'applique. Ceci est utile pour créer des copies d'objets qui ne changent pas de position mais dont on veut juste modifier la couleur ou bien y ajouter des textures. On évite de stocker une matrice unité en marquant ce nœud comme étant de type “sans transformation”. Nous appelons ce genre de nœud un nœud `NOP`.

2.2.3 Les opérations booléennes

Là encore, nous retrouvons les opérations classiques en modélisation CSG, à savoir l'union, la différence et l'intersection. On peut toutefois noter que notre modèle se restreint aux opérations binaires, qui sont plus simples à implanter. Ceci nécessite donc une binarisation des opérations booléennes que l'on manipule souvent comme opérations n-aires. Plusieurs méthodes sont possibles pour réaliser cette binarisation : méthode en râteau, équilibrage du graphe, ou bien utilisation des boîtes englobantes.

Notons aussi que pour éviter des incohérences topologiques, la différence est toujours réalisée avec l'intérieur topologique du second objet : on évite ainsi les problèmes de faces manquantes ou de faces trouant un volume. Les primitives étant toujours des fermés topologiques, les opérations booléennes ne génèrent elles aussi que des fermés.

Enfin, nous avons introduit une opération booléenne particulère que nous appelons *fusion*, qui sera détaillée dans le paragraphe sur les priorités entre attributs.

2.3 Les attributs

Les attributs sont toutes les informations non géométriques que l'on peut associer à un objet. Les attributs sont en fait divisés en deux classes et en deux types (soit en fait quatre espèces d'attributs).

2.3.1 Les deux classes d'attributs

La première distinction porte sur la notion d'attribut volumique ou surfacique. Un attribut *volumique* concerne uniquement l'intérieur d'un objet, alors qu'un attribut *surfacique* concerne la surface des objets.

On peut voir une utilité à cette distinction dans l'exemple suivant. Supposons que nous modélisions deux objets, l'un étant un cube en or, et l'autre étant un cube en acier recouvert d'un placage en or. Extérieurement, les deux objets sont identiques. Mais ceci n'est plus le cas si l'on perce un trou dans chacun des deux cubes : dans le premier cas (attribut volumique), l'intérieur du trou a également une surface en or, alors que dans le deuxième cas (attribut surfacique), on voit apparaître l'acier.

Ces notions sont importantes pour l'intersecteur qui traite différemment les deux classes.

2.3.2 Les deux types d'attributs

La deuxième distinction porte sur la notion de couleur ou de texture.

Par définition, une couleur est une propriété non géométrique d'un objet qui est constante pour un objet : tous les points de l'objet ont la même couleur. Encore une fois, l'accent doit être mis sur le fait que le mot couleur est pris dans un sens le plus large possible.

A l'inverse, une texture est une propriété qui peut varier pour un objet. La propriété peut dépendre de la position du point, mais la loi de variation est la même pour tout l'objet.

Une autre distinction intervient entre les couleurs et les textures. Un objet ne peut avoir au plus qu'une couleur, alors qu'il peut avoir des textures multiples. En particulier, sur un nœud du graphe CSG, on stocke une liste de textures appliquée à ce nœud. Il faut prendre garde à stocker ces listes dans leur ordre de définition, car l'ordre des texturations est significatif.

2.3.3 Les deux espèces de textures

Nous utilisons en fait deux espèces de textures.

2.3.3.1 Textures de couleur

Ce sont les textures permettant de définir localement la couleur. Nous nous restreignons à l'usage de textures que nous appelons 3-D, c'est à dire que la couleur est définie par la position du point. Dans le cas des textures plaquées, ceci oblige la fonction de texturation à retrouver les deux paramètres de placage.

Il faut noter que la couleur générée par la texture peut dépendre de la couleur préexistante : ceci est par exemple le cas lorsque que l'on définit une texture de peinture appliquée avec un pulvérisateur. Aux endroits non recouverts de peinture, on voit la couleur du support. Ceci implique que l'ordre d'application des textures est significatif.

D'autres textures au contraire ne dépendent pas de la couleur préexistante et ne dépendent que de leurs paramètres de définition. Ceci est par exemple le cas de la texture en damier 3-D, où la couleur peut prendre deux valeurs en fonction des valeurs de $E(x)$, $E(y)$ et $E(z)$, (x, y, z) étant les coordonnées du point de calcul et $E(.)$ désignant la fonction partie entière.

2.3.3.2 Textures de normale

En tout point de la surface d'un objet, on définit une normale extérieure à cet objet. On peut créer des effets visuels intéressants en perturbant ces normales par des fonctions de texturation. Ceci permet de générer très aisément des vagues, des aspects de métal poli ou érodé,....

Le fait que ces textures concernent la normale implique évidemment que ces textures sont des attributs de surface des objets.

Ici encore, on peut superposer des textures de normale (pour génerer des effets d'interférences de vagues par exemple), ce qui implique aussi que l'ordre de texturation est significatif.

2.3.4 Priorité et non-colorabilité

Puisque les attributs peuvent être associés à n'importe quel nœud du graphe, il est naturel que des problèmes de priorité entre ces attributs apparaissent.

Le premier problème, que l'on pourrait qualifier de *priorité verticale* est le suivant : si un objet défini avec un attribut, se voit utilisé dans un autre objet auquel on affecte un attribut de même nature, quel est alors le *bon* attribut à utiliser ?

Le deuxième problème, que l'on pourrait qualifier de *priorité horizontale* est le suivant : si l'on définit une opération booléenne entre deux objets portant chacun des attributs, quels sont les *bons* attributs à utiliser pour l'objet résultant ?

Nous allons détailler les règles que notre modèle utilise pour résoudre ces problèmes.

2.3.4.1 Priorité verticale

La règle peut se décomposer en deux sous-règles en fonction du type des attributs.

- pour la couleur, on retient la définition la plus *récente*, c'est-à-dire la plus haute dans le graphe. Ceci est en effet logique : si on peint complètement un objet, la couleur sous-jacente disparaît.

- pour les textures, elles sont simplement superposées en respectant leur ordre (on doit d'abord appliquer la texture la plus basse dans le graphe).

2.3.4.2 Priorité horizontale

Ici, c'est la classe des attributs qui va déterminer les règles de priorité.

En ce qui concerne les attributs surfaciques, la règle est simple : un morceau de surface de l'objet résultat hérite des attributs surfaciques de l'objet initial auquel il appartient. En un point où les deux surfaces sont confondues, on en choisit arbitrairement une. Cette règle de priorité ne résoud donc pas le problème des faces coplanaires.

Pour les attributs volumiques, la règle est plus compliquée car elle dépend du type des objets. Plus précisément,

- les objets actifs sont prioritaires par rapport aux objets neutres. En particulier, un point de l'espace appartient soit à zéro ou un objet actif, soit à un ou plusieurs objets neutres. Tout point situé simultanément dans un objet neutre et un objet actif est considéré comme appartenant à l'objet actif.

- un point appartenant à l'intérieur de l'objet résultat d'une opération booléenne hérite des attributs volumiques de l'objet initial auquel il appartient. Si ce point appartient à l'intersection des deux objets, il hérite des attributs volumiques de l'opérande "cité en premier", sauf indication contraire de l'utilisateur.

Ces règles ont les conséquences suivantes :

- un objet neutre ne peut "trouer" un objet actif.

- lors d'une différence, les attributs volumiques de l'objet retranché sont inutiles, par contre ses attributs surfaciques seront affectés aux morceaux de surface générés par cette différence.

Rappelons que le problème de priorité horizontale pour une union de deux objets a été résolu par Wyvill en définissant la notion d'opération booléenne régulière [WS88]. Ainsi, la réunion régulière entre deux objets A et B est définie par

$$A \uplus B = (A \backslash B) \cup B$$

ce qui revient à dire que dans une telle opération, l'objet B est prioritaire par rapport à l'objet A. Notre méthode est donc assez similaire.

Nous avons toutefois apporté une amélioration importante par rapport à la technique de Wyvill en définissant la notion de *fusion*, présentée dans le paragraphe sur les opérations booléennes. Une fusion est simplement une union de deux objets dont l'utilisateur sait par avance qu'ils ont les mêmes propriétés. Il n'y a alors pas d'ambigüité sur les propriétés de l'objet résultat, puisque quelle que soit la méthode de résolution du conflit, le résultat sera identique.

Il existe un effet supplémentaire qui est la disparition des morceaux de surface internes. Ceci est particulièrement utile lorsque les objets sont des objets réfringents (en verre par exemple) car on évite ainsi de générer une réfraction à la frontière entre les deux objets (que ce soit sur la surface de l'un ou de l'autre) qui n'existe pas dans la réalité. Ceci est intéressant dans la mesure où l'on peut définir des primitives sans attributs et les utiliser dans des opérations boolénnes : on construit ainsi des objets qui sont de pures définitions géométriques. Il suffit alors de donner un attribut à l'objet ainsi crée pour que toutes les composantes de l'objet héritent de cet attribut.

Dans notre modèle, nous n'avons pas créé de structure de données spécifique pour la fusion : la fusion est simplement une union avec un drapeau particulier qui indique cette particularité. Notons dès à présent que ce drapeau est utilisé par l'intersecteur pour effectuer correctement les opérations booléennes.

Enfin, afin de faciliter le travail de l'utilisateur, notre modèle peut détecter les fusions implicites, c'est-à-dire les unions entre objets ne comportant aucun attribut. Il est cependant beaucoup plus difficile de détecter certains cas de fusion : ceci est par exemple le cas lorsque l'on réalise une union entre deux objets, que ces objets sont "hétérogènes" (comportent des objets avec des attributs différents) mais que les parties effectivement communes à ces deux objets possèdent les mêmes attributs. Notre système accepte donc l'aide de l'utilisateur mais en lui faisant confiance : une erreur dans une définition de fusion peut donner des résultats visuels surprenants !

2.3.4.3 Non-colorabilité

Notre système prévoit cependant une exception aux règles de priorité verticale énoncées ci-dessus. Pour certains objets, on peut vouloir interdire toute redéfinition ultérieure d'attributs (ce qui est le cas pour des objets de bibliothèque par exemple).

Nous avons donc défini la notion de *non-colorabilité*. Un objet non-colorable est tout simplement un objet qui ignorera toutes les redéfinitions ultérieures d'attribut. Un drapeau particulier sur chaque objet permet de marquer de tels objets, drapeau dont l'algorithme d'intersection doit tenir compte pour affecter les bons attributs aux objets.

2.3.5 Attributs ultérieurs

Dans le paragraphe sur la réutilisation des objets, nous nous sommes limités à l'aspect géométrique. Un objet peut être utilisé plusieurs fois en le déplaçant, éventuellement en lui affectant de nouveaux attributs.

Il y a cependant un cas où ce modèle est insuffisant : supposons que nous voulions modéliser des voitures de même type mais avec des couleurs de carrosserie et d'intérieur différents. Si nous redéfinissons la couleur pour chaque voiture générée, il n'y a aucun moyen d'affecter deux couleurs différentes à la carrosserie et à l'intérieur.

Nous avons donc introduit dans notre modèle la notion d'attribut ultérieur. Un tel attribut comporte un nom et éventuellement une valeur par défaut. Si un objet comportant un attribut ultérieur est utilisé dans un sous-graphe sur lequel on définit un attribut portant le nom de l'attribut ultérieur, on remplace l'attribut ultérieur par l'attribut de même nom. Dans le cas contraire, on utilise la valeur par défaut (si elle existe).

Dans le cadre d'un outil de modélisation interactif, on pourrait alors prévoir des outils d'interrogation permettant de connaître pour un objet donné les attributs ultérieurs n'ayant pas encore de valeur.

Si l'on veut faire une nouvelle utilisation du langage de l'approche objet, on peut dire que notre système permet de définir des classes d'objets (les nœuds), et que chaque chemin d'accès à ce nœud est une instance, les transformations géométriques instanciant la dimension, la position et l'orientation de l'objet, les attributs instanciant les attributs ultérieurs de l'objet.

2.3.6 Objets homogènes

Nous avons déja présenté la notion de fusion dans le paragraphe des opérations booléennes. Nous allons maintenant généraliser quelque peu cette notion en introduisant la notion d'*objet homogène*. Par définition, un objet homogène est un objet constitué de sous-objets qui ont tous les mêmes attributs (éventuellement aucun).

Il est alors facile de déterminer récursivement l'homogénéité des objets par les règles suivantes :

- un nœud de type primitive est toujours homogène.
- un nœud de type transformation affine est homogène si son opérande est homogène et ne contient pas d'attributs.
- un nœud de type opération booléenne est homogène si ses deux opérandes sont homogènes et ne contiennent pas d'attributs.

On peut remarquer que tout objet de type union que l'on peut qualifier d'homogène peut être considéré comme une fusion implicite. Ceci évite donc à l'utilisateur de préciser que ce sont des fusions.

Enfin, si l'on étudie bien la position des objets homogènes dans le graphe CSG, on s'aperçoit que ceux-ci occupent des sous-graphes complets, c'est-à-dire contenant tous les objets issus d'un des sommets du graphe. Ceci sera utilisé pour la numérotation des objets.

2.4 Nommage et numérotation

Ces deux techniques sont utilisées afin de pouvoir désigner des objets. Elles ont entre elles une distinction très importante : le *nom* d'un objet est une propriété locale (tout objet peut avoir un nom, et être utilisé dans d'autres objets en conservant ce nom), alors que le numéro d'objet est lui une propriété globale (un objet utilisé deux fois aura deux numéros différents).

Ceci implique en particulier que le nom peut être stocké directement sur l'objet que l'on nomme, alors que le numéro n'est qu'une valeur implicite. On peut dire que le nom est une propriété de la classe d'objet alors que le numéro est une propriété de chaque instance de la classe.

2.4.1 Nommage

Notre système permet d'associer un nom à tout objet. Ceci permet de réutiliser les objets, en faisant simplement référence à leur nom. On peut d'ailleurs noter que c'est le seul moyen que nous autorisons pour réutiliser les objets créés. Le nommage facilite ainsi grandement l'écriture de bibliothèques d'objets, et permet donc une meilleure mise au point des scènes.

Le nommage présente un deuxième avantage : puisqu'un objet ne peut être accédé que par son nom, on peut contrôler le nombre de références qui sont faites à chaque objet. A cet effet, chaque fois que l'utilisateur demande quel est l'objet dont il nous passe le nom en paramètre, le système vérifie bien sûr que cet objet est connu, auquel cas il incrémente le nombre de références qui sont faites à cet objet.

Ce nombre de références est une donnée très importante dans l'algorithme de compactage présenté ci-après. Il permet d'autoriser ou d'interdire des modifications physiques sur l'objet nommé : en effet, un objet partagé ne doit pas être modifié.

On peut également noter que le nommage est facultatif. Ceci évite d'avoir à nommer d'éventuels objets temporaires générés par exemple en appliquant des transformations successives à un objet (voir ci-après). Un objet non nommé (ou *anonyme*) ne peut alors être utilisé que par son créateur et ne peut donc être réutilisé.

2.4.2 Numérotation

La numérotation des objets repose sur la notion d'objet homogène que nous avons définie précédemment.

Nous avions remarqué la structure particulière des objets homogènes dans le graphe. Cette structure va nous servir à définir quels sont les couples (objet, chemin d'accès) que l'on va numéroter.

Par définition, un couple (objet, chemin) est numérotable si et seulement si l'objet est homogène et si le nœud situé immédiatement au-dessus de l'objet relativement au chemin considéré ne l'est pas.

Ceci peut intervenir si l'objet possède des attributs, ou s'il intervient dans une opération booléenne dont l'autre opérande est inhomogène ou contient des attributs.

Enfin, le numéro attribué à un objet numérotable est simplement le numéro d'ordre (commençant à 1) lorsque l'on parcourt le graphe en parcourant d'abord l'opérande gauche des opérations booléennes.

Par souci d'efficacité, on conserve sur chaque nœud inhomogène le nombre d'objets numérotables contenus dans son ou ses opérandes. De même, un drapeau permet de marquer les nœuds que l'on a déjà numérotés.

On peut alors écrire ainsi la fonction de numérotation :

```
int numerote ( objet )
{
        if (objet.homogene==VRAI)
                return 1;
        if (objet.numerote==FAUX) {
                objet.numerote=VRAI;
                switch (objet.type) {
                        case TRANSFO_AFFINE:
                          objet.nb_objets_1=numerote(objet.operande);
                          objet.nb_objets_2=0;
                          break;
                        case BOOLEENNE:
                          objet.nb_objets_1=numerote(objet.operande_1);
                          objet.nb_objets_2=numerote(objet.operande_2);
                          break;
                        }
                }
        return objet.nb_objets_1 + objet.nb_objets_2;
}
```

Il est également très simple de retrouver un objet connu par son numéro ainsi que le chemin qui amène à cet objet. Ceci peut être utile par exemple dans le cas d'une mise au point interactive d'une scène. On peut, pour désigner un objet, cliquer sur l'objet à l'écran. On lance alors un rayon passant par ce point de l'écran et on détermine les intersections avec la scène en demandant comme seule information le numéro de l'objet intersecté. Retrouver l'objet correspondant dans le graphe est alors possible.

Le pseudocode de cette fonction de recherche est lui aussi très simple et peut être écrit comme suit :

```
trouve_objet_par_son_numero ( numero )
{
        objet=racine;
        chemin=NULL;
        num=numero;
        while (1) {
                if (num==0)
                        break;
                if (num > (objet.nb_objets_1 + objet.nb_objets_2)) {
                        erreur("pas d'objet avec ce numero");
                        chemin=NULL;
                        break;
                        }
                empiler_dans_chemin(objet);
                switch (objet.type) {
                        case TRANSFO_AFFINE:
                          objet=objet.operande_1;
                          break;
                        case BOOLEENNE:
                          if (numero <= objet.nb_objets_1)
                                objet=objet.operande1;
                          else {
                                num-=objet.nb_objets_1;
                                objet=objet.operande_2;
                                }
                        }
                }
}
```

A la fin de cet algorithme, s'il n'y a pas eu d'erreur, `objet` contient l'objet désiré et `chemin` contient la pile des nœuds à parcourir pour parvenir à la racine.

On peut légitimement s'interroger sur la définition de la numérotation que nous avons retenue. Elle se justifie en fait par son usage, qui est restreint au problème d'antialiassage.

Lorsque l'on calcule une image en tracé de rayons, une technique souvent utilisée pour résoudre les problèmes d'antialiassage est le suréchantillonage adaptatif : si deux pixels voisins intersectent deux objets différents, on relance un rayon entre ces deux pixels.

Tout le problème dans notre cas réside dans la notion d'*objets différents*. Puisqu'un objet n'est pas nécessairement unique, comparer les primitives intersectées pourrait conduire à certaines erreurs puisque la même primitive peut être atteinte par plusieurs chemins : il faut donc tenir compte du chemin d'accès dans le graphe. On peut donc comparer les attributs des deux objets intersectés. S'ils sont différents, il est vraisemblablement nécessaire de relancer un rayon. Or justement, avoir des attributs différents signifie simplement avoir des numéros d'objet intersecté différents.

Notons d'ailleurs que la comparaison des attributs n'est peut-être pas suffisante pour décider de relancer des rayons. En utilisant des tests statistiques sur plusieurs critères, Jean-Luc Maillot a développé des techniques de raffinements successifs d'image adaptées à notre système de tracé de rayons [MCP92].

Enfin, notre système admet une autre possibilité de numérotation qui numérote simplement les primitives. Par rapport aux deux fonctions décrites ci-dessus, seul le test d'arrêt des algorithmes est différent : au lieu de tester l'homogénéité des objets, il suffit de tester si l'objet est de type nœud -primitive. Il est en outre possible d'utiliser simultanément les deux types de numérotation.

2.5 Compactage

La réutilisation des objets permet déjà un gain de place mémoire pour stocker une scène. Nous avons utilisé une autre méthode afin de gagner encore de la place.

Lorsque l'on utilise une modélisation CSG avec primitives canoniques, on utilise beaucoup de transformations affines. Par exemple, pour définir un parallélépipède, on part d'un cube, auquel on applique tout d'abord une affinité pour lui donner des dimensions convenables, puis on effectue une rotation pour lui donner une orientation convenable, enfin on lui applique une translation pour lui donner sa position définitive.

Le processus est le même lorsque l'on utilise un objet défini en bibliothèque : il y a peu de chances qu'il ait la position et l'orientation voulue par l'utilisateur.

On peut alors économiser de la place on comprimant ces transformations pour n'en garder qu'une seule. Ceci explique aussi pourquoi on stocke les transformations sous une forme banalisée : ceci permet cette compression par simple multiplication des matrices.

De façon plus générale, on peut essayer de ne pas créer un nœud du graphe CSG pour toute transformation. La création d'un tel nœud n'est rendue nécessaire que par la nécessité de pouvoir associer à ce nœud des attributs.

C'est pour ces diverses considérations que nous avons ajouté la possibilité de stocker dans chaque nœud une ou plusieurs transformations affines. Plus précisément,

- tout nœud du graphe peut contenir une matrice de transformation affine qui s'applique globalement à tout le sous-graphe CSG issu de ce nœud (une telle matrice est appelée *matrice globale*).
- les opérations booléennes peuvent contenir une ou deux matrices de transformation qui s'appliquent globalement à l'un ou à l'autre des opérandes de l'opération (de telles matrices sont appelées *matrices d'opérande*).

Il faut cependant prendre garde en effectuant ce compactage aux problèmes suivants :

- il ne faut pas modifier d'objet partagé
- il faut respecter les couleurs et l'ordre des textures
- il faut respecter les repères de texturation des textures existantes

De plus, pour améliorer l'efficacité de l'intersecteur, l'algorithme de compactage retire les attributs réputés inutiles, c'est-à-dire ceux situés immédiatemment au-dessus de nœud non-colorables. Ceci ne permet pas de retirer tous les attributs inutiles mais économise cependant une certaine place mémoire.

Enfin, pour permettre une plus grande souplesse, notre système autorise le compactage d'objets anonymes mais également le compactage d'objets qui ne sont référencés qu'une seule fois. Il faut cependant prendre garde au fait qu'un objet monoréférencé a un nom, et il faut donc retirer ce nom de la table des objets connus par le système si l'on ne veut pas générer d'erreur ultérieure.

Nous allons maintenant détailler l'algorithme de compactage. Au début de l'algorithme, aucun nœud ne contient de matrice globale. De même, aucune opération booléenne ne contient de matrice d'opérande.

L'algorithme procède de manière récursive à partir de la racine du graphe.

Les invariants de l'algorithme de compactage sont les suivants :

- un nœud de type transformation ne contient de matrice globale que si ce nœud contient des textures (sinon, cette matrice globale pourrait être combinée avec la matrice de transformation).
- un nœud de type opération booléenne ne contient de matrice globale que si ce nœud contient des textures, ou bien s'il ne contient pas de matrices d'opérande (il est alors plus économique de stocker une matrice globale que deux matrices d'opérande).

Enfin, les notations suivantes sont utilisées :

- si N désigne un nœud, on note $N.M_G$ sa matrice globale.
- si N est un nœud de type transformation, on note $N.M_T$ la matrice de transformation et $N.OP$ l'objet auquel s'applique cette transformation.
- si N est un nœud de type opération booléenne, on note $N.OP_1$ et $N.OP_2$ ses deux opérandes et $N.M_{OP_1}$ et $N.M_{OP_2}$ les matrices d'opérande.

2.5.1 Cas de la transformation affine (ou du nœud NOP)

L'algorithme peut se décomposer comme suit :

compacter $N.OP$
si $N.OP$ est multiréférencé ou si N et $N.OP$ contiennent tous deux des textures
    ne rien faire.
    STOP.
si $N.OP$ contient des textures
    règle 1
    faire $N.OP.M_G = N.M_T * N.OP.M_G$.
sinon si $N.OP$ est un nœud de type transformation ou NOP
    règle 2
    faire $N.M_T = N.M_T * N.OP.M_T$.
    faire $N.OP = N.OP.OP$.
sinon si $N.OP$ est un nœud d'opération booléenne
    si $N.OP$ ne contient pas de matrices d'opérande
    et si N ne contient pas de textures
        règle 3
        faire $N.OP.M_G = N.M_T * N.OP.M_G$
    sinon si $N.OP$ contient des matrices d'opérande
        règle 4
        faire $N.OP.M_{OP_1} = N.M_T * N.OP.M_{OP_1}$
        faire $N.OP.M_{OP_2} = N.M_T * N.OP.M_{OP_2}$
    sinon (N contient des textures)
        règle 5
        faire $N.OP.M_{OP_1} = N.M_T$
        faire $N.OP.M_{OP_2} = N.M_T$
remplacer N par $N.OP$.

2.5.2 Cas des opérations booléennes

Dans le cas des opérations booléennes, le seul compactage possible est l'utilisation des matrices d'opérande.

Les conditions qui rendent ce compactage possible sont ici plus strictes, puisque l'on ne peut effectuer ce compactage que si l'un des opérandes est une transformation affine, n'est pas multiréférencé et que de plus il ne contient pas de textures.

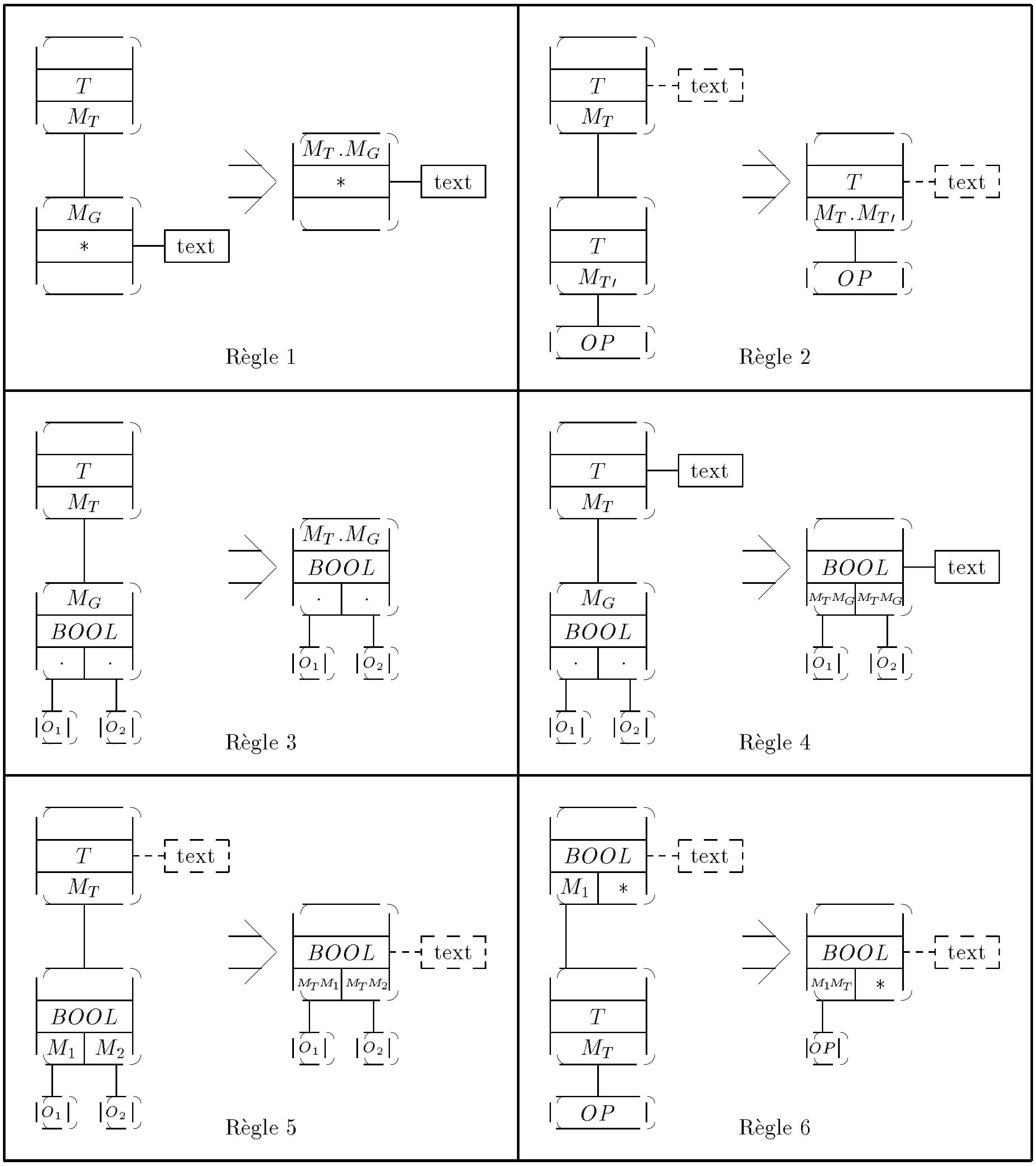


FIG. 2.1 – *Règles de compactage*

Plus précisément, l'algorithme peut se décomposer comme suit :

compacter $N.OP_1$
compacter $N.OP_2$
pour i variant de 1 à 2
  si $N.OP_i$ est un nœud de type transformation (ou NOP)
  et s'il ne contient pas de textures
    règle 6
    faire $N.M_{OP_i} = N.OP_i.M_T$
    remplacer $N.OP_i$ par $N.OP_i.OP$
  sinon
    ne rien faire.

La figure 2.1 récapitule les règles de compactage utilisées aussi bien pour les transformations affines que pour les opérations booléennes.

2.6 L'environnement *Illumines*

Illumines est le nom de l'environnement développé à l'Ecole des Mines de Saint-Etienne pour la synthèse d'images [MB89]. Le coeur de cet environnement est un modeleur basé sur la modélisation par arbre CSG. Son architecture générale est détaillée sur la figure 2.2. Dans ce schéma, les rectangles représentent des fichiers et les ovales des programmes. Précisons maintenant les composantes de ce système.

- `ca` est un facétiseur-perspectiveur-fenêtreur qui admet en entrée un fichier au format *CASTOR* et génére en sortie un arbre CSG de polyèdres. Il peut éventuellement résoudre les opérations booléennes en utilisant le module `resolver`. Ceci permet de ne générer en sortie que des unions, ce qui permet l'utilisation pour le rendu d'un z-buffer.

  Une autre possibilité est de générer un arbre contenant des différences ou des intersections et d'utiliser pour le rendu un algorithme d'Atherton.

  `ca` a été développé par Michel Beigbeder [Bei88] et le module `resolver` par Mohand Ourabah Benouamer [Ben89].

- Le tracé de rayons quant à lui n'a pas besoin du facétiseur puisqu'il traite directement les primitives de modélisation. Une première version de tracé de rayons (`yield`) a été implantée par Jacqueline Argence [Arg88] et améliorée par Gilles Fertey [Fer90], la deuxième version (`yart` pour *Yet Another Ray Tracer*) est la nouvelle implantation décrite dans cette thèse.

- `voir` est un algorithme de visualisation en fil de fer avec élimination des parties cachées et résolution des opérations booléennes utilisant des cartes planaires. Il a été écrit par Dominique Michelucci [Mic87].

2.6.1 Le langage *CASTOR*

Le langage *CASTOR* est un langage de définition de scènes modélisées à l'aide de graphes CSG [Bei88]. Il permet de définir les objets et les attributs affectés à ces objets, les paramètres de vue (position de l'oeil, du point de vue,...) et un certain nombre d'informations non géométriques (comme la position des sources lumineuses, leur intensité, les paramètres de calcul de la couleur du fond,...).

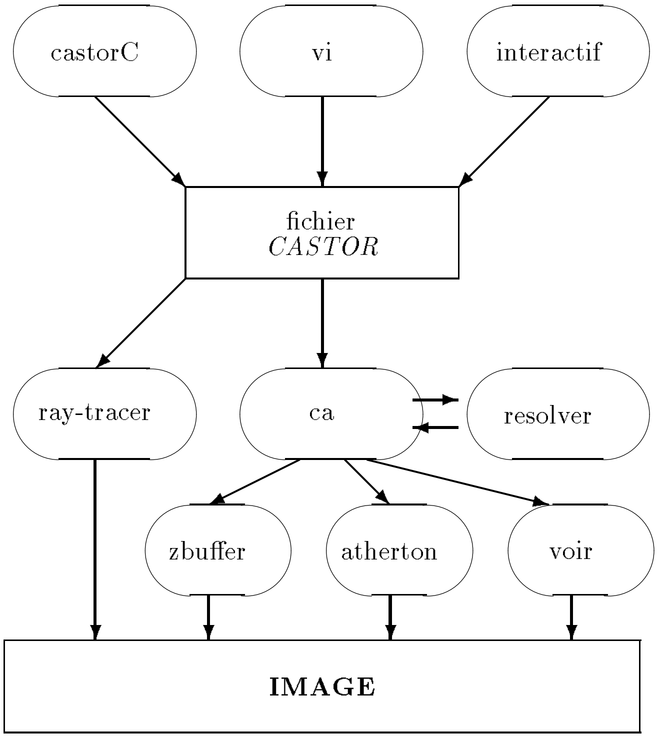


FIG. 2.2 – *Architecture d'Illumines*

Voici un exemple très simple de fichier *CASTOR*.

```
>"(ciel 200 200 200 30 30 100)";

%eye(10,10,10);
%aim(0,0,0);

coeff=2;

un_cube_rouge= !cu(255,0,0,0);
un_cone_vert= !co(0,255,0,0) :(transp 50);
la_scene= $U( @t(2,0,2) un_cube_rouge, @h(coeff) un_cone_vert);

>la_scene;
```

Dans cet exemple,

– la première commande donne un paramètre de rendu concernant le fond, qui est de type ciel : on précise la couleur à l'horizon et la couleur au zénith (en système RVB). Notons que, au sens *CASTOR*, cette commande est un commentaire. Le langage ne précise pas ce qui doit se trouver entre les deux caractères ". En général, on y trouve des données utilisées par l'algorithme de rendu (les sources de lumière par exemple) ou pour la fabrication de l'image finale (le nom, la taille de l'image par exemple).

- les deux commandes suivantes positionnent le point de vue et le point visé.
- la commande suivante définit un paramètre. Un paramètre diffère de la notion classique de variable en ce sens qu'il ne peut changer de valeur une fois défini. Un paramètre joue plutôt un rôle de symbole de préprocesseur.
- la commande suivante définit un objet de type primitive qui est un cube (abréviation `cu`) et dont la couleur est précisée en système RVB. Cet objet est nommé par le nom `un_cube_rouge`.
- la commande suivante définit une primitive (ici un cône). Ce cône possède un attribut, qui est une définition de transparence (ici 50 %).
- la commande suivante définit une opération booléenne (ici une union) entre deux objets anonymes eux-mêmes construits à partir d'objets précédemment définis en leur appliquant des transformations affines (une translation pour le premier objet et une homothétie pour le second). On peut noter que l'homothétie utilise le paramètre précédemment défini.
- enfin, la dernière commande précise quel est l'objet à visualiser.

Comme on peut le voir, la syntaxe d'un fichier *CASTOR* est très simple. Les possibilités du langage sont elles-mêmes restreintes : on n'a pas d'expressions syntaxiques complexes. Ceci est cependant très utile pour deux raisons :

- il est très facile d'écrire un interpréteur de langage *CASTOR*.
- il est très facile pour un programme écrit dans un langage de haut niveau (C, Lisp, C++) de générer des fichiers au format *CASTOR*.

Il faut donc plutôt voir ce langage comme un langage intermédiaire que l'utilisateur n'a pas nécessairement à connaître. Il correspond plutôt à une représentation interne. Le fait d'avoir un langage interprété facilite également la mise au point et une modification facile des scènes modélisées.

Notons également que le langage définit la syntaxe externe des attributs mais pas la syntaxe interne. Ainsi, pour les attributs globaux (exemple de la première commande), le langage précise qu'ils commencent par le couple `>"` et se terminent par `"`, et tout ce qui se trouve entre ces deux marques n'est pas du ressort du langage, mais des algorithmes de visualisation qui utiliseront ces descriptions.

La même remarque est valable pour les attributs d'objets, qui sont introduits par `:(` et se terminent par `)`. Le langage demande simplement qu'il y ait autant de parenthèses ouvrantes que fermantes.

Enfin, notons que la couleur est une entité reconnue par le langage alors qu'elle est en fait un attribut : ceci est une survivance de versions précédentes du langage où les attributs n'existaient pas. Nous l'avons donc conservé dans un souci de compatibilité, même si la définition de couleur est aussi possible par la déclaration d'un attribut.

2.6.2 Quelques améliorations du langage *CASTOR*

Nous avons apporté au langage *CASTOR* des améliorations qui le transforment en un langage un peu plus puissant et surtout qui permet une réutilisation plus facile de scènes déjà écrites. En effet, jusqu'à présent, une base *CASTOR* était une entité unique (un seul fichier), ce qui ne facilite pas le développement entre plusieurs personnes. On pouvait remédier à certains problèmes en utilisant un préprocesseur (le préprocesseur standard du C, par exemple) mais nous avons préféré introduire des notions complémentaires dans le langage lui-même.

Nous avons ainsi introduit la notion de *module*. Un module *CASTOR* est une entité autonome, qui contient une suite d'instructions *CASTOR*. Un module peut requérir l'inclusion d'autres modules, ce qui se représente par une macro-instruction *CASTOR*. Cette décomposition d'une scène en modules apporte une possibilité supplémentaire qui est la gestion locale ou globale des noms d'objets ou de paramètres.

De façon plus précise, à chaque module est associée une table d'identificateurs (noms d'objets ou de paramètres). Cette table est aussi appelée table locale des identificateurs. Il existe également une table globale (unique), qui permet d'éviter la redéfinition dans chaque module d'objets ou de paramètres "universels" (on peut penser à des constantes comme π). Alors, chaque module ne "connaît" que les identificateurs se trouvant dans sa table locale ou dans la table globale.

Afin de permettre la définition des deux types de symboles (locaux et globaux), nous utilisons deux macro-instructions complémentaires, l'une permettant de définir tous les symboles qui suivent cette macro-instruction dans la table locale, et l'autre qui définit les symboles dans la table globale. Par défaut, c'est la table locale qui est utilisée. Notons également que ces macro-instructions ne sont valables que dans le module où elles ont été écrites.

Si l'on respecte la règle énoncée ci-dessus, il est impossible à un module qui inclut un autre module de connaître les objets qui y sont définis : nous avons donc introduit un mécanisme appelé exportation qui résoud ce problème. L'exportation d'identificateurs consiste simplement à rendre accessible au module qui a demandé une inclusion certains identificateurs du module inclus. Si une instruction d'exportation est déclarée dans le module principal, cette exportation se fait alors dans la table globale.

Enfin, en s'inspirant du langage C où l'inclusion de fichiers en-tête utilise un mécanisme de recherche dans des répertoires par défaut (modifiables par l'utilisateur), nous avons ajouté une macro-instruction qui permet de préciser les répertoires où les recherches de modules peuvent être effectuées. Lorsque l'on demande l'inclusion d'un module, on cherche successivement dans les répertoires ainsi indiqués le module en question et on charge le premier module trouvé. Encore une fois, il convient de préciser que cet ensemble de répertoire n'est valable que pour le module dans lequel il est défini : on évite ainsi les effets de bord que peuvent générer de tels mécanismes (inclusions de fichiers non voulus lorsque l'ordre des répertoires de recherche a été changé).

On peut maintenant dresser une liste des macro-instructions que nous avons ajoutées. Toujours en s'inspirant du langage C, ces macro-instructions sont introduites par le caractère `#`. Il en existe donc cinq :

- l'inclusion, qui s'écrit `#include("base.cst","objet.cst");`. On peut noter que l'on peut inclure plusieurs modules avec une seule instruction d'inclusion. Notons également la syntaxe quelque peu différente du préprocesseur C (utilisation des parenthèses).

- la déclaration des chemins de recherche, qui s'écrit `#path(".","/usr/local/cst");`. Encore une fois, notons que l'on peut préciser plusieurs répertoires par une seule instruction. Si l'on veut ajouter des répertoires à ceux déja existants, on utilise une macro-instruction `#patha("./cst")`.

- l'exportation, qui s'écrit `#export(objet_1,objet_2);`. Les arguments de la macro-instruction sont des noms d'objets, qui doivent bien sûr exister dans le module courant (et être définis avant l'exportation) et ne pas exister dans le module qui a demandé l'inclusion.

- le passage en table locale, qui s'écrit `#local;`. On peut noter qu'il existe une version de cette macro-instruction avec argument, qui s'écrit `#local(25);`, l'argument étant le nombre maximal de symboles que doit contenir la table locale. Ceci peut être utile pour affiner les tailles des tables d'identificateurs.

- le passage en table globale, qui s'écrit `#global;`. De même que pour la macro-instruction précédente, celle-ci admet un paramètre optionnel qui est le nombre maximal d'identificateurs que contiendra cette table globale.

2.6.3 La bibliothèque *castorC*

Comme il a été précisé dans la section précédente, il serait fastidieux pour un utilisateur de devoir écrire toutes ses scènes directement en *CASTOR*. C'est pour cette raison que l'environnement *Illumines* contient une bibliothèque de fonctions écrites en C qui permet de générer du *CASTOR*, connue sous le nom de bibliothèque *castorC*. Nous avons alors à notre disposition toute la puissance d'un *vrai* langage de programmation. Cette bibliothèque a été développée dans sa version initiale par Dominique Michelucci [Mic87].

Afin de rendre cette bibliothèque la plus évolutive possible, il a été choisi d'implanter un noyau *à-la-LISP* comme noyau de *castorC*. Ainsi les types de base de *castorC* sont l'entier, le nombre flottant, la paire pointée, le pointeur (en fait un ersatz d'entier), et la chaîne de caractères. Tous les autres types sont construits à partir de ces objets de base.

Toujours en s'inspirant du langage LISP, toutes les données sont gérées dans un tableau unique, et chaque donnée est identifiée par son numéro dans ce tableau. De même, on peut construire très facilement à partir de ce noyau les notions classiques de liste, de car, de cdr,... que l'on trouve dans tout système LISP.

Les extensions qui ont été apportées à cette bibliothèque sont essentiellement les suivantes :

- toutes les fonctionnalités de nommage. En effet, dans les versions précédentes, les noms des objets étaient générés de façon automatique, ce qui rendait quasiment impossible la réutilisation des objets. Pour éviter toutefois d'avoir à nommer tous les objets, y compris les objets intermédiaires dont les noms nous importent peu, nous avons également ajouté la possibilité d'une génération automatique des noms sous la forme `préfixe+numéro`.

- la possibilité de générer des primitives purement géométriques. Initialement, toute primitive devait obligatoirement contenir une “couleur”, c'est-à-dire trois valeurs en RVB.

- la possibilité d'ajouter des attributs à n'importe quel objet (ceux-ci étaient réservés aux seules primitives).

- la prise en compte d'attributs particuliers qui permettent de modifier la construction des opérations booléennes. Ceci nous a permis d'intégrer les notions de priorité horizontale et la notion de fusion.

- toutes les fonctions nécessaires pour générer des macro-instructions.

D'autres modifications d'implantation ont également été apportées pour améliorer l'efficacité de *castorC*. On peut citer en particulier la gestion d'un tableau de tableau de données au lieu d'un tableau unique, ce qui permet d'avoir des tailles de données raisonnables pour les petites scènes, mais autorise cependant les scènes importantes : il devient alors inutile de devoir recompiler la bibliothèque pour créer des scènes plus importantes.

Une meilleure modularité a également été apportée, ce qui réduit encore la taille des codes puisque les parties de code inutiles ne sont pas chargées avec l'exécutable.

Enfin, nous avons intégré la possibilité de définir des déformations libres (considérées comme des tranformations). Ce travail a été réalisé par Zhigang Nie [Nie91], et nous l'avons intégré dans la bibliothèque. Notons qu'il est alors possible de définir des déformations de bas niveau (en donnant un maillage avant déformation puis le maillage après déformation), ou en utilisant des déformations de plus haut niveau, comme le pliage, la torsion, l'amincissement. Les maillages utilisés peuvent être adaptés aux besoins, et utiliser à volonté des coordonnées cartésiennes, cylindriques ou sphériques. Il convient de noter que notre version actuelle de tracé de rayons ne permet pas l'utilisation de ces déformations.

2.7 Extensions du modèle pour l'animation

Il existe un paramètre dont nous n'avons jusquà présent jamais tenu compte : le temps. Il est en fait très simple d'introduire une notion temporelle dans le modèle que nous avons défini. Ceci peut avoir les intérêts suivants :

- possibilités de modéliser des animations, c'est-à-dire des séquences d'images où certains paramètres peuvent varier (position de certains objets, position de l'oeil,...).

- possibilité d'introduire des effets de flou. Ceci permet de rendre les images extrêment réalistes en simulant ce qui se passe avec une caméra ou un appareil photo : l'image n'est pas prise de façon instantanée mais pendant un intervalle de temps. Si des objets sont en mouvement dans la scène (ou si le dispositif de prise de vues est lui-même en mouvement), des flous sont générés. Cette technique a été utilisée par Robert Cook [CPC84], et permet de générer des images d'un réalisme impressionnant.

Nous nous sommes toutefois limités dans ces extensions à la prise en compte du temps de façon explicite : nous n'avons pas traité les problèmes d'objets en mouvement les uns par rapport aux autres avec des lois de mouvement dont on ne connaît que des équations.

2.7.1 Mouvements affines

Par définition, un mouvement affine est une transformation affine dont les paramètres peuvent varier en fonction du temps. Par exemple, le mouvement d'un solide indéformable peut s'exprimer comme la combinaison de deux mouvements affines :

- une rotation d'angle θ autour de l'axe instantané de rotation Δ.

- une translation de vecteur $\vec{T}$.

Les valeurs de θ, Δ et $\vec{T}$ sont bien sûr des fonctions du temps.

2.7.2 Opérateur de présence/absence

Cet opérateur est défini par un objet et deux temps limites. Entre les deux limites, l'objet représenté par l'opérateur est l'objet de base, en dehors de ces limites, c'est un objet nul.

L'opérateur de présence/absence permet de représenter les objets qui apparaissent ou disparaissent au cours de l'animation.

2.7.3 Variantes d'objets

L'opérateur de variantes est défini par n instants $(t_1, ..., t_n)$ et par $n+1$ objets $(O_0, O_1, ..., O_n)$. L'objet représenté par l'opérateur est alors

- l'objet O_0 si le temps est inférieur à t_1.

- l'objet O_n si le temps est supérieur ou égal à t_n.

- l'objet O_i si le temps est supérieur ou égal à t_i et inférieur à t_{i+1}.

Cet opérateur de variantes permet d'éviter une combinaison d'unions et d'opérateurs de présence/absence pour représenter des objets polymorphes, c'est-à-dire des objets dont la définition change avec le temps.

2.7.4 Changement des paramètres de vue

Afin de permettre la modélisation des déplacements du point de vue ou du point visé, ou encore de n'importe quel autre paramètre de vue (angles d'ouverture par exemple), il suffit de remplacer chacun de ces paramètres par des expressions symboliques, avec la restriction que la seule “variable” pouvant intervenir dans ces expressions est le temps.

2.7.5 Expressions symboliques

Aussi bien pour les mouvements affines que pour les changements des paramètres de vue, il est nécessaire d'introduire dans notre modèle des expressions symboliques. Nous avons imposé les restrictions suivantes :

- la seule variable intervenant dans ces expressions est le temps.
- toutes les expressions sont des expressions en virgule flottante (`double` en C).
- les constructeurs d'expressions sont les suivants : addition binaire, soustraction binaire, multiplication binaire, division binaire, appels de fonctions à un ou deux arguments.

Il faut noter que pour les appels de fonctions, nous nous limitons aux fonctions de la bibliothèque mathématique “standard” du C, qui acceptent un ou deux arguments de type `double` et rendant un résultat de type `double`. On peut donc utiliser les fonctions suivantes : sin, cos, tan, $asin$, $acos$, $atan$, $atan2$, exp, log, $log10$, pow, $sqrt$, $j0$, $j1$, $y0$, $y1$, $cosh$, $sinh$, $tanh$, $acosh$, $asinh$, $atanh$, $ceil$, $fabs$, $floor$, $fmod$, erf, $erfc$, $gamma$ et $cbrt$ si cette fonction existe.

2.7.6 Réévaluation partielle du graphe CSG

Nous avons choisi d'intégrer ces extensions dans notre modèle en respectant la structure déjà existante. En effet, si l'on examine ces extensions, on constate que celles-ci ne modifient pas la structure de notre modèle. Il suffit de l'adapter légérement.

Par exemple, un mouvement affine ressemble (ce qui semble naturel) à une transformation affine. La seule différence est que l'on stocke en plus la description symbolique du mouvement. Ainsi, un nœud de type mouvement affine contient une matrice de transformation qui est la représentation du mouvement à un instant donné. En utilisant le langage de l'approche objet, on pourrait dire que le mouvement représente une classe de transformations, dans la valeur à un instant donné est une instance.

De la même façon, les nœuds de type présence/absence ou variantes sont considérées comme un nouveau type d'opération booléenne, et on y stocke leurs données de définition (c'est-à-dire les instants-clés et les objets correspondants) ainsi qu'un opérande qui n'est donc valable qu'à un instant donné.

Toutes les remarques précédentes impliquent que le graphe CSG doit être modifié (ou réévalué) chaque fois que l'on modifie l'instant de modélisation.

Dans ce but, on stocke sur chaque nœud du graphe un drapeau qui indique si le sous-graphe issu du nœud en question contient des opérateurs “temporels”. Chaque fois que l'on modifiera l'instant de rendu d'une scène, on parcourera préalablement le graphe pour effectuer les mises à jour.

2.7.7 Extensions apportées à *CASTOR*

Afin de réaliser dans le langage *CASTOR* les opérateurs précédemment définis, nous avons introduit les extensions suivantes :

- nous avons introduit la nouvelle notion de *variable*, qui diffère d'un paramètre par la possibilité de changer de valeur. Ceci est réalisé par la définition d'une nouvelle macro-instruction, notée

`#defvar(_time)`. `_time` est le nom de variable que nous avons retenu pour représenter le temps, l'unité de cette variable étant laissée libre à l'utilisateur (selon les cas, on peut prendre la micro-seconde ou la centaine de siècles). Le temps est actuellement la seule variable que nous utilisons mais on pourrait en trouver beaucoup d'autres applications. Notons également qu'une variable est toujours globale, et est donc connue dans tous les modules.

- pour les mouvements affines, on garde la syntaxe des transformations affines, la seule différence étant qu'un mouvement affine se repère par le fait que un ou plusieurs de ses paramètres font intervenir la variable *CASTOR* `_time`.
- pour les opérateurs de présence/absence et de variantes, on crée de nouvelles opérations booléennes.

Voici un exemple de fichier *CASTOR* utilisant ces nouvelles fonctionnalités.

```
#defvar(_time);

%eye(4+2*_time,5,0.4*cos(6*pi*_time));
%aim(0,0,0);

cube_qui_bouge = @t(0,5*_time,0) !cu(255,0,0,0);

objet_furtif = @p(0,1) !cy(0,0,255,0);

objet_variant = $V(0,1)(!co(255,0,0,0),!cy(255,0,0,0),!to(0.5)(255,0,0,0);

la_scene = $U(cube_qui_bouge,objet_furtif,objet_variant);

_time=-0.5;
> la_scene;

_time=0;
> la_scene;

_time=0.5;
> la_scene;
```

Dans cet exemple,

- l'oeil est en mouvement sur un arc de sinusoïde.
- `cube_qui_bouge` est un cube soumis à un mouvement affine (ici une translation rectiligne uniforme).
- `objet_furtif` est un objet n'existant que pour un instant compris entre 0 et 1, représentant alors un cylindre.
- `objet_variant` représente un cône pour $t < 0$, un cylindre pour $0 \leq t < 1$ et un tore pour $t \geq 1$.
- on demande trois visualisations aux instants -0.5, 0 et 0.5.

2.8 L'interprète *readCASTOR*

Afin de faciliter l'inclusion dans un algorithme de rendu d'un module de lecture de fichiers au format *CASTOR*, nous avons réalisé un interprète programmable connu sous le nom de **readCASTOR**. Son rôle est de réaliser l'interface entre un fichier *CASTOR* et des fonctions écrites dans un langage de haut niveau qui construisent les objets, leur affectent des attributs, positionnent les paramètres de vue. C'est cet interprète qui reconnaît les entités lexicales et syntaxiques du langage.

2.8.1 Fonctionnement de l'interprète

L'interprète lit le fichier d'entrée et reconnaît des instructions *CASTOR*. Une instruction est une suite de caractères allant jusqu'au caractère ; suivant.

Certaines de ces instructions ne concernent que le fonctionnement de l'interprète : ce sont les définitions de paramètres et les macro-instructions. Ainsi, l'utilisateur n'a pas à gérer ces instructions, ce qui est particulièrement intéressant pour les macro-instructions d'inclusion. Notons d'ailleurs que **readCASTOR** introduit une limitation sur les modules inclus : la profondeur de l'arbre d'inclusions est limitée à 8.

D'autres instructions déclenchent des actions, ces actions possédant des valeurs par défaut et ayant la possibilité d'être redéfinies par l'utilisateur. Une action est en fait représentée par un pointeur sur fonction, que l'utilisateur peut redéfinir à sa guise, en respectant bien sûr la syntaxe d'appel (nombre et types des arguments de la fonction et type du résultat) voulue par l'interprète.

L'interprète gère également un grand nombre d'erreurs qui peuvent être rencontrées lors de la lecture d'un fichier *CASTOR* : en particulier, toutes les erreurs lexicales et syntaxiques sont reconnues, de même que les erreurs sur les macro-instructions. Les erreurs sur le nombre d'arguments pour les primitives, les transformations affines ou les opérations booléennes sont également détectées.

Dès qu'une erreur est rencontrée, l'interprète continue à analyser le fichier mais plus aucune action n'est effectuée. Il peut alors continuer à détecter des erreurs lexicales ou syntaxiques. Notons que pour simplifier la mise au point, l'interprète signale toujours les erreurs en indiquant le fichier mis en cause ainsi que le numéro de la ligne où l'erreur a été détectée. Certaines de ces erreurs sont toutefois difficiles à détecter, comme par exemple un caractère `"` manquant en fin de commentaire : l'interprète essaie alors de considérer toute la suite du fichier (jusqu'au caractère `"` suivant) comme un commentaire, ce qui peut générer des erreurs qui ne sont détectées que bien plus loin.

2.8.2 Implantation

Cet interprète est réalisé sous forme d'une bibliothèque exécutable contenant le code de l'interprète et d'un fichier entête contenant la définition de quelques constantes et d'un type structuré utilisé pour retourner de l'information sur le fichier lu.

Tout le code est réalisé en langage C, et utilise les outils `UNIX` standard que sont `lex` et `yacc` pour la partie analyse lexicale et syntaxique du fichier *CASTOR* (on peut trouver dans l'annexe A le source `yacc` correspondant à la grammaire de **readCASTOR**). L'interprète peut donc être porté sur toute machine `UNIX`. On peut d'ailleurs noter dès à présent que ceci nous a tout de même posé un problème lors de la réalisation d'une version parallèle de notre algorithme, car la machine parallèle utilisée n'est pas une machine `UNIX` (voir le chapitre 6). Nous avons contourné ce problème en ne réécrivant que l'analyseur lexical pour la version parallèle : en effet, `lex` utilise une bibliothèque livrée avec le système d'exploitation de nos machines et il est donc impossible de l'utiliser. `yacc` en revanche génére un fichier source en C qu'il suffit de compiler pour la machine parallèle, la conversion source `yacc` vers source C étant réalisée sur la station `UNIX` hôte de notre machine parallèle.

2.8.3 Utilisation

Utiliser cet interprète programmable est en fait très simple. Une fois un algorithme écrit (dans notre cas un tracé de rayons), il suffit d'écrire des fonctions réalisant l'interface entre les actions de construction de l'algorithme (construction des structures de données par exemple) et les actions au sens **readCASTOR**.

Une fois ces fonctions-interfaces écrites, la lecture se fait en affectant tout d'abord ces fonctions comme actions à effectuer puis en appelant la fonction de lecture d'un fichier : ce fichier est alors ouvert puis analysé, et toutes les actions sont exécutées au fur et à mesure. Une erreur dans le fichier provoque l'arrêt de ces exécutions sans interrompre la lecture.

Comme résultat de la fonction de lecture, un pointeur sur une structure de données contenant le nom du fichier lu, le nombre de lignes, le nombre d'instructions, le nombre d'erreurs (syntaxiques ou sémantiques) est retourné et l'utilisateur peut donc consulter cette structure.

Notons également que l'utilisateur peut préciser au moment de l'appel à la fonction de lecture s'il désire utiliser les extensions pour l'animation décrites précédemment. Si tel n'est pas le cas, l'interprète se charge d'évaluer toutes les expressions symboliques rencontrées, les variables non affectées prenant alors la valeur 0.

Chapitre 3

L'intersecteur

3.1 Définitions

3.1.1 Rayon

Il convient de définir tout d'abord ce qu'est pour nous un "rayon", c'est-à-dire l'entité que nous allons intersecter avec un objet représenté à l'aide du modèle défini au chapitre précédent.
Un rayon est défini par :

- son origine O qui est un point de l'espace
- sa direction $\vec{V}$ qui est un vecteur

Tout point P du rayon peut être identifié de façon unique par son abscisse λ sur ce rayon définie par $P = O + \lambda\vec{V}$.
On peut remarquer que rien n'oblige la direction du rayon à être un vecteur normé.

3.1.2 Intervalle d'intersection

Par définition, un *intervalle d'intersection* est une partie convexe d'un rayon, maximale au sens suivant : tous les points de l'intérieur de l'intervalle appartiennent au même objet, les points extrémaux sont sur la surface de l'objet.
Un intervalle d'intersection contient quatre informations essentielles :

- une abscisse d'entrée dans l'objet et une abscisse de sortie de l'objet
- la description de l'intérieur de l'objet
- la description du point d'entrée
- la description du point de sortie

La description de l'intérieur contient les attributs volumiques de l'objet intersecté. Il faut noter que dans le cas d'un intervalle d'intersection avec un ou plusieurs objets neutres, on trouve dans cette description les attributs de tous les objets intersectés. C'est le processus de rendu qui détermine comment combiner ces informations pour effectuer un rendu correct.

Nous appelons une telle description une *cellule-intervalle* ou bien un *contenu*.

La description des points d'entrée et de sortie contient les informations suivantes : normale à la surface de l'objet [1], attributs surfaciques, ainsi que des informations utiles telles que le numéro de l'objet intersecté, le numéro de la face de l'objet intersecté. Ces informations peuvent être utiles pour effectuer certaines texturations ou bien pour effectuer un antialiassage.

Nous appelons ces descriptions de points d'entrée et de sortie des *cellules-point*. Il y a donc une cellule-point pour le point d'entrée et une autre pour le point de sortie.

On peut également trouver dans l'intervalle d'intersection une information complémentaire sur les attributs volumiques d'un éventuel objet accolé contre l'objet courant. Ceci est en effet très utile dans le cas des rayons réfractés : il faut non seulement connaître le point de sortie de l'objet réfractant mais également les attributs volumiques de l'objet que l'on rencontre juste à la sortie. Ceci est absolument indispensable pour effectuer correctement les calculs de réfraction.

Cette information ne se trouve évidemment que dans les intervalles décrivant des objets actifs (ce sont les seuls éventuellement concernés par la réfraction). Deux cas peuvent alors se présenter : soit il y a un objet actif juste après la sortie, auquel cas on trouve les informations voulues dans l'intervalle d'intersection, soit il y a un ou plusieurs objets neutres ou encore pas d'objet du tout (on se trouve alors dans le milieu ambiant) auquel cas ce sont les propriétés du milieu ambiant que l'on utilise pour appliquer les lois de Fresnel.

Enfin, on stocke sur chaque intervalle deux informations qui sont le type d'objet intersecté (neutre ou actif) et le type d'intervalle. Le type d'intervalle décrit si l'objet intersecté est d'intérieur vide (au sens topologique) ou non. Cette information est utile pour que l'intersecteur respecte la règle selon laquelle un objet surfacique ne peut trouer un objet volumique.

Hormis les types d'intervalle et d'objet intersecté, la présence de ces informations n'est pas obligatoire. Par exemple, dans le cas d'une différence, on ne génére pas les contenus des intervalles car ils n'influent en rien l'objet résultat d'après les règles de priorité. Lors de l'appel à l'intersecteur, on décrit par un ensemble de drapeaux les informations que l'on désire. Ces drapeaux sont eux-mêmes modifiés par l'intersecteur.

De même, ces informations ne sont générées que si elles ont un sens. Par exemple, s'il se trouve que l'origine du rayon est à l'intérieur d'un objet, il n'y a pas lieu de décrire un point d'entrée qui n'existe pas : on ne saurait ni quelle normale ni quels attributs surfaciques lui affecter.

3.1.3 Intersection entre un rayon et un objet

Une fois définie la notion d'intervalle d'intersection, on peut préciser ce que l'on considère comme le résultat d'une intersection entre un rayon et un objet représenté par un graphe CSG : c'est tout simplement une liste ordonnée d'intervalles d'intersection deux-à-deux disjoints.

Il faut noter que cette liste peut contenir zéro, un ou plusieurs intervalles. En effet, pour effectuer correctement les opérations booléennes, il est nécessaire de générer tous les intervalles. De même, pour effectuer certains rendus (calculs de masse ou de moments d'inertie), on doit connaître la totalité des intervalles intersectés.

3.1.4 Environnement

Nous avons insisté dans le chapitre 2 sur le fait que la réutilisabilité des objets entraine qu'un objet du graphe CSG ne représente pas nécessairement un objet unique de la scène. Il y a donc tout un ensemble d'informations que l'on doit connaître pour effectuer correctement les intersections et qui dépendent du chemin d'accès à l'objet dans le graphe. Nous avons regroupé toutes ces informations sous le nom d'*environnement*.

Détaillons maintenant ce que contient cet environnement et comment il est mis à jour au cours du parcours du graphe.

1. Par convention, une normale est toujours orientée vers l'extérieur de l'objet

3.1.4.1 Segment d'étude

Une particularité de notre algorithme est le fait que les intersections entre un rayon est un objet ne sont recherchées que sur un segment du rayon est non pas sur le rayon entier (ou même sur un demi-rayon). On précise alors le segment d'études en donnant à l'intersecteur une abscisse minimale et une abscisse maximale sur le rayon : tous les intervalles d'intersection générés ont des abscisses d'entrée et de sortie comprises entre l'abscisse minimale et l'abscisse maximale. Ceci explique pourquoi dans certains cas les points d'entrée et de sortie ne sont pas générés : si un intervalle a été tronqué par le segment d'étude, le point d'entrée (ou le point de sortie) n'a aucun sens.

De plus, on utilise pour chaque type d'objet une donnée supplémentaire qui est l'abscisse maximale du point d'entrée. Comme son nom l'indique, un intervalle d'intersection n'est généré que si son point d'entrée a une abscisse inférieure à cette abscisse maximale de point d'entrée. Il faut noter que cette valeur est toujours inférieure à l'abscisse maximale mais non forcément égale. Dans le cas de rayons réfractés par exemple, ceci évite de calculer des intersections avec des objets qui sont en dehors de l'objet réfracté..

Ce segment d'étude peut être modifié en cours d'algorithme par les règles suivantes :

- sur un nœud de type intersection ou différence, si l'on a trouvé des intervalles d'intersection avec l'opérande gauche, le segment d'étude pour l'opérande droit est modifié comme suit : son abscisse minimale est l'abscisse d'entrée du premier intervalle d'intersection avec l'opérande gauche, son abscisse maximale est l'abscisse de sortie du dernier intervalle d'intersection. Il est en effet inutile de rechercher des intersections en dehors de ce segment : elles seraient éliminées lors de la combinaison des intervalles par l'opération booléenne.

- dans le cas où on ne demande que la première intersection avec un objet actif (cas des rayons primaires,réfléchis ou réfractés dans un algorithme classique de rendu), sur un nœud de type union intervenant dans un sous-graphe ne contenant pas d'intersection ou de différence, si l'on a trouvé des intervalles d'intersection avec des objets actifs de l'opérande gauche, on modifie les abscisses maximales des points d'entrée comme suit : si l'objet gauche est prioritaire, les abscisses maximales de points d'entrée prennent la valeur de l'abscisse d'entrée de l'intervalle d'intersection de type actif avec l'objet gauche, si l'objet droit est prioritaire, l'abscisse maximale d'entrée pour les objets actifs prend la valeur de l'abscisse de sortie de l'intervalle d'intersection de type actif avec l'objet gauche tandis que l'abscisse maximale d'entrée pour les objets neutres prend la valeur de l'abscisse d'entrée dans ce même intervalle.

La figure 3.1 résume l'influence du segment d'étude sur la génération des intervalles d'intersection. Il convient de noter que les abscisses maximales d'entrée n'influent pas sur la troncature des intervalles : elles ne jouent que sur l'utilité ou la non-utilité des intervalles.

3.1.4.2 Intervalles demandés

Cette indication informe l'algorithme d'intersection sur le type et le nombre d'intervalles qu'il doit générer. Nous appelons cette donnée le *niveau d'intersection*. Selon le type de rendu et le type du rayon généré, ce niveau peut prendre des valeurs très différentes.

Prenons le cas où le rendu est un calcul classique d'image avec ombres, réflexion et réfraction, et détaillons pour chaque type de rayon les informations nécessaires.

- pour les rayons primaires et les rayon réfléchis, il est nécessaire de connaître l'intervalle d'intersection avec le premier objet actif rencontré, ainsi que les intervalles d'intersection avec les objets neutres se trouvant devant ce premier actif.

- pour les rayons d'ombre, on a besoin de connaître les objets neutres se trouvant entre le point considéré et la source lumineuse. Mais s'il l'on trouve une intersection avec un objet actif, on peut

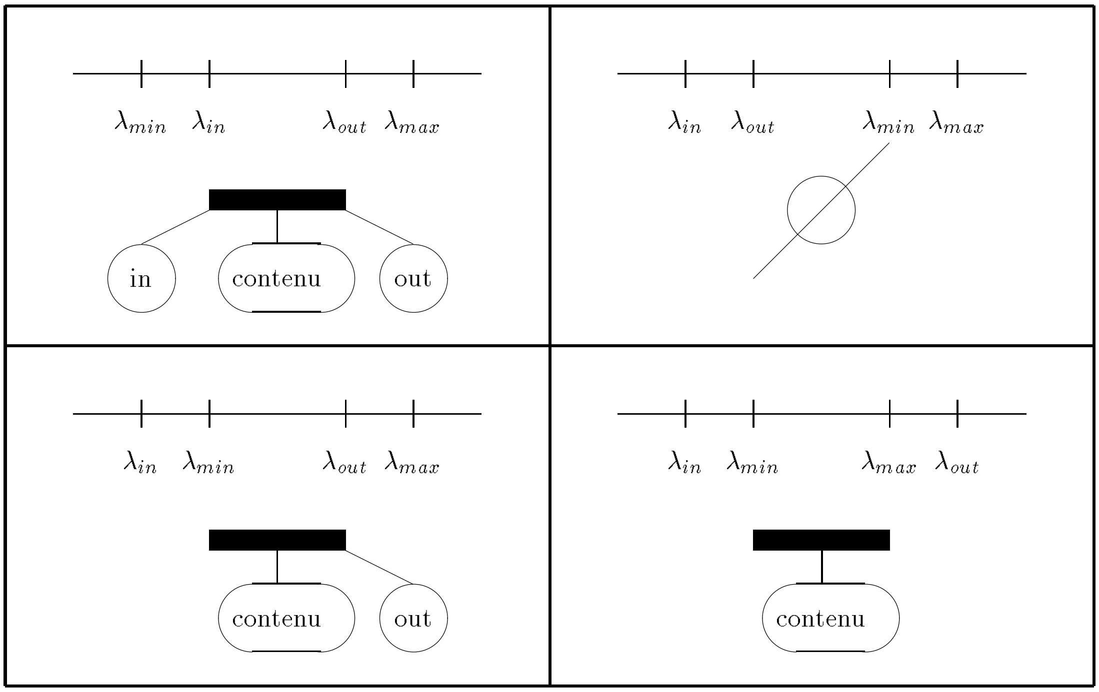


FIG. 3.1 – *Influence du segment d'étude sur les intervalles d'intersection*

immédiatement s'arrêter : on est sûr que l'objet est à l'ombre. On peut d'ailleurs remarquer que dans ce cas, peu importe le fait que l'objet actif ainsi trouvé soit le premier sur le rayon : il suffit d'en trouver un.

– en ce qui concerne les rayons réfractés, notons tout d'abord que de tels rayons ne peuvent intervenir à l'intérieur d'objets neutres (en effet, ceux-ci ne dévient pas la lumi'ere par définition). On a alors besoin de connaître le point de sortie de l'objet réfractant, ainsi que la nature de l'objet se trouvant derrière cette sortie. Il est donc nécessaire de calculer l'intervalle d'intersection avec le premier actif, et on peut se passer dans ce cas des objets neutres.

Dans le cas où le rendu est un calcul de masse ou de moments d'inertie, on doit alors générer tous les intervalles d'intersections avec les objets actifs (les neutres n'ont pas vraiment de sens pour un tel calcul).

Il est aussi possible par exemple, de ne demander aucune intersection avec les objets neutres (ou avec les objets actifs) afin de simplifier dans un premier temps les calculs d'intersection et de rendu. Ceci est particulièrement intéressant pendant les phases de mise au point des scènes. Il faut noter que ce cas est différent du cas des rayons d'ombres puisqu'ici, on ne tient absolument pas compte des intersections alors que pour les rayons d'ombres, on ne tient pas compte de la position des intersections mais on a néanmoins besoin de connaître leur présence ou leur absence.

L'intersecteur utilise donc pour chaque type d'objets une valeur de niveau parmi les quatre valeurs suivantes :

– `IGNORE` : on ignore simplement le type correspondant.

– `ZERO` : l'intersecteur stoppe dès qu'il trouve une intersection avec un objet du type correspondant mais ne génère pas d'intervalle d'intersection.

– `UN` : l'intersecteur génère le premier intervalle d'intersection avec l'objet de type correspondant.

– `TOUS` : l'intersecteur génère tous les intervalles.

Ces niveaux sont passés à l'appel initial à l'intersecteur et peuvent être modifiés en cours d'algorithme par les opérations booléennes.

On peut d'ailleurs noter que, même si on ne demande la génération d'aucun intervalle, on peut utiliser le segment d'étude pour rendre un résultat partiel : si on trouve qu'il y a au moins une intersection, on peut tronquer le segment d'étude à un segment contenant ladite intersection. Ceci est utilisé pour les intersections avec le cube, le cône, le cylindre (voir plus loin) et également pour les intersections avec les boîtes englobantes (voir chapitre 4).

3.1.4.3 Couleur en cours

Rappelons ici que la couleur ne peut prendre qu'une valeur par classe pour un objet. En revanche, on doit gérer les deux classes de couleur (surfacique et volumique). L'environnement contient donc en fait deux *couleurs courantes*.

L'environnement doit être modifié en ce qui concerne les couleurs par les deux facteurs qui influent sur la valeur de la couleur : la règle de priorité verticale d'une part et la notion de non-colorabilité d'autre part.

Les modifications à apporter à l'environnement sont alors, pour chaque classe, les suivantes :

– lorsque l'on trouve un nœud non colorable, retirer de l'environnement la couleur en cours.

– lorsque l'on trouve une couleur sur un nœud, deux cas sont possibles :

  – soit une couleur est déjà présente dans l'environnement, auquel cas on ne modifie rien (priorité verticale).

  – soit aucune couleur n'est présente, auquel cas on affecte cette couleur à la couleur courante de l'environnement.

Il faut noter que la règle de priorité horizontale n'intervient pas dans la gestion de l'environnement : elle intervient lors de la combinaison des intervalles d'intersection pour les opérations booléennes.

Une autre particularité est le fait que l'intersecteur, lorsqu'il effectue ces manipulations, n'a aucun besoin de connaître le type des couleurs qu'il manipule : ce peut être un pointeur sur un ensemble de données dont il ignore la structure, un numéro dans une table de couleurs, un pointeur sur un nom de couleur. Ce n'est que le processus de rendu qui interprétera cette couleur.

3.1.4.4 Matrice de passage monde-local

Il est nécessaire de connaître cette matrice pour effectuer les texturations dans les bons repères. En effet, lors du rendu, on connaît les données (position des points, normales) dans le repère du monde. Or, nous avons expliqué au chapitre 2 que les textures sont toujours définies dans un repère local.

Cette matrice est nécessaire aussi bien pour les textures de couleur (pour lequelles il faut calculer la position du point dans le repère local) que pour les textures de normale (pour lesquelles il faut de plus convertir la normale dans le repère local, effectuer la texturation et ensuite reconvertir la normale dans le repère du monde).

Enfin, cette matrice peut être modifiée aussi bien par la matrice globale de chaque nœud, que par la matrice de transformation d'un nœud de type transformation, ou que par les matrices d'opérandes d'une opération booléenne.

3.1.4.5 Textures

La gestion des textures est très similaire à celle de la couleur. Il y a cependant une particularité des textures dont il faut tenir compte : elles peuvent se superposer. L'environnement doit donc gérer l'ensemble des textures rencontrées au cours du parcours du graphe, en respectant l'ordre de texturation (les textures les plus basses du graphe doivent être appliquées avant les plus hautes).
De plus, une texture prend des valeurs différentes pour un objet en fonction du point où l'on se trouve. Comme les textures sont toujours définies par rapport à un repère local, il est nécessaire de stocker pour chaque texture la matrice de passage du repère du monde au repère local de la texture. En effet, au moment ou le rendu est effectué, les points d'intersection sont connus dans le repère du monde.

C'est pour ces raisons que l'environnement utilise ce que nous appelons un *bloc-texture*. Un bloc-texture contient la matrice de passage dans le repère local, ainsi qu'un pointeur sur la liste des textures à appliquer dans ce repère. Dans le cas de textures multiples appliquées à un nœud, ceci permet d'éviter la duplication de la matrice de passage (qui est relativement encombrante) pour chacune des textures.

Encore une fois, nous insistons sur le fait que l'intersecteur n'a pas à connaître le type des textures ni ce qu'elles contiennent. Chaque nœud du graphe CSG contient deux listes de textures. Chaque liste peut être un pointeur sur une liste chaînée de textures, ou bien un numéro de tables de textures,. . . . L'intersecteur se contente de recopier dans l'environnement cette valeur accompagnée de la matrice de passage.

Afin de respecter l'ordre des textures, l'environnement gère une pile de blocs-texture par classe de textures. En fait, les deux piles de blocs-textures sont uniques : ce ne sont que les pointeurs de tête de pile qui sont gérés par l'algorithme.
L'algorithme de gestion peut alors s'écrire comme suit :

```
mettre à jour la matrice de passage monde-local (matrice globale)
si l'objet est non-colorable
        vider la pile de blocs-texture surfaciques
        vider la pile de blocs-texture volumiques
si l'objet contient des textures surfaciques
        empiler le bloc-texture surfacique correspondant
si l'objet contient des textures volumiques
        empiler le bloc-texture volumique correspondant
calculer les intervalles d'intersection avec l'objet
si l'objet contient des textures surfaciques
        dépiler le bloc-texture surfacique de tête
si l'objet contient des textures volumique
        dépiler le bloc-texture volumique de tête
```

Lors de l'intersection avec des primitives, il suffit de dépiler chacune des deux piles de blocs-texture pour retrouver dans le bon ordre les textures surfaciques et volumiques à appliquer.

3.1.4.6 Numéros d'objets

Comme il a été expliqué au chapitre 2, le principe de numérotation des objets dépend du chemin d'accès dans le graphe. Il est donc naturel que l'environnement gère cette donnée.

Au début de l'algorithme d'intersection, le numéro d'objet est initialisé à zéro dans l'environnement (les numéros d'objet débutent à 1). Ce numéro d'objet courant ne doit être modifié que pour les opérations booléennes.

L'algorithme de gestion de ce numéro est alors le suivant :

intersecter l'opérande gauche
incrémenter le numéro d'objet courant du nombre d'objets
    contenus dans l'opérande gauche
intersecter l'opérande droit

3.1.5 Résultat d'une fonction d'intersection

Par convention, nous avons choisi que toutes les fonctions d'intersection avec des primitives ainsi que les fonctions de combinaison de listes d'intervalles d'intersection par des opérations booléennes rendent une valeur de résultat. Cette valeur est un entier, qui donnent simultanément le nombre d'intersections trouvées avec des objets neutres et le nombre d'intersections trouvées avec des objets actifs [2].

La valeur rendue par la fonction d'intersection n'est pas le seul moyen de récupérer des résultats de la fonction d'intersection : la fonction d'intersection peut modifier certaines parties de l'environnement pour y stocker des résultats non stockables ailleurs. Nous préciserons par la suite les utilisations de cette possibilité.

3.2 Intersection avec les primitives

Nous allons maintenant détailler comment l'intersecteur calcule des intersections avec les primitives. Nous allons simplement détailler comment nous calculons les abscisses d'entrée et de sortie ainsi que les normales. En effet, quelle que soit la primitive, l'affectation des textures et couleurs volumiques et surfaciques est indépendante du type de primitive intersectée.

Par exemple, pour un point d'entrée, il suffit de recopier la valeur de la couleur surfacique dans la cellule de point d'entrée ainsi que de copier la pile de blocs-texture surfaciques : ces deux données se trouvent simplement dans l'environnement.

De même, pour le contenu, il suffit de recopier la valeur de la couleur volumique dans le contenu, ainsi que la pile de blocs-texture volumique. On peut d'ailleurs noter qu'au niveau de la primitive, il n'y a pas de superposition d'objets neutres : celles-ci n'apparaissent que lors des opérations booléennes entre objets neutres.

Dans toute la suite du paragraphe, on note λ_{min} et λ_{out} les bornes du segment d'étude, et λ_{max}^{in} l'abscisse maximale d'entrée pour le type de l'objet en cours d'intersection.

3.2.1 Intersection avec une tranche d'espace

Bien que cet objet ne soit pas une primitive de notre modèle à proprement parler, elle est utilisée en interne pour réaliser une intersection avec un cube, un cylindre ou un cône ainsi qu'avec les boîtes englobantes (voir le chapitre 4).

Une tranche d'espace est la portion d'espace comprise entre deux plans parallèles. Elle peut être définie par la donnée d'un vecteur $\vec{S}$ et de deux constantes α et β. Un point P de l'espace appartient à cette tranche si et seulement si $\beta \leq < \vec{CP}, \vec{S} > +\alpha \leq 0$ (C est l'origine du repère du monde). On remarque que dans cette définition, β doit être strictement négatif pour que la tranche représente bien un volume et non une surface ou un objet vide.
On note $P_1 = P_1(\vec{S}, \alpha, \beta)$ le plan de $\mathbb{R}^3$ défini par $< \vec{CP}, \vec{S} > +\alpha = 0$ et de façon similaire $P_2 = P_2(\vec{S}, \alpha, \beta)$ le plan de $\mathbb{R}^3$ défini par $< \vec{CP}, \vec{S} > +\alpha - \beta = 0$. Ainsi, P_1 et P_2 sont les deux plans qui contiennent la tranche d'espace.

2. Comme nous utilisons des entiers sur 32 bits, ceci limite à 65535 le nombre maximal d'intervalles d'intersection générés pour une classe d'objets. Est-ce vraiment une limitation ?

Pour résoudre l'intersection, on paramétrise le point courant du rayon par $P = O + \lambda\vec{V}$, et on reporte cette expression dans les deux équations de plans. On obtient alors :

$$\lambda < \vec{S}, \vec{V} > \; = \; - < \vec{S}, \vec{CO} > -\alpha \tag{3.1}$$

$$\lambda < \vec{S}, \vec{V} > \; = \; - < \vec{S}, \vec{CO} > -\alpha + \beta \tag{3.2}$$

Plusieurs cas peuvent alors se présenter.

- si $< \vec{S}, \vec{V} >= 0$, le rayon est parallèle à la tranche. Il est donc soit entièrement en dehors de la tranche, soit entièrement en dedans. On convient de considérer le rayon comme en-dehors si est inclus dans l'un des deux plans limites. On a alors les deux sous-cas :
  - soit $< \vec{S}, \vec{CO} > +\alpha \geq 0$, soit $\beta - < \vec{S}, \vec{CO} > -\alpha \geq 0$, auquel cas le rayon n'intersecte pas la tranche.
  - sinon, le rayon est entièrement dans la tranche, et on génére si le niveau d'intersection est `UN` ou `TOUS` un intervalle d'intersection dont les bornes sont les bornes du segment d'étude. On affecte à ce segment son contenu (s'il est demandé) mais on ne génére évidemment pas de points d'entrée ou de sortie. Si le niveau est `ZERO`, on tronque éventuellement le segment d'étude au segment d'intersection et on rend comme résultat le nombre d'intersections trouvées, soit dans le cas présent une seule intersection.
- sinon, on calcule les abscisses des deux points d'intersection avec les plans frontières par

$$\lambda_1 = \frac{- < \vec{S}, \vec{CO} > -\alpha}{< \vec{S}, \vec{V} >}$$

$$\lambda_2 = \frac{\beta - < \vec{S}, \vec{CO} > -\alpha}{< \vec{S}, \vec{V} >}$$

  On peut alors calculer les abscisses d'entrée et de sortie :

$$\lambda_{in} = \min(\lambda_1, \lambda_2)$$

$$\lambda_{out} = \max(\lambda_1, \lambda_2)$$

  On calcule également un drapeau appelé indicateur d'échange. Ce drapeau est mis à la valeur `VRAI` si $\lambda_{in} = \lambda_2$ et à `FAUX` si $\lambda_{in} = \lambda_1$. Ceci permet de savoir par la suite quels sont les plans P_1 ou P_2 correspondant respectivement aux points d'entrée et de sortie : si l'indicateur d'échange est à `FAUX`, λ_{in} est l'abscisse d'un point du plan P_1 et λ_{out} est l'abscisse d'un point du plan P_2, sinon c'est le contraire.

  Une fois ce premier calcul effectué, on détermine si l'on doit générer ou non l'intervalle d'intersection. Ceci se fait en comparant les valeurs de λ_1, λ_2, λ_{min} et λ_{max}^{in}. Pour que l'intervalle soit utile, on doit avoir $\lambda_{in} < \lambda_{max}^{in}$ et $\lambda_{out} > \lambda_{min}$.

  On doit maintenant tenir compte de la troncature éventuelle de l'intervalle par le segment d'étude. Ceci peut s'écrire :

  si $\lambda_{in} < \lambda_{min}$
      faire $\lambda_{in} = \lambda_{min}$
      ne pas générer le point d'entrée
  si $\lambda_{out} > \lambda_{max}$
      faire $\lambda_{out} = \lambda_{max}$
      ne pas générer le point de sortie

Parvenu à ce point, on peut générer l'intervalle d'intersection si le niveau d'intersection est **UN** ou **TOUS**. On trouve dans l'environnement les informations à générer pour le contenu et les points d'entrée/sortie.

La normale peut se calculer simplement : si l'indicateur d'échange est à **FAUX**, la normale au point d'entrée est $\vec{S}$ et la normale au point de sortie est $-\vec{S}$, sinon, c'est le contraire.

Enfin, on peut affecter des numéros de surface aux deux plans de chaque tranche (comme on le verra par la suite, on n'utilise en fait que trois tranches prédéfinies pour intersecter des primitives). Là-encore, l'indicateur d'échange est utilisé pour affecter les bons numéros.

3.2.2 Intersection avec une quadrique

Nous avons regroupé toutes les fonctions d'intersection avec les quadriques sous une seule forme générale. Ceci n'est peut-être pas le plus efficace en temps de calcul mais est justifié par notre souci de condenser le code aussi bien sous forme lisible que sous forme éxécutable. En effet, un de nos objectifs étant de paralléliser cet algorithme sur un réseau de transputers ne disposant chacun que d'une mémoire de 1 mégaoctet, cette réduction de code nous permet de stocker plus de données. L'écriture est également grandement facilitée (y compris les corrections d'erreurs...).

Une quadrique est simplement définie par une matrice carrée 4×4 symétrique M. Afin que la primitive représentée par cette matrice M soit bien une quadrique, il suffit d'assurer que la sous-matrice m de M de taille 3×3 constituant le coin supérieur gauche soit non nulle.
L'intérieur de la quadrique est alors l'ensemble des points P vérifiant ${}^t\vec{CP}M\vec{CP} < 0$. Les matrices des 9 quadriques utilisées sont les suivantes :

$$M_{co} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \quad M_{cy} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \quad M_{cyh} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

$$M_{cyp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & -\frac{1}{2} & 0 \end{pmatrix} \quad M_{h1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \quad M_{cyh} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_{pa} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & -\frac{1}{2} & 0 \end{pmatrix} \quad M_{ph} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & -\frac{1}{2} & 0 \end{pmatrix} \quad M_{sp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Pour déterminer les intersections entre le rayon et la quadrique, nous reportons simplement l'expression du rayon dans l'équation de la quadrique, ce qui donne l'équation aux λ :

$$\lambda^{2t}\vec{V}M\vec{V} + 2\lambda^t\vec{V}M\vec{CO} + {}^t\vec{CO}M\vec{CO} = 0$$

qui peut se résoudre par la méthode classique de résolution des équations du second degré. On trouve donc soit une infinité de racines, soit deux racines, soit aucune racine[3].

3. Une racine double est comptée deux fois.

3.2.2.1 Cas d'une infinité de racines

Ce cas peut se produire lorsque le rayon est contenu dans la surface d'un cône par exemple, ou lorsqu'il est une des génératrices d'un paraboloïde hyperbolique ou d'un hyperboloïde à une nappe.

Nous avons pris la convention de dire dans ce cas que le rayon est en dehors de l'objet.

3.2.2.2 Cas où il n'y a pas de racines

Dans ce cas, il suffit simplement de déterminer la position d'un point quelconque du rayon par rapport à la quadrique. On peut par exemple regerder le signe du coefficient de degré deux du trinôme, c'est-à-dire considérer la valeur de ${}^t\vec{CO}M\vec{CO}$. Deux cas peuvent se présenter :

- le coefficient est strictement positif, auquel cas le rayon est entièrement hors de la quadrique.

- le coefficient est strictement négatif, auquel cas on génère si besoin un intervalle d'intersection dont les bornes sont les bornes du segment d'étude. On lui affecte si besoin son contenu mais on ne génère ni point d'entrée ni point de sortie.

3.2.2.3 Cas de deux racines

On note $\lambda_1 \leq \lambda_2$ les deux racines trouvées. Les racines sont triées par l'algorithme de résolution des équations algébriques.

Le premier traitement consiste à détecter une éventuelle racine double, auquel cas le traitement est similaire à celui du cas où il n'y pas de racines.

Il est très important de déterminer la position de la quadrique par rapport aux deux racines trouvées. En effet, puisque nous utilisons des quadriques non forcément convexes, rien ne garantit que la quadrique soit entre les racines.

La position de la quadrique par rapport aux racines est déterminée encore une fois en considérant le coefficient dominant du trinôme d'intersection. Nous allons détailler chacun des deux cas possibles. quadrique.

Dans le cas "simple" (quadrique entre les racines), on ne génère au plus qu'un intervalle d'intersection. Le traitement de ce cas est très semblable à celui de la tranche d'espace.

Dans le cas "compliqué" (quadrique hors des racines), il faut générer selon le niveau d'intersection et les positions des racines zéro, un ou deux intervalles d'intersection.

Le schéma 3.2 précise quels intervalles générer en fonction de la position de la quadrique. Ce schéma ne tient pas compte des éventuelles troncatures par le segment d'étude, ou de la non-présence de certains intervalles dans le cas où le point d'entrée a une abscisse supérieure à l'abscisse minimale d'entrée.

3.2.2.4 Calcul de la normale

La normale aux points d'intersection éventuellement trouvés peut simplement s'exprimer par :

$$\vec{N} = M\vec{CP} = M\vec{CO} + \lambda M\vec{V}$$

Une remarque cependant : lorsque la quadrique est un cône et que le point d'intersection considéré est le sommet du cône, le vecteur normal est alors nul. Mais comme cela sera expliqué dans le paragraphe suivant, ce cas ne se présente jamais (aux erreurs d'imprécision près).

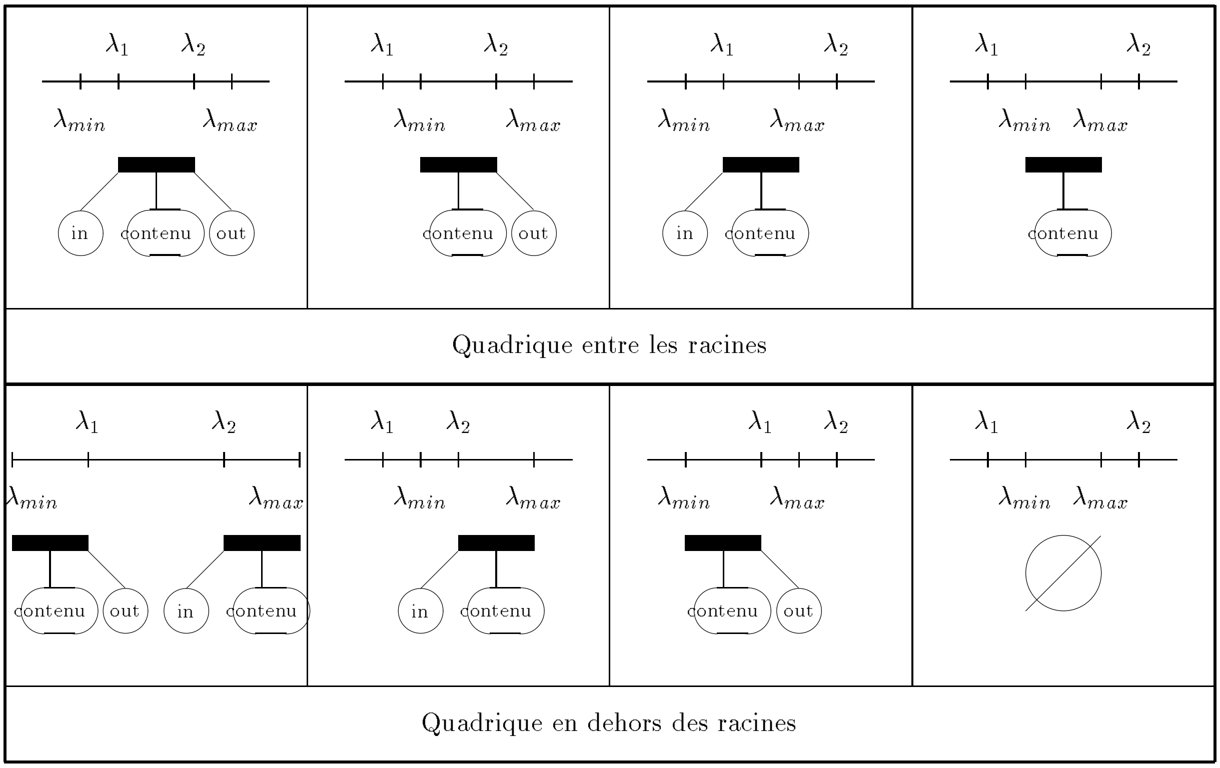


FIG. 3.2 – *Intersection avec une quadrique*

3.2.2.5 Cas particulier du cône et du cylindre

Le cône et le cylindre ont ceci de particulier qu'ils sont tous les deux limités à la portion d'espace des points vérifiant $0 \leq z \leq 1$. Ceci signifie que ce sont en fait les intersections entre les quadriques non bornées correspondantes et une tranche d'espace que noous appelons *tranche en z*.

Il serait cependant inefficace de gérer cette intersection comme une vraie intersection entre objets. Nous avons donc préféré la méthode suivante :

– on intersecte d'abord la tranche en z en ne demandant aucun intervalle (en affectant `ZERO` au niveau d'intersection) et en demandant la troncature du segment d'étude comme résultat. S'il n'y a pas intersection, on est sûr d'être en dehors. Sinon, on récupère dans l'environnement les indicateurs qui précisent s'il y a effectivement un point d'entrée ou un point de sortie ainsi que l'indicateur d'échange éventuel.

– Ensuite, on tronque le segment d'étude par l'intervalle d'intersection trouvé avec la tranche en z et on intersecte la quadrique en demandant cette fois de calculer tout ce qui est nécessaire.

– Enfin, on teste si l'on a généré un point d'entrée dans la quadrique. Si oui, il n'y a rien de plus à faire. Dans le cas contraire, on regarde les indicateurs générés lors de l'intersection avec la tranche en z. S'il y a un point d'entrée dans la tranche, on génere le point d'entrée ainsi que la normale correspondante (dont on trouve l'orientation grâce à l'indicateur d'échange). On procède de façon identique avec le point de sortie.

3.2.3 Intersection avec un cube

Encore une fois, nous allons utiliser les tranches d'espace. En effet, un cube peut être considéré comme l'intersection de trois tranches, la première d'équation $0 \leq x \leq 1$ (que l'on nomme *tranche en x*), la deuxième d'équation $0 \leq y \leq 1$ (que l'on nomme *tranche en y*) et la dernière étant la tranche en z déjà citée.

Comme nous l'avons expliqué pour le cône et le cylindre, il serait inefficace de gérer cette intersection entre tranches comme une vraie intersection d'objets. La méthode est la suivante :

- on utilise un tableau statique contenant les trois tranches.
- pour chaque tranche i ($i \in \{0, 1, 2\}$), on intersecte le rayon avec la tranche avec `ZERO` comme valeur de niveau d'intersection. S'il n'y a pas intersection, on peut s'arrêter et rendre le résultat. Sinon, s'il y a un point d'entrée, on affecte la valeur de i comme numéro de tranche d'entrée, s'il y a un point de sortie, on affecte la valeur de i comme numéro de tranche de sortie, et on conserve également un indicateur d'échange de tranche d'entrée et un indicateur d'échange de tranche de sortie.
- une fois les trois tranches intersectées, on sait qu'il y a effectivement intersection et on peut générer si besoin l'intervalle correspondant. Les normales aux points d'entrée et de sortie peuvent être calculées grâce au numéros de tranches d'entrée et de sortie, en consultant bien sûr les éventuels indicateurs d'échange. Les numéros de surface d'entrée et de sortie sont gérés de la même façon.

Encore une fois, on peut noter qu'il n'y a pas d'ambigüité sur les attributs à affecter pour les différentes tranches, puisque ce sont les mêmes qui sont valables pour le cube tout entier.

3.2.4 Intersection avec un tore

Le tore est une primitive qu'il est facile de rajouter dans un modeleur car il a une équation algèbrique de degré 4, et on possède des techniques de résolution exactes de ce type d'équation (méthode de Ferrari).

Plus précisément, l'équation d'un tore de rayon r est

$$(x^2 + y^2 + z^2 + 1 - r^2)^2 = 4(x^2 + y^2)$$

l'intérieur du tore étant déterminé par

$$(x^2 + y^2 + z^2 + 1 - r^2)^2 \leq 4(x^2 + y^2)$$

On peut aussi écrire cette équation

$$(||\vec{CP}||^2 + 1 - r^2)^2 = 4(||\vec{CP}||^2 - z^2)$$

On paramétrise le point courant du rayon par $P = O + \lambda \vec{V}$ et on reporte cette expression dans l'équation précédente. On obtient une équation du type

$$a_4 \lambda^4 + a_3 \lambda^3 + a_2 \lambda^2 + a_1 \lambda + a_0 = 0$$

avec

$$\begin{aligned}
a_4 &= ||\vec{V}||^4 \\
a_3 &= 4 < \vec{CO}, \vec{V} > ||\vec{V}||^2 \\
a_2 &= 4 < \vec{CO}, \vec{V} >^2 + 2\,(||\vec{CO}||^2 - (1 + r^2)) + 4\,V_z^2 \\
a_1 &= 4 < \vec{CO}, \vec{V} > (||\vec{CO}||^2 - (1 + r^2)) + 8\,O_z V_z \\
a_0 &= (||\vec{CO}||^2 - (1 - r)^2)(||\vec{CO}||^2 - (1 + r)^2) + 4\,O_z^2
\end{aligned}$$

On effectue ensuite la résolution de l'équation algébrique et trois cas peuvent alors se présenter : soit on a 4 racines, soit 2, soit aucune[4]. Nous allons détailler maintenant le traitement dans chacun des cas.

4. une racine double est comptée deux fois

3.2.4.1 Cas de 4 racines

On doit tout d'abord vérifier que l'on a effectivement affaire à 4 racines, ce qui se fait en plusieurs étapes. On suppose que l'algorithme de résolution donne des racines $\lambda_1 \leq \lambda_2 \leq \lambda_3 \leq \lambda_4$.

Le premier traitement permet d'éliminer les cas où toutes les racines sont en dehors du segment d'étude, ce qui peut s'écrire :

> si $\lambda_4 \leq \lambda_{min}$ ou $\lambda_1 \geq \lambda_{max}^{in}$
>   il n'y a pas d'intersection
>   STOP

Le deuxième traitement consiste à éliminer d'éventuelles racines doubles, ce que l'on peut écrire :

> si $\lambda_1 = \lambda_2$
>   faire $\lambda_1 = \lambda_3$
>   faire $\lambda_2 = \lambda_4$
>   continuer comme s'il y avait 2 racines
> si $\lambda_2 = \lambda_3$
>   faire $\lambda_2 = \lambda_4$
>   continuer comme s'il y avait 2 racines
> si $\lambda_3 = \lambda_4$
>   continuer comme s'il y avait 2 racines

Le troisième traitement permet d'éliminer, dans le cas où $r > 1$ d'éventuelles intersections avec les morceaux de surface internes au tore. Dans ce cas, au point $P_2 = 0 + \lambda_2 \vec{V}$, on a la relation

$$||\vec{CP_2}||^2 + 1 - r^2 \leq 0$$

soit encore

$$\lambda_2^2 ||\vec{V}||^2 + 2 < \vec{CO}, \vec{V} > + ||\vec{CO}||^2 + 1 - r^2 \leq 0$$

L'algorithme peut alors s'écrire :

> si la condition est vraie
>   faire $\lambda_2 = \lambda_4$
>   continuer comme s'il y avait 2 racines

Arrivé à ce point, on est sûr qu'il a quatre vraies intersections, et comme le tore est une primitive bornée, on sait que les éventuels intervalles d'intersection sont $[\lambda_1, \lambda_2]$ et $[\lambda_3, \lambda_4]$. On doit alors tester la position de ces intervalles par rapport au segment d'étude, ce qui peut s'écrire :

> si $\lambda_2 \leq \lambda_{min}$
>   si $\lambda_3 \geq \lambda_{max}^{in}$
>   il n'y a pas d'intersection
>   STOP
>   sinon
>   faire $\lambda_1 = \lambda_3$
>   faire $\lambda_2 = \lambda_4$
>   continuer comme s'il y avait 2 racines
> sinon
>   si $\lambda_3 \geq \lambda_{max}^{in}$
>   continuer comme s'il y avait 2 racines

Arrivé à ce point, on est sûr qu'il y a deux intervalles d'intersection potentiels. On les génère si besoin, en prenant soin de ne générer le second que s'il est vraiment utile, c'est-à-dire si le niveau vaut `TOUS`. Il faut également prendre soin à la troncature éventuelle des intervalles par le segment d'étude. Tout ceci peut s'écrire :

si $\lambda_1 < \lambda_{min}$
    `le premier intervalle n'a pas de point d'entrée`
si le deuxième intervalle est nécessaire
    si $\lambda_4 > \lambda_{max}$
        `le deuxième intervalle n'a pas de point de sortie`

3.2.4.2 Cas de 2 racines

Ce cas est identique au cas de la tranche d'espace ou de la quadrique. On ne détaillera donc pas le traitement.

3.2.4.3 Calcul de la normale

Celle-ci peut se calculer en un point P du tore par

$$\vec{N}(P) = (||\vec{CP}||2 + 1 - r^2)\vec{CP} - 4(\vec{CP} - z\vec{k})$$

3.3 Gestion des transformations affines

Nous avons vu comment nous intersectons des primitives. Il nous faut maintenant détailler comment nous intersectons des objets, c'est-à-dire des nœuds de type transformation affine ou opération booléenne.

En ce qui concerne les transformations affines, Roth [ROTH 82] avait déjà précisé le traitement à effectuer.On effectue en fait deux traitements : le premier permet de modifier le rayon pour calculer les intersections, le deuxième s'effectue a posteriori pour modifier les normales sur les intervalles d'intersection trouvés. L'algorithme d'intersection peut donc s'écrire :

transformer le rayon
`intersecter l'objet sur lequel est appliqué la transformation`
s'il y a des intersections
    pour chaque intervalle d'intersection
        s'il y a une cellule-point d'entrée
            transformer la normale d'entrée
        s'il y a une cellule-point de sortie
            transformer la normale de sortie

Détaillons maintenant chacun de ces deux traitements.

3.3.1 Transformation du rayon

Le résultat fondamental de transformation du rayon est le suivant :

> Soit A un objet, T une transformation affine, $R = (O, \vec{V})$ un rayon et I un intervalle. Alors I est un intervalle d'intersection entre R et $T(A)$ si et seulement si I est un intervalle d'intersection entre A et le rayon $R' = T^{-1}(R) = (T^{-1}(O), T^{-1}(\vec{V}))$.

Le traitement à effectuer avant de calculer les intersections avec l'objet que l'on transforme est donc simple : il suffit d'appliquer à l'origine et à la direction du rayon la transformation inverse.

On peut d'ailleurs noter que rien d'autre n'est transformé,en particulier ni le segment d'étude, ni les abscisses maximales d'entrée.

3.3.2 Transformation des normales

Ce traitement est encore une fois très simple, et peut être précisé par le résultat suivant :

> Soit $\vec{n}$ la normale sortante au point d'intersection entre le rayon R' et l'objet A. Alors la normale extérieure au point d'intersection entre le rayon R et l'objet $T(A)$ est ${}^tT^{-1}(\vec{n})$.

Remarque 1

> Les résultats que nous venons d'énoncer peuvent bien sûr s'appliquer aux matrices globales, qui représentent des transformations affines compactées.

Remarque 2

> Les résultats énoncés ci-dessus expliquent pourquoi on conserve dans les nœuds de type transformation affine la matrice inverse de la transformation : c'est celle-ci qui est utile, aussi bien pour la transformation du rayon que pour la transformation des normales.

3.4 Opérations booléennes

La gestion des opérations booléennes est relativement simple dans son principe : pour réaliser une intersection entre un rayon et un objet de type opération booléenne, on réalise l'intersection entre le rayon et chacun des opérandes, puis on combine les listes d'intervalles d'intersection trouvées.

Il y a cependant deux paramètres importants dont il faut tenir compte : la notion d'homogénéité et la notion de priorité horizontale.

Pour tous les types d'opérations, l'algorithme de combinaison parcourt les deux listes d'intervalles d'intersection, et calcule la liste résultante. On peut d'ailleurs noter que l'algorithme utilise la liste d'intervalles de l'opérande gauche pour stocker le résultat. La liste droite est libérée en fin d'algorithme et il faut donc prendre garde à ce que cette liste ne partage pas de données avec la liste résultat.

D'un point de vue pratique, les listes d'intervalles d'intersection sont stockées sous forme de listes chaînées, et on utilise un pointeur courant pour chacune des listes. A chaque étape de l'algorithme, on détermine l'intervalle résultant de la combinaison des deux intervalles pointés par les pointeurs courants. Puis on incrémente l'un ou l'autre des pointeurs, voire même les deux selon les positions relatives.

3.4.1 Positions relatives de deux intervalles

Pour chaque couple d'intervalles d'intersection (un avec l'opérande gauche et un avec l'opérande droit), il est important de déterminer la position relative de l'un par rapport à l'autre. On peut en fait trouver 11 positions relatives différentes, que nous allons détailler maintenant.

On note λ_{in}^G et λ_{out}^G les abscisses d'entrée et de sortie de l'intervalle gauche et λ_{in}^D et λ_{out}^D les abscisses d'entrée et de sortie de l'intervalle droit. Les différentes positions sont les suivantes :

- `G_ET_D_DISJOINTS` : ceci équivaut à $\lambda_{out}^G \leq \lambda_{in}^D$.
- `D_ET_G_DISJOINTS` : ceci équivaut à $\lambda_{out}^D \leq \lambda_{in}^G$.
- `G_AVANT_D` : ceci équivaut à $\lambda_{in}^G < \lambda_{in}^D < \lambda_{out}^G < \lambda_{out}^D$.
- `D_AVANT_G` : ceci équivaut à $\lambda_{in}^D < \lambda_{in}^G < \lambda_{out}^D < \lambda_{out}^G$.
- `G_CONTIENT_D` : ceci équivaut à $\lambda_{in}^G < \lambda_{in}^D \leq \lambda_{out}^D < \lambda_{out}^G$.
- `D_CONTIENT_G` : ceci équivaut à $\lambda_{in}^D < \lambda_{in}^G \leq \lambda_{out}^G < \lambda_{out}^D$.
- `G_G_CONTIENT_D` : ceci équivaut à $\lambda_{in}^G = \lambda_{in}^D < \lambda_{out}^D < \lambda_{out}^G$.

- G_D_CONTIENT_G : ceci équivaut à $\lambda_{in}^G = \lambda_{in}^D < \lambda_{out}^G < \lambda_{out}^D$.
- G_CONTIENT_D_D : ceci équivaut à $\lambda_{in}^G < \lambda_{in}^D < \lambda_{out}^D = \lambda_{out}^G$.
- D_CONTIENT_G_D : ceci équivaut à $\lambda_{in}^D < \lambda_{in}^G < \lambda_{out}^D = \lambda_{out}^G$.
- G_EGAL_D : ceci équivaut à $\lambda_{in}^D = \lambda_{in}^G \leq \lambda_{out}^D = \lambda_{out}^G$.

La fonction qui détermine les positions relatives est employée très fréquemment, elle a donc tout intérêt à être la plus efficace possible. La position relative peut en fait se calculer par 4 tests seulement, comme l'indique la fonction suivante.

```
#define G_ET_D_DISJOINTS 15
#define D_ET_G_DISJOINTS 0
#define G_AVANT_D        14
#define D_AVANT_G        4
#define G_CONTIENT_D     6
#define D_CONTIENT_G     12
#define G_G_CONTIENT_D   5
#define G_D_CONTIENT_G   13
#define G_CONTIENT_D_D   10
#define D_CONTIENT_G_D   8
#define G_EGAL_D         9

unsigned int position_relative (gauche,droit)
{
unsigned int res=0;

if (droit->absc_in > gauche->absc_in)
        if (droit->absc_in >= gauche->absc_out)
                return(G_ET_D_DISJOINTS);
        else
                res+=2;
else
        if (droit->absc_in==gauche->absc_in)
                res+=1;
        else
                res+=0;
if (droit->absc_out >= gauche->absc_out)
        if (droit->absc_out==gauche->absc_out)
                res+=8;
        else
                res+=12;
else
        if (droit->absc_out <= gauche->absc_in)
                return(D_ET_G_DISJOINTS);
        else
                res+=4;
return(res);
}
```

La numérotation peut sembler bizarre mais elle correspond en fait à la chose suivante :

- la position de λ_{in}^D par rapport aux abscisses de l'intervalle gauche peut prendre 4 valeurs.
  - 0 si $\lambda_{in}^D < \lambda_{in}^G$
  - 1 si $\lambda_{in}^D = \lambda_{in}^G$
  - 2 si $\lambda_{in}^G < \lambda_{in}^D < \lambda_{out}^G$
  - 3 si $\lambda_{in}^D \geq \lambda_{out}^G$
- la position de λ_{out}^D par rapport aux abscisses de l'intervalle gauche peut aussi prendre 4 valeurs.
  - 0 si $\lambda_{out}^D \leq \lambda_{in}^G$
  - 4 si $\lambda_{in}^G < \lambda_{out}^D < \lambda_{out}^G$
  - 8 si $\lambda_{out}^D = \lambda_{out}^G$
  - 12 si $\lambda_{out}^D > \lambda_{out}^G$
- la position relative des deux intervalles se calcule alors en ajoutant les deux valeurs précédentes [5].

Pour calculer la position d'une abscisse par rapport aux deux autres, il suffit alors d'au plus 2 tests, soit en tout un maximum de 4 tests.

3.4.2 Gestion de la priorité horizontale

C'est en effet lors de la combinaison des listes d'intervalles d'intersections que l'on tient compte de cette priorité horizontale.

La priorité est stockée sur chaque nœud de type intersection ou réunion (elle n'a pas de sens pour une différence) sous forme de deux variables (une pour chaque type d'objet) pouvant prendre 4 valeurs :

- `GAUCHE` si l'objet gauche est prioritaire.
- `DROIT` si l'objet droit est prioritaire.
- `FUSION` si l'opération booléenne a été déclarée comme étant homogène.
- `MELANGE` si l'opération booléenne mélange les propriétés des objets neutres.

Les valeurs par défaut de ces priorités sont `GAUCHE` en ce qui concerne les objets actifs et `MELANGE` en ce qui concerne les objets neutres. Ces valeurs peuvent bien sûr être redéfinies.

Lors de l'algorithme de combinaison des intervalles, pour chaque couple d'intervalles, on calcule une priorité "locale" de la façon suivante : si les objets sont de même type, la priorité locale prend la valeur de la priorité pour ce type d'objet, si les objets sont de type différents, c'est l'objet de type actif qui est prioritaire.

3.4.3 Différence

Ce cas est peut-être le plus simple à traiter. En effet, la notion de priorité horizontale n'y intervient que par le fait que ls objets actifs sont prioritaires par rapport aux neutres. Ainsi, un objet neutre ne peut trouer un objet actif. En toute rigueur, un objet neutre peut trouer un autre neutre : cependant, un message signale un tel événement car il semble en effet peu logique et sans grande signification physique.

De plus, seule la liste gauche est modifiée lors d'une différence. En effet, aucun intervalle résultat ne correspond à une intersection avec l'objet que l'on retranche !

5. On peut noter au passage que les valeurs 1, 2, 3, 7 et 11 sont des valeurs incohérentes

Enfin, ces modifications concernent soit la disparition complète d'un intervalle (lorsque l'intervalle droit contient le gauche) soit la modification des points d'entrée ou de sortie. Le contenu d'un intervalle n'est quant à lui jamais modifié.

Pour effectuer ces modifications, il suffit simplement de modifier les valeurs des abscisses d'entrée ou de sortie, ainsi que les cellules-point correspondantes. Il suffit d'inverser la normale de la cellule-point de l'objet retranché et on peut alors simplement échanger les cellules-point. En effet, un intervalle d'intersection avec l'objet retranché n'intervient au plus que sur un intervalle de l'objet à qui l'on retranche. On est donc certains de ne jamais perdre d'informations.

3.4.4 Intersection

Ce cas est un peu plus compliqué puisqu'il faut ici tenir compte de façon plus importante de la priorité horizontale.

Une première remarque que l'on peut faire est le fait que deux intervalles d'intersection ne vont générer un résultat que si les deux objets intersectés sont de même type et que si les deux intervalles ont une intersection.

Ensuite, pour économiser le temps d'exécution, on choisit de conserver le plus petit des deux intervalles (ou le premier des deux si aucun ne contient l'autre). Il faut bien sûr tenir compte de la priorité horizontale et affecter correctement le contenu de l'intervalle résultat (ou les contenus en cas d'objets neutres).

Enfin, dans le cas où aucun intervalle ne contient l'autre, il faut tronquer l'intervalle résultat et ne pas oublier d'affecter les bonnes cellules-point.

3.4.5 Union

C'est l'opération booléenne la plus compliquée. En effet, il faut tenir compte de la priorité horizontale et de la fusion.

La fusion peut se réaliser assez simplement puisque dans le cas où les deux intervalles n'ont pas d'intersection, il suffit d'incrémenter le pointeur courant du premier intervalle. S'il y a intersection, il faut combiner les deux intervalles : on remplace le premier des deux intervalles par la combinaison des deux. Il suffit donc de modifier l'abscisse de sortie ainsi que l'éventuelle cellule-point de sortie.

Dans le cas d'une union à priorité entre deux objets actifs, ceci équivaut à retrancher le plus prioritaire du moins prioritaire, et à effectuer ensuite la réunion entre ces deux intervalles disjoints : dans le cas où le moins prioritaire est entièrement contenu dans le plus prioritaire, le premier disparaît.

3.5 Clignotant et variante

Ces deux objets sont quant à eux extrêmement simples à intersecter.

Pour intersecter un clignotant, il suffit d'intersecter l'objet si le temps courant est compris entre l'instant d'apparition et l'instant de disparition de l'objet, sinon l'intersection donne toujours un résultat vide.

Pour une variante, il suffit d'intersecter l'objet correspondant à la définition pour le temps courant.

Chapitre 4

Boîtes englobantes

4.1 Introduction

L'un des principaux problème pour un tracé de rayons est le temps de calcul : l'utilisateur d'un tel programme désire souvent avoir un résultat dans des temps “acceptables”. Or, si l'on considère une scène moyenne contenant mille objets, et que l'on calcule une image en résolution 500×500 (ce qui est une résolution moyenne), le nombre d'intersections rayon-objet atteint le nombre respectable de 250 millions uniquement pour les rayons primaires. Même les machines puissantes actuelles passent donc un temps non négligeable sur de tels calculs. Il est alors naturel d'envisager de réduire ce temps de calcul.

L'une des techniques possibles consiste á diminuer le nombre de rayons à calculer. Par exemple, on commence par générer une image en résolution inférieure à la résolution demandée, puis on ne raffine l'image que dans les parties de l'image où les variations entre les pixels calculés sont importantes. On peut ainsi varier entre un sous-échantillonage de 1 sur 64 à un sur-échantillonage de l'ordre de 16 pour 1, ce qui permet aussi de régler une partie du problème de l'aliassage des images générées.

Une autre technique consiste à diminuer le temps de calcul en ne considérant des intersections que avec des objets ayant une forte probabilité d'intersecter le rayon considéré. Il se pose alors le problème de la classification des objets en fonction de leur probabilité d'intersection avec un rayon. La plupart du temps, cette classification est effectuée dans une phase de pré-traitement de l'algorithme et apporte des gains non négligeables.

Nous ne présenterons dans ce chapitre que des méthodes relevant de la seconde catégorie, car l'une des extensions apportée à notre logiciel relève précisément de la première catégorie : les personnes intéressées pourront trouver les informations nécessaires dans [MCP92].

Plus précisément, nous allons présenter les techniques habituellement utilisées pour accélérer les calculs.

4.2 Techniques classiques

4.2.1 Essai de classification

Je reprends ici la classification introduite par Excoffier [Exc88]. Celle-ci classe les méthodes en deux grands groupes : les méthodes à englobants et les méthodes à partition.

4.2.1.1 Méthodes à englobants

Le principe des méthodes à englobants est d'associer à chaque objet une structure de données qui permet d'effectuer un test rapide pour savoir si le rayon ne possède pas d'intersection avec l'objet. Cette structure définit en général un volume de forme simple qui contient l'objet : la simplicité de la forme entraîne la rapidité du test d'intersection.

Ainsi, si p est la probabilité d'intersecter l'englobant, si t est le temps nécessaire pour intersecter l'englobant et T le temps nécessaire pour intersecter l'objet réel, le temps moyen pour intersecter l'objet avec son englobant est $t + pT$. Le rapport entre ce temps moyen et le temps T est donc $k = p + \frac{t}{T}$. Le temps t peut être considéré comme constant (bien que cela ne soit pas strictement vrai dans notre cas, on peut au moins en donner une borne t_0). On constate alors que si T est très supérieur à t_0, le gain de temps est en moyenne de p. Ainsi, on obtient le résultat qu'un englobant est intéressant si sa probabilité d'intersection est faible par rapport à 1. Il est donc primordial d'avoir des englobants les "meilleurs" possibles, c'est-à-dire ayant le moins de probabilité d'être intersectés.

Il faut d'ailleurs noter que l'utilisation des englobants ne se fait pas à un seul niveau : les objets (ou plus exactement leurs englobants) sont ensuite le plus souvent hiérarchisés [Whi80]. Cette hiérarchie est d'ailleurs naturelle pour un modèle CSG (c'est la hiérarchie de construction), et dans le cas d'autres modèles (non-CSG), cela revient à considérer que l'on effectue des unions entre les objets. Dans ce dernier cas, on peut d'ailleurs utiliser des méthodes de hiérarchisation regroupant d'abord les objets proches.

Diverses formes d'englobants ont été essayées, et on peut citer parmi celles-ci les sphères [Whi80], les boîtes (c'est-à-dire des parallélépipèdes dont les côtés sont parallèles aux axes du repère du monde) [Rot82], puis comme extensions des ces formes les ellipsoïdes [WHG84][Bou85], les boîtes dont les côtés sont parallèles à un nombre limité de directions [KK86] et les polyèdres convexes [DS84]. Bien entendu, lorsque l'on utilise une forme plus complexe, on gagne en précision (la probabilité d'intersecter l'objet lorsque l'on intersecte l'englobant croît, la probabilité d'intersecter l'englobant décroît)) mais ceci au prix d'un surcoût en mémoire nécessaire au stockage de la structure de données et en temps de calcul d'intersection avec l'englobant (t plus important). Il y a donc toujours un compromis à trouver.

On a également utilisé des englobants plans, obtenus après projection de la scène sur un plan. On se ramène alors à l'intersection de la projection du rayon avec les boîtes planes, qui sont la plupart du tant des rectangles. Un cas très intéressant est celui où le plan de projection est le plan de l'écran [BWJ84]. Un rayon primaire se projette alors en un point et on peut connaître directement tous les objets susceptibles d'être intersectés. Les rayons non primaires sont moins intéressants mais la technique reste valable.

Dans [Arg88], on utilise un algorithme de Bentley-Ottman pour décomposer ces englobants en morceaux uniformes, c'est-à-dire en rectangles ne comportant aucun sous-englobant. A chaque rectangle est associée une scène réduite, obtenue en regroupant les objets dont l'englobant contient ce rectangle et en utilisant de plus un élagage de l'arbre. Pour les rayons non primaires, on applique une technique similaire en utilisant trois plans formant un système orthogonal. Pour un rayon déterminé, on choisit le plan tel que la projection du rayon sur ce plan soit de longueur minimale : on a ainsi moins de rectangles à tester.

4.2.1.2 Méthodes à partition

Dans ces méthodes, on cherche à partitionner un ensemble au sens de la décomposition en sous-ensembles disjoints. On peut distinguer trois types de partitions : les partitions spatiales, les partitions planes et les partitions de l'espace des rayons.

Dans le cas des partitions spatiales, on partitionne l'espace de modélisation, puis on indique pour chaque partition les objets qui lui appartiennent. Lors du calcul de l'intersection d'un rayon avec la scène, on commence par calculer l'intersection du rayon avec la partition, ce qui permet de déterminer le premier élément de la partition traversé par le rayon. On teste alors les intersections entre le rayon et les objets appartenant à cet élément : si une intersection est trouvée, on s'arrête sinon on détermine l'élément suivant et on continue. Cette technique de partition est donc le plus souvent associée à un parcours incrémental de l'espace.

<table>
<tr><td rowspan="6">ENGLOBANTS</td><td rowspan="4">ESPACE</td><td colspan="2">Boîtes</td><td>[Rot82]</td></tr>
<tr><td colspan="2">Sphères</td><td>[Whi80]</td></tr>
<tr><td colspan="2">Ellipsoïdes</td><td>[WHG84][Bou85]</td></tr>
<tr><td colspan="2">Polyèdres</td><td>[DS84][KK86]</td></tr>
<tr><td rowspan="2">PLAN</td><td colspan="2">Sol</td><td>[Coq84]</td></tr>
<tr><td colspan="2">Ecran</td><td>[BWJ84][Arg88]</td></tr>
<tr><td rowspan="8">PARTITION</td><td rowspan="4">$\mathbb{R}^3$</td><td colspan="2">Pavage</td><td>[Mul85][FTI86]</td></tr>
<tr><td colspan="2">BSP</td><td>[Mul86][BPA88]</td></tr>
<tr><td colspan="2">Octree</td><td>[Gla84]</td></tr>
<tr><td colspan="2">Macro-régions</td><td>[PD88][Dev90]</td></tr>
<tr><td>$\mathbb{R}^5$</td><td colspan="2"></td><td>[AK87]</td></tr>
<tr><td rowspan="3">$\mathbb{R}^2$</td><td rowspan="2">Ecran</td><td>pavage</td><td>[Arg88]</td></tr>
<tr><td>quadtree</td><td>[Ama84]</td></tr>
<tr><td colspan="2">Lampe</td><td>[Arg88]</td></tr>
</table>

TAB. 4.1 – *Classification des méthodes d'accélération*

Plusieurs types de partition spatiale ont été utilisés et on peut citer le pavage régulier (partition en cubes identiques ou voxels) [Mul85][FTI86], la partition à l'aide d'arbres séparants (BSP) [Mul86][BPA88], ou l'utilisation d'arbres octants (octree) [Gla84]. Les structures régulières facilitent en général le parcours incrémental, alors que les structures non régulières permettent une meilleure adaptativité des algorithmes de subdivision.

On peut également classer dans cette catégorie la méthode des macro-régions [PD88][Dev90]. Cette méthode consiste à effectuer une décomposition en voxels, puis à regrouper les voxels voisins ne contenant aucun objet. Ces regroupements sont appelés des macro-régions. Lorsque l'algorithme de parcours incrémental arrive sur une macro-région, on sait que le rayon va la traverser sans aucune intersection.

On peut également utiliser des partitions planes, obtenues après projection de la scène sur un plan. Chaque objet est alors associé à une partie de l'espace projeté, et l'on effectue de la même façon une projection du rayon. Là encore, on utilise souvent un parcours incrémental de la partition pour effectuer le calcul de l'intersection.

On peut citer parmi ces techniques la projection sur le plan de l'écran (technique particulièrement efficace pour les rayons primaires), ou sur des plans normaux aux directions de propagation de la lumière en provenance des sources (technique adaptée aux rayons d'ombre) [Arg88]. La partition peut alors se présenter sous forme d'un pavage régulier [ET87], ou d'un arbre quadrant (quadtree) [Ama84]. Cette technique est aussi utilisable lorsque les objets à visualiser sont d'un type particulier, par exemple dans le cas d'un terrain se comportant pas de surplomb : il est alors naturel de projeter le terrain sur un plan horizontal et le suivi incrémental du rayon s'apparente alors à un algorithme de Bresenham [Coq84].

Enfin, la dernière technique de partition, introduite par Arvo et Kirk [AK87], est en fait une double partition. On commence par effectuer une partition de l'espace des rayons : chaque rayon est considéré comme élément d'un espace à cinq dimensions (trois pour l'origine du rayon et deux pour sa direction), et cet espace est partitionné. A chaque élément de la partition, on associe les objets de la scène que les rayons appartenant à cet élément ont une chance d'intersecter. Lors du calcul de l'intersection d'un rayon avec la scène, on commence par classifier ce rayon puis on l'intersecte avec les objets candidats. Cette technique permet d'associer des partitions plus fines pour les rayons issus des sources lumineuses ou de l'oeil et est donc bien adaptée aux rayons primaires et d'ombre.

4.2.2 Inconvénients de ces techniques

L'inconvénient principal que l'on peut trouver à certaines de ces techniques est leur inutilisabilité pour notre modèle.

En effet, nous utilisons la possibilité de réutiliser les objets et d'introduire des transformations affines à n'importe quel niveau de l'arbre. Comme nous transformons le rayon en cours de calcul, ceci rend très difficile (et surtout très inefficace) l'utilisation de techniques à parcours incrémental du rayon.

De même, ainsi qu'on le verra dans le chapitre 6, les perspectives que nous utilisons rendent parfois très difficile les projections sur l'écran.

De plus, pour des modèles complexes, les techniques de partition deviennent très coûteuses en espace mémoire si l'on veut conserver une certaine efficacité : une partition est efficace si elle diminue beaucoup le nombre d'objets à tester, et s'il y a beaucoup d'objets, il faut donc des partitions très précises.

Enfin, il nous a semblé préférable d'utiliser une même technique pour tous les types de rayons et ceci pour deux raisons essentielles :

- ceci diminue l'espace mémoire nécessaire au stockage des structures de données et également du code spécifique pour différentes méthodes d'accélération (rappelons que la concision du code et des données est un de nos objectifs prioritaires).
- nous n'avons aucune idée a priori sur la répartition des rayons entre les différents types (primaire, secondaire ou d'ombre) et nous refusons donc de choisir une méthode efficace pour un seul type de rayons.

Une technique à englobants semble donc bien mieux adaptée à un modèle tel que celui que nous utilisons car un englobant est une donnée "locale" de l'objet (elle est propre à l'objet et non à l'instance de l'objet, c'est-à-dire l'utilisation qui en est faite dans la scène).

Cependant, la plupart de ces techniques présentent également des inconvénients notables. Prenons l'exemple où les englobants sont des boîtes à côtés parallèles aux axes du repère du monde et exhibons deux exemples simples de perte notoire d'efficacité :

- un cube de côté 1 (donc de volume 1) dont la diagonale principale est colinéaire à l'un des axes du repère possède une boîte englobante minimale dont le volume est $\frac{2}{3}(2\sqrt{3}+3)$, soit plus de 4 fois plus importante que le cube initial (voir en annexe B la démonstration de ce résultat).
- l'union de deux cubes identiques de côté ε dont les centres sont situés respectivement en $(0,0,0)$ et $(1,1,1)$ possède une boîte englobante de volume égal à 1 soit $\frac{1}{2\varepsilon^3}$ plus importante que le volume cumulé des deux cubes.

Nous avons donc choisi comme méthode d'accélération une méthode à englobants, que nous appelons *boîtes englobantes généralisées* pour remédier à tout ou partie de ces inconvénients.

4.3 Boîtes englobantes généralisées

Nous allons tout d'abord présenter dans cette section la notion de *tranche d'espace*, qui est la notion de base de notre algorithme, puis celle de boîte généralisée. Nous indiquerons ensuite comment ces boîtes sont calculées, et tout particulièrement la gestion des transformations affines et des opérations booléennes.

4.3.1 Tranche d'espace

Rappelons ce concept introduit au chapitre 3 pour réaliser les intersections avec un cube.

Une tranche d'espace est par définition une partie de l'espace comprise entre deux plans parallèles. Elle est complètement définie par la donnée d'un vecteur $\vec{V}$ de $\mathbb{R}^4$ et d'une constante négative δ. A un point p de l'espace correspond un vecteur $\vec{P}$ de $\mathbb{R}^4$. Le point p appartient à la tranche si et seulement si $\delta \leq < \vec{V}, \vec{P} > \leq 0$ où $< ., . >$ désigne le produit scalaire de $\mathbb{R}^4$.

4.3.2 Boîte généralisée

Une boîte généralisée est simplement définie comme étant l'intersection de trois tranches d'espace. Les trois tranches ont comme seule contrainte d'avoir des vecteurs qui forment un système libre. En particulier, rien n'impose aux tranches d'avoir des vecteurs deux à deux orthogonaux, ou encore d'être colinéaires à des directions privilégiées.

Ainsi, la définition et le stockage de tels englobants est assez simple : il suffit de prévoir le stockage de trois vecteurs de $\mathbb{R}^4$ et de trois constantes, soit quinze valeurs au total.

Enfin, la fonction de calcul d'intersection avec une boîte est très semblable à celle utilisée pour un cube, et consiste à intersecter successivement les trois tranches en tronquant à chaque intersection le segment d'étude. Aucun intervalle d'intersection n'est cependant généré.

4.3.3 Transformations affines des boîtes

Le résultat essentiel est le suivant : si $(\vec{V}, \delta)$ est une tranche d'espace contenant un objet A, si T est une transformation affine, alors la tranche d'espace $({}^tT^{-1}(\vec{V}), \delta)$ est une tranche d'espace contenant $T(A)$.

Ainsi, il est très simple de calculer la boîte du transformé d'un objet : c'est tout simplement la boîte dans les trois tranches sont obtenues à partie des tranches de la boîte initiale par la transformation indiquée ci-dessus.

On peut alors constater que, en dehors des imprécisions dûes à la machine (erreurs d'arrondis dans les calculs), il n'y a aucune perte en précision lors des transformations affines sur les englobants.

4.3.4 Opérations booléennes sur les boîtes

Dans cette section, G désigne l'opérande gauche de l'opération booléenne et D l'opérande droit.

4.3.4.1 Union

C'est certainement le cas le plus complexe. Le principe de l'algorithme est le suivant :

- on commence par déterminer les huit sommets de la boîte de G et les huit sommets de la boîte de D.
- on effectue une phase de marquage, qui consiste à marquer les tranches de G qui contiennent les huits points de la boîte de D, et celles de D qui contiennent les huits points de la boîte de G.
- plusieurs cas peuvent alors se présenter :
  - l'une des boîtes a toutes ses tranches marquées, ce qui signifie que cette boîte contient l'autre, et l'union est simplement la plus grande des deux.
  - l'une des boîtes possède deux tranches marquées, auquel cas on fabrique une nouvelle boîte dont les deux premières tranches sont les tranches marquées et dont la troisième est obtenue à partir de la tranche non marquée en l'agrandissant de façon à ce qu'elle contienne les huits points de l'autre boîte.

- aucune des tranches n'est marquée, auquel cas on utilise une technique d'analyses en composantes principales (ACP) sur les seize points obtenus. Cette ACP donne trois directions privilégiées et il ne reste alors qu'a calculer les constantes de définition de façon que la boîte contienne les seize points.
- si une boîte possède une tranche marquée, on utilise une technique similaire d'ACP mais bidimensionnelle après projection orthogonale sur un plan parallèle au plan de la tranche marquée.

On peut remarquer que dans ces deux derniers cas, les vecteurs des tranches sont deux-à-deux orthogonaux (c'est une propriété de l'ACP).

4.3.4.2 Intersection

Dans ce cas, on calcule une boîte à côtés parallèles aux tranches de G qui contiennent $G \cap D$, ainsi qu'une boîte à côtés parallèles aux tranches de D. On retient la boîte de volume minimal.

4.3.4.3 Différence

C'est le cas le plus simple, la boîte associée à $G \backslash D$ est la boîte de G.

4.3.5 Utilité des englobants

Le test d'intersection entre un rayon et une boîte généralisée, s'il est rapide, prend quand même un certain temps (le même temps que pour intersecter un cube). On a donc tout intérêt à essayer de diminuer ce nombre de tests.

Ceci est par exemple possible lorsque l'on intersecte un rayon avec un nœud de type transformation affine. Si l'on intersecte la boîte transformée, il est tout à fait inutile de tester le rayon avec la boîte initiale : il n'y a en effet pas de perte de précision lors des transformations affines.

De même, lors que l'on teste un rayon avec une boîte d'un nœud de type intersection, il est inutile de tester les boîtes des objets que l'on intersecte : s'il y a intersection avec l'intersection des boîtes, il y a nécessairement intersection avec chacune des boîtes. Similairement, lorsque qu'un rayon intersecte une boîte associée à un nœud différence, il n'est pas nécessaire de tester la boîte de l'objet qui subit cette différence : c'est la même boîte. Enfin, pour les noeuds de type union, il est possible que la boîte de l'union soit l'une des deux boîtes initiales et il est encore inutile de retester cette intersection.

On utilise dans ce but la notion d'*utilité de boîte englobante*. Cette utilité est notée sur chaque nœud pour les boîtes du ou des nœuds qui sont juste sous le nœud considéré. Elle est calculée en même temps que les boîtes elles-mêmes et elle indique lesquelles parmi les tranches de la boîte considérée sont à tester.

Il faut ainsi noter que l'utilité n'est pas définie pour une boîte *a priori*, car elle dépend du chemin d'accès dans l'arbre de modélisation. Il est alors naturel de stocker cette utilité courante dans la structure d'environnement dont se sert l'intersecteur (voir le chapitre 3).

4.3.6 Deux principes d'accélération

On utilise encore les deux techniques suivantes pour essayer d'améliorer l'efficacité de l'intersecteur :

- après un test d'intersection positif entre un rayon et une boîte associée à un nœud de type intersection, on tronque l'intervalle de recherche $[\lambda_{min}, \lambda_{max}]$ aux abscisses contenues dans la boîte : ceci évite de retenir des intersections que l'on est sûr de voir éliminées lors du calcul d'intersection des listes d'intersection. Sur l'exemple de la figure 4.1, on doit intersecter l'objet $A \cap B$. La boîte englobante est représentée par la partie grisée. On ne génère alors pas le premier intervalle d'intersection avec l'objet A, car il est en dehors de la boîte.

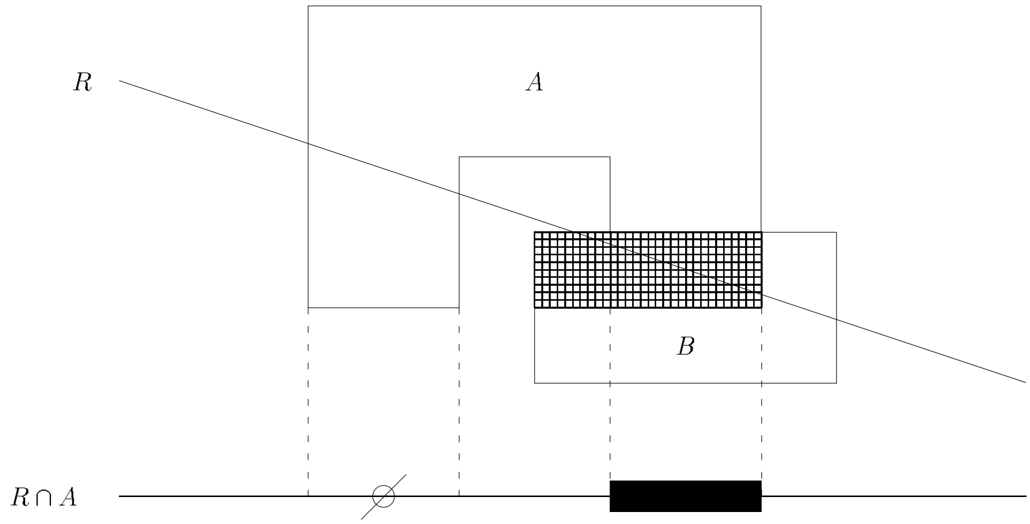


FIG. 4.1 – *Elimination des intersections inutiles*

méthode	scène	objets touchés	boîtes touchées	efficacité	temps
BEG	`un_cube`	12960	12960	1.	11s
BEG	`deux_cubes`	32	1420	0.0255	9.5s
STD	`un_cube`	12960	40656	0.319	13.85s
STD	`deux_cubes`	32	53616	0.000597	15.8s

TAB. 4.2 – *Comparaison des 2 types de boîtes englobantes*

– lorsque l'on intersecte un nœud de type union, et que l'on ne demande que la première intersection, on effectue un test a priori entre le rayon et les boîtes des deux opérandes de cette réunion : on commence alors par intersecter l'opérande dont la boîte englobante est la plus proche.

4.4 Quelques résultats

Nous avons comparé les deux méthodes (boîtes englobantes classiques et boîtes généralisées) sur deux scènes-test. La première scène (`un_cube`) contient un unique cube dont la diagonale principale est colinéaire à l'axe Ox. La seconde scène (`deux_cubes`) contient deux petits cubes de côté 0.05, l'un centré en (-5,-5,-5) et l'autre centré en (10,10,10). Les deux images ont été calculées en résolution 320x256, et seuls les rayons primaires ont été générés. Le tableau 4.2 regroupe les résultats obtenus. On constate que dans le cas de la première scène, on obtient une efficacité de 1, ce qui correspond à la non-perte de précision lors des transformations affines, alors que les boîtes classiques induisent une efficacité de 0.319 seulement. Notons que les temps ne sont pas proportionnels dans ce cas à l'efficacité, car ces temps incluent toutes les initialisations ainsi que les calculs d'intersection proprement dits, ces derniers étant communs aux deux méthodes.

Les résultats sont encore plus flagrants pour la scène `deux_cubes` puisque l'efficacité des boîtes généralisées est plus de 42 fois supérieure aux boîtes classiques. Le gain en temps est alors encore plus important.

Il reste cependant que ces améliorations sont ici idéales car les scènes testées sont choisies de façon à mettre en évidence cette efficacité. Il resterait à déterminer l'amélioration apportée sur des scènes “standard”, ce qui impliquerait aussi de définir ce qu'est une telle scène.

4.5 Améliorations possibles

Nous allons présenter dans cette dernière partie un ensemble d'améliorations que nous pourrions apporter à notre méthode à englobants pour encore améliorer leur efficacité. Il convient de noter qu'aucune de ces améliorations n'est actuellement implantée et que les idées que nous donnons ne sont donc que des intuitions. Dans certains cas, nous avons également procédé à des simulations qui permettent de donner une bonne idée des résultats à obtenir.

4.5.1 Amélioration de la précision des boîtes

4.5.1.1 Problème rencontré

Notre méthode, si elle résoud certains problèmes, n'est cependant pas optimale. Les principaux problèmes se rencontrent lors de la construction des unions de boîtes. Si les résultats obtenus sont localement bons, ils peuvent être globalement moins bons que des englobants classiques en raison de la non-associativité de la méthode de construction, comme le montre l'exemple de la figure 4.2. Comme on le voit sur la figure de gauche, les deux techniques (boîtes classiques et généralisées) donnent le même résultat lorsque l'on effectue l'union $(A \cup B) \cup (C \cup D)$. Par contre, les résultats sont différents lorsque l'on effectue l'union $(A \cup D) \cup (B \cup D)$. Plus précisément,

- la boîte généralisée de $A \cup D$ (ou celle de $B \cup C$) est plus précise que la boîte classique
- mais la boîte généralisée globale est moins précise.

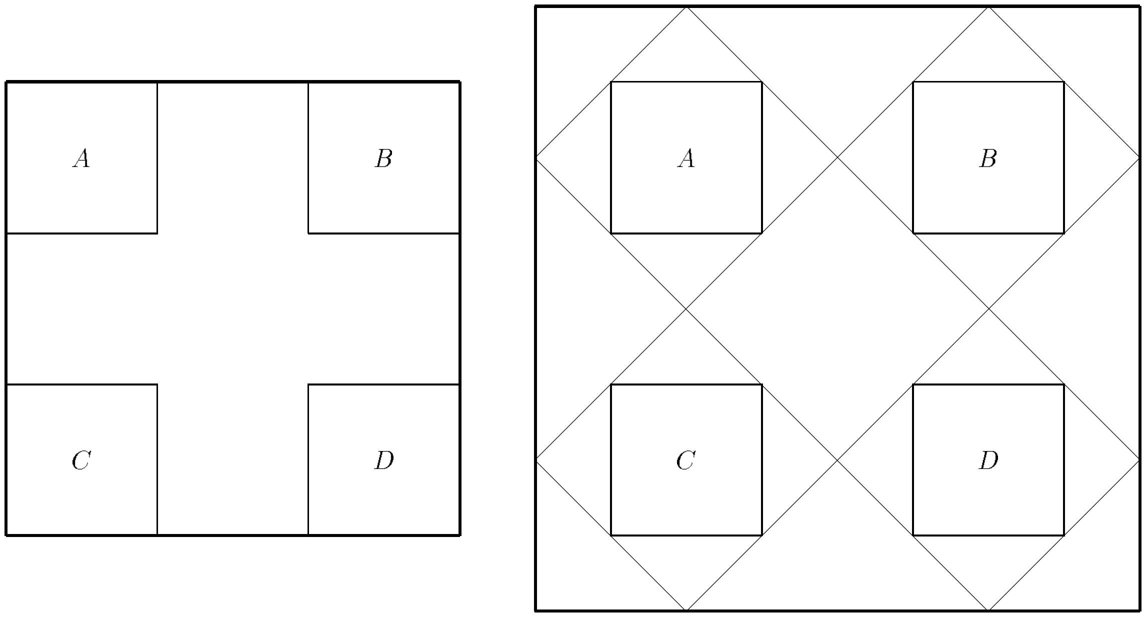
Ce phénomène est dû au fait que les points que l'on calcule sont les sommets de la boîte généralisée et ne proviennent donc pas forcément de points des boîtes initiales, ce qui est par contre le cas avec les boîtes classiques.

4.5.1.2 Algorithme optimal

Nous allons présenter ici un algorithme optimal de construction de boîtes généralisées. Il est optimal en ce sens qu'il n'est pas possible localement ou globalement de trouver des boîtes plus petites que celles générées par celui-ci.

L'algorithme repose sur le principe suivant : puisque les englobants que l'on recherche sont des convexes (car intersections de convexes), le meilleur englobant possible d'un objet est aussi celui de son enveloppe convexe. On peut donc détailler l'algorithme de construction :

- on suppose connues pour toutes les primitives leur enveloppe convexe (sous formes d'une liste de points extrémaux par exemple).
- ensuite, on calcule les enveloppes convexes de chaque nœud de l'arbre de construction. Ceci est relativement facile pour les transformations affines et les différences, un peu plus délicat pour les unions et encore plus difficile pour les intersections.
- étant donné un noeud dont on connaît les points extrémaux de l'enveloppe convexe, on doit calculer la boîte de volume minimal contenant ces points.



$(A \cup B) \cup (C \cup D)$ $\qquad$ $(A \cup D) \cup (B \cup D)$

FIG. 4.2 – *Exemple de mauvaises boîtes*

Cet algorithme est malheureusement purement théorique, car outre les difficultés rencontrées pour calculer la hiérarchie des enveloppes convexes, nous n'avons aucun algorithme permettant d'effectuer la troisième partie, c'est-à-dire la construction d'une boîte à volume minimal contenant un semis de points. Les seuls algorithmes que l'on peut utiliser sont des algorithmes approximatifs, comme une ACP qui reste possible (mais qui est sensible au nombre des points), ou bien une recherche partielle en restreignant les boîtes à avoir des côtés parallèles à des directions privilégiées.

Les recherches continuent pour améliorer encore ces algorithmes.

4.5.2 Recomposition des unions

Ainsi que ceci a été indiqué dans le chapitre 2, le langage *CASTOR* autorise les unions n-aires, et notre système nécessitant des arbres de construction binaires, nous sommes obligés de binariser ces unions. La technique actuellement utilisée consiste à décomposer une union de n objets selon les puissances de 2 inférieures à n.

On peut alors essayer d'utiliser une technique de binarisation basée sur l'efficacité des boîtes englobantes, c'est-à-dire d'essayer de réunir d'abord les objets "proches" les uns des autres. Il est bien sûr possible pour l'utilisateur de décomposer "à la main" ses unions d'objets, mais nous nous sommes intéressés à des techniques de partition automatiques.

Cette étude a été menée pendant le stage de travail de fin d'études d'Antoine Tillier [Til90]. Nous nous sommes intéressés à trois méthodes, reposant sur des principes de boîtes englobantes classiques, et donc à côtés parallèles aux axes de coordonnées. La première et la troisième de ces méthodes sont cependant extensibles à des boîtes généralisées.

4.5.2.1 Première méthode

Le principe en est très simple et consiste à subdiviser l'union initiale en deux sous-unions, le critère étant le fait que les volumes cumulés des deux boîtes ainsi générées doit être minimal.

L'algorithme utilisé pour réaliser cette partition est alors le plus brutal possible : une boîte possède 6 faces et elle est minimale si chaque face est tangente avec au moins un des objets. Il suffit donc de considérer tous les sous-ensembles d'objets de l'union initiale, et pour chaque sous-ensemble S, de calculer le volume de la boîte minimale contenant S et celle contenant son complémentaire. Si le volume cumulé des deux boîtes ainsi générées est inférieur à la solution optimale connue jusqu'alors, on remplace cette solution optimale par S.

Comme tous les algorithmes brutaux, la compléxité est exécrable car, pour un nombre n d'objets, la compléxité est au pire en n^7 (union en râteau). Les résultats obtenus sont par contre excellents puisque cette méthode est optimale pour le critère que nous avons fixé, à savoir le volume cumulé des boîtes.

4.5.2.2 Deuxième méthode

Le principe est maintenant de vouloir réunir les boîtes proches les unes des autres. Encore convient-il de définir une notion de distance entre deux boîtes. Nous proposons la définition suivante :

- une boîte (à côtés parallèles aux axes) est connue par son centre C et la longueur de ses côtés (X, Y, Z).
- la distance d entre deux boîtes $B_1 = (C_1, X_1, Y_1, Z_1)$ et $B_2 = (C_2, X_2, Y_2, Z_2)$ est calculée par

$$d = ||\vec{C_1C_2}|| + \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2 + (Z_1 - Z_2)^2}$$

On utilise alors le principe suivant pour regrouper les objets :

- on commence par calculer la boîte englobante des n objets.
- parmi les n objets, on choisit celui dont la boîte est la plus éloignée de la boîte globale.
- enfin, on groupe cet objet avec l'objet dont la boîte est la plus proche. Puis on recommence avec les $n - 1$ nouveaux objets

La deuxième phase de cet algorithme consiste en fait à essayer de regrouper d'abord les objets de petit volume et situés à la périphérie de la boîte globale. Une fois cet objet détecté, on l'unit à l'objet le plus proche.

La complexité de cet algorithme est alors bien meilleure puisqu'elle est globalement en n^2. En effet, à chaque étape, chacune des trois phases de l'algorithme défini ci-dessus est linéaire en fonction du nombre d'objets.

Cependant, en utilisant des simulations dans le plan, nous nous sommes aperçus que cette méthode pouvait donner dans certains cas des résultat exécrables. Ceci est particulièrement le cas lorsqu'un petit objet est centré dans la boîte globale. Ce petit objet est alors choisi comme premier objet à grouper, alors qu'il serait plus intéressant de privilégier la position des objets plutôt que leur taille. Nus avons alors pensé à introduire dans la métrique un coefficient multiplicateur k, la distance étant alors définie par

$$d = ||\vec{C_1C_2}|| + k\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2 + (Z_1 - Z_2)^2}$$

Affecte une valeur de k inférieure à 1 permet de privilégier l'excentricité. Mais il convient alors de choisir une "bonne" valeur pour ce coefficient, ce qui semble encore délicat.

Il nous a donc semblé peut raisonnable d'utiliser cette méthode, bien que des tests sur des cas réels n'aient pas été effectués. De plus, il semble bien plus difficile de définir une métrique sur les boîtes généralisées, ce qui rend donc quasiment impossible l'utilisation de cette méthode.

4.5.2.3 Troisième méthode

Cette troisième méthode est née de la volonté d'essayer d'obtenir des résultats comparables à ceux générés par la première méthode avec des temps similaires à ceux de la deuxième.

Nous avons donc conservé le principe de regrouper à chaque étape deux objets, mais en utilisant cette fois-ci comme critère de regroupement le critère de la première méthode, c'est-à-dire le volume de la boîte générée. On choisit donc de regrouper les deux objets dont la boîte englobante globale est de volume minimal.

La complexité de cet algorithme est alors proportionnelle à n^3. En effet, à chaque étape, le choix de deux objets à regrouper se fait en temps quadratique, ce qui donne donc cette complexité.

Les résultats simulés dans le plan semblent bien meilleurs et d'ailleurs très proche de la première méthode. De plus, il n'y a aucune difficulté à étendre cette méthode aux boîtes généralisées. C'est donc vraisemblablement dans cette direction que nous poursuivrons nos études.

4.5.3 Remplissage/occupation des boîtes

Nous avons fait la remarque que dans une hiérarchie de boîtes, certaines de ces boîtes sont "inutiles", et il est en particulier inutile de calculer l'intersection entre de telles boîtes et un rayon (cas des objets intervenant dans une intersection ou en partie retranchante d'une différence).

Il est possible de prolonger cette notion en définissant deux coefficients liés au remplissage des boîtes. Plus précisément, on définit pour chaque objet du graphe CSG :

- un coefficient de remplissage, égal au quotient de la somme des volumes des boîtes des sous-objets composant cet objet par le volume de la boîte de l'objet.
- un coefficient d'occupation, égal au quotient du volume de la boîte de l'objet par le volume de la boîte du sur-objet qui l'utilise.

De la même façon que l'utilité des boîtes, le coefficient de remplissage peut être stocké directement sur l'objet, alors que le coefficient d'occupation doit être stocké sur le sur-objet, ce qui implique que la gestion en incombe à la pile d'environnement. Remarquons que dans notre cas, le coefficient de remplissage pour les transformations affines est toujours égal à 1. Remarquons encore que ces notions sont indépendantes du type de boîtes utilisé et peuvent donc être définies pour des boîtes standard ou généralisées.

L'utilisation de ces deux coefficients est alors subordonnée à la valeur de deux seuils, appelés *seuil minimal de remplissage* et *seuil maximal d'occupation*. L'algorithme d'intersection gère alors un taux d'occupation courant, qui est stocké dans la pile d'environnement. Sa valeur est initialisée à 0. On peut alors ainsi décrire l'algorithme de gestion des ces taux :

```
si le taux d'occupation est inférieur au seuil maximal d'occupation
        si le coefficient de remplissage est supérieur au seuil minimal de remplissage
                intersecter la boîte de l'objet
                affecter 1 au taux d'occupation
        sinon
                multiplier le taux d'occupation par le coefficient d'occupation
sinon
        multiplier le taux d'occupation par le coefficient d'occupation
```

Ainsi, le taux d'occupation représente le quotient de volume de la boîte englobante d'un objet par rapport au volume de la dernière boîte testée. On choisit donc de n'intersecter que les boîtes des objets qui ont un volume suffisamment petit par rapport à la dernière boîte intersectée, et qui sont suffisamment remplies.

On gagne ainsi du temps en évitant :

- d'intersecter des boîtes quasiment vides où la probabilité d'intersecter l'un des sous-objets est faible même si l'on intersecte l'objet (cas de l'union des deux petits cubes décrit précédemment).

- d'intersecter des boîtes quasiment identiques à des boîtes déjà intersectées (la probabilité d'intersecter une sous-boîte sachant que l'on intersecte la boîte est alors très grande).

On peut même envisager que les deux seuils puissent être adaptatifs et être affinés en cours de calcul d'une image. On a alors un algorithme d'apprentissage qui permet d'optimiser la gestion des boîtes englobantes.

Ces développements nous semblent très intéressants et nous allons essayer de poursuivre leur étude. Des travaux récents [Viv93] ont d'ailleurs utilisé une approche similaire pour effectuer une décomposition adaptative d'une scène en utilisant un test d'arrêt sur le coefficient de remplissage (notons que Vivian utilise le nom de taux d'occupation pour ce qui est appelé ici coefficient de remplissage).

Chapitre 5

Le rendu

5.1 Introduction

Après avoir détaillé les principes de fonctionnement de l'intersecteur, il convient maintenant de préciser la deuxième partie importante d'un tracé de rayons, à savoir le rendu.

Rappelons que pour notre modèle, le rendu est l'utilisation qui est faite des données calculées par l'intersecteur. Ces données consistent en une liste d'intervalles d'intersection, chaque intervalle pouvant contenir une description des propriétés volumiques de l'objet traversé (ou des objets traversés dans le cas d'objets neutres) et une description des points d'entrée et de sortie (normales aux surfaces, numéros des surfaces, couleurs et textures). Cette liste, qui peut bien sûr être vide, est associée au rayon qui a permis de générer ces intersections, ainsi qu'à une structure d'information sur ce rayon.

Il nous a semblé intéressant d'essayer d'unifier les méthodes de rendu traditionnellement utilisées ne synthèse d'images, cette unification devant aussi permettre des extensions à d'autres modèles comme un calcul de masse ou de moment d'inertie. Nous sommes ainsi parvenus à la définition d'un "modèle de modèle" de rendu (les spécialistes de l'approche orientée objet parleraient de méta-modèle).

Une telle unification présente de nombreux avantages : tout d'abord, elle incite à formaliser beaucoup plus ce qui se passe dans un algorithme de rendu. D'autre part, dans la mesure ou nous voulons pouvoir utiliser successivement (ou même parfois simultanément) plusieurs modèles différents, il est nécessaire de pouvoir changer dynamiquement le rendu utilisé. Ce changement est beaucoup plus simple si les deux algorithmes suivent un même modèle. Enfin, d'un point de vue pratique, l'implantation des divers rendus est facilitée par le fait qu'un grand nombre des fonctions à écrire pour un tel rendu sont des fonctions génériques, c'est-à-dire utilisables quel que soit le rendu.

On peut noter que cette idée d'unification n'est pas à proprement parler nouvelle puisque l'on peut trouver dans la littérature un certain nombre d'exemples. Le principe utilisé dans *RenderMan* [Ups89], avec la définition des *shaders*, est très similaire à notre approche, même si les *shaders* peuvent aussi modifier la géométrie du modèle. De la même façon, Hall [Hal88] avait défini un algorithme générique de rendu, avec cependant un certain nombre de restrictions : la couleur n'est définie que par un tableau de valeurs, on ne maitrise que très peu la génération des rayons secondaires. Nous avons simplement essayé d'enrichir au maximum les possibilités de l'algorithme générique de rendu.

Précisons maintenant les concepts essentiels de ce modèle. Nous allons d'abord introduire les données manipulées par le rendu, et nous détaillerons ensuite les diverses fonctions que l'on utilise pour effectuer le rendu. Enfin, nous présenterons l'algorithme général de rendu.

A titre d'exemple, nous présenterons diverses implantations "classiques" en synthèse d'images. Ces implantations concernent les modèles de Lambert, Phong et Whitted.

Dans la dernière partie de ce chapitre, nous allons également présenter une implantation possible de l'utilisation d'un tracé de rayons pour répartir l'énergie dans une scène. Ceci peut être très utile dans une phase de prétraitement avant calcul d'image afin de répartir correctement l'énergie lumineuse dans la scène. Cette notion a été introduite sous le nom de *tracé de rayons inverse* [Arv86]. Nous pourrons d'ailleurs voir que notre algorithme générique ne doit être que fort peu modifié afin de remplir ce rôle.

5.2 Les données du rendu

5.2.1 Energie

Cette notion est à la base de tout le processus de rendu. Par définition, l'énergie est la grandeur calculée par le processus de rendu, c'est-à-dire une certaine interprétation des caractéristiques des objets.

Dans le cadre d'un rendu "classique" (Phong ou Whitted), l'énergie est le plus souvent représentée par des valeurs de rouge, vert et bleu.

Notons toutefois que l'unité de ces grandeurs est la plupart du temps non précisée. Dans la plupart des cas, on entend par valeur de rouge une quantite comprise entre 0. et 1. ou bien entre 0. et 255., cette quantité étant alors interprété comme une valeur sur l'échelle de rouge à l'affichage d'un moniteur vidéo. Ceci pose alors de nombreux problèmes lors de l'algorithme de rendu. Comment faire par exemple lorsqu'en additionnant des énergies, les valeurs dépassent les limites permises ? La plupart du temps, on se contente de tronquer ces valeurs hors-limites, ce qui peut causer des erreurs importantes dans le résultat final.

Il est alors plus correct d'employer des grandeurs physiques, comme par exemple l'énergie lumineuse ou la luminance. Il est toutefois impossible de calculer le spectre complet de l'énergie, qui est une fonction, mais on peut très bien représenter ce spectre par un ensemble de valeurs de ce spectre pour des longueurs d'ondes régulièrement échantillonnées ou bien pour des longueurs d'ondes choisies de façon optimale [MG87].

Si l'on s'intéresse à un calcul de masse, alors l'énergie est précisement la masse. Dans le cas d'un calcul de moments d'inertie, l'énergie est le moment d'inertie exprimé en $kg.m^2$.

5.2.2 Sources

Dans le cadre de notre modèle, une *source* est entendue au sens *source d'énergie*, c'est-à-dire un élément de l'environnement capable de générer de l'énergie.

Dans le cadre d'un rendu classique, les sources sont les sources lumineuses que l'on trouve dans tout modeleur. Dans le cadre de calcul de masse ou de moments d'inertie, les sources sont inexistantes car rien ne correspond à la définition que nous avons retenue.

Il convient de noter que les sources ne sont pas des objets du modèle géométrique. En particulier, elles ne peuvent pas générer d'intersection avec un rayon.

5.2.3 Couleur

La couleur est le terme générique qui regroupe toutes les propriétés des objets de modélisation pour propager l'énergie ainsi que pour en émettre. Il faut ainsi distinguer les objets qui émettent de la lumière et les sources lumineuses : les sources ne sont pas des objets de modélisation, et un rayon ne peut jamais intersecter une source. On verra que l'algorithme de rendu traite différemment ces deux types d'émetteurs d'énergie.

Outre l'émission d'énergie, la couleur comprend aussi les propriétés de transmission et de réflexion de l'énergie.

Il faut ici noter que ces propriétés peuvent avoir un sens pour l'intérieur des objets ou uniquement pour leur surface : la couleur a donc un sens en tant qu'attribut surfacique et en tant qu'attribut volumique (voir le chapitre 2). Pour chacune des propriétés, nous détaillerons donc le sens surfacique et le sens volumique de cette propriété.

5.2.3.1 Réflectance

Par définition, la réflectance précise comment l'objet reflète l'énergie qui lui parvient de son entourage (aussi bien des sources que des autres objets du modèle).

Notons que cette réflexion intervient toujours à la surface des objets. Ainsi, la réflectance définie pour l'intérieur d'un objet est la réflectance à prendre en compte si cet objet n'a pas de réflectance définie pour une de ses surfaces frontière. Si l'objet a une réflectance définie pour sa surface, on utilise donc cette dernière.

Dans notre modèle, la réflectance est simplement décrite par une structure de données, cette structure étant spécifique du modèle de rendu utilisé.

5.2.3.2 Transmittance

Ce terme désigne les propriétés de transmission de l'énergie. On peut rappeler que l'on suppose que le phénomène de transmission n'intervient que sur la valeur de l'énergie et non sur sa direction. Il convient de noter également que la transmittance possède des sens différents selon qu'elle est un attribut surfacique ou volumique.

En tant qu'attribut surfacique, la transmittance permet de représenter les effets générés par des films minces et transparents appliqués sur les objets. Prenons l'exemple d'un objet en bois sur lequel on a appliqué une couche de vernis. Les propriétés de réflexion de la lumière de l'objet se trouvent modifiées, et le bois reste toujours apparent sous le vernis. Il est peu raisonnable de vouloir représenter par un objet de modélisation géométrique la couche de vernis : vu la faible épaisseur d'une telle couche, les problèmes d'imprécision risqueraient de générer un aliassage important (obtention de textures non voulues). Il est en revanche bien plus réalisable de représenter cette couche de vernis par un attribut surfacique, avec les propriétés de réflectance et de transmittance voulues.

En tant qu'attribut volumique, la transmittance permet de représenter tous les effets d'atténuation de l'énergie lors de son parcours à travers les objets.

On peut noter ici que la notion de transmittance volumique est assez indépendante de la notion d'objet neutre. En effet, un objet neutre peut fort bien ne pas avoir de transmittance volumique définie. Dans ce cas, on considère donc que la transmittance est "totale", soit encore que l'énergie ne subit aucune modification lors de son parcours à travers l'objet neutre. A l'opposé, un objet actif peut lui aussi avoir une transmittance volumique. Prenons l'exemple d'une sphère en verre coloré : cet objet n'est pas un objet neutre (négliger les effets de réfraction sur une sphère conduit à des écarts importants par rapport à la réalité) et lors de la réfraction, l'énergie est modifiée par le spectre d'absorption du verre.

Comme pour la réflectance, la transmittance est stockée dans une structure de donnée spécifique au modèle de rendu.

Il convient ici de faire une remarque importante : la transmittance n'est pas forcément indépendante de la réflectance. En effet, dans un modèle "physique", ces deux notions peuvent être trouvées dans un ensemble de données communes, à savoir l'indice de réfraction et le coefficient d'extinction. Ces deux valeurs permettent alors de calculer aussi bien les coefficients de Fresnel pour la réflexion et la réfraction de la lumière que l'atténuation de la lumière lors de son parcours à travers la matière.

Il nous a cependant semblé plus pratique d'utiliser deux structures de données distinctes formellement, même si dans certains cas ces deux structures sont en fait identiques. Dans le cas des modèles plus classiques (Lambert et Phong par exemple), cette distinction est même totale puisque les structures sont elles-même différentes.

5.2.3.3 Energie propre

Les notions de réflectance et de transmittance que nous venons de détailler sont classiques dans les modèles de rendu. La notion d'*énergie propre* l'est peut-être moins.

Comme son nom l'indique, l'énergie propre est l'énergie émise par les objets eux-mêmes. Un objet possédant une énergie propre émet donc de l'énergie même si son environnement ne contient aucune source d'énergie. On peut en donner une illustration si l'on considère un objet recouvert d'une peinture phosporescente : dans le noir le plus complet (absence de sources), une certaine énergie est émise par la peinture et l'objet est ainsi "visible".

L'énergie propre peut intervenir sous deux formes : l'énergie propre surfacique (la peinture phosporescente en est un exemple) et l'énergie propre volumique (celle-ci intervient par exemple dans les primitives de lumière et les densités volumiques).

Nous utilisons deux champs dans la couleur pour stocker une énergie propre :

- un indicateur du type d'énergie propre,
- une structure de données contenant elle-même deux champs : une éventuelle fonction d'énergie propre et une structure de données auxiliaire.

Notre système autorise trois types différents d'énergie propre que nous allons détailler.

- la constante d'énergie propre : comme son nom l'indique, l'énergie propre est supposée être constante dans tout le volume ou sur toute la surface considérée. Dans ce cas, la structure de données ne comporte pas de fonction d'énergie propre et la structure auxiliaire contient simplement la valeur de la constante, cette constante étant bien sûr de type énergie.
- la fonction d'énergie propre : dans ce cas, l'énergie propre peut dépendre d'un certain nombre de paramètres comme la normale à la surface et la direction d'émission pour une énergie propre surfacique, ou encore les données géométriques comme le point d'entrée et le point de sortie pour une émission volumique, ou le point de calcul pour une émission surfacique.
- la constante temporaire d'énergie propre : cette notion, très proche de la constante d'énergie propre, a été introduite afin que l'algorithme de rendu puisse "construire" si besoin est, une énergie propre (il est en effet très difficile de construire une fonction). L'utilité de cette notion sera détaillée dans le paragraphe sur les textures et le mixage des objets neutres.

Il existe une distinction supplémentaire entre la constante et la constante temporaire : une constante d'énergie propre est un attribut de modélisation et peut donc être à ce titre partagé entre plusieurs objets, alors que la constante temporaire n'est qu'un artifice algorithmique et n'est donc valable que pour le rendu en cours. Il faut alors tenir compte de cette distinction pour la gestion dynamique de la mémoire : une constante d'énergie est construite lors de la modélisation et son existence durera jusqu'à la fin de l'algorithme de rendu, alors que la constante temporaire n'est utilisée que pour une phase de l'algorithme de rendu, c'est-à-dire pour un intervalle d'intersection.

5.2.4 Textures

Comme nous l'avons rapidement expliqué au chapitre 2, une texture est par définition une propriété de l'objet qui dépend de la position du point où l'on s'intéresse à cette propriété. Nous insistons encore ici sur le fait que cette position est définie dans un repère local (en fait, le repère du monde de l'objet au moment où la texture lui est appliquée).

Afin d'effectuer correctement les texturations, il est nécessaire de pouvoir connaître les points de texturation dans le repère local, ceci est la raison pour laquelle on stocke des couples (matrice,textures) dans les données générées par l'intersecteur.

De plus, les textures peuvent exister pour la surface des objets aussi bien que pour leur intérieur. On trouvera donc des textures volumiques et des textures surfaciques.

Dans le cadre de notre modèle, nous avons retenu les textures qui créent ou modifient les paramètres suivants :

- la normale à la surface d'un objet
- la couleur d'un objet

Les textures qui modifient la normale à la surface sont évidemment de type surfacique, les textures de couleur pouvant être quant à elles de type surfacique et volumique. Ces dernières peuvent d'ailleurs modifier tout ou seulement une partie de la couleur (réflectance, transmittance ou énergie propre). Les textures d'énergie propre sont toutefois peu utilisées puisque l'énergie propre autorise déjà l'utilisation d'une fonction d'énergie propre qui peut rendre les mêmes servies qu'une texture.

Enfin, il convient de faire ici une dernière remarque : nous avons dit que les textures ne dépendaient que de la position du point où elles étaient calculées. Notre modèle autorise cependant des dépendances additionnelles : les textures volumiques peuvent dépendre du numéro de l'objet, les textures surfaciques peuvent dépendre du numéro de l'objet et du numéro de surface de l'objet. Rappelons ici que le numéro de l'objet est unique pour chaque instance d'un objet (un objet utilisé deux fois possède deux numéros différents), et que le numéro de surface n'est pas un numéro de facette plane. Une des utilisations de cette possibilité est l'exemple d'une texture qui définit une couleur différente pour chacune des faces d'un objet (on peut penser à une texture “dé”).

D'un point de vue stockage, nous utilisons pour les textures une structure contenant les champs suivants :

- le type de la texture,
- une fonction de texturation,
- une structure auxiliaire pour stocker les paramètres de définition de la texture. Ces paramètres peuvent être de nombre et de types variables et dépendent de la fonction de texturation.

Les textures peuvent être stockées sur un objet sous forme de liste chaînée, les textures en tête de liste devant être appliquées les premières.

5.2.5 Fond

On peut se poser la question suivante : quelle est l'énergie apportée par un rayon qui n'intersecte aucun des objets de la scène ?

On pourrait y apporter une réponse très simple : ce rayon apporte une énergie nulle. Ceci nous a cependant semblé insuffisant et nous avons introduit la notion de *fond*.

Par définition, l'*énergie de fond* est l'énergie apportée par un rayon primaire ou secondaire qui n'intersecte aucun des objets de la scène. Il faut ainsi noter que le traitement des rayons d'ombre est particulier : en effet, un rayon d'ombre qui n'intersecte aucun objet apporte comme contribution l'énergie provenant de la source. Il nous a donc semblé plus pratique de faire cette distinction entre les divers types de rayon.

Le fond est stocké dans les données globales de la scène, sous la forme d'une structure de donnée contenant trois champs :

- un indicateur du type de fond,
- une fonction de calcul de l'énergie de fond,

- une structure de données auxiliaire utilisable pour stocker toute information utile pour la fonction de calcul. La taille et le type de cette structure dépend du type de fond utilisé.

Parmi les indicateurs de type de fond figure obligatoirement la valeur `PAS_DE_FOND`, pour laquelle la fonction de calcul et la structure sont absentes, et qui signifie simplement qu'une énergie nulle doit être attribuée comme énergie de fond.

5.3 L'arbre des rayons

5.3.1 Principe général

De façon classique en tracé de rayons, le processus de rendu utilise un ensemble de rayons que l'on peut structurer sous forme d'arbre. Dans un but de simplicité, nous conservons pour notre modèle les dénominations usuelles pour les différents type de rayons.

Ainsi, les rayons qui constituent les racines des arbres de rayons sont appelés *rayons primaires*. Les rayons que l'on peut générer pour déterminer l'énergie en provenance des sources sont appelés *rayons d'ombre*. Enfin, les rayons n'entrant dans aucune des deux catégories précédentes sont appelés *rayons secondaires*. On ne distingue donc pas à ce niveau les rayons réfléchis et les rayons réfractés : cette distinction est en effet dûe au modèle utilisé pour le rendu.

Il convient cependant de noter que dans notre modèle, les rayons d'ombre sont des feuilles de l'arbre des rayons : aucun autre rayon n'est généré à partir d'un rayon d'ombre.

5.3.2 Informations liées à un rayon pour le rendu

Un certain nombre d'informations doivent être associées à tout rayon traité par l'algorithme de rendu. Certaines de ces informations sont valables pour tout modèle de rendu, d'autres sont spécifiques à un modèle particulier. Nous allons donc maintenant détailler les informations communes, les informations spécifiques seront détaillées avec chaque modèle.

5.3.2.1 Type du rayon

Le type de chaque rayon doit être connu par l'algorithme de rendu. En effet, dans le cas des rayons d'ombre par exemple, le traitement à faire s'il n'y a pas d'intersections détectée n'est pas le même que pour les rayons primaires ou secondaires.

De la même façon, une fois le rendu effectué, l'utilisation faite du résultat est différente selon le type : dans le cas d'un rayon d'ombre ou d'un rayon secondaire, le résultat doit être interprété dans l'arbre des rayons correspondant et le traitement peut nécessiter l'utilisation d'autres rayons, alors que dans le cas d'un rayon primaire, le rendu est pratiquement terminé et il ne reste que l'interprétation finale de ce résultat (par exemple le calcul de la couleur d'un pixel dans le cas d'un calcul d'image). On verra que cette distinction est très importante dans le cas de l'algorithme parallèle.

5.3.2.2 Rayon primaire correspondant

Il est important pour tout rayon de savoir à quel arbre des rayons appartient celui-ci. Dans le cas d'un algorithme parallèle par exemple, plusieurs rayons primaires peuvent être traité simultanément : il faut donc savoir à quel rayon primaire correspond tel rayon secondaire.

Cette information est donc stockée sous forme d'un numéro, étant entendu que ce numéro doit pouvoir identifier de façon unique un rayon primaire. Les deux fonctions de correspondance entre le numéro et le rayon primaire sont donc spécifiques à chaque modèle de rendu.

5.3.2.3 Profondeur du rayon

On sait que dans de nombreux rendus, le véritable arbre des rayons à générer est infini. Les ordinateurs ayant beaucoup de difficultés à traiter l'infini, on doit choisir des techniques de troncature de cet arbre des rayons.

L'une des techniques utilisées pour cette troncature est très simple et est basée sur la notion de *profondeur dans l'arbre des rayons*. Un rayon primaire est de profondeur 0 et tout rayon secondaire ou d'ombre est de profondeur égale à la profondeur du rayon qui l'a généré augmentée de 1. La troncature consiste alors à ne pas générer les rayons de profondeur strictement supérieure à un seuil donné, ce seuil étant un entier positif (éventuellement nul) stocké dans les données globales de la scène.

5.3.2.4 Contribution d'un rayon

Cette notion est également utilisée pour tronquer l'arbre des rayons. Par définition, la contribution d'un rayon est une borne supérieure du quotient $\frac{\text{énergie du rayon}}{\text{énergie du rayon primaire associé}}$.

Pour tronquer l'arbre, on décide de ne pas générer les rayons dont la contribution tombe en-dessous d'un certain seuil, ce seuil étant comme le seuil de profondeur des rayons, stocké dans les données globales de la scène.

5.3.2.5 Informations complémentaires

Afin de pouvoir stocker d'autres informations, on prévoit dans la structure de données un pointeur sur une structure de données adéquate.

Cette possibilité est utilisée dans le cas des rayons d'ombre : on y stocke un pointeur sur la source correspondant à ce rayon. Il est en effet nécessaire de pouvoir retrouver cette source comme ceci sera détaillé dans l'algorithme de rendu.

5.4 Le processus générique de rendu

Nous allons tout d'abord présenter toutes les fonctions utilisées par le processus de rendu. Ces fonctions doivent être définies pour chaque modèle. Le processus de rendu est quant à lui, générique.

On pourra noter cependant que certaines des fonctions peuvent aussi être génériques et ainsi être réutilisées d'un modèle à l'autre.

Enfin, faisons deux remarques importantes pour toutes les fonctions que nous allons décrire :

- toutes les fonctions (exceptée la fonction auxiliaire de rendu) ne “rendent” aucun résultat. Si l'on veut récupérer les données calculées par une fonction, on doit lui passer en argument de quoi stocker ce résultat.
- toutes les fonctions concernant une surface d'objet ont comme arguments obligatoires le numéro d'objet (unique) et le numéro de surface de l'objet, toutes les fonctions concernant l'intérieur d'un objet ont comme argument obligatoire le numéro d'objet. Comme ces arguments sont obligatoires, ils seront systématiquement oubliés dans les descriptions des arguments des fonctions.

5.4.1 Fonction auxiliaire de rendu

Dans le souci de généricité que nous avons expliqué, nous avons regroupé tout un ensemble de fonctionnalités à trouver dans un processus de rendu sous le nom de *fonction auxiliaire de rendu*.

Cette fonction est bien sûr dépendante du modèle utilisé et remplit les fonctionnalités suivantes :

- recopie d'une énergie

- recopie d'une transmittance
- recopie d'une réflectance
- multiplication d'une énergie par une constante
- addition de deux énergies
- initialisation d'une énergie
- conversion d'une énergie en couleur de pixel
- combinaison linéaire de deux énergies
- combinaison linéaire de deux réflectances
- combinaison linéaire de deux transmittances

D'un point de vue implantation, cette fonction auxiliaire est une seule fonction. Son premier argument est un code permettant d'identifier la fonctionnalité désirée. Les arguments suivants sont en nombre et en types variables et dépendent de la fonctionnalité. Le résultat de cette fonction est selon les cas une énergie, une réflectance ou une transmittance.

On peut d'ailleurs noter que cette fonction peut aussi être utilisée pour réaliser l'allocation dynamique des structures de données puisque les tailles de ces structures dépendent du modèle de rendu choisi. Afin de réaliser cette fonctionnalité, nous avons choisi la technique suivante :

- le pointeur sur la structure de données destinataire est passé en argument de la fonction auxiliaire;
- si ce pointeur est non nul, il est utilisé tel quel, sinon la fonction auxiliaire alloue la structure de données correspondante;
- dans tous les cas, le pointeur destinataire est rendu comme résultat de la fonction. La fonction appelante a alors la possibilité de récuperer ce pointeur.

5.4.2 Energie en provenance d'une source

A divers moments dans le processus de rendu vont intervenir les sources lumineuses. Détaillons maintenant comment nous calculons l'énergie en provenance d'une source.

La première remarque que l'on peut faire est le fait que le calcul de l'énergie en provenance d'une source suppose connue la direction de provenance de cette énergie. En effet, comme on le verra plus loin, c'est une autre partie de l'algorithme de rendu appelée *générateur de rayons d'ombre* qui détermine cette direction. Ceci permet de ne pas restreindre les sources aux sources ponctuelles ou à direction unique, et on peut donc utiliser des sources surfaciques.

La seconde remarque est le fait qu'il convient de distinguer tout de suite deux fonctions de calcul de l'énergie en provenance d'une source, que nous nommons *énergie directe* et *énergie simple* en provenance de la source. Précisons tout de suite ces deux notions.

- l'énergie directe est l'énergie qui parviendrait au point de calcul de l'éclairement en provenance de la source considérée si aucun objet n'était présent dans la scène. Il n'y a alors pas lieu de tenir compte des ombres portées ou propres, ni de l'influence d'éventuels objets transparents entre la source et le point.
- l'énergie simple doit en revanche tenir compte de l'influence éventuelle des autres objets de la scène. On doit en particulier tenir compte des éventuelles ombres portées, ainsi que des objets transparents placés entre le point considéré et la source.

Précisons maintenant les modalités de calcul de ces deux types d'énergie.

5.4.2.1 Energie directe

La fonction de calcul de l'énergie directe est très simple: elle admet comme arguments

- le point où l'on calcule cette énergie,
- la source lumineuse,
- la direction de provenance de l'énergie,

et elle fournit comme résultats

- l'énergie en provenance de cette source dans cette direction.

5.4.2.2 Energie simple

Cette fonction admet pratiquement la même interface que la fonction d'énergie directe (mêmes arguments et mêmes résultats), avec une petite différence : elle admet un argument supplémentaire qui est la structure d'information liée au rayon qui a généré ce calcul.

C'est en effet à l'intérieur de cette fonction de calcul d'énergie simple que l'on peut décider ou non de relancer un rayon pour calculer l'éclairement. Il est alors nécessaire de connaître le rayon primaire associé. Notons d'ailleurs que les rayons d'ombre ont la particularité de ne pas générer d'autres rayons : la profondeur ainsi que la contribution du rayon sont ainsi inutiles dans ce cas.

Notons enfin que cette fonction d'énergie simple peut aussi utiliser des valeurs d'éclairement précalculées par un tracé de rayon arrière, utiliser des valeurs de radiosité, ou tout autre technique.

5.4.3 Générateur de directions d'ombre

Ainsi que nous venons de le préciser, les fonctions de calcul d'énergie en provenance d'une source supposent connues la source et la direction de provenance. Le *générateur de directions d'ombre* s'occupe, comme son nom l'indique, de générer toutes les directions incidentes d'énergie en provenance des sources. Le nombre de directions d'ombre à générer étant variable, nous avons choisi le principe de fonctionnement suivant :

- le premier appel au générateur est un appel d'initialisation. En particulier, lors de cet appel, le générateur construit une structure de données connue de lui seul, qu'il utilise pour savoir quel est la prochaine direction à générer. Un pointeur banalisé sur cette structure de données est rendu comme résultat de la fonction.
- tous les appels suivants génèrent une nouvelle direction. Le pointeur sur la structure de données est passé comme argument à chaque fois.
- lorsque le générateur constate qu'il n'y a plus de direction à générer, il libère la structure de données et indique à la fonction appelante que la génération de directions d'ombre est terminée.

Les arguments de la fonction de génération sont alors les suivants :

- la position du point considéré,
- la normale à la surface,
- le pointeur sur la structure de données auxiliaire,

et le résultat est constitué de

- le pointeur sur la source,

– la direction d'incidence,

. Le choix de cette architecture pour la génération des directions d'ombre permet une très grande souplesse. En particulier, il est très facile d'ajouter des effets de pénombre en modifiant simplement ce générateur : il suffit de générer plusieurs directions différentes pour une même source. Le nombre de directions peut même dépendre de la position du point par rapport à la source : en effet, pour des points éloignés de la source, le nombre de directions peut être inférieur au nombre de directions générées pour des points proches.

Le fait de choisir l'utilisation d'une structure passée en argument au lieu d'une structure statique définie dans le corps de la fonction de génération permet une extension très simple lors d'une parallélisation de cet algorithme (voir le chapitre 6).

5.4.4 Gestion du fond

Rappelons que le fond est la possibilité d'attribuer une énergie à un rayon n'intersectant aucun objet de la scène, ce rayon devant être de type primaire ou secondaire.

Cette énergie se calcule en connaissant

– le point de calcul,

– la direction du rayon,

– la structure auxiliaire pour stocker les données du fond.

Actuellement, notre système prévoit deux types de fond : le fond uniforme et le ciel. Ces deux types de fond sont des types génériques et peuvent être utilisés pour n'importe quel modèle de rendu : en effet, les fonctions de calculs des énergies de fond correspondantes utilisent simplement les fonctionnalités de la fonction auxiliaire de rendu.

5.4.4.1 Fond uniforme

C'est l'expression la plus simple : pour tout point et pour toute direction, l'énergie de fond est constante. Cette constante est simplement stockée dans la structure auxiliaire, et il suffit de recopier cette valeur d'énergie pour avoir l'énergie de fond.

5.4.4.2 Ciel

Comme son nom l'indique, un fond de type *ciel* permet de donner une couleur de fond assez semblable à un ciel. Plus précisément, un ciel est défini par deux énergies appelées *énergie haute* (E_h) et *énergie basse* (E_b) et par quatre constantes appelées *sinus haut* ($\sin_h$), *sinus bas* ($\sin_b$), *gradient haut* (grad_h) et *gradient bas* (grad_b).

Si (x, y, z) est la direction du rayon, on calcule s le sinus de l'angle entre cette direction et la verticale, soit

$$s = \frac{z}{\sqrt{x^2 + y^2 + z^2}}$$

Puis on compare cette valeur de s avec les valeurs de sinus haut et bas. Plus précisément,

– si $s \leq \sin_b$, alors la valeur de l'énergie de fond est la valeur de l'énergie basse.

– si $s \geq \sin_h$, alors la valeur de l'énergie de fond est la valeur de l'énergie haute.

– si $\sin_b < s < \sin_h$, alors on calcule

$$\begin{aligned} k &= \frac{s - \sin_b}{\sin_h - \sin_b} \\ &\text{puis} \\ k_h &= k^{\text{grad}_h} \\ k_b &= (1-k)^{\text{grad}_b} \end{aligned}$$

L'énergie du fond est alors une combinaison linéaire de l'énergie haute affectée du coefficient k_h et de l'énergie basse affectée du coefficient k_b (la combinaison linéaire de deux énergies est l'une des fonctionnalités de la fonction auxiliaire).

Cette fonction de ciel est totalement empirique. Elle permet cependant d'obtenir des dégradés assez comparables aux couleurs naturelles. Elle n'est pas parfaite et il serait tout à fait envisageable d'introduire les fonctions de ciel utilisées dans les modèles d'architecture ou du batiment.

5.4.5 Gestion des textures

Rappelons qu'il existe des textures volumiques et des textures surfaciques. Elles peuvent modifier la réflectance, la transmittance (aussi bien surfacique que volumique) et également la normale (les textures de modification de normale sont uniquement surfaciques). Il n'existe pas de textures de l'énergie propre tout simplement parce que celles-ci sont inutiles : en effet, l'énergie propre a déjà la possibilité d'être définie par une fonction, et peut donc dépendre de la position des points où elle est calculée. Cependant, dans un but d'uniformisation, il serait possible de définir une énergie propre par un couple constante/textures.

On peut toutefois noter qu'une texture peut s'appliquer globalement à une couleur : en effet, lorsque l'on veut décrire un objet qui est un mélange de deux matériaux, on conçoit que l'on modifie en même temps la réflectance et la transmittance de l'objet. Il faut alors prendre garde au problème évoqué ci-dessus, à savoir le problème de la texturation des énergies propres.

Ces textures sont toutes à appliquer dans un repère local qui est le repère de leur instant de définition. Rappelons qu'à cet effet, les textures sont stockées dans les intervalles ou dans les points d'intersection sous forme d'un couple contenant la matrice de passage dans le repère local et un pointeur sur une liste de textures à appliquer dans ce repère.

Il existe donc deux fonctions de texturation différentes, l'une qui effectue les texturations pour les intervalles, l'autre pour les points.

5.4.5.1 Recopie des couleurs

Un point doit être précisé dès maintenant. Dans le chapitre 2, nous avons précisé qu'une couleur pouvait être partagée par plusieurs objets. Il faut donc prendre garde lorsque l'on effectue une texturation à ne pas modifier de structure de données partagée.

On utilise à cet effet la technique suivante. Tout d'abord, chaque couleur est "marquée" en fonction de sa classe d'allocation. Il existe ainsi trois types possibles :

– `STATIQUE` ce qui correspond à une couleur partagée par plusieurs objets. Toute couleur définie lors de la modélisation de la scène est systématiquement marquée `STATIQUE`. Les données que contient une telle structure ne peuvent être changées.

– `DYNAMIQUE` ce qui indique que la couleur est une copie. On peut ainsi modifier les données qu'elle contient sans perturber le reste du modèle.

- `PILE` ce qui est quasiment équivalent à `DYNAMIQUE` sauf que la structure a été allouée dans la pile d'éxécution : l'allocation et la libération sont alors plus rapides que pour une structure `DYNAMIQUE`.

Ensuite, avant d'effectuer une texturation qui concerne la couleur, on effectue une copie de la couleur à texturer si elle n'est pas de type `DYNAMIQUE` ou `PILE`.

Enfin, une fois l'algorithme de rendu terminé, on parcourt la liste d'intervalles d'intersection et on libère toutes les couleurs de type `DYNAMIQUE`. Les couleurs de type `PILE` sont libérées lors du retour de la fonction, et les couleurs de type `STATIQUE` sont conservées pour les rendus suivants.

5.4.5.2 Texturation des contenus

Les textures de contenu peuvent concerner la réflectance, la transmittance ou bien globalement la couleur. Toutes les textures possèdent des paramètres de définition et une fonction de texturation. Ces paramètres peuvent être des coefficients d'échelle, diverses constantes ou bien des couleurs (réflectances ou transmittances) de base à partie desquelles on calcule la couleur texturée. Cette couleur peut aussi dépendre d'une couleur préexistante.

Pour effectuer la texturation d'un contenu, pour chaque bloc-texture, on suit l'algorithme suivant :

- en utilisant la matrice de passage dans le repère local, on calcule les points d'entrée et de sortie dans le repère local.

- on effectue une copie de la couleur de base si elle existe, et si elle est de type `STATIQUE`.

- pour chaque texture de ce bloc-texture, on appelle la fonction de texturation.

La fonction de texturation admet quant à elle les arguments suivants :

- le point de sortie dans le repère local,

- le point d'entrée dans le repère local,

- la donnée initiale (couleur,réflectance ou transmittance),

- les paramètres de texturation.

5.4.5.3 Texturation des points

L'algorithme est presque identique au précédent sauf que les textures des cellules-point peuvent aussi concerner la normale. L'algorithme est donc, pour chaque bloc-texture, le suivant :

- en utilisant la matrice de passage dans le repère local, on calcule le point d'entrée (ou de sortie) dans le repère local.

- on effectue une copie de la couleur de base si elle existe et si elle est de type `STATIQUE`.

- pour chaque texture de ce bloc-texture,

  - si cette texture est une texture de normale, si la normale n'a pas déjà été transformée, on transforme la normale dans le repère local.
  - on appelle la fonction de texturation.

- si la normale a été transformée, on la transforme dans le repère du monde.

Il faut se rappeler que la matrice stockée dans le bloc-texture est la matrice de passage du repère du monde au repère local. C'est donc la transposée de la matrice inverse qu'il faut appliquer à la normale dans le repère du monde pour la transformer dans le repère local. Pour revenir dans le repère du monde, on applique simplement la transposée de la matrice du bloc-texture.

C'est parce que ces opérations peuvent être coûteuses en temps de calcul (en particulier l'inversion de matrice), que l'on effectue les tests pour savoir si une texture de normale a été appliquée.

Dans un but de simplification, les fonctions de texturations, qu'elles agissent sur la couleur ou sur les normales, admettent les mêmes arguments :

- le point de texturation dans le repère local.
- la normale à la surface dans le repère local.
- la donnée initiale (couleur, réflectance ou transmittance).
- les paramètres de texturation.

Il faut noter que dans le cas des textures de normale, la donnée passée en troisième argument est un pointeur nul.

On peut constater que, comme les points et les normales sont stockées dans des structures de données identiques, les prototypes des fonctions de texturation sont les mêmes pour les textures volumiques et pour les textures surfaciques (aussi bien les textures de normale que les textures de couleur). Ceci simplifie également l'écriture de ces fonctions de texturation.

Enfin, le modèle que nous avons choisi n'interdit pas d'un point de vue formel les textures de couleur dépendant de la normale à la surface de l'objet : tous les arguments nécessaires sont passés à la fonction de texturation. Bien qu'aucne texture de ce type ne soit implantée actuellement, ceci reste possible à l'avenir. Il faudrait alors modifier légérement l'algorithme de texturation décrit ci-dessus afin que la normale soit toujours convertie dans le repère local même si aucune texture de normale n'est appliquée : sans cette précaution, les normales seraient toujours calculées dans le repère du monde.

5.4.6 Gestion d'une superposition d'objets

Nous avons indiqué dans le chapitre 2 que les objets neutres possédaient la propriété de se superposer. Nous allons maintenant préciser comment ces superpositions sont gérées. On peut rappeler ici que les superpositions d'objets ne concernent que le contenu des objets et non leur surface.

Le principe général est le calcul d'une *couleur équivalente* pour cette superposition d'objet. On parcourt donc la liste des contenus et pour chaque contenu, on effectue l'algorithme suivant :

- si le contenu ne comprend ni couleur ni textures, il est simplement ignoré.
- s'il contient des textures, on effectue une texturation du contenu comme décrit ci-dessus.
- si ce n'est pas le premier contenu, on effectue le mélange de cette couleur avec la couleur équivalente des objets précédents. Sinon, on copie la couleur de ce contenu comme couleur équivalente.

Plusieurs remarques peuvent être faites sur cet algorithme. Tout d'abord, le principe de mélange doit être “associatif”, c'est-à-dire ne pas dépendre de l'ordre de mélange. Ensuite, on utilise la possibilité de décrire l'énergie propre par une constante : on est en effet incapable de construire une fonction d'énergie propre dans ce cas (sauf dans certains langages ou l'on peut dynamiquement construire du code). La constante ainsi fabriquée est ainsi une constante temporaire qu'il ne faut pas oublier de libérer une fois le processus de rendu terminé. C'est ici que l'on utilise la possibilité de décrire une énergie propre de type `constante temporaire`.

La fonction de mélange est bien sûr dépendante du modèle de rendu utilisé. L'une des solutions les plus simples consiste simplement à faire des combinaisons linéaires de couleurs au moyen des fonctionnalités correspondantes de la fonction auxiliaire. On verra des exemples dans les descriptions de modèles classiques.

5.4.7 Réflexion de l'énergie

Précisons les modalités de calcul de l'énergie réfléchie par une surface. Notre modèle utilise une fonction de réflexion, qui accepte comme arguments

- l'énergie incidente,
- la direction d'incidence de l'énergie,
- la normale à la surface,
- la réflectance de la surface,
- la direction dans laquelle on demande l'énergie réfléchie,
- l'énergie réfléchie.

Il faut noter que la réflectance peut être la réflectance surfacique ou bien la réflectance volumique sous-jacente si aucune réflectance surfacique n'a été définie pour cette surface.

Cette fonction de réflexion peut bien sûr être utilisée pour les rayons d'ombre et les rayons secondaires, que ceux-ci soient des rayons réfléchis dans une direction privilégiée ou dans une direction quelconque.

Notons enfin que la notion d'énergie réfléchie suppose que la direction incidente et la direction réfléchie sont du même côté par rapport à la normale à la surface de l'objet. Si ces deux directions sont du côté de la normale, on parlera de réflexion *externe*, alors que si elles sont du côté opposé, on parlera plutôt de réflexion *interne*.

5.4.8 Réfraction de l'énergie

A l'inverse de l'énergie réfléchie, l'énergie réfractée se caractérise par le fait que les directions incidente et réfractée sont opposées par rapport à la normale à la surface. On trouve alors la notion de réfraction classique dans le modèle de Whitted.

La réfraction de l'énergie est déterminée par une fonction de réfraction, qui admet comme arguments :

- l'énergie incidente,
- la direction d'incidence de l'énergie,
- la transmittance volumique du côté incident,
- la normale à la surface,
- la transmittance volumique du côté réfracté,
- la direction dans laquelle on demande l'énergie réfractée,
- l'énergie réfractée.

Notons que nous ce sont les transmittances volumiques qui contiennent les données nécessaires à ce calcul.

5.4.9 Transmission de l'énergie

De la même façon, on utilise deux fonctions pour calculer les énergies transmises : une pour la transmission surfacique et une pour la transmission volumique. Détaillons ces deux fonctions.

5.4.9.1 Transmission surfacique

La fonction de transmission surfacique d'énergie admet comme arguments

- l'énergie incidente,
- la direction de propagation,
- la normale à la surface,
- la transmittance surfacique
- l'énergie transmise.

On peut noter que la direction de propagation détermine en même temps la direction incidente et la direction de transmission puisque l'on suppose qu'il n'y a pas déviation de l'énergie lors d'une transmission.

5.4.9.2 Transmission volumique

De son côté, la fonction de transmission volumique de l'énergie admet comme arguments

- l'énergie initiale,
- le point d'entrée de l'énergie,
- le point de sortie de l'énergie,
- la transmittance volumique,
- l'énergie transmise.

On peut noter que la transmission ne dépend pas des normales aux points d'entrée et de sortie. Encore une fois, nous rappelons que l'énergie transmise n'est pas déviée : la direction de propagation est la droite joignant le point d'entrée au point de sortie.

5.4.10 Energie totale

Par définition, nous appelons *énergie totale de surface* ou plus simplement *énergie totale* le cumul de toutes les énergies qu'une surface émet dans une direction déterminée, énergies ayant leur origine dans l'environnement de l'objet auquel appartient cette surface, et ayant une direction de provenance autre que la direction d'émission.

On exclut ainsi de l'énergie totale l'énergie propre de la surface ainsi que l'énergie transmise à travers la surface. Dans la plupart des modèles, l'énergie ainsi prise en compte est l'énergie réfléchie sur la surface ainsi que l'énergie réfractée par l'objet auquel cette surface appartient. On peut d'ailleurs noter que l'énergie réfléchie peut prendre en compte l'énergie ambiante, l'énergie en provenance des sources, l'énergie en provenance d'autres objets,....

Cette fonction d'énergie totale est donc l'une des fonctions-clés du modèle de rendu. C'est dans cette fonction que l'on va trouver la génération de tous les rayons secondaires pour un modèle de Whitted, ou bien l'échantillonnage autour de la direction de réflexion ou de réfraction pour un tracé de rayons distribué. La fonction d'énergie totale utilise dans la plupart des cas plusieurs des fonctions que nous avons déjà décrites, comme les fonctions d'énergie en provenance des sources, les fonctions de réflexion de l'énergie.

La fonction d'énergie totale utilise de plus trois fonctions de génération de rayons : la fonction de génération de directions d'ombre déjà détaillée, ainsi qu'une fonction de génération de rayons réfléchis et une fonction de génération de rayons réfractés.

Le fonctionnement des fonctions de génération des rayons réfléchis ou réfractés est très similaire à celui du générateur de directions d'ombre. On utilise ainsi un premier appel d'initialisation, puis tous les appels suivants génèrent les rayons correspondants. Il existe cependant une différence essentielle : ces deux générateurs doivent connaître les caractéristiques de rendu des objets, alors que celà n'était pas le cas pour le générateur de directions d'ombre. Plus précisément,

- le générateur de rayons réfléchis admet comme arguments supplémentaires la réflectance de la surface,
- le générateur de rayons réfractés admet comme arguments supplémentaires les transmittances volumiques des objets situés de part et d'autre de la surface.

Rien n'interdit alors de pouvoir gérer au sein des ces générateurs des effets de réflexion (ou de réfraction) distribuée, c'est-à-dire un échantillonnage de rayons réfléchis autour d'une direction privilégiée.

5.4.11 Energies propres

Précisons maintenant le mode de calcul des énergies propres surfacique et volumique.

Si l'énergie propre est de type `constante` (ou `constante temporaire`), il suffit simplement de recopier l'énergie pour avoir l'énergie propre : cette fonction de recopie est une des fonctionnalités de la fonction auxiliaire.

Sinon, on appelle la fonction de calcul de l'énergie propre, mais les arguments de cette fonction dépendent de la nature de l'énergie :

- pour l'énergie propre surfacique, les arguments sont
  - le point de calcul de l'énergie propre,
  - la normale à la surface,
  - la direction d'émission,
  - la structure de donnée auxiliaire pour l'énergie propre,
  - l'énergie émise.
- en ce qui concerne l'énergie propre volumique, les arguments sont
  - le point initial d'émission,
  - le point final d'émission,
  - la structure de données auxiliaire pour l'énergie propre,
  - l'énergie émise.

  Il convient d'ailleurs de noter que l'énergie propre volumique ne dépend pas des normales aux surfaces limites, comme c'est aussi le cas pour la transmittance volumique.

5.4.12 Profondeur de rendu

Nous avons défini pour le rendu d'un intervalle d'intersection la notion de *profondeur de rendu*. Celle-ci permet de définir les différents éléments qui vont intervenir dans le rendu. La profondeur peut ainsi prendre quatre valeurs dont nous allons tout de suite préciser le sens. Ces valeurs sont données par ordre croissant.

- `SURFACE_IN` : ceci signifie que l'intervalle d'intersection possède un point d'entrée, que ce point d'entrée possède une couleur, que cette couleur contient une réflectance et pas de transmittance. L'énergie arrivant selon la direction du rayon ne dépend que des caractéristiques de ce point d'entrée : énergie propre éventuelle, réflexion d'énergies diverses sur cette surface.

- `VOLUME_IN` : ou bien l'intervalle d'intersection n'a pas de point d'entrée, ou bien ce point d'entrée n'a pas de couleur (auquel cas il est interprété comme de transmittance totale et de réflectance nulle), ou bien cette couleur contient une transmittance. L'intervalle doit être de type intersection avec un objet actif, doit posséder un contenu, et ce contenu ne doit pas contenir de transmittance. L'énergie calculée dépend alors de l'énergie propre volumique, des réflexions d'énergie sur la surface de ce volume, et des modifications apportées par l'éventuel point d'entrée : transmitance, énergie propre surfacique et réflexions d'énergie sur la surface.

- `SURFACE_OUT` : les contraintes sur le point d'entrée sont identiques à celles du paragraphe précédent. L'intervalle doit être de type intersection avec un neutre, ou bien de type intersection avec un actif et posséder une transmittance volumique. De plus, le point de sortie doit exister, avec une couleur, une réflectance et pas de transmittance. Les effets supplémentaires sont l'énergie propre surfacique de la surface de sortie,les réflexions sur cette surface, et la transmission de l'énergie résultante à travers le volume.

- `DERRIERE` : par rapport au paragraphe précédent, le point de sortie doit être absent, ou bien contenir une transmittance. Le calcul de l'énergie doit alors de plus prendre en compte l'énergie arrivant au point de sortie dans la direction du rayon : il faut alors appeler le processus de rendu sur l'éventuel intervalle d'intersection suivant.

Ainsi, selon les valeurs croissantes de la profondeur de rendu, de plus nombreuses énergies sont à prendre en compte.

Il faut d'autre part noter que le fait d'avoir ou non une réflectance, d'avoir ou non une transmittance ne peut s'apprécier que lorsque les texturations ont été effectuées. Afin d'éviter de calculer des texturations inutiles pour le rendu, on entrelace le processus de calcul des textures et le calcul de la profondeur de rendu. Dans le cas d'une intersection avec un ou plusieurs objets neutres, le calcul de la couleur équivalente est éffectué en même temps.

5.4.13 Algorithme générique de rendu

Les notions qui permettent le rendu ayant été précisées, on peut maintenant préciser comment se déroule un algorithme de rendu. Encore une fois, nous essayons ici de préciser un algorithme générique qui peut être utilisé pour tout modèle de rendu.

Précisons tout d'abord les données utilisées par le processus de rendu. On trouve dans ces données :

- une liste (éventuellement vide) d'intervalles d'intersection. Chaque intervalle de cette liste peut contenir un point d'entrée, un point de sortie et un contenu. Chaque intervalle contient également le type de l'objet (ou des objets) intersecté. Cette liste peut être une liste d'intersections générée par l'intersecteur, ou bien seulement une partie de cette liste : le processus de rendu est récursif est peut être appelé sur la liste privée des n premiers intervalles.

- le rayon que l'on a intersecté avec la scène. Celui-ci est nécessaire pour pouvoir calculer la position des points dans le repère du monde car ils ne sont connus que par leur abscisse sur le rayon.
- la structure d'information associée à ce rayon. On y trouve le type du rayon, sa profondeur, son indice de contribution et la donnée complémentaire.
- l'abscisse courante sur le rayon. Cette abscisse permet d'effectuer un rendu correct dans le cas où il n'y a pas d'intersection, ou lorsque l'on n'a trouvé que des intersections avec des neutres : il faut alors déterminer l'énergie arrivant au point de sortie du dernier objet neutre. Le processus de rendu doit alors calculer la position du point considéré.

Le processus de rendu calcule l'énergie arrivant à l'origine du rayon et dans la direction du rayon. Dans le cas classique d'un calcul d'image, cette énergie est ensuite convertie en une couleur de pixel, soit en général des valeurs de rouge, vert et bleu entières et comprises entre 0 et 255. Dans le cas d'un calcul de masse ou de moment d'inertie, on ajoutera la contribution dûe à ce rayon aux valeurs déjà calculées.

L'algorithme de calcul de l'énergie peut alors se décomposer comme suit :

- s'il n'y a plus d'intervalle d'intersection, deux cas peuvent se présenter. Soit le rayon est de type `OMBRE` auquel cas on calcule l'énergie directe en provenance de la source correspondante (la source se trouve dans la donnée complémentaire de la structure d'information associée à ce rayon), soit le rayon est de type `PRIMAIRE` ou `SECONDAIRE` auquel cas on appelle la fonction de calcul de l'énergie de fond. Dans les deux cas, la position se calcule à l'aide de l'origine et de la direction du rayon ainsi que de l'abscisse courante sur le rayon.
- si la profondeur est `DERRIERE`,
  - on commence par calculer l'énergie arrivant au point de sortie de l'intervalle selon la direction du rayon : ceci se réalise en appelant de façon récursive le processus de rendu sur l'intervalle d'intersection suivant, avec comme abscisse courante l'abscisse du point de sortie, en conservant le rayon et la structure d'information associée.
  - s'il y a un point de sortie, si ce point de sortie a une transmittance, on calcule l'énergie transmise à travers la surface.

  sinon

  - on initialise l'énergie à zéro (ceci peut être effectué par la fonction auxiliaire de rendu).
- si la profondeur est supérieure ou égale à `SURFACE_OUT`
  - s'il y a un point de sortie, si ce point de sortie a une couleur, on ajoute à l'énergie courante l'énergie propre éventuelle de la surface de sortie, ainsi que son énergie totale. Il faut prendre garde au fait que c'est une énergie totale ąlculer "par l'intérieur" (penser à une sphère de verre métallisée sur une demi-surface), et il convient donc d'inverser la normale à la surface avant de calculer cette énergie totale. Notons que cette énergie totale peut prendre en compte l'énergie réfractée à la sortie de l'intervalle. On peut trouver la transmittance volumique de l'objet situé derrière le point de sortie dans l'intervalle lui-même.
  - on calcule l'énergie transmise à travers le volume de l'objet intersecté.
  - on ajoute à l'énergie courante l'énergie propre volumique éventuelle.
- si la profondeur est supérieure ou égale à `VOLUME_IN`
  - s'il y a un point d'entrée, on ajoute à l'énergie courante l'énergie totale ce cette surface d'entrée. Il faut noter que la couleur à utiliser pour cette surface d'entrée contient la réflectance

volumique et non la réflectance de la surface. La normale à cette surface peut se trouver quant à elle dans la cellule-point d'entrée. Notons encore une fois que l'énergie réfractée intervient dans ce calcul. Dans ce cas, vu le principe retenu pour notre algorithme, on est certain que le milieu que l'on trouve avant ce point d'entrée est le milieu “ambiant”. La transmittance volumique de l'objet intersecté se trouve bien sûr dans le contenu de l'intervalle d'intersection.

  - s'il y a un point d'entrée, si le point d'entrée a une transmittance, on calcule l'énergie transmise à travers la surface.

- si la profondeur est supérieure ou égale à `SURFACE_IN`[1]

  - on ajoute à l'énergie courante l'énergie propre de la surface d'entrée ainsi que son énergie totale.

On peut faire plusieurs remarques sur l'algorithme que nous venons de décrire :

- cet algorithme est assez logique et facile à implanter : on part de la contribution d'énergie “la plus lointaine” et on “remonte” vers le point d'entrée de l'intervalle d'intersection.

- la récursion dans le processus de rendu peut intervenir à plusieurs endroits : dans le calcul de l'énergie `DERRIERE`, dans les calculs d'énergie totale. Il ne faut pas oublier lors de ces processus de tenir compte des modifications apportées à la structure d'information associée au rayon : profondeur, type et surtout indice de contribution. Cet indice doit aussi être mis à jour lors de l'appel sur l'intervalle suivant en tenant compte de toutes les transmittances : volumique, surfaciques d'entrée et de sortie.

5.5 Modèles usuels

Nous allons maintenant décrire les modèles usuellement utilisés en synthèse d'images. Il convient toutefois de noter que ces modèles ne traitent en général que la réflexion de la lumière : il est ainsi plus rare de parler de transparence, et encore plus rare de considérer des mélanges d'objets. Nous allons donc décrire ces modèles en précisant les extensions que nous avons apportées.

5.5.1 Modèle de Lambert

La première tentative pour rendre le rendu “réaliste” a été apportée par Gouraud. L'algorithme qu'il utilisait était un algorithme de lissage pour des facettes planes dont les sommets possédaient une normale. La couleur est calculée en chaque sommet de la facette puis interpolée pour les autres points. C'est pour différencier le modèle de réflexion utilisé de l'algorithme utilisé que nous employons le terme de *modèle de Lambert*.

Détaillons maintenant les divers éléments de ce modèle.

5.5.1.1 Energie

L'énergie est constituée de trois valeurs représentant les longueurs d'onde des couleurs primaires, à savoir le rouge, le vert et le bleu. Ces valeurs sont des réels compris entre 0 et 1, le triplet $(0,0,0)$ correspondant à l'énergie minimale (le noir) et le triplet $(1,1,1)$ à l'énergie maximale (le blanc).

1. ce qui est toujours le cas...

5.5.1.2 Réflectance

La réflectance est elle aussi définie par trois valeurs de rouge, vert, bleu. Selon les diverses versions de ce modèle, les valeurs sont des réels comprise entre 0 et 1, ou bien des entiers compris entre 0 et 255. Notre version utilise des valeurs réelles entre 0 et 1.

5.5.1.3 Transmittance

La transmittance volumique ainsi que la transmittance surfacique sont simplement définies par une constante de transparence comprise entre 0 et 1. Cette valeur représente le pourcentage de l'énergie qui subsiste après la transmission.

On peut noter que la transmittance volumique ne dépend pas de l'épaisseur de matière traversée : ce modèle n'est donc pas très physique mais simplifie les calculs et améliore donc le temps de calcul.

5.5.1.4 Sources

Les sources d'énergie utilisées dans ce modèle sont caractérisées par une énergie d'émission, ce qui permet d'introduire des sources colorées en utilisant des valeurs différentes pour le rouge, le vert et le bleu. Cette énergie peut subir une atténuation en fonction de la position du point où l'on calcule l'éclairement. La fonction d'atténuation dépend du type de la source lumineuse.

De façon classique également, nous utilisons trois types de sources lumineuses : les soleils, les sources ponctuelles isotropes et les sources ponctuelles directionnelles. Précisons les diverses propriétés de ces divers types de source.

- un soleil est caractérisé pour une direction d'émission. L'intensité ne subit aucune atténuation en fonction de la distance.
- une source ponctuelle isotrope est caractérisée par une position dans la scène et par trois constantes :
  - la distance de début d'atténuation (d_0),
  - la distance de fin d'atténuation (d_1),
  - le gradient d'atténuation en distance (g_d).

  Le coefficient d'atténuation k se calcule en fonction de la distance d du point d'éclairement à la position de la source de la façon suivante :
  - si $d \leq d_0$, alors $k = 1$ (pas d'atténuation).
  - si $d \geq d_1$, alors $k = 0$ (atténuation totale).
  - sinon ($d_0 < d < d_1$), $k = \left(\frac{d_1 - d}{d_1 - d_0}\right)^{g_d}$.
- une source ponctuelle directionnelle possède outre les valeurs caractéristiques d'une source isotrope
  - une direction privilégiée $\vec{D}$,
  - un cosinus minimal c_{min},
  - un gradient d'atténuation angulaire g_a.

  Le coefficient d'atténuation k se calcule alors comme produit d'un coefficent d'atténuation en fonction de la distance k_d et d'un coefficient d'atténuation angulaire k_a. Le coefficient k_d se calcule de la même façon que pour une source ponctuelle. Le coefficient k_a se calcule en fonction du cosinus c de l'angle formé par la direction $\vec{D}$ et le vecteur joignant la position de la source au point d'éclairement à de la façon suivante :
  - si $c \leq c_{min}$, alors $k_a = 0$ (atténuation totale).

- sinon ($c_{min} < c \leq 1$), $k = \left(\frac{c-c_{min}}{1-c_{min}}\right)^{g_a}$.

Cette liste de sources n'est pas limitative: elle représente les sources actuellement utilisées dans notre modèle de Lambert. De façon générale, pour calculer l'énergie en provenance d'une source, on donne comme arguments à la fonction de calcul la position du point et la source considérée : la fonction fournit en résultat l'énergie en provenance de la source ainsi que la direction d'incidence de cette énergie.

5.5.1.5 Energie simple

Notons que les formules données ci-dessus permettent en principe le calcul de l'énergie directe en provenance de la source. Dans le cas du modèle de Lambert, l'énergie simple est identique à l'énergie directe car on choisit de ne pas tenir compte des ombres.

5.5.1.6 Réflexion de l'énergie

Ce point est le point crucial du modèle. Les surfaces des objets dans un modèle de Lambert sont supposées être rugueuses (comme un crépi sur un mur par exemple) et la loi de réflexion de l'énergie est la suivante :

- chaque valeur de l'énergie (rouge, vert et bleu) et modifiée indépendamment des deux autres.
- chaque valeur de l'énergie incidente est multipliée par le coefficient de réflectance de la surface, puis par un coefficient dépendant de la direction incidente.
- ce coefficient est égal au cosinus de l'angle entre la direction incidente et la normale à la surface si ce cosinus est positif, et nul s'il est négatif.

On peut noter en particulier que ce calcul ne dépend pas de la direction de réflexion de l'énergie.

Remarquons également que ce modèle tient compte des ombres propres, c'est-à-dire des objets qui tournent le dos à la lumière, ou encore donc la direction d'incidence de l'énergie en provenance de la source est du côté opposé à la normale à la surface de calcul. On affecte alors une énergie réfléchie nulle dans ce cas. Ceci présente toutefois l'inconvénient d'une perte totale de visibilité pour les faces orientées du côté opposé aux sources de lumière. Ajouter un coefficient ambiant rend les faces visibles mais ne donne aucun relief.

Nous avons donc choisi de "corriger" le modèle de réflexion précisé ci-dessus. Au lieu de multiplier l'énergie incidente par la partie positive du cosinus c de l'angle entre la normale et la direction incidente, on la multiplie par un coefficient k calculé en fonction de c à l'aide de deux paramètres positifs ou nuls, notés p_- et p_+. Ces paramètres sont stockés dans les données globales du modèle de rendu.

Le mode de calcul de ce coefficient k en fonction de c est le suivant :

- si $c \leq 0$ (ombre propre), alors $k = p_-(1 + c)$.
- si $c > 0$, alors $k = p_+ + (1 - p_+)c$.

On peut tout de suite faire quelques remarques sur ce modèle modifié :

- lorsque c vaut 1 (incidence normale), le coefficient k vaut 1 et il n'y a donc pas atténuation.
- lorsque c vaut -1 (incidence antinormale), le coefficient k vaut 0 et il y a donc atténuation totale.
- le coefficient k tend vers p_+ lorsque c tend vers 0 (incidence rasante) par valeurs positives, et vers p_- lorsque c tend vers 0 par valeurs négatives. Ainsi, p_- et p_+ représentent les coefficients d'atténuation de l'énergie incidente de part et d'autre de la limite de l'ombre propre. Il semble donc logique d'avoir

$p_- \leq p_+$ bien que ceci ne soit pas imposé dans le modèle. On peut remarquer aussi que des valeurs distinctes de p_- et p_+ introduisent une discontinuité de la fonction de réflexion d'énergie : ceci peut être intéressant pour mettre en relief les limites d'ombre propre.

– lorsque $p_- = p_+ = 0$, on retrouve le modèle de Lambert classique. Notre modèle en est donc une simple extension.

En illustration de ce modèle, nous présentons sur la figure 5.1 trois photos de la même scène constituée d'une seule sphère éclairée par une source de type soleil. Les coefficients utilisés pour (p_-,p_+) sont de gauche à droite (0,0) (soit aucun traitement), (0.5,0.5) et enfin (0.25,0.5) pour illustrer la discontinuité. Afin de voir les faces à l'ombre, on utilise un coefficient ambiant (voir plus loin le paragraphe sur l'énergie totale) égal à 0.2.

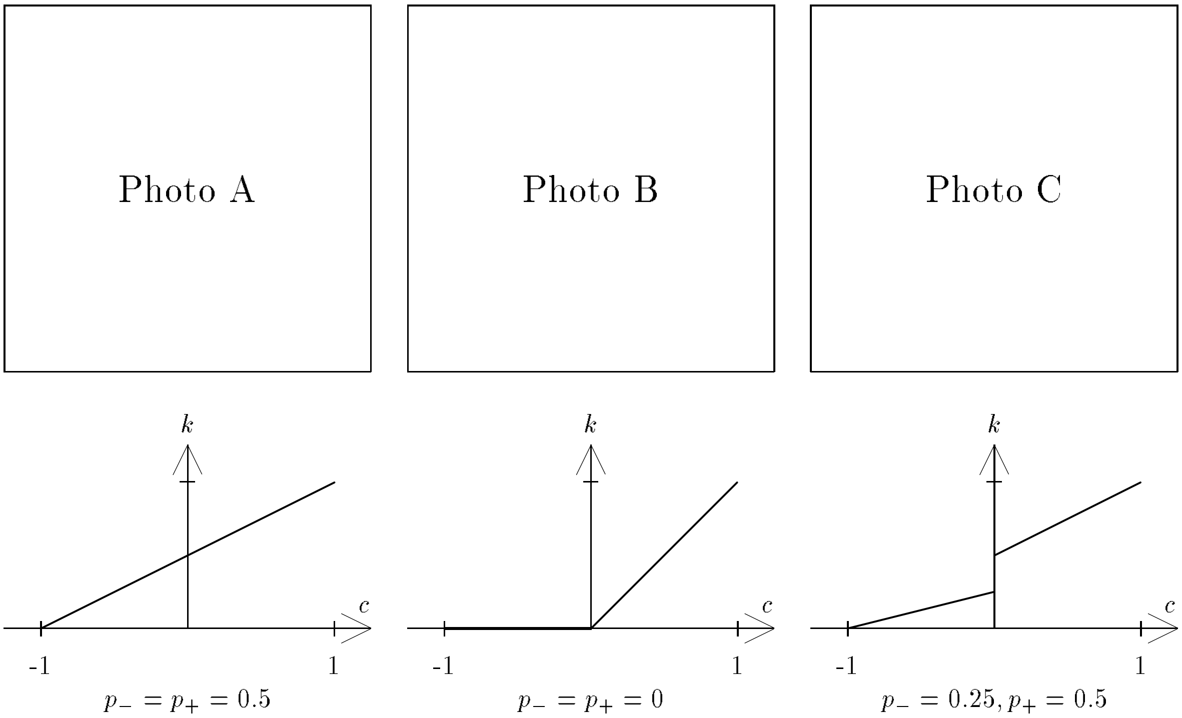


FIG. 5.1 – *Variations de l'éclairement selon p_- et p_+*

5.5.1.7 Energie totale

Nous pouvons maintenant préciser quelles sont les énergies dont va tenir compte notre modèle. Ce modèle est très simple et tient compte de deux types d'énergies :

– l'énergie provenant des sources de lumière et réfléchie par la surface. Celle-ci se calcule en utilisant l'énergie simple (ou directe) provenant des sources, et en lui appliquant la loi de réflexion précisée ci-dessus. Le générateur de directions d'ombre génère simplement une direction par source, cette direction étant la direction joignant le point de calcul à la position de la source dans le cas des sources ponctuelles (directionnelles ou non), ou la direction privilégiée dans le cas des sources directionnelles (soleil).

- une énergie "ambiante", qui représente les interréflexions de l'énergie entre les surfaces des objets. De façon très classique, cette énergie ambiante se calcule en fonction d'un paramètre appelé *coefficient ambiant*, qui est une valeur réelle comprise entre 0 et 1. L'énergie ambiante se calcule alors en multipliant simplement ce coefficient ambiant par les coefficients de réflectance de la surface.

  En particulier, l'énergie ambiante ne dépend ni de la normale à la surface, ni de la direction de calcul. Ceci est logique puisque l'on n'a aucune indication sur la provenance de ladite énergie : la seule hypothèse que l'on puisse faire est donc l'isotropie.

  Enfin, le fait d'utiliser un coefficient ambiant unique indique que l'énergie d'interréflexion est toujours "blanche", c'est-à-dire de valeurs identiques pour le rouge, le vert et bleu. On pourrait très bien modifier ceci en définissant une énergie ambiante par une véritable énergie, pouvant donc avoir des valeurs différentes pour le rouge, le vert et le bleu. L'énergie ambiante se calculerait alors en multipliant chacune des composantes par le coefficient de réflectance correspondant.

5.5.1.8 Transmission de l'énergie

La gestion de la transmission est extrêmement simple dans ce modèle puisqu'elle consiste simplement à multiplier l'énergie incidente par le coefficient de transmission, aussi bien pour la transmission surfacique que volumique.

5.5.1.9 Energies propres

Dans ce modèle, nous n'avons pas encore implanté d'exemples d'énergies propres, aussi bien volumique que surfacique. Il serait très simple de remédier à ce manque, mais ceci ne nous a pas semblé utile pour ce modèle de Lambert, lui-même très simple.

5.5.1.10 Superposition d'objets

Puisque nous utilisons des objets neutres dans le modèle de Lambert, nous pouvons aussi utiliser la superposition de tels objets. Précisons maintenant comme cette superposition est établie.

Soient $(O_1, \ldots, O_n)$ n objets à superposer, de coefficients de transparence (volumique) respectifs $(k_1, \ldots, k_n)$. Alors, si l'on note

$$\begin{aligned} z_i &= \frac{1}{k_i} - 1 \\ Z &= \sum_{i=1}^{n} z_i \end{aligned}$$

la couleur équivalente est calculée comme suit :

- la réflectance et l'énergie propre de la couleur équivalente sont calculées comme combinaisons linéaires des propriétés initiales affectées des coefficients $\frac{z_i}{Z}$.
- le coefficient de transparence équivalent est calculé par la formule $k_{equiv} = \frac{1}{1+Z}$.

On obtient par ces formules les propriétés suivantes :

- un objet totalement transparent ($k_i = 1, z_i = 0$) n'intervient pas dans un mélange.
- si l'un des objets du mélange tend vers l'opacité (k_i tendant vers $0, z_i$ tendant vers $+\infty$), la couleur équivalente du mélange est la couleur de cet objet opaque.

- le coefficient de transparence d'un mélange est toujours inférieur à tous les coefficients de transparence du mélange.

5.5.2 Modèle de Phong

Ce modèle (empirique) a été introduit par Phong pour permettre la visualisation de reflets spéculaires sur une surface. Nous avons donc implanté un tel modèle.

Ce modèle est très proche du modèle de Lambert, et nous allons donc simplement présenter les différences avec celui-ci.

5.5.2.1 Réflectance

Outre les trois coefficients de réflectance utilisés dans le modèle de Lambert, on utilise trois coefficients supplémentaires, appelés *coefficient de réflexion diffuse* (k_d), *coefficient de réflexion spéculaire* (k_s) et *indice de réflexion spéculaire* (n_s). On verra la signification et l'utilisation de ces coefficients dans le paragraphe suivant.

5.5.2.2 Réflexion de l'énergie sur une surface

Toute surface est en fait considérée comme un mélange entre deux surfaces : l'une parfaitement rugueuse possède comme loi de réflexion la loi de Lambert précisée ci-dessus, l'autre étant une surface spéculaire dont la loi de réflexion est précisée ci-dessous :

- comme pour la surface rugueuse, l'énergie incidente est multipliée par les coefficients de réflectance de la surface.
- on multiplie ensuite cette énergie par un coefficient k dépendant de la normale à la surface, de la direction incidente mais également de la direction de calcul. Plus précisément, si l'on note c_{med} le cosinus de l'angle entre la normale à la surface et la bissectrice de la direction incidente et de la direction de calcul, le coefficient k vaut $c_{med}^{n_s}$.

Ceci signifie que le coefficient d'atténuation vaut 1 si la direction incidente est le symétrique par rapport à la normale à la surface de la direction de calcul (on appelle cette direction la *direction conjuguée*). L'indice spéculaire a une influence sur le gradient de décroissance du coefficient d'atténuation lorsque l'on s'éloigne de la direction conjuguée.

En fait, nous avons modifié ce modèle de Phong de la même façon que nous avions modifié le modèle de Lambert. Le coefficient d'atténuation pour la parite rugueuse de la surface est calculé avec les mêmes corrections. Quant à la partie spéculaire, on affecte simplement un coefficient d'atténuation nul lorsque la direction incidente est du côté opposé à la normale.

Pour calculer lénergie réfléchie par une surface, il suffit de calculer les deux énergies réfléchies (sous formes diffuse et spéculaire) et d'effectuer une simple combinaison linéaire dont les coefficients sont les coefficients de réflexion diffuse et de réflexion spéculaire. On peut donc remarquer que le modèle de Lambert est un simple sous-modèle du modèle de Phong, puisqu'il suffit d'affecter à toutes les surfaces un coefficient k_d égal à 1 et un coefficient k_s égal à 0 pour retrouver le modèle de Lambert. Dans notre implantation, nous avons donc choisi d'implanter le modèle de Lambert comme sous-modèle du modèle de Phong : les valeurs par défaut de k_d, k_s et n_s dépendent du modèle choisi, et dans la fonction de calcul de réflexion de l'énergie, un drapeau indique si l'on est en sous-modèle Lambert, ce qui évite alors le calcul (relativement couteux) de la bissectrice et de son cosinus.

Enfin, une dernière remarque concernant les coefficients k_d et k_s : bien que ceci ne soit absolument pas obligatoire, on peut estimer préférable d'avoir $k_d + k_s = 1$ afin d'assurer une certaine "conservation de l'énergie".

5.5.3 Modèle de Whitted

Ce modèle, introduit par Whitted en 1980, a certainement fait une grande part de la réputation du tracé de rayons : il est donc naturel que nous l'ayons implanté. De nombreuses notions sont similaires à celles du modèle de Phong et nous ne détaillerons donc que les différences.

5.5.3.1 Réflectance

Outre les deux coefficients de réflexion diffus et spéculaire, la réflectance contient également un troisième coefficient appelé *coefficient de réfraction* et noté k_t. Celui-ci permet de calculer la contribution de l'énergie réfractée dans l'énergie totale (voir plus loin le paragraphe sur l'énergie totale).

5.5.3.2 Transmittance volumique

Si la transmittance surfacique est identique, la transmittance volumique est ici définie par deux coefficients : un *indice de réfraction* (noté n) et un *coefficient d'extinction* (noté k). L'indice de réfraction permet de calculer la déviation angulaire subie par la lumière à la séparation entre deux milieux, le second permet de calculer l'atténuation subie par celle-ci lors de son parcours à travers un milieu supposé homogène.

5.5.3.3 Energie totale

Outre les notions du modèle de Phong que l'on conserve (énergie provenant des sources, énergie ambiante, réflexion diffuse et spéculaire), on trouve dans l'énergie totale deux termes supplémentaires : l'énergie de réflexion spéculaire et l'énergie de réfraction.

L'énergie de réflexion spéculaire est par définition l'énergie arrivant au point de calcul en provenance de la direction conjuguée de la direction de calcul. Il faut remarquer que la loi de réflexion est identique à celle du modèle de Phong et tient compte des coefficients diffus et spéculaire de la surface. Comme la direction de provenance est la direction conjuguée, le cosinus entre la normale à la surface et la bissectrice des directions incidentes et de calcul vaut 1 : la réflexion spéculaire est alors maximale. Le générateur de rayons réfléchis génère donc un unique rayon.

L'énergie réfractée est l'énergie provenant du côté opposé à la direction de calcul par rapport à la surface de séparation entre deux milieux. Dans le modèle de Whitted proprement dit, la direction incidente est unique et est déterminée par la loi de Descartes[2]. Rappelons rapidement cette loi : soit S la surface de séparation entre deux milieux noté 1 et 2, d'indice de réfraction respectifs n_1 et n_2, soit $P \in S$ le point de calcul de la réfraction, soit $\vec{n}$ la normale à S en P ($\vec{n}$ est orientée de 2 vers 1), soit I la direction d'une énergie arrivant en P par le milieu 2 et i_2 l'angle entre la direction I et l'antinormale, alors l'énergie lumineuse subit une déviation au passage de S, la direction R dans laquelle est réfractée l'énergie est caractérisée par un angle i_1 avec la normale vérifiant la relation $n_1 \sin i_1 = n_2 \sin i_2$.

Pour calculer cette énergie réfractée, il est nécessaire de relancer un rayon dans la direction de réfraction et d'appeler le processus de rendu sur ce rayon. Lors de ce calcul, on peut ne pas calculer les intersections avec les objets neutres car ils ne peuvent intervenir dans un calcul de réfraction. Une fois calculée cette énergie de réfraction, on la multiplie par le coefficient k_t pour déterminer sa contribution dans l'énergie totale.

5.5.3.4 Transmission de l'énergie

On a vu que l'indice de réfraction influait sur la direction de l'énergie. Le coefficient d'extinction quant à lui influe sur les modules de l'énergie. De façon plus précise, lorsque l'énergie traverse un milieu

2. ou de Snell par les Anglo-Saxons...

de coefficient d'extinction k sur une longueur d, les modules de rouge-vert-bleu sont multipliés par un coefficient e^{-kd}. Ceci ne concerne bien sûr que la transmission volumique de l'énergie, puisque le modèle de Whitted est identique à celui de Phong (ou Lambert) en ce qui concerne la transmittance.

On peut donc remarquer que la transmission n'est pas modifiée par l'indice de réfraction. D'autre part, un milieu qui n'atténue pas l'énergie possède simplement un coefficient d'extinction nul.

Enfin, il est parfois plus pratique d'utiliser pour la définition de la transmittance volumique la quantité $d_0 = \frac{\log 2}{k}$. La distance d_0 représente alors la longueur de parcours après laquelle il ne subsiste que la moitié de l'énergie initiale : le coefficient multiplicateur de l'atténuation est simplement $2^{\frac{d}{d_0}}$.

5.6 Tracé de rayons inverse

Dans tout ce que nous avons expliqué précédemment, on cherche à calculer l'énergie arrivant à l'origine du rayon selon la direction du rayon en connaissant les intersections du rayon avec la scène. Une autre utilisation d'un tracé de rayons est possible est consiste, connaissant l'énergie émise à l'origine du rayon selon la direction du rayon et les intersections du rayon avec la scène, à "distribuer" cette énergie dans la scène. Nous appelons cette technique un *tracé de rayons inverse*. Cette notion a été introduite par Arvo en 1986 [Arv86], et elle permet de résoudre certains problèmes comme les sources éclairant une partie de la scène après réflexion sur un miroir, ou réfraction à travers une lentille.

Il faut noter que dans ce cas, on modifie les données de la scène puisque les énergies ainsi réparties doivent être stockées dans lesdites données.

Le processus de rendu fonctionne donc à l'envers par rapport au processus de rendu décrit auparavant. On part de l'énergie initiale arrivant au point d'entrée de l'intervalle d'intersection, puis on répartit cette énergie en allant vers le point de sortie. La notion de profondeur de rendu définie pour le rendu "classique" reste cependant valable ici. Plus précisément, l'algorithme de rendu inverse peut s'écrire :

- s'il n'y pas d'intervalle d'intersection, distribuer l'énergie sur la source dans le cas d'un rayon de type `OMBRE`, dans le fond dans le cas d'un rayon de type `SECONDAIRE` ou `PRIMAIRE`.
- si la profondeur est supérieure ou égale à `SURFACE_IN`
  - distribuer l'énergie totale de cette surface.
  - si le point d'entrée a une transmittance, calculer l'énergie transmise à travers la surface.
- si la profondeur est supérieure ou égale à `VOLUME_IN`
  - distribuer l'énergie totale de la surface d'entrée si celle-ci existe, c'est-à-dire si le point d'entrée existe. Comme pour le processus de rendu classique, il faut affecter comme réflectance à la surface la réflectance volumique.
  - si le contenu a une transmittance, calculer l'énergie transmise à travers le volume.
- si la profondeur est supérieure ou égale à `SURFACE_OUT`
  - distribuer l'énergie totale de la surface de sortie.
  - si le point de sortie a une transmittance, calculer l'énergie transmise à travers la surface de sortie.
- si la profondeur est supérieure ou égale à `DERRIERE`
  - appeler le processus de rendu de façon récursive sur l'intervalle d'intersection suivant.

On peut remarquer que dans cet algorithme, on ne tient pas compte de l'énergie propre (surfacique ou volumique). Ceci tient au fait que ce processus inverse est le plus souvent utilisé pour calculer lesdites énergies propres (radiosités surfacique et volumique par exemple). Le processus de distribution permet donc très souvent de mettre à jour des structures de données qui seront utilisées par les fonctions d'énergie propre.

Chapitre 6

Quelques applications

6.1 Introduction

Ainsi que nous l'avons expliqué dans les chapitres précédents, notre algorithme de tracé de rayons permet de nombreuses extensions à tous les niveaux de l'algorithme.

Nous allons en présenter quelques unes dans ce chapitre, parmi celles qui nous semblent les plus intéressantes ou les moins usuelles. Ces extensions sont :

- l'utilisation de perspectives (ou projections) non classiques,
- l'utilisation de primitives de lumière,
- la visualisation de densités volumiques,
- un essai de parallélisation de l'algorithme.

6.2 Perspectives non classiques

Le problème de la perspective (ou de la projection) est le suivant : comment passer d'une représentation du monde qui est tridimensionnelle à une représentation de l'image qui n'est que bidimensionnelle ? Choisir une méthode de passage reflète une certaine façon de voir le monde.

Il est très traditionnel d'utiliser en synthèse d'images une perspective classique cônique. Cependant, en particulier grâce à la technique à englobants utilisée pour l'accélération des calculs, on peut utiliser pour un calcul d'image n'importe quel type de projection respectant le principe suivant : l'ensemble des points de l'espace se projetant en un pixel de l'écran constitue une droite.

Partant de ce principe, et avec l'aide artistique de Véronique Bourgoin [Bou90], nous avons essayé des perspectives qui nous ont semblé intéressantes soit par leur aspect historique, soit par leurs propriétés géométriques. Il convient de noter que des travaux ont été menés parallèlement à cette étude par Michel Beigbeder [BB92] en utilisant des primitives modélisées par des systèmes de particules et un algorithme de visualisation de type z-buffer. La nécessité de l'utilisation des particules provient du fait que les polygônes traditionnellement utilisés en modélisation sont transformés en morceaux de surfaces gauches par nos perspectives non classiques : il est alors impossible d'utiliser un z-buffer en visualisation.

6.2.1 Repère de vue

On utilise un repère dit *repère du monde* $Oxyz$ pour définir la position des objets dans une scène. Pour définir les caractéristiques des perspectives que nous utilisons, il est commode de définir un *repère* *de vue*. Si nous avons choisi ce nom à la place du classique *repère de l'oeil*, c'est précisément parce que nos perspectives n'utilisent pas forcément un oeil fixe placé à l'origine du repère de l'oeil, et donc la notion de repère de vue nous a semblé plus juste. Dans toutes les perspectives que nous considérons, nous utilisons les caractéristiques suivantes :

- un point de vue E, qui est un point quelconque de l'espace,
- un point de visée A, qui est un autre point quelconque,
- un angle de roulis ψ.

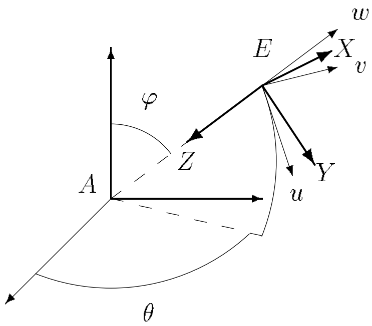


FIG. 6.1 – *Repère de vue*

On note alors $\vec{V}$ le vecteur joignant E à A, θ l'angle entre $\vec{V}$ et le plan Oxz, φ l'angle entre $\vec{V}$ et Oz (θ et φ sont les coordonnées sphériques de $\vec{V}$). Dans le cas où l'angle φ est nul, on considère que l'angle θ est également nul (il y a indétermination pour l'angle θ). On construit alors à partir de ces données le repère de vue de la façon suivante :

- on considère le repère $Axyz$ obtenu par translation en A du repère initial.
- dans ce repère, on considère le trièdre local sphérique en E, que l'on note $Euvw$. L'axe Ew est colinéaire à $\vec{V}$, l'axe Ev est orthogonal à Ew et dans un plan parallèle à Axy, et l'axe Eu complète le repère. Dans le cas où $\vec{V}$ est colinéaire à Az, il y a indétermination du trièdre local: on utilise alors la même convention que ci-dessus, à savoir que l'on prend l'axe Ev colinéaire à Ay.
- on considère alors un repère $EX_1Y_1Z_1$ construit à partir de $Euvw$ de la façon suivante: l'axe EX_1 est colinéaire à Ev, l'axe EY_1 est colinéaire à Eu et l'axe EZ_1 est opposé à Ew.
- on construit le repère de vue $EXYZ$ par rotation du repère $EX_1Y_1Z_1$ autour de l'axe EZ_1, l'angle de cette rotation étant l'angle de roulis ψ.

Cette définition du repère de vue est valable pour toutes les perspectives que nous utilisons. On utilise souvent des dénominations pour les axes du repère qui ne sont valables que dans certains cas. En particulier, on appelle *axe horizontal* l'axe EX, *axe vertical* l'axe EY et enfin *axe de profondeur* l'axe EZ.

6.2.1.1 Plans près et loin

Dans les algorithmes de type z-buffer (ou Atherton), on utilise classiquement un *plan près* et un *plan loin* pour fenêtrer les polygônes à visualiser. Ceci permet également d'éviter un certain nombre de problèmes lorsque des polygônes passent par l'oeil, ou encore ont une partie devant l'oeil et une partie derrière.

Une telle chose n'est pas strictement nécessaire pour un algorithme de tracé de rayons. Il n'y a en effet aucun problème à ce que l'oeil se situe dans un objet : on peut penser par exemple à calculer une image où l'oeil est dans l'eau. ependant, ainsi que nous l'avons indiqué au chapitre 3, nous ne cherchons les intersections entre un rayon et une scène que sur un segment de ce rayon, c'est-à-dire entre une abscisse minimale et une abscisse maximale sur le rayon. L'abscisse maximale peut être par exemple la distance à partir de laquelle on est sûr de ne plus rencontrer d'objet. Quant à l'abscisse minimale, on la prend la plupart du temps nulle (rien n'interdit d'ailleurs une valeur négative).

On utilise pour définir ces deux abscisses limites la distance à l'oeil. Plus précisément, l'utilisateur précise deux distances, appelées $d_{près}$ et d_{loin}, et les intersections ne seront cherchées qu'entre ces deux distances à l'oeil. Si $\vec{D}$ est le vecteur de direction d'un rayon, on calcule alors les abscisses limites par

$$\begin{aligned}\lambda_{min} &= \frac{d_{près}}{||\vec{D}||} \\ \lambda_{max} &= \frac{d_{loin}}{||\vec{D}||}\end{aligned}$$

Notons ici que l'unité pour les distances est laissée à l'appréciation de l'utilisateur : selon les modèles, celà peut aller du nanomètre à l'année-lumière....

6.2.1.2 Coordonnées écran

Les images que nous utilisons habituellement sont des images rectangulaires. De façon à simplifier les notations par la suite, nous allons introduire tout de suite ce que l'on appelle les *coordonnées image*. Une image est en principe un tableau de valeurs numériques (des couleurs), mais il est souvent plus pratique d'associer à cette représentation discrète une représentation continue.

Nous utilisons dans ce but un repère-image, dont l'origine est le centre I de l'image, dont l'axe IX correspond aux lignes horizontales de l'image parcourues de gauche à droite, et dont l'axe IY correspond aux colonnes de l'image, parcourues de haut en bas. Nous normalisons ensuite ces coordonnées entre -1 et 1, le coin supérieur gauche de l'image possède ainsi les coordonnées $(-1,-1)$ et le coin inférieur droit les coordonnées $(1,1)$.

Notons que cette convention est tout à fait arbitraire, et ne possède donc aucune justification : d'autres systèmes utilisent des coordonnées comprises entre 0 et 1, ou entre $-\frac{1}{2}$ et $\frac{1}{2}$.

6.2.2 Perspective classique

6.2.2.1 Un peu d'histoire

Rappelons tout d'abord les origines de la perspective que nous nommons *classique*, à savoir la perspective cônique. Son objectif est le *réalisme*, c'est-à-dire la volonté de restituer des images planes aussi fidèles que possible à la réalité.

Cette technique n'a rien de neuf puisqu'elle provient de la technique dite *camera obscura* (la chambre noire). Une pièce sombre possède l'un de ses murs percé d'un trou permettant le passage de la lumière. Alors, sur le mur opposé au perçage se forme une image (inversée) de ce qui est visible par le perçage.

On obtient une perspective comparable en utilisant un miroir et en tournant le dos à la direction dans laquelle se trouve la scène à visualiser : il se forme sur le miroir une image plane par réflexion de la scène. Dans ce cas, la scène subit une inversion de sens gauche-droite.

Cette technique fut ensuite réintroduite durant le XV$^{\text{ème}}$ siècle (le *Quattrocento* très riche dans l'histoire artistique) par Brunelleschi, puis formalisée de façon mathématique par Alberti en 1435 [Alb35]. On peut considérer que cette formalisation est la première description de l'algorithme du tracé de rayons.

A la fin du XVI$^{\text{ème}}$ siècle, Giambattista della Porta améliore la technique en installant des lentilles dans le perçage. L'utilisation de ces lentilles s'améliorera au fil des siècles. Ensuite, au XVIII$^{\text{ème}}$ siècle, la camera obscura va devenir portable et donc permettre la saisie de plus nombreuses scènes. Puis, en 1822, Nicéphore Niepce introduira une plaque sensible, inventant la photographie. Enfin, la synthèse d'images actuelle perpétue cette notion de réalisme.

6.2.2.2 Principe de projection

Une fois défini le repère de vue, on peut définir le principe de projection cônique. Ceci nécessite un certain nombre de paramètres supplémentaires, qui sont :

- le demi-angle d'ouverture en largeur α
- le demi-angle d'ouverture en hauteur β

Ces deux angles doivent être positifs et strictement inférieurs à $\frac{\pi}{2}$. Le plan de projection est alors tout simplement le plan d'équation $Z = 1$ dans le repère de vue. L'*écran* est la partie de ce plan vérifiant de plus $|X| \leq \tan\alpha, |Y| \leq \tan\beta$. Le centre de cet écran est donc le point de coordonnées $(0, 0, 1)$.

L'oeil est quant à lui situé à l'origine du repère de vue. En un point de l'écran se projettent les objets se trouvant sur le segment dont le support est la droite issue de l'oeil et passant par ce point, limité par les restrictions sur la distance à l'oeil. La couleur vue en ce point est alors bien sûr la couleur de l'objet le plus proche.

6.2.2.3 Calcul d'une image

La dernière phase de la projection consiste à calculer une *image numérique*, c'est-à-dire un échantillonnage de l'image continue représenté par l'écran de projection. Il est nécessaire alors de connaître la taille de l'image, soit le nombre de lignes n_L et le nombres de colonnes n_C. On appelle alors *pixel-écran* une partie de l'écran obtenue en effectuant un maillage bidimensionnel régulier de l'écran, le nombre de subdivisions en X étant n_C et le nombre de subdivisions en Y étant n_L. On appelle *pixel-image* un élément de la matrice de données stockée pour former l'image. Il y a donc une correspondance biunivoque entre les pixels-écran et les pixels-image.

Il reste maintenant à savoir comment on calcule les pixels-image. Notre système n'impose aucune restriction pour ce calcul, nous avons choisi actuellement de calculer la couleur d'un rayon lancé au centre du pixel-écran correspondant.

Dans un but d'antialiassage, il est tout à fait possible d'utiliser un point tiré aléatoirement dans le pixel-écran, ce qui remplace alors l'aliassage par du bruit. On peut même utiliser des points multiples et ensuite effectuer un filtrage. Enfin, il est possible d'effectuer un suréchantillonage adaptatif, ce qui a été réalisé dans un cadre plus général de raffinements successifs d'une image [MCP92].

6.2.2.4 Pixels carrés

Il est souvent très désagréable de décrire une scène contenant par exemple des sphères, et de voir apparaître ces sphères comme des ellipsoïdes à l'écran. Ceci provient parfois du fait que le découpage de l'écran en pixels ne respecte pas toujours l'isométrie, c'est-à-dire que la largeur du pixel n'est pas toujours égale à la hauteur.

Nous avons donc introduit la possibilité pour l'utilisateur de demander à ce que les pixels de l'écran soient bien des carrés. Le système recalcule alors le demi-angle d'ouverture en hauteur β en fonction du demi-angle d'ouverture en largeur α, du nombre de lignes n_L et du nombre de colonnes n_C. Ce calcul est le suivant :

$$\beta = \text{arctg}\left(\frac{n_L}{n_C}\text{tg}\alpha\right)$$

Cette correction n'est pas obligatoire, ce qui laisse à l'utilisateur la possibilité de créer des effets particuliers en forçant des pixels non carrés.

6.2.3 Perspectives à oeil mobile

Dans ces types de perspective, l'oeil, au lieu d'être fixe comme pour une perspective classique, possède une position différente en fonction du pixel (ou de la position sur l'écran) où l'on calcule une couleur. Dans les deux exemples que nous donnons ici, l'oeil subit un déplacement selon l'axe vertical (dans le repère de vue).

6.2.3.1 Perspective en arête de poisson

Ce type de perspective a été très utilisé avant la Renaissance et la perspective Brunelleschienne. Le déplacement en Y est de la forme $\Delta Y = k|x_{pixel}|$, où x_{pixel} désigne la coordonnée en X du point de calcul.

Cette perspective tient son nom du fait qu'une ligne horizontale dans le repère de vue (c'est-à-dire d'équations $Z = z_0, Y = y_0$) se trouve projetée sous forme d'un chevron, soit deux segments de droite symétriques par rapport à l'axe médian de l'écran.

La constante k peut être choisie arbitrairement et l'utilisateur peut en préciser la valeur, cependant, dans les exemples que nous avons pris, nous avons choisi k de façon à ce que la droite d'équation $Y = Z = 0$ passe par les coins de l'image, et elle est donc égale au quotient de la résolution verticale par la résolution horizontale.

Un autre type de perspective appelée perspective à faisceaux croisés utilise un principe similaire, à la différence près que les points se projetant sur un pixel de l'écran constituent deux droites, symétriques par rapport au plan médian horizontal de l'écran : nous n'avons pas implanté cette perspective pour le moment mais ceci resterait possible en effectuant deux tracés de rayons simultanés, et en retenant pour chaque pixel le rayon intersectant l'objet le plus proche.

6.2.3.2 Perspective hyperbolique

Dans ce cas, le déplacement en Y est de la forme $\Delta Y = \frac{k}{x_{pixel}}$, où x_{pixel} désigne toujours la coordonnée en X du pixel de calcul.

Là encore, la constante k peut être choisie arbitrairement et nous utilisons une valeur par défaut telle que le déplacement soit égal à la moitié de la hauteur de l'écran lorsque $|x_{pixel}|$ est égal au sixième de la largeur de l'écran. La largeur de l'écran étant $2\text{tg}\alpha$ et la hauteur $2\text{tg}\beta$, on a donc

$$k = \frac{\text{tg}\alpha\text{tg}\beta}{3}$$

Cette perspective tient son nom du fait qu'une ligne horizontale dans le repère du monde se trouve sous forme de deux branches d'hyperbole après projection.

Remarquons enfin que la position de l'oeil n'est pas définie pour $x_{pixel} = 0$ (dans ce cas, l'oeil devrait se trouver à l'infini). Il faut donc prendre garde à ne pas calculer de rayon sur l'axe médian vertical de l'écran.

6.2.4 Perspectives à grand angle

Ainsi que ceci a été noté pour la perspective classique (et il en est de même pour les perspectives à oeil mobile), on ne peut utiliser une ouverture horizontale ou verticale supérieure à $\frac{\pi}{2}$, ce qui veut dire que le champ de vision est limité à un demi-espace.

Encore convient-il de noter que pour des valeurs d'angles importantes, on obtient des distorsions visuelles importantes. En particulier, les objets placés vers les bords de la pyramide de vision se trouvent artificiellement grandis.

6.2.4.1 Perspective stéréographique

La projection stéréographique permet de projeter une sphère privée de l'un de ses points sur un plan diamétral orthogonal à la droite joignant le point au centre de la sphère. Par analogie avec la sphère terrestre, on appelle *pôle nord* le point choisi et *plan équatorial* le plan de projection. Le principe est alors le suivant : soit S la sphère, N le pôle et P_E le plan équatorial, alors à tout point P de $S\backslash\{N\}$ est associé le point de P_E qui est l'intersection de la droite NP et du plan P_E (voir figure 6.2). Ainsi, dans

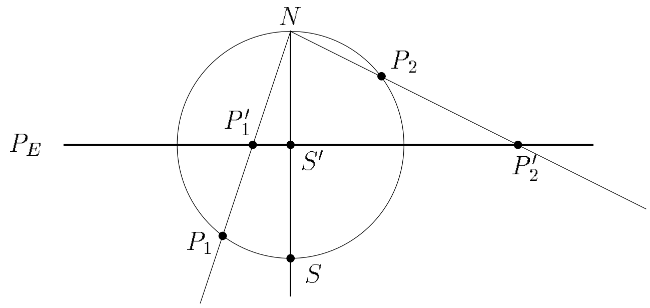


FIG. 6.2 – *Projection stéréographique*

une telle perspective, le pôle opposé (ou *pôle sud*) se projette à l'origine du plan équatorial, les points de l'équateur sont invariants, les points de l'hémisphère nord (P_2 par exemple) se projettent à l'extérieur du cercle équatorial (en P'_2), les points de l'hémisphère sud se projettent à l'intérieur de ce cercle (voir P_1 et P'_1). D'autre part, les parallèles de la sphère se projettent sur des cercles, les méridiens se projettent sur des droites passant par l'origine du plan.

Une autre propriété intéressante de cette perspective est la conservation des angles : deux courbes tracées sur la sphère et s'intersectant en faisant un angle α sont projetées en deux courbes sur le plan s'intersectant avec le même angle α. Ceci se retrouve en particulier sur les parallèles et méridiens qui s'intersectent orthogonalement.

Le principe de la perspective stéréographique est alors le suivant : l'oeil se trouve au centre d'une sphère unitaire et la direction de visée détermine la position du pôle sud. Ainsi, le pôle nord se trouve "derrière" l'oeil. Tout rayon issu de l'oeil intersecte la sphère en un point I, et on projette ce point I stéréographiquement sur le plan équatorial en I'.

Ainsi, l'espace visible est l'espace entier privé de la direction du pôle nord. Dans la pratique, on interdit les directions dont l'angle avec la direction du pôle nord est supérieur à une valeur limite δ_{max}.

L'image obtenue par cette perspective est alors un disque, dont le rayon est $\mathrm{tg}\frac{\delta_{max}}{2}$, le centre de ce disque étant ce que l'on voit dans la direction principale de visée, et le cercle limite étant ce que l'on voit dans les directions limites. Comme nous calculons des images rectangulaires, nous avons pris la convention de centrer le disque représentant l'image dans ce rectangle, et de l'y rendre tangent. Les pixels se trouvant dans le rectangle en dehors du disque prennent par convention la couleur noire. On peut détailler le calcul

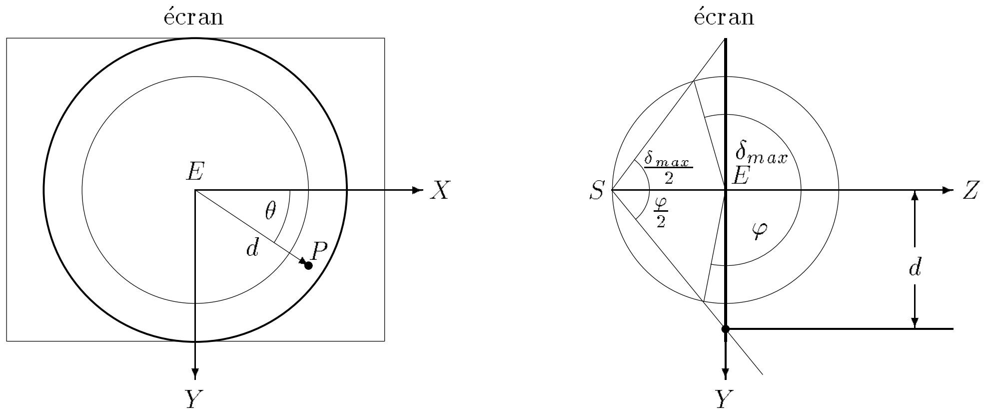


FIG. 6.3 – *Calcul de la direction d'un rayon primaire en perspective stéréographique*

de la direction des rayons primaires en fonction de la position sur l'écran. On note :

- D le diamètre du cercle de diamètre maximal inscrit dans l'écran et dont le centre est le centre de l'écran,
- x_p et y_p les coordonnées écran du pixel de calcul,
- d la distance du pixel au centre de l'écran (soit $\sqrt{x_p^2 + y_p^2}$).

Le rayon primaire associé à ce pixel a comme origine (dans le repère de vue) l'oeil, soit $(0, 0, 0)$, et la direction de rayon peut être déterminée par ses deux angles (θ, φ) des coordonnées sphériques (toujours dans le repère de vue). L'angle θ se calcule très facilement :

$$\theta = \mathrm{arctg}\frac{y_p}{x_p}$$

Pour calculer l'angle φ (voir figure 6.3), on note ψ l'angle $(\widehat{EZ, SP})$. On a alors la relation :

$$\frac{D}{d} = \frac{\mathrm{tg}\frac{\delta_{max}}{2}}{\mathrm{tg}\psi}$$

Or on sait que l'arc intercepté par un secteur pointé sur la circonférence d'un cercle est égal à la moitié de l'arc intercepté par un secteur pointé au centre du cercle et passant par les mêmes points de la circonférence. On a donc $\varphi = 2\psi$ et on peut calculer φ par

$$\varphi = 2\mathrm{arctg}\left(\frac{d}{D}\ \mathrm{tg}\frac{\delta_{max}}{2}\right)$$

La direction $\vec{V}$ du rayon se calcule alors par

$$\vec{V} = \begin{pmatrix} \cos\theta\sin\varphi \\ \sin\theta\sin\varphi \\ \cos\varphi \end{pmatrix}$$

Il convient de noter la cas particulier où θ est indéterminé, c'est-à-dire au centre de l'écran. On constate cependant que l'indétermination n'est qu'apparente, car φ étant nul, la direction du rayon est colinéaire à EZ.

6.2.4.2 Perspective sphérique

La perspective stéréographique décrite au paragraphe précédent, ne permet pas la vue de l'espace entier. De plus, lorsque l'on prend des valeurs faibles de l'angle limite, le disque projeté prend des valeurs importantes de rayon, et les changements d'échelle nécessaires pour intégrer cette projection dans une image rectangulaire “compriment” les directions proches de la direction principale.

Nous avons donc essayé de remédier à ce phénomène en introduisant une perspective, dite *sphérique*, qui permet de projeter tout l'espace en limitant les compressions.

Le principe en est le suivant : l'oeil est encore situé au centre de la sphère S, et le plan de projection est tangent à cette sphère, orthogonal à la direction principale de visée. Le point de tangence est A, le point de visée. Par tout point P de la sphère, on peut faire passer un plan Π contenant également le centre de la sphère (l'oeil) et le point A. Ce plan Π intersecte la sphère selon un arc de cercle C et le plan de projection selon une droite Δ (en fait, on ne retient que la demi-droite située du même côté que le point P). Alors, tous les points de la demi-droite $[S, P)$ se projettent au point p de Δ dont la distance à A est égale à la longueur de l'arc de cercle C.

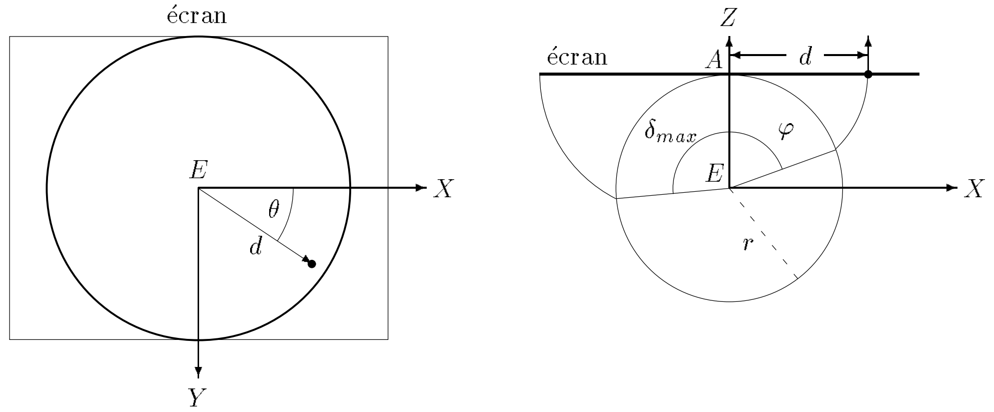


FIG. 6.4 – *Calcul de la direction d'un rayon primaire en perspective sphérique*

Le principe ci-dessus est applicable pour un écran infini, mais dans la pratique, nous utilisons un écran fini. En fait, on utilise un angle d'ouverture δ_{max}, tel que le diamètre de l'écran soit égal à $\delta_{max} r$. Notons d'ailleurs que rien n'interdit d'avoir un angle d'ouverture supérieur à π. Dans ce cas, on trouve sur l'image une série de cercles centrés sur le centre de l'écran correspondant à ce qui est vu dans la direction opposée à la direction de visée (la distance au centre de l'écran est alors un multiple impair de πr), et éventuellement une série de cercles correspondant à ce qui est vu dans la direction de visée (la distance est alors un multiple pair de πr).

Comme pour la perspective stéréographique, ce sont les coordonnées sphériques qui conviennent le mieux pour calculer la direction d'un rayon primaire (notons que l'oeil est toujours en $(0, 0, 0)$). Plus précisément, si (x_p, y_p) sont les coordonnées écran du pixel de calcul, on calcule θ comme précédemment soit

$$\theta = \operatorname{arctg}\frac{y_p}{x_p}$$

L'angle φ est très simple à calculer puisque l'on a

$$\begin{aligned}\varphi &= \frac{d}{r}\\ \delta_{max} &= \frac{D}{r}\end{aligned}$$

On a donc

$$\varphi = \delta_{max}\frac{d}{D}$$

6.2.5 Perspectives convergentes

Les deux perspectives précédentes permettent de voir tout l'espace (ou tout du mois une partie importante de l'espace) depuis un point d'observation unique. Nous allons maintenant décrire deux types de perspectives permettant de réaliser l'inverse, c'est-à-dire l'observation d'un point unique depuis de multiples points d'observation.

D'un point de vue artistique, on peut noter que les artistes ont souvent cherché à représenter sur une seule image un même objet vu sous des angles différents, c'est-à-dire avec des points de vue différents. On peut citer dans cette catégorie les artistes de l'Antiquité Egyptienne, qui tracaient toujours les personnages simultanément de face et de profil. A une époque plus récente, les peintre cubistes utilisaient des principes similaires.

Notons enfin que les deux perspectives dites "à oeil mobile" citées au début de ce paragraphe (perspective en arête de poisson et perspective hyperbolique) ne sont pas considérées comme appartenant à cette famille car il n'y a pas convergence des rayons en un point (ou en une droite) unique.

6.2.5.1 Perspective boulique

Dans ce type de perspective, c'est maintenant le point de visée A qui se situe au centre d'une sphère S, et l'écran de projection est développé sur cette sphère.

On utilise un système de projection en latitude et longitude. Plus précisément, on associe à tout point de l'image une latitude et une longitude. Par convention, le centre de l'écran a une latitude et une longitude nulle, le bord droit de l'écran correspond à une latitude maximale l_{max}, le bord gauche à l'opposé de cette latitude maximale, le bord supérieur à une longitude maximale L_{max} et le bord inférieur à l'opposé de cette longitude maximale. Les angles de latitude et de longitude limites sont des paramètres de la projection.

Ensuite, pour un point (x_p, y_p) de l'écran de projection, on calcule sa longitude et sa latitude par simple interpolation linéaire entre les longitudes et latitudes limites définies sur les bords de l'écran. L'oeil est alors placé sur la sphère au point défini par ces longitude et latitude. Les lignes horizontales de l'écran correspondent alors aux parallèles, les lignes verticales correspondant quant à elles aux méridiens.

On peut ainsi noter que ce type de perspective permet de voir un objet sous toutes ces faces sur une seule vue. Dans le cas où la longitude limite est supérieure à 90 degrés, on obtient des lignes horizontales de couleur uniforme, correspondant à la couleur vue en regardant d'un des pôles : en effet, pour une longitude multiple de 90 degrés, l'oeil est placé au pôle quelle que soit la latitude (la latitude n'est pas définie aux pôles).

Notons enfin un phénomène curieux avec cette perspective : si un objet est situé derrière le point visé (le centre de la sphère), il est vu à l'envers sur l'image. De même, si un objet est présent exactement au centre de la sphère, il est potentiellement visble sur tous les pixels de l'image (si aucun autre objet ne vient le masquer).

6.2.5.2 Perspective circulaire

Le concept de cette perspective se rapproche de celui de la perspective boulique, à savoir la volonté de visualiser un objet sous différents points de vue simultanément. Dans le cas de la perspective circulaire, l'oeil est mobile sur un arc de cercle $\mathcal{C}$, et l'écran est développé sur une portion de cylindre coaxiale de $\mathcal{C}$. Deux angles servent à déterminer complétement cette perspective, un angle d'ouverture en largeur α

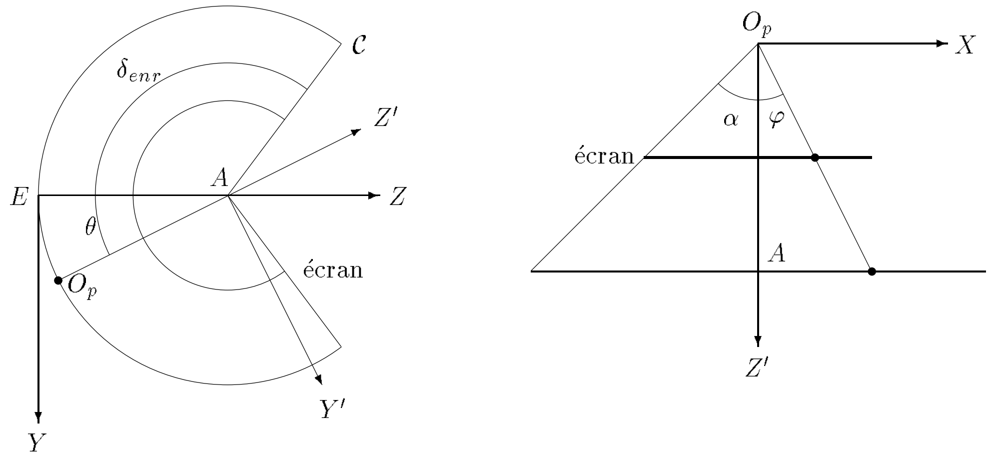


FIG. 6.5 – *Calcul d'un rayon primaire en perspective circulaire*

(similaire à celui utilisé en perspective classique ou hyperbolique), et un angle d'enroulement δ_{enr}. On note d la distance entre le point de vue et le point visé. Alors,

- l'oeil O_p se déplace sur l'arc de cercle $\mathcal{C}$, dans le plan $X = 0$, centré en A, de rayon d, et tel que l'angle $(\widehat{AE, AO})$ soit inférieur à l'angle d'enroulement.
- le point visé se déplace sur un segment centré en A, parallèle à l'axe EX, de longueur $2d\text{tg}\alpha$. Les rayons ne sont donc pas convergents en un point comme pour la perspective boulique, mais sur un segment de droite.
- l'écran peut être développé en une portion de cylindre, d'axe AX, de rayon $\frac{d}{2}$, limité à $|X| \leq \frac{d}{2}\text{tg}\alpha$.

Détaillons maintenant le calcul de la position de l'oeil (l'origine du rayon), et de la direction du rayon (voir la figure 6.5). On note toujours x_p et y_p les coordonnées écran du point de calcul. Alors, si l'on note θ l'angle $x_p\delta_{enr}$, et A_p le point visé pour ce point de l'image, on a

$$O_p \quad = \quad \begin{pmatrix} 0 \\ d\sin\theta \\ d(1-\cos\theta) \end{pmatrix}$$

$$A_p = \begin{pmatrix} Y_p d\text{tg}\alpha \\ 0 \\ d \end{pmatrix}$$

La direction du rayon se calcule alors par simple différence entre ces coordonnées.

Notons que pour des valeurs d'angle d'enroulement supérieures à 180 degrés, on observe une périodicité entre les lignes de l'image. D'autre part, de la même façon que pour la perspective boulique, un objet situé derrière le point visé apparaît inversé.

6.2.6 Implantation

D'un point de vue pratique, les diverses perspectives décrites précédemment sont implantées sous la forme de fonctions de génération de rayons primaires. Ces fonctions admettent trois modes de fonctionnement :

- un mode d'initialisation, qui permet d'effectuer toutes les initialisations nécessaires, comme le calcul de la matrice de passage du repère de vue au repère du monde, ou encore le recalage des angles d'ouverture pour obtenir des pixels carrés.
- un mode "automatique", qui permet de générer successivement tous les rayons primaires. Nous avons pris la convention de générer les pixels ligne par ligne, et de gauche à droite au sein d'une même ligne car c'est dans cet ordre que nous stockons les fichiers-image.
- un mode "absolu", où l'utilisateur précise quel est le pixel pour lequel il faut générer le rayon primaire.

Toutes les fonctions de génération de rayons primaires admettent les mêmes arguments, que nous allons détailler.

- le premier argument est un pointeur sur les données globales de la scène. Celles-ci sont en effet nécessaires puisque l'on doit connaître les positions du point de vue et du point visé, les éventuels angles caractéristiques, ainsi que le nombre de lignes et de colonnes de l'image finale.
- le deuxième argument est un pointeur sur une structure "rayon", dont la fonction doit remplir les champs (origine, direction,...).
- le troisième argument est un pointeur sur la structure d'informations relatives à ce rayon. Cette structure est utilisée par l'algorithme de rendu pour relancer d'éventuels rayons secondaires (voir le chapitre 5). Notons également que c'est dans la fonction de génération de rayons primaires qu'est établie la numérotation des rayons primaires, nécessaire pour l'algorithme de parallélisation en particulier.
- le quatrième argument est un pointeur sur un pointeur une structure de données spécifique à chaque type de perspective. Cette structure est utilisée pour stocker le *pixel courant* dans le cas du mode automatique de génération, ainsi que pour stocker diverses valeurs qui permettent d'accélérer les calculs. Comme cette structure est spécifique de chaque perspective, le pointeur est passé sous forme d'un pointeur sur pointeur banalisé. Remarquons que c'est la fonction de génération qui est chargée d'allouer la place mémoire nécessaire au stockage de cette structure : en effet, cette structure n'est connue que de la fonction en question. Cette allocation est réalisée lors de la procédure d'initialisation.
- le cinquième argument est le mode de fonctionnement désiré.

- les sixième et septième arguments sont optionnels et servent à préciser le pixel désiré dans le cas où le mode est le mode absolu.

Notons que cette fonction rend toujours un résultat, qui permet de détecter d'éventuelles erreurs (valeurs manquantes pour les paramètres de vue, pixel demandé en dehors des limites de l'écran) et également la fin de la génération dans le cas du mode automatique.

Notons également que l'initialisation est obligatoire en début de génération (elle peut d'ailleurs être réappelée si besoin est), et que les modes automatique et absolu peuvent être appelés alternativement : le mode absolu ne perturbe pas le mode automatique.

Le choix entre les différentes primitives se fait par l'intermédiaire d'un pointeur sur fonction, dont la valeur est choisie en fonction du type de perspective. Ce pointeur est stocké dans les données globales de la scène. Nous utilisons une fonction qui permet, en fonction du nom de la perspective désirée, de trouver la fonction de génération de rayons primaires correspondante.

Le grand avantage de ce système est de permettre une très grande évolutivité. En effet, ajouter de nouvelles perspectives nécessite simplement l'écriture de la fonction de génération de rayons primaires correspondante, et une très légère modification de la fonction de recherche.

6.3 Primitives de lumière

6.3.1 Principe général

Si l'on considère les sources lumineuses, la plupart des modèles ne prennent en compte que l'éclairement des objets placés dans la scène. Il y a cependant un cas où cette prise en compte est insuffisante, c'est le cas où une partie de l'espace contient des particules en suspension (brouillard, fumée,...), et que cette partie de l'espace est éclairée : le faisceau lumineux devient alors visible et apporte ainsi une contribution supplémentaire à l'éclairement.

Afin d'intégrer ces phénomènes dans un modèle de rendu, nous avons choisi de représenter ces faisceaux lumineux par des primitives de modélisation et de leur associer les propriétés de la source lumineuse correspondante. A partir d'une première version de tracé de rayons développée dans notre équipe [Arg88], une extension a été apportée intégrant le principe des primitives de lumière [Fer90]. Cependant, ce traitement se faisait au prix d'un traitement spécifique pour l'algorithme d'intersection et posait donc de grands problèmes d'évolution. De plus, l'algorithme de rendu était tout à fait empirique.

Notre nouveau modèle se prête bien mieux à une telle extension puisque les primitives de lumière ne sont qu'un des exemples possibles d'objets neutres (voir le chapitre 2). Nous avons également introduit un modèle plus physique et ainsi déterminé un nouvel algorithme de rendu pour ces primitives ([RFP90]). Nous allons maintenant détailler comment nous prévoyons d'intégrer ces primitives de lumière dans notre modèle (cette intégration n'est pas réalisée à ce jour).

6.3.2 Modélisation

6.3.2.1 Sources et primitives

Ainsi que nous l'avons indiqué, les primitives de lumière font partie des objets neutres : elles ont en effet la propriété de ne pas dévier la lumière mais simplement d'en modifier l'intensité. Il est donc nécessaire de tenir compte des objets situés derrière une éventuelle primitive de lumière afin d'effectuer un rendu correct.

Dans la première version des primitives de lumière, on se restreint à deux types de primitives (qui permettent donc deux formes de faisceau lumineux) qui sont le cône et le cylindre. De nombreux effets de lumière sont ainsi représentables à l'aide de ces deux primitives. A ces primitives sont associées des sources dont le type varie selon le type de faisceau:

- une source ponctuelle pour un cône, la source étant située au sommet du cône.
- une source surfacique pour un cylindre, celle-ci ayant comme surface émettrice la face supérieure du cylindre. Notons que la lumière est émise dans une seule direction, qui est l'axe de ce cylindre.

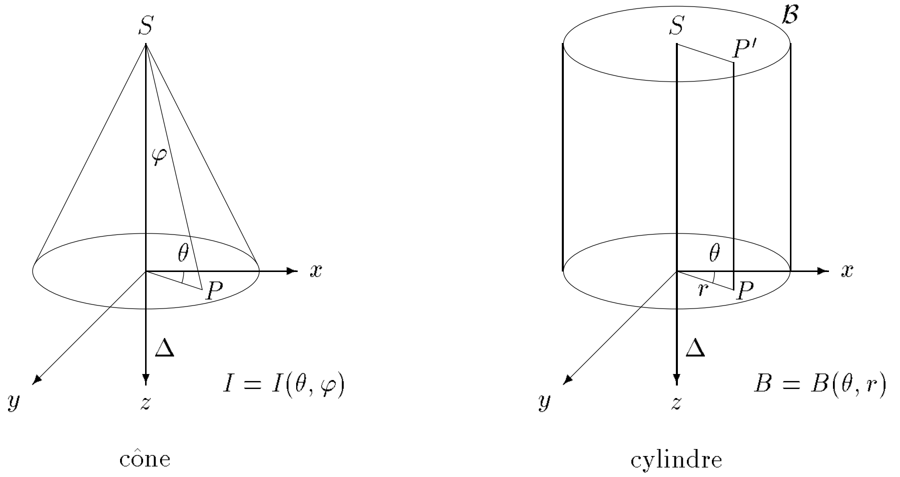


FIG. 6.6 – *Géométrie des primitives de lumière*

Pour chaque source est supposée connue la distribution spatiale d'énergie. Plus précisément, la distribution d'énergie peut se représenter par une fonction dépendant de deux paramètres de direction. Le plus simple est d'utiliser pour les sources ponctuelles (et donc les primitives de type cône) une distribution en fonction des angles des coordonnées sphériques associées au cône, et pour les sources cylindriques une paramétrisation en coordonnées cylindriques.

Notons que dans le cas du cône, la distribution est une distribution d'intensité (source ponctuelle), alors que dans le cas du cylindre, c'est une distribution de radiosité (source surfacique).

Enfin, pour chacune des primitives, on utilise un système de coordonnées le plus adapté. Ainsi, tout point P situé à l'intérieur d'une primitive de lumière sera repéré

- dans le cas du cône, par des coordonnées sphériques associées à un repère dont l'origine est le sommet S du cône et dont l'axe Δ est l'axe du cône. On notera ainsi
  - $r(P)$ la distance de P à S,
  - $\varphi(P)$ l'angle entre $\vec{SP}$ et Δ,
  - $\theta(P)$ l'angle de rotation autour de Δ. Ce dernier angle est défini par rapport à un axe arbitraire orthogonal à Δ.
- dans le cas du cylindre, par des coordonnées cylindriques associées à un repère dont l'origine est le centre S de la base $\mathcal{B}$ du cylindre, dont l'axe est l'axe Δ du cylindre. On notera ainsi
  - $r(P)$ la distance de S à P', projection orthogonale de P sur $\mathcal{B}$,

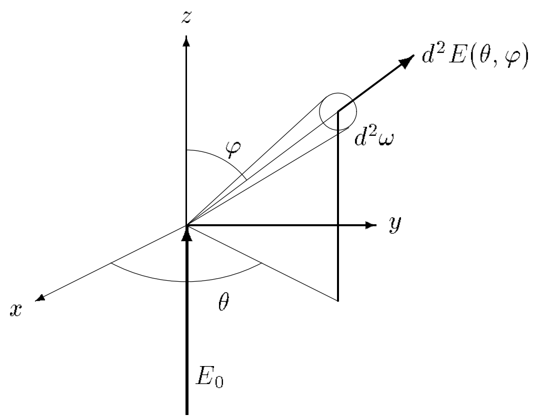


FIG. 6.7 – *Fonction de phase*

- $\theta(P)$ l'angle de rotation autour de Δ, défini comme dans le cas du cône par rapport à une référence arbitraire,
- $z(P)$ la distance de $\mathcal{B}$ à P.

6.3.2.2 Intérieur des primitives

Le volume occupé par ces primitives est supposé être rempli par des particules sphériques. La distribution de ces particules est uniforme à l'intérieur de la primitive, et deux paramètres permettent de contrôler cette distribution :

- le rayon des particules ρ (supposé constant),
- la densité volumique moyenne μ, représentant le pourcentage de l'espace occupé par lesdites particules. On verra dans la suite que cette densité est supposée faible par rapport à 1, ce qui permet un certain nombre d'approximations.

Un autre caractéristique importante des particules est précisée par la *fonction de phase*. Cette fonction de phase permet de connaître le comportement de la lumière après avoir rencontré une particule. Plus précisément, la fonction de phase est le quotient de l'énergie réémise dans un angle solide élémentaire $d^2\omega$ autour d'une direction D divisé par l'énergie incidente (voir la figure 6.7). On peut donner une interprétation très simple de la fonction de phase : si l'on considère la particule comme une source ponctuelle, la distribution spatiale d'intensité associée à cette source est précisément la fonction de phase.
La fonction de phase peut alors être exprimée par :

$$f(\theta,\varphi) = \frac{d^2E(\theta,\varphi)}{d^2\omega E_0}$$

La conservation de l'énergie s'exprime par le fait que toute l'énergie incidente est réémise, soit

$$\int f(\theta,\varphi)d^2\omega = 1$$

ou encore

$$\int_{\theta=0}^{2\pi}\int_{\varphi=0}^{\pi} f(\theta,\varphi)\sin\varphi d\theta d\varphi = 1$$

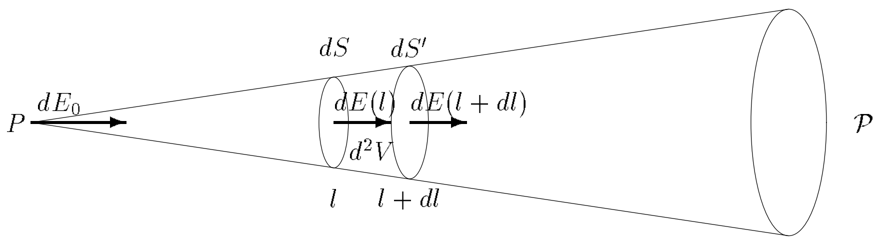


FIG. 6.8 – *Transmission de la lumière*

La plupart du temps, cette fonction de phase possède une symétrie cylindrique autour de l'axe d'incidence et ne dépend donc pas de θ. L'équation de conservation peut alors s'écrire

$$\int_0^{\pi} f(\varphi) \sin \varphi d\varphi = \frac{1}{2\pi}$$

La fonction de phase peut prendre diverses formes parmi lesquelles on peut citer la fonction isotropique (constante), la fonction de Rayleigh, de Mie, de Henyey-Greenstein [PR92]. Le type de fonction de phase peut être choisi d'après le type de particules que l'on veut simuler.

Notons que l'équation de conservation d'énergie n'est pas toujours utilisée. En effet, dans certains cas, on utilise la valeur de l'albédo, qui représente la proportion d'énergie réémise par la particule. Cet albédo n'est pas toujours égal à 1, ce qui correspond alors à une transformation de l'énergie lumineuse en une autre forme d'énergie (thermique par exemple). Si on note a l'albédo, l'équation de conservation de l'énergie s'écrit alors

$$\int_0^{\pi} f(\varphi) \sin \varphi d\varphi = \frac{a}{2\pi}$$

6.3.3 Rendu

Les primitives de lumière interviennent alors de deux façons dans l'algorithme de rendu :

- par leur transmittance volumique, car les particules interceptent la lumière provenant des objets situés derrière la primitive ou à l'intérieur de la primitive,
- par l'éclairement des particules elles-mêmes, ce qui peut être modélisé par de l'énergie propre volumique.

Détaillons ces deux composantes.

6.3.3.1 Transmittance volumique

Précisons comment l'énergie est transmise à travers le milieu partiellement occupé par les particules. On note $\mathcal{P}$ un pinceau élémentaire de lumière, de sommet P et d'angle solide $d\omega$ (voir la figure 6.8). Soit dE_0 une quantité élémentaire d'énergie émise dans $\mathcal{P}$. On note

- dS une section du pinceau $\mathcal{P}$ à une distance l de P,

- dS' la section à la distance $l + dl$,
- d^2V le volume élémentaire compris entre ces deux sections,
- $dE(l)$ l'énergie traversant la section dS,
- $dE(l + dl)$ l'énergie traversant la section dS',

Alors on peut écrire

$$dE(l + dl) = dE(l)(1 - dK)$$

où dK est la proportion d'énergie interceptée par les particules situées dans le volume d^2V. On peut considérer que cette proportion est le quotient du cumul des surfaces des projections des particules situées dans d^2V sur dS par la surface dS. Si l'on note d^2S_i ce cumul, on a donc

$$dK = \frac{d^2S_i}{dS}$$

Si l'on note d^2n le nombre de particules comprises dans le volume d^2V, étant donné que l'on suppose la distribution des particules uniforme, on a

$$\mu = \frac{d^2nV_0}{d^2V}$$

où V_0 est le volume d'une particule, soit

$$V_0 = \frac{4}{3}\pi\rho^3$$

On en déduit

$$d^2n = \frac{3\mu d^2V}{4\pi\rho^3}$$

Maintenant, si l'on note S_0 la surface projetée d'une particule, on a, d'une part

$$S_0 = \pi\rho^2$$

et d'autre part

$$d^2S_i = d^2nS_0$$

On obtient ainsi

$$\begin{aligned} d^2S_i &= \frac{3\mu\pi\rho^2 d^2V}{4\pi\rho^3} \\ &= \frac{3\mu}{4\rho} d^2V \end{aligned}$$

et on en déduit

$$\begin{aligned} dK &= \frac{3\mu}{4\rho}\frac{d^2V}{dS} \\ &= \frac{3\mu}{4\rho} dl \end{aligned}$$

On obtient alors

$$dE(l + dl) = dE(l)(1 - \frac{3\mu}{4\rho} dl)$$

et en intégrant ce résultat

$$dE(l) \quad = \quad dE_0 e^{-\frac{3\mu}{4\rho}l}$$

Ainsi, l'énergie subit une atténuation exponentielle, dont la pente à l'origine est $-\frac{3\mu}{4\rho}$. On retrouve ainsi un résultat classique pour la propagation de la lumière dans la matière, où le coefficient de décroissance est le coefficient d'extinction. Il convient toutefois de noter ici que cette atténuation ne dépend pas de la longueur d'onde de l'énergie correspondante.

6.3.3.2 Dispersion volumique

On peut définir en tout point P de la primitive de lumière la *dispersion volumique*, définie comme étant la quantité d'énergie par unité de volume qui est diffractée en ce point.

Calculer cette dispersion volumique dans le cas général est assez compliqué puisqu'il faudrait par exemple tenir compte des interréflexions entre les particules. On obtiendrait alors une équation intégrale devant être vérifiée par la fonction de dispersion volumique.

Nous faisons ici deux hypothèses primordiales pour déterminer analytiquement cette dispersion volumique :

- on néglige les phénomènes d'interréflexions (ce qui est par exemple justifié si l'albédo des particules est très faible),
- on considère que les particules ne sont éclairées que par la seule source associée à la primitive. Ceci peut se justifier dans le cas où les autres sources ont un éclairement relativement faible par rapport à la source de la primitive.

Grâce à ces deux hypothèses, la dispersion volumique en un point ne dépend alors que de la position du point et des caractéristiques de la source lumineuse associée. Détaillons le calcul de cette dispersion pour le cône et le cylindre.

Cas du cône Un point P de la primitive lumineuse est repéré par ses coordonnées sphériques (θ, φ, r), et on note $I(\theta, \varphi)$ l'intensité de la source dans la direction de P.

Alors la dispersion moyenne est donnée par

$$D(P) \quad = \quad \frac{3\mu}{4\rho}\frac{I(\theta,\varphi)}{r^2}e^{-\frac{3\mu}{4\rho}r}$$

Cas du cylindre Dans ce cas, le point P est reperé par ses coordonnées cylindriques (θ, r, z), et on note $B(\theta, r)$ la radiosité de la source au point P', projeté orthogonal de P sur la base de la source.

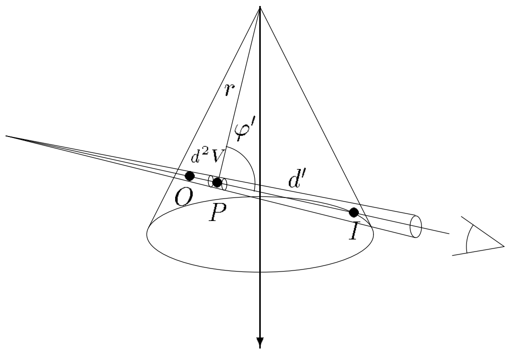
Alors la dispersion moyenne est donnée par

$$D(P) \quad = \quad \frac{3\mu}{4\rho}B(\theta,r)e^{-\frac{3\mu}{4\rho}z}$$

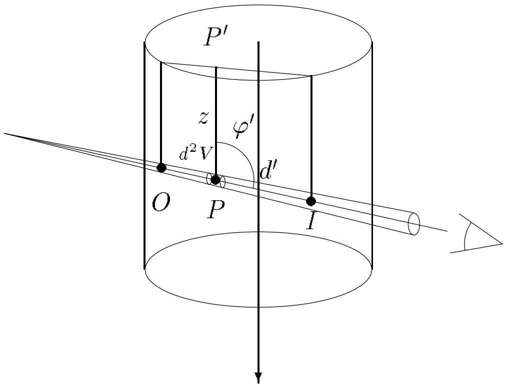
6.3.3.3 Energie propre volumique

Il s'agit maintenant de calculer la contribution en énergie apportée par la traversée d'une primitive de lumière. On considère un pinceau lumineux traversant la primitive, et on note (voir la figure 6.9)

- O le point de sortie de la primitive,
- I le point d'entrée dans la primitive,
- $d'(P)$ la distance IP,



(a) cas du cône



(b) cas du cylindre

FIG. 6.9 – *Energie propre volumique*

- $\varphi'(P)$ l'angle $(\widehat{SP, PO})$ dans le cas du cône, l'angle $(\widehat{P'P, PO})$ dans le cas du cylindre ($\varphi'(P)$ représente l'angle entre la direction du pinceau et la direction d'émission de l'énergie)

Alors, si l'on considère un point P situé dans le pinceau lumineux entouré d'un élément de volume d^2V, la quantité d'énergie d^2E diffractée vers le sommet du pinceau est

$$d^2E = D(P)f(\varphi'(P))d^2V$$

Il faut aussi tenir compte de l'atténuation dûe à la propagation de cette énergie entre le point P et le point de sortie I, et on obtient donc

$$d^2E_a = D(P)f(\varphi'(P))e^{-\frac{3\mu}{4\rho}d'}d^2V$$

Si l'on note dS la surface apparente du volume d^2V vu depuis le sommet du pinceau, et dl la longueur de cet élément de volume, on peut calculer la luminance élémentaire apportée par cet élément de volume par

$$\begin{aligned} dL &= \frac{d^2E_a}{dS} \\ &= D(P)f(\varphi'(P))e^{-\frac{3\mu}{4\rho}d'}dl \end{aligned}$$

Il suffit alors d'intégrer la quantité précédente pour obtenir la contribution en luminance totale lors de la traversée de la primitive. On obtient donc :

$$L = \int_{\vec{OI}} D(P)f(\varphi'(P))e^{-\frac{3\mu}{4\rho}d'}dl$$

On obtient donc, dans le cas du cône,

$$L_{\text{cône}} = \frac{3\mu}{4\rho}\int_{\vec{OI}} \frac{I(\theta,\varphi)}{r^2}f(\varphi'(P))e^{-\frac{3\mu}{4\rho}(r+d')}dl$$

et dans le cas du cylindre

$$L_{\text{cylindre}} = \frac{3\mu}{4\rho} f(\varphi') \int_{\vec{O}I} B(\theta, r) e^{-\frac{3\mu}{4\rho}(z+d')} dl$$

Remarquons que dans le cas du cylindre, l'angle $\varphi'(P)$ est constant et le terme relatif à la fonction de phase peut donc être sorti de l'intégrale.

6.3.3.4 Remarques

Dans les formules précédentes, nous avons considéré que la primitive était entièrement traversée, et que de plus aucun objet ne portait ombre à l'intérieur de la primitive.

Il subsiste néanmoins le problème de l'évaluation de l'intégrale permettant de calculer la luminance. La technique la plus envisageable est une technique d'échantillonnage du segment OI. Le nombre d'échantillons peut alors être choisi adaptativement en fonction de l'énergie globale de la primitive, de la distance moyenne des points du pinceau à la source (si l'on est loin de la source et que l'atténuation est importante, il suffit d'échantillonner faiblement) ou d'autres critères.

Cette technique d'échantillonnage permet aussi de résoudre le problème des ombres portées à l'intérieur de la primitive. Il suffit en chacun des échantillons de relancer un rayon vers la source de la primitive : si le point est à l'ombre, on utilise alors une dispersion volumique nulle, sinon on calcule la dispersion comme précisé ci-dessus.

Enfin, un des problèmes importants à résoudre est le fait que les primitives que nous utilisons sont des primitives unitaires. Si l'on veut utiliser des primitives de formes variées (en appliquant à des primitives des transformations affines par exemple), il ne faut pas oublier de convertir les points non pas dans le repère de la primitive mais dans le repère de l'objet de lumière. Nous avons pour cela défini les primitives de lumière comme des objets texturés, et nous pouvons ainsi retrouver la matrice de passage dans le repère de l'objet de lumière dans la description des textures.

6.4 Densités volumiques

6.4.1 Principe général

Ce travail a été réalisé dans le cadre du stage de DEA de Gilles Mathieu [Mat92]. Le problème est ici de permettre la visualisation de densités volumiques, c'est-à-dire de fonctions de densité de matière. Ainsi, au contraire des primitives de lumière, la densité de particules n'est pas uniforme.

Les densités volumiques constituent dans notre modèle un autre exemple d'objets neutres. Les différences avec les primitives de lumière sont, d'une part, la non-uniformité de la densité, et d'autre part le fait qu'il n'y a pas d'association entre la densité et une source lumineuse particulière : les densités volumiques sont éclairées par les sources définies pour la scène.

L'utilisation de densités volumiques peut se décomposer en trois étapes :

- définir les structures de données pour stocker les densités,
- générer les densités,
- effectuer le rendu des densités.

Dans un souci de simplicité, nous nous sommes limités à la génération de densités utilisant un maillage régulier cartésien tridimensionnel. La structure de données utilisée pour le stockage est alors simplement un tableau tridimensionnel de valeurs, chaque valeur étant la densité de particules dans l'élément de volume correspondant du maillage.

6.4.2 Génération des densités

La génération n'est pas le problème nous intéressant prioritairement, aussi nous serons assez brefs sur ce sujet. On peut se reporter pour plus de détails à [Mat92].

Trois techniques principales sont utilisées :

- la **méthode fractale** repose sur la notion de fonction brownienne fractionnaire [Man89].
- la **méthode spot noise** qui utilise la répétition à intervalles aléatoires d'impulsions (spots) indépendantes [vW91].
- la **méthode de turbulence** qui utilise une fonction pseudo-aléatoire dite de turbulence qui permet de représenter de façon correcte les écoulements de fluides, et donc les nuages ou les brouillards [Per85].

On pourrait obtenir une modélisation bien plus proche de la réalité en considérant la nature physique de ces densités. En particulier, des techniques basées sur l'application de la mécanique des fluides permettraient de simuler avec plus de réalisme la dynamique des densités, en tenant compte des phénomènes de diffusion, de propagation, voire de condensation. De même, la météorologie permettrait de modéliser de façon réaliste les nuages.

6.4.3 Rendu

Les techniques de rendu sont globalement identiques à celles indiquées pour les primitives de lumière. Il existe cependant un certain nombre de différences qui rendent les calculs de rendu encore plus délicats.

La première différence (et non la moindre) est le fait que l'atténuation ne peut plus être calculée par une simple exponentielle tenant compte de la distance à la source émettrice. Il faut alors effectuer un parcours incrémental de la structure (de type Bresenham-3D) et cumuler ainsi les atténuations.

De plus, à l'inverse des primitives de lumière, les densités volumiques n'ont pas de source lumineuse associée. Leur éclairement doit donc être déduit de la définition des sources du modèle. Ainsi, en chaque point d'échantillonnage du pinceau, il faudrait relancer un rayon vers toutes les sources de lumière, et calculer pour chacune de ces sources la dispersion volumique liée à cette source.

Ce calcul des dispersions volumiques est alors extrêmement coûteux en temps de calcul, et la première idée qui vient à l'esprit pour diminuer ce temps de calcul est le précalcul, c'est-à-dire un tracé de rayons inverse calculant les dispersions. Mais il se pose alors un problème important dû à la réutilisation des objets : il faut en effet distinguer la structure qui définit la densité en chaque point du maillage et qui est une structure propre à l'objet, et la structure qui va définir la dispersion en chacune des mailles, qui dépend donc de la position de l'instance de la densité dans la scène.

Nous utilisons pour remédier à ce problème la technique suivante :

- dès la fin de la construction de la scène, on effectue un préparcours de cette scène pour détecter les densités volumiques. On compte alors pour chaque densité le nombre de fois qu'elle est utilisée dans la scène.
- on crée alors pour chaque densité un tableau de structures, la taille du tableau étant le nombre d'utilisations de cette densité. Chacune des structures permet de stocker les dispersions liées à l'une des utilisations de la densité. On conserve également un tableau de correspondance entre le numéro d'objet (déterminé lors de la phase de numérotation, voir le chapitre 2) et l'indice correspondant à cet objet dans le tableau.
- la densité volumique est considérée comme un objet texturé, et nous avons vu dans le chapitre 5 que le numéro d'objet était passé systématiquement en argument des fonctions de texturations. Ceci

permet aux dites fonctions de savoir quel est l'indice dans le tableau (et donc la bonne structure) à utiliser.

Ce principe permet alors de garder un partage de toutes les informations partageables (comme la définition de la densité) et la création de structures spécifiques à l'utilisation de la densité. Notons qu'il convient toujours de calculer les atténuations entre le point sur le pinceau et le point d'entrée par un algorithme incrémental.

6.5 Un essai de parallélisation

6.5.1 Matériel et logiciel

Ainsi que nous l'avons déja indiqué, l'un des nos objectifs initiaux était de réaliser une implantation de notre tracé de rayons sur une machine parallèle de type MIMD à mémoire privée.

Plus précisément, cette machine est constituée d'un ensemble de douze transputers de type `T800`, chaque transputer possédant une mémoire locale de 1 mégaoctet. Chaque transputer possède quatre liens de communication, lesquels sont centralisés sur deux cartes de jonction configurables, afin de pouvoir diversifier les topologies pour le réseau. Les deux cartes de jonction ainsi que l'un des douze transputers sont connectées à une machine hôte `UNIX` par l'intermédiaire d'une carte de communication au format `AT`. Cette carte comprend un transputer `T800`, 2 mégaoctets de mémoire et une interface entre le bus `AT` et l'un des liens du transputer. Les trois autres liens sont alors disponibles pour les connexions vers le réseau.

En ce qui concerne l'environnement de programmation, nous disposons de l'*Inmos C Toolset*, environnement de développement en C pour systèmes à base de transputers. Cet environnement comprend un compilateur, un éditeur de liens, un chargeur permettant l'envoi sur un réseau de transputers d'un ensemble de programmes, et une bibliothèque d'exécution gérant en particulier l'interface avec le système de fichiers de la machine hôte. Il comprend également un débogueur, mais son utilisation est assez délicate.

Notons enfin que les développements sur la parallélisation ont été accomplis pendant le stage de DEA de Hervé Lamure [Lam92]. Ils ne constituent donc qu'une première approche de la parallélisation et les résultats en sont donc forts partiels.

6.5.2 Principes de parallélisation

Deux principes ont en fait été retenus pour paralléliser notre algorithme:

- une parallélisation "brutale", où les processeurs sont utilisés en ferme, et où chaque processeur contient toute la scène et tout le code. Chaque processeur traite un pixel (ou un ensemble de pixels).
- une parallélisation de type "rayon", où les processeurs sont divisés en deux fermes, l'une concernant les calculs d'intersection et l'autre les calculs de rendu.

6.5.3 Première parallélisation

Comme il a été indiqué précédemment, le principe est ici très simple: le réseau comprend un maître et douze processeurs esclaves configurés en ferme de processeurs. Le maître est chargé de construire la base de données (à partir d'un fichier `CASTOR`) et de l'envoyer sur chacun des processeurs.

Une fois cette phase d'initialisation accomplie, le maître envoie des requêtes à chaque processeur. Une requête est simplement une demande de calcul d'une sous-image de l'image de départ. Cette sous-image

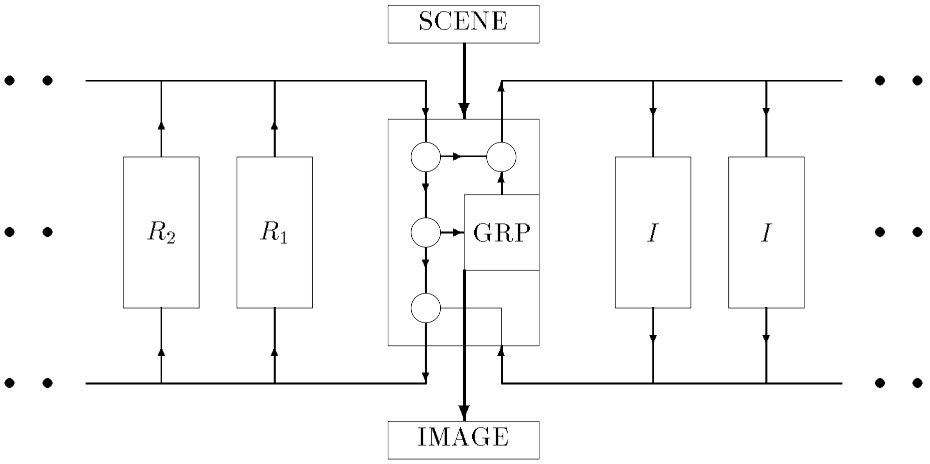


FIG. 6.10 – *Algorithme parallèle (deuxième version)*

peut être réduite à un seul pixel, mais pour des raisons d'efficacité (diminution des communications), il est préférable d'utiliser des tailles de 64 ou 128 pixels par sous-image. Dès qu'un processeur a terminé sa sous-image, il envoie le tableau de pixels au maître, qui peut alors lui renvoyer une nouvelle sous-image s'il en reste à calculer.

Les résultats de cette parallélisation sont acceptables, car le problème d'équilibrage de la charge des processeurs est résolu. Le problème crucial est celui de la place mémoire, car nous ne disposons que de 1 mégaoctet pour stocker le code et les données : la taille des problèmes traitables par cette méthode est ainsi limitée.

6.5.4 Seconde parallélisation

Nous avons basé notre deuxième méthode de parallélisation sur les constatations suivantes :

- le code utilisé par l'intersecteur représente environ 30 % du code pour environ 90 % du temps de calcul. Le code du rendu représente environ 20 % du code pour seulement 10 % du temps.
- avec la réutilisation des objets, la base de données géométriques garde des tailles réduites. La base de données pour le rendu tend elle à devenir de plus en plus importante, surtout si l'on veut intégrer des modèles complexes, nécessitant des tailles de stockage importantes (on peut penser à un échantillonnage d'une vingtaine de longueur d'ondes pour définir une couleur ou une texture).
- dans notre modèle, la partie intersection ne concerne que la géométrie et est fortement indépendante de la partie rendu qui ne concerne que la photométrie.

Ce sont ces constatations qui nous ont poussés à utiliser une parallélisation séparant également la partie intersection de la partie rendu. La figure 6.10 résume l'organisation de notre algorithme parallèle. Les processeurs se décomposent en trois groupes :

- les processeurs d'intersection, qui calculent les intersections entre un rayon et une scène. Ils rendent un résultat sous forme de liste d'intervalles d'intersection. Ils ne connaissent les caractéristiques de rendu (couleurs et textures) que par un numéro. Ces intersecteurs sont banalisés et fonctionnent comme une ferme de processeurs.

- les processeurs de rendu effectuent le rendu. Ils ne connaissent pas la scène, mais ont une table de correspondance entre les numéros d'attributs et les attributs eux-mêmes. Ces processeurs sont identifiés par un numéro, et fonctionnent eux-aussi comme un ferme de processeurs.
- le processeur maître, qui assure un quadruple rôle
  - le chargement de la scène (à partir d'un fichier *CASTOR* par exemple) et sa distribution sur les intersecteurs. Le compactage de la scène est effectué avant la distribution. C'est au cours de cette phase de chargement qu'est établie la correspondance attribut-numéro. La table de correspondance est alors envoyée vers les processeurs de rendu. Notons que le processeur maître est le seul à connaître un certain nombre d'informations comme le type de perspective utilisé, les positions de l'oeil et du point visé, la résolution de l'image.
  - la génération des rayons primaires.
  - une fonction de transit entre la ferme d'intersections et la ferme de rendu.
  - la conversion du résultat final du rendu d'un rayon primaire. Cette conversion est la plupart du temps le stockage dans un fichier image, mais on pourrait aussi calculer des facteurs de forme ou des caractéristiques physiques.

Nous allons maintenant détailler le principe de fonctionnement de notre algorithme.

6.5.4.1 Fonctionnement général

La première phase de l'algorithme concerne le chargement de la base de données sur les différents processeurs (on suppose le code chargé bien entendu). Cette tâche est principalement assignée au processeur maître, les autres processeurs se contentant de consommer et de stocker les informations envoyées par le maître.

Il convient de noter que c'est au cours de cette phase que s'effectue la numérotation des couleurs et des textures : seuls les numéros sont envoyés vers les processeurs d'intersection, alors que les couples (numéro, couleur) sont envoyés vers les processeurs de rendu.

Ensuite, le processeur maître fonctionne en générateur de rayons primaires. Ces rayons primaires sont envoyés vers les intersecteurs. Notons que le générateur construit également une structure d'informations liées à ce rayon, mais cette structure est conservée par le processeur maître : les intersecteurs n'en ont pas besoin.

Un intersecteur reçoit un rayon, et calcule les intersections entre ce rayon et la scène. L'intersecteur reçoit également une structure d'informations associées au calcul d'intersection (voir le chapitre 3). Le résultat est fourni sous forme d'une liste d'intervalles d'intersections. C'est le processeur maître qui reçoit ce résultat pour le transmettre à un processeur de rendu.

L'un des processeurs de rendu reçoit donc l'ensemble composé du rayon, de la structure d'informations associée à ce rayon et de la liste d'intersections trouvées : il peut donc commencer à effectuer le rendu pour ce rayon. Il est alors possible que le rendu ait besoin de relancer de nouveaux rayons (ombres ou secondaires). Ce sont alors les processeurs de rendu qui construisent les nouveaux rayons, ainsi que la structure d'informations pour le rendu, et la structure d'informations pour l'intersecteur. L'ensemble rayon-informations pour l'intersecteur est envoyé vers la ferme d'intersecteurs, via le processeur maître qui assure la liaison. On indique également dans cette structure le numéro d'indentification du processeur de rendu qui a généré ce nouveau rayon. Ceci permet à ce processeur de récupérer tous les rayons secondaires qu'il a générés.

Le génération des rayons secondaires se fait en série, c'est-à-dire que le processeur de rendu attend le retour du résultat du rendu du rayon qu'il a généré avant de poursuivre le rendu du rayon initial. Par contre, il peut accepter d'effectuer le rendu pour d'autres rayons.

Une fois le rendu du rayon primaire terminé, le résultat final est envoyé vers le processeur maître qui effectue le rendu terminal, qui consiste le plus souvent en un stockage des valeurs de rouge, vert et bleu du pixel dans le fichier image. Enfin, s'il reste des rayons primaires à générer, le processeur maître génère un nouveau rayon.

6.5.4.2 Remarques

Le principe de fonctionnement énoncé ci-dessus conduit au résultat suivant : à un instant donné, pour un rayon primaire donné, il n'y a qu'un et un seul rayon issu de ce rayon primaire dans l'ensemble du processus de calcul (intersecteur, maître ou rendu). Ceci est en effet lié à la sérialisation des rayons secondaires.

On peut donc immédiatement affirmer que ce système conduit à de nombreuses pertes d'efficacité dûes à l'inoccupation des processeurs si le nombre de rayon primaires initiaux est inférieur à la somme du nombre de processeurs d'intersection et du nombre de processeurs de rendu. En fait, pour compenser les temps de communication, on a tout intérêt à générer initialement le double de ce nombre. Il est alors nécessaire d'utiliser des tampons de communication entre les divers processeurs.

On peut également remarquer que les processeurs de rendu peuvent traiter plusieurs rayons simultanément : il est donc nécessaire qu'ils disposent d'un ensemble de structures de données locales pour stocker provisoirement ces rayons “en attente”. Lors de la réception du résultat, ils peuvent alors consulter leur table locale pour retrouver le rayon concerné et ainsi générer le rayon suivant : la clé utilisé est simplement le numéro du rayon primaire correspondant, qui identifie de façon unique le rayon en question vu la remarque précedente. Il faut alors aussi stocker dans la structure locale la position dans l'algorithme de rendu afin de bien repartir au bon endroit.

6.5.4.3 Résultats

L'algorithme présenté ci-dessus a été implanté avec un modèle de Phong qui ne génère aucun rayon secondaire. Les résultats obtenus sont donc forts partiels et ne reflètent pas une réalité d'utilisation. De plus, la première implantation semblait pécher par le modèle choisi pour la ferme de processeurs, à savoir un pipeline. Les développements sont en cours pour améliorer cet algorithme parallèle, et les résultats ne seront alors appréciables qu'à ce moment-là.

On peut toutefois constater que cette technique est intéressante par la répartition des données (et du code) entre la partie géométrique et la partie photométrique. Il serait même envisageable de pousser encore cette répartition en utilisant une répartition interne des données entre les processeurs de rendu d'une part, et les processeurs d'intersection d'autre part.

Chapitre 7

Conclusion

7.1 Implantation

7.1.1 Environnement de travail

Une grande attention a été portée à l'environnement de travail, c'est-à-dire l'ensemble des façons d'apporter des informations au programme de calcul d'image.

Afin de faciliter le travail de l'utilisateur, *yart* admet 5 façons de recevoir ces informations :

- un certain nombre de valeurs par défaut,
- un fichier de configuration global pour l'utilisateur, situé dans son répertoire d'accueil (ce fichier se nomme `.yrc`),
- un fichier de configuration situé dans le répertoire où est lancé le programme (qui se nomme aussi `.yrc`,
- le fichier `CASTOR` contenant la description de la scène,
- la ligne de commande du programme.

On utilise également une notion de priorité entre des divers moyens, sachant que la liste donnée ci-dessus est donnée par priorité croissante.

Certaines des informations ne peuvent être apportées que par un nombre restreint de ces moyens, mais il existe autant que possible une version pour chacun de ces moyens.

Les fichiers de configuration ont une forme très simple : ce sont des fichiers texte, contenant un certain nombre de lignes, chaque ligne étant de la forme

```
set <nom_parametre>=<valeur_parametre>
```

Ce système de configuration (largement inspiré de l'exemple de l'éditeur de texte `vi`) est ainsi très simple et très souple.

7.1.2 Le code développé

L'implantation de l'environnement décrit dans cette thèse a été effectuée à l'aide du langage `C` sur des machines `HP/APOLLO` (`DN10000` et `433s`) sous le système `Domain/OS`, version `HP/APOLLO` d'`UNIX`.

Etant donné sa faible liaison avec les aspects particuliers des systèmes d'exploitation, le portage sur d'autres plateformes `UNIX` ne pose pas de problèmes et un portage sur machines `SUN` sous le système `SunOS 4.1.3` a été réalisée sans changer la moindre ligne de code.

Le code proprement dit pour l'environnement complet totalise environ 14000 lignes de code, se répartissant comme suit :

- 2600 lignes pour l'interprète *readCASTOR*, y compris les sources *yacc* et *lex*.
- 1100 lignes de code pour les parties annexes comprenant une bibliothèque de gestion des expressions symboliques, une bibliothèque de manipulation de tables de hachage, ainsi qu'une bibliothèque d'analyse d'expressions parenthésées utilisée pour la gestion des attributs globaux et des attributs des objets.
- 10300 lignes pour le programme *yart* lui-même, que l'on peut décomposer encore en plusieurs parties :
  - 3200 lignes pour l'intersecteur (y compris les opérations booléennes),
  - 1900 lignes pour l'algorithme de rendu (algorithme générique et algorithme de Phong, gestion du fond et des textures),
  - 3200 lignes pour la construction de la scène (parmi lesquelles 800 lignes pour la construction des boîtes englobantes et 400 lignes pour l'interfaçage avec *readCASTOR*),
  - 800 lignes pour le générateur de rayons primaires (avec 7 types de perspective),
  - 1200 lignes de fonctions diverses (entrées-sorties, gestion de l'environnement et des valeurs par défaut, programme principal).

On peut noter que dans cette répartition, la partie dédiée au rendu n'est pas prédominante. Ceci est dû au fait que le rendu est pour le moment très simple, et qu'il n'y a pas de textures implantées, ni de gestion des ombres ou des rayons secondaires. Ce code est donc appelé à croître rapidement.

7.2 Travaux futurs

7.2.1 Extensions possibles

Ainsi que cela a été précisé, notre environnement permet de très nombreuses extensions. Ces extensions sont permises par la très importante modularité du code développé. On peut ainsi ajouter des textures indépendamment des problèmes géométriques, ou réciproquement ajouter des procédures d'intersection pour de nouveaux objets indépendamment du rendu : l'objectif de séparation entre la géométrie et le rendu que nous nous etions fixés est donc atteint.

Les extensions auxquelles nous pensons sont les suivantes :

- ajouter des primitives de type polygône, polyèdre, carreau de surface, terrain.
- ajouter tout un catalogue de textures (bois, turbulence, textures plaquées),
- améliorer encore le fonctionnement des boîtes englobantes généralisées,
- poursuivre l'implantation parallèle de notre algorithme,
- intégrer des modèles de rendu plus complexes, en particulier des modèles spectraux,
- utiliser cet environnement pour des calculs de radiosité,

Nous sommes également très intéressés par l'intégration de phénomènes physiques les plus variés possibles, comme par exemple les phénomènes de polarisation, de fluorescence, de réfraction dispersive [PR92].

Une version encore plus complète de l'algorithme générique de rendu est également à l'étude, de façon à intégrer encore d'autres possiblités, comme par exemple le phénomène d'interférences, difficilement modélisable dans le modèle actuel.

7.2.2 Amélioration de l'environnement

L'environnement *Illumines* peut certainement être encore amélioré et des travaux ont débuté sur ce point. La bibliothèque `castorC` a déjà subi de nombreuses améliorations mais son développement doit se poursuivre afin d'intégrer pleinement les possibilités du tracé de rayons.

Dans le même esprit d'amélioration, nous réfléchissons à la possibilité d'ajouter dynamiquement des textures ou des modèles de rendu. Ces parties ont en effet tendance à devenir de plus en plus importantes (en termes de taille d'exécutable) et la possiblité de ne charger que les parties de code réellement utilisées semble très intéressante. C'est d'ailleurs l'optique retenue dans *RenderMan* où les *shaders* sont compilés en un pseudo-code qui est ensuite interprété par le logiciel. Dans le mesure où nos machines nous permettent le chargement dynamique de code, il semble alors plus simple d'utiliser le compilateur `C` pour compiler ces parties de code, ce qui donne une bien meilleure efficacité.

Enfin, nous avons souvent utilisé des termes ou des concepts des langages orientés objets. L'une des question que l'on peut se poser concerne l'implantation de notre environnement en utilisant l'un de ces langages. Nous avons ainsi des objets bien définis (primitives, opérations booléennes, attributs, rayon, algorithme générique de rendu), auxquels on peut associer des méthodes (calcul de la boîte englobante, calcul d'une intersection, rendu). Ces langages nous éviteraient alors la gestion manuelle de l'instanciation et de l'héritage.

7.2.3 Le traitement de l'aliassage

Un grand absent dans nos travaux est effectivement ce problème de l'aliassage. Des travaux ont démarré parallèlement sur les problèmes de tracé de rayons progressif, qui permettent de résoudre une partie de l'aliassage.

Il semblerait toutefois souhaitable d'intégrer des techniques classiques d'antialiassage, comme un sur-échantillonage local adaptatif. Il suffirait de définir une métrique sur les listes d'intervalles d'intersection, qui pourrait prendre en compte la distance du premier point rencontré, la normale à la surface en ce point, l'énergie incidente en ce point (en provenance des sources ou des autres objets par exemple).

Ce traitement serait possible au sein de l'algorithme de rendu, où la fonction de conversion d'une énergie en couleur de pixel pourrait se charger du calcul d'une structure de données spécifique pour ce problème d'aliassage. Ceci ne semble à première vue pas très compliqué et nous pensons bien intégrer ce traitement dans les versions futures de l'environnement.

7.3 Conclusion

Nous avons présenté un environnement pour le tracé de rayons. Cet environnement est loin d'être complet à l'heure actuelle. Cependant, le soin (et le temps) apporté à la modularité des diverses parties de cet environnement ainsi que la volonté d'assurer la plus grande généricité devraient permettre son extension rapide.

Annexe A

Grammaire yacc de *readCASTOR*

```
%token END_OF_FILE 0
%token SUP DBL_QUOTE SEMI_COLON ARROBAS EXCL DIESE
%token LEFT_PAR RIGHT_PAR SLASH STAR TWO_POINTS
%token COMMA PERCENT EQUAL ELSE DOLLAR PLS MINS
%token STRING ATTRIB IDENT NUMBER

%left MINS PLS
%left SLASH STAR
%left UMINS

%start prog


%%
attribute :
     TWO_POINTS ATTRIB
   ;
qstring :
     DBL_QUOTE STRING DBL_QUOTE
   ;
ident:
     IDENT
   ;
func_name :
     ident
   ;
expression :
     ident
   | NUMBER
   | func_name LEFT_PAR expression RIGHT_PAR
   | func_name LEFT_PAR expression COMMA expression RIGHT_PAR
   | LEFT_PAR expression RIGHT_PAR
   | expression PLS expression
   | expression MINS expression
   | expression STAR expression
```

```
    | expression SLASH expression
    | MINS expression %prec UMINS
    | PLS expression %prec UMINS
    ;
empty_list :
      /* nothing */
    | LEFT_PAR RIGHT_PAR
    ;
enum_of_expressions :
      expression
    | enum_of_expressions COMMA expression
    ;
list_of_expressions :
      LEFT_PAR enum_of_expressions RIGHT_PAR
    ;
enum_of_idents :
      ident
    | enum_of_idents COMMA ident
    ;
list_of_idents :
      LEFT_PAR enum_of_idents RIGHT_PAR
    ;
enum_of_qstrings :
      qstring
    | enum_of_qstrings COMMA qstring
    ;
list_of_qstrings :
      LEFT_PAR enum_of_qstrings RIGHT_PAR
    ;
enum_of_complex_objects :
      complex_object
    | enum_of_complex_objects COMMA complex_object
    ;
list_of_complex_objects :
      LEFT_PAR enum_of_complex_objects RIGHT_PAR
    ;
seq_of_attributes :
      /* nothing */
    | seq_of_attributes attribute
    ;
arg1 :
      empty_list
    |
      list_of_expressions
    ;
arg2 :
      arg1
    | list_of_qstrings
    | qstring /* HORRIBLE POUR MP */
```

```
   ;
def_a_view_parameter :
     PERCENT ident
     arg1 SEMI_COLON
   ;
redefinition:
     ident EQUAL ident SEMI_COLON
   ;
def_a_parameter :
     ident EQUAL expression SEMI_COLON
   ;
complex_object :
     ident
   | EXCL ident
     arg1 arg2 arg1 seq_of_attributes
   | ARROBAS ident
     seq_of_attributes arg1
     arg1 arg1 /* ONLY FOR DEFORMATIONS */
     seq_of_attributes complex_object
   | complex_object ARROBAS ident
     seq_of_attributes arg1
     arg1 arg1 /* ONLY FOR DEFORMATIONS */
     seq_of_attributes
   | DOLLAR ident
     seq_of_attributes
     list_of_complex_objects
     arg1 /* ONLY FOR BLINKS AND VARIANTS */
     seq_of_attributes
   ;
def_an_object :
     ident EQUAL complex_object SEMI_COLON
   ;
definition:
     redefinition
   | def_a_view_parameter
   | def_an_object
   | def_a_parameter
   ;
macro_args :
     empty_list
   | list_of_idents
   | list_of_qstrings
   | LEFT_PAR NUMBER
     RIGHT_PAR
   ;
macrodefinition :
     DIESE ident
     macro_args SEMI_COLON
   ;
```

```
command :
     SUP ident SEMI_COLON
   ;
speech :
     SUP qstring SEMI_COLON
   ;
statement:
     macrodefinition
   | definition
   | command
   | speech
   | error SEMI_COLON
   | error END_OF_FILE
   ;
statements:
     statement
   | statements statement
   ;
prog :
     /* empty file*/
   | statements
   ;
%%
```

Annexe B

Transformations de boîtes : le pire des cas

Lorsque l'on transforme une boîte à côtés parallèles aux axes par une transformation affine contenant une partie rotative, la boîte à côtés parallèles aux axes contenant la boîte transformée est de volume plus grand.

Pour comparer la méthode á englobants classiques à notre méthode à englobants généralisés, nous cherchons la borne supérieure du quotient

$$Q = \frac{\text{volume de la boîte transformée standard}}{\text{volume de la boîte transformée généralisée}}$$

On se restreint dans cette étude aux rotations. L'ensemble des rotations étant un compact, on sait que le quotient ci-dessus admet un maximum. Nous n'avons pas trouvé la valeur de ce maximum ni la rotation correspondante, mais nos donnons tout de même un minorant de ce maximum. Notons d'ailleurs que pour les rotations, les boîtes généralisées conservent leur volume.

On considère donc une boîte $[0, d]^3$, et nous allons nous restreindre aux rotations qui amènent la diagonale principale de la boîte sur l'axe Ox. On considère ensuite le problème projeté sur le plan Oyz. Les sommets de la boîte $(0, 0, 0)$ et (d, d, d) se projettent en $(0, 0)$ et les six autres sommets se projettent en un hexagone régulier de centre $(0, 0)$ et de rayon $R = \sqrt{\frac{2}{3}}d$ (voir la figure B.1). La longueur D de la diagonale est $\sqrt{3}d$. On note alors θ l'angle entre l'axe Oy et le point situé dans le premier sextant du plan (on a $0 \leq \theta < \frac{\pi}{3}$).

Par symétrie, on peut même supposer que θ est inférieur à $\frac{\pi}{6}$. Dans ces conditions, le volume de la boîte englobante est donc

$$V = DlH$$

où l est la largeur (selon l'axe Oy) et H la hauteur (selon l'axe Oz). Or, on a

$$\begin{aligned} D &= \sqrt{3}d \\ l &= R\cos\theta \\ H &= R\sin\left(\theta + \frac{\pi}{3}\right) \end{aligned}$$

soit

$$V = 4\sqrt{3}R^2d^3\cos\theta\sin\left(\theta + \frac{\pi}{3}\right)$$

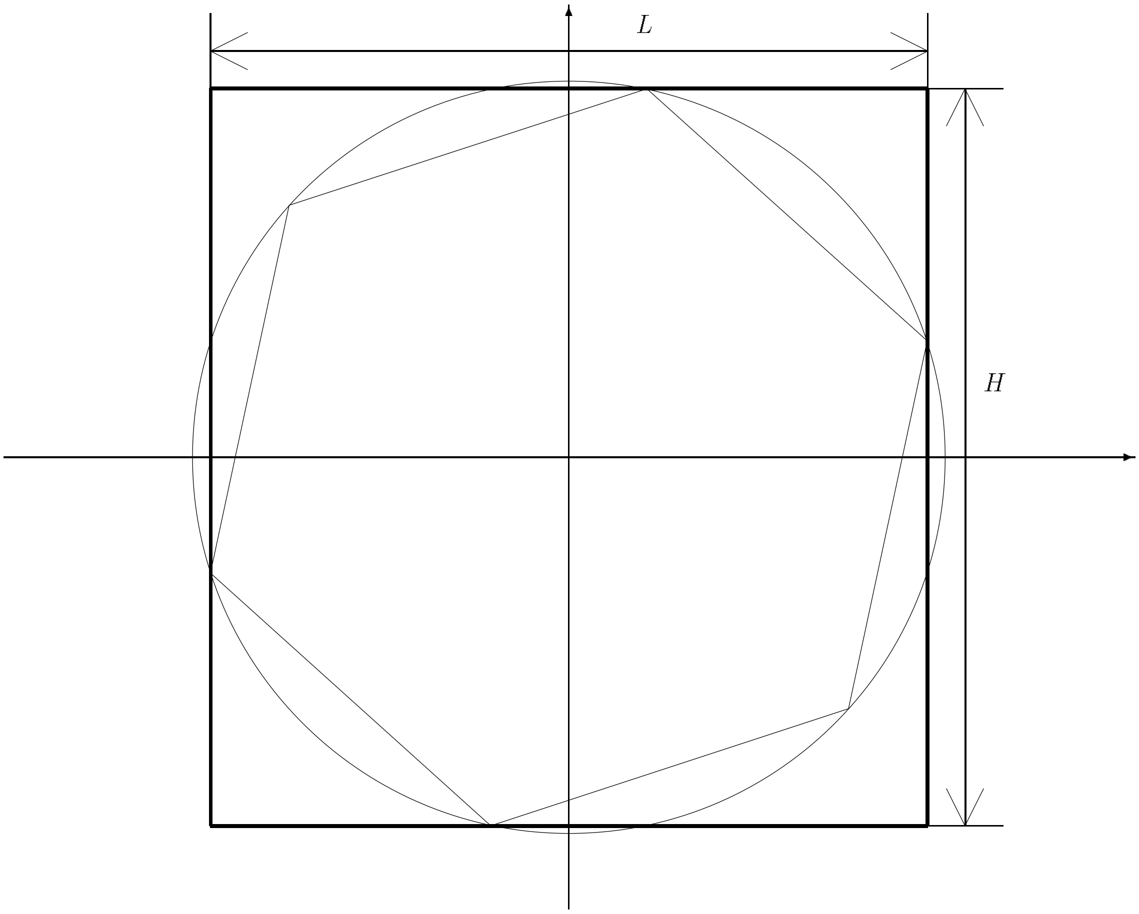


FIG. B.1 – *Projection d'une boîte orthogonalement à la diagonale principale*

soit encore

$$V = \frac{4\sqrt{3}}{3} d^3 \left(\sin\frac{\pi}{3} + \sin\left(2\theta + \frac{\pi}{3}\right)\right)$$

et donc

$$\begin{aligned} Q &= \frac{V}{d^3} \\ &= \frac{4\sqrt{3}}{3}\left(\frac{\sqrt{3}}{2} + \sin\left(2\theta + \frac{\pi}{3}\right)\right) \\ &= 2 + \frac{4\sqrt{3}}{3}\sin\left(2\theta + \frac{\pi}{3}\right) \end{aligned}$$

Il apparaît donc que ce quotient est maximal pour $\theta = \frac{\pi}{12}$, et que la valeur ce ce maximum est alors

$$\begin{aligned} Q &= 2 + \frac{4\sqrt{3}}{3} \\ Q &\approx 4.3094 \end{aligned}$$

On constate ainsi qu'il y a multiplication du volume de la boîte par un facteur supérieur à 4. Or, on peut admettre que la probabilité a priori pour un rayon quelconque d'intersecter une boîte est proportionnelle à son volume : on risque donc d'obtenir 4 fois plus de fausses intersections positives avec cette boîte (intersection détectée avec la boîte alors que le rayon n'intersecte aucun des objets de cette boîte), et le temps de calcul est ainsi augmenté dans des proportions non négligeables.

Il faut noter ici deux points importants :

- nous avons trouvé **une** rotation particulière pour lequel le quotient Q est le plus grand parmi toutes les rotations que nous avons étudiées, mais rien ne nous dit que c'est effectivement un maximum.
- le quotient Q peut prendre des valeurs aussi grandes que l'on veut en autorisant d'autres transformations que les rotations. Par exemple, considérons une transformation constituée de trois transformations successives, la première étant une rotation amenant la diagonale principale d'un cube sur l'axe Ox, la deuxième étant une affinité de rapports $(k_x, 1, 1)$ et la dernière étant la rotation inverse de la première. On constate alors que le volume de la boîte standard est équivalent pour de grandes valeurs de k_x à k_x^3, alors que le volume de la boîte généralisée augmente linéairement. Le quotient Q est ainsi proportionnel à k_x^2.

Bibliographie

[AK87] Arvo (J.) et Kirk (D.). – Fast ray tracing by ray classification. *In : Proceedings SIGGRAPH 84*, pp. 55–64.

[Alb35] Alberti (Leone Battista). – *De pictura*, 1435. Manuscrit en latin.

[Ama84] Amanatides (J.). – Ray tracing with cones. *In : Proceedings SIGGRAPH'84*, pp. 129–135.

[App68] Appel (A.). – Some techniques for shading machine renderings of solids. *SJCC*, 1968, pp. 37,45.

[Arg88] Argence (J.). – *Algorithmes pour le tracé de rayons dans le cadre d'une modélisation par arbre de construction.* – Thèse de PhD, Ecole Nationale Supérieure des Mines de Saint-Etienne, Novembre 1988.

[Arv86] Arvo (J.). – Backward ray tracing. *In : SIGGRAPH 86 Course Notes.*

[BB89] Bouville (C.) et Bouatouch (K.). – *A unified approach to global illumination models.* – Rapport technique, Rennes, CCETT/IRISA, 1989.

[BB92] Beigbeder (M.) et Bourgoin (V.). – New perspectives for image synthesis. *In : Proceedings COMPUGRAPHIC'S 93.* – to be published.

[Bei88] Beigbeder (M.). – *Un développement pour la modélisation et la visualisation en synthèse d'images : CASTOR.* – Thèse de PhD, Ecole Nationale Supérieure des Mines de Saint-Etienne, Avril 1988.

[Ben89] Benouamer (M. O.). – *Evaluation des frontières de solides polyédriques définis à l'aide d'un arbre de construction. Problèmes d'imprécision numérique.* – Rapport technique, Ecole Nationale Supérieure des Mines de Saint-Etienne, 1989. Rapport de DEA.

[Bli77] Blinn (J.F.). – Models of light reflection for computer synthesized pictures. *In : Proceedings SIGGRAPH'77*, pp. 192–198.

[Bou85] Bouville (C.). – Bounding ellipsoids for ray-fractal intersection. *In : Proceedings SIGGRAPH'85*, pp. 45–52.

[Bou90] Bourgoin (V.). – *Polyptyque pour une Trinité cathodique renaissante.* – Rapport technique, Saint-Etienne, Université Jean Monnet, 1990. Rapport de DEA Arts Plastiques.

[BPA88] Bouatouch (K.), Priol (T.) et Arnaldi (B.). – Subdivision spatiale et modélisation CSG pour le lancer de rayons. *In : Proceedings MICAD 88*, pp. 391–407.

[BS63] Beckmann (P.) et Spizzichino (A.). – *The scattering of electromagnetic waves from rough surfaces.* – New York, Pergamon Press, 1963.

[BWJ84] Bronsvoort (W.), Wijk (J. Van) et Jansen (F.). – Two methods for improving the efficiency of ray casting in solid modelling. *Computer Aided Design*, vol. 16, n° 1, Janvier 1984.

[CCWG88] Cohen (M.), Chen (S.), Wallace (J.) et Greenberg (D.). – A progressive refinement approach to fast radiosity image generation. *In : Proceedings SIGGRAPH'88*, pp. 75–84.

[CG85] Cohen (M.) et Greenberg (D.). – The hemi-cube - a radiosity solution for complex environments. *In : Proceedings SIGGRAPH'85*, pp. 31–40.

[Coq84] Coquillart (S.). – *Représentation de paysages et tracé de rayons.* – Thèse de PhD, Ecole Nationale Supérieure des Mines de Saint-Etienne, Unknown 1984.

[CPC84] Cook (R.), Porter (T.) et Carpenter (L.). – Distributed ray tracing. *In : Proceedings SIGGRAPH'84*, pp. 137–145.

[CT82] Cook (R.) et Torrance (K.). – A reflectance model for computer graphics. *In : Proceedings SIGGRAPH'1982*, pp. 307–316.

[Dev90] Devillers (O.). – The macro-region : an efficient space subdivision structure for ray tracing. *In : Proceedings EuroGraphics 1990*, pp. 27–39.

[DS84] Dippe (M.) et Swensen (J.). – An adaptative subdivision algorithm and parallel architecture for realistic image synthesis. *In : Proceedings SIGGRAPH'84*, pp. 149–158.

[ET87] Excoffier (T.) et Tosan (E.). – Une méthode d'optimisation du lancer de rayon. *In : Actes MICAD 87.* pp. 549–563. – Hermès.

[Exc88] Excoffier (T.). – *Construction géométrique de solides et accélération des algorithmes de lancer de rayons.* – Lyon, Thèse de PhD, Université Claude Bernard, Novembre 1988.

[Fer90] Fertey (G.). – *Deux problèmes en synthèse d'images : les sources directionnelles et une interface évoluée.* – Thèse de PhD, Ecole Nationale Supérieure des Mines de Saint-Etienne, Mai 1990.

[FTI86] Fujimoto (A.), Tanaka (T.) et Iwata (K.). – Arts : accelerated ray tracing system. *Computer Graphics and Applications*, vol. 6, n° 4, 1986, pp. 16–26.

[Gla84] Glassner (A.). – Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, vol. 4, n° 10, 1984, pp. 15–22.

[GN71] Goldstein (R.) et Nagel (R.). – 3-d visual simulation. *Simulation*, vol. 16, n° 1, Janvier 1971, pp. 25–31.

[Gou71] Gouraud (H.). – Continuous shading of curved surfaces. *IEEE Transactions on Computers*, vol. C-20, n° 6, Juin 1971, pp. 623–629.

[GTGB84] Goral (C.), Torrance (K.), Greenberg (D.) et Battaile (B.). – Modelling the interaction of light between diffuse surfaces. *In : Proceedings SIGGRAPH'84*, pp. 213–222.

[Hal88] Hall (R.). – *Illumination and color in computer generated imagery.* – New York, Springer Verlag, 1988.

[HTSG91] He (X.), Torrance (K.), Sillion (F.) et Greenberg (D.). – A comprehensive physical model for light reflection. *In : Proceedings SIGGRAPH'91*, pp. 175–186.

[ICG86] Immel (D.), Cohen (M.) et Greenberg (D.). – A radiosity method for non-diffuse environments. *In : Proceedings SIGGRAPH'86*, pp. 133–142.

[KK86] Kay (T.) et Kajiya (J.). – Ray tracing complex scenes. *In : Proceedings SIGGRAPH'86*, pp. 269–277.

[Lam92] Lamure (H.). – *Parallélisations d'un algorithme de tracé de rayon sur un réseau de transputers.* – Rapport technique n° 92-12, Saint-Etienne, Ecole Nationale Supérieure des Mines de Saint-Etienne, 1992. Rapport de DEA.

[Man89] Mandelbrot (B.). – *Les objets fractals.* – Paris, Flammarion, 1989.

[Mat92] Mathieu (G.). – *Modélisation et rendu de densités volumiques.* – Rapport technique n° 92-11, Saint-Etienne, Ecole Nationale Supérieure des Mines de Saint-Etienne, 1992. Rapport de DEA.

[MB89] M.Beigbeder et B.Peroche. – Un système de synthèse d'images 3d : Illumines. *In : Journées UNIX Internationales de Grenoble*, pp. 265–276.

[MCP92] Maillot (J.), Carraro (L.) et Péroche (B.). – Progressive ray tracing. *In : Third Eurographics Workshop on Rendering*, éd. par Chalmers (Alan) et Paddon (Derek), pp. 9–17.

[MG87] Meyer (G.) et Greenberg (D.). – *Color and the computer.* – Boston, Academic Press, 1987.

[Mic87] Michelucci (D.). – *Les représentations par les frontières : quelques constructions, difficultés rencontrées.* – Saint-Etienne, Thèse de PhD, Ecole Nationale Supérieure des Mines de Saint-Etienne, Novembre 1987.

[Mul85] Muller (H.). – *Ray-tracing complex scenes by grid.* – Rapport technique n° 22/85, Karlruhe (RFA), Fakultät für Informatik, 1985.

[Mul86] Muller (H.). – *Image generation by ray tracing in balanced spatial subdivision.* – Rapport technique n° 7/86, Karlruhe (RFA), Fakultät für Informatik, 1986.

[Nie91] Nie (Z.). – *Utilisation des déformations pour la modélistion des solides de forme libre en synthèse d'images.* – Saint-Etienne, Thèse de PhD, Ecole Nationale Supérieure des Mines de Saint-Etienne, Octobre 1991.

[PD88] Puech (C.) et Devillers (O.). – *Une subdivision spatiale en deux étapes pour le tracé de rayon.* – Rapport technique n° 3, Paris, Laboratoire d'Informatique de l'Ecole Normale Supérieure, 1988.

[Per85] Perlin (K.). – An image synthesizer. *In : Proceedings SIGGRAPH'85*, pp. 287–296.

[Pho75] Phong (B.T.). – Illumination for computer generated pictures. *Communications of the ACM*, vol. 18, n° 6, Juin 1975, pp. 311–317.

[PR92] Péroche (B.) et Roelens (M.). – Advanced lighting effects. *In : Eurographics 92 State of the Art Reports*, pp. 1–41.

[RFP90] Roelens (M.), Fertey (G.) et Péroche (B.). – Light sources in a ray tracing environment. *In : Photorealism in Computer Graphics*, pp. 195,210. – Proceedings of the First Eurographics Workshop on Rendering.

[Rot82] Roth (S.). – Ray casting for modelling solids. *Computer Graphics and Image Processing*, vol. 18, 1982, pp. 109–144.

[Smi67] Smith (B.). – Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, vol. AP15, n° 5, 1967, pp. 668–671.

[SP89] Sillion (F.) et Puech (C.). – A general two-pass method integrating specular and diffuse reflection. *In : Proceedings SIGGRAPH'89*, pp. 335–344.

[SSS74] Sutherland (I.), Sproull (R.) et Schumacker (R.). – A characterization of ten hidden-surface algorithms. *Computing Surveys*, vol. 6, n° 1, Janvier 1974, pp. 1–55.

[Str90] Strauss (P.). – An improved lighting model for animation. *IEEE Computer Graphics Applications*, vol. 10, n° 7, 1990, pp. 56–64.

[Til90] Tillier (A.). – *Normalisation des arbres CSG et optimisation des regroupemnents d'objets dans ces arbres – Espaces de couleurs.* – Rapport technique n° 2474.11, Ecole Nationale Supérieure des Mines de Saint-Etienne, 1990. Rapport de DEA.

[TS67] Torrance (K.) et Sparrow (E.). – Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America*, no57, 1967, pp. 1105–1114.

[Ups89] Upstill (S.). – *The RenderMan companion.* – Addison-Wesley, 1989.

[Viv93] Vivian (R.). – *Définition d'une approche adaptative : application à la visualisation en CFAO.* – Thèse de PhD, Université de Metz, Janvier 1993.

[vW91] van Wijk (J.). – Spot noise : texture synthesis for data visualisation. *In : Proceedings SIGGRAPH'91*, pp. 309–318.

[War83] Warm (D.). – Lighting controls for synthetic images. *In : Proceedings SIGGRAPH'83*, pp. 13–21.

[WCG87] Wallace (J.), Cohen (M.) et Greenberg (D.). – A two-pass solution to the rendering equation : a synthesis of ray tracing and radiosity methods. *In : Proceedings SIGGRAPH'87*, pp. 311–320.

[WHG84] Weghorst (H.), Hooper (G.) et Greenberg (D.). – Improved computational methods for ray-tracing. *ACM Transactions on Graphics*, vol. 3, n° 1, Janvier 1984, pp. 52–69.

[Whi80] Whitted (T.). – An improved illumination model for shaded display. *Communications of the ACM*, vol. 23, n° 6, Juin 1980, pp. 343–349.

[WRC88] Ward (G.), Rubinstein (F.) et Clear (R.). – A ray tracing solution for diffuse interreflexion. *In : Proceedings SIGGRAPH'88*, pp. 85–92.

[WS88] Wyvill (G.) et Sharp (P.). – Volume and surface properties in CSG. *In : New Trends in Computer Graphics*. pp. 257–266. – Springer Verlag.

Table des figures

Liste des tableaux

Table des matières