Algorithmique parallèle pour la synthèse d'images

François PELLEGRINI
ENSERB
pelegrin@enserb.u-bordeaux.fr

3 décembre 2001

Ouvrages de référence

- *The Computer Image*,A. Watt et F. Policarpo.Addison Wesley, 1998.
- *Computer Graphics: principles and practice*,Foley, Van Dam, Feiner, et Hugues.Addison Wesley.
- *Infographie*,Schweizer. Presses Polytechniques Romandes.
- *Parallel Processing for Computer Vision and Display*,Dew, Earnshaw, et Heywood.
- *Multiprocessor methods for computer graphics rendering*,Jones et Barthlett.
- *ACM SIGGRAPH, Eurographics, IEEE Transactions on Graphics, ...*

Chapitre 1

Introduction

1.1 Infographie

1.1.1 Domaines d'application

Les domaines d'application de l'imagerie sont extrêmement nombreux : simulations tridimensionnelles (atterrissages, combat, paysages), reconstruction volumique (imagerie médicale)

On a deux classes d'applications, que l'on cherche à faire converger :

- les applications temps-réel (simulateurs, réalité virtuelle, imagerie médicale, ...), dont on souhaite améliorer la qualité ;
- les applications de rendu réaliste (cinéma, ...), que l'on souhaite accélérer.

1.1.2 Définitions

Structure d'une chaîne graphique :

- programme d'application : algorithmes de traitement, données de l'application ;
- logiciel de modélisation : algorithmes de description, scène ;
- logiciel de préparation à la visualisation : algorithmes de préparation à la visualisation, données graphiques ;
- logiciel élémentaire : algorithmes de tracé, listes d'affichage ;
- écran.

Une image est un plan bidimensionnel de pixels.
On divise l'espace en voxels.

1.1.3 Ordres de grandeur

Voici quelques chiffres sur les traitements à effectuer, et les données nécessaires :

- une image de qualité ordinaire est actuellement produite sur un écran comportant entre 512×512 et 1024×1024 points ; pour des applications

telles que le cinéma, on souhaite atteindre des résolutions de l'ordre de 4096×4096, ce qui représente entre 1 et 16 millions de points (sur 3 ou 4 octets) dont il faut calculer la couleur à afficher ;

- les images sont produites à partir de modèles géométriques représentant les objets à afficher. Dans le cas de modèles moléculaires, on peut avoir besoin de cent mille sphères ; pour modéliser un paysage, il faut 1 million de facettes planes (sur $3 \times 3 \times 4$ octets) ;

- pour les opérations de reconstruction tridimensionnelle à partir d'appareils d'imagerie médicale (scanners, IRM), on travaille sur des volumes découpés en $512 \times 512 \times 512$ voxels, dont il faut extraire l'information structurante (frontières entre organes) avant l'affichage ;

- dans le cas de l'algorithme d'élimination de parties cachées par lancer de rayons, on étudie dans une première étape les intersections potentielles entre 1 million de demi-droites (pour un écran de 1024×1024 pixels) et l'ensemble des objets de la scène, chaque rayon primaire pouvant générer une dizaine de rayons secondaires ;

- avec le souhait de disposer de modèles de lumière de plus en plus complexes (car complets), le nombre de traitements à effectuer par pixel augmente de façon encore plus vertigineuse ; on en est (presque) à calculer le parcours des photons dans la scène !

1.2 Utilisation du parallélisme

La synthèse d'images est un domaine d'application évident du parallélisme, du fait de la masse de calculs mis en jeu et de la nécessité d'obtenir rapidement des images de plus en plus complexes. La très grande répétitivité des algorithmes utilisés et la grande quantité de données manipulées laissent espérer de grands gains.

L'objectif final des recherches en imagerie parallèle est la synthèse d'images réalistes en temps réel, qui reste à l'heure actuelle un défi audacieux, puisque l'on n'en est à l'heure actuelle qu'au rendu Z-buffer en temps réel.

1.2.1 Mesures de performance

Idéalement, on cherche toujours à faire varier le temps de traitement comme l'inverse de la puissance de calcul mise en œuvre. En pratique, les gains en performance sont limités par les surcoûts induits par le parallélisme (communication, synchronisation, contrôle).

Soit $d_{C,1}$ la durée de l'algorithme parallèle sur un processeur (et pas celle de l'algorithme séquentiel !), et $d_{C,n}$ la durée d'exécution du même algorithme sur n processeurs. Pour mesurer la performance d'une implémentation parallèle, on considère :

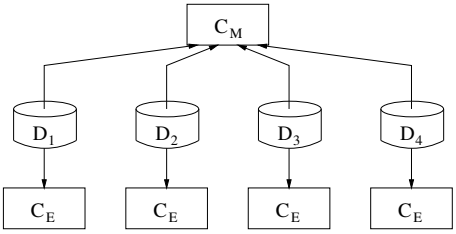
- l'efficacité : $e_{C,n} = \dfrac{d_{C,1}}{d_{C,n} \times n}$.

– l'accélération : $a_{C,n} = \dfrac{d_{C,1}}{d_{C,n}} = n \times e_{C_n}$.

1.2.2 Types de parallélisme

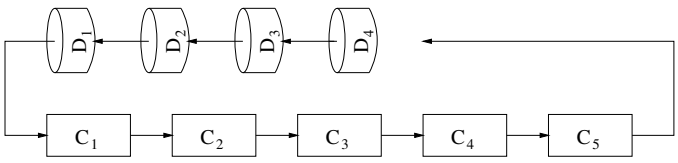
Si la production d'une image implique la réalisation d'une masse de calculs C sur un ensemble de données D, et que l'on dispose pour cela d'un ensemble de processeurs P, on peut envisager plusieurs approches de parallélisation :

– parallélisme de données : on duplique les calculs à effectuer C sur chacun des processeurs, sur lesquels on répartit équitablement des sous-ensembles D_j des données de D.



On organise le traitement des données selon un modèle dit de "ferme de processeurs", où un processus serveur se charge de l'envoi des D_j aux différents processeurs de calcul, et un processus collecteur (souvent le même) agrège les résultats partiels produits.

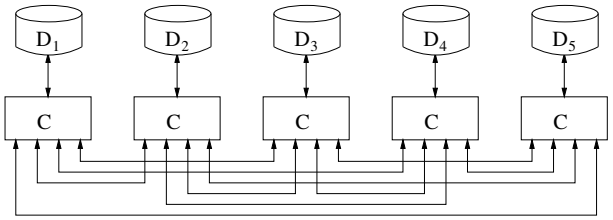
– parallélisme de flux : les calculs C sont fractionnés et organisés en une chaîne séquentielle de sous-calculs C_i, chaque C_i étant affecté à un processeur P_i fixé. De même, les données D sont découpées en et organisées en un flux séquentiel de D_i indépendantes, qui sont présentées en séquence à l'entrée de la chaîne de calculs, chaque pas de temps faisant avancer les D_i d'une étape dans la chaîne de calcul.



On organise les calculs sous forme de pipe-line, dont le débit est borné par le couple C_i/P_i le plus lent, et la D_j la plus grosse. En théorie, si tous les C_i/P_i sont identiques et toutes les D_j de même taille, on obtient en régime stationnaire une accélération égale au nombre de processeurs.

– parallélisme de contrôle : les calculs C sont dupliqués en autant d'exécutables concurrents que de processeurs, mais il existe des dépendances et synchronisations, dues à l'interdépendance des données. On a ici une

vision distribuée du problème, à résoudre parallèlement dans sa globalité.



1.2.3 Granularité

Selon le nombre de processeurs mis en œuvre et le découpage des données réalisé, on définit plusieurs niveaux de granularité :

- gros grain : peu de processeurs (jusqu'à 64), mais souvent de forte puissance ;
- grain fin : un très grand nombre de processeurs (au dessus de 1024), mais souvent de faible puissance. On parle alors de parallélisme massif ;
- grain moyen : entre les deux.

1.2.4 État de l'art

L'avancée de l'état de l'art en imagerie parallèle suit fidèlement la hiérarchie des complexités en temps et en espace des différents problèmes. Ainsi, parmi les problèmes que l'on sait déjà traiter efficacement en parallèle, en temps réel, on trouve :

- tracé de primitives 2D ;
- affichage d'objets 3D représentés sous forme de polygones, avec élimination des parties cachées (Z-buffer) ;
- lancer de rayons.

Ce à quoi on travaille aujourd'hui est :

- la radiosité,

mais on a peu d'espoir de réduire fortement la complexité des calculs mis en œuvre, et il faudra donc attendre l'arrivée de machines suffisament puissantes pour franchir cette étape. En revanche, ce à quoi on pense pour demain, mais qui ne pose pas de réel problème de complexité algorithmique, est :

- la modélisation géométrique parallèle, pour disposer d'une chaîne de traitements complètement parallèle, permettant de manipuler des volumes de données importants.

Chapitre 2

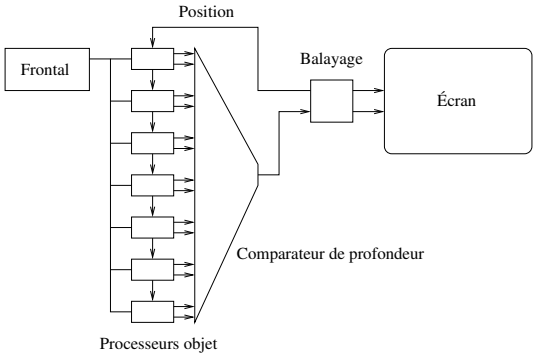
Architecture des machines

Au début des années 80, avec le développement de la technologie VLSI et des réseaux, de nombreuses architectures dédiées à l'imagerie ont été proposées, dont peu ont effectivement été réalisées, et moins encore se sont révélées efficaces en pratique.

Parmi les dispositifs matériels utilisés en imagerie, on trouve :

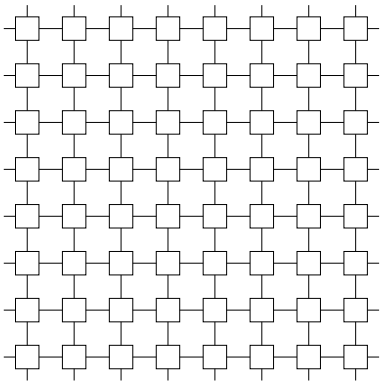
- les processeurs spécialisés : ils sont conçus pour traiter efficacement un problème donné. Leur utilisation couvre toutes les phases de la synthèse d'images :
  - manipulateurs de bitmaps et pixmaps (opérations "*bitblt*" : affichage de pixmaps, masquage, transparence, etc.) ;
  - générateurs de polygones point par point ;
  - générateurs de courbes ou surfaces : quadratiques, Bézier, B-spline ;
  - dispositifs de traitement d'image : transformations géométriques, projections, découpages. Le plus souvent, ces traitements s'effectuent indépendamment sur les données différentes, et l'on peut donc grouper plusieurs processeurs pour les faire travailler en parallèle. Le plus souvent aussi, ces processeurs utilisent à plein les techniques pipe-line ;
- les architectures généralistes : elles sont constituées de machines indépendantes, reliées entre elles par un réseau haut débit. Ces architectures ont un mode de programmation MIMD, et implémentent un parallélisme à gros grain. Pour accélérer les traitements, chaque nœud peut disposer de processeurs spécialisés. Dans ces architectures, le goulet d'étranglement est l'affichage sur la mémoire d'image ;
- les architectures à objets : elles sont constituées d'un ensemble de processeurs spécialisés, synchronisés par le balayage de l'écran, dont chacun est chargé du calcul du rendu d'un très petit nombre d'objets. Chaque processeur reçoit en temps réel les coordonnées du pixel à afficher, et renvoie en temps réel la couleur produite à ce point par l'objet qu'il possède, ainsi que la distance de ce point par rapport à l'observateur. Toutes ces infor-

mations sont collectées par un comparateur de distances, qui renvoie la couleur de l'objet le plus proche, qui est directement affichée à l'écran.



Cette architecture est dédiée à la synthèse d'images temps réel, pour des modèles d'illumination simples, qui ne peuvent prendre en compte les interactions mutuelles entre objets. L'avantage de ce système consiste en la suppression de la mémoire d'image, et du goulet d'étranglement qu'elle représente ;

– les architectures cellulaires : ce sont des assemblages réguliers de processeurs formant un réseau 2D ou 3D (habituellement de type grille à 4 ou 8 voisins), ayant le plus souvent un fonctionnement de type SIMD ou systolique. L'application principale des architectures cellulaires 2D est la parallélisation de traitements au niveau du pixel.



En affectant à chaque processeur les traitements associés à un pixel ou à un petit groupe (carré) de pixels, on obtient des architectures pouvant atteindre le million de processeurs. Le problème principal de ces architectures est le réseau de diffusion qui relie les processeurs, surtout dans le cas d'un fonctionnement MIMD, où les échanges s'effectuent de façon asynchrone.

Chapitre 3

Algorithmes de tracés élémentaires

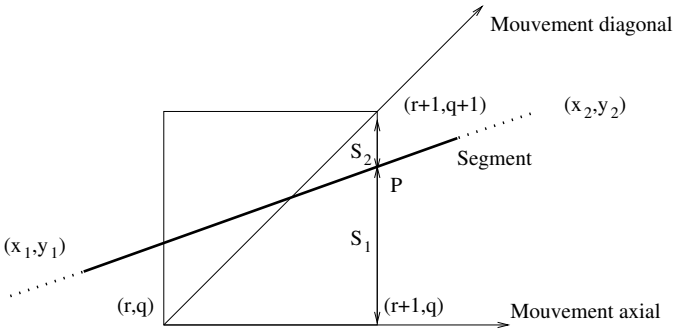
En bout de la chaîne graphique, l'affichage des images calculées s'exprime souvent au moyen de primitives 2D telles que le tracé de droites ou de polygones, qu'il est donc nécessaire de paralléliser afin d'afficher en temps réel des masses de données de plus en plus volumineuses.

3.1 Tracé de droites

De nombreuses applications graphiques utilisent le tracé de droites comme moyen de visualisation. C'est le cas par exemple des modeleurs graphiques interactifs, qui utilisent une représentation "*fil de fer*" des objets de la scène (affichage des arêtes des objets avec une couleur différente pour chacun) afin d'autoriser un déplacement en temps réel à l'intérieur de la scène en construction.

3.1.1 Algorithme séquentiel de Bresenham

L'algorithme de Bresenham est une des méthodes de tracé de droite les plus populaires, car elle ne nécessite que des calculs en arithmétique entière. Son principe consiste, en partant d'une extrémité du segment à tracer, à allumer un par un les points du segment, en choisissant à chaque fois le point le plus proche de la droite idéale du segment.



À titre d'exemple, considérons le tracé dans le premier octant, c'est à dire que la pente de la droite tracée à partir de (x_1, y_1) soit inférieure à 1, et supposons que le dernier mouvement ait conduit au point de coordonnées (r, q), exprimées dans le repère centré en (x_1, y_1) (c'est-à-dire que $0 \leq r \leq (x_2 - x_1)$ et $0 \leq q \leq (y_2 - y_1)$. Les deux candidats possibles pour l'étape suivante sont :

- le point $(r+1, q+1)$, correspondant au mouvement diagonal ;
- le point $(r+1, q)$, correspondant au mouvement axial.

Le choix entre ces deux possibilités est basé sur la minimisation de l'erreur commise, c'est-à-dire la comparaison des longueurs des segments S_1 et S_2. Si $S_1 < S_2$, on aura un déplacement axial, sinon un mouvement diagonal. Le point P de coordonnées $(r+1, y)$ appartient au segment à tracer, donc en posant :

$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{b}{a} \ ,$$

on obtient :

$$y = \frac{b}{a}(r+1) \ ,$$

et donc :

$$\begin{array}{rclcl} S_1 & = & y - q & = & \frac{b}{a}(r+1) - q \\ S_2 & = & (q+1) - y & = & (q+1) - \frac{b}{a}(r+1) \ . \end{array}$$

Il en découle que :

$$S_1 < S_2 \Longleftrightarrow 2(br - aq) + 2b - a < 0 \ ,$$

puisque dans cet octant on a toujours $a > 0$. Définissons l'erreur relative au point (r, q) comme :

$$S(r, q) = 2(br - aq) + 2b - a \ ,$$

avec $S(0, 0) = 2b - a$. Dans le cas d'un mouvement axial, le point suivant est $(r+1, q)$, et :

$$\begin{array}{rcl} S(r+1, q) & = & 2(b(r+1) - aq) + 2b - a \\ & = & S(r, q) + 2b \ . \end{array}$$

Dans le cas d'un mouvement diagonal, le point suivant est $(r+1, q+1)$, et :

$$S(r+1, q+1) \quad = \quad S(r, q) + 2b - 2a \ .$$

On a donc, dans le premier octant, l'algorithme suivant :

```
bresenham (x1, y1, x2, y2)
{
  plot (x1, y1);                          /* Allume un pixel */
  a = x2 - x1;
  b = y2 - y1;
  S = 2 * b - a;
  for (x = x1 + 1, y = y1; x <= x2; x ++) {
    if (S < 0)                            /* Mouvement axial */
      S = S + 2 * b;
```

```
    else {                                  /* Mouvement diagonal */
      y ++;
      S = S + 2 * (b - a);
    }
    plot (x, y);
  }
}
```

En précalculant les constantes $2b$ et $2(b-a)$, on arrive à un coût d'itération d'une ou deux additions, et d'un test de signe. On le voit, l'algorithme de Bresenham est extrêmement efficace, mais est intrinsèquement séquentiel, puisque le calcul d'un point nécessite le calcul préalable de tous les précédents.

3.1.2 Algorithme parallèle par dichotomie

Un moyen simple de paralléliser le tracé de droites consiste à découper le segment à tracer en deux, récursivement, et à distribuer chacun des sous-segments obtenus à des processeurs de tracé qui fonctionneront alors en parallèle.

Cette solution, à grain moyen, ne fonctionne cependant que pour des segments suffisamment longs, pour lesquels le coût de calcul induit par le découpage est amorti par le parallélisme, et pour un nombre suffisamment grand de segments

3.1.3 Algorithme parallèle de Leprêtre

L'algorithme proposé par Leprêtre est utilisable sur des architectures SIMD où chaque processeur est en charge des calculs liés à un pixel, et détermine si le segment de droite qui lui est proposé passe par lui ou non.

Au départ, les paramètres liés au segment sont calculés, puis diffusés à l'ensemble du réseau. L'équation implicite de la droite passant par les points (x_1, y_1) et (x_2, y_2) s'écrit :

$$D \;=\; \{(x,y) \,/\, ay - bx + bx_1 - ay_1 = 0\} \;\;,$$

avec $a = (x_2 - x_1)$ et $b = (y_2 - y_1)$. L'erreur due à l'approximation lorsqu'on échantillonne est donc, au pixel (i,j) :

$$E(i,j) = aj - bi + bx_1 - ay_1 \;\;.$$

Elle doit être comparée à un seuil d'affichage S, qui permet de n'allumer que les pixels les plus proches de la droite idéale. Pour calculer cette valeur, considérons la variation d'erreur entre deux pixels voisins ; elle est de b sur l'axe horizontal, et de a sur l'axe vertical. Comme on veut que le segment soit centré sur la droite idéale, on définit donc :

$$S = \frac{\min\left(\mathrm{abs}(a), \mathrm{abs}(b)\right)}{2} \;\;,$$

pour n'allumer que les pixels ne s'éloignant pas de la droite idéale de la moitié de la plus petite valeur d'erreur conduisant à l'allumage d'un pixel erroné.

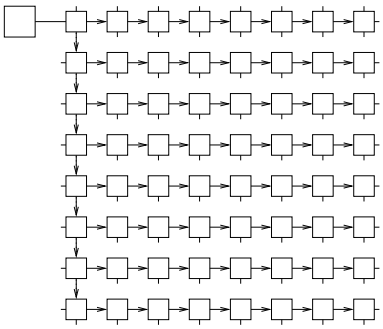
De plus, on ne doit conserver de la droite que la partie comprise entre les deux extrémités du segment, c'est-à-dire les pixels tels que $x_1 \leq i \leq x_2$, et

$y_1 \leq j \leq y_2$. Si les coefficients et le seuil de tolérance sont diffusés à tous les pixels, il ne reste donc plus à ceux-ci qu'à vérifier la condition :

```
lepretre (x1, y1, x2, y2, a, b, S)
{
  if ((i >= x1) && (i <= x2) && (j >= y1) && (j <= y2) &&
      (abs (E(i,j)) <= S))
    plot (i, j);
}
```

Le coût réel de cet algorithme, qui s'exécute en temps constant sur chacun des processeurs, est en fait le coût de la diffusion des paramètres de tracé à l'ensemble des processeurs.

La diffusion en temps constant est irréaliste, du fait du trop grand nombre de connexions point-à-point mis en jeu, à moins de disposer d'un réseau de diffusion spécifique de type bus. On peut cependant imaginer une diffusion de type multi pipe-line par lignes ou par colonnes (en prenant la plus petite dimension), en considérant le cas d'une architecture cellulaire systolique.



Dans le cas d'une architecture systolique, on peut optimiser l'algorithme, en calculant la valeur d'erreur $E(i,j)$ de manière incrémentale sur la direction de progression du pipe-line.

3.2 Tracé de polygones

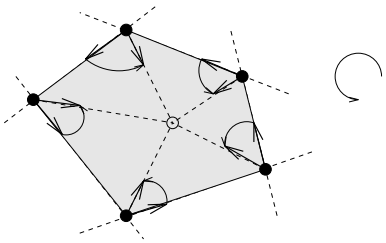
Le tracé de polygones est une autre primitive essentielle du tracé 2D, d'une complexité supérieure d'un ordre de grandeur à celle du tracé de droites (on affiche des surfaces au lieu de lignes), et qui demande donc à être parallélisée de manière efficace.

Tous les algorithmes massivement parallèles décrits dans cette section sont basés sur des architectures cellulaires.

3.2.1 Algorithme parallèle basé sur l'équation

Son principe dérive de celui de l'algorithme de tracé de droites de Leprêtre : chaque pixel est géré par un processeur, qui doit déterminer si ce pixel appartient au polygone ou non, en déterminant de quel côté il se trouve par rapport

à chacun des segments de droite qui lui sont passés.

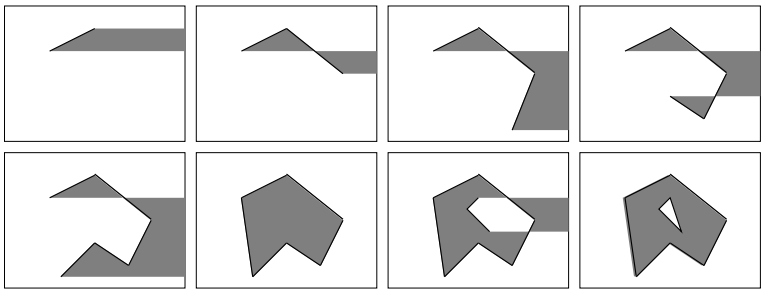


Cet algorithme ne traite pas correctement les polygones concaves, qui doivent nécessiter un pré-traitement coûteux de détection et de scission en polygones convexes, qui est de plus intrinsèquement séquentiel. Il ne peut non plus traiter simplement les polygones troués.

De plus, les messages qui transitent sur l'architecture cellulaire sont de taille variable et génèrent des coûts de traitement très différents. Cet algorithme, de granularité moyenne, est en fait mal adapté à l'architecture cellulaire, qui ne peut exécuter efficacement que des algorithmes réguliers à grain fin.

3.2.2 Algorithme parallèle basé sur la complémentation ("*edge filling*")

C'est un algorithme multi-pipeline à grain fin, dans lequel toutes les commandes traversent le réseau dans le même sens. Il consiste à faire circuler sur le pipe-line tous les segments qui composent le polygone, à charge pour chacune des cellules de déterminer sa position relative par rapport à chacun des segments, et de complémenter tous les pixels situés "après" les segments, par rapport à la direction de propagation des données dans le pipe-line.



Le dernier segment d'un polygone doit être identifié par un drapeau, pour que les cellules réalisent l'affichage effectif des pixels complémentés et interprètent les segments suivants comme appartenant à un nouveau polygone.

Cet algorithme traite naturellement les polygones concaves et troués. Seuls les segments parallèles à la direction des pipe-lines nécessitent une détection et un traitement spécifiques.

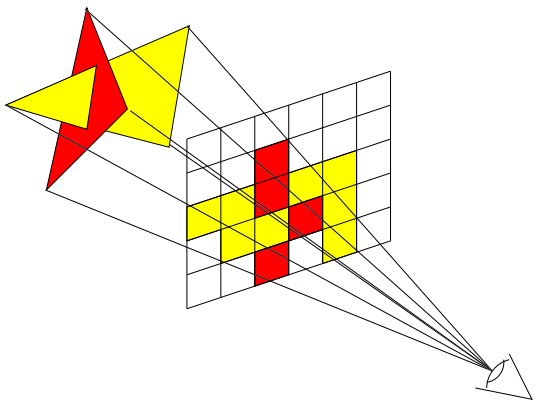
Chapitre 4

Élimination des parties cachées par Z-buffer

4.1 Algorithme séquentiel

L'algorithme de rendu par tampon de profondeur ("Z-buffer") est basé sur l'utilisation d'une mémoire de profondeur (d'où le nom) servant à conserver la distance par rapport à l'observateur des pixels affichés à l'écran. Il consiste à projeter sur le plan de l'écran les polygones constituant la scène, en calculant pour chaque pixel du polygone en cours d'affichage la distance à laquelle celui-ci se trouve de l'observateur. Si cette distance est supérieure à celle contenue dans la mémoire de profondeur, c'est que le polygone courant est masqué par un autre, et le pixel n'est pas affiché. Sinon, c'est que ce pixel du polygone courant est actuellement le plus proche de l'observateur ; le pixel est affiché, et sa profondeur est conservée dans le tampon de profondeur. L'algorithme peut donc s'exprimer de la façon suivante :

```
zbuf (polygones, image)
{
  pour (tous_les_pixels (image)) {
    image.coul[pixel] = couleur_du_fond;
    image.tamp[pixel] = infini;
  }
  pour (tous les polygones) {
    projeter_sur_ecran (polygone);
    pour (tous_les_pixels (polygone_projete)) {
      si (profondeur (point (pixel)) < image.tamp (pixel)) {
        image.coul[pixel] = couleur_polygone (point (pixel));
        image.tamp[pixel] = profondeur       (point (pixel));
      }
    }
  }
}
```



Cet algorithme diffère de l'algorithme dit "du peintre" en ce que le tri de la profondeur ne se fait pas au niveau des polygones (avec les risques d'incohérences que cela implique), mais au niveau des pixels.

4.2 Estimation de puissance

Une estimation a été donnée sur la puissance nécessaire à l'affichage en une seconde de 100000 polygones à quatre côtés, se projetant chacun sur une surface de 10×10 pixels, et rendus par la méthode de Z-buffer avec ombrage de Gouraud :

- transfert des données : il faut manipuler 100000 polygones × 4 sommets × 6 valeurs flottantes (coordonnées et normales) × 4 octets, ce qui donne un débit de 10 Mo/s ;
- calcul de transformation des sommets : 11 Mflop/s ;
- calcul de transformation des normales : 9,5 Mflop/s ;
- calcul d'éclairage : 11 Mflop/s ;
- découpe des bords : 2,5 Mflop/s ;
- projections : 4,5 Mflop/s ;
- calcul de fenêtres et clôtures : 3,5 Mflop/s.

Le total de ces opérations conduit à un débit de 10 Mo/s et 46,5 Mflop/s pour 100000 polygones par seconde, ce qui est une image de petite taille et un rendu loin du temps réel.

4.3 Algorithmes parallèles de Li et Miguet

Li et Miguet ont proposé plusieurs algorithmes parallèles à gros grain de l'algorithme de Z-buffer sur réseau reconfigurable de Transputers. Les Transputers sont des processeurs développés par Inmos, disposant chacun de quatre liens bidirectionnels de communication. La machine utilisée par Li et Miguet est un T.Node, dont les 32 Transputers sont connectés par l'intermédiaire d'un réseau

cross-bar programmable qui permet de réaliser toutes les topologies de degré inférieur ou égal à quatre.

Les deux algorithmes proposés par Li et Miguet sont chacun basés sur la parallélisation d'une des deux boucles de l'algorithme séquentiel.

4.3.1 Parallélisation de la boucle externe

Première approche

Le premier algorithme est basé sur la parallélisation de la boucle externe. Dans une première phase, les polygones sont répartis sur les processeurs, qui exécutent chacun en parallèle l'algorithme séquentiel sur les polygones qui leur ont été attribués. Dans une deuxième phase, ces images partielles sont successivement fusionnées deux à deux, en comparant les profondeurs des pixels qui les composent et en gardant les pixels associés aux plus petites profondeurs :

```
zbuf_fusion (imr, imp)
{
  pour (tous_les_pixels (imr)) {
    si (imp.tamp[pixel] < imr.tamp[pixel]) {
      imr.coul[pixel] = imp.coul[pixel];
      imr.tamp[pixel] = imp.tamp[pixel];
    }
  }
}
```

Pour réaliser efficacement la deuxième phase, il faut utiliser une topologie arborescente. Comme le degré de chaque processeur est d'au plus quatre, les trois sortes d'arbres utilisables sont les arbres unaires (c'est-à-dire que la topologie est une chaîne), les arbres binaires, et les arbres ternaires.

Dans tous les cas cependant, lors de la fusion, seuls les nœuds d'un même niveau sont actifs, ce qui constitue une perte significative de parallélisme. De plus, chaque processeur doit d'abord calculer l'intégralité de son image avant d'entamer la phase de communication, ce qui nécessite le stockage en mémoire de toute la matrice d'image, alors que les Transputers disposent d'une mémoire de très petite taille (1 Mo à l'époque). Ce problème est encore aggravé par la nécessité de recevoir les images partielles des fils pour effectuer localement leur fusion.

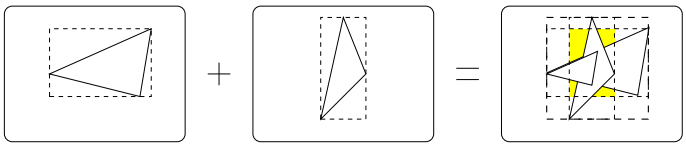
Algorithme pipe-liné

Pour remédier à cela, on décompose l'image partielle que chaque processeur doit calculer en bandes horizontales, ce qui permet de pipe-liner les opérations de calcul et de fusion des bandes sur le réseau arborescent, et que tous les processeurs soient actifs en même temps.

À chaque étape, chaque processeur effectue en parallèle l'envoi de la bande (B-1) qu'il a fusionné à l'étape précédente, le calcul local de sa bande B, et démarre la réception des bandes B provenant de ses fils, puis fusionne les bandes B reçues avec sa bande B locale.

Pour éviter de traiter tous les polygones lors du calcul de chaque bande, ceux-ci ne sont projetés qu'une seule fois dans le plan de l'image. On détermine ainsi leur rectangle englobant, qui permet de les trier dans l'axe vertical afin que seuls les polygones apparaissant dans la bande en cours de calcul subissent l'intégralité des traitements.

Afin de réduire les surcoûts résultants de la communication des bandes et de leur fusion, il est possible de ne transmettre que la partie de la bande constituant le rectangle englobant de tous les polygones projetés. On réduit ainsi le temps de communication, mais également le temps de fusion, puisque celle-ci n'a effectivement lieu que sur l'intersection du rectangle reçu et du rectangle conservé localement. Le nouveau rectangle englobant à transmettre au père est celui qui englobe tous les rectangles reçus des fils et celui calculé localement.



Performance

La taille des bandes a une influence sur le temps d'exécution de l'algorithme : si elles sont trop grandes, le temps d'inititialisation du pipe-line est trop important, et si elles sont trop petites, le surcoût de contrôle devient trop important.

L'arité de l'arbre joue également. Si l'on utilise un arbre ternaire, on minimise la profondeur du pipe-line ainsi que le temps de communication, puisque les réceptions des trois bandes issues des trois fils se feront en parallèle sur les liens de communication. Cependant, on introduit un déséquilibre de calcul, puisque les nœuds internes de l'arbre auront trois fusions à réaliser, alors que les feuilles, qui sont les plus nombreuses avec l'arbre ternaire, seront en attente de synchronisation. Cela est également vrai, dans une moindre mesure, pour l'arbre binaire.

Li et Miguet constatent que, pour des images de grande taille, le pipe-line linéaire est le plus efficace, car seul un processeur n'a pas de fusion à réaliser, et le recouvrement des communications par le calcul est assuré (une seule communication à réaliser).

4.3.2 Parallélisation de la boucle interne

Première approche

Les résultats produits par l'algorithme précédent montrent que la parallélisation de la boucle externe induit des surcoûts de fusion qui sont loin d'être négligeables, et que la topologie linéaire est bien adaptée à l'équilibrage de la charge.

Le deuxième algorithme proposé par Li et Miguet est basé sur la parallélisation de la boucle interne. Chaque processeur est responsable du calcul d'une

partie de l'image. Comme on ne peut pas, pour des raisons d'encombrement, dupliquer l'ensemble des polygones sur chacun des processeurs, ceux-ci n'en possèderont qu'une partie, qu'ils s'échangeront pour mener à bien l'intégralité des calculs.

Considérons que l'image est divisée en bandes de même taille. Dans une première étape, chaque processeur réalise la projection des polygones qu'il possède, et peut ainsi déterminer les processeurs qui en auront effectivement besoin, à qui il faudra les diffuser (on aura donc duplication uniquement des polygones à cheval entre plusieurs bandes). Il faut donc pouvoir réaliser efficacement la diffusion par chaque processeur d'un message différent vers chaque autre, c'est-à-dire effectuer une multi-diffusion ("*multi-scattering*", ou ATAP), qui s'effectue bien sur les topologies annulaires.

Multi-diffusion sur l'anneau

Supposons tout d'abord que l'anneau de processeurs est orienté, c'est-à-dire que le processeur P_i ne recevra de messages que du processeur P_{i-1} et n'émettra que vers le processeur P_{i+1} (modulo N). Supposons que le processeur P_i désire envoyer un ensemble de messages $m_{i,j}$ aux processeurs P_j, avec $j \neq i$.

Dans une première étape, il enverra le message $m_{i,i-1}$ à son successeur P_{i+1}, et recevra le message $m_{i-1,i-2}$ de son prédécesseur P_{i-1}. À l'étape suivante, il fusionnera son propre message $m_{i,i-2}$ au message $m_{i-1,i-2}$ pour créer un message unique M_{i-2}, qu'il émettra vers P_{i+1} en même temps qu'il reçevra M_{i-3} de P_{i-1}. À l'étape k, il fusionnera son propre message $m_{i,i+N-k}$ au message $m_{i-1,i+N-k}$ pour créer un message unique M_{i+N-k}, qu'il émettra vers P_{i+1} en même temps qu'il reçevra $M_{i+N-k-1}$ de P_{i-1}.

Après $N-1$ étapes semblables, P_i recevra un unique message M_i contenant tous les messages m_{*},i provenant des autres processeurs. On a donc l'algorithme suivant :

```
multiscatter_unidir ()
{
  M[(i + (N - 1)) % N] = {};
  pour k allant de 1 a (N-1) {
    j = (i + N - k) % N;
    M[j] += m[i,j];
    envoyer  (i + 1, M[j]);
    recevoir (i - 1, M[j - 1]);
  }
}
```

On suppose que la transmission d'un message de taille L entre deux processeurs prend un temps $t = \beta + L\tau$, où β est le temps d'initialisation de la communication, et τ est le temps de communication élémentaire pour un octet. Si tous les messages $m_{i,j}$ sont de longueur L, le message M_j envoyé par P_i au temps t est de longueur Lt, et le temps total de la multidiffusion est :

$$T_1 = \sum_{t=1}^{N-1} (\beta + Lt\tau) = \frac{(N-1)(2\beta + NL\tau)}{2} \quad .$$

Si l'anneau est bidirectionnel, les messages peuvent circuler dans les deux directions à la fois. Dans ce cas, deux messages M_i et Q_i partiront de P_i dans les

deux directions opposées, pour utiliser au mieux la bande passante disponible, et seulement $\lceil \frac{N}{2} \rceil$ étapes de communication seront nécessaires :

```
multiscatter_bidir ()
{
  M[(i + N/2) % N] = {};
  Q[(i + N/2) % N] = {};
  pour k allant de 1 a N/2 {
    j1 = (i + (N/2) - k + 1) % N;
    j2 = (i - (N/2) + k - 1) % N;
    M[j1] += m[i,j1];
    Q[j2] += m[i,j2];
    envoyer  (i + 1, M[j1]);
    envoyer  (i - 1, Q[j2]);
    recevoir (i - 1, M[j1 - 1]);
    recevoir (i + 1, Q[j2 + 1]);
  }
}
```

Si N est pair, le même message est envoyé dans les deux directions à la fois lors de la première étape. Pour éviter cela, on peut effectuer une étape unidirectionnelle, suivie de $\frac{N-1}{2}$ étapes bidirectionnelles, ou envoyer le premier message en deux moitiés dans chacune des deux directions.

Pour les machines à base de Transputers, il est plus lent d'utiliser un lien dans les deux directions que dans une seule. Cependant, il est quand-même plus efficace de réaliser une communication bidirectionnelle que deux communications unidirectionnelles. Les valeurs de β et τ dans ces deux cas sont données ci-dessous, en micro-secondes.

	Unidirectionnel	Bidirectionnel
β	$25,8$	$36,7$
τ	$1,1$	$1,6$

Calcul de la taille des bandes

Afin d'avoir une répartition homogène des bandes sur l'image, on se sert du rectangle englobant la projection de chaque polygone sur l'écran, et on ne fait calculer par les processeurs que la partie d'image située entre les deux extrémités verticales, qui est équitablement répartie entre les processeurs. On réalise le calcul des bornes par réductions min et max sur les coordonnées extrêmes déterminées localement.

4.4 Architectures spécialisées

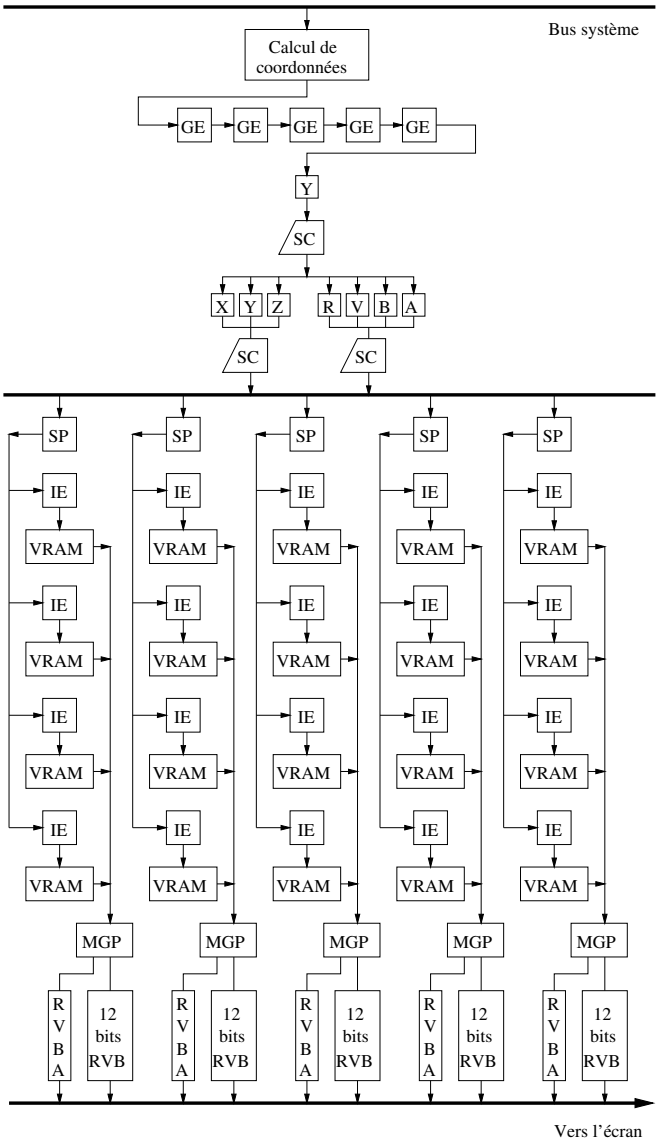
Plusieurs architectures spécialisées dans le Z-buffer ont été définies. Pour l'essentiel, elles sont similaires à celle proposée par [1] et reprise dans [5, pages 186–187]. Cette dernière est constituée :

- d'un système géométrique chargé de tous les calculs en flottant, qui comporte, entre autres, cinq calculateurs géométriques (GE, pour « *Geometry Engine* ») ;

– d'un système de découpage chargé de traiter les polygones. Le processeur de polygones les transforme en trapèzes et prépare tous les coefficients permettant un suivi incrémental des arêtes. Celles-ci sont traitées par les processeurs d'arêtes, qui sont de simples interpolateurs linéaires (SC, pour « *Slope Calculator* »). Les résultats, à savoir des listes de segments de droite horizontaux, sont traités par les processeurs de ligne (SP, pour « *Span Processor* ») ;

– d'un système de calcul d'image, composé de vingt dispositifs de tracé d'image (IE, pour « *Image Engine* ») permettant chacun de gérer un vingtième de mémoire d'affichage vidéo (VRAM, pour « *Video RAM* ») ;

– d'un système d'affichage, composé de cinq processeurs graphiques (MGP, pour « *Multimode Graphics Processor* »), qui transforme en informations RVB les valeurs contenues dans la mémoire vidéo.


Bus système
Calcul de coordonnées
GE
GE
GE
GE
GE
Y
SC
X
Y
Z
R
V
B
A
SC
SC
SP
IE
VRAM
MGP
R V B A
12 bits RVB
Vers l'écran

Chapitre 5

Modèles d'illumination

5.1 Généralités

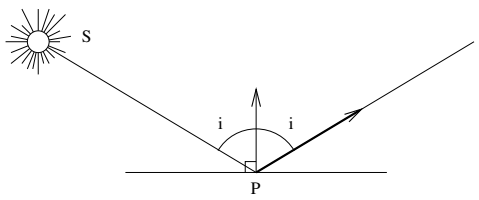
Les modèles d'illumination définissent la manière dont on modélise la lumière à l'intérieur de la scène, et dont celle-ci interagit avec les objets. Ils définissent ainsi la qualité de rendu de l'image à l'écran : plus un modèle est complexe et plus il permet de prendre en compte de phénomènes physiques liés à la lumière, et donc plus le réalisme des images sera grand.

Les calculs d'illumination représentent à l'heure actuelle la majorité du coût de calcul d'une image, et n'ont cessé de croître du fait de l'introduction de modèles d'illumination de plus en plus sophistiqués. De plus, ces nouveaux modèles nécessitent des volumes de données extrêmement importants, qui dépassent les capacités des machines séquentielles classiques. Ce domaine est donc propice à l'utilisation du parallélisme.

5.1.1 Spécularité et diffusivité

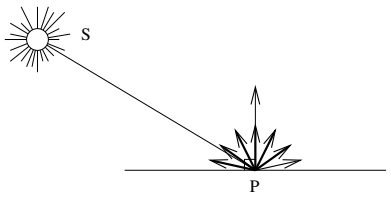
L'interaction entre la lumière et la matière au niveau macroscopique a été très étudiée par les physiciens des XVIII^{e} et XIX^{e} siècles. Ceux-ci ont dégagé deux types de comportement lorsqu'un faisceau lumineux se réfléchit sur une surface :

- réflexion spéculaire : il y a réémission unidirectionnelle de la lumière avec un angle égal à l'angle d'incidence du rayon lumineux (loi de Descartes–Snell), et conservation de l'énergie. Ce type de réflexion est observable sur les objets finement polis, tels les miroirs ("*speculum*" en latin, d'où le nom).

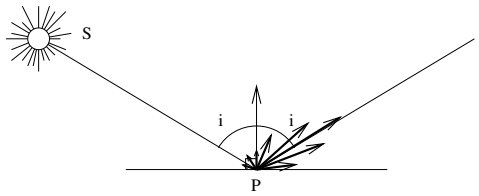


- réflexion diffuse : il y a réémission omnidirectionnelle isotrope de la lumière incidente, avec perte d'énergie. Les surfaces provoquant des réfle-

xions diffuses, dites "lambertiennes", sont en général mates et rugueuses. Le comportement diffus a été modélisé par les physiciens comme le piégeage de la lumière à l'intérieur des irrégularités de la surface, la lumière ne ressortant qu'après de multiples réflexions avec les parois des cavités, dont l'orientation aléatoire donne son caractère isotrope à la réflexion.

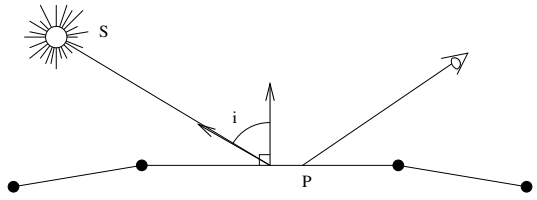


Dans la réalité, on observe souvent un mélange de ces deux types de comportements, qui donne lieu à une réflexion mixte.



5.1.2 Modèle de Bouknight [Bouk70]

Le modèle de Bouknight, adapté aux modélisations polygonales, consiste à associer une couleur à chaque facette des objets. L'intensité lumineuse reçue d'une source lumineuse ponctuelle par la facette est calculée comme le produit de l'intensité de la source par la valeur du produit scalaire entre la normale à la facette et la direction de la lumière incidente, si celui-ci est positif (sinon, la facette est cachée). La somme des intensités reçues est alors pondérée par la couleur de la facette, pour donner l'aspect final de la facette à l'écran.

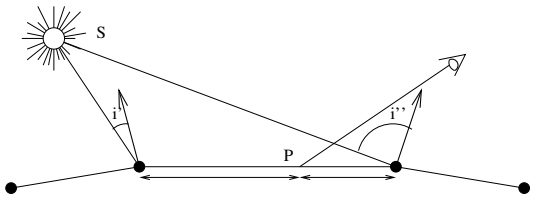


Cet algorithme est extrêmement simple, et adapté aux faibles puissances de calcul disponibles à l'époque. Cependant, il souffre de nombreuses faiblesses, telles que la création de discontinuités très visibles entre facettes voisines servant à l'approximation de surfaces courbes. Qui plus est, ces discontinuités sont renforcées par l'apparition de bandes de Mach au voisinage de la frontière (ce sont des aberrations liées à la physiologie de l'œil humain).

5.1.3 Modèle de Gouraud [Gour71]

Pour réduire les discontinuités aux interfaces entre facettes, l'algorithme de Gouraud interpole en chaque pixel de la facette les intensités lumiseuses calculées

aux sommets de la facette, la normale prise en un sommet étant la moyenne des normales des facettes partageant ce sommet.



Cet algorithme est plus coûteux que celui de Bouknight, en ce qu'il nécessite un calcul d'intensité par pixel de la facette, au lieu d'un calcul par facette. Cependant, l'interpolation entre les couleurs calculées aux sommets de la facette peut se faire de manière incrémentale, à l'image du calcul incrémental de profondeur qui est réalisé en chaque pixel d'une facette projetée dans le cadre de l'algorithme de Z-buffer. De fait, de nombreuses implémentations optimisées de l'algorithme de Z-buffer utilisent le modèle de Gouraud.

5.1.4 Modèle de Phong [Phon75]

Le modèle de Phong a été le premier à prendre en compte la texture des objets, et non plus simplement leur couleur propre. En chaque pixel de la facette, on calcule l'intensité lumineuse réémise par chaque source lumineuse S dans la direction de vision au moyen de la formule

$$I_{O,P} = (R_p \cos(i) + w(i) \cos^n(s))\, I_s \ ,$$

avec :

$I_{O,P}$: intensité réfléchie dans la direction de vision O au point P ;

I_s : intensité lumineuse émise par la source ponctuelle S dans la direction de P ;

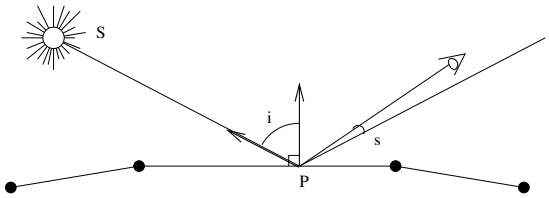
i : angle d'incidence de la lumière, avec $0 \leq i \leq \frac{\pi}{2}$;

s : angle entre le rayon réfléchi et la droite de vue ;

R_p : coefficient de réflexion de l'objet, avec $0 \leq R_p \leq 1$ pour chacune des longueurs d'onde considérées ;

$w(i)$: coefficient de réflexion spéculaire, avec $0 \leq w(i) \leq 1$;

n : coefficient de brillance, avec $1 \leq n \leq 10$. 1 correspond à un objet terne, et 10 à un objet luisant.



Le premier terme modélise l'illumination diffuse, telle que l'exprime Bouknight, alors que le deuxième terme représente l'illumination spéculaire due au positionnement de l'observateur dans la direction de réflexion privilégiée des sources lumineuses.

Ce modèle est totalement empirique, mais donne en pratique de bons résultats, avec un rendu "vinyle" des objets. En effet, le vinyle est une résine transparente, à laquelle on ajoute des pigments. La résine donne un aspect fortement spéculaire, et l'orientation aléatoire des particules de pigment dans la matrice crée la composante diffuse.

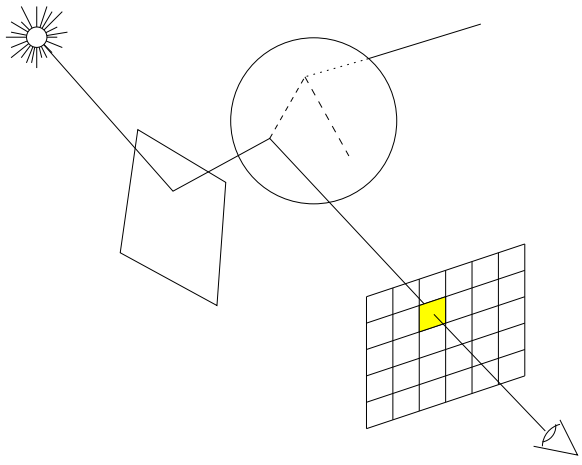
5.2 Algorithme de lancer de rayons

L'algorithme de lancer de rayons, dû à Whitted et Turner [Whit80], constitue la première tentative de prise en compte des réflexions et réfractions multiples.

5.2.1 Principe

Dans la réalité, une infinité de rayons lumineux part de chaque source lumineuse, et se reflète et/ou se réfracte dans les objets de la scène, avant d'atteindre l'œil de l'observateur. En pratique, cela est impossible à simuler.

L'idée originale de l'algorithme de lancer de rayons consiste à considérer l'inverse du trajet effectivement réalisé par les rayons lumineux qui arrivent jusqu'à l'œil de l'observateur. Pour cela, on engendre pour chaque pixel de l'image un rayon partant de l'observateur et intersectant le plan de l'image en ce pixel, on calcule pour l'objet le plus proche rencontré par le rayon les sous-rayons réfléchi et réfracté induisant le rayon initial, et on recommence récursivement tant que tous les rayons fils n'ont pas atteint une source lumineuse ou ne sont pas sortis de la scène.



Le modèle d'illumination initial de Whitted dérive de celui de Phong. L'illumination $I_{O,P}$ d'un point P touché par un rayon arrivant de O est décomposée en :

$$I_{O,P} = I_{Pd} + I_{Ps,O} + I_{Pt,O} \ ,$$

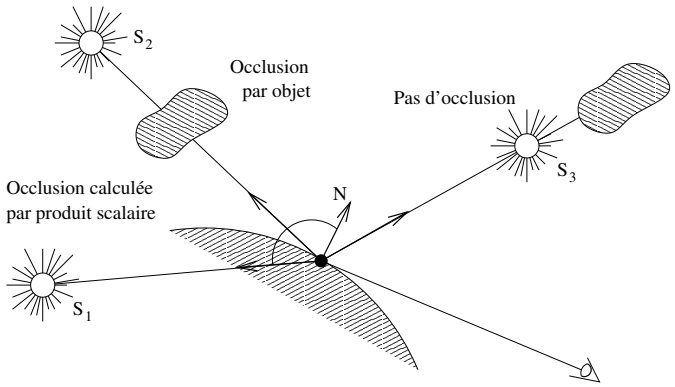
avec :

I_{Pd} : illumination diffuse. Cette illumination correspond à l'émission dans la direction du rayon de la fraction diffuse de l'illumination ambiante de la scène :

$$I_{Pd} = R_p I_{dt} \ ,$$

où I_{dt} est l'illumination ambiante, c'est-à-dire une approximation de l'intensité résultant des multiples réflexions diffuses subies par la lumière issue des sources primaires ;

$I_{Ps,O}$: illumination due aux sources ponctuelles. Cette contribution est calculée pour chaque source ponctuelle au moyen de l'équation de Phong. Elle correspond donc aux contributions diffuses et spéculaires produites par les sources lumineuses au point P. Pour savoir si la lumière d'une source S arrive effectivement au point P, on peut tirer un rayon du point P vers S. Si tous les objets intersectants sont situés après S, la source est visible, et sa contribution peut être ajoutée. Sinon, un objet fait obstruction, et la source ne contribue pas à l'illumination de P ;



$I_{Pt,O}$: illumination spéculaire dans les directions privilégiées de Descartes–Snell, par réflexion et éventuellement réfraction. On a :

$$I_{Pt,O} = w(i) I_{P',P} + t(i) I_{P'',P} \ ,$$

où P' est le point intersecté par le rayon réfléchi, P'' est le point intersecté par le rayon réfracté s'il existe, et $t(i)$ est le coefficient de réfraction.

Le fait de prendre en compte les directions privilégiées de Descartes–Snell dans le modèle revient, avec le modèle d'illumination de Phong, à considérer les points origines des rayons réfléchi et réfracté comme des sources secondaires, qui auront le même impact sur l'illumination que les sources primaires. En effet :

- pour les sources primaires, on aura une grande valeur de I_s, mais fortement atténuée par le $\cos^n(s)$;
- pour les points origine, on aura une plus faible valeur d'illumination, mais par nature $\cos^n(s) = 1$, donc la contribution sera intégralement restituée, multipliée par $w(i)$. Ainsi, pour des objets fortement spéculaires ($w(i) \approx 1$), on aura une image presque parfaite des objets touchés par les rayons réfléchis, c'est-à-dire un effet de miroir.

L'algorithme de lancer de rayons, sous sa forme initiale, est donc le suivant :

```
lancer (obs, objets, sources, image)
{
  pour (tous_les_pixels (image)) {
    rayon = ray_init (obs, pixel);
    image[pixel] = illumination (rayon, objets, sources);
  }
}

illumination (rayon, objets, sources) {
  point = intersection_liste (rayon, objets);
  si (point == RIEN)
    renvoie (Idt);
  si (est_source (objet (point)))
    renvoie (Is);

  I = Rp * Idt;
  si (w(i) > 0) {
    rayon_reflechi = ray_refl (point, rayon);
    I += w(i) * illumination (rayon_reflechi, objets, sources);
  }
  si (t(i) > 0) {
    rayon_refracte = ray_refr (point, rayon);
    I += t(i) * illumination (rayon_refracte, objets, sources);
  }
  pour (toutes les sources) {
    rayon_source = ray_sour (point, source);
    point_ombre  = intersection_liste (rayon_source, objets);
    si (distance (point_ombre) >= distance (source))
      I += Is * (w(i) * cos^n(s) + Rp * cos (i));
  }
  renvoie (I);
}

intersection_liste (rayon, objets)
{
  point = RIEN;
  pour (tous les objets) {
    point_t = intersection_objet (rayon, objet);
    si ((distance (point_t) < distance (point)) &&
        (distance (point_t) > 0.0))
      point = point_t;
  }
  renvoie (point);
}
```

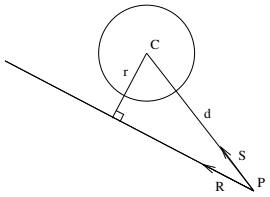
L'algorithme de lancer de rayons est un algorithme simple, qui traite de nombreux phénomènes : parties cachées, ombrages, réflexions et réfractions multiples, aspects spéculaires. Il est cependant très coûteux en temps, car il dépend de la position de l'observateur (il faut recalculer entièrement l'illumination à

chaque déplacement de celui-ci). De plus, il ne prend pas en compte les aspects diffus, qui constituent la majorité des cas réels. Pour limiter les effets de crénelage ("*aliasing*") sur les ombres et les contours, il est possible de réaliser un sur-échantillonnage stochastique, en lançant plus de rayons à chaque fois, mais cette solution est très coûteuse.

5.2.2 Optimisation des calculs d'intersections

Les calculs d'intersection représentent entre 75 à 90 pour cent du coût global de l'algorithme. Pour réduire ce coût, plusieurs solutions ont été proposées.

Une première idée consiste à utiliser une hiérarchie de volumes englobants : les objets sont encapsulés à l'intérieur de formes géométriques simples (sphères, parallelépipèdes), qui donnent lieu à un test d'intersection simplifié. La recherche de l'intersection entre un rayon et les objets et/ou sous-volumes contenus dans un volume englobant n'aura lieu que si le test d'intersection avec celui-ci est positif. Dans le cas d'une sphère, on a intersection potentielle avec les objets contenus si $d \sin(R, S) < r$, ce qui revient à calculer un produit vectoriel.



Une autre approche consiste à décomposer l'espace de la scène en volumes élémentaires appelés voxels, dont chacun possède la liste des objets qu'il contient. L'intersection d'un rayon avec les objets de la scène s'effectue par une progression incrémentale du rayon à travers les voxels, en partant du voxel contenant l'origine du rayon et en s'arrêtant au premier voxel contenant une intersection valide.

Les voxels sont obtenus par décomposition rectilinéaire de l'espace, qui peut être soit adaptative, avec une décomposition récursive (structure d'octree) donnant des voxels de taille différente, soit uniforme, avec des voxels de tailles semblables propices à des progressions incrémentales (de type Bresenham), avec possibilité d'utiliser plusieurs niveaux de grilles (approche dite "multi-grille") dans les zones à forte densité d'objets.

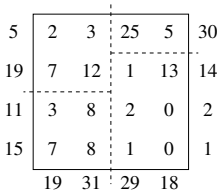
5.2.3 Parallélisation par découpage des traitements

Puisque le traitement de chaque rayon primaire est indépendant des autres, l'idée consiste à répartir le traitement des rayons primaires et de leurs descendants sur les processeurs de l'architecture parallèle. Dans la pratique, cela consiste à découper l'image en zones attribuées à chaque processeur.

Le premier problème soulevé est celui de l'équilibrage de charge. Du fait de l'hétérogénéité du placement des objets dans la scène, un découpage uniforme de l'image en zones de même taille conduit à de grands déséquilibres. Pour remédier à cela, plusieurs techniques ont été proposées :

– un sous-échantillonnage préalable de l'image, en tirant un rayon par zone

de 8×8 pixels, pour évaluer le coût des zones, puis répartition par découpages binaires successifs ("*Binary Space Partitioning*", ou BSP).



Cette technique statique peut cependant conduire à des déséquilibres significatifs ;

- la décomposition de l'image en zones de petite taille, de l'ordre de 8×8 pixels, qui sont dynamiquement distribuées aux processus oisifs.

Le deuxième problème soulevé, qui est encore plus critique que le premier, est que l'ensemble de la scène doit être dupliqué sur chacun des processeurs, ce qui est irréalisable pour des scènes de grande taille.

Une solution intéressante a été proposée par Badouel [Bado90]. Elle repose sur le principe de cohérence spatiale de l'image, qui fait que les chemins suivis par les rayons des arborescences issues de deux rayons primaires voisins sont en moyenne relativement proches. La base de données constituant la scène est distribuée par pages sur l'ensemble des processeurs, chacun de ceux-ci conservant une grande partie de sa mémoire centrale sous forme de cache d'objets.

Lors du calcul local de la progression des rayons à travers les voxels de la scène, le processus référence les objets au moyen d'un numéro unique. Si l'objet à intersecter n'est pas présent localement, une requête est envoyée à son processus propriétaire, qui en retour renvoie les données de l'objet, qui sont placées dans le cache d'objets local, qui est géré avec une politique LRU. Comme la base de données est accédée uniquement en lecture, on n'a pas de problèmes de cohérence, ni de mise à jour.

Cette solution offre une très bonne efficacité (de l'ordre de 90 pour cent sur 64 processeurs), du fait de la cohérence des zones élémentaires calculées par les processeurs (de 32×32 pixels), ce qui permet une stabilisation très rapide de l'algorithme. L'aspect dynamique de l'algorithme est cependant indispensable, du fait de l'imprévisibilité des demandes. L'équilibrage de la charge est également géré de façon dynamique : lorsqu'un processus a fini de traiter sa zone, il demande du travail aux autres en envoyant une requête sur un anneau logique, et un autre processus en retard peut alors lui céder une partie de sa zone. Sinon, si aucun autre travail ne lui est proposé, il termine.

5.2.4 Parallélisation par découpage de l'espace

L'espace de la scène est découpé en régions (en s'appuyant sur le découpage en voxels préexistant), qui sont affectées aux processeurs. La progression des rayons s'effectue de façon incrémentale, avec envois de messages entre processus voisins si les rayons traversent les frontières entre régions.

Le principal problème de cette approche est l'équilibrage de la charge, qui est totalement imprévisible. Les découpages statiques, basés sur les nombres d'objets, fonctionnent mal, car il s'agit plutôt d'évaluer la surface efficace présentée

aux rayons par les objets. Il faut donc mettre en œuvre des politiques dynamiques d'équilibrage, basées sur des migrations d'objets entre régions voisines.

5.3 Algorithme de radiosité

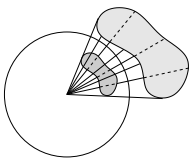
L'algorithme de radiosité est le premier modèle d'illumination s'appuyant sur des bases physiques solides, bien qu'il n'ait originellement traité que les surfaces purement diffuses. Il est originellement issu de l'ingéniérie thermique [Hott54], et son adaptation à l'imagerie est due, de façon indépendante, à Goral et *al.* [GoToGrBa84], et Nishita et Nakamae [NiNa85].

5.3.1 Grandeurs physiques

Les définitions de cette section sont extraites de [Subr95].

Angle solide

Un angle solide est en trois dimensions ce qu'un angle est en deux dimensions. Il mesure la portion d'espace occupée par un objet vu d'une position donnée. L'angle solide sous-tendu par la sphère est de 4π stéradians, noté sr.



L'angle solide sous-tendu par une surface élémentaire ΔS d'aire ΔA située à une distance r du point de référence peut être approximé par la projection sur le plan tangent à la sphère de rayon unité et perpendiculaire à la droite passant par le centre de la sphère et le centre de la surface élémentaire. On a donc :

$$\Delta\omega = \frac{\Delta A \cos\theta}{r^2} \ ,$$

où θ est l'angle polaire, ou angle zénital, c'est-à-dire l'angle entre une direction et la verticale.

Flux lumineux ou puissance radiante

La lumière se propage le long de rayons lumineux sous forme de distribution spectrale d'énergie. Comme on désire obtenir des images à des instants précis, on raisonne plutôt en terme de flux lumineux, que l'on nomme aussi puissance radiante ou flux radiant, noté P, qui est une énergie par unité de temps, c'est-à-dire exprimée en Watts.

La grandeur photométrique correspondante est la puissance lumineuse, exprimée en lumens (lm).

Radiance

L'œil n'est pas sensible à l'énergie, mais à une grandeur qui est directionnelle et qui ne dépend ni de la surface de la source, ni de sa distance à l'œil, et qui est la radiance, notée L.

La radiance L en un point est une énergie partant (ou arrivant) dans une direction, par unité de temps, par unité d'aire projetée (c'est-à-dire par unité d'aire perpendiculaire à la direction d'émission), par unité d'angle solide. Elle s'exprime en $Wm^{-2}sr^{-1}$.

5.3.2 Principe

Dans la réalité, les scènes que nous observons correspondent à des états d'équilibre énergétique : pendant chaque unité élémentaire de temps, une quantité quasiment constante d'énergie lumineuse est émise par chaque source primaire et atteint les objets de la scène, qui se comportent alors à leur tour comme des sources secondaires en réémettant une fraction de l'énergie reçue. En régime stationnaire, l'énergie reçue et émise par chaque surface élémentaire ("*patch*") d'un objet est constante, puisqu'on est en état d'équilibre énergétique.

L'idée originale de l'algorithme de radiosité consiste à calculer les énergies émises à l'équilibre par chaque surface élémentaire. Une fois connue l'illumination globale de la scène, on pourra alors déterminer la fraction de cette énergie reçue par l'observateur.

Pour mener à bien l'algorithme, il est donc nécessaire de déterminer, pour chaque surface élémentaire, la visibilité relative de toutes les autres, afin de connaitre la fraction d'énergie provenant de la surface émettrice qui parviendra aux autres. Dans les faits, on décompose les surfaces géométriques de la scène en "patches", surfaces élémentaires sur lesquelles on suppose que l'énergie surfacique émise par unité de temps (radiosité) est constante. Ayant défini :

B_i : radiosité propre du patch i (non-nulle si c'est une source lumineuse),
E_i : radiosité totale du patch i,
P_i : fraction de lumière incidente réémise par le patch i,
F_{ij} : fraction de lumière quittant le patch j qui atteint le patch i, qui représente la visibilité de i par j (ces coefficients sont appelés "facteurs de forme"),

la loi de conservation de l'énergie appliquée aux patches donne l'ensemble d'équations :

$$E_i = B_i + P_i \sum_{j=1}^{N} F_{ij} Ej \ .$$

C'est ce système qu'il faut résoudre.

5.3.3 Radiosité globale

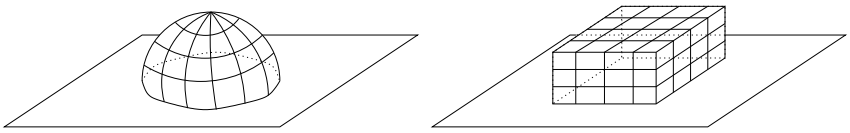
Cette classe de méthodes est celle décrite dans le premier article traitant du sujet [Cohe85]. Elle consiste à résoudre globalement le système d'équations en trouvant la solution du système linéaire :

$$\begin{bmatrix} 1-P_1F_{11} & -P_1F_{12} & \cdots & -P_1F_{1N} \\ -P_2F_{21} & 1-P_2F_{22} & \cdots & -P_2F_{2N} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ -P_NF_{N1} & & \cdots & 1-P_NF_{NN} \end{bmatrix} \cdot \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ \vdots \\ E_N \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ \vdots \\ B_N \end{bmatrix}$$

Une solution exacte du système nécessiterait une inversion de la matrice, ce qui est irréalisable en pratique, vu la taille de celle-ci. En fait, ce système est résolu par des méthodes itératives, telles celle de Gauss-Siedel, la solution se trouvant alors dans le vecteur E.

Le principal problème des méthodes globales est le calcul des facteurs de forme, pour lequel plusieurs approches ont été proposées:

- le calcul par hémicubes revient à approximer l'échantillonnage de la demi-sphère de visibilité d'un patch par l'échantillonnage d'un demi-cube (d'où le nom) centré sur le patch.



  On détermine la visibilité des autres patches par le patch considéré en effectuant leur projection par Z-buffer sur chacune des faces de l'hémi-cube. Le coefficient de visibilité d'un patch donné est calculé en faisant la somme du nombre de pixels de l'hémi-cube sur lesquels il apparait, en réalisant éventuellement une pondération pour limiter les aberrations dues à l'approximation de la demi-sphère par l'hémicube.

  Cette méthode, qui revient à projeter l'ensemble des autres patches vers le patch considéré, est donc de type "*gather*". Elle est coûteuse, puisqu'on doit effectuer cinq Z-buffer par hémicube, pour chaque patch, chaque Z-buffer nécessitant la projection de l'ensemble des facettes sur les parois des hémi-cubes. Ce calcul est donc quadratique par rapport au nombre de facettes de l'image.

- le calcul par l'échantillonage stochastique consise à tirer à partir de chaque patch un grand nombre de rayons, et à compter le nombre de fois que ceux-ci touchent les autres patches. On échantillonne ainsi les facteurs de forme des patches visibles à partir du patch émetteur.

  Cette méthode est conceptuellement simple, et traite bien les patches proches, pour lesquels les calculs de parcours des rayons seront de plus rapides. Cependant, elle ne permet pas d'échantillonner avec précision les patches lointains et de petite taille, ce qui peut provoquer des différences de luminosité entre patches voisins lorsque la scène comporte des sources quasi-ponctuelles lointaines et très lumineuses, et selon qu'elles ont été cibles d'un rayon ou non.

5.3.4 Radiosité stochastique progressive

L'inconvénient majeur des méthodes globales de radiosité est qu'elles nécessitent le calcul de tous les facteurs de forme pour toutes les surfaces composant la scène, ce qui est extrêmement coûteux en espace et en temps (complexités en $O(N^2)$, au moins en espace), d'autant plus que l'on risque de calculer les contributions de patches d'influence très faible sur le résultat global, que l'on aurait donc pu négliger.

L'idée sous-tendant les méthodes progressives est de ne calculer les échanges énergétiques que pour les patches qui contribuent de façon importante au rendu de la scène. Pour cela, on applique un algorithme incrémental (d'ou le terme de "progressive"), qui ne considère à chaque instant que le patch le plus brillant de la scène. Il consiste à émettre l'énergie possédée par ce patch, stockée dans un "compteur de radiosité", vers les autres patches, chaque patch augmentant la valeur de son compteur de l'énergie reçue du patch émetteur. L'émission remettant à zéro le compteur de radiosité du patch émetteur, on recherche un autre candidat sur lequel appliquer l'algorithme, jusqu'à ce que le maximum des énergies émissibles des patches devienne inférieur à un seuil donné. La somme des contributions reçues par chaque patch durant le déroulement complet de l'algorithme donne la radiosité finale de ces patches. On a donc l'algorithme suivant :

```
radiosite_progressive (patches)
{
  pour (tous les patches) {             /* Initialise les compteurs */
    rad_emissible[patch] =
    rad_accumulee[patch] = rad_propre[patch];
  }

  faire {
    patch_max = patch[premier];         /* Recherche le plus brillant */
    pour (tous les patches) {
      si (rad_emissible[patch] > rad_emissible[patch_max])
        patch_max = patch;
    }
    rad_max                  = rad_emissible[patch_max];
    rad_emissible[patch_max] = 0;       /* L'energie est emise */

    pour (r allant de 1 a (rad_max / quantum)) {
      rayon = rayon_partant (patch_max);
      patch = intersection  (rayon);
      rad_emissible[patch] += quantum;
      rad_accumulee[patch] += quantum;
    }
  } tant que (rad_max > epsilon);
}
```

Pour émettre l'énergie d'un patch vers ses voisins, on a le choix entre des méthodes déterministes, qui échantillonnent l'hémisphère visible par un patch et répartissent l'énergie sur celui-ci [Lesa90], et des méthodes stochastiques,

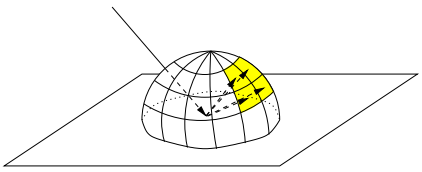
qui émettent l'énergie sous forme de rayons dans des directions de l'hémisphère choisies aléatoirement avec une fonction de répartition adéquate [Mall88,Shir90].

L'avantage des méthodes progressives est qu'elles donnent rapidement une solution approchée rendant compte des contributions les plus importantes, avant de traiter les échanges plus fins entre sources secondaires. Cela permet une prévisualisation en cours de calcul, afin de savoir si l'on souhaite poursuivre celui-ci ou non.

5.3.5 Prise en compte de la spécularité

Sous sa forme initiale, l'algorithme de radiosité ne modélise que l'illumination purement diffuse. Cela est modélisé par le fait que, pour chaque surface élémentaire, on ne stocke qu'une seule valeur d'énergie, qui correpond à l'énergie émise de façon isotrope.

Pour modéliser les aspects spéculaires des transferts énergétiques, il faut introduire une notion de directionnalité de l'émission, et donc de disposer de plusieurs valeurs d'émission associées à des portions différentes de la demi-sphère d'émission.



L'énergie incidente provenant d'une direction donnée est alors distribuée aux valeurs d'émission concernées au moyen d'une fonction de réflexion bidirectionnelle généralisée, qui permet de prendre en compte tous les aspects du matériau dont est constituée la surface élémentaire : spécularité, anisotropie, etc.

5.3.6 Rendu

Le rendu de la scène est réalisé indépendamment du calcul de radiosité. Dans l'algorithme initial, qui ne tenait pas compte de la spécularité, le rendu s'effectuait par Z-buffer. Dans les versions supportant la spécularité, on effectue un lancer de rayons à partir de la position de l'observateur, en prenant comme valeur de pixel la radiance du patch touché par le rayon dans la direction du rayon de vision.

5.3.7 Parallélisation par données véhiculées

Chronologiquement, les premières approches proposées étaient basées sur des algorithmes parallèles existants.

Dans l'algorithme à données véhiculées, chaque processus dispose d'une partie de la base de données des patches de la scène, et doit calculer les lignes de la matrice de facteurs de forme associées à ceux-ci. Pour cela, on organise les processus en anneau logique, et chaque processus envoie une partie de ses patches à son successeur sur l'anneau, en même temps qu'il en reçoit de son prédécesseur, et effectue la projection des patches reçus sur les hémicubes qu'il possède.

Une fois que l'ensemble des patches a transité sur l'anneau, chaque processus a terminé le calcul de ses hémicubes, et le système ainsi construit est résolu en utilisant une version parallèle de la méthode de résolution de Gauss–Seidel.

L'efficacité du calcul des facteurs de forme est correcte, car le temps de communication est faible par rapport au temps de projection sur les hémicubes (possibilité de recouvrement calcul/communication). Cependant, le calcul et le stockage de la matrice sur les processus est très limitant en taille. De fait, la plupart des algorithmes parallèles proposés dans la littérature adoptent un schéma de résolution progressif.

5.3.8 Parallélisation par données dupliquées

L'algorithme à données dupliquées, qui fonctionne avec un principe maître/esclave, est plus destiné à utiliser la puissance de réseaux de stations de travail. Le maître, qui conserve l'ensemble des informations énergétiques, sélectionne le patch le plus brillant, et délègue à un esclave disponible le calcul de distribution énergétique, les facteurs de forme étant alors renvoyés au processus maître.

Afin d'occuper simultanément tous les esclaves, on peut calculer en parallèle les contributions des e patches les plus brillants à un instant donné, où e est le nombre de processus esclaves disponibles. On introduit alors un indéterminisme par rapport à l'algorithme séquentiel. Il a été montré par Cohen [Cohe88] que l'ordre de traitement des patches n'influe pas sur le résultat final, mais sur la vitesse de convergence vers l'état d'équilibre. La parallélisation induit donc des itérations supplémentaires, qui sont cependant largement compensées par l'accélération des calculs.

Les inconvénients de cette approche sont la nécessité de dupliquer l'ensemble des informations géométriques sur chacune des machines, qui limite la taille de la scène par la plus petite taille mémoire disponible, ainsi que le goulot d'étranglement que constitue le maître du point de vue des communications.

5.3.9 Parallélisation par données centralisées

Un premier algorithme parallèle à données centralisées est le portage de l'algorithme précédent sur une machine parallèle à mémoire partagée : tous les processeurs fonctionnent alors de manière identique, en parcourant chacun le tableau centralisé des radiosités émissibles à la recherche du patch le plus énergétique, en remettant sa valeur à zéro dès qu'ils l'ont trouvé, et en se chargeant eux-mêmes des calculs de distribution de la radiosité du patch.

Une autre technique consiste à utiliser une mémoire virtuellement partagée sur une architecture à mémoire distribuée, comme l'avait fait Badouel pour le lancer de rayons. En pratique, un processus choisit un patch, calcule les facteurs de forme associés, puis distribue l'énergie à travers la scène. Cette formulation, similaire à l'algorithme séquentiel, n'est possible que parce que le problème des synchronisations est repoussé au niveau de la mémoire virtuelle. Les données sont réparties aléatoirement dans les caches objets des processus au démarrage de l'algorithme, puis sont dupliquées lorsque d'autres processus en font la demande. Afin d'éviter les pertes d'efficacité dues aux sections critiques pour éviter

les conflits d'écriture sur les variables partagées, les compteurs de radiosité sont mis à jour de façon distribuée.

Cette approche, si elle a donné d'excellents résultats pour le lancer de rayons, est plus délicate à mettre en œuvre pour la radiosité. En effet, dans le premier cas, deux rayons voisins possèdent une forte probabilité de rencontrer les mêmes objets, et cette cohérence spatiale minimise le risque de défauts de page. En revanche, dans l'algorithme de radiosité, deux patches, mêmes voisins, peuvent interagir avec l'ensemble de la scène, et donc annuler l'effet de cache.

5.3.10 Parallélisation par données réparties

Tous les modèles vus précédemment nécessitent soit une duplication des données, soit ne sont pas efficaces. L'approche suivante a pour but de traiter des scènes de grande taille, et nécessite donc de répartir les données géométriques sur les processeurs. Pour cela, on s'appuie sur la division de l'espace en voxels, qui accélère les calculs d'intersection, en attribuant à chaque processeur une portion de ces voxels.

Les objets à cheval sur une frontière sont dupliqués sur chacun des processeurs concernés, afin que chaque processeur sache si le rayon qu'il est en train de traiter intersecte ou non un objet de son sous-espace, avant de le passer au processeur en charge du sous-espace voisin. Dans le cas contraire, il faudrait effectuer des « retours arrière » entre processeurs, qui génèreraient de nombreux messages supplémentaires et augmenteraient la complexité du programme.

Le principal problème posé par cette approche est l'équilibrage de la charge : comme on répartit les objets en conservant la cohérence spatiale, la masse de calculs à effectuer par processeur dépend fortement de la géométrie de la scène et de son placement : présence ou non de sources primaires de lumière, localisation des obstacles à sa progression, ... Il ne s'agit donc pas d'équilibrer le nombre d'objets par processeur, mais le nombre de rayons à traiter, ce qui ne peut être prédit statiquement. La seule méthode possible pour évaluer la charge sur la scène consiste à démarrer le calcul réel (qui ne peut, vu la taille de la scène, se faire que sur la machine parallèle elle-même), et donc à mettre en place un mécanisme d'équilibrage dynamique de la charge.

De fait, le découpage de l'espace ne peut se faire en plus d'une dimension, car sinon l'équilibrage de charge entre voisins ferait intervenir tous les processeurs voisins de la paroi, et ne serait pas obtensible dans tous les cas (problème de rectilinéarité). Cependant, le partitionnement unidimentionnel n'est pas scalable, car les processeurs situés au milieu voient leur trafic augmenter lorsque le nombre de processeurs agmente. Pour se détacher du problème de rectilinéarité, on peut mettre en place un mécanisme d'indirection, pour que tout processeur sache quel processeur est responsable d'un voxel donné, et lui redirige ses rayons entrants. Il se pose alors des problèmes de cohérence de ces tables d'indirection lorsqu'un voxel change de propriétaire alors que des rayons sont en chemin. Ce problème n'est pas insoluble, mais complique l'algorithme, surtout quant au nombre de canaux de communications et de tampons à mettre en place (voir ci-dessous).

Les informations énergétiques sont elles aussi distribuées : chaque patch possède un compteur de radiosité (ou plusieurs, lorsque la spécularité est prise en

compte) localisé sur le processeur propriétaire de la géométrie de la facette. Si la facette est à cheval sur plusieurs processeurs, l'un d'entre eux est désigné comme le propriétaire des informations énergétiques, qui ne sont donc pas dupliquées. Lorsqu'un rayon touche une facette à cheval sur une portion n'appartenant pas au processeur propriétaire, le processeur qui a calculé l'intersection lui envoie un message de contribution pour que l'énergie incidente soit correctement accumulée et puisse éventuellement donner lieu à l'émission d'une nouvelle gerbe de rayons.

Un autre problème important est celui de la congestion : lorsque chaque processeur lance une ou plusieurs gerbes de rayons correspondant à ses patches les plus brillants, le volume de rayons générés peut conduire à un blocage du système, chaque processeur étant en attente que les rayons qu'il produit soient acceptés par le processeur voisin pour faire de la place pour accepter et traiter ceux qui en viennent. Afin d'empêcher ce cas de figure, il faut mettre en place des mécanismes de tampons à l'extrémité de chaque canal de communication, dédier à des tâches spécialisées la réception et l'envoi des rayons, et mettre en place un mécanisme de limitation du nombre de rayons simultanément actifs, qui ne doit pas être supérieur à quatre fois la taille des tampons (pour ne pas créer de boucle propice à l'interblocage). Ici encore, on se trouve face à une limitation non scalable, car pour que l'efficacité soit constante la taille des tampons situés sur chaque processeur doit être proportionnelle au nombre de processeurs.

Le mécanisme de comptage des rayons actifs est basé sur un jeton de comptage circulant sur un anneau logique parcourant l'ensemble des processeurs. Chaque processeur dispose d'un nombre maximum de rayons à créer, et comptabilise donc le nombre de rayons qu'il crée, chaque rayon emportant avec lui le numéro du processeur qui l'a crée. Lorsqu'un processeur détermine l'intersection d'un rayon avec un des objets qu'il possède, il détruit le rayon et incrémente dans un tableau spécial la case correspondant au numéro du processeur créateur du rayon, comptant ainsi le nombre de rayons émis pas ce processeur et qu'il a détruits.

Parallèlement à cela, un jeton collecteur, constitué d'un tableau comportant autant de cases que de processeurs, circule sur l'anneau logique. Chaque fois que le jeton arrive sur un processeur, il collecte les nombres de rayons détruits par celui-ci, et l'informe du nombre de ses rayons détruits par les autres processeurs, puis met à zéro la case du jeton correspondant au processeur courant. Ainsi, le processeur sait combien de ses rayons ont été supprimés, et peut donc relancer au plus ce nombre de rayons avant un autre passage du jeton.

La terminaison de cet algorithme n'est pas simple à déterminer. En effet, un unique rayon en transit sur le réseau peut apporter suffisament d'énergie à un patch pour déclencher l'émission d'une gerbe de rayons, qui pourra elle-même en déclencher une autre, provoquant ainsi une flambée d'activité sur des processeurs jusqu'alors inactifs. De fait, on ne peut être sûr que l'algorithme a terminé que si d'une part aucun processeur n'a de patch dont l'énergie courante est supérieure au seuil, et si d'autre part il ne reste plus de rayon en transit.

Ici encore, il s'agit de compter les rayons actifs, et le mécanisme de terminaison est donc lui aussi basé sur un jeton circulant, qui peut être le même que celui utilisé pour le comptage des rayons. Chaque fois que le jeton rencontre

un processeur actif ou dont le nombre de rayons vivants (après soustraction des rayons détruits) est supérieur à zéro, le test de terminaison est désactivé, et ne sera réactivé qu'au premier processeur inactif rencontré, dont il stockera la valeur. Le test ne sera réussi que lorsque le jeton actif sera revenu à son point de départ, et se transformera alors en message de terminaison.

Il est nécessaire de compter le nombre de rayons émis par chaque processeur, et non le nombre total de rayons émis, car sinon le test n'est pas assez précis et peut conduire à une terminaison prématurée.

Chapitre 6

Rendu volumique

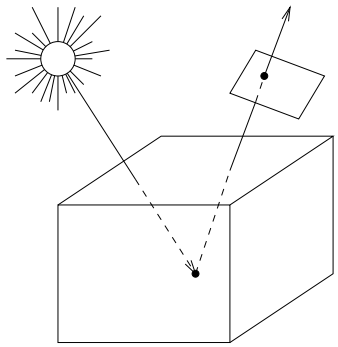
6.1 Équations de rendu volumique

De nombreuses applications, comme l'imagerie médicale, la sismologie, ou la microscopie électronique, construisent et utilisent de grands tableaux tridimensionnels de données. Ces volumes peuvent contenir plusieurs Méga-octets de données, et le rendu interactif de telles quantités de données nécessite énormément de calculs et de transferts mémoire.

Les méthodes de rendu surfacique extraient du tableau volumique des surfaces de contour, qui sont ensuite affichées au moyen d'algorithmes de rendu surfacique. Ces méthodes permettent d'atteindre des fréquences d'affichage temps-réel, mais ne rendent pas compte de l'intégralité des données.

Les méthodes de rendu volumique utilisent les informations de couleur et d'opacité possédées par chaque voxel pour afficher sur le plan image le résultat de la traversée du volume par la lumière issue de sources extérieures.

Afin de simplifier l'équation de rendu, il est presque toujours supposé que la lumière n'est diffractée qu'une seule fois avant de toucher l'œil de l'observateur, qu'elle n'est pas atténuée avant sa première diffraction, et que l'absorbtion de la lumière est isotrope.



6.2 Algorithmes basés sur le lancer de rayons

Les algorithmes basés sur le lancer de rayons produisent une image en lançant un rayon à travers le volume pour chaque pixel de l'image, et en intégrant les informations de couleur et d'opacité le long du rayon. Les algorithmes de ce type sont qualifiés d'« *image-order* », car leurs boucles les plus externes itèrent sur les pixels de l'image :

```
for (yi = 0; yi < Ymax; yi ++) {
  for (xi = 0; yi < Xmax; xi ++) {
    for (zi = 0; zi < Zmax_rayon; zi ++) {
      for (xv = fx(xi, yi, zi); xv < Fx(xi, yi, zi); xv ++) {
        for (yv = fy(xi, yi, zi); yv < Fy(xi, yi, zi); yv ++) {
          for (zv = fz(xi, yi, zi); zv < Fz(xi, yi, zi); zv ++)
            image[xi, yi] += contrib (xv, yv, zv, zi);
        }
      }
    }
  }
}
```

Les deux boucles les plus externes itèrent sur les pixels de l'image. La boucle suivante itère sur les points d'échantillonnage le long d'un rayon. Finalement, les trois boucles les plus internes itèrent sur les voxels nécessaires au filtre de rééchantillonnage pour reconstruire un échantillon le long du rayon.

L'inconvénient majeur des algorithmes basés sur le lancer de rayons est qu'ils n'accèdent pas aux données du volume dans l'ordre naturel du stockage, puisqu'ils le traversent dans des directions arbitraires. De fait, ils passent plus de temps à calculer les positions des points d'échantillonnage (c'est-à-dire les indices des voxels dans les boucles les plus internes) et à effectuer les calculs d'adresses qu'à calculer le rendu.

Les techniques d'optimisation basées sur la répartition spatiale des données aggravent encore plus ce problème, en imposant le calcul supplémentaire de volumes englobants.

6.3 Algorithmes d'« aplatissement » (« *splatting* »)

À la différence des algorithmes basés sur le lancer de rayons, les algorithmes d'aplatissement itèrent sur les voxels.

```
for (xv = 0; xv < Xv; xv ++) {
  for (yv = 0; yv < Yv; yv ++) {
    for (zv = 0; zv < Zv; zv ++) {
      for (zi = fz(xv, yv, zv); zi < Fz(xv, yv, zv); zi ++) {
        for (yi = fy(xv, yv, zv); yi < Fy(xv, yv, zv); yi ++) {
          for (xi = fx(xv, yv, zv); xi < Fx(xv, yv, zv); xi ++)
            image[xi, yi] += contrib (xv, yv, zv, xi, yi, zi);
        }
      }
```

```
      }
    }
  }
```

Comme les algorithmes d'aplatissement itèrent sur le volume, ils peuvent parcourir celui-ci selon son ordre naturel de stockage. En revanche, le calcul du filtre de rééchantillonnage est très coûteux, car celui-ci dépend de la position de l'observateur : il peut être mis à l'échelle, tourné, et transformé de façon arbitraire.

En théorie, les algorithmes d'aplatissement peuvent fournir les mêmes images qu'avec les algorithmes de lancer de rayons. En pratique, du fait que le calcul des paramètres des filtres est difficile, des approximations sont utilisées, qui permettent soit d'obtenir une exécution efficace, soit des images de qualité, mais jamais les deux simultanément.

6.4 Algorithme *Shear-Warp*

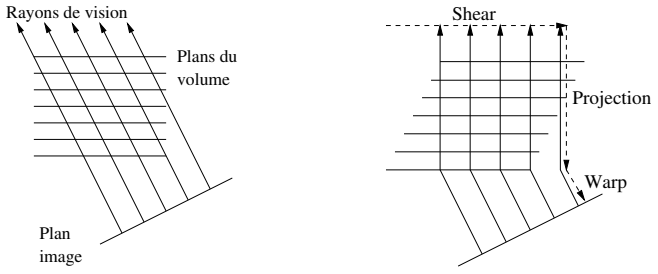
L'algorithme de rendu volumique *shear-warp* permet de gagner plus d'un ordre de grandeur en exploitant conjointement les cohérences spatiales du volume de données et du plan image.

6.4.1 Principe

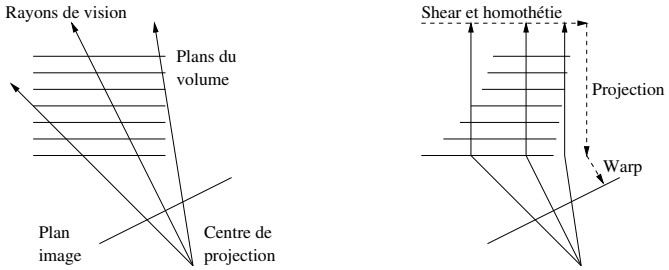
Dans cet algorithme, on n'itère pas sur les pixels de l'image, mais sur les voxels de la scène. Cependant, on évite la complexité des algorithmes d'aplatissement en décomposant la projection des voxels sur le plan image en deux étapes :

- tout d'abord, on effectue une projection des voxels sur un plan objet parallèle à l'une des faces du volume à visualiser. Comme le plan objet utilise le même système de coordonnées que le volume, cette projection peut se faire de façon efficace en suivant l'ordre naturel de stockage du volume ;
- ensuite, le plan objet est projeté sur le plan image, à un coût très bas car il ne s'agit plus de projections d'informations volumiques mais bidimensionnelles.

Afin d'optimiser la construction de l'image intermédiaire, il est possible de réorganiser les plans du volume de façon à ce que la projection des voxels sur le plan objet soit toujours parallèle à l'axe de profondeur. Pour cela, on décale (« *shear* ») les plans du volume en fonction de la direction initiale des rayons de vision, comme on le voit ci-dessous dans le cas d'une perspective isométrique :



Cette technique est également valable pour les perspectives classiques, au prix d'un changement d'échelle des plans en fonction de leur profondeur :



On obtient donc un algorithme en trois phases (voir [3] pour la projection isométrique et [4] pour la perspective normale) :

- transformation des données du volume dans l'espace objet déformé, par translation et remise à l'échelle de chaque plan. La direction des plans est choisie comme celle étant la plus perpendiculaire à la direction de vision. Comme les coefficients de translation et d'homothétie ne sont pas nécessairement entiers, chaque plan objet doit être correctement rééchantillonné ;
- composition des plans objet déformés, du plus proche de l'observateur vers le plus lointain, en utilisant l'opérateur de recouvrement : les contributions ajoutées derrière un pixel translucide sont atténuées, et celles ajoutées derrière un pixel opaque sont ignorées. On obtient ainsi une image intermédiaire distordue, car non perpendiculaire à la direction réelle de vision et non alignée avec les axes de l'image finale ;
- projection de l'image intermédiaire distordue sur le plan image, pour obtenir l'image finale.

Afin de ne pas manipuler de gros volumes de données, les mises à l'échelle peuvent se faire de façon logique, en calculant à la volée ces transformations lors du parcours des plans du volume selon la direction choisie.

6.4.2 Structures de données

Des études statistiques sur des cas réels ont montré qu'entre 65 et 90 pour cent des voxels sont considérés comme transparents. Pour éviter de les stocker, on utilise une structure de type « *run-length* ».

Comme le parcours de la structure compactée ne peut se faire que dans la direction de compactage, on pré-calcule trois structures compactées à partir des données volumiques, une par direction principale, de sorte qu'un changement de direction passe inaperçu lors de la phase de visualisation.

6.4.3 Parallélisation

L'étape *warp* représente, pour des volumes de données de taille moyenne, moins de 20 % du temps total de rendu, l'essentiel du temps étant consommé par la phase de composition des voxels sur l'image intermédiaire, sur laquelle doit se porter l'effort de parallélisation.

La parallélisation de l'algorithme de *shear-warp* se réalise assez naturellement sur une machine multi-processeurs. En revanche, la parallélisation sur machine distribuée pose de nombreux problèmes, en particulier la redistribution des données entre processeurs lorsque la direction de vision change brutalement.

Chaussumier a implémenté une version parallèle de l'algorithme *shear-warp* sur une grappe de PC reliés par un réseau d'interconnexion Myrinet. La distribution des calculs se fait par lignes de l'image intermédiaire, afin de conserver une certaine localité des données, ainsi que les algorithmes de terminaison anticipée. Les données des lignes de voxels correspondant aux lignes de l'image intermédiaire sont distribuées sur les processeurs, de façon à minimiser la réplication. Ceci n'est pas simple, car cette distribution est dépendante du point de vue et morcelle chaque coupe de données sur chaque processeur.

Les communications engendrées par cette distribution de données ont lieu chaque fois que le volume est déformé, c'est-à-dire à chaque changement de point de vue. Les données n'étant pas répliquées, chaque processeur demande à tous les autres les lignes qu'il lui manque dans chacune des coupes du volume qu'il doit traiter. Il faut alors réaliser une multi-distribution de morceaux de structures creuses, puisque chaque processeur envoie des données différentes à tous les autres.

L'utilisation de structures creuses fait qu'on ne peut prédire simplement la quantité de travail à partir de la quantité totale de données et de l'angle de vue. Un équilibrage de charge simple basé sur l'entrelacement de blocs de lignes de l'image intermédiaire n'étant pas efficace, il faut avoir recours à des techniques d'équilibrage dynamique de la charge. Cependant, un équilibrage adaptatif se basant sur le rendu précédent n'est pas suffisant lorsque l'on change de point de vue de manière arbitraire.

Pour remédier à cela, chaque processeur calcule, sur les portions de données qu'il possède, un coût partiel pour chacune des lignes du nouveau rendu. Les tableaux des coûts partiels de lignes sont alors sommés par l'ensemble des processeurs (réduction de type « *all-reduce* »), de telle sorte que chaque processeur puisse calculer le coût total du rendu (en sommant les cases du tableau réduit), et en déduise localement une distribution des lignes ; il ne reste plus alors à chacun qu'à effectuer la redistribution des données en conséquence.

Cependant, la redistribution est coûteuse, et empêche toute scalabilité si les communications ne sont pas recouvertes par le calcul. Pour recouvrir les communications, il faut identifier, à l'intérieur de l'algorithme, des communications et des calculs indépendants pouvant s'effectuer simultanément. Dans l'algorithme *shear-warp*, la plus petite granularité de composition efficace est au niveau des lignes (pour bénéficier du codage *run-length*). La granularité choisie pour le recouvrement a été la coupe (ensemble de lignes), car elle permet un recouvrement efficace sans multiplier le nombre de messages.

En pratique, sur une grappe de 4 PC interconnectés par un réseau Myrinet, une image de 512^3 voxels est calculée en 1.5 secondes, avec un déséquilibre de charge n'excédant pas 10 %.

Annexe A

Cartes graphiques

Ce chapitre est dédié aux architectures pipe-linées mises en œuvre dans les cartes graphiques pour PC. Une présentation des capacités de ces périphériques et de leur programmation avec Direct3D est disponible dans [2].

A.1 Pipe-line graphique

Le rendu interactif d'images sur ces cartes graphiques est réalisé au moyen de cinq étapes principales :

- la génération des données graphiques à visualiser, et leur organisation en des structures de données adaptées à une visualisation efficace. Cette étape est réalisée pour partie au niveau de la conception des modèles (définition des objets, des textures, de leur animation, définition et positionnement des sources lumineuses), et pour partie à l'exécution de l'application (présence ou non des objets, placement et orientation de ceux-ci).

  En particulier, lorsque plusieurs modèles, de complexité croissante, ont été fournis pour chaque objet, il s'agit de choisir le modèle qui rendra un effet visuel satisfaisant avec le moins de primitives possibles. Ce choix du niveau de détail souhaité (LOD, pour « *Level Of Detail* ») s'effectue en fonction de la distance de l'objet à l'observateur ;

- le parcours des structures de données graphiques, afin de transférer les données correspondantes à l'API du système graphique (bibliothèque et matériel). La variabilité des traitements à effectuer à ce niveau fait que cette phase est le plus souvent réalisée de façon logicielle, mais un support matériel peut être fourni (circuits DMA permettant le chargement rapide des données dans la mémoire de la carte graphique, par exemple) ;

- la transformation des données graphiques structurées depuis le système de coordonnées de la scène vers le système de coordonnées lié à l'observateur, le clipping des objets situés à l'extérieur de la pyramide de vision, le calcul d'illumination, et la projection sur le plan de vision.

  Le plus souvent, seul un sous-ensemble de toutes les transformations possibles est implémenté de façon matérielle. Par exemple, certaines architectures ne peuvent accélérer que les traitements mettant en œuvre une

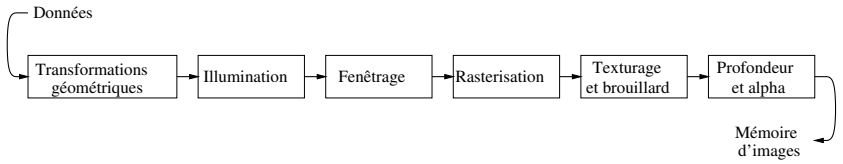
unique source lumineuse située à l'infini, alors que d'autres n'en bénéficient pas. De même, certains accélérateurs peuvent avoir des circuits dédiés qui transforment plus rapidement les vecteurs pairs de données géométriques du fait de l'architecture parallèle qu'ils mettent en œuvre. La connaissance précise des capacités et limitations du matériel permet de construire des applications graphiques optimisées en se basant sur les chemins de traitements les plus efficaces (« *fast paths* ») ;

- la rasterisation des primitives projetées, c'est-à-dire leur transformation en un ensemble de pixels sur lesquels sont appliquées des opérations de tramage (« *stencil* »), de transparence (« *alpha* »), d'élimination des parties cachées (« *Z-buffer* » ou « *W-buffer* »), et de texturage. À l'issue de cette étape, les données produites en sortie consistent en des informations variées telles que profondeur, couleur, alpha, et coordonnées de texture, qui sont utilisées pour mettre à jour de la mémoire d'image (« *frame buffer* »).

  Ici encore, seul un sous-ensemble des traitements est le plus souvent implémenté de façon matérielle. Les raisons en sont nombreuses, et comprennent le coût, la complexité, le nombre de transistors nécessaires, la taille du marché potentiel, et la vitesse des processeurs. Certains matériels accélèrent uniquement les textures codées dans certains formats (ABGR et pas RGBA), alors que d'autres n'accélèrent pas la génération de textures car dédiées au marché de la CAO où les effets de texture ne sont pas importants ;

- l'affichage, qui transfère les pixels de la mémoire d'image vers le dispositif d'affichage à une fréquence constante.

Les cartes graphiques actuelles basent l'augmentation de leurs performances sur l'implémentation par matériel des étages de transformation géométrique, de rasterisation, et de texturage, qui sont eux-mêmes décomposés en plusieurs unités fonctionnelles. Les pipe-lines de ces cartes prennent en entrée des données géométriques codant des primitives (triangles, bandes de triangles, éventails de triangles, lignes, points, et textures), qui en sortent sous forme d'ensembles de pixels dans la mémoire d'images.



Les étages de ce pipe-line correspondent aux traitements suivants :

- transformation des coordonnées géométriques et projection sur le plan image ;
- calculs d'illumination ;
- clipping, calculs de perspective, et mise à l'échelle par rapport à la taille finale de l'image ;

- rasterisation des primitives à partir de leurs coordonnées projetées ;
- plaquage des textures ;
- modification de la prise en compte des pixels en fonction de la profondeur (Z-buffer ou W-buffer), de leurs valeurs alpha, etc.

A.2 Interfaçage

Une caractéristique importante de la carte graphique est la manière dont elle est reliée au processeur et à la mémoire centrale, qui conditionne grandement les latences et les débits des transferts de données géométriques et de texture.

Typiquement, les cartes bas de gamme sont directement connectées sur le bus PCI, et disposent d'une bande passante maximale de 132 Mo/s (32 bits à 33 MHz) qu'elles doivent partager avec tous les autres périphériques connectés au bus. Qui plus est, les données transférées de la mémoire centrale vers la mémoire de la carte graphique doivent transiter par la CPU, augmentant ainsi la demande de puissance processeur et le risque de goulot d'étranglement.

À l'opposé, les cartes haut de gamme peuvent utiliser un bus dédié, de type AGP, offrant une liaison exclusive entre la mémoire centrale et celle de la carte graphique au moyen de circuits DMA, et permettant un débit de 264 Mo/s (AGP), 528 Mo/s (AGP2X), ou bien 1,056 Go/s (AGP4X).

Une autre approche est l'UMA (« *Unified Memory Architecture* »), dans laquelle un bus dédié relie la CPU à la carte graphique, avec un débit actuel de 3.2 Go/s.

A.3 Pipe-line de transformation géométrique

Le pipe-line de transformation géométrique a pour but de transformer les coordonnées géométriques des objets constituant la scène en coordonnées équivalentes dans le système de l'observateur, en vue de leur projection sur le plan image.

A.3.1 Transformations géométriques

Les sommets constituant chacun des objets ou composants d'objets sont représentés dans un système de coordonnées local (« model coordinate system », défini par son origine et ses trois axes orthogonaux).

Les objets sont assemblés et dupliqués par transformations géométriques au sein du système de coordonnées de la scène (« *world space* »), qui sert également à définir la position et l'orientation de l'observateur, ainsi que des sources lumineuses.

Pour pouvoir être rasterisées, les données géométriques doivent être exprimées par rapport au système de coordonnées de l'observateur (« *camera space* »), par transformation du point de vue (« *view transform* »).

Finalement, les objets sont mis à l'échelle en fonction de leur distance à l'observateur et projetés sur le plan image (« espace écran », ou « *post-perspective homogeneous space* ») pour donner l'illusion de la profondeur de la scène (« *projection transformation* »)

Le pipe-line de transformation graphique transforme donc les coordonnées $(X_m, Y_m, Z_m, 1)$ des objets dans l'espace des modèles en coordonnées (X_s, Y_s, Z_s, W_s) dans l'espace écran. Cette transformation s'effectue par multiplications successives des vecteurs coordonnées par des matrices 4×4, ainsi qu'au moyen d'opérations logiques simples.

Un point $P_m = (X_m, Y_m, Z_m, 1)$ est transformé en un point $P_w = (X_w, Y_w, Z_w, 1)$ au moyen d'une matrice 4×4, produit de matrices élémentaires de rotation, de translation, et d'homothétie.

Un point P_w est transformé en un point $P_c = (X_c, Y_c, Z_c, 1)$ au moyen d'une matrice 4×4, produit de matrices élémentaires de rotation et de translation.

Un point P_c est transformé en un point $P_p = (X_p, Y_p, Z_p, W_p)$ au moyen de la matrice de projection :

$$\begin{pmatrix} \frac{2Z_n}{S_w} & 0 & 0 & 0 \\ 0 & \frac{2Z_n}{S_h} & 0 & 0 \\ 0 & 0 & \frac{Z_f}{Z_f - Z_n} & 1 \\ 0 & 0 & \frac{-Z_f Z_n}{Z_f - Z_n} & 0 \end{pmatrix} \text{, où}$$

- S_w est la largeur de l'écran, dans l'espace de l'observateur, sur le plan de clipping proche ;
- S_h est la hauteur de l'écran, dans l'espace de l'observateur, sur le plan de clipping proche ;
- Z_n est la distance au plan de clipping proche dans l'espace de l'observateur ;
- Z_f est la distance au plan de clipping lointain dans l'espace de l'observateur.

La plupart des cartes nécessitent que la matrice de projection ait cette forme, afin que les optimisations du calcul des effets brouillard soient correctes.

Le volume de clipping des points P_p projetés est alors défini par l'ensemble d'inégalités suivant :

$$\begin{cases} -W_p < X_p \leq W_p \\ -W_p < Y_p \leq W_p \\ 0 < Z_p \leq W_p \end{cases} .$$

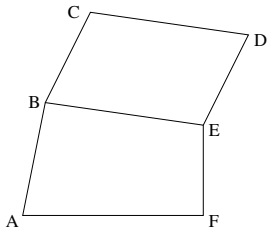
Finalement, les coordonnées écran sont calculées avant d'être soumises au rasteriseur :

$$X_s = \frac{X_p}{W_p} \ , \ Y_s = \frac{Y_p}{W_p} \ , \ Z_s = \frac{Z_p}{W_p} \ , \ W_s = \frac{1}{W_p} \ .$$

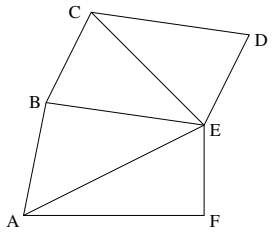
A.3.2 Compactage des données

La plupart des cartes graphiques basent leur rendu sur des primitives de type triangle, qui ont l'avantage d'être toujours planaires et convexes, à la différence des quadrangles qui peuvent être mal formés et/ou concaves. Cependant, la

représentation des objets par des triangles est plus coûteuse au niveau du pipe-line, en ce que les sommets des triangles, devant passer plus de fois dans le pipe-line, engorgent celui-ci.

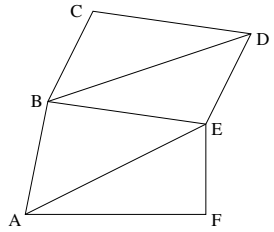


{(A,B,E,F), (B,C,D,E)}

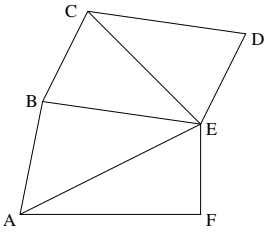


{(A,E,F), (A,B,E), (B,C,E), (C,D,E)}

Pour réduire le nombre des sommets à traiter dans le pipe-line de transformation géométrique, la plupart des cartes disposent de primitives « bande de triangles » (« *triangle strip* ») et « éventail de triangle » (« *triangle fan* »). Une bande de triangle est définie par un ensemble de sommets dont chaque sous-ensemble consécutif de trois forme un triangle. Un éventail de triangles est défini par un ensemble de sommets dont chaque sous-ensemble consécutif de deux, auquel est ajouté le premier sommet de la liste, forme un triangle.



(F,A,E,B,D,C)



(E,F,A,B,C,D)

A.4 Illumination

Les calculs d'illumination s'effectuent après la rasterisation, après que les primitives projetées sur le plan écran ont été converties en fragments, c'est-à-dire en pixels auxquels sont associés des informations de couleur, d'alpha, de profondeur, de coordonnées de texture, etc.

L'apparence des éléments d'une scène dépend de plusieurs paramètres, dont le type de matériau utilisé pour habiller l'objet. Le matériau décrit les propriétés d'illumination de la surface : comment elle traite la lumière et si elle utilise une texture.

L'illumination est calculée pour chaque fragment en calculant une expression qui peut dépendre de nombreux paramètres comme les coordonnées du point courant, les paramètres des sources lumineuses, la normale au point courant, et les caractéristiques courantes du matériau.

Les calculs d'illumination se font habituellement dans l'espace du modèle, afin de ne pas avoir à transformer l'intégralité des informations des normales. La direction et la position des sources lumineuses sont traduites dans le système du modèle en les multipliant par l'inverse de la matrice de transformation de la scène. Si la matrice de transformation n'était pas orthogonale, l'illumination sera erronée.

La formule utilisée fait intervenir une partie diffuse et une partie spéculaire, cette dernière utilisant la technique de Phong pour rendre la brillance des matériaux.

Les types de sources lumineuses comprennent les lumières situées à l'infini, les sources ponctuelles, et les spots, qui sont directionnelles et dont l'intensité décroît lorsque le point considéré s'éloigne de la direction d'illumination (ce sont les plus coûteuses à calculer).

A.5 Texturage

Une texture est un bitmap de couleurs servant à déterminer la couleur de chaque point de la surface sur laquelle est plaquée la texture. Le texturage des surfaces permet de rendre facilement l'apparence des objets ayant des irrégularités de surface (bois, marbre, poussière, roches, végétation, ...), ou de plaquer des images sur des surfaces (panneaux de signalisation, affiches, ...).

A.5.1 Coordonnées de texture

La plupart des textures sont des tableaux bidimensionnels de valeurs de couleurs (et aussi d'alpha). Les valeurs individuelles sont appelés « éléments de texture », ou texels.

Les texels sont référencés dans le système de coordonnées de texture. Quand une texture est appliquée à un objet dans l'espace de la scène, les positions des texels doivent être converties en positions sur l'objet dans le système de la scène, puis dans le système de l'écran. En pratique, les positions des texels sont directement converties dans le système de l'écran. Il s'agit en fait d'un plaquage inverse : pour chaque pixel de l'écran, on calcule la position du texel correspondant dans l'espace de la texture courante, et l'on échantillonne les couleurs présentes en ce point et en ses voisins (on parle alors d'échantillonnage de texture, ou « *texture filtering* »).

A.5.2 Mip-mapping

Afin que l'échantillonnage de texture s'effectue rapidement et de façon la plus exacte possible, on dispose pour chaque texture importante d'une famille de sous-textures de tailles de plus en plus petites, chaque pixel d'une texture étant obtenu par moyennage des valeurs de quatre pixels de la texture du niveau immédiatement supérieur.

Ainsi, lors de l'échantillonnage de texture, le système graphique peut choisir la texture dont la résolution est la plus proche possible du niveau de rendu souhaité, et n'a a effectuer de moyennage que sur les texels immédiatement voisins de celui pointé. Cette technique, appelée « *mip-mapping* », augmente grandement la qualité des rendus de texture, avec un surcoût d'un tiers en mémoire par rapport à la taille de la plus grosse texture stockée.

A.5.3 Illumination par plaquage de textures

La possibilité de combiner plusieurs textures permet de générer des effets d'illumination évolués à ce niveau. En appliquant une ou plusieurs textures lu-

mineuses, on peut ainsi ajouter des effets de lumière et d'ombres qui ne peuvent être traités par la fonction d'illumination.

Une texture lumineuse est une texture ou un groupe de textures qui contiennent de l'information d'illumination. Cette information peut être stockée dans les valeurs alpha de la texture, ou bien dans les valeurs de couleur, ou bien dans les deux.

Les textures lumineuses diffuses doivent être rendues en premier, avant d'y combiner la texture propre de l'objet, les textures spéculaires étant plaquées en dernier.

A.6 Performances

Il existe deux mesures principales de la performance des cartes graphiques, qui testent en fait deux capacités distinctes.

Le débit de remplissage (« *fill rate* ») mesure la vitesse à laquelle les primitives sont converties en fragments et tracées dans la mémoire d'image. On mesure la vitesse de remplissage en nombre de pixels pouvant être tracés par seconde. Cependant, ce nombre est sans signification s'il n'est pas accompagné d'informations supplémentaires sur le type des fragments tracés : profondeur des couleurs (8 bits RGB, 24 bits RGBA, ...), utilisation de textures et de mélanges de textures, techniques d'interpolation, etc.

Le débit de remplissage n'est pas directement équivalent au nombre de pixels affichés à l'écran. Lorsqu'une image est produite, les fragments peuvent être modifiés plusieurs fois, comme dans le cas de polygones tracés en Z-buffer du plus lointain vers le plus proche. Le phénomène consistant à écrire plusieurs fois les mêmes fragments est appelé « complexité en profondeur » (« *depth complexity* ») de la scène. La complexité en profondeur est une valeur moyenne du nombre de fois qu'un fragment est réécrit avant affichage. Plus la complexité en profondeur sera élevée, et plus le risque de limitation de la vitesse d'affichage sera élevé. Il faudra alors avoir recours à des techniques logicielles d'élimination des parties cachées afin de limiter le nombre de polygones à remplir.

Le débit de polygones (« *polygon rate* ») mesure la vitesse à laquelle les polygones peuvent être traités dans le pipe-line graphique. Il est généralement donné en nombre de polygones par seconde, mais ici encore n'est significatif qu'accompagné du type des polygones manipulés : quantité d'information par sommet (coordonnées, normales, couleurs, coordonnées de texture, ...), taille projetée des polygones, etc.

Le débit de remplissage correspond à la phase de rasterisation du pipe-line, alors que le débit de polygones correspond à la phase de transformation. Comme ces deux phases sont celles qui font le plus souvent l'objet d'une accélération matérielle, il est essentiel que celle-ci se fasse de façon équilibrée. Par exemple, si un fabricant annonce un débit de polygones élevé mais un débit de remplissage faible, la carte sera de peu d'intérêt pour la simulation de vol. En effet, ce type d'applications dessine relativement peu de polygones, mais ceux-ci sont texturés et utilisent les effets de brouillard, et se recouvrent souvent (comme des arbres devant un immeuble devant des vallées de terrain), donc ont une grande complexité de profondeur. En revanche, les applications de CAO dessinent un grand

nombre de petit polygones sans utiliser beaucoup les capacités de remplissage de la carte (pas de texture, de brouillard, ou d'effets sur les pixels), et sont donc plus limitées par le débit de polygones.

À titre d'exemple de capacités, on peut regarder les performances théoriques annoncées de la carte Voodoo3, en 2D et en 3D :

8 bits par pixel, 1024x768

Lignes 10 pixels	4 M/s
Lignes 100 pixels	400 K/s
Lignes 500 pixels	80 K/s
Rectangles 10x10	2-4 M/s
Rectangles 100x100	170 K/s
Rectangles 500x500	11.5 K/s
Blit 10x10	40 Mo/s
Blit 100x100	50 Mo/s
Blit 500x500	350 Mo/s

16 bits par pixel, 640x480

1 pixel, Z, Gouraud, sans texture	1.8 Mtri/s
5 pixel, Z, Gouraud, sans texture	1.8 Mtri/s
50 pixel, Z, Gouraud, sans texture	1.4 Mtri/s
1000 pixel, Z, Gouraud, sans texture	70 Ktri/s
50 pixel, Z, texture bilinéaire	1 Mtri/s
50 pixel, Z, texture trilinéaire mip-mappée	375 Ktri/s

Bibliographie

[1] K. Akeley and T. Jermoluk. High performance polygon rendering. *ACM Computer Graphics*, 22(4):239–246, August 1988.

[2] D. Duplan and S. Bontemps. *Direct3D – 3D Temps-Réel sous Windows.* Eyrolles, 1999. ISBN 2-212-09061-7.

[3] F. Klein and O. Kübler. A pre-buffer algorithm for instant display of volume data. In *Proceedings of SPIE (Architectures and Algorithms for Digital Image Processing)*, volume 596, pages 54–58, 1985.

[4] P. Lacroute. *Fast volume rendering using a shear-warp factorization of the viewing transformation.* Thèse de Doctorat, Computer Systems Laboratory, Stanford University, September 1995. Disponible à partir de l'URL `http://www-graphics.stanford.edu/~lacroute/`.

[5] M. Lucas. Parallélisme et synthèse d'images. *Technique et Science Informatiques*, 10(3):171–202, 1991.