

Pour un système de synthèse d'images flexible et évolutif : quelques propositions

Ghassan Jahami

▶ **To cite this version:**

Ghassan Jahami. Pour un système de synthèse d'images flexible et évolutif : quelques propositions. Algorithme et structure de données [cs.DS]. Ecole Nationale Supérieure des Mines de Saint-Etienne; Université Jean Monnet - Saint-Etienne, 1991. Français. NNT : 1991STET4005 . tel-00817677

HAL Id: tel-00817677

https://theses.hal.science/tel-00817677

Submitted on 25 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT ETIENNE

Université de Saint Etienne

N° D'ordre :54 ID

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE
INFORMATIQUE
158, cours Fauriel
42023 SAINT-ETIENNE CEDEX 2
Téléphone 77-42-01-23

THESE

Présentée par

Ghassan JAHAMI

pour obtenir le titre de

DOCTEUR
DE L'UNIVERSITE DE SAINT-ETIENNE

ET DE L'ECOLE NATIONALE SUPERIEURE DES MINES
DE SAINT ETIENNE

(Spécialité: Informatique, Image, Intelligence Artificielle et Algorithmique)

POUR UN SYSTEME DE SYNTHESE D'IMAGES FLEXIBLE ET EVOLUTIF :

QUELQUES PROPOSITIONS

Soutenue à Saint-Etienne le 21 Mars 1991

COMPOSITION DU JURY

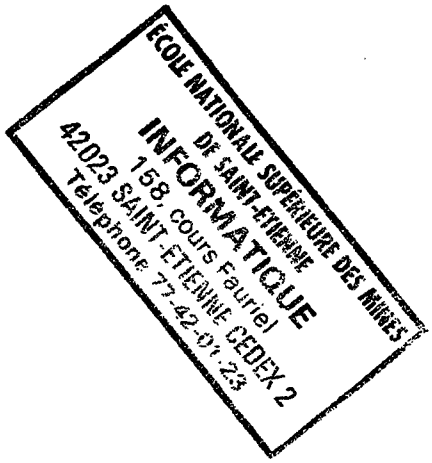
Monsieur	B. PEROCHE	Président
Messieurs	F. FONTENIER Y. GARDAN	Rapporteurs
Madame Monsieur	Y. AHRONOVITZ A. GAGALOWITZ	Examinateurs

PL 2611.1.8

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT ETIENNE

Université de Saint Etienne

N° D'ordre :54 ID



THESE

Présentée par

Ghassan JAHAMI

pour obtenir le titre de

DOCTEUR
DE L'UNIVERSITE DE SAINT-ETIENNE

ET DE L'ECOLE NATIONALE SUPERIEURE DES MINES
DE SAINT ETIENNE

(Spécialité: Informatique, Image, Intelligence Artificielle et Algorithmique)

POUR UN SYSTEME DE SYNTHESE D'IMAGES FLEXIBLE ET EVOLUTIF :

QUELQUES PROPOSITIONS

Soutenue à Saint-Etienne le 21 Mars 1991

COMPOSITION DU JURY

Monsieur	B. PEROCHE	Président
Messieurs	F. FONTENIER Y. GARDAN	Rapporteurs
Madame Monsieur	Y. AHRONOVITZ A. GAGALOWITZ	Examinateurs

PL 2611.1.8

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT ETIENNE

Directeur	:	**M. Philippe SAINT RAYMOND**
Directeur délégué à la recherche	:	**M. Michel DARRIEULAT**
Directeur des études et de la formation	:	**M. Jean-Pierre LOWYS**
Secrétaire général	:	**M. Jean.Claude PIATEK**

PROFESSEURS DE 1ère CATEGORIE

MM.	BISCONDI	Michel	Matériaux
	COINDE	Alexandre	Economie
	DAVOINE	Philippe	Hydrogéologie
	FORMERY	Philippe	Mathématiques Appliquées
	GOUX	Claude	Matériaux
	LE COZE	Jean	Matériaux
	LOWYS	Jean-Pierre	Physique
	MATHON	Albert	Gestion
	PEROCHE	Bernard	Informatique
	PLA	Jean-Marie	Mathématiques
	RIEU	Jean	Matériaux
	SOUSTELLE	Michel	Génie des procédes
	VAUTRIN	Alain	Mécanique et Matériaux
	VERCHERY	Georges	Mécanique et Matériaux

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE
INFORMATIQUE
158, cours Fauriel
42023 SAINT-ETIENNE CEDEX 2
Téléphone 77-42-01-23

PROFESSEUR DE 2ème CATEGORIE

M.	TOUCHARD	Bernard	Physique Industrielle

DIRECTEURS DE RECHERCHE

MM.	LESBATS	Pierre	Matériaux
	THEVENOT	François	Génie des Matériaux

MAITRES DE RECHERCHE

MM.	COURNIL	Michel	Chimie
	DRIVER	Julian	Matériaux
	GIRARDOT	Jean-Jacques	Informatique
	GUILHOT	Bernard	Génie des procédés
	GUY	Bernard	Géologie
	KOBYLANSKI	André	Matériaux
	LALAUZE	René	Génie des procédés
	LANCELOT	Francis	Génie Industriel-Biotechnologie
	MONTHEILLET	Frank	Matériaux
	THOMAS	Gérard	Génie des procédés
	TRAN MINH	Cahn	Génie Industriel-Biotechnologie

PERSONNALITES HABILITEES OU DOCTEURS D'ETAT

MM.	AIVAZZADEH	Sahram	Mécanique et Matériaux
	BIGOT	Jean-Pierre	Génie Industriel-Biotechnologie
	BRODHAG	Christian	Matériaux
	DECHOMETS	Roland	Stratégie du Développement
Mme.	GOEURIOT	Dominique	Matériaux
MM.	LONDICHE	Henry	Génie Industriel-Biotechnologie
	PIJOLAT	Christophe	Génie des procédés
Mme.	PIJOLAT	Michèle	Génie des procédés

PERSONNALITES EXTERIEURES A L'ECOLE AUTORISEES A ENCADRER DES THESES

M.	BENHASSAINE	Ali	Génie des procédés (Ecole des mines Ales))
	BOURGOIS	Jacques	Génie Industriel-Biotechnologie (Univ St Etienne)
	GRAILLOT	Didier	Ingénieur RHEA
	MAGNIN	Thierry	Matériaux (Univ Lille)
	VERGNAUD	Jean-Marie	Génie des procédés (Univ St Etienne)

ÉCOLE NATIONALE SUPÉRIEURE DES MINES
DE SAINT-ETIENNE
INFORMATIQUE
158, cours Fauriel
42023 SAINT-ETIENNE CEDEX 2
Téléphone 77-42-01-23

Remerciements

Je voudrais remercier les personnes qui ont accepté de juger ce travail ainsi que toutes celles qui m'ont aidé au cours de ces années de thèse.

- Monsieur le professeur Bernard Peroche, directeur du département Informatique Appliquée de l'Ecole des Mines de Saint Etienne, de m'avoir accueilli au sein de son équipe ainsi que pour sa compétence et sa disponibilité. Ses très précieuses remarques ont été essentielles pour les études que j'ai effectuées. Aussi, je voudrais le remercier pour sa grande patience et ses nombreuses remarques au niveau de la rédaction de ce manuscrit.

- Messieurs les professeurs
Yves Gardan, de l'université de Metz,
Guy Fontenier, de l'Ecole Nationale Supérieure des Télécommunications de Paris,
pour avoir accepté la lourde charge d'être rapporteurs de cette thèse.

- Monsieur le professeur André Gagalowitz, directeur de recherche à l'INRIA, d'avoir accepté le rôle d'examinateur dans le jury de cette thèse, et de m'avoir donné l'occasion de faire une pré-soutenance devant lui.

- Madame Yoland Ahronovitz, maître de conférence à l'université de Saint Etienne, de m'avoir fait l'honneur de participer au jury.

- Tous les membres passés, présents de l'équipe images, pour leurs supports moral et scientifiques : Jacqueline Argence, Michel Beigbeder, Helmi Ben Amara, Annie Corbel-Bougeat, Mohamed Cheaito, Gilles Fertey, Gabriel Hanoteaux, Roland Jegou, Dominique Michelucci, Jean-Michel Moreau, Zhigang Nié, Philippe Jaillon, Mohand Ourabah Benouamer, Marc Roelens, Pascal Roudier, Samy aït-Aoudia, François Jaillet.
Je voudrais plus spécialement remercier Dominique michelucci pour ses précieuses remarques qui m'ont beaucoup aidé dans la rédaction du chapitre 4 de cette thèse.

- Tous les membres du département Informatique Appliquée pour leur amitié et leur aide.

- Mes parents, mes deux frères Chawki et Ali et mes deux sœurs Khouloude et Maha, pour leur affection et leur soutien que je sentais malgré les milliers de kilomètres qui nous séparent. Je voudrais aussi remercier mon amie Colette Gillet qui était toujours à côté de moi pour m'aider à surmonter les difficultés. Enfin, j'aimerais remercier tous mes copains et amis qui m'ont beaucoup aidé et soutenu.

TABLE DES MATIERES

Introduction 9

Chapitre 1 : Un système de synthèse d'images tridimensionnelles ILLUMINES

1.1 Introduction 15
1.2 L'architecture d'ILLUMINES 15
1.3 Le modeleur CASTOR 17
1.3.1 La modélisation volumique 17
1.3.2 Castor 20
1.4 La saisie de la scène 21
1.5 Les algorithmes de visualisation 24
1.5.1 Visualisation en fil de fer 24
1.5.2 Visualisation par le tampon de profondeur 24
1.5.3 Visualisation par balayage de ligne 25
1.5.3.1 Principe général de l'algorithme d'Atherton 26
1.5.3.2 Version programmée à l'EMSE 27
1.5.4 L'algorithme VOIR 29
1.5.5 Visualisation en tracé de rayons 29
1.5.5.1 Principe de l'algorithme du tracé de rayons 29
1.5.5.2 SKY 32
1.6 Premiers constats. 34

Chapitre 2 : Un modeleur flexible et évolutif orienté objet.

2.1 Introduction 39
2.2 Présentation des travaux déjà publiés 41
2.3 La programmation orientée objet. 43

2.3.1 Définitions. 45
2.3.2 Avantages et inconvénients de la programmation orientée objet 49
2.3.3 Une méthodologie pour le choix des classes 51
2.4 Les caractéristiques du modeleur CASTORC++ 56
2.4.1 La conception de la hiérarchie des classes. 56
2.4.2 Flexibilité du modeleur. 60
2.4.3 Implémentation. 64
2.5 L'utilisation de C++ 68
2.5.1 Présentation de C++ 68
2.5.2 Quelques problèmes avec C++ 70
2.5.3 Un exemple de description de scène en CastorC++. 73
2.6 Conclusion. 74

CHAPITRE 3 : Mixage d'algorithmes en synthèse d'images

3.1 Introduction 79
3.2 Mixage des algorithmes d'Atherton et du tracé de rayons avec interactions en réflexion et en transparence. 82
3.2.1 Description de la méthode. 82
3.2.2 Avantages de la méthode. 86
3.2.3 La gestion de la place mémoire. 91
3.3 Mixage des algorithmes d'Atherton et du tracé de rayons avec interaction en ombre portée. 96
3.4 Implémentation et résultats. 98
3.5 Conclusion. 102

CHAPITRE 4 : La gestion des niveaux des détails.

4.1 Introduction. 105
4.2 Rappels des solutions déjà proposées. 106
4.3 Génération des différentes descriptions d'un objet. 111
4.3.1 Génération manuelle des différentes descriptions d'un objet. 111
4.3.2 Génération automatique des différentes descriptions d'un objet. 116
4.4 L'utilisation de la texture. 118
4.4.1 La détermination de la texture. 120
4.4.2 La détermination de l'objet_silhouette. 123
4.4.3 critère de platitude. 127
4.4.4 Le problème de l'éclairement. 128
4.4.5 Résultats et perspectives. 129
4.5 Les objets fondants. 133
4.5.1 Principe 133
4.5.2 Changement de couleur entre deux descriptions 134
4.5.3 Changement de la matière d'un objet entre deux descriptions. 136

4.6 Conclusion 140

Conclusion 143

INTRODUCTION

Dans le processus qui conduit à la réalisation d'une image de synthèse, la première étape consiste à introduire dans la mémoire de l'ordinateur les informations nécessaires à la définition de la scène. A partir de différents outils mis à sa disposition, le concepteur de l'image crée un *modèle* de la scène, d'où le nom de *modélisation* de cette étape. Ce processus aboutit ensuite à la visualisation de l'image sur un support (écran, table traçante, ...), c'est la deuxième étape : la visualisation. Cette étape consiste à exploiter les données introduites lors de la modélisation, afin de produire une image de la scène. On présente généralement le processus de la fabrication d'une image de synthèse par le pipeline de la figure 1. Chacune de ces deux étapes peut être divisée en deux sous étapes comme le montre la figure 2. L'étape géométrie consiste en la conversion de toutes les primitives de modélisation en une seule primitive géométrique (généralement un polygone) sur laquelle travaille l'algorithme de visualisation. Cette étape n'est pas indispensable si l'algorithme utilisé visualise directement les primitives de modélisation, et, par suite, n'existe pas forcément dans le pipeline. Les différentes étapes de la figure 2 forment des entités distinctes qui ont fait l'objet, parallèlement, de beaucoup d'études. Des résultats importants ([Newm 79], [Fole 82], [Pero 88]) ont été acquis dans ces différents domaines, surtout dans les domaines de la visualisation et du rendu.

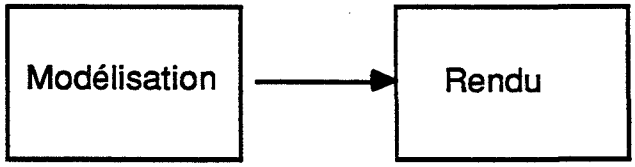


FIGURE 1

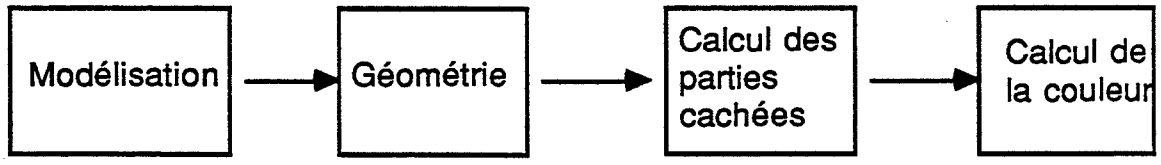


FIGURE 2

Depuis plusieurs années, l'école des Mines de Saint Etienne développe un système de synthèse d'images baptisé ILLUMINES ([Beig 90]). Basé sur un modeleur de type arbre de construction, ce système comporte plusieurs moyens de saisie et plusieurs algorithmes de visualisation et méthodes de calcul de rendu (cf chapitre 1). L'objectif principal de ce système est de favoriser nos recherches dans le domaine de la synthèse d'images. A ce titre, il doit comporter le maximum de *flexibilité* et d'*évolutivité* possible :

évolutivité : en tant que laboratoire de recherche, nous sommes souvent amenés à faire des modifications dans les méthodes et les algorithmes existants. Ces modifications consistent à tester de nouvelles idées afin de les évaluer et les comparer aux méthodes déjà disponibles dans le système. Par *évolutivité*, nous entendons la possibilité d'effectuer ces modifications d'une façon aisée. Toutes les étapes du pipeline graphique sont concernées : de nouvelles méthodes peuvent apparaître pour la modélisation des scènes, ainsi que pour leur visualisation et leur rendu. Pour cette raison, une vue générale du système est nécessaire, et l'étude de l'*évolutivité* doit s'étendre sur toutes les étapes de la figure 2.

flexibilité: le simple pipeline de la production d'une image, avec un seul flux d'informations entre les différentes étapes (cf figure 2), est assez limité et manque de souplesse. En effet, le concepteur de la scène à visualiser est obligé d'utiliser une seule méthode de modélisation par exemple pour tous les objets de la scène. Or, selon le type des objets qu'on souhaite modéliser, on peut avoir besoin d'utiliser des méthodes plus ou moins sophistiquées. Un système de synthèse d'images *flexible*, permettant l'utilisation de plusieurs méthodes de modélisation (cf figure 3) et/ou l'utilisation de plusieurs méthodes de visualisation et de rendu (cf figure 4) dans la même scène, est donc nécessaire.

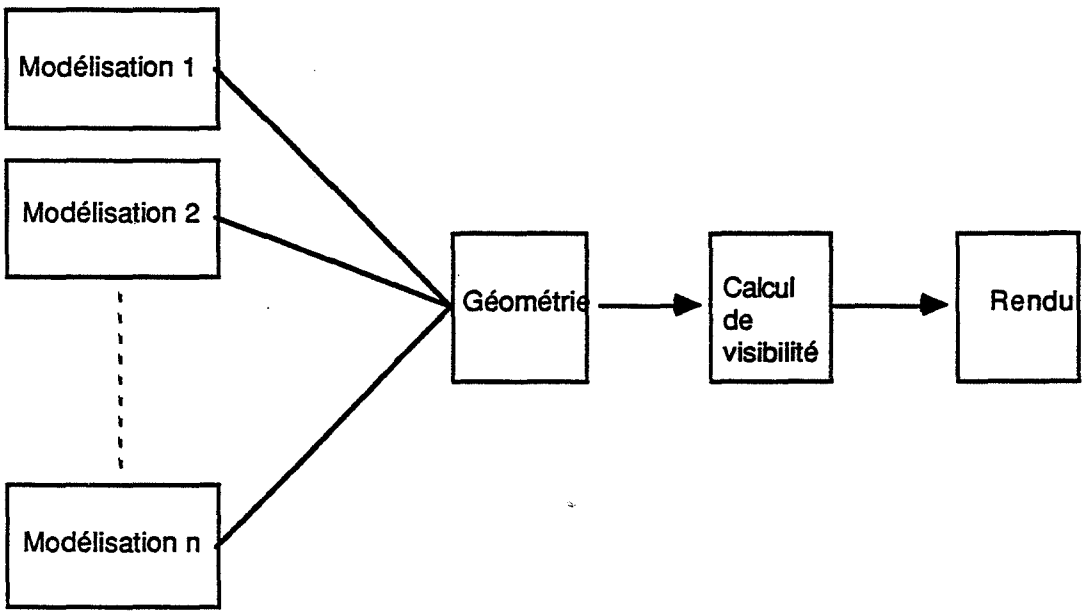


FIGURE 3

Le système doit encore être *flexible* par rapport à l'utilisateur naïf en allégeant au maximum son travail. Par exemple, le système doit fournir des outils pour aider l'utilisateur à gérer les niveaux de détails dans la scène qu'il est en train de concevoir ; il doit assurer un choix automatique du degré de facettisation d'une

primitive de modélisation et des choix automatiques de l'algorithme d'élimination des parties cachées, de la méthode de lissage et du modèle d'éclairement à utiliser pour un objet ou un groupe d'objets donnés.

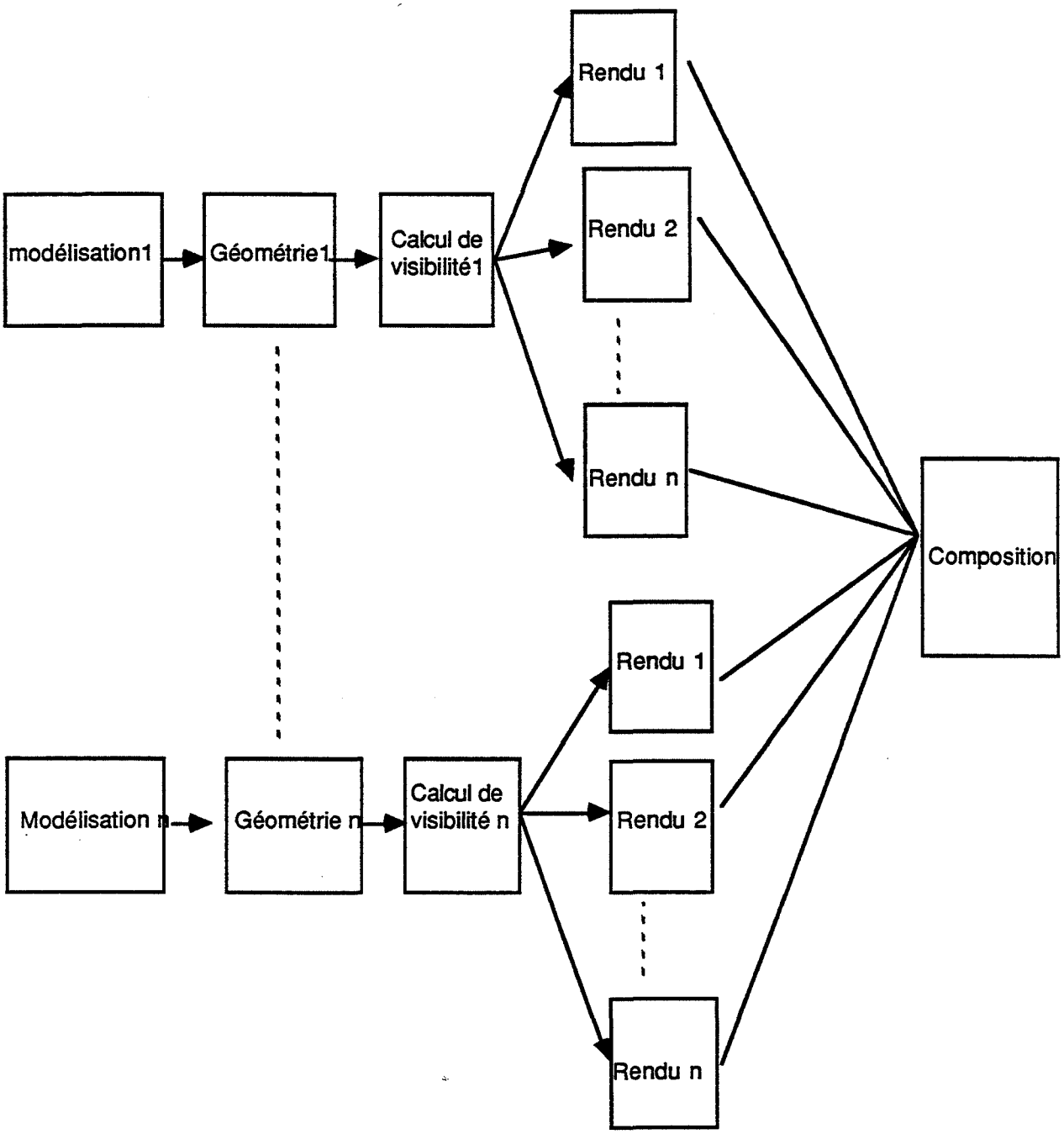


FIGURE 4

Pour nous aider à atteindre la *flexibilité* et l'*évolutivité* souhaitées pour notre système de synthèse d'images, nous avons utilisé la programmation orientée objet. Cette

dernière s'est largement répandue dans le monde informatique et a touché ses différents domaines. Après avoir décrit brièvement le système de synthèse d'images ILLUMINES dans le chapitre 1, nous discuterons, dans le chapitre 2, de l'utilisation de la programmation orientée objet pour la conception d'un système de modélisation. Nous étudierons comment on peut choisir les classes et concevoir la hiérarchie entre ces classes dans un modeleur pour atteindre la *flexibilité* et l'*évolutivité* souhaitées. Nous analyserons l'influence de ces choix sur nos objectifs. Enfin, nous expliquerons comment nous avons assuré un choix automatique de l'algorithme d'élimination des parties cachées, la méthode de lissage et le modèle d'éclairement pour chaque objet dans la scène.

Dans le chapitre 3, nous expliquerons le rôle de la boîte composition de la figure 4. Nous discuterons des avantages et des inconvénients d'une structure d'un système de synthèse d'images qui ressemble à celle de cette figure et dans lequel on fait un mixage d'algorithmes d'élimination des parties cachées. Nous proposerons alors des solutions pour certains de ces inconvénients.

Nous sommes souvent amenés à visualiser plusieurs fois une même scène avec des positions différentes de l'observateur, comme pendant une séquence d'animation par exemple. La gestion des niveaux de détails devient indispensable pour avoir un système de synthèse d'images efficace. La *flexibilité* que nous souhaitions pour notre système n'est pas limitée au concepteur ; nous avons voulu également rendre le système flexible pour l'utilisateur. Pour cela, nous avons intégré plusieurs outils et méthodes pour la gestion des niveaux de détails que nous décrirons dans le chapitre 4. Chacune des représentations d'un objet est visualisée et rendue par des algorithmes et méthodes différentes que le système juge les mieux adaptées pour elle, et sans que l'utilisateur s'en aperçoive.

Chapitre 1

Un système de synthèse d'images tridimensionnelles :

illumines

1 INTRODUCTION

Les recherches de l'équipe IMAGES de l'EMSE touchent aux différents domaines de la synthèse d'images, modélisation comme visualisation. Nos nouveaux algorithmes, nos idées se doivent d'être testés sur un logiciel dont nous soyons entièrement maîtres ; ainsi ILLUMINES ([Beig 90]) est né du fruit des travaux de l'équipe IMAGES de l'EMSE.

Ce chapitre se veut une présentation rapide, de ce système de synthèse d'images tridimensionnelles. Il se découpe en six parties :

- l'architecture d'ILLUMINES
- CASTOR : le modeleur 3D d'ILLUMINES
- la saisie de la scène
- les algorithmes de rendu
- Premiers constats

2 L'ARCHITECTURE D'ILLUMINES

ILLUMINES est un système de synthèse tridimensionnelle centré sur un modeleur du type arbre de construction : **CASTOR** (voir le paragraphe 3).

Plusieurs interfaces, plusieurs techniques de visualisation peuvent être utilisées. Le concepteur d'image peut ainsi visualiser rapidement, en cas d'interaction, ou au contraire produire une image présentant un certain réalisme (ombres, transparences, sources de lumière, ...) avec un temps de calcul, bien sûr, plus important.

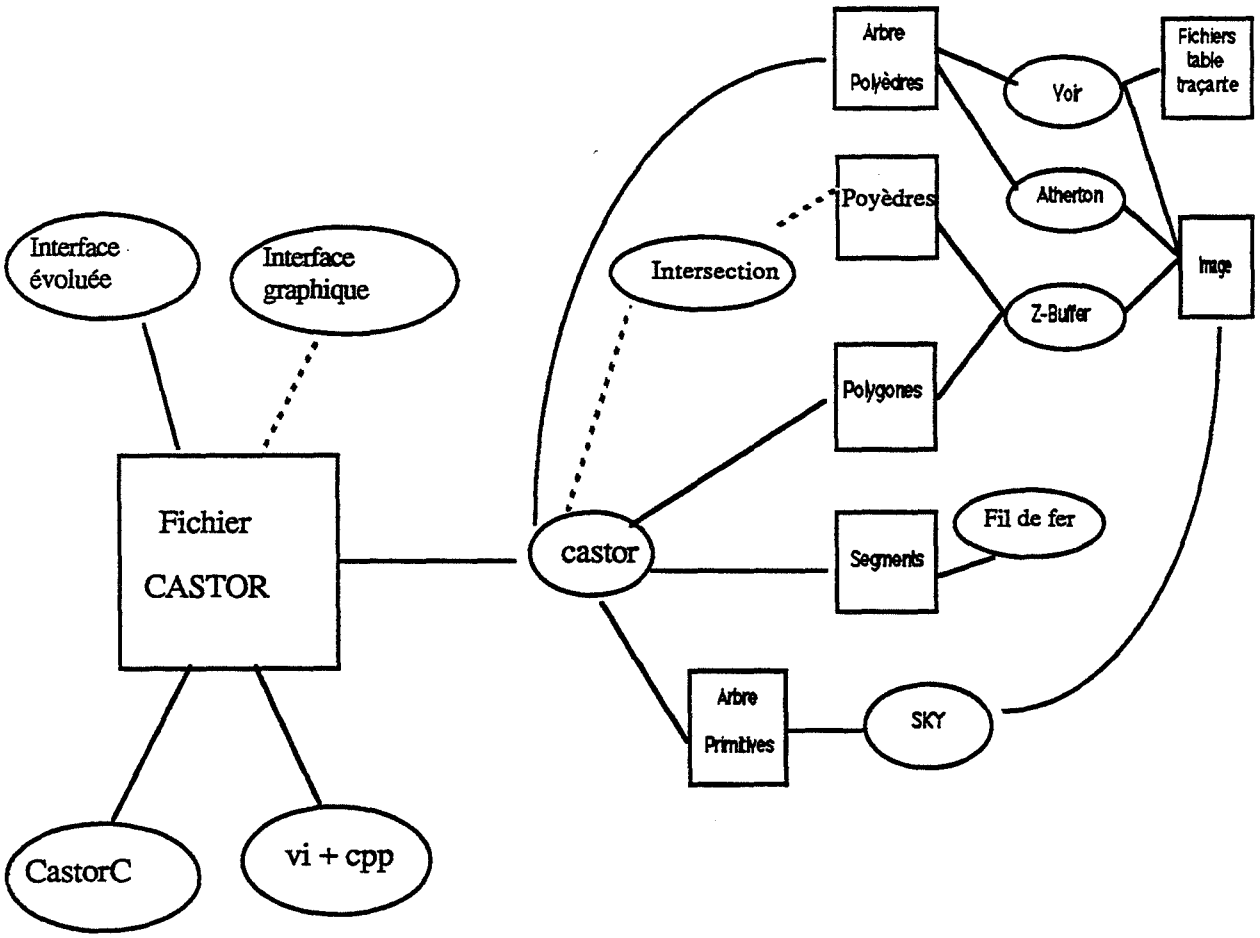


FIGURE 1.1

Ce système est de conception très modulaire et peut être représenté par le dessin de la figure 1.1, où les rectangles représentent des fichiers et les ellipses des modules de traitement (des processus UNIX). Les flèches en trait plein représentent ce qui existe à l'heure actuelle, les pointillés correspondent à des travaux en cours.

Le système complet peut être décomposé logiquement en quatre sous-ensemble :

- la partie saisie, constituée de l'interface *éditeur de texte*, de l'interface graphique, de l'interface évoluée et de castorC, une sur-couche construite avec la philosophie du langage LISP sur le modeleur ; cette partie sera décrite dans le paragraphe 4.

- un modeleur (CASTOR) de type arbre de construction qui sera décrit dans le paragraphe 3.

- la partie visualisation qui comporte deux familles d'algorithmes. La première est constituée du tracé de rayons, qui prend en entrée un arbre de primitives ; la seconde ne traite que des polygones. Pour cette deuxième famille d'algorithmes, le modeleur doit facettiser les primitives de l'arbre de construction, effectuer le coupage et la transformation perspective

pour fournir l'entrée voulue à l'un des algorithmes d'élimination des parties cachées. Ces algorithmes seront décrits dans le paragraphe 5.

- la partie rendu qui s'occupe du calcul de la couleur du pixel, en prenant en compte le type du matériau dont est constitué l'objet, les réflexions, les transparences, les ombres portées... Cette partie comporte plusieurs modèles d'éclairement et méthodes de lissage pour les algorithmes de visualisation travaillant sur des facettes. Nous la décrirons dans le paragraphe 6.

Les deux dernières parties, élimination des parties cachées et rendu, ne sont pas physiquement séparées dans l'implémentation actuelle du système.

3 LE MODELEUR CASTOR

Le maquettiste construit son *objet* en accolant des éléments de surface. Le sculpteur définit une silhouette à partir d'un élément volumique de base. Par analogie, la modélisation tridimensionnelle en synthèse d'images peut s'effectuer de deux façons possibles: par une modélisation surfacique où la scène est décrite par un ensemble de surfaces planes ou gauches (la maquette) ; par une modélisation volumique (la sculpture).

La modélisation surfacique décrit l'objet par une liste de faces (planes ou gauches). Son principal inconvénient est le manque d'information de nature topologique. La notion d'intérieur et d'extérieur n'existe pas. La cohérence entre les surfaces doit être gérée manuellement par le concepteur de l'image.

La modélisation volumique permet de distinguer l'intérieur de l'extérieur d'un solide. Les frontières d'un objet ne sont pas toujours connues explicitement dans toutes les modélisations volumiques. L'espace de modélisation étant l'espace affine euclidien de dimension trois, Requicha [Requ 80] précise les propriétés que doit vérifier une partie de R3 pour être susceptible de modéliser un solide réel.

3.1 La Modélisation Volumique

Plusieurs méthodes de modélisation volumique existent, dont les plus connues sont :

- **La représentation par les frontières** (Boundary REPresentation BREP) décrit explicitement les frontières d'un objet ([Baum 72], [Mant 82], [Ansa 85], [Mich 87]) ce qui constitue son avantage principal. Cette représentation n'est pas unique, un objet peut être représenté par différents assemblages de surfaces. Une BREP doit assurer qu'un ensemble de surfaces définit bien un volume qui a un intérieur et un extérieur. C'est alors que deux phénomènes étroitement liés interviennent : tout d'abord les éventuelles imprécisions

numériques peuvent perturber la cohérence du modèle ; ensuite l'utilisation des opérations booléennes. Ces dernières (notamment l'intersection et la différence) posent des problèmes de vérification de cohérence et augmentent les imprécisions numériques.

- **L'énumération spatiale** code les objets dans un espace borné et discret. L'espace ainsi défini, est une trame cubique 3D et la représentation la plus simple d'un solide consiste à fournir la liste des cellules qui sont occupées par le solide ; chaque cellule est appelé voxel. Le voxel ne peut prendre que deux valeurs : plein ou vide. L'espace est donc un tableau 3D de bits (Picture Element ARyaY : PEARY). Le PEARY ne pose strictement aucun problème de validité : la représentation est toujours cohérente ; les opérations booléennes se font de façon simple et efficace. Informatiquement ce tableau est codé et compacté sous forme d'un arbre octal (octree) [Same 84]. Cette représentation a aussi ses inconvénients :
- elle perd la continuité et les normales des surfaces. Ces dernières peuvent être reconstituées approximativement par une étude locale sur les voxels voisins ;
- l'acquisition, en l'abscence de l'existance préalable du solide et d'un moyen de saisie adéquat (saisie tomographique dans les applications médicales, ...), est rebutante.

- **La modélisation par arbre de construction** (Constructive Solid Geometry : CSG) [Requ 80] est une technique de modélisation qui combine des solides au moyen d'opérations ensemblistes. La structure associée est un arbre dont les noeuds sont des opérateurs booléens (union, intersection, différence) ou des transformations géométriques (affines : rotation, translation,...., ou non linéaires comme par exemple les déformations libres [Nie 89]) et les feuilles des objets élémentaires (voir la figure 1.2). La modélisation par arbre de construction n'est pas unique : un objet peut être en effet obtenu à partir de plusieurs arbres différents.

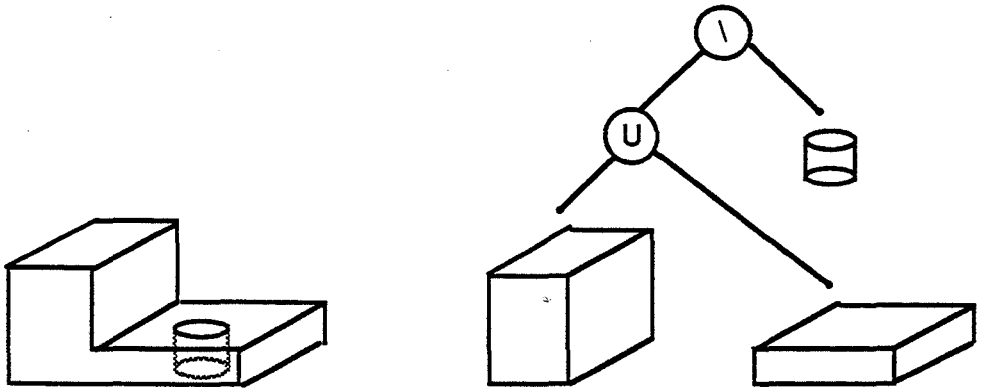


FIGURE 1.2

Diverses implémentations sont également possibles :

- les différents sous arbres de l'arbre de construction peuvent être partagés ; dans ce cas nous avons affaire à un graphe sans circuit à la place d'une structure d'arbre.
- les transformations géométriques peuvent se situer à différents niveaux de l'arbre. Certaines implémentations préfèrent appliquer les transformations au niveau le plus bas de l'arbre, à savoir sur chaque feuille, alors que d'autres les font porter sur tous les noeuds, et pas seulement les feuilles.Cette dernière solution permet de partager davantage de noeuds. L'implémentation de CASTOR (voir le paragraphe 3.2) utilise à bon escient le partage des noeuds.

L'arbre de construction fut la structure de modélisation tridimensionnelle choisie lors de la construction de notre modeleur. Elle présente des avantages dont :

- la structure arborescente permet d'attribuer des qualités à tout un sous arbre (textures, déformations, ...) ; par exemple il est possible d'effectuer des animations simples, ou l'ajustement de la position dans tout l'espace de tout un sous arbre, par la seule modification de paramètres de certaines transformations géométriques,

- cette description est essentiellement symbolique, elle peut être produite, manipulée, modifiée en utilisant des langages ou des techniques de manipulations symboliques : CASTOR en est un bon exemple.

L'utilisation de primitives géométriques comme objets élémentaires (cube, sphère, cylindre,...) fait de la modélisation par arbre de construction un outil bien adapté à la description de certaines pièces mécaniques ou d'autres objets manufacturés. Cette technique de modélisation est très utilisée pour la modélisation d'objets tridimensionnels, elle ne décrit toutefois pas explicitement l'objet par ses frontières ce qui peut être un inconvénient. L'espace des objets modélisables est très lié aux primitives et aux transformations qu'il est possible de leur appliquer. Ainsi toute modélisation naturelle (plantes, animaux, montagnes,...) peut s'avérer très délicate et nécessiter beaucoup de manipulations pour un résultat souvent décevant. Enfin, L'arbre de construction ne fait que décrire le processus de construction de l'objet ; il ne les effectue pas. Connaître la frontière ou le volume de l'objet ne sont donc pas choses évidentes.

Enfin, il existe des méthodes qui permettent d'effectuer des conversions d'une représentation à une autre. De telles conversions sont très interessantes car chaque technique de modélisation possède des avantages et des inconvénients. Il n'existe pas de technique universelle qui s'adapte à toutes les applications, mais chacune peut être plus ou moins mieux adaptée à un certain types d'application.

3.2 Castor

Comme nous l'avons déjà précisé dans le paragraphe précédent, CASTOR ([Beig88]) est un langage de description d'une modélisation du type arbre de construction. Il permet de décrire des objets sous forme textuelle dans ce mode de représentation.

Au sein de CASTOR se trouve un interpréteur : *castor*. Son rôle premier est d'effectuer une analyse lexicale, syntaxique, sémantique d'un fichier. Tout objet est représenté par un identificateur alphanumérique. Lors de l'analyse sémantique, une table des identificateurs est tenue à jour. Ainsi, on peut retrouver le pointeur sur le graphe qui définit un objet.

La notion de pointeur permet d'utiliser un objet, une fois celui-ci défini, pour la définition d'autres objets. Les graphes obtenus sont des graphes sans circuit, dont la racine est accessible à travers un identificateur. Un parcours en profondeur de l'arbre permet alors d'atteindre toutes les feuilles, ou objets élémentaires. L'exemple des figures 1.3 (a) et 1.3 (b), tirée de [Beig 88] illustre bien cette représentation.

Supposons que les objets *roue* et *carrosserie* aient déjà été définis.

```
voiture = $U(
        carrosserie,
        @t(X1,Y1,0) roue,
        @t(X2,Y1,0) roue,
        @t(X2,Y2,0) roue,
        @t(X1,Y2,0) roue
        );
```

Une voiture est définie comme la réunion ($U) de la *carrosserie* et de quatres *roues* positionnées grâce aux translations (@t(...)) qui utilisent les paramètres numériques X1,Y1,X2,Y2.

FIGURE 1.3 a

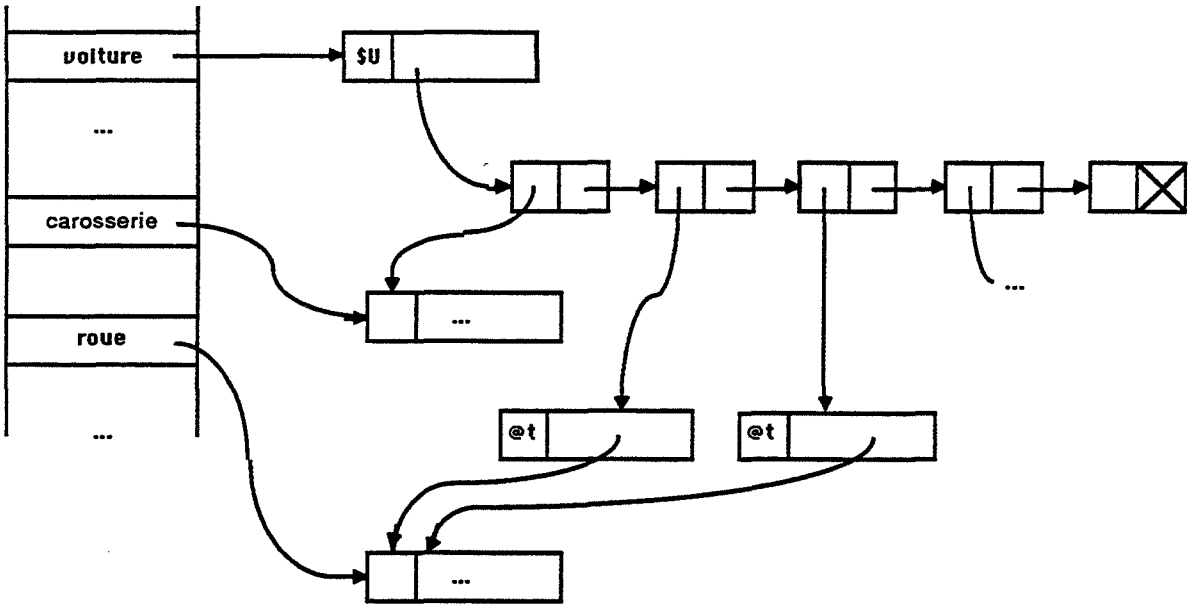


FIGURE 1.3 b

L'utilisateur crée un fichier CASTOR comprenant des données concernant la description de la vue (position de l'œil, point de visée, profondeur de champ, intensité ambiante,...) et une description de la scène. La deuxième fonction de *castor* est de transformer les descriptions des objets en liste de polygones : tout objet précédé de la commande '>' est facettisé.

4 LA SAISIE DE LA SCENE

La partie saisie du système ILLUMINES est la première étape avant l'entrée en jeu du programme CASTOR. Elle est constituée de plusieurs modules indépendants entre eux (chacun correspond à une possibilité offerte à l'utilisateur) . Ces modules ont pour seul point commun de créer le flot d'entrée à fournir au programme CASTOR, sous la forme d'un fichier.

La première des interfaces est tout simplement l'éditeur de texte. L'utilisateur crée donc son propre fichier avec l'éditeur de texte *vi*, éditeur pleine page d'UNIX. L'apport du préprocesseur C permet une écriture ainsi qu'une lecture de fichiers *castor* assez fonctionnelles. Grâce à lui en effet, l'utilisateur peut modulariser la description d'une scène complexe

- d'une part, en la séparant en plusieurs fichiers puis en réunissant ces différents fichiers avec des directives include,

- d'autre part, en utilisant des macros définitions permettant

• de s'affranchir des caractères $,!,@ peu lisibles

```
# define union $U
# define inter  $I
```

• de définir des classes d'objets paramétrés qui apparaissent plusieurs fois avec des valeurs de paramètres différents.

CASTOR, malgré l'apport du préprocesseur C, ne possède ni structures de contrôle, ni notion de paramètres. Dans le but de fournir un véritable langage de programmation, une sur-couche de CASTOR a été écrite : CastorC ([Mich 89]). Cette sur-couche rend accessible, depuis C, les fonctionnalités de CASTOR, avec toute la puissance du langage initial utilisé. On peut fournir comme exemple la fonction répétition qui permet de répéter un objet en lui appliquant à chaque instant une itération supplémentaire d'une certaine transformation. Ainsi un escalier peut être défini comme indiqué sur la figure 1.4. Ainsi, l'utilisateur décrit la scène, en langage C, dans un fichier, que CastorC prend en entrée pour fournir un fichier CASTOR en sortie.

```
marche = bloc (-larg_es/2.,larg_es/2.,-Eg_co2,prof_m,0.,-ep_m,CSOL).;
escalier = reunion (
        marche,
        repetition ( nb_m,
                marche,
                trans ( 0.,prof_m,_)),
                fin) ;
```

FIGURE 1.4

L'inconvénient de ces interfaces est qu'elles ne sont pas interactives, puisque l'utilisateur ne peut pas visualiser l'objet qu'il est en train de décrire. Le deuxième type d'interface, disponible dans le système ILLUMINES, permet une saisie interactive des scènes [Amar 89]. Au fur et à mesure de la manipulation, il y a visualisation en fil de fer, sans élimination des parties cachées, de ce que l'on construit. L'utilisateur a à sa disposition des menus, d'icônes, de multifenêtrage, afin de bâtir l'arbre de construction. En sortie de ce module, il y a création d'un fichier à la syntaxe CASTOR qui peut donc être traité par la suite des modules du système.

Vue l'importance de la saisie en synthèse d'images, une interface possédant des opérations de haut niveau comme *poser_sur, coller_contre, grossir, déplacer_vers*, ... a été développée pour faciliter l'utilisation de CASTOR. C'est une interface *évoluée*, c'est à dire capable de réagir aux demandes de manipulation de l'opérateur. Cette interface a été conçue et implémentée par Gilles Fertey en langage KOOL développé par BULL [Fert 90].

La figure 1.5 résume les deux paragraphes 3.2 et 4.

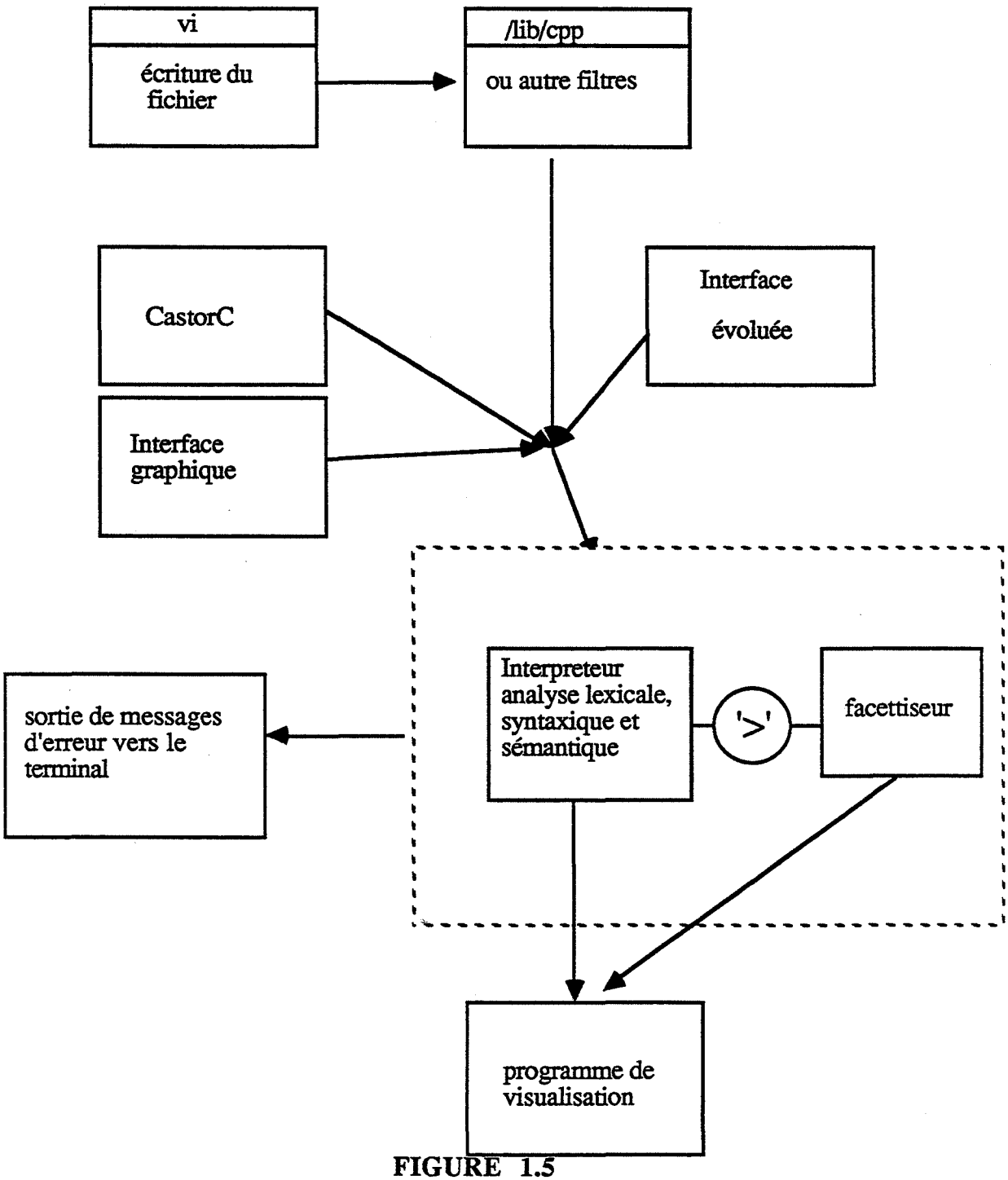


FIGURE 1.5

5 LES ALGORITHMES DE VISUALISATION

ILLUMINES propose à l'utilisateur plusieurs possibilités de visualisation. Il est important de rappeler que c'est indépendamment des possibilités offertes que l'utilisateur aura, au préalable, créé un fichier CASTOR (voir le paragraphe 3.2). Voyons maintenant, plus en détail, les cinq procédés de visualisation possibles.

5.1 Visualisation En Fil De Fer

Visualisation classique en fil de fer où le système se contente d'envoyer des segments au format demandé par le périphérique qui va créer l'image. Nous pouvons ainsi visualiser en fil de fer sur divers terminaux graphiques (LEXIDATA, TEKTRONIX, SUN, APOLLO,...).

5.2 Visualisation Par Le Tampon De Profondeur

L'algorithme du tampon de profondeur, ou tampon en Z (Z-BUFFER) génère, pour chaque pixel, deux valeurs : une couleur et une profondeur ; d'où le nom de l'algorithme. Cette profondeur est la distance à l'œil de l'objet vu en ce pixel. Les profondeurs étant initialisées à la plus grande valeur possible, on détermine pour chaque élément de la scène, dans un ordre quelconque, les pixels de l'image en lesquels il se projette. En chacun de ces pixels, la profondeur de l'élément est calculée ; si elle est inférieure à la profondeur actuelle en ce pixel, alors l'élément est plus proche de l'œil que celui qui avait été déterminé auparavant : la profondeur et la couleur de l'élément sont affectées au pixel. Ce traitement, effectué pour tous les éléments de la scène, donne la valeur des pixels de l'image finale.

```
Tampon-en-Z (scène, œil)
{
  pour chaque pixel (i,j) faire Z[i,j] <- infini ;
   pour chaque élément E de la scène faire
    pour chaque pixel (i,j) où E se projette faire
     {
       z <- profondeur de E en (i,j) ;
       si (z < Z[i,j])
         alors {
                 Z[i,j] <- z ;
                 couleur[i,j] <- couleur de E en (i,j) ;
                }
     }
}
```

L'inconvénient majeur du tampon de profondeur est qu'il est générateur d'images d'un réalisme assez pauvre : pas de transparence, des défauts d'aliassage dus à la visualisation des polygones sans aucun tri préalable (des travaux ont été réalisés dans l'équipe pour remédier à ce défaut [Ghaz 85], [Ghaz 90])

Son principal avantage est sa simplicité, ce qui a permis des implantations matérielles tout à fait performantes.

Une implémentation a été faite de cet algorithme, de plus, nous disposons d'une version microprogrammée du tampon de profondeur sur notre mémoire d'images (LEXIDATA).

5.3 Visualisation Par Balayage De Ligne

Les algorithmes de cette famille s'appuient sur le principe du balayage d'un écran de type télévision suivant des lignes horizontales. Ils décomposent donc la scène à visualiser par des plans perpendiculaires à l'écran passant par les lignes de balayage (voir la figure 1.6). Ce principe a été introduit par Wylie, Rommey, Evans et Erdahl [Wyli 67] pour des scènes composées de facettes polygonales planes.

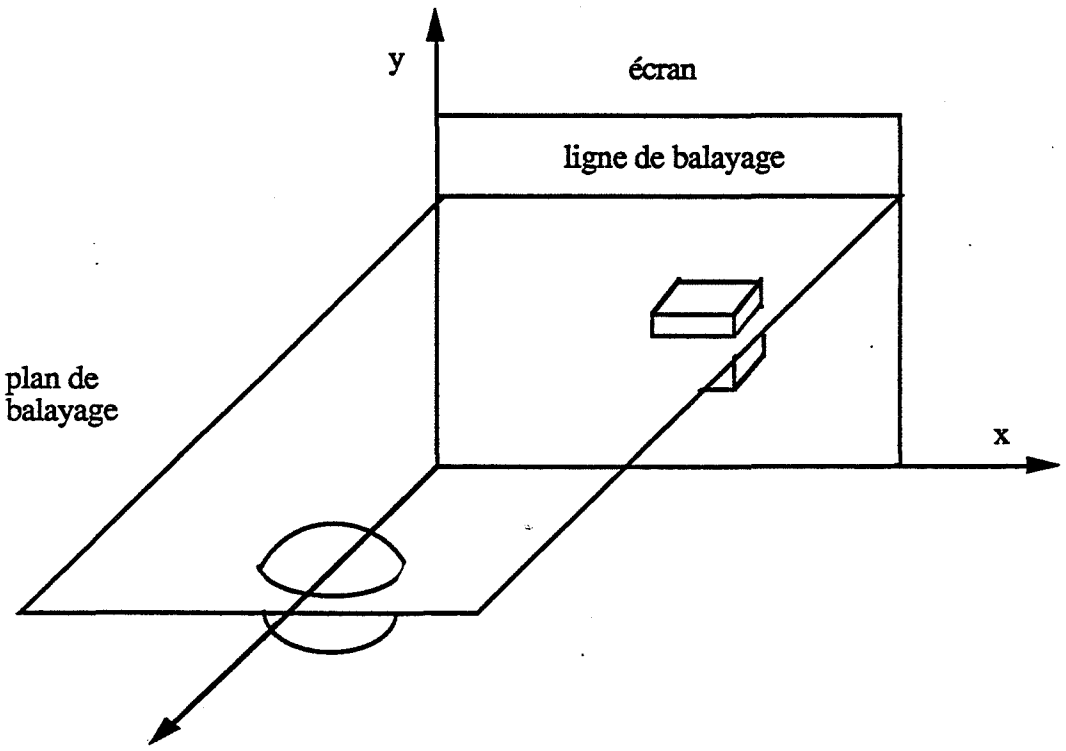


FIGURE 1.6

Atherton ([Athe 83]), a proposé un algorithme utilisant ce principe pour des objets définis à l'aide d'un arbre de construction. Comme nous utiliserons cet algorithme dans le chapitre 3, nous allons brièvement rappeler le principe de cet algorithme et les particularités de son implémentation dans le laboratoire.

5.3.1 Principe De L'algorithme D'Atherton

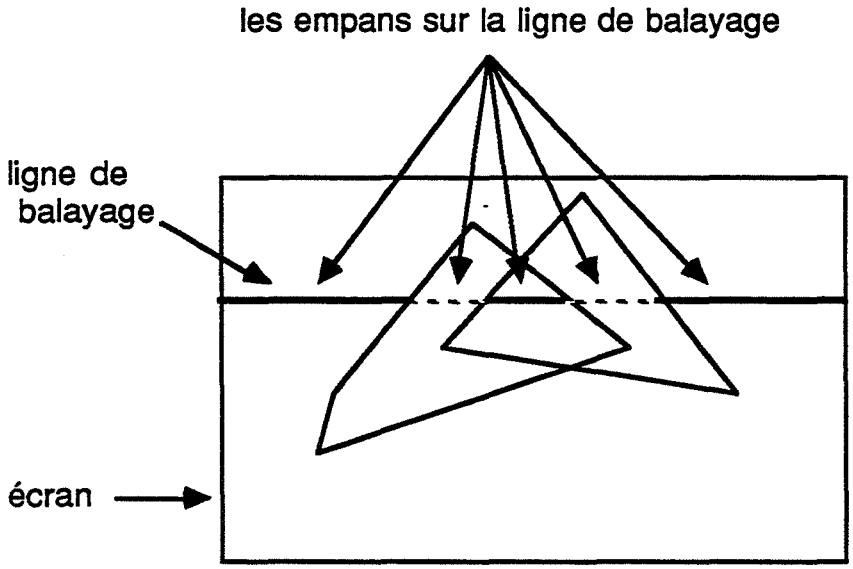


FIGURE 1.7

L'auteur propose un algorithme qui est une extension de celui de Watkins [Watk 70] pour des scènes modélisées par un arbre de construction. On peut le résumer par la procédure suivante :

1 - { Pour chaque arête de polygone faire
trier les sommets selon l'axe des y ;
/ ce tri suivant le sens de balayage de l'écran a pour but de faciliter le calcul ultérieur des plans de balayage */

2 - Pour chaque ligne de balayage faire
{ Pour chaque polygone actif faire
déterminer l'intersection du plan de balayage et du polygone appelée segment;
Trier ces segments suivant x ;
Déterminer la frontière des empans ;

/* un empan est une portion continue de la ligne de balayage comprise entre deux extrémités voisines de segments (cf figure 1.7), tous les pixels appartenant à un même empan possèdent la même liste d'objets et dans le même ordre. */

```
        Pour chaque empan faire
          {
           Couper les segments actifs par les frontières des empans ;
           Si les segments ont même z-ordre à la frontière alors
             Afficher le segment
           Sinon
              Subdiviser l'empan aux intersections de segments et revenir en 3 pour
               chaque sous-segment ;
            }
      }
}
```

5.3.2 Version Programmée A L'EMSE

Nous avons implanté l'algorithme d'Atherton qui est bien adapté à un modeleur du type arbre de construction et présente des temps de calculs acceptables. Dans cette implémentation, pour chaque ligne de balayage, nous extrayons de l'arbre de construction décrivant la scène, un arbre de construction simplifié. Ce dernier est obtenu en enlevant les objets de la scène qui ne se projettent pas sur cette ligne, et ceux qui s'y projettent sans faire partie de la scène à cause des opérations booléennes (figure 1.8).

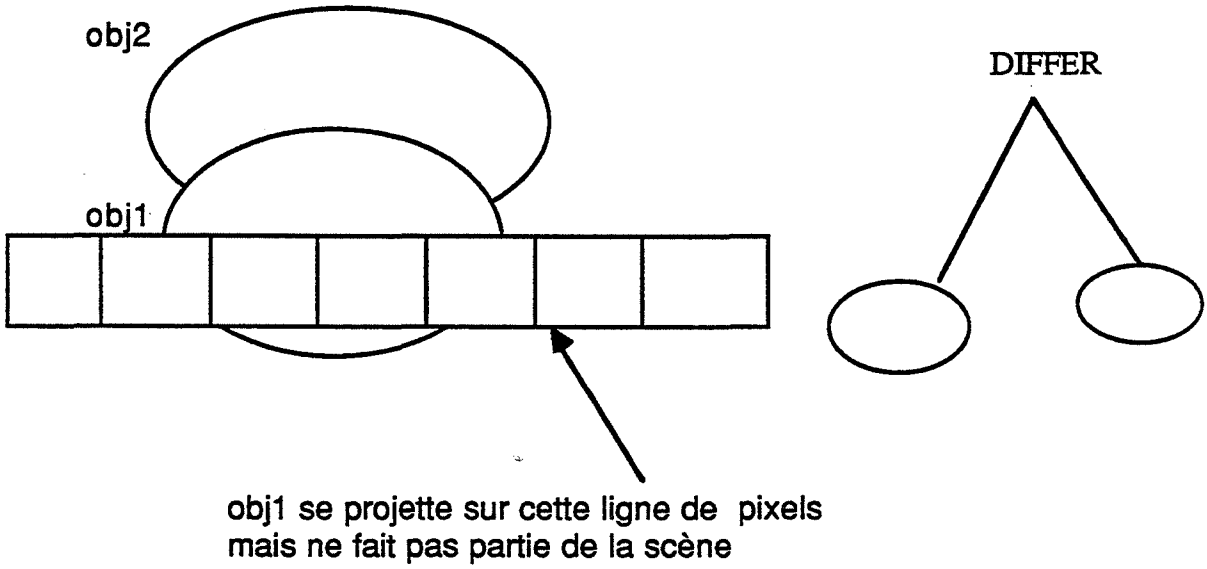


FIGURE 1.8

Pour déterminer le premier point visible en un pixel, il suffit de trouver le point dont la profondeur est la plus petite en balayant une seule fois la liste des points vus en ce pixel. Si l'arbre simplifié contient des opérations booléennes -intersection ou différence-, alors le

point le plus proche de l'œil peut ne pas appartenir à la scène, il faut donc rebalayer cette liste jusqu'à ce que l'on obtienne un point appartenant à la scène. Dans ce cas nous avons préféré trier la liste en profondeur par une méthode de tri rapide. On aura donc, pour chaque pixel, une liste de points vus triée en profondeur ou non selon que l'arbre simplifié sur la ligne courante contient des opérations booléennes ou non. Tous les calculs qu'effectue la méthode d'Atherton se passent dans un repère tel que l'axe des z ait même direction et même sens que le regard, et que le plan xoy soit le plan de l'écran.

D'autre part, dans notre implémentation de l'algorithme d'Atherton, on utilise le modèle empirique de Phong ([Phon 75]) pour le calcul de l'éclairement :

$$E = K_a + [K_d*(L.N) + K_s*(R.V)^n \ ) \ ] E_i$$

N est le vecteur normal en un point de la surface ;
V est le vecteur unitaire de visée ;
L est le vecteur unitaire de direction de la lumière ;
R est le vecteur unitaire réfléchi ;
K_a est le coefficient simulant la lumière ambiante ;
K_d et Ks sont respectivement les cœfficients de diffusion et de spécularité qui dépendent de l'objet ;
n est un coefficient de brillance propre au matériau dont est formé l'objet.

D'autre part, les deux méthodes de lissage de Gouraud [Gour 71] et de Phong [Phon 75] sont utilisés par l'algorithmes d'Atherton. La première consiste en une interpolation des intensités, la seconde en une interpolation des normales qui est plus lente mais qui donne des meilleurs résultats permettant de prendre en comptes les effets de specularités des surfaces.

Par rapport au papier original d'Atherton, nous avons ajouté la gestion des transparences : si le point visible appartient à un objet transparent, on cherche le second point appartenant à la scène dans la liste des points vus en un pixel. Ce dernier est alors pris en compte en mélangeant les couleurs de ces deux points et en utilisant le coefficient de transparence propre à l'objet pour simuler les effets de transparence sur la couleur du deuxième point. Ce traitement est appliqué récursivement dans le cas où ce dernier appartient, lui aussi, à un objet transparent.

D'autre part, notre implémentation de l'algorithme d'Atherton utilise la méthode de Williams [Will 78] pour le calcul des ombres portées. Cette méthode opère en deux étapes. La première est le calcul d'une image depuis la source lumineuse. Les profondeurs des objets les plus proches de la lumière sont stockés dans une carte de profondeur. Seule la profondeur est importante et il est inutile d'effectuer des calculs de rendu pour cette carte de profondeur. Les objets présents sur la carte de profondeur sont les objets de la scène éclairés par la source lumineuse. Lors de la deuxième étape, l'image est calculée depuis le point de vue de l'observateur. Chaque point visible est transformé, par un calcul matriciel approprié, dans le repère de la source lumineuse. La profondeur du point transformé (c'est à dire la

distance de ce point à la source, dans le repère de la source) est comparée à celle contenue dans la carte de profondeur, calculée lors de la première étape. Si le point transformé est derrière le point correspondant de la carte de profondeur, le point est à l'ombre ; sinon il est éclairé par la source. Dans notre implémentation de cette méthode [Hano 88], nous calculons un pourcentage d'ombre pour chaque pixel pour obtenir des ombres plus douces.

Enfin, notre implémentation de l'algorithme d'Atherton a recours à un sur-échantillonnage globale de tous les pixels pour la résolution des problèmes d'aliassage.

5.4 L'algorithme VOIR

Un algorithme d'élimination des parties cachées s'appliquant à un arbre de construction dont les primitives sont facettisées a été développé par un membre de l'équipe ([Mich 87]) et implanté. A noter les effets possibles de transparence ainsi que la résolution du phénomène de l'aliassage.

5.5 Visualisation En Tracé De Rayons

Introduit en 1968 par Appel ([Appe 68]), le tracé de rayons prend naissance véritablement en 1980 à la suite des travaux de Whitted ([Whit 80]) qui réalisa des images présentant des effets optiques inédits de réflexion et d'ombrage avec des objets opaques ou transparents. L'algorithme, simple et général, prend en compte la plupart des effets optiques: réflexions multiples, transparences, réfractions, sources lumineuses multiples. Depuis, de nombreux travaux, en quête d'un plus grand réalisme de l'image synthétique, ont été mené à bien et notamment au sein de notre équipe ([Arge 88a], [Arge 88b], [Fert 89]). Afin de faciliter la lecture et la compréhension du chapitre 3, nous allons brièvement rappeler le principe général du tracé de rayons et présenter SKY ([Arge 88a]), le tracé de rayons disponible actuellement à l'école.

5.5.1 Principe De L'algorithme Du Tracé De Rayons

Le tracé de rayons calcule l'image pixel par pixel. Pour chaque pixel, on désire calculer une couleur. Dans la réalité, il s'agit de la couleur qui arrive à l'œil par un rayon lumineux. Le tracé de rayons effectue le parcours inverse des rayons lumineux, tout en prenant en compte la plupart des lois de l'optique géométrique. Le principe de l'algorithme est simple et peut être récapitulé dans l'algorithme suivant, illustré par la figure 1.9.

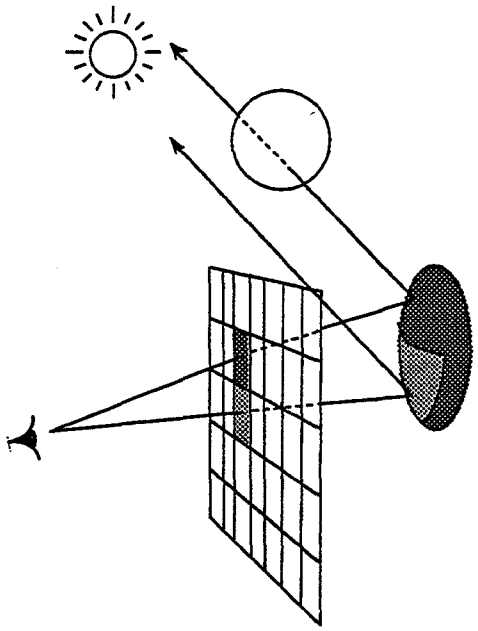


FIGURE 1.9

POUR chaque pixel de l'écran FAIRE
    définir le rayon primaire œil-pixel
    POUR chaque objet FAIRE
        tester les intersections rayons-scène
        trouver l'intersection la plus proche de l'œil
        relancer des rayons vers les sources lumineuses
        calculer l'éclairement du point d'intersection
        affecter la couleur au pixel
    FINPOUR
FINPOUR

Par chaque pixel passe un rayon, dit primaire, qui part de l'œil dans la direction de la scène. Il faut qu'il renvoie une couleur. Cette couleur résulte de la couleur de l'objet touché en premier par le rayon (l'objet vu par l'œil) et de divers phénomènes optiques.

La première étape consiste donc à chercher l'objet vu. Les points d'intersection du rayon œil-pixel avec tous les objets de la scène sont calculés, et le plus proche de l'œil est retenu, soit le point P.

Une fois l'objet vu trouvé , le calcul de son éclairement permet de trouver une couleur qui sera affectée au pixel. Cette couleur est dûe, nous l'avons dit, à plusieurs causes :

- *ombrage* : la couleur propre de l'objet et le calcul d'ombrage au point vu sur ce dernier.

- *ombre portée* : l'objet vu peut être caché de la source de lumière par un autre objet, ce dernier lui fait alors de l'ombre. Pour détecter ce cas, des rayons secondaires, dit rayons d'ombre, sont relancés dans la scène, issus du point d'intersection et dirigés vers chaque source lumineuse. Toute intersection avec un autre objet opaque de la scène permet de conclure que l'objet n'est pas éclairé par la source lumineuse considérée. On peut lancer plus d'un rayon d'ombre vers la même source de lumière quand celle-ci n'est pas ponctuelle pour pouvoir prendre en compte les effets de pénombre. On récupère alors un pourcentage d'ombre au lieu d'obtenir un point qui est soit à l'ombre soit éclairé.

- *reflexions et transparences* : l'objet vu peut avoir une surface réfléchissante et/ou transparente ; sa couleur dépend alors des autres abjets de la scène, ceux qui se reflètent sur lui et qu'on voit à travers de lui. Un rayon secondaire issu de P, dit rayon réfléchi ou réfracté, est donc relancé dans la scène. Le rayon réfléchi possède une direction symétrique à celle du rayon incident (le rayon primaire) par rapport à la normale de l'objet en P (loi de Descartes). Le rayon réfracté possède une direction qui dépend de l'indice de réfraction de l'objet et de celui du milieu dans lequel se trouve ce dernier (loi de Snell). Le processus est relancé : le rayon secondaire est testé avec tous les objets de la scène afin de connaître la couleur vue dans sa direction. La détermination de cette couleur est exactement la même que pour le rayon primaire : le point P' de l'objet que heurte le rayon secondaire peut lui aussi être cachée des sources lumineuses, et êrte transparent ou réfléchissant ; des rayons tertiaires sont donc lancés... On définit un niveau maximum de reflexion puisque les rayons transportent de moins en moins d'énergie lumineuse, et au bout d'un certain temps, il devient inutile de lancer d'autres rayons.

Si l'algorithme de tracé de rayons permet d'obtenir des images très réalistes, il possède quelques défauts :

- le temps de calculs nécessaires est important, Le réalisme obtenu se paye cher.

- l'ensemble des calculs effectués dépendent de la position de l'œil ; il faut recommencer tous ces calculs lors d'un changement de la position de l'observateur.

- le tracé de rayon ne propose pas de solutions pour simuler les interréflexions entre les objets dans la scène ; pour cela, il faut utiliser une autre méthode comme la radiosité [Gora 84].

La qualité première de l'algorithme du tracé de rayons tient dans la simplicité de sa méthode : les seuls calculs géométriques nécessaires sont ceux de l'intersection entre une demi-droite et un objet. Il en ressort :

• que le tracé de rayons est bien adapté à la modélisation par arbre de construction. La procédure d'intersection d'un noeud A se traduit par une comparaison des résultats récursifs des procédures d'intersection sur son fils gauche et son fils droit, tout en prenant en compte les éventuelles opérations booléennes. La procédure intersection d'un nœud union peut s'écrire :

```
procedure_intersection (A)
{
si A = feuille de l'arbre alors
        intersection_primitive (A) ;
 sinon
{
        procédure_intersection(A_filsGauche) ;
        procédure_intersection(A_filsDroit) ;
        prendre l'intersection  la plus proche
}
}
```

• que le tracé de rayons nécessite n procédures d'intersection, parfaitement indépendantes, relatives aux n types d'objets disponibles dans le modeleur. Toute extension (ajout, retrait, modification d'un type d'objet) s'effectue aisément respectivement en ajoutant, supprimant ou modifiant la procédure d'intersection associée au type d'objet.

5.5.2 SKY

SKY est une implantation de l'algorithme du tracé de rayons dans le cadre d'une modélisation par arbre de construction (pour plus de détail, se reporter à la thèse [Arge 88a]).

En plus de tous les effets de rendu classiques possibles avec cette méthode (réflexions multiples, transparences, réfractions, ombres portées, textures,...) SKY propose

• trois optimisations spécifiques à chacun des types de rayons

- pour les rayons primaires : par un algorithme original de balayage du plan, cette méthode détermine les intersections entre les rectangles englobants (à coordonnées entières, et dont les cotés sont parallèles aux bords de l'écran) calculés à partir de la projection des boîtes englobantes sur le plan de l'écran ; ainsi pour chaque zone rectangulaire de l'écran, une estimation de la scène minimale s'y projettant est effectuée.

- pour les rayons d'ombre : les boîtes englobantes des objets sont projetées sur un écran virtuel dans le cas du soleil (considéré comme étant à l'infini), ou sur six écrans virtuels parallèles aux trois plans définis par les axes (toutefois si la source se situe hors de la boîte englobante de la scène, un seul plan suffit) ; une technique similaire à la précédente est ensuite adoptée avec en plus la notion de portée maximale pour chaque source lumineuse ; au delà de cette portée, la boîte englobante n'a pas besoin alors d'être projetée.

- pour les rayons de réflexion/réfraction : la solution choisie consiste à prendre trois des plans de la boîte englobante de la scène ; sur chacun de ces plans, on projette les boîtes englobantes des objets de la scène, et on définit un quadtree ; chaque cellule du quadtree contiendra une sous-scène CSG ; pour chaque rayon, on détermine le plan (et donc le quadtree) qui réduira au mieux le nombre de primitives à tester : celui dont la projection du rayon traverse le moins de cellules.

• une amélioration des temps de calcul d'intersection entre un rayon et un arbre de construction : par un balayage local, chaque primitive hérite, dans son repère local, de la position de l'œil et de la position de l'écran ; le passage du rayon dans le repère local est ainsi simplifié ; cela nécessite cependant la connaissance de la matrice de transformation au niveau des feuilles de l'arbre.

• Une méthode originale de résolution du phénomène de l'aliassage ([Arge 88b]) a été utilisée dans le tracé de rayons SKY : un algorithme local et adaptatif sur-échantillonne uniquement les pixels présentant un problème. Cette solution nécessite plusieurs informations en chaque pixel :

- le numéro de la primitive intersectée,

- le numéro de la surface C1 intersectée (à chaque surface C1 de chaque primitive est associée un numéro),

- la liste des pointeurs sur les objets intersectés.
Dans le cas d'un objet simple (un noeud dont les ancêtres sont uniquement des unions), un pointeur sur le premier noeud suffit ; s'il existe au moins un opérateur différence (ou intersection), toutes les primitives considérées sont à garder dans la liste,

- un drapeau qui indique si la primitive est éclairée par une source lumineuse ou pas,

- la liste des pointeurs sur les objets simples faisant de l'ombre pour une source lumineuse.

L'antialiassage est effectué par sur-échantillonage des pixels autres que ceux dont les huit voisins ont le même numéro de primitive, le même numéro de surface et sont éclairés par les mêmes sources lumineuses. Pour chacun des pixels sur-échantillonnés, une sous-scène pour les rayons primaires est construite, une autre pour les rayons vers les sources lumineuses, et une autre pour les rayons de réflexions,réfractions.... La couleur du pixel considéré est alors la moyenne des valeurs des neuf sous-pixels.

6 PREMIERS CONSTATS

Illumines est constitué de plusieurs parties susceptibles d'être modifiées, améliorées.

Le modeleur CASTOR représente une véritable entité, une réécriture est actuellement en cours de développement, où les transformations affines sont étendues aux déformations libres. ([Nie 89]). Programmé dans un environnement UNIX, CASTOR prend en fait l'entrée standard comme fichier d'entrée, et envoie la liste des facettes sur la sortie standard. De ce fait, il agit comme un filtre au sens de UNIX ([Bour 83]). Le flot d'entrée doit contenir une description des différents objets constituant la scène selon la syntaxe détaillée dans [Beig 88]. D'autres filtres Unix peuvent traiter le fichier d'entrée avant de le passer à CASTOR. Le premier utilisé fut le préprocesseur du langage C : /lib/cpp.

La partie visualisation propose plusieurs algorithmes de visualisation dont les algorithmes d'Atherton et de tracé de rayons que nous allons utiliser dans le chapitre 2. Ces algorithmes reçoivent en entrée standard la sortie de CASTOR. Pour réaliser une image il faut ainsi placer plusieurs filtres l'un à la suite de l'autre de la façon suivante :

fichier_castor -> /lib/cpp | castor | atherton | rendu -> *fichier_RVB*. (1)

Cette utilisation des tubes UNIX n'est pas sans inconvénient. En effet, le flot de données se dirige dans un seul sens ; il n'existe aucun moyen pour que les données circulent

dans les deux sens à travers le tube. Cela peut être assez gênant pour la réalisation des interfaces interactives graphiques par exemple.

Comme nous l'avons signalé dans l'introduction, il est souhaitable qu'un système de synthèse d'images permette l'utilisation de plusieurs algorithmes d'élémination des parties cachées, des méthodes de lissage et des fonctions d'éclairement dans la même scène (cf figure 4). Or, lorsque j'ai commencé mes recherches à l'EMSE, le système ILLUMINES ne permettait que d'avoir des pipelines figés comme en (1). Mes travaux ont consisté à introduire de la flexibilité et de l'évolutivité dans le système comme le montreront les chapitres suivants.

Chapitre 2

Un modeleur flexible et évolutif orienté objet

1 INTRODUCTION

Dans le chapitre précédent, nous avons décrit le modeleur CASTOR et les différents moyens de saisie existant dans le système ILLUMINES. Dans ce chapitre nous allons décrire un modeleur orienté objet, écrit en langage C++, tout à fait indépendant de CASTOR, et que nous avons appelé CastorC++. Ce dernier reçoit en entrée un fichier, décrivant la scène, écrit en C++. Il ressemble beaucoup à la couche CastorC décrit dans le chapitre 1. La différence est que CastorC++ ne produit pas de fichier *castor* comme le fait CastorC, mais fournit directement en sortie les bases de données nécessaires pour chaque algorithme de visualisation, sans passer par CASTOR.

Tout au long de la conception et de l'implémentation de castorC++, nous avons voulu favoriser au maximum l'évolutivité et la flexibilité de ce modeleur. Ces deux caractéristiques nous paraissent d'une importance capitale pour un système de modélisation. En effet, de très nombreux algorithmes d'élimination des parties cachées et de rendu ont été proposés dans la littérature de la synthèse d'images ([Gour 71], [Suth 74], [Phon 75], [Blin 77], [Cook 82], [Gora 84], [Pero 88]). Ces algorithmes diffèrent de par le type des primitives auquelles ils s'appliquent, le type de modélisation de la scène accepté, le degré de réalisme souhaité, le temps de calcul admis,... Pour pouvoir implémenter et évaluer ces différentes techniques, nous avons voulu rendre notre système le plus flexible possible en permettant l'utilisation dans une même scène de plusieurs algorithmes d'élimination des parties cachées ou de rendu (lissage, modèle d'éclairement, antialiassage, ...). De même nous avons voulu qu'il soit évolutif et qu'il nous permette d'effectuer facilement des modifications, d'ajouter de nouveaux algorithmes et de nouvelles méthodes, pour pouvoir les tester et les comparer.

On trouve ces exigences de flexibilité et d'évolutivité pour un système de synthèse d'images dans de nombreux articles . Dans [Nada 87], les auteurs présentent GRAPE, un environnement flexible pour la synthèse d'images. Les différents modules de ce système, appelés nœuds, sont interconnectés et peuvent être structurés sous la forme d'un graphe acyclique. Cet approche permet, d'une manière aisée, l'assemblage de plusieurs nœuds de différentes façons, ce qui permet de concevoir et d'essayer des architectures variées pour le système de synthèse d'images. GRAPE est implémenté sous le système d'exploitation UNIX [Bour 83], les nœuds sont écrits en C, et la communication entre ces derniers est décrite en utilisant un macro-language que les auteurs ont définis. En présentant GRAPE, les

auteurs ont déclaré deux buts à atteindre : *flexibilité* et *réutilisabilité*. La flexibilité doit permettre d'essayer de nouveaux concepts et méthodes sans modifier le code existant. La réutilisabilté doit faciliter l'utilisation des modules existants pour pouvoir effectuer de nouveaux traitements. La flexibilité et la réutilisibilité sont deux caractéristiques de la programmation orientée objet. En utilisant la programmation classique, GRAPE ne profite pas de ces deux avantages de la programmation par objet et reste limité par rapport aux buts qu'il a déclarés vouloir atteindre.

Cook [Cook 84] signale que dans la plupart des systèmes de synthèse d'images on peut utiliser une ou plusieurs fonctions d'éclairement dans une scène. Cependant, ces fonctions sont basées sur des modèles fixes auxquels doivent se conformer les différentes surfaces. Il présente un modèle flexible d'éclairement basé sur une structure d'arbre. Un nœud dans l'arbre est associé à une opération (opération arithmétique, produit scalaire, ...). Les descendants d'un nœud constituent les entrées de l'opération associée à ce nœud qui envoie, à son tour, un résulat à ses parents. Ce modèle permet de spécifier des fonctions d'éclairement assez complexes et spécifiques aux différents types de surfaces. Ainsi, c'est l'utilisateur qui fabrique lui-même sa fonction d'éclairement qu'il associe à une surface donnée.

Pour nous aider à atteindre une flexibilité et une évolutivité satisfaisante pour notre modeleur, nous avons utilisé la programmation orientée objet. Les concepts de ce nouveau style de programmation ont été utilisés par bien des chercheurs dans des domaines variés de l'informatique. L'infographie est un de ces domaines, et a essayé de bénéficier des avantages que ce nouveau style de programmation offre. Le succès varie avec les diverses disciplines de l'informatique graphique : les systèmes d'animation, les interfaces utilisateurs, les systèmes de modélisation et de rendu. De nombreux travaux ont été publiés pour l'animation ([Reyn 82], [Lore 87], [Magn 85], [Bree 87], [Bree 89], [Fium 87], [Reyn 87]), et les interfaces utilisateur ([Card 85], [Bart 86], [Born 86], [Hill 86], [Reis 86], [Sabe 87], [Cout 88]), mais beaucoup moins pour les systèmes de modélisation et de rendu.

Dans ce chapitre, nous étudions comment on peut choisir les classes et concevoir la hiérarchie entre ces classes dans un modeleur pour atteindre la flexibilité et l'évolutivité souhaitées. Dans les articles décrivant des systèmes de modélisation orientés objet, les auteurs ne mettent pas en évidence l'apport de la programmation orientée objet au système. Ils ne lient pas le problème de l'évolutivité du modeleur au choix de la hiérarchie des classes. Ils ne font, en général, que détailler les hiérarchies utilisées et la communication des messages entre objets sans justifier les choix des classes et des relations d'héritage entre

elles, et sans exposer les problèmes rencontrés pour effectuer ces choix. Ces hiérarchies n'arrivent pas forcément à convaincre le lecteur que ce sont les meilleurs choix qui ont été effectués. Dans le paragraphe 2, nous citerons les principaux travaux publiés concernant la réalisation de systèmes de modélisation orientés objets. Dans le paragraphe 3, nous rappellerons rapidement les principes de la programmation orientée objet, les avantages et les inconvénients de cette technique de programmation et nous discuterons du problème du choix des classes. Dans le paragraphe 4, nous exposerons quelques caractéristiques de notre modeleur qui favorisent son évolutivité et sa flexibilité, et nous décrirons la hiérarchie des classes dans notre implémentation. Dans le paragraphe 5, nous ferons une brève présentation du langage C++ et nous donnerons un exemple de description d'une scène en CastorC++. Enfin, nous conclurons dans le paragraphe 6.

2 PRESENTATION DES TRAVAUX DEJA PUBLIES

Grant [Gran 86] était parmi les premiers chercheurs à avoir analysé la possibilité de l'exploitation des notions de *classe* et d'*héritage* de la programmation orientée objet pour la conception d'un système de synthèse d'images. Il propose d'utiliser l'héritage pour mettre en commun, le plus haut possible dans la hiérarchie, les opérations communes qu'il a identifiées entre différentes fonctions. Ainsi, il arrive à diminuer la complexité du système. Chmilar [Chmi 89] a utilisé d'une façon similaire la programmation orientée objet pour diminuer la complexité d'un système intégrant des outils de modélisation et d'animation.

Sabella [Sabe 87] décrit une approche orientée objet pour la construction, la manipulation et l'affichage de modèles géométriques de puits de pétrole. Il mélange des informations 3D, 2D et 1D pour pouvoir prendre en compte l'extrême irrégularité du réservoir. De plus, il a étendu la technique de modélisation traditionnelle par arbre de construction en ajoutant une nouvelle opération booléenne : l'opérateur *over* . Ce dernier est parfois utilisé à la place de l'opérateur *union*. En effet, supposons qu'on veuille faire l'union de deux primitives de modélisation qui s'intersectent. L'operateur *over* permet de préciser lequel des deux doit occuper le volume de l'intersection :

$$A \bullet B = \begin{cases} A & \text{si A INTER B est non vide} \\ A \cup B & \text{sinon} \end{cases}$$

Fleischer [Flei 87] décrit un système de modélisation orienté objet et insiste sur le pouvoir d'abstraction que ce style de programmation offre, de façon à séparer nettement la modélisation du rendu, et à maintenir indépendantes les différentes opérations entre objets grâce au mécanisme d'envoi de messages.

[Yamr 87] présente une version orientée objet de l'algorithme du tracé de rayons, mais ne met pas en évidence l'avantage de cette version par rapport à un algorithme de tracé de rayons utilisant la programmation classique. Au contraire, il signale que le mécanisme de l'envoi de messages augmente le temps de calcul.

Beek [Beek 89] et Launay [Laun 89] décrivent chacun un système de Conception Assistée par Ordinateur (CAO). Launay signale que la signification de l'orientation vers les objets est interprétée d'une façon différente selon les gens. Il met en cause la définition courante d'un système de CAO orienté objet (un système de CAO est orienté objet s'il est écrit en un langage orienté objet), et propose une nouvelle définition pour un tel système :
un système de conception assistée par ordinateur orienté objets doit vérifier les propriétés suivantes :
- les modèles crées par un utilisateur peuvent facilement être modifiés et améliorés ultérieurement.
- un utilisateur n'a pas à définir un contexte de travail (ouverture de fichiers), et doit pouvoir accéder en permanence aux différents objets et différentes fonctions.
- le système possède une base de données dans laquelle les modèles définis par un utilisateur, pour une application donnée, peuvent être partagés par un autre utilisateur et pour une autre application.

Guitton et Schlick [Guit 90] présentent un système de synthèse d'image orienté objet basé sur l'algorithme de tracé de rayons. Ils signalent que plusieurs tâches comme le calcul du point d'intersection entre un rayon et une primitive donnée, le calcul de la couleur de ce point, ... peuvent être effectués en utilisant différentes techniques qui dépendent de la primitive. Par conséquent, il est nécessaire d'implémenter toutes ces techniques pour obtenir les meilleurs résultats. Ils proposent la programmation orientée objet, qui, par les facilités de réutilisabilité et d'encapsulation qu'elle offre, facilite ce mélange des méthodes. Par contre, ils n'évoquent pas l'utilisation de la programmation orientée objet pour définir les éléments dans une scène : l'aspect modélisation est complétement absent dans l'article.

3 LA PROGRAMMATION ORIENTEE OBJET

Les premiers programmes étaient écrits en langage machine et dépendaient donc étroitement de l'architecture des ordinateurs sur lesquels ils étaient implantés. Les techniques de programmations ont naturellement évolué vers une séparation de plus en plus nette entre les concepts manipulés dans les programmes et leur représentation interne en machine. De nouveaux langages ont donc été créés pour dépasser le niveau d'abstraction du langage machine. Cela permettait de programmer de manière plus lisible, en mettant en évidence les aspects algorithmiques des solutions proposées.

Devant la complexité croissante des problèmes abordés, il s'est donc avéré rapidement indispensable de mieux structurer les programmes : un programme sera considéré comme un ensemble de procédures et un ensemble de données sur lequel agissent ces procédures. Les méthodes d'analyse consistent à découper la tâche à effectuer en un ensemble de modules indépendants, plus faciles à réaliser, considérés comme des boîtes noires. On parle alors de programmation *dirigée par les traitements*. Le langage Pascal permet un bon découpage d'une application en procédures. Cependant, cette technique peut entraîner de profonds bouleversement dans l'organisation des procédures lors du moindre changement de la structuration des données.

La notion d'encapsulation pallie à cet inconvénient : les données et les procédures qui les manipulent sont regroupées dans une même entité, l'objet. Les détails d'implantation sont cachés, le monde extérieur n'ayant accès aux données que par l'intermédiaire d'un ensemble d'opérations constituant l'interface de l'objet. Le programmeur n'a pas à se soucier de la représentation physique des entités utilisées et peut raisonner en termes d'abstractions.

Ainsi, l'objet regroupe une partie statique, un ensemble de données, et une partie dynamique, un ensemble de procédure manipulant ces données. L'objet est ainsi défini par son comportement et non par sa structure. Il est muni d'une interface qui spécifie les interactions qu'il peut avoir avec l'extérieur, et la seule manière de communiquer avec lui est d'invoquer une des procédures de son interface. C'est l'objet lui-même qui est responsable de la manière dont l'opération est effectuée. Dans certains langages, l'objet peut même refuser d'exécuter l'action ou bien le faire exécuter par un autre objet.

La programmation est dirigée par les données : on commence par définir les types d'objets avec leurs opérations spécifiques. L'univers de l'application est par conséquent

composé d'un ensemble d'objets qui détiennent, chacun pour sa part, les clés de leur comportement. Ce type de programmation est appelé *programmation orientée objet.*

Simula fut le premier langage à regrouper données et procédures dans une même entité. Sa deuxième version, SIMULA 67, a formalisé les concepts d'objet et de classe. Cette démarche remettait en cause l'habituelle séparation entre données et procédures, en mettant l'accent sur l'encapsulation et l'abstraction des données.

Smalltalk a généralisé la notion d'objet qui devient l'unique entité manipulée dans les programmes. Plusieurs versions successives ont vu le jour introduisant chacune de nouveaux concepts comme la notion de métaclasse. Smalltalk est maintenant un véritable environnement de programmation très sophistiqué avec une interface homme-machine agréable (écran bitmap avec souris, système de fenêtrage, ...). De nombreux langages, dont une bonne partie sont construits à partir de langages existants, se sont inspirés de Smaltallk-80 et de Simula comme Flavors, Ceyx (sur-couches de Lisp), Objective-C, C++ (sur-couches de C).

Simultanément à l'avènement de Smalltalk, la notion d'acteur dans le langage Plasma a été introduite. Comme l'objet, l'acteur est la seule entité de son univers, il est constitué d'un filtre et d'un script. Si l'on compare un acteur avec une fonction, le filtre généralise la liste des paramètres tandis que le script décrit le corps de la fonction.

Les langages de frames sont des langages objets plutôt orientés vers l'intelligence artificielle qui était, avec le génie logiciel, les deux branches de l'informatique à influer le plus sur le développement des langages orientés objets. Les premiers langages de frames furent KRL, FRL, NETL. Leur influence se retrouve actuellement dans la plupart des langages dédiés à la conception de systèmes experts qui marient l'approche objet pour la description des connaissances avec un mécanisme déductif à base de règles, pour l'exploitation des connaissances. Kool et Yafool sont des exemples de tels langages.

Dans le paragraphe 3.1 nous présenterons rapidement les notions de base de la programmation orientée objet. Nous discuterons ensuite, dans le paragraphe 3.2, des avantages et des inconvénients de ce style de programmation. Enfin, dans le paragraphe 3.3, nous abordons le problème du choix des classes.

3.1 Définitions

Les classes :

Une *classe* est la description d'une famille d'objets ayant même structure et même comportement. Une classe sert de modèle pour la création des objets. Ce modèle décrit une structure comprenant des données, que nous appelerons les *champs* dans la suite, et des procédures, les *méthodes*. Les champs constituent la composante statique : ce sont les données décrivant les caractéristiques communes des objets de la classe. Les méthodes, en revanche, constituent la composante dynamique : ce sont les procédures décrivant leurs comportements communs. Définir une classe revient donc à exhiber une structure de données (les champs) et des procédures pour manipuler ces données (les méthodes). De plus, on ne peut accéder à ces données (pour les consulter ou les modifier) que par ces procédures : on dit que ces champs sont les variables privées de la classe.

L'instanciation :

une classe est un prototype ou une entité conceptuelle. Sa définition sert en quelque sorte de moule pour construire ses représentants que sont les *objets* (ou les *instances*). On dit qu'un objet est *instancié* de sa classe et il possède les mêmes champs et méthodes que les autres instances de cette classe, les champs prenant cependant des valeurs a priori différentes avec chaque objet. Les classes ressemblent aux types dans les langages procéduraux.

Illustrons ces notions sur un exemple :
Supposons qu'on introduise une structure de données *Conique* dans notre programme :

classe Conique(a,b,c,d,e,f : R).

Les six réels constituent les coefficients dans l'équation de la conique.

Si on se limite à cette définition, on n'a aucune idée de ce que l'on peut faire avec les objets de la classe *Conique*. On ne comprend pas à quoi servent les six paramètres. On ne sait pas quelles opérations peuvent être accomplies sur les paramètres et sur la conique. C'est ici l'intérêt de l'intégration dans une seule entité des aspects dynamiques et statiques (champs et méthodes) : l'ensemble des opérations qui peuvent agir sur un objet sont alors connues et c'est seulement à partir de ces opérations qu'on peut accéder aux champs.

classe Conique (a,b,c,d,e,f)
champs :
a, b, c, d, e, f ;

opérations:
lire_coefficient ;
changer_coefficient ;
translation ;
rotation ;
affinité ;

Toute consultation ou modification des champs de cette classe se fait à travers son interface, en utilisant les méthodes *lire_coefficient* et *changer_coefficient.*

Soit *coni* une instance de la classe Conique :

coni: Conique(4,1,2,5,1,3).

L'objet instance coni est alors :

coni
a = 4 ; b = 1 ; c = 2 ; d = 5 ; e = 1 ; f = 3 ;
changer_coefficient ; lire_coefficient ; translation ; rotation ; affinité ;

L'héritage :

Un des concepts importants de la programmation orientée objets est l'*héritage*. C'est un outil très intéressant du point de vue du génie logiciel. Il s'agit d'un moyen de définir de nouvelles classes à partir d'autres. Dès sa déclaration, une sous-classe possède les caractéristiques de ses parents. Tout se passe comme si les champs et les méthodes de la classe mère étaient recopiées dans la sous-classe. On peut, par exemple, vouloir définir la classe *cercle* ; au lieu de définir une nouvelle classe n'ayant aucune relation avec la classe Conique, on définit la classe *cercle* comme une sous-classe de la classe Conique.

Class Cercle (a,b : r)
hérite Conique.

Les opérations sur la classe mère seront encore des opérations sur cette nouvelle classe. Cependant, on est souvent amené à redéfinir un certain nombres d'opérations pour la nouvelle classe parce qu'on peut les implanter de façon plus efficace. Par exemple, si la classe Conique possède une fonction *tracer* qui trace une conique sur un écran, cette fonction sera sans doute applicable pour un cercle. Cependant on préfère en général redéfinir cette fonction pour la classe cercle car l'algorithme de tracé d'un cercle est plus efficace que celui qui trace une conique quelconque. De même, la sous-classe peut posséder de nouvelles variables et méthodes qui lui soient propres et qui n'existent pas dans la classe mère. L'héritage favorise une conception par raffinements successifs qui petit à petit rajoutent de nouvelles caractéristiques aux classes. Ceci est possible lorsqu'on arrive à faire une analyse descendante, ce qui n'est pas toujours évident. En effet, on part parfois dans une application du cas particulier pour remonter vers le cas général.

Par conséquent, la création d'un graphe d'héritage permet de structurer hiérarchiquement les classes en factorisant les propriétés communes à plusieurs classes dans une classe supérieure. La relation d'héritage est transitive : les caractéristiques des classes supérieures sont héritées par les classes inférieures, qui sont d'autant plus spécialisées qu'elles sont proches des feuilles de l'arbre.

Il existe deux sortes d'héritage :

- L'héritage *simple* où une classe ne peut posséder qu'une super-classe. Dans ce cas, le parcours du graphe d'héritage est très simple : il suffit de remonter l'arbre jusqu'à la classe où la propriété est disponible.

- L'héritage *multiple* où une classe peut hériter de plusieurs classes sans que ces dernières ne soient liées par des relations hiérarchiques. Supposons que dans un programme de gestion de stock dans un magasin, il existe les classes *article_fragiles* et *article_de_luxe*. Une télévision par exemple est en même temps un objet fragile et de luxe. Un exemple de l'héritage multiple est la classe téléviseur qui hérite des deux classes *article_fragiles* et *article_de_luxe*.

Quand un même attribut apparaît dans plusieurs super-classes, il se pose alors un problème de résolution de conflit pour déterminer quelle définition utiliser pour cet attribut. Chaque langage objet disposant de l'héritage multiple propose une stratégie de parcours du graphe d'héritage. Les recherches en profondeur d'abord permettent d'obtenir des extensions linéaires, comme l'ont montré les auteurs dans [Habi 88]. Une version simplifiée de l'algorithme qu'ils proposent consiste à retenir les nœuds du graphe dans l'ordre d'empilement du parcours en profondeur d'abord, sans marquer les nœuds déjà visités : le résultat contient donc plusieurs occurrences de certains nœuds. La liste obtenue est parcourue à l'envers, en supprimant au fur et à mesure les éléments déjà rencontrés au moins une fois. De cette façon, seule la dernière occurence de chaque élément dans la liste initiale est conservée dans la liste finale.

La transmission des messages :

Une méthode est activée par un envoi de *message* qui est, en quelque sorte, l'équivalent d'un appel de procédure. Les messages représentent donc le moyen de communiquer entre les objets. Un objet est pourvu d'une *interface* de communication qui permet de traiter les messages qui lui sont adressés. Un objet étant une entité indépendante, il ne peut pas agir directement sur un autre objet. Il doit utiliser une des méthodes appartenant à l'interface de cet autre objet et lui envoyer un message, qui demande l'exécution de la méthode en question. Le message est une requête que l'objet auquel elle est adressée se charge de satisfaire. L'interface d'un objet comprend un dictionnaire de sélecteurs auxquels sont associées les procédures représentant les méthodes correspondantes. Normalement, on confond sélecteur de méthode et nom de méthode. A la réception d'un message, l'objet recherche le nom de la méthode (le sélecteur) dans son dictionnaire puis il exécute la procédure associée. Lorsque le sélecteur n'existe pas dans le dictionnaire, il est alors recherché dans les dictionnaires des classes supérieures en remontant dans l'arbre des classes. Ce mécanisme correspond à l'héritage des méthodes.

Le mécanisme d'envoi de messages diffère suivant le langage utilisé. En Smalltalk, l'univers est complètement uniforme : la seule entité est l'objet et la seule structure de contrôle est l'envoi de message qui permet de communiquer avec les objets. Plus qu'une simple description, une classe Smalltalk-80 est elle-même un objet instance d'une autre classe appelée sa métaclasse. Une transmission de message mentionne le receveur suivi du message proprement dit, composé d'un sélecteur et d'arguments. Le sélecteur permet de retrouver la méthode à exécuter par consultation du dictionnaire des méthodes de la classe de l'objet receveur. La méthode sélectionnée est ensuite appliquée aux arguments.

A l'opposé de Smalltalk, C++ [Stro 86] ne possède pas de mécanisme d'envoi de messages. Un objet peut accéder directement à la méthode d'un autre objet (à condition qu'elle ne soit pas privée). Le programmeur peut définir des fonctions sans les attacher à des objets quelconques en tant que méthodes ; ces fonctions sont utilisables par tous les objets.

En résumé, vu sous l'angle programmation orientée objet, un programme peut être considéré comme un ensemble d'objets communiquant entre eux par envoi de messages. Il faut donc changer la manière d'aborder et d'analyser un problème. Alors qu'une approche classique consiste avant tout à définir les fonctionnalités du programme dèsiré, une analyse orientée objets doit d'abord cerner les entités qui existent, puis spécifier la manière dont ces entités interagissent.

3.2 Avantages et inconvénients de la programmation orientée objet

La programmation orientée objet possède les avantages suivants par rapport à la programmation procédurale classique :

1. Conceptualisation : La programmation orientée objet facilite la conceptualisation des problèmes au moment de leur spécification. Quand on pense à *quelque chose*, on y pense en termes de classes d'objets et de messages auxquels ils doivent répondre. Ce pouvoir d'abstraction permet de se focaliser sur ce qui est vraiment important.

2. Affinage des concepts : L'héritage favorise une conception par raffinements successifs qui petit à petit rajoutent de nouvelles caractéristiques aux classes.

3. Meilleure vue générale du système : L'encapsulation des données et des procédures dans des objets et la hiérarchie des classes permettent de refléter la structure générale du système.

4. Renforcement de la modularité *:* La programmation classique permet de changer la manière dont un module réalise une tâche qui lui est assignée sans que cela ait une influence sur le reste du programme. Mais, à cause de la séparation des données et des procédures, l'accès aux données est réparti dans l'ensemble du programme ; la moindre modification de la structure des données peut entraîner la mise à jour de nombreuses parties du programme. En programmation orientée objet, un objet regroupe données et fonctions exploitant ces données. Il constitue une unité de programmation ou un module autonome et indépendant des autres modules du système. En effet, un objet définit des interfaces qui permettent aux autres objets de l'application d'accéder à ses données. Toute modification dans une classe ne touchant pas aux interfaces de cette classe ne provoque que des perturbations locales grâce à la privatisation des champs et au mécanisme d'héritage. C'est une modularité supplémentaire par rapport à la programmation classique qui rend l'extensibilité et la maintenance d'un programme plus aisés.

5. Réutilisabilité et économie de code : L'héritage favorise la réutilisabilité et l'économie du code. En effet, en mettant en commun, grâce à l'héritage, des opérations communes à plusieurs classes dans une nouvelle classe dont héritent les autres, nous arrivons à économiser considérablement du code.

6. Facilité du déboguage : Le déboguage du programme se trouve facilité par le fait que les champs d'une classe sont des variables privées. Ainsi, nous savons quelles sont les procédures qui sont capables de modifier ces variables et toute anomalie dans les valeurs de ces variables sera détectée plus facilement.

L'orientation vers les objets est donc un mécanisme général de structuration des composantes d'un programme qui entraîne des avantages considérables en génie logiciel, mais ce n'est pas une panacée, comme on peut avoir tendance à le croire. Elle possède d'ailleurs un certain nombre d'inconvénients :

1. Surcharge du système : on accuse les langages orientés objet d'entraîner une surcharge système non négligeable, principalement à cause du mécanisme de communication par messages. Cependant, cela est dû au fait que ces langages existent sur des ordinateurs à

architectures classiques ; il est tout à fait concevable que cette surcharge soit négligeable sur une machine spécialisée objets où ces mécanismes seront mieux gérés [Masi 90].

2. Dépendance de l'application : Il n'est pas certain qu'une analyse dirigée par les données soit dans tous les cas meilleure qu'une approche procédurale. Ceci peut dépendre du problème étudié. Par exemple, une approche strictement orientée objets comme celle de SMALLTALK [Alth 81] n'est pas toujours naturelle et avantageuse. D'autre part, l'efficacité des avantages de la programmation orientée objet cités ci-dessus dépend beaucoup de l'application qui peut être bien adaptée ou pas à ce style de programmation. Par exemple, l'affinage des concepts n'est possible que si on arrive à faire une analyse descendante du problème, ce qui n'est pas toujours évident. En effet, on part souvent dans une application du cas particulier pour remonter vers le cas général. Par exemple, nous avons commencé, en développant notre modeleur, par l'introduction de deux primitives de modélisation seulement: le cube et le cylindre. Ensuite, après avoir obtenu des résultats satisfaisants pour ces deux primitives, nous avons introduit la sphère et le cône. Nous nous sommes aperçu alors qu'il fallait ajouter une classe *quadrique* dont héritent ces primitives de modélisation. Ainsi, nous avons été amenés à remonter du cas particulier (les primitives de modélisation) au cas général (la quadrique).

3. Diffusion limitée : L'orientation vers les objets étant un style de programmation, les langages orientés objets ont été conçus pour le renforcer. Le problème de ces langages est qu'ils ne possèdent pas une large diffusion comme certains langages classiques comme Pascal ou C. Pour cela, les chercheurs préfèrent souvent adopter un tel style de programmation à l'intérieur d'un langage de programmation traditionnel [Lore 87], [Bree 87]. Cela explique, en partie, le succès du langage C++ [Stro 86] - surtout dans le monde UNIX - qui est une extension du langage C : tout à fait compatible avec le langage C, il profite du succès et de la portabilité de ce dernier. Il est par exemple très facile d'installer C++ sur un site UNIX, et de l'utiliser avec des outils de l'environnement de programmation C, comme le débogueur symbolique dbx.

3.3 Une méthodologie pour le choix des classes

Le choix des classes et des relations d'héritage les liant dans une application quelconque n'est pas automatique. C'est une étape importante et assez difficile dans le processus de conception qui influe sur l'architecture de tout le système et sur la

compréhension de ce système. Un mauvais choix des classes risque de limiter les aides que la programmation orientée objet pourrait apporter à cette application. En particulier, un choix inadapté des classes ou des relations hiérarchiques entre celles-ci pour notre modeleur serait susceptible de limiter l'aide que la programmation orientée objet pourrait apporter pour atteindre nos objectifs d'évolutivité et d'extensibilité. Pour cela nous avons trouvé indispensable d'étudier, dans ce paragraphe, le problème du choix des classes dans un système de modélisation.

Les classes sont les entités et les concepts du problème étudié. Parfois il est facile de les identifier et de décider quels champs et méthodes on doit affecter aux classes les décrivant. Ces classes, facilement identifiables, sont normalement rencontrées dans tous les systèmes qui étudient un même problème. Ce sont les objets physiques du problème étudié qui s'imposent d'une façon naturelle en tant que classes. D'autres entités peuvent être plus difficiles à cerner. La modélisation de ces entités en classes dépend alors beaucoup du concepteur. En effet, celui-ci essaie d'utiliser une analogie avec d'autres systèmes sur d'autres problèmes qu'il a déjà traités. La hiérarchie des classes dépend alors de l'expérience du concepteur. C'est sans doute une des raisons pour laquelle cette hiérarchie n'est pas toujours la même dans tous les systèmes qui traitent un problème donné.

Certains auteurs ont proposé une technique pour trouver les classes dans une applications donnée : elle consiste à souligner les noms et les verbes dans le cahier des charges de cette dernière ; aux noms correspondent les classes et aux verbes les méthodes. Cette technique est assez simpliste et ne peut aboutir qu'à des résultats grossiers comme le signale Meyer [Meye 88] ; en particulier, elle engendre de très nombreuses classes.

Meyer [Meye 88] pose le problème du choix des classes et signale qu'il n'existe pas de réponse absolue. Il signale qu'une telle réponse serait une technique infaillible pour la conception des logiciels ; or une telle technique n'est pas plus facile à définir qu'une méthode infaillible pour démontrer les théorèmes. Il remarque que la création de classes inutiles est une erreur de conception, mais, là aussi, il n'existe pas de méthode pour éviter cette erreur. Il conseille d'obtenir les classes fondamentales d'une application directement à partir du monde réel que cette application modélise : aux classes des objets physiques du monde correspondent les classes dans les logiciels. C'est ce que nous avons appelé *les entités facilement identifiables*.

Lorensen [Lore 87] a proposé une méthodologie pour construire une hiérarchie de classes, dont la première étape consiste à identifier les entités pour en faire des classes. Mais il n'a donné aucune règle pour effectuer cette étape, et pour effectuer les choix.

Si les classes correspondent à des entités facilement identifiables, le choix des champs et méthodes est normalement aisé. Une partie d'entre eux se trouvera dans ces classes dans tous les systèmes qui traitent le même problème. Dans cette étape de conception il suffit de regarder les fonctionnalités des méthodes sans se préoccuper de l'implémentation.

Pour chaque nouvelle classe, il faut regarder si elle peut hériter d'une autre classe, et détecter les méthodes qui doivent être redéfinies. Parfois, dans l'étape de la conception, et en faisant le raffinement successifs des classes grâce à l'héritage, des données et des opérations communes peuvent être détectées entre différentes classes ne possédant pas de relations d'héritage entre elles. Ces données et méthodes sont alors combinées dans une nouvelle classe qui sera une classe-mère des classes précédentes. Cette nouvelle classe peut ne pas avoir un sens en tant qu'objet physique dans le problème étudié. Son seul rôle est de contenir des champs et des méthodes communs. Grâce à cette technique on arrive à économiser beaucoup de code et diminuer la complexité du système. Par exemple, supposons que nous possédions les deux classes Phong et Gouraud. Chacune d'entre elles possède une méthode qui, dans son corps, utilise une interpolation linéaire entre deux nombres. Nous pouvons alors mettre l'opération interpolation dans une classe à part dont hériteront les deux classes Phong et Gouraud.

La division d'une application en classes utiles et compréhensibles est donc un art. Elle dépend beaucoup du concepteur et de son expérience. Pour alléger ce problème, il faut essayer de trouver des règles qui guident le programmeur dans son choix des classes. Mais trouver ces règles est un problème encore plus difficile.

Après ces généralités sur le choix des classes, nous allons approfondir ce problème dans le cas d'un système de modélisation. En particulier, nous allons proposer des règles qui nous semblent permettre de justifier les choix effectués.

Dans un système de modélisation et de rendu, les entités facilement identifiables sont surtout les primitives de modélisation (sphère, cube, cylindre, surface, ...), les primitives géométriques (polygones, triangles ....), les primitives d'affichage (pixel, segment,....) et les sources lumineuses. Ces classes, ou, au moins, une partie d'entre elles, existent pratiquement dans tous les systèmes de modélisation orientés objet. Comme on n'hésite pas à décrire par une classe chacune de ces entités, nous proposons la règle suivante pour le choix des classes dans notre système :

Règle 1 : toute entité visible sur l'écran est décrite par une classe dans le système.

Les transformations géométriques (translation, rotation, affinité,...) sont des fonctions qui reçoivent un certain nombre de paramètres et calculent une matrice. On a donc tendance à mettre ces fonctions dans les classes décrivant les primitives de modélisation en tant que méthodes.

Cependant, on peut considérer les transformations non pas en tant que fonctions mais comme des entités indépendantes. La translation, par exemple, peut être vue comme une entité définie par les trois paramètres de translation et par la fonction fabriquant la matrice de translation à partir de ces paramètres. On peut donc décrire les transformations à l'aide de classes. Les paramètres d'appel de la fonction deviennent les champs de cette classe et la fonction elle même sera une méthode opérant sur ces champs.

L'exemple précédent peut se généraliser : il existe une dualité entre fonctions et classes. Les paramètres d'appel d'une fonction deviennent les champs de la classe et la fonction elle même devient une méthode dans cette classe. Par exemple, nous pouvons fabriquer une classe *créer_facettes* et des classes filles à cette dernière : *créer_facettes_sphère*, *créer_facettes_cylindre*, ... Dans la classe *créer_facettes*, il y aura une variable d'instance *degré_de_facettisation* (qui est le paramètre d'appel des fonctions de facettisation).

Faut-il alors construire autant de classes que de fonctions? Il faut sans doute s'arrêter et trancher dans cette dualité entre fonctions et classes. Pour ce faire, nous proposons une deuxième règle:

Règle 2 : si l'entité duale d'une fonction est utilisée fréquemment comme entité abstraite, cette fonction est représentée par une classe.

Pour les transformations par exemple, la personne qui modélise une scène aura besoin de définir une transformation donnée pour l'appliquer à plusieurs primitives de modélisation à différents moments. Il a donc besoin d'utiliser l'aspect *entité abstraite* de la transformation en la définissant en tant qu'entité pour pouvoir s'en servir plusieurs fois. Les entités translations, rotations, ... vérifient donc la seconde règle. Nous les avons donc représentées par des classes.

ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE SAINT-ÉTIENNE
INFORMATIQUE
158 cours Fauriel
42023 Saint-Étienne Cedex 2

Dans notre système, le degré de facettisation des primitives de modélisation est calculé automatiquement par la fonction qui effectue la facettisation. Les éventuelles classes associées à ces fonctions ne correspondraient pas à des entités que l'utilisateur aurait besoin d'utiliser fréquemment. Il en est de même pour les fonctions qui fabriquent les boîtes englobantes des primitives de modélisation dans le repère unitaire. Les éventuelles classes correspondantes auraient un rôle purement fonctionnel ; leur aspect *entité abstraite* est trop faible pour qu'il soit utile pour l'utilisateur. Pour cela, nous n'avons pas introduit de classes *créer_facettes* ou *fabriquer_boîte* dans notre système.

La troisième règle que nous avons utilisée pour créer une classe (duale d'une fonction) est :

Règle 3 : si le code de la fonction est assez grand il faut créer la classe correspondante et utiliser l'héritage pour mettre en commun des opérations avec d'autres fonctions et diminuer ainsi la complexité du système en économisant du code.

Si par exemple nous décidons d'introduire les déformations libres [Nie 89] dans notre modeleur, nous les décrirons par des classes, d'après la troisième règle, afin de pouvoir économiser du code, puisque les fonctions de déformation sont assez complexes.

De même, si nous décidons d'étendre notre système de modélisation pour qu'il devienne un système de synthèse d'images complet, il serait naturel que les algorithmes d'élimination des parties cachées soient des méthodes dans la classe qui décrit les nœuds de l'arbre CSG. Mais, dans ce cas, nous aurions à gérer un système très complexe qui supporterait les fonctions de modélisation, les méthodes de calcul de visibilité (tracé de rayons, tampon de profondeur, Atherton, ...), les méthode de lissage (Phong, Gouraud, ...), les différents modèles d'éclairement, ... L'utilisation de la règle 3 permettrait de mettre en commun les opérations entre ces différents algorithmes et d'économiser alors du code, diminuant ainsi la complexité du système. C'est ce que Grant [Gran 86] a fait dans son système de synthèse d'images qui intègre des fonctions de modélisation, des algorithmes d'élimination des parties cachées et des fonctions de rendu. Il a utilisé l'héritage pour remonter le plus possible dans la hiérarchie les opérations communes qu'il a identifiées entre les différentes fonctions.

Dans le paragraphe 5, nous allons détailler notre hiérarchie de classes. Nous signalerons alors l'utilisation de ces trois règles dans nos choix de classes.

4 LES CARACTERISTIQUES DU MODELEUR CASTORC++

Comme nous l'avons signalé dans l'introduction de ce chapitre, nous avons voulu rendre notre modeleur le plus évolutif et le plus flexible possible. Dans le paragraphe 4.1, nous exposerons les principes que nous avons utilisés pour concevoir la hiérarchie des classes de notre modeleur, afin de favoriser au maximum l'évolutivité et l'extensibilité de celui-ci. Dans le paragraphe 4.2, nous décrirons comment nous avons rendu notre modeleur flexible et nous signalerons l'apport de la programmation orientée objet à cette flexibilité.

4.1 La conception de la hiérarchie des classes

Comme nous l'avons signalé dans le paragraphe précédent, les primitives de modélisation, de géométrie et d'affichage sont des entités facilement identifiables dans un modeleur. Nous les retrouvons décrites par des classes dans tous les systèmes de modélisation orientés objet que nous connaissons, avec une hiérarchie semblable à celle de la figure 2.1. Dans ce qui suit, nous allons montrer qu'une telle hiérarchie n'est pas suffisante pour faire bénéficier notre système de modélisation de tous les avantages de la programmation orientée objet. En particulier, la modularité et la vue générale du système se trouvent limités alors qu'ils sont importants pour l'évolutivité dèsirée. Nous proposerons ensuite une autre hiérarchie qui nous paraît mieux correspondre aux objectifs que nous nous sommes fixés.

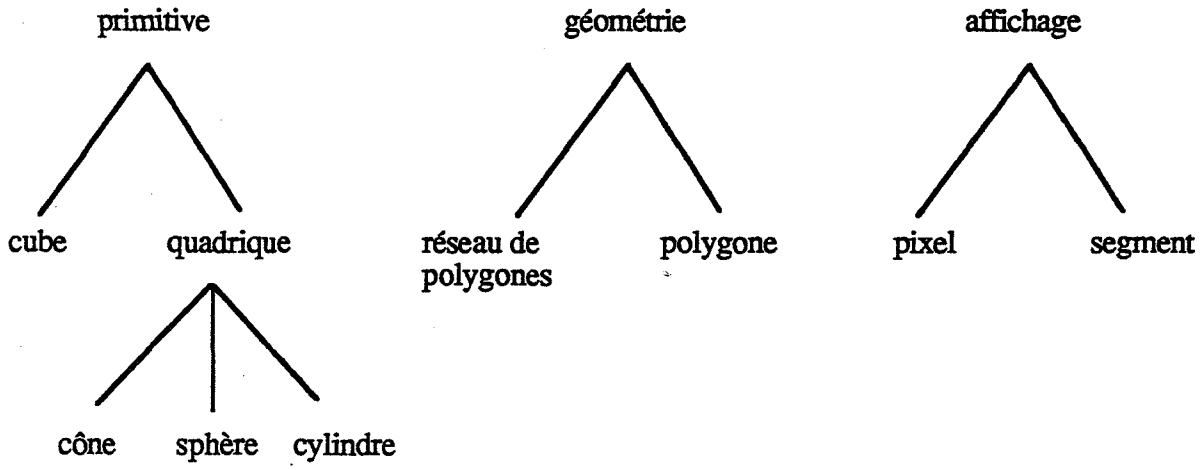


FIGURE 2.1

Dans la hiérarchie des classes de la figure 2.1, les primitives de modélisation d'une part et les primitives géométriques d'autre part sont décrites par des classes indépendantes. A première vue, ceci semble être satisfaisant : on a mis en évidence les objets avec les opérations opérant sur eux et les relations d'héritage entre ces objets. Cependant, les relations entre les éléments graphiques sont trop complexes pour qu'elles puissent être exprimées par ces objets et ces simples relations hiérarchiques entre eux. Plusieurs aspects du système graphique restent non suffisamment mis en évidence. En effet, de manière globale, le système graphique a une structure de pipeline semblable à celle de la figure 2.2.



FIGURE 2.2

Pendant la phase de la modélisation, une sphère, par exemple, est vue comme une primitive. Après l'avoir facettisée pour la visualisation, elle sera vue comme un réseau de polygones. Dans la figure 2.1, les classes *sphères* et *réseau_de_polygones* sont chacune dans des sous-arborescences séparées. Aucun lien n'existe entre elles (sauf le fait qu'elles sont petits fils de la classe racine de l'arborescence), bien qu'il s'agisse du même élément graphique, sous deux représentations différentes, dans le pipeline graphique. L'encapsulation des données et procédures et l'héritage simple utilisé ne sont pas suffisants pour exprimer les relations entre la sphère et le réseau de polygones. Il en est de même pour la relation entre un polygone et les pixels de l'image finale, ces pixels étant la troisième représentation de la sphère dans le pipeline graphique.

En fait, la programmation orientée objets ne met en évidence, avec la hiérarchie des classes, que des relations statiques entre celles-ci : on définit (au sens mathématique) une hiérarchie d'ensembles et de sous-ensembles. Le pipeline graphique, pour sa part, représente un aspect dynamique: un objet est associé à d'autres éléments qui ne sont pas nécessairement éléments de la même classe (et donc encore moins d'une sous-classe) que lui.

Nous pouvons donc constater que, pour un choix de hiérarchie des classes comme nous le trouvons le plus souvent dans la littérature (cf figure 2.1), l'avantage de la vue générale du système cité dans le paragraphe 3.2 se trouve limité.

D'autre part, les applications qui sont bien adaptées pour bénéficier de l'avantage de modularité sont celles dont les interfaces entre les classes sont simples. Ce n'est pas le cas pour un système de modélisation à cause de la structure du pipeline graphique et des différentes représentations d'une même primitive. Par exemple, les méthodes de conversion d'une sphère en un réseau de polygones doivent bien connaître la structure de cette dernière classe. Par suite, l'interface entre ces deux modules est loin d'être simple. Si nous effectuons des modifications dans la structure du réseau de polygones, nous changeons alors la deuxième représentation de la sphère dans le pipeline graphique. Ces modifications ont toutes les chances de toucher à l'interface entre la sphère et le réseau de polygones. Par suite, toutes les méthodes de conversion entre ces deux représentations devront être changées. Une mise à jour de plusieurs parties du programme est donc nécessaire.

Le problème est identique si on crée la classe *réseau_de_triangles* comme une sous-classe de *réseau_de_polygones* ; on définit alors une nouvelle représentation de la sphère et une mise à jour est nécessaire pour la convertir dans cette nouvelle représentation. Un autre exemple est fourni par la méthode *afficher* dans la classe polygone, qui affiche un polygone sur l'écran. Cette méthode convertit un polygone en sa deuxième représentation sur l'écran, à savoir un ensemble de pixels. Une mise à jour est nécessaire après tout changement dans la classe pixel.

Ces exemples montrent que les interfaces entre les composantes d'un système de modélisation sont complexes, à cause de la structure de pipeline de ce système. Avec le choix des classes de la figure 2.1, l'avantage de modularité du paragraphe 3.2 se trouve limité puisque tout changement dans une classe de la hiérarchie géométrie, par exemple, nécessite la mise à jour de plusieurs autres classes qui n'ont aucun lien d'héritage avec celle-ci.

Si nous gardons la hiérarchie des classes de la figure 2.1, tout essai pour résoudre ces problèmes consistera à inventer des conventions pour exprimer des relations comme celles qui existent entre la sphère et le réseau de polygones. Ceci peut être relativement satisfaisant pour un système donné et pour les gens qui le connaissent. Mais, par manque de standardisation, chaque système aura ses conventions, et la portabilité deviendra limitée puisque ce ne sont que des conventions et non des outils standards offerts par la programmation orientée objet.

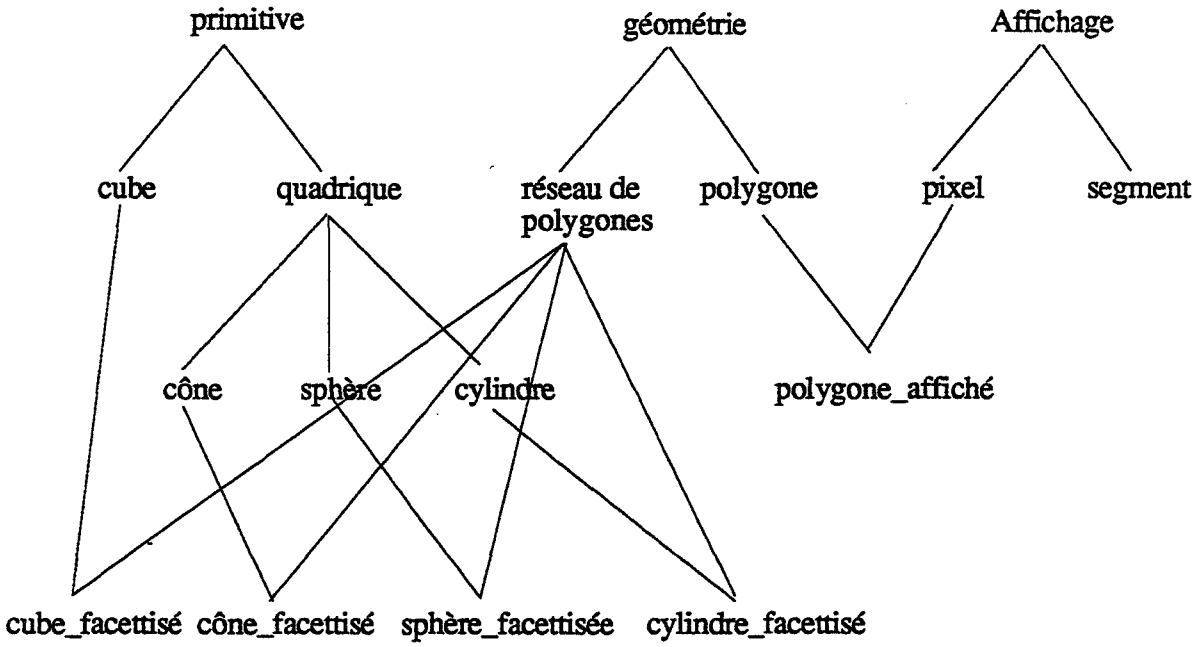


FIGURE 2.3

Pour résoudre les problèmes soulevés ci-dessus, nous avons utilisé l'héritage multiple. Nous avons crée la classe *sphère_facettisée* (figure 2.3) qui hérite des deux classes *sphère* et *réseau_de_polygones*. Cette classe sert de lien entre les deux représentations de la sphère. Elle met en évidence la relation entre la sphère et le réseau de polygones et reflète mieux la conception du système de synthèse d'image. Nous avons fait de même pour les autres primitives de modélisation (*cube_facettisé, cylindre_facettisé, Bspline_facettisée, ...*). D'autre part, nous avons créé la classe *polygone_affiché* qui hérite des deux classes polygone et pixel, mettant ainsi en évidence la relation entre ces deux entités dans le système. Nous arrivons de cette façon à refléter la structure du pipeline graphique.

La hiérarchie proposée nous permet de bénéficier d'avantage de la modularité qu'offre la programmation orientée objets. En effet, toute modification dans la classe *réseau_de_polygones*, par exemple, se propage à la classe *sphère_facettisée* puisque cette dernière est une de ses sous-classes maintenant. Les parties du programme à mettre à jour après de telles modifications sont donc connues et sont des sous-classes de la classe *réseau_de_polygones*. L'inconvénient de notre solution est que nous aboutissons à un graphe d'héritage assez complexe.

4.2 Flexibilité du modeleur

Nous allons maintenant décrire comment nous avons introduit de la flexibilité dans notre système de modélisation orienté objet de type arbre de construction.

Par flexibilité, nous entendons le fait que l'utilisateur puisse choisir, pour chaque primitive ou groupe de primitives de la scène, l'algorithme de visualisation, la méthode de lissage ou le modèle d'éclairement qu'il souhaite leur appliquer. Cela donne beaucoup de souplesse au système. La plupart des systèmes de synthèse d'image utilisent une même méthode pour calculer toute une scène. Or celle-ci peut comporter des primitives de types variés : cubes, quadriques, surfaces gauches ... L'algorithme de visualisation utilisé n'est pas forcément adapté à toutes ces primitives. D'où l'idée de faire un mixage d'algorithmes de visualisation dans une même scène [Crow 82], [Duff 85], [Port 84], [Jaha 90], chaque primitive étant visualisée à l'aide de l'algorithme le plus approprié (cf chapitre 3). Une sphère, par exemple, est généralement affectée au tracé de rayons, sauf si, étant située très loin de l'observateur, on préfère l'afficher à l'aide d'un algorithme plus rapide. Il en est de même pour les méthodes de lissage et les fonctions de calcul d'éclairement : dans une scène, plus une primitive est éloignée de l'observateur, moins elle se voit affecter des méthodes coûteuses en temps de calcul.

Pour réaliser la flexibilité souhaitée, nous avons introduit une classe *aspect* qui contient les champs suivants : couleur de la primitive, caractéristiques du matériau dont elle est constituée, visibilité, lissage, *fonction_d'éclairement*. Chaque élément graphique pointe sur un objet de cette classe. Ainsi, une primitive de modélisation se décrit à un algorithme de visualisation, une méthode de lissage et une fonction d'éclairement. Dans le cas où l'utilisateur n'indique pas quelles sont les méthodes qu'il souhaite utiliser, le système assure un choix automatique de ces méthodes. Ce choix est effectué de la façon suivante : chaque primitive qui ne connait pas l'algorithme de visibilité avec lequel il va être visualisé, interroge sa méthode *décider_visibilité* pour choisir l'algorithme de visualisation le mieux adapté. Cette méthode associe un paramètre à la primitive pour pouvoir prendre cette décision. Soit tmax la longueur maximale des côtés du rectangle englobant la projection de la boîte englobante de la primitive sur l'écran. Le paramètre est la longueur de l'image divisée par tmax. Plus la valeur de ce paramètre est grande, plus la primitive est petite sur l'écran. Par exemple quand tmax prend la moitié de l'image, le rapport vaut 2.

Nous avons introduit une classe *décider_rendu* et des classes filles de cette dernière : *décider_visibilité*, *décider_lissage* et *décider_éclairement*. La classe *décider_visibilité* possède des sous-classes correspondant chacune à un algorithme de calcul de visibilité : la

classe *décider_tracé_de_rayons* pour l'algorithme du tracé de rayons, la classe *décider_atherton* pour l'algorithme d'Atherton, ... La classe *décider_rendu* contient, parmi ses champs, un seuil pour chaque primitive de modélisation : *seuil_cylindre, seuil_sphère, seuil_cube, seuil_Bspline,...*

Pour savoir si une primitive A sera affectée à un algorithme de visualisation B, il suffit d'interroger la méthode *décision* de la classe *décider_B*. Cette méthode reçoit le type de A et son paramètre calculé, et retourne un booléen qui autorise ou non de l'affecter à l'algorithme de visualisation B. La méthode *décision* n'est pas la même pour tous les algorithmes d'élimination des parties cachées. Pour la classe *décider_tracé_de_rayon*, par exemple, la méthode *décision* consiste simplement à tester si cette primitive est réfléchissante et/ou transparente, et à comparer la valeur du paramètre de la primitive A avec celle du champ *seuil_A* de l'objet instance de la classe *décider_tracé_de_rayon*. Elle retourne un booléen autorisant ou non la visualisation de cette primitive avec l'algorithme du tracé de rayons. Les méthodes *décision* dans les classes *décider_Atherton* et *décider_Z_Buffer* testent en plus si la primitive A subit des opérations booléennes différence ou intersection, ou si la scène contient des primitives transparentes de tailles non négligeable sur l'écran. Dans ces cas, la primitive A sera affectée à l'algorithme d'Atherton, car l'algorithme du tampon de profondeur ne sait ni calculer les opérations booléennes ni prendre en comptes les effets de transparence.

Chaque fois qu'on programmera un nouvel algorithme d'élimination des parties cachées, C, il suffit d'ajouter la classe *décider_C*. Dans cette dernière on définira une méthode *décision* et on spécifiera, pour chaque primitive de modélisation, un seuil. Ce dernier sera utilisé par la méthode *décision*, avec d'autres informations dépendant de l'algorithme C, pour décider si on affecte une primitive donnée à l'algorithme C.

Dans chacune des classes filles de la classe *décider_visibilité*, il existe une méthode permettant d'écrire dans un fichier un *objet_CSG* (élément de l'arbre de construction) et/ou un *réseau_de_polygones* avec le format correspondant à l'algorithme représenté par cette classe.

Considérons une surface Bspline. Supposons que l'utilisateur n'ait pas spécifié à quel algorithme d'élimination des parties cachées elle doit être affectée ; le système va alors déterminer avec quel algorithme elle va être visualisée. Pour cela, il calcule le paramètre de cette primitive, puis, il trie les instances des sous-classes de la classe *décider_visibilité* en ordre croissant selon les valeurs du champ *seuil_Bspline* pour ces objets. Par exemple, supposons que nous possédions trois algorithmes de visualisation : le tracé de rayons [Appe

68], [Whit 80], l'algorithme de Catmull [Catm 75], et l'algorithme d'Atherton [Athe 83]. Les méthodes *décision* dans les classes *décider_tracé_de_rayon*, *décider_Catmull* et *décider_Atherton* sont définies de telle manière que si la valeur du paramètre de la Bspline est plus petite que celle de *seuil_Bspline* dans la classe *décider_tracé_de_rayon*, cette primitive sera affectée au tracé de rayons ; si cette valeur est comprise entre les valeurs de *seuil_Bspline* dans les classes *décider_tracé_de_rayon* et *décider_Catmull*, elle sera visualisée à l'aide de l'algorithme de Catmull ; si elle est comprise entre les valeurs de *seuil_Bspline* dans les classes *décider_Catmull* et *décider_Atherton*, la primitive sera affectée à l'algorithme d'Atherton. Sinon, la Bspline sera affectée à l'algorithme dont la valeur du champ *seuil_Bspline* dans l'instance de la classe le représentant parmi les sous-classes de la classe *décider_visibilité* est la plus grande.

On remarque qu'avec le choix des valeurs de *seuil_Bspline* dans les différentes sous-classes de la classe *décider_visibilité*, le concepteur du système impose un ordre de réalisme entre les différents algorithmes de visualisation disponibles : le tracé de rayons est plus réaliste que l'algorithme de Catmull, lui-même plus réaliste que celui d'Atherton. Cela peut convenir pour un utilisateur non-informaticien ; par contre, si l'utilisateur de ce système est un expert, il peut être en dèsaccord avec cet ordre, et souhaiter définir son propre ordre. Certes il peut spécifier, pour chaque primitive, l'algorithme d'élimination des parties cachées qu'il souhaite lui appliquer. Mais il peut aussi désirer utiliser quand même le choix automatique en définissant lui même un ordre de réalisme entre les différents algorithmes d'élimination des parties cachées. Pour cela nous avons prévu, dans la classe *décider_rendu*, une fonction qui affiche les valeurs-seuils de toutes les primitives (*seuil_cube*, *seuil_sphère*, *seuil_Bspline*, ...) et une autre fonction qui permet de les modifier.

Si deux primitives différentes sont liées par une opération booléenne intersection ou différence, elles sont affectées au même algorithme de visualisation comme nous l'expliquerons dans le chapitre suivant. Selon des directives données dès le départ par l'utilisateur, le modeleur interroge uniquement la méthode *décider_visibilité* d'une de ces deux primitives et les affecte toutes deux au même algorithme. Par exemple, dans le cas d'une intersection entre une sphère et un cube, si l'utilisateur privilégie le réalisme sur la rapidité, le système affecte les deux primitives au tracé de rayons si la taille de la sphère est suffisamment grande sur l'écran.

Nous avons introduit la classe *décider_visibilité* et ses sous-classes pour renforcer la modularité du système. Chaque fois que l'on programme un nouvel algorithme de visibilité, il suffit de créer la classe correspondante parmi ces sous-classes pour que ce nouvel

algorithme soit pris en compte dans le choix automatique par le système. Ceci nous permettra d'essayer de nouveaux algorithmes de visualisation et de les comparer à d'autres algorithmes dans la même scène.

Nous avons effectué le même travail pour les méthodes de lissage et les formules d'éclairement. Pour cela, nous avons introduit les classes *décider_lissage_Gouraud décider_lissage_Phong, décider_éclairement_Lambert, décider_éclairement_Phong* et *décider_éclairement_Whitted.* Le choix automatique se fait de la même façon que dans le cas des algorithmes de visualisation. En effet, chacune de ces classes contient une méthode *décision* qui utilise des seuils associés aux différentes primitives de modélisation, avec d'autres informations, pour décider s'il faut affecter une primitive donnée à cette méthode ou pas. Par exemple, les fonctions *décisions* des classes *décider_lissage_Gouraud* et *décider_lissage_Phong* testent si cette primitive va être visualisée par un algorithme travaillant sur des facettes. Dans ce cas, selon la comparaison de son paramètre calculé avec le seuil correspondant à cette primitive dans les instances de ces deux classes, elle sera affectée à la méthode de lissage de Gouraud, à celle de Phong, ou à aucune des deux.

Comme dans le cas des algorithmes d'élimination des parties cachées, il existe un ordre de réalisme que nous avons imposé, la méthode de lissage de Phong est plus réaliste que celle de Gouraud par exemple. Des fonctions ont été prévues pour autoriser l'utilisateur à modifier l'ordre de réalisme entre ces différentes méthodes.

L'approche orientée objet n'améliore pas la fonctionnalité du système de modélisation. Toutes les fonctionnalités de ce système auraient pu être obtenues en programmation classique. Comme nous l'avons signalé dans le paragraphe 3.2, l'orientation vers les objets est un style de programmation, un mécanisme général de structuration du code ; ce n'est pas une panacée. L'apport de l'approche objet à la flexibilité de notre système réside surtout dans le pouvoir d'abstraction qu'il offre. Grâce à l'encapsulation des données et des procédures, nous réalisons une séparation maximale entre les entités géométriques décrites par les classes *sphère, cylindre, cube,* .... et les entités qui représentent les algorithmes de visualisation décrits par la sous hiérarchie *décider_visibilité*. La communication entre ces entités s'effectue par envoi de messages respectant un protocole. Cette communication se trouve alors plus nette et plus facile à comprendre.

4.3 IMPLEMENTATION

Dans ce paragraphe, nous allons décrire la hiérarchie des classes que nous avons conçue pour notre implémentation en langage C++ d'un modeleur de type arbre de construction orienté objet (cf figure 2.4). Chaque classe possède une méthode *help* qui affiche les champs et les méthodes de la classe et qui donne des indications sur leurs significations et leurs fonctionnalités. Chaque classe possède aussi des méthodes pour lire ou modifier ses champs privés. Dans ce qui suit, nous allons décrire brièvement les différentes classes :

- **Objet_CSG** : Cette classe permet de fabriquer des instances qui seront les éléments de l'arbre CSG. Elle contient un champ qui indique si l'objet est le résultat d'une union, d'une intersection, d'une différence, ou si c'est une primitive. Elle possède des méthodes permettant de gérer les relations de dépendance entre objets dans cet arbre. Ainsi, si une instance de cette classe est modifiée, elle est capable de reconnaître les objets qui dépendent d'elle et qui sont alors à modifier selon les liens de dépendance imposés. Par exemple, si on applique une transformation à un noeud *union* de l'arbre CSG, ce nœud est capable de trouver toutes les feuilles du sous-arbre dont il est racine pour éventuellement leur appliquer cette transformation. Les méthodes *union, intersection et différence* créent un nouvel objet de la classe *Objet_CSG* dont les fils, dans l'arbre CSG, sont les paramètres d'appel de ces fonctions. La méthode *écrire* appelle la méthode de même nom dans la classe *décider_A* (voir § 4) pour décrire, dans un fichier, l'objet selon le format nécessité par l'algorithme A. Cette classe possède, en outre, les méthodes suivantes, qui ne nécessitent pas de commentaires particuliers :

définir_sphère_englobante ;
intersection_avec_un_rayon ;
afficher_objet ;
appliquer_translation ;
appliquer_homothétie ;
appliquer_affinité ;
appliquer_rotation ;
appliquer_aspect ;
décider_visibilité ;
décider_lissage ;
décider_fonction_d'éclairement ;
union ;

intersection ;

différence ;

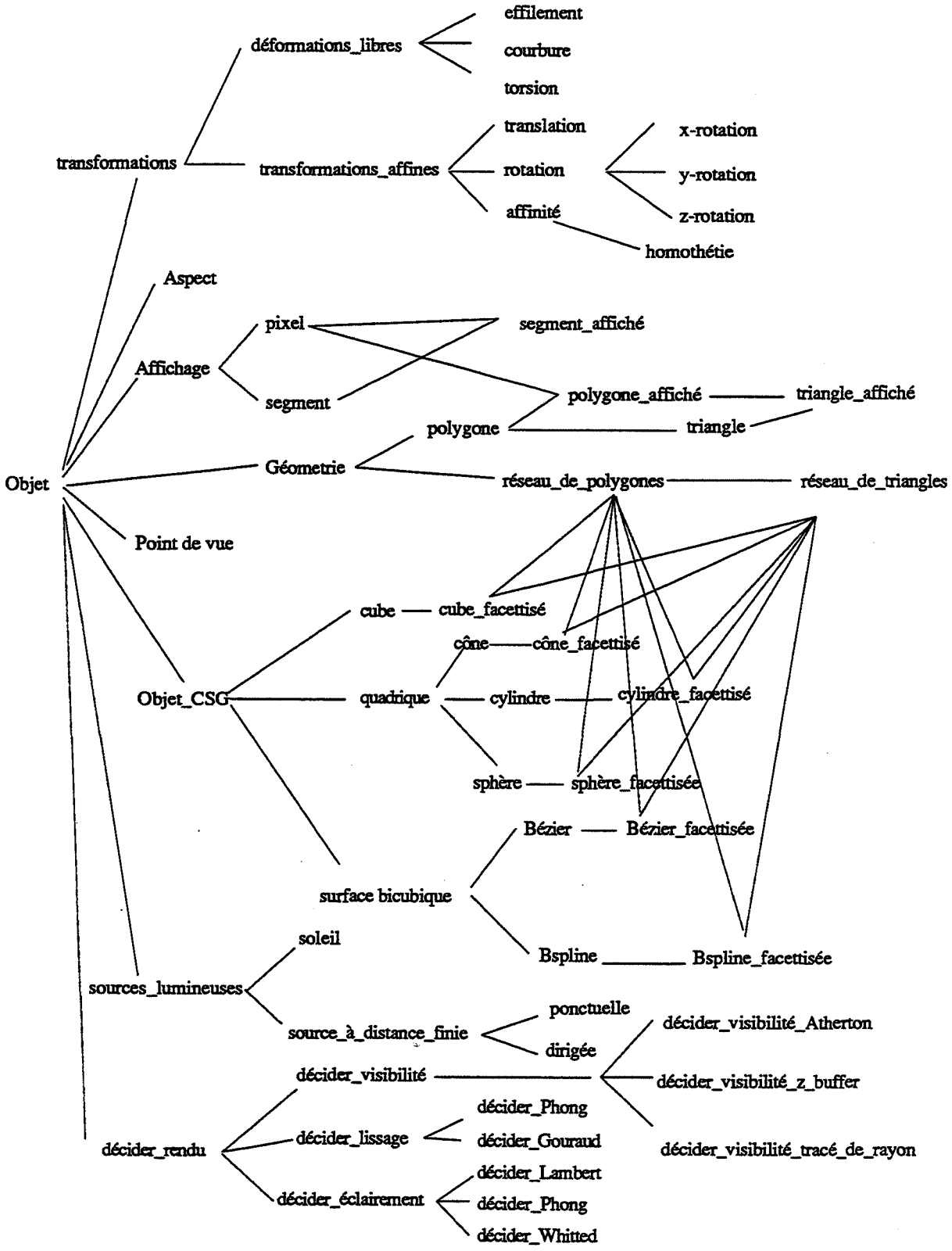


FIGURE 2.4

Cette classe possède des sous-classes : *cube, quadrique, cylindre, cône, ...* qui sont les primitives de modélisation. Ces classes contiennent les mêmes méthodes que celles citées dans la classe *objet_CSG* plus une méthode supplémentaire qui calcule le paramètre de la primitive utilisé pour effectuer le choix automatique de l'algorithme de visibilité (cf paragraphe 4.2). Les sous-classes *sphère_facettisée, cylindre_facettisé, ...* ont été introduites pour résoudre les problèmes signalés dans le paragraphe 4.1. Ces classes contiennent les méthodes qui facettisent ces primitives ainsi que celles qui les visualisent en fil de fer, puisque ces méthodes utilisent la deuxième représentation (réseau de polygones) des primitives de modélisation. Nous utilisons ici l'héritage multiple puisque ces classes héritent des primitives de modélisation et de la classe *réseau_de_polygones.*

La classe Quadrique a été introduite pour factoriser des opérations communes entre ses sous-classes (position d'un point par rapport à une quadrique, messages d'erreurs...). Il en est de même pour la classe *Surface_bicubique* ; celle-ci possède notamment une méthode qui effectue un test d'arrêt de la subdivision récursive de la surface (en comparant la taille de la projection d'un sous carreau sur l'écran par rapport à un pixel).

- **Géométrie**, contient les méthodes :

intersection d'un polygone avec un rayon,
afficher ;
appliquer_translation ;
appliquer_homothétie ;
appliquer_affinité ;
appliquer_rotation ;
appliquer_aspect ;
couper ;
écrire ;

Elle a pour sous-classes les primitives géométriques : *polygone, réseau_de_polygones, triangle, réseau_de_triangles.* Les méthodes de la classe *Géométrie* sont redéfinies pour les classes *réseau_de_polygones* et *réseau_de_triangles.* La sous-classe *polygone_affiché* permet de résoudre les problèmes signalés dans le paragraphe 4.1. Elle contient les méthodes qui affichent un polygone. Cette sous-classe hérite aussi de la classe pixel.

- **Transformation** : Elle possède deux sous-classes:

- la classe *transformation_affine*, redéfinie en sous-classes *translation, rotation* ... Chacune de ces sous-classes contient des champs qui définissent la transformation et des méthodes qui fabriquent la matrice de transformation et qui les appliquent à des points. La classe *transformation_affine* sert à factoriser certaines opérations communes entre les méthodes de ses sous-classes : application d'une matrice à un point, multiplication de deux matrices...

- la classe *déformation_libre* [Nie 89], redéfinie en sous-classe *torsion, courbure, effillement*, contient les opérations communes entre les méthodes tordre, courber, effiller de ses sous-classes (l'interpolation volumique, par exemple).

- **Aspect** : Les informations de rendu telles que la couleur, la visibilité, le lissage, l'éclairement, le coefficient de spécularité... sont les champs de cette classe. Chaque objet de la classe *objet_CSG* pointe sur un objet de cette classe. Cette classe contient des méthodes de conversion entre différents modèles de couleurs et le modèle RVB. Cela permet à l'utilisateur de choisir le modèle de couleur le plus adéquat. Elle contient aussi des méthodes qui permettent de vérifier si certains coefficients sont cohérents. Par exemple les composantes r, v et b doivent être plus petites que 255, les coefficients de transparence et de réflexion positifs et leur somme plus petite que 1, les coefficients de spécularité et de diffusion positifs et plus petits qu'un seuil ... Enfin, cette classe est la classe duale à la fonction *appliquer_aspect* qui reçoit comme paramètres un aspect et l'*objet_CSG* sur lequel elle doit l'appliquer. En effet, en modélisant une scène, nous avons parfois besoin de définir une entité contenant un ensemble d'attributs de rendu et d'appliquer cet ensemble à plusieurs éléments de la scène ; nous avons donc pensé que cette fonction verifiait les conditions de la règle 2 du paragraphe 3.3, et par suite nous avons crée la classe Aspect qui contient la méthode *appliquer_aspect.*

- **Sources lumineuses** : Les sous-classes de cette classe décrivent les sources lumineuses pouvant être utilisées dans une scène. Les champs décrivent les caractéristiques de ces sources:

- la direction de la source et l'intensité pour le soleil ;
- la position, la direction de visée, l'intensité, la couleur de la source, les paramètres qui déterminent la façon suivant laquelle l'intensité diminue avec la distance, pour une source ponctuelle ;

- pour une source dirigée, il existe les mêmes champs, plus les paramètres qui caractérisent le cône lumineux.

La classe *source_à_distance_finie* contient les champs communs aux classes *source_ponctuelle* et *source_dirigée* ; elle possède une méthode translater pour positionner la source dans la scène, et une méthode tourner qui permet de modifier la direction de visée en les faisant *tourner* d'un certain angle autour d'un axe passant par la position de la source. La classe *sources_lumineuses* contient les champs et méthodes communs entre le soleil et les autres genres de sources (intensité d'une source, vérification de la valeur de cette intensité ...).

- **Affichage** : elle possède deux sous-classes qui sont les primitives d'affichage. La classe pixel possède deux champs correspondants aux degrés de sur-échantillonnage dans les deux directions, et trois autres champs qui expriment la couleur R, V, B de ce pixel. La classe *segment* contient deux champs qui décrivent les deux extrémités du segment. Les méthodes permettant d'afficher un segment se trouvent dans la classe *segment_affiché*.

- **Vue** : cette classe contient les données de la prise de vue : position de l'œil, sens du regard, plan près et plan loin, angle de vision, ... Cette classe contient une méthode *translater* pour positionner l'oeil dans la scène, et une méthode *tourner* qui permet de changer la direction de visée en faisant tourner cette direction d'un certain angle autour d'un axe passant par la position de l'oeil.

- **Décider_rendu** : Nous avons décrit ces classes dans le paragraphe 4.2.

5 L'UTILISATION DE C++

5.1 Présentation de C++

C++ [Stro 86] est le résultat des recherches et expérimentations menées par Bjarne Stroustrup dans les années 80 au sein des Bell Labs pour définir un successeur au langage C. Langage à objets, sur-couche de C, C++ comporte la vérification de type et les facilités de la programmation objet. Le langage se présente le plus souvent sous la forme d'un traducteur frontal écrit en C++ traduisant un programme C++ en un programme en C. Sa conception a été guidée par le souci de répondre à quatre objectifs principaux [Bert 88] :

- améliorer le langage C le plus possible, tout en restant compatible avec lui ;
- obtenir un support pour la création de structures de données formelles ;
- ouvrir le langage à la programmation orientée objet ;
- conserver un compilateur efficace produisant du code compact et rapide, particulièrement pour des programmes écrits en C n'utilisant pas les nouveautés.

L'amélioration du langage C se fait sentir facilement, pour un utilisateur de ce dernier, dès qu'il commence à programmer en C++. Parmi ces améliorations, nous pouvons citer :

- l'introduction de la vérification des paramètres de fonctions tout en laissant au programmeur une certaine liberté. L'utilisateur n'est pas emprisonné par un contrôle total comme en Pascal ; il passe du laxisme total de C à la liberté surveillée de C++.

- la possibilité de gérer de manière sûre des fonctions acceptant un nombre variable d'arguments en affectant des valeurs par défaut à ces derniers.

- la surcharge de nom d'une fonction est rendue possible et permet de fournir un nom général commun à des fonctions effectuant le même genre de travail sur des objets de types différents. Le programmeur arrive de cette façon à utiliser des interfaces variables à la fois par le nombre et le type des paramètres.

- l'introduction des références qui permet de rendre équivalents des identificateurs, de façon à ce qu'ils dénotent le même objet (même concept que les liens sur fichiers sous Unix).

- les fonctions expansibles *en ligne* dont la définition est préfixée du mot-clef *inline* demandant de remplacer tous les appels à cette fonction par une expansion de code équivalente au corps de cette fonction. Ces fonctions permettent d'alléger la charge du préprocesseur.

C++ propose aussi d'autres outils et avantages dont une vraie déclaration de constante par le biais du mot-clef *const*.

En tant que langage orienté objet, C++ est directement inspiré de Simula. La classe est une généralisation de la structure C : elle regroupe des variables et des fonctions *en ligne* ou des fonctions ordinaires représentant les méthodes de la classe. Le mécanisme de protection des données et des méthodes est très perfectionnés en C++ : chaque variable

d'instance peut être rendue indépendamment privée, et par suite inaccessible, sauf par une méthode explicite. Elle peut être publique et donc accessible par tous les objets. Elle peut être publique pour une classe mais privée pour les sous-classes de celle-ci. Enfin, la protection des variables d'instance privées peut être affinée en les rendant accessibles à une classe ou fonction déclarée amie.
Toutefois, à la différence de Smaltallk-80, le domaine protégé n'est pas l'instance mais la classe. On ne peut pas cacher des données d'une instance à une autre instance de la même classe.

Le langage fournit un mécanisme permettant d'assurer une instanciation correcte des objets : les *constructeur*. Ce sont des fonctions publiques particulières, portant le même nom que la classe, qui s'occupent, en particulier, de l'initialisation de certaines variables d'instance. C'est le compilateur qui se charge de choisir le constructeur à utiliser, en fonction des paramètres d'instanciation. L'appel du constructeur est géré automatiquement par le compilateur dès la définition d'un nouvel objet instance d'une classe quelconque.

Réciproquement, une fonction unique de nettoyage, appelée destructeur, peut être définie. Elle sert, par exemple, à libérer les places mémoires allouées dynamiquement. L'appel du destructeur, comme celui du constructeur, se fait par le compilateur juste avant que l'objet ne termine sa vie en tant que variable dans le programme.

L'héritage en C++ est simple, mais l'héritage multiple a été ajouté dans les versions les plus récentes. Ce langage est devenu un langage populaire, particulièrement dans le monde UNIX. Il constitue un compromis, consistant à améliorer sensiblement le langage C tout en gardant la compatibilité avec ce langage, sans pénaliser la vitesse d'exécution.

5.2 Quelques problèmes avec C++

Nous avons été confrontés au problème de l'absence d'héritage multiple en C++ (tout au moins la version du langage sur laquelle nous avons travaillé) qu'il a fallu simuler : Supposons que nous voulions que la classe C hérite des classes A et B. Grâce à C++, l'héritage de B vers C, par exemple, est parfaitement défini ; pour simuler l'héritage de A vers C, il suffit d'ajouter une méthode dans la classe C qui reçoit les messages adressés à cette classe et qui redirige ceux destinés en fait à des instances de la classe A. Nous devons donc ajouter aux champs de la classe C un pointeur, *ptr_A*, sur un objet de la classe A. Par ailleurs, il n'existe pas de mécanisme d'envoi de messages en C++ ; si, par exemple, m1 est

une des méthodes de la classe A, nous ajoutons une méthode m1 dans la classe C qui se contente d'appeler la méthode m1 de l'objet de la classe A sur lequel pointe *ptr_A*. Cette technique a cependant un inconvénient majeur : elle augmente de beaucoup le nombre de fonctions de la classe C.

D'autre part, l'utilisation de l'héritage en C++ pour définir un objet CSG n'est pas très simple. Prenons l'exemple suivant :

```
class OBJET {
  ----------;
  ----------;
  public :
  OBJET   une_fonction(OBJET, OBJET) ;
  }

class CUBE : OBJET {          // CUBE est une sous-classe de OBJET
      ---------;
      ---------;
}

class SPHERE : OBJET {     // SPHERE est une sous-classe de OBJET
      ---------;
      ---------;
}

OBJET un_objet ;
CUBE un_cube ;
SPHERE une_sphere ;
un_objet.une_fonction(un_cube, une_sphere) ;
```

Cette dernière instruction amènera, lors de la compilation, des messages et demandera de forcer la conversion de un_cube et une_sphère à OBJET. On ne peut demander à un utilisateur de faire explicitement la conversion à chaque emploi d'une primitive :

```
un_objet.une_fonction( (OBJET) un_cube, (OBJET) une_sphere) ;
```

Pour éviter ce problème nous pouvons utiliser la possibilité de surcharger les fonctions en C++ :

```
class OBJET {
  --------
  --------
  public :
OBJET une_fonction(OBJET, OBJET) ;
OBJET une_fonction(CUBE, OBJET) ;
OBJET une_fonction(OBJET,CUBE) ;
OBJET une_fonction(SPHERE, OBJET) ;
OBJET une_fonction(OBJET, SPHERE) ;
}
```

Ce qui devient rapidement fastidieux quand on a beaucoup de classes et de méthodes. De plus, cela oblige à modifier la classe OBJET dès que l'on veut ajouter une nouvelle primitive ce qui est plutôt contradictoire avec la philosophie de la programmation orientée objets.

Nous avons choisi de travailler sur un seul type OBJET. Pour créer un cube, il faut appeler une fonction CUBE qui retournera un objet de type OBJET qui sera initialisé aux valeurs d'un cube.

```
OBJET obj ;
obj = CUBE(r, v, b) ;
```

Pour cela nous avons développé une interface en définissant les fonctions: SPHERE, CUBE, CYLINDRE, ... qui retournent un OBJET et permet alors d'éviter les problèmes signalés plus haut.

Les fonctions REUNION, INTERSECTION, DIFFERENCE, TRANSLATION, ROTATION, ... ont été introduites parce que nous avons trouvé qu'il est plus naturel de les utiliser à la place de la syntaxe habituelle, par exemple :

```
OBJET obj1, obj2, obj3
obj1 = CUBE(r, v, b) ;
```

obj2 = CYLINDRE(r1,v1, b1) ;

obj3 = obj1.reunion(obj2) ; *est équivalente à* obj3 = REUNION(obj1, obj2) ;
obj3 = obj3.homothétie(a) ; *est équivalente à* obj3 = HOMOTHETIE(obj3, a) ;

Cependant, cette notation fonctionnelle est un peu verbeuse. Une troisième notation a été introduite (les trois notations sont compatibles, elles peuvent donc être utilisées en même temps). Elle se base sur la redéfinition des opérateurs que le langage C++ rend possible, l'union se définit naturellement sous forme d'addition :

obj3 = obj1 + obj2 ;
Le caractère associatif est conservé :
obj5 = obj1 + obj2 + obj3 + CUBE(r, v, b) ;

les opérateurs utilisés sont :
obj + obj -> union
obj - obj -> différence
obj * obj -> intersection
obj << vect -> selon le type de vect (ROTA, TRANS, HOMO, AFFI),
vect >> obj c'est une rotation, une translation, une homothétie
... sur obj. ROTA, TRANS, HOMO, AFFI étant des
classes définissant des applications affines.

Pour l'instruction (A = B), si on fait une copie bit à bit, qui est l'option par défaut en C++, cela peut poser des problèmes, puisque les pointeurs qui existent dans A et B seront identiques, et que nous voulons éviter les effets de bord. Nous redéfinissons, par conséquent, l'opérateur affectation qui fera une recopie.

5.3 Un exemple de description de scène en CastorC++

Nous allons présenter comme exemple de description de scène en C++ la construction d'un escalier.

include <objet.h>
// inclusion du fichier contenant la description des types et des fonctions

```
// pouvant être utilisés
define couleur_mur 180, 180, 180

créer()  {

double hauteur_marche   = 0.5 ;
double hauteur          = 2.1 ;
double angle            = 10.3 ;

OBJET piece     = AFFINITE( CUBE(Couleur_mur), 10, 10, 2.3) ;
OBJET marche    = AFFINITE(CUBE(100,100, 80), 2, 1, hauteur_marche) ;
OBJET escalier = marche ;

TRANS trans(0, 0,hauteur_marche) ;
ROTA  rota(0, 0, 1, angle) ;

while (trans[2] < hauteur)  {
        escalier   = escalier + (marche << rota) << trans ;
        rota[3]    = rota[3] + angle ;
        trans[2]   = trans[2] + hauteur_marche ;
        }
VISU(escalier) ; VISU(piece) ;
}
```

L'utilisateur construit la scène en définissant la fonction qui s'appelle créer(). Cette fonction est ensuite compilée puis liée aux bibliothèques qui contiennent les fonctions de création des primitives.

6 Conclusion

Dans notre souci d'atteindre un niveau satisfaisant de flexibilité et d'évolutivité, nous avons été amenés à faire une étude sur l'adaptation d'un système de synthèse d'images à la programmation orientée objet. En effet, nous avons remarqué que la progression de ce style de programmation pour la réalisation de systèmes de synthèse d'images est lente par rapport à d'autres domaines de l'informatique. Certes, la lenteur d'exécution que peut apporter cette technique de programmation, à cause du mécanisme

des envois de messages, y est pour quelque chose ; mais ce n'est pas, à notre avis, la seule raison. En effet, nous avons mis en cause la hiérarchie classique des classes qu'on trouve dans la littérature, et nous avons proposé une autre hiérarchie qui nous a semblé plus efficace pour atteindre nos buts de flexibilité et d'évolutivité.

Les méthodes de choix automatiques d'algorithmes d'élimination des parties cachées, de techniques de lissage et de fonctions de rendu pour un objet donné, que nous avons exposées dans ce chapitre, sont, en grandes parties, basées sur un test de la taille de la boîte englobante de cet objet sur l'écran. D'autres approches plus sophistiquées peuvent être envisagées, pour que ce choix automatique soit plus performants.

Chapitre 3

Mixage d'algorithmes en synthèse d'images

1 INTRODUCTION

Le choix d'un algorithme permettant de résoudre le problème de l'élimination des parties cachées ([Suth 74], [Pero 88]) dépend du type d'objets auquel il s'applique, du type de modélisation de la scène accepté, du degré de réalisme souhaité, du temps de calcul admis, ... Dans la plupart des systèmes de synthèse d'images, le même algorithme sert à calculer toute la scène. Or, dans une scène, on peut trouver des objets de tous types : polygones, quadriques, surfaces gauches, ... L'algorithme de visualisation utilisé n'est pas forcément adapté à tous ces types. Par exemple, si on utilise un algorithme travaillant sur des facettes, celui-ci nécessite l'approximation d'une sphère ou d'un carreau bicubique par un ensemble de facettes pour pouvoir les visualiser, ce qui peut nuire au réalisme de l'image. Pour visualiser ces deux primitives de modélisation, nous préférerions utiliser un autre algorithme qui ne fasse pas recours à la facettisation, comme, par exemple, l'algorithme du tracé de rayons. Une solution pourrait être d'utiliser le tracé de rayons pour calculer la visibilité de toute la scène. Or, cet algorithme, s'il produit des images très réalistes, est coûteux en temps de calcul, et ce coût augmente avec le nombre d'objets dans la scène. Comme il existe souvent des contraintes sur le coût de calcul d'une image, on a intérêt à ce que la base de données sur laquelle travaille le tracé de rayons soit la plus réduite possible. Pour cela, nous préférons utiliser un algorithme travaillant sur des facettes, qui est moins coûteux (mais moins performant) que le tracé de rayons, pour visualiser les objets de la scène dont la facettisation ne nuit pas au réalisme de l'image, comme les cubes ou les sphères et les carreaux bicubiques qui sont éloignés de l'observateur.
D'où l'idée de faire un mixage d'algorithmes de visualisation dans une même scène, chaque objet étant visualisé par la méthode la mieux adaptée. En continuant avec le même exemple, le cube sera visualisé par l'algorithme travaillant sur des facettes, alors que la sphère et le carreau bicubique seront visualisés par le tracé de rayons.

Nous ne sommes pas les premiers à avoir posé ce problème. Crow [Crow82] a proposé un système en deux étapes. D'abord, il détecte des groupes d'objets indépendants les uns par rapport aux autres pour la visibilité et leur affecte un ordre de priorité pour l'affichage. Ensuite l'image est fabriquée en affichant ces groupes d'objets les uns après les autres selon leur priorité. Pour chaque groupe d'objets, l'algorithme de visibilité le plus approprié est utilisé.

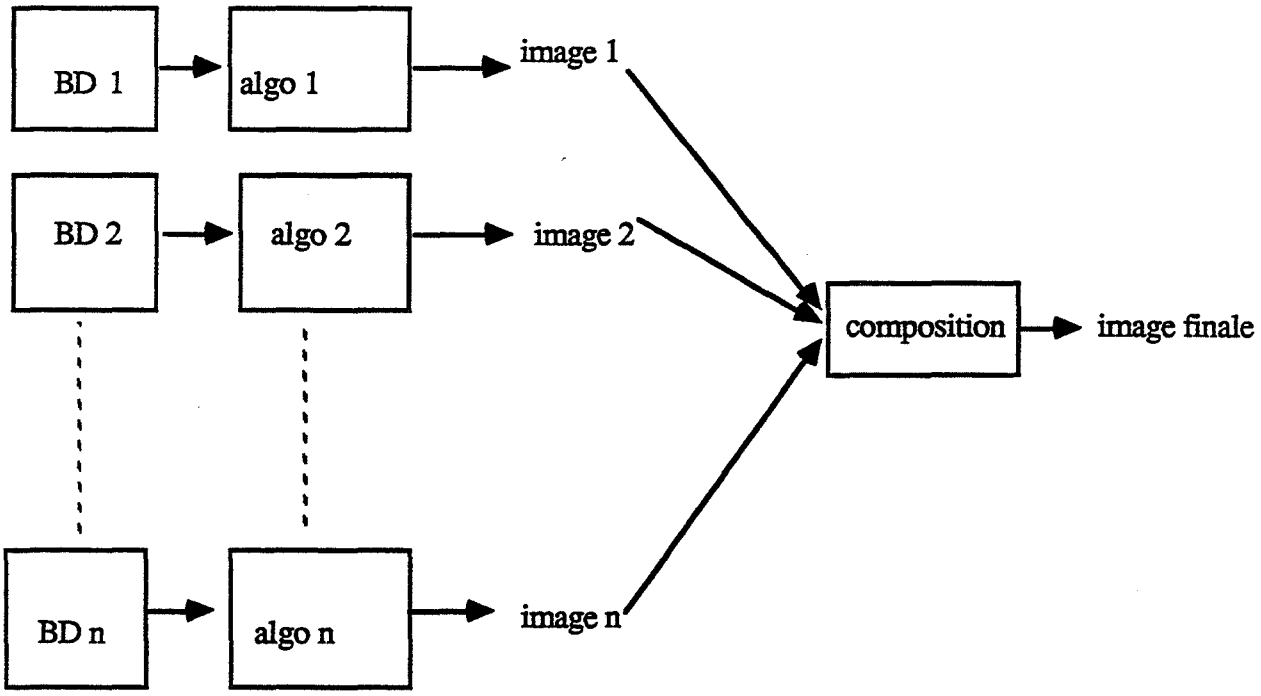


FIGURE 3.1

D'autres méthodes sont basées sur la composition des sorties de plusieurs algorithmes pour fabriquer une image finale [Duff 80]. La figure 3.1 schématise cette idée de composition : chaque méthode reçoit une partie de la scène et en fabrique une image. Après avoir fabriqué les n images partielles, il faut en déduire l'image finale. C'est le rôle de la boîte *composition* qui reçoit pour cela plusieurs types d'informations. En particulier, elle reçoit, en chaque pixel et pour chaque algorithme de visualisation, la profondeur du point vu en ce pixel. L'algorithme de composition effectue une comparaison entre les profondeurs des points vus en un même pixel et affiche la couleur du point dont la profondeur est la plus petite. Nous appelerons cette technique la méthode de composition dans la suite de ce chapitre.

Utilisée de cette façon, cette méthode est responsable de problèmes d'aliassage. En effet, un des cas où le phénomène de l'aliassage se manifeste est lorsqu'un premier objet de la scène se projette sur une partie d'un pixel, et un second objet sur l'autre partie. Si on a recours à un seul algorithme d'élimination des parties cachées pour toute la scène, la méthode d'antialiassage utilisée par cet algorithme réalise un certain mélange des couleurs de ces deux objets ; ce mélange sera la couleur affectée à ce pixel. Si on utilise une composition entre deux algorithmes d'élimination des parties cachées, il est possible que chacun des deux objets appartienne à une base différente. Dans ce cas, chaque méthode d'antialiassage, associée à un algorithme d'élimination des parties cachées mélangera la couleur de l'objet avec celle d'un fond incorrect.

Pour résoudre ce problème, Porter et Duff [Port 84] introduisent la notion de *canal alpha* . Pour chaque pixel, chacun des algorithmes d'élimination des parties cachées fournit une information supplémentaire *alpha* à l'algorithme de composition : il s'agit du pourcentage de la surface du pixel couvert par l'objet vu en ce pixel. Grâce à cette information, l'algorithme de composition effectue la mélange des couleurs pour chaque pixel et traite ainsi l'aliassage. Duff [Duff 85] améliore cette méthode en calculant les profondeurs aux quatre coins du pixel et en interpolant ces valeurs pour calculer la valeur de *alpha.*

D'autre part, la séparation complète des différentes bases de données, alors qu'elles constituent ensemble la scène finale, rend les interactions entre elles très difficiles. C'est le cas des effets de réflexion, de transparence ou d'ombre portée d'un objet de la base i sur un objet de la base j. En effet, la méthode de composition décrite ci-dessus se contente de s'occuper de l'aspect élimination des parties cachées entre les bases de données correspondant aux différents algorithmes de visualisation, et ne suggère rien pour le rendu. L'objectif du travail présenté dans ce chapitre est de proposer une autre méthode de composition entre les algorithmes d'Atherton et du tracé de rayons, qui font partie des algorithmes de visualisation disponibles dans le système ILLUMINES (cf chapitre 1), pour y introduire plus de flexibilité. Cette méthode permet de prendre en compte les interactions entre les objets de la scène (réflexions, transparences, ombres portées), même quand ils n'appartiennent pas à la même base de données. Nous l'appellerons la méthode de composition de rendu, par opposition à la composition, décrite ci-dessus, d'élimination des parties cachées.

[Crow82] a utilisé deux méthodes très simples pour résoudre les problèmes d'interactions entre les deux bases de données dans le cas du mixage de l'algorithme du tracé de rayons et d'un algorithme travaillant sur des facettes : une première méthode, pour assurer l'interaction en termes de réflexion et de transparence, consiste à ajouter la base de données de l'algorithme travaillant sur des facettes à celle du tracé de rayons. Ce dernier ne tiendra compte de cette partie de la base que pour le calcul des rayons réfléchis et réfractés. Cette méthode sera appelée la méthode de Crow pour les réflexions et les transparences dans la suite de ce chapitre. Son inconvénient est que chacun des deux algorithmes travaille tout seul, sans échanger des informations susceptibles de faciliter ses travaux, comme par exemple l'accélération du calcul des rayons réflechis et réfractés en tracé de rayons.

Une seconde méthode, pour assurer l'interaction en termes d'ombres portées, consiste à convertir la base de données du tracé de rayons en polygones et à calculer alors

une image de toute la scène à partir de la source lumineuse qu'on appelle image auxiliaire. Cette dernière sera alors utilisée pour trouver les ombres portées. Cette méthode sera appelée la méthode de Crow pour les ombres portées dans la suite. En plus de l'inconvénient cité pour la première méthode, cette technique facettise les objets de la base TR, ce qui peut nuire au réalisme de l'image.

Dans le paragraphe 2, nous présenterons la solution que nous proposons pour permettre les interactions en termes de réflexion et de transparence entre les deux bases de données du tracé de rayons et de l'algorithme d'Atherton. Cette solution aboutira à une accélération du calcul des rayons réfléchis et réfractés dans le tracé de rayons. Dans le paragraphe 3, nous présenterons notre solution pour rendre possibles les interactions en termes d'ombres portées. Dans le paragraphe 4, nous exposerons les résultats des tests effectués pour évaluer les solutions proposées. Enfin, nous conclurons dans le paragraphe 5.

2 MIXAGE DES ALGORITHMES D'ATHERTON ET DU TRACE DE RAYONS AVEC INTERACTIONS EN REFLEXION ET EN TRANSPARENCE

Dans ce paragraphe, nous proposons notre méthode de composition de rendu permettant l'interaction en réflexion et réfraction entre les deux bases de données d'une composition entre le tracé de rayons et l'algorithme d'Atherton. Nous montrerons que cette méthode permet d'accélérer le calcul des rayons réfléchis et réfractés en tracé de rayons par rapport à la méthode de Crow ([Crow 82]).

2.1 Description de la méthode

La base de données de la scène est scindée en deux parties: la première, appelée TR, sur laquelle on utilise le tracé de rayons pour l'élimination des parties cachées ; la seconde, baptisée AT, est affectée à l'algorithme d'Atherton. C'est le modeleur qui assure cette scission. Par exemple, sauf indication contraire de l'utilisateur, il affecte les sphères et les objets réfléchissants et/ou transparents à la base TR, et les cubes diffus à la base AT. Les objets de petite dimension sur l'écran sont affectés à la base AT. Si deux objets différents sont liées par une opération booléenne intersection ou différence, ils sont affectés au même algorithme de visualisation. Par exemple, supposons qu'une sphère et un cube font partie de la scène, et qu'ils sont liés par une opération d'intersection. Si l'utilisateur privilégie le réalisme sur la rapidité, ils sont affectés tous les deux au tracé de rayons si la

taille de la sphère est suffisamment grande sur l'écran. Nous supposerons, dans un premier temps, que tous les objets réfléchissants et/ou transparents dans la scène sont affectés au tracé de rayons.

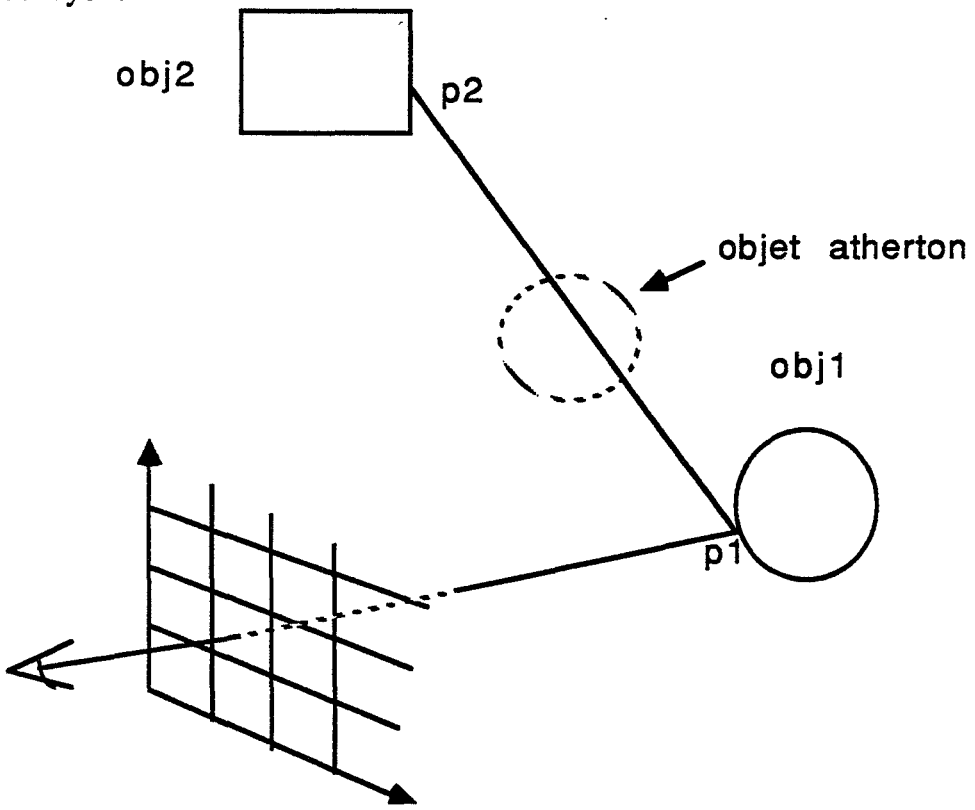


FIGURE 3.2

Notre méthode de composition de rendu que nous proposons dans ce chapitre fonctionne de la manière suivante :

Nous appliquons d'abord l'algorithme d'Atherton sur sa base de données et nous obtenons en sortie trois fichiers : un premier contenant la couleur rvb de chaque pixel, un second contenant la profondeur du point vu en chaque pixel, et un troisième contenant la liste des objets vus au centre de chaque pixel. Nous appliquons alors l'algorithme du tracé de rayons à la base TR. Supposons qu'un rayon primaire heurte un objet réfléchissant (ou réfractant) *obj1* en p1, un rayon réfléchi (réfracté) est alors formé. Nous testons son intersection avec la base de données TR. Supposons qu'il intercepte un objet *obj2* au point p2 (dans le cas où ce rayon n'intercepte pas d'objets, on prend son intersection avec la boîte englobante de la scène). Il s'agit de savoir s'il existe un objet atherton sur le trajet de ce rayon entre p1 et p2 (cf figure 3.2). Pour cela, on transforme les deux points p1 et p2 du repère du monde dans le repère où l'algorithme d'Atherton effectue ses calculs (qui est tel que l'axe des z ait même direction et même sens que le regard, et que le plan xoy soit le plan de l'écran). Ce repère

sera appelé par le repère AT dans la suite. On obtient alors les deux points M1 et M2. On les projette sur le plan de l'écran et on obtient le segment de droite M'1M'2. En utilisant l'algorithme de Bresenham, on peut trouver les pixels traversés par ce segment.

En partant de M'1, on prend le premier pixel appartenant au segment. La liste des objets de la base AT vus en ce pixel est connue d'après le troisième fichier fourni par l'algorithme d'Atherton. Le tracé de rayons utilise alors la base AT et teste l'intersection du rayon avec cette liste d'objets. S'il y a des intersections, il prend celle qui est la plus proche du point p2 et il s'arrête sinon, il recommence avec le pixel suivant du segment M'1M'2. Si aucune intersection n'est trouvée pour tous les pixels du segment M'1M'2, alors aucun objet atherton fantôme n'intercepte le rayon entre les points p1 et p2.

Nous disposons à présent d'une liste d'objets primitifs par pixel. Pour un pixel donné, il faut extraire à partir de l'arbre représentant la scène complète un arbre simplifié ne contenant que les objets existants dans la liste.

La méthode qui consiste à parcourir l'arbre entier en notant les objets dont les numéros correspondent prend un temps trop long. Pour éviter d'explorer des branches sans interêt, l'algorithme d'Atherton fournit avec le numéro de chaque objet le chemin d'accés dans l'arbre si celui-ci comporte des opérations booléennes intersection ou différence. Le tracé de rayons subdivise, pour chaque nœud, la liste des objets en une liste droite et une liste gauche. Cette subdivision est simple à faire et donc rapidement calculée. C'est cette même solution qui a été retenue dans [Exco 87].

Avec ce renseignement, nous ne passons que par les nœuds utiles. D'autre part, ceci permet d'éliminer des parcours inutiles : c'est le cas, si l'opérateur est l'intersection par exemple, et si le liste des objets à droite ou à gauche est vide.

Voici un exemple de simplification d'arbre :

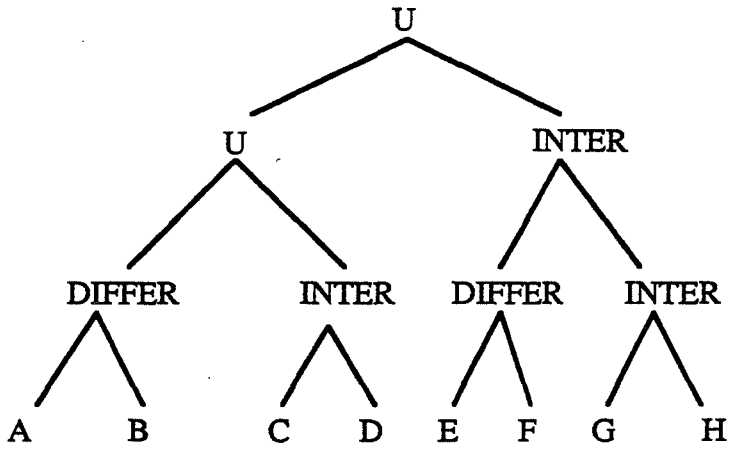


FIGURE 3.3 (a)

Supposons que les objets visibles soient A, C, D, G, H. Voici, sous forme d'un arbre, les différentes phases du calcul. Sur chaque nœud, on indique la liste des objets situés en dessous de lui dans l'arbre. A chaque niveau, connaissant le chemin des objets, on subdivise la liste en deux sous-listes.

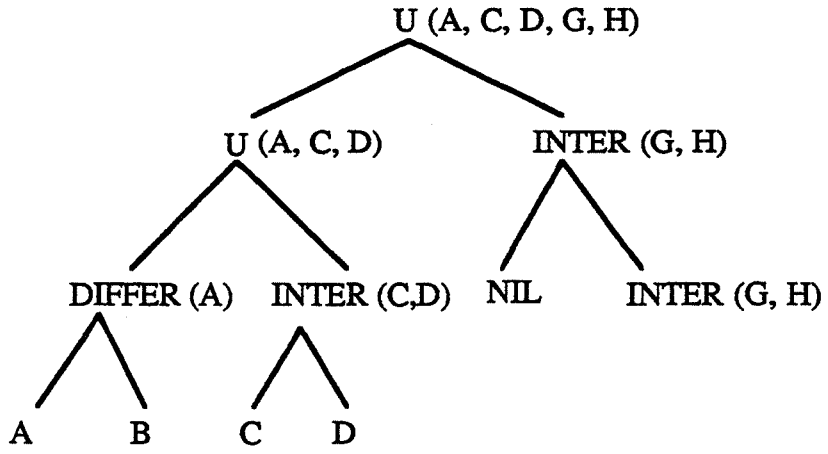


FIGURE 3.3 (b)

Dans cet exemple, lorsque la routine de simplification est arrivée à INTER(G,H), elle est immédiatement arrêtée car la liste gauche est vide. Le résultat est donc une liste de deux arbres : le premier est l'arbre composé de A et le second est INTER(C,D).

Ce travail de simplification d'arbre n'est pas à répéter en chaque pixel. Il est réalisé une seule fois pour l'ensemble des pixels qui possédent la même liste d'objet. Dans le paragraphe 2.3, nous expliquerons la façon suivant laquelle nous déterminons l'ensemble des pixels possédant une même liste.

Outre qu'elle permet de réaliser les interactions en réflexion et en transparence, nous pouvons remarquer que cette méthode de composition de rendu permet une optimisation pour le calcul des intersections entre un rayon réfléchi ou réfracté et les objets de la base AT. C'est là le principal avantage de cette méthode par rapport à la méthode de Crow, et qui repose sur le fait que les deux algorithmes d'Atherton et de tracés de rayons échangent des informations. Cette optimisation n'est pas nouvelle en tracé de rayons. C'est la méthode qui consiste à projeter le rayon sur une grille régulière et à tester son intersection avec les listes des objets qui se projettent sur les cellules de la grille traversées par le rayon [Bron 84]. La seule différence est que le pré-traitement destiné à déterminer les objets qui se projettent sur chaque cellule est réalisé par l'algorithme d'Atherton.

Remarque : s'il existe des objets atherton en dehors du champ de vue, ils ne se projettent sur aucun pixel. Or, ces objets peuvent intervenir pour le calcul de réflexion et de réfraction en tracé de rayons. [Hall 86] signale que l'influence de ces objets sur l'image est limitée, et propose d'utiliser la méthode de Blinn [Blin 76] qui est plus rapide que le lancer de rayons secondaires, pour résoudre le problème soulevé ci-dessus. Cette méthode consiste à positionner un objet réfléchissant au centre d'une sphère, à l'intérieur de laquelle une image de l'environnement est peinte. Pour trouver la réflexion en un point de cet objet, il suffit de trouver l'intersection du rayon réfléchi, partant de ce point, avec la sphère. Dans notre méthode, pour résoudre ce problème, on transmet au tracé de rayons tous les objets atherton en-dehors du champ de vue. On teste l'intersection entre un rayon et ces objets si la projection de ce rayon dépasse les limites de l'écran.

2.2 Avantages de la méthode

Comme nous l'avons signalé ci-dessus, le principal intérêt de notre méthode de composition de rendu par rapport à celle de Crow est qu'elle profite du travail effectué par l'algorithme d'Atherton sur sa propre base pour offrir une optimisation du calcul des intersections entre un rayon réfléchi ou réfracté et les objets de la base de données AT. Cette optimisation procure les avantages suivants :

1) Pas de pré-traitement : c'est le calcul éxécuté par l'algorithme d'Atherton qui le remplace pour les objets de la base AT.

2) Une subdivision très fine: nous avons une liste d'objets par pixel. Cela améliore beaucoup l'optimisation parce qu'il permet une séparation maximale des objets de la scène. Généralement on préfère utiliser un quadtree ou une grille régulière dont la taille d'une cellule est plus grande que celle d'un pixel, à cause de la place mémoire nécessaire. Dans notre méthode, nous arrivons à avoir une liste d'objets par pixel tout en évitant le problème de la place mémoire. Ce problème sera discuté au paragraphe 2.3.

3) Pas d'utilisation de boîtes englobantes : la méthode d'optimisation la plus classique en tracé de rayons consiste en l'utilisation de boîtes englobantes. Si on prend pour boîtes englobantes des primitives simples comme des parallélépipèdes à faces parallèles aux axes de la scène, des sphères ou des ellipsoïdes, leur manipulation sera très facile mais il y aura perte de précision (l'exemple typique est un long cylindre incliné à projeter sur un des plans du repère). Si on prend des englobants qui approximent bien l'objet (polyèdre [Kay 86], volumes englobantes pour des objets procéduraux [Kaji 83], [Bouv 85]), leur fabrication peut prendre un temps important diminuant le gain espéré. Dans notre méthode, grâce à l'algorithme d'Atherton, nous savons comment se projette chaque objet de la base AT sur la grille des pixels.

4) L'information de tri : Comme nous l'avons déjà signalé dans le chapitre 1, la liste des objets vus en un pixel peut être triée par l'algorithme d'Atherton. Ce tri est utilisé dans le tracé de rayons, de la façon suivante :
Considérons le segment M'1M'2 (défini dans le paragraphe 2.1) ; nous devons tester l'intersection du rayon avec la liste des objets associée à chaque pixel traversé par M'1M'2. Appelons zcourant1 et zcourant2 les profondeurs des intersections du rayon avec les plans parallèles à l'axe des z, dans le repère AT, et passant, soit par les bords verticaux du pixel courant, si l'angle que fait la projection du rayon sur le plan de l'écran avec l'axe des x est plus grand que 45 degré, soit par les bords horizontaux, sinon (le calcul de zcourant1 et de zcourant2 se fait de façon incrémentale). Si Zmin (resp Zmax) de la boîte englobante d'un objet est plus grande (resp plus petite) que max(zcourant1, zcourant2) (resp min(zcourant1, zcourant2)), alors le rayon n'intercepte ni cet objet ni les objets situés derrière lui (resp devant lui) dans la liste. On peut même faire une recherche dichotomique pour cela.

D'autre part, supposons que le rayon intercepte un objet primitif. Si le rayon est dirigé dans le sens des z positifs (resp. des z négatifs), nous arrêtons de tester l'intersection du rayon avec les objets situés derrière lui (resp. avant lui) dans la liste. Pour pouvoir faire la même chose dans le cas où cet objet est un sous arbre, il faut que l'intervalle [Zmin,Zmax] de sa boîte englobante soit disjoint de ceux des boîtes des autres objets de la liste.

Cette information de tri s'avère particulièrement intéressante lorsque les objets vus en un pixel sont nombreux, et lorsqu'il existe, parmi eux, des objets complexes dont le test d'intersection avec un rayon soit coûteux en temps de calcul. Pour cette raison, l'algorithme d'Atherton effectue, pour chaque pixel, le tri en profondeur de la liste des objets vus en ce pixel si la condition précédente est remplie.

Dans le paragraphe 3, nous décrirons les résultats d'un certain nombre de tests permettant de vérifier pratiquement ces avantages.

Remarque 1

La figure 3.4 montre deux objets *obj1* et *obj2* dont les projections sur l'écran se partagent un même pixel, ce qui risque de provoquer de l'aliassage. C'est *obj1* qui est vu au centre du pixel. Il se situe, alors, devant *obj2* dans la liste des objets vus en ce pixel. Supposons qu'un rayon intercepte *obj1* en I1 et qu'il se dirige, pour fixer les idées, dans la direction des z positifs. On s'arrête alors de tester l'intersection de ce rayon avec les objets situés derrière *obj1* dans la liste dont l'objet *obj2* ; par suite le point d'intersection retenu est donc I1 à la place de I2. Cela n'est pas gênant puisque c'est *obj1* qui est vu en ce pixel, car c'est l'image avec aliassage que nous voyons.

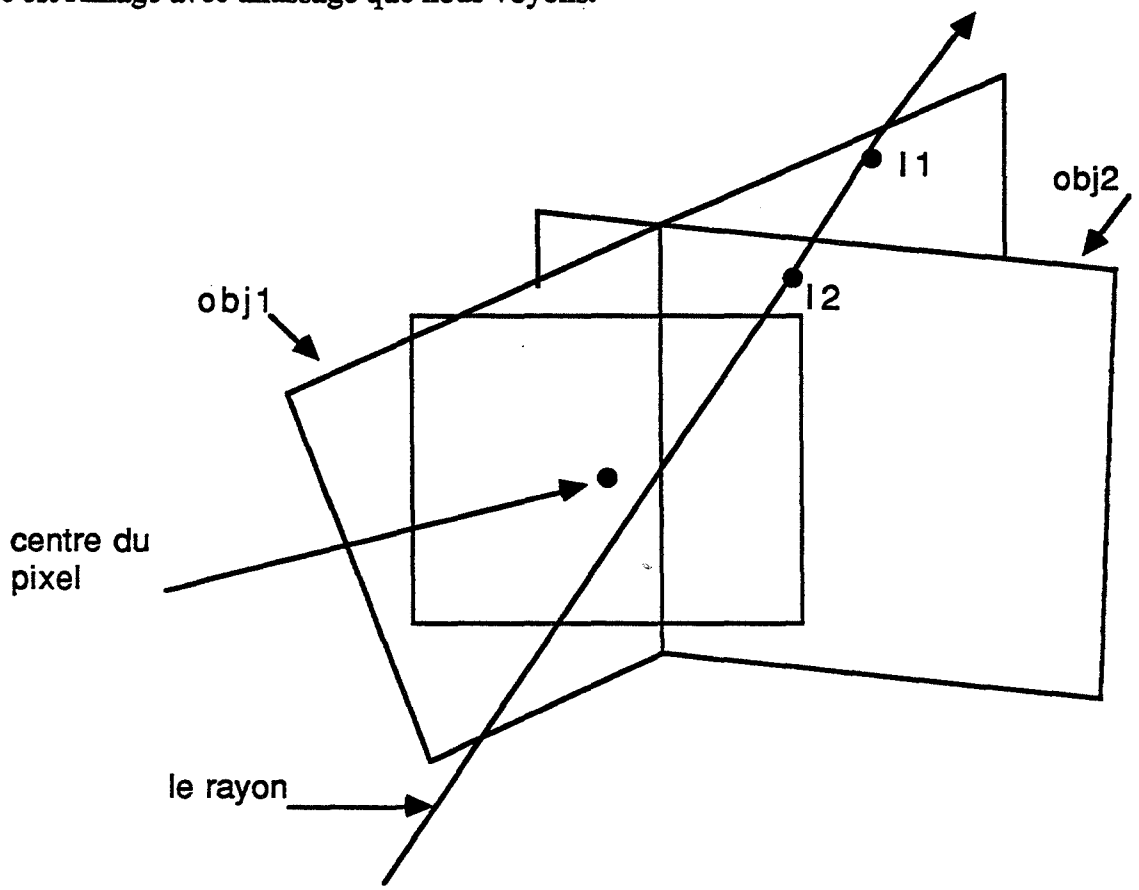


FIGURE 3.4

D'autre part, grâce aux tests entre Zmin et Zmax de la boîte englobante de l'objet et zcourant1 et zcourant2, nous évitons d'effectuer le calcul d'intersection du rayon avec un objet tel que le point d'intersection ne se projette pas sur le pixel courant mais sur un pixel situé plus loin sur le trajet du rayon. En effet, sur la figure 3.5, nous ne faisons pas de test d'intersection entre le rayon et *obj1* avant d'arriver sur le pixel où se projette le point A, bien que cet objet se projette sur d'autres pixels traversés par le rayon avant d'arriver au pixel contenant le point A1. Notons que si, malgré ces tests, le point d'intersection entre un rayon et un objet ne se projette pas sur le pixel courant, ce point est sauvegardé pour ne pas avoir à refaire le même calcul lorsqu'on passe au pixel suivant sur le trajet du rayon.

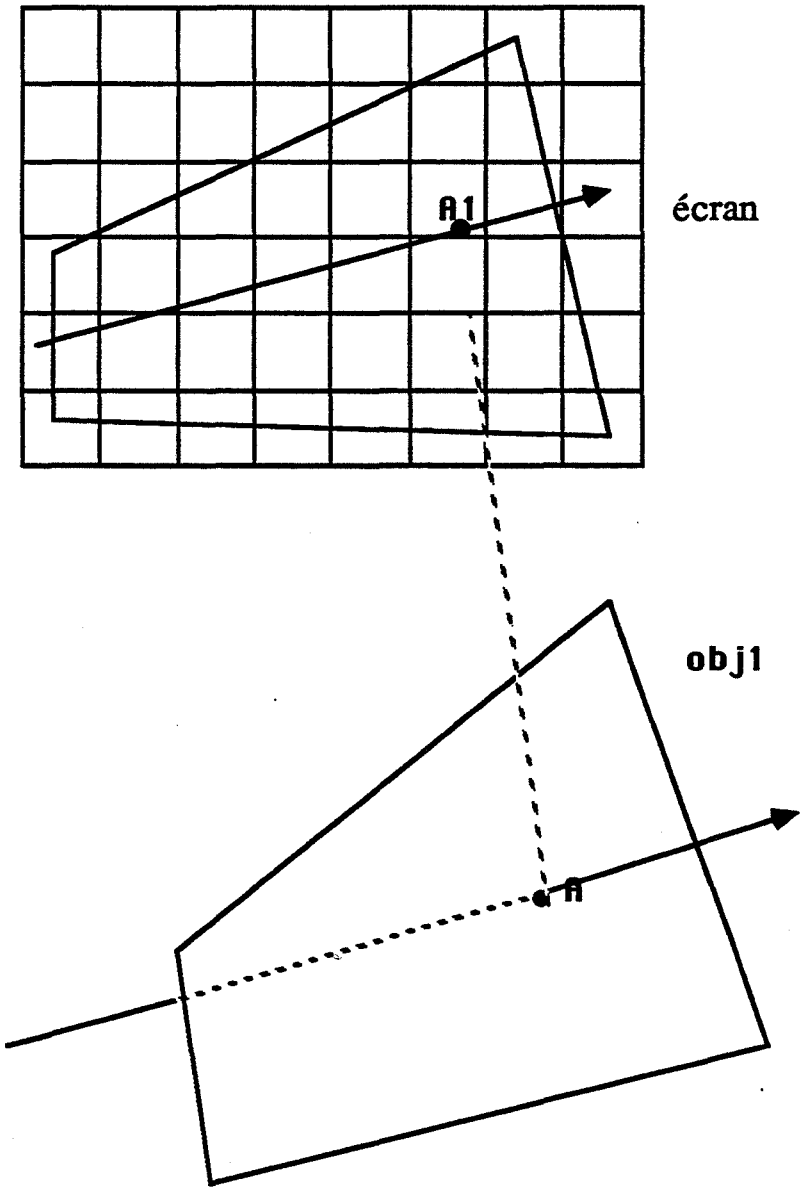


FIGURE 3.5

Remarque2 :

Supposons que le rayon soit dirigé vers les z négatifs. Sur la figure 3.6, l'objet *obj2* est plus éloigné de l'œil que *obj1*. *obj2* est donc placé après *obj1* dans la liste des objets se projetant sur le pixel considéré. Si le rayon intercepte d'abord obj2 en I2, alors son intersection I1 avec obj1 ne sera pas testée. Ainsi, au lieu de prendre I1 comme intersection, on prend I2. Cette erreur provient du fait qu'on trie les objets de la liste dans un seul sens, du plus proche au plus éloigné.

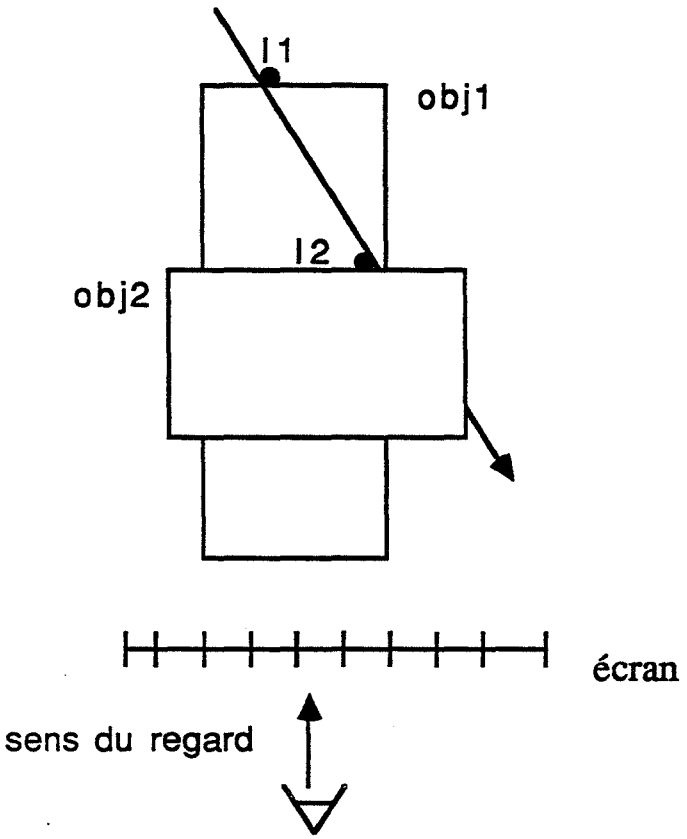


FIGURE 3.6

Pour remédier à ce problème, il faudrait que la méthode d'Atherton fournisse deux listes triées chacune dans un sens. Cela nécessiterait de la place mémoire supplémentaire. Nous avons préféré détecter les pixels dont les listes font apparaître ce problème. Pour cela, nous procédons de la façon suivante : soit E_i et S_i respectivement l'entrée et la sortie de l'objet numéro i dans la liste attachée au pixel courant (chaque objet dans la scène posséde un numéro). Nous avons alors deux suites : une suite des entrées E_i, E_j, E_k, ... et une autre des sorties S_i, S_j, S_k, ... A chacune de ces deux suites correspond une suite de numéros

d'objets i, j, k, ... Un problème existe en ce pixel si ces deux suites de numéros d'objets ne sont pas identiques et classées dans le même ordre. Pour l'exemple de la figure 3.6, la suite des entrées est E1, E2, celle des sorties est S2, S1. Les suites des numéros d'objets correspondants sont 1,2 et 2,1 respectivement. On remarque bien qu'ils ne sont pas dans le même ordre, d'où le problème présent dans cet exemple.
La méthode d'Atherton fournit une information supplémentaire au tracé de rayons pour l'avertir de l'existance d'un tel problème en un pixel. Par suite, ce dernier n'utilise pas l'information de tri en ce pixel.

Remarque 3 :

A cause de la présence d'un objet réfléchissant et/ou réfractant dans un sous-arbre contenant beaucoup d'objets, on peut être amené à affecter l'ensemble de ce sous-arbre au tracé de rayons. De cette façon, la base TR risque de devenir très grande alors que nous préférerions la garder la plus petite possible, ne contenant des objets comme ceux réfléchissants et réfractants par exemple, ou les objets que nous ne dèsirons pas facettiser ; par contre, nous souhaitons affecter le maximum d'objets à la l'algorithme d'Atherton qui est plus rapide. Pour remédier à cet inconvénient, on affecte ce sous-arbre à la base AT et on procède ainsi : supposons que, pendant le déroulement de l'algorithme d'Atherton, on voit en un pixel un objet réfléchissant et/ou réfractant. On transmet cette information au tracé de rayons, ainsi que les coordonnées du point vu, la normale en ce point et d'autres renseignements concernant l'aspect de cet objet. Ensuite, lors du calcul de la visibilité sur sa propre base, le tracé de rayons teste s'il existe une telle information pour chaque pixel. Si c'est le cas, il lit les renseignements fournis par l'algorithme d'Atherton, et lance alors des rayons réfléchis et/ou réfractés à partir de ce point. De cette façon, on peut affecter le sous-arbre à la base AT sans perdre l'avantage des effets optiques du tracé de rayons.

2.3 La gestion de la place mémoire

Parmi les avantages signalés dans le paragraphe précédent, nous avons cité la subdivision très fine du plan de projection (l'écran). Mais ceci ne peut-il pas être pénalisant en place mémoire puisqu'on doit stocker une liste d'objets par pixel ? Nous

allons utiliser la cohérence de ligne de la méthode d'Atherton pour éviter ce problème. Appelons segment-objet la partie de la ligne de balayage sur laquelle se projette un objet (figure 3.7), et un empan-objet une partie homogène d'un segment-objet (figure 3.8). Tous les pixels appartenant à un même empan-objet possèdent donc la même liste d'objets.

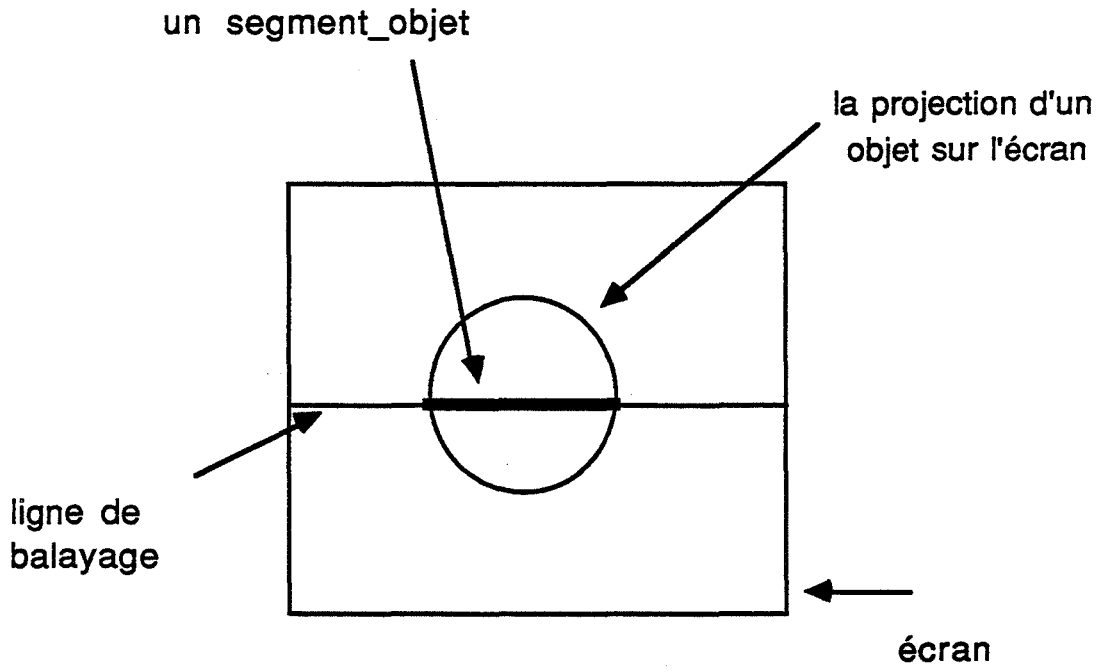


FIGURE 3.7

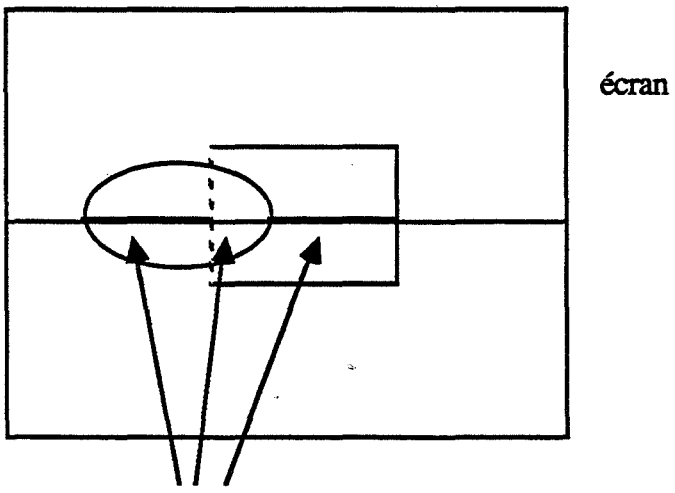


FIGURE 3.8

L'algorithme d'Atherton fournit la liste des objets vus par le premier pixel de l'empan_objet ; pour les autres, il positionne un drapeau qui indique que ce pixel possède la même liste d'objets que son voisin de gauche. Ceci permettra au tracé de rayons de n'allouer qu'une place mémoire pour cette liste.

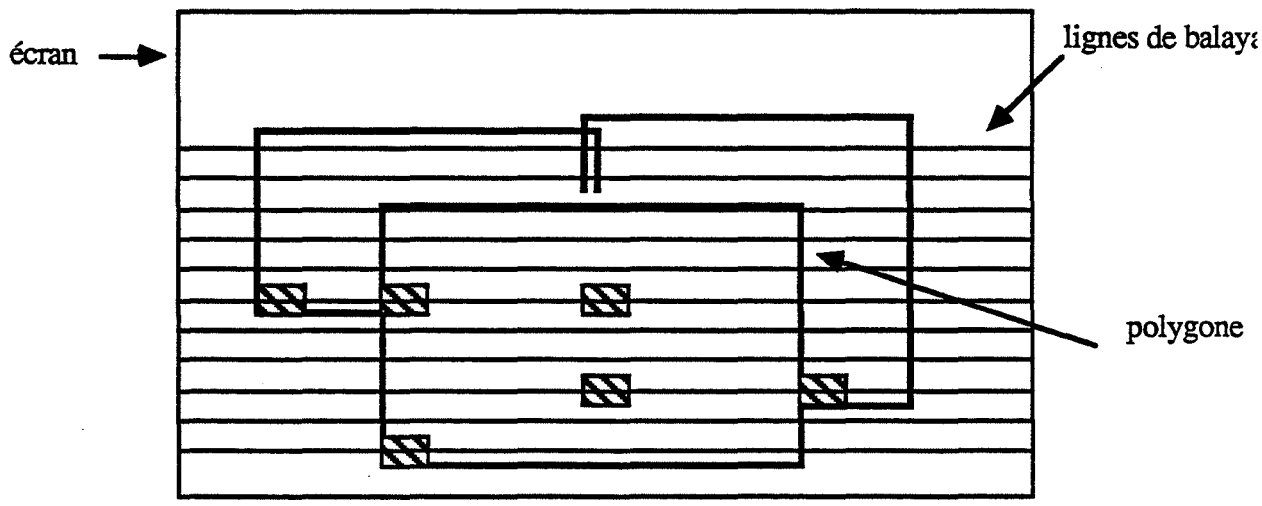


FIGURE 3.9

En plus de la cohérence sur une ligne de balayage, nous avons utilisé la cohérence entre deux lignes de balayage consécutives. En effet, entre deux lignes de balayage, nous gardons un certain nombre d'informations qui permettront de savoir pour chaque pixel de la ligne courante, s'il possède la même liste d'objets qu'un pixel de la ligne précédente. Si c'est le cas, l'algorithme d'Atherton permet d'indiquer pour ce pixel qu'il possède la même liste d'objets que tel autre sur la ligne qui précéde. Ceci permettra au tracé de rayons de n'allouer qu'une place mémoire pour la liste des objets vus par ces deux pixels.

Ainsi pour un objet, on n'alloue qu'une place mémoire pour décrire son emplacement : c'est le pixel le plus à gauche de la ligne la plus basse interceptant cet objet.
La figure 3.9 montre quels sont les pixels (ceux hachurés) pour lesquels l'algorithme d'Atherton doit fournir une liste d'objets. Pour tous les autres, il suffit d'un code signifiant que ce pixel possède la même liste que son voisin ou qu'un autre situé au dessous de lui.

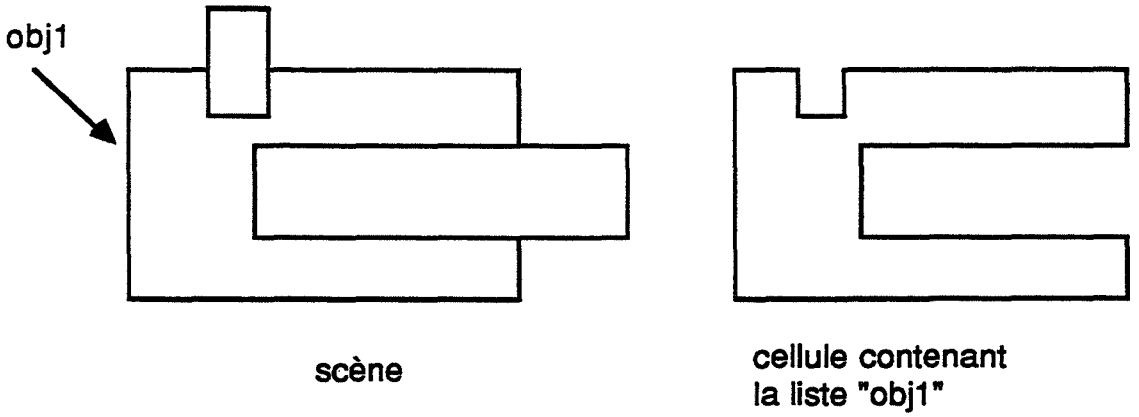


FIGURE 3.10

En général, dans les méthodes d'optimisation par projection sur l'écran, chaque cellule englobe plusieurs pixels. Chaque cellule est un carré obtenu par une subdivision régulière ou récursive de l'écran. Il n'y a aucune raison pour que les objets qui se projettent sur un pixel d'une cellule donnée soient les mêmes pour tous les autres pixels de cette cellule. C'est ce qui affaiblit cette optimisation.

Dans l'optimisation offerte par notre méthode, nous choisissons les cellules de façon à ce qu'elles s'adaptent au mieux à la géométrie de la scène. La figure 3.10 montre une scène et une des cellules correspondantes. Nous économisons ainsi des listes d'objets à stocker et par conséquent de la mémoire.

Excoffier et Tosan [Exco 87] démontrent que la subdivision optimale de l'écran est la partition de l'ensemble des pixels en cellules telle que les mêmes objets se projettent sur tous les pixels d'une cellule donnée. Ils démontrent que cette partition, tout en possédant un nombre minimum d'éléments, permet de minimiser le nombre de calculs d'intersection effectués.

Notre optimisation, de par la manière dont elle stocke les listes d'objets, s'approche beaucoup de cette partition optimale. Elle ne l'atteint pas parce que nous n'utilisons la cohérence qu'entre deux lignes et deux colonnes consécutives seulement. Par exemple, bien qu'ils possèdent la même liste d'objets, les pixels 1 et 2 sur la figure 3.11 auront besoin d'une place mémoire chacun. Il en est de même pour les pixels 3 et 4.

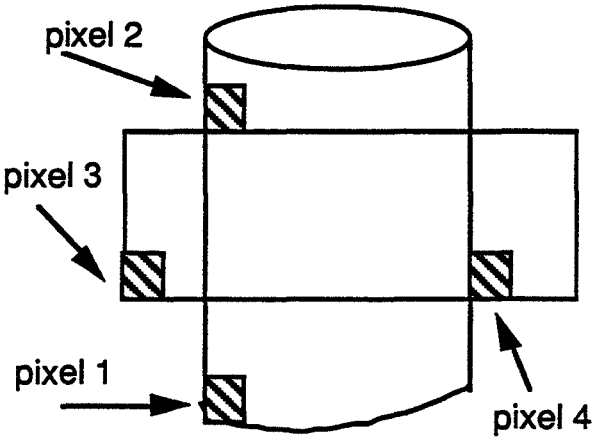


FIGURE 3.11

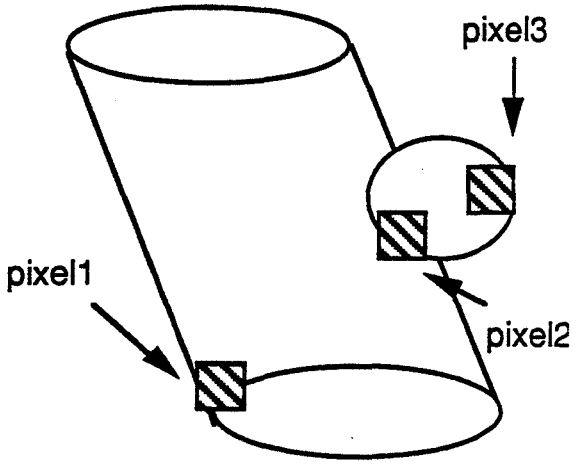


FIGURE 3.12

D'autre part, bien que nous utilisions la cohérence, la place mémoire nécessaire peut devenir très importante s'il y a beaucoup d'objets dans la scène. C'est pourquoi nous choisissons un seuil tel que si la projection d'un objet sur l'écran occupe moins de pixels que ce seuil, on ne crée pas de nouvelle liste pour cet objet baptisé petit. Considérons la figure 3.12. Pour les objets de cette figure, il faudrait allouer de la place mémoire pour trois listes d'objets :
l'une, contenant le cylindre pour pixel1, une autre contenant le cylindre et la sphère pour pixel2, et une troisième contenant la sphère seule pour pixel3. Mais comme la sphère est un petit objet, on ne créera qu'une seule liste contenant le cylindre et la sphère pour tous les pixels concernés.

Le choix de ce seuil est assez difficile. Il dépend de la taille de la mémoire centrale, de celle de l'image, du nombre des objets dans la scène et de la dispersion des projections des objets sur l'écran. Plus ces projections interfèrent, plus la place mémoire nécessaire augmente. C'est ce critère qui est le plus difficile à capter d'une façon simple et efficace. Pour résoudre ce problème nous avons procédé de la façon suivante: nous calculons le nombre moyen n d'objets atherton vus par pixel. Si N dèsigne le nombre d'objets dans la scène, nous approchons le nombre de pixels en lesquels nous avons besoin de stocker une liste d'objets par le produit n*N. Le seuil est alors n*N/a, où a dépend de la mémoire disponible et de la taille de l'écran. Plus la mémoire disponible est faible et la taille de l'écran est importante, plus la valeur de a doit être petite.

C'est le modeleur qui calcule ce seuil et décide si un objet est petit ou non et qui transmet cette information à la méthode d'Atherton. Pour prendre cette décision, le modeleur calcule le nombre de pixels couverts par le rectangle englobant la projection de la boîte englobante de l'objet sur l'écran. Si ce nombre est plus petit que le seuil, alors l'objet est baptisé petit.

3 MIXAGE DES ALGORITHMES D'ATHERTON ET DU TRACE DE RAYONS AVEC INTERACTION EN OMBRE PORTEE

Dans ce paragraphe, nous exposons une méthode de composition de rendu pour permettre l'interaction en termes d'ombres portée entre les deux bases de données d'une composition entre les algorithmes d'Atherton et de tracé de rayons. Comme la méthode de composition de rendu pour les réflexions et les transparences, cette méthode est basée sur l'échange des informations entre ces deux algorithmes.

La base de données de la scène est toujours scindée en deux : la base TR et la base AT. Contrairement à la méthode de Crow pour les ombres portées, la méthode que nous proposons dans ce paragraphe ne convertit pas la base TR en polygones ; elle fonctionne de la façon suivante:

1- Calculer une image de la base de données AT à partir de la source de lumière en utilisant l'algorithme d'Atherton. Il n'y a pas calcul de la couleur des points vus, mais uniquement de leur profondeur pour obtenir un fichier appelé carte de profondeurs.

2- Calculer l'image de la base de données AT à partir de l'œil à l'aide de l'algorithme d'Atherton en prenant seulement en compte les ombres qui proviennent de la base de données AT en utilisant la carte de profondeur calculée dans l'étape précédente. Il y a stockage dans un fichier des coordonnées du point vu en chaque pixel et d'une information indiquant si le point est à l'ombre ou pas.

3- Calculer l'image de la base TR par l'algorithme du tracé de rayons en utilisant la carte de profondeur de (1) pour savoir si un point vu en un pixel est ombragé par un objet de la base AT ou pas. Puis, lancer un rayon d'ombre à partir de ce point permettent pour savoir s'il est à l'ombre d'un objet de la base TR.

4- Un programme ayant connaissance des deux bases de données TR et AT récupère le fichier obtenu en (2). Pour un pixel donné, ce programme lit les coordonnées du point correspondant dans ce fichier, lance à partir de ce point des rayons d'ombre pour déterminer s'il est ou pas à l'ombre d'un objet appartenant à la base TR, et pour modifier en conséquence la couleur du pixel correspondant dans l'image de la base AT.
Ce programme doit stocker en mémoire les coordonnées des points vus en chaque pixel, les normales en ces points, et d'autres informations. Cela pourrait être pénalisant en place mémoire. Pour cela, pendant la deuxième étape, l'algorithme d'Atherton peut transmettre dans un fichier, non les coordonnées du point vu, mais le numéro de l'objet vu en chaque pixel et une information indiquant si c'est le point d'entrée ou de sortie qu'il faut prendre (à cause des opérations booléennes).

Remarque

Au début du paragraphe 2.1, nous avons supposé que tout objet réfléchissant ou transparent était affecté à la base TR. En général, si on utilise le tracé de rayons, c'est pour pouvoir simuler les effets optiques et augmenter le réalisme de l'image. Nous sommes alors normalement automatiquement amenés à attribuer tout objet réfléchissant ou transparent au tracé de rayons (sauf dans le cas où un tel objet appartient à un sous arbre de construction que nous traitons alors comme nous l'avons expliqué dans le paragraphe 2.1). Cette supposition reste peu contraignante. Cependant, si pour certaines raisons nous décidons d'utiliser, pour un objet, les méthodes de calcul des transparences et des réflexions qui existent dans l'algorithme d'Atherton au lieu du tracé de rayons, des problèmes peuvent surgir. En effet, si un objet transparent par exemple appartient à la base AT, il faut prendre

en compte le fait que nous pouvons voir des objets de la base TR à travers lui. Une solution similaire à celle que nous venons de proposer dans ce paragraphe pour les ombres portées est envisageable :

- l'algorithme d'Atherton stocke dans un fichier la liste des coordonnées des points vus à travers d'un objet transparent jusqu'au premier objet opaque. Soit M le premier point de cette liste appartenant à l'objet transparent, et N le dernier point appartenant à l'objet opaque.

- Ensuite, le tracé de rayons lance à partir du point M un rayon réfracté. Si ce rayon intercepte un objet dans la base TR, et si le point d'intersection est situé entre les points M et N sur le trajet du rayon, il faut alors changer la couleur du pixel dans l'image produite par l'algorithme d'Aterhton.

Il faut noter que cette solution dépend étroitement de la méthode utilisée par l'algorithme d'Atehrton pour calculer les effets de transparence.

4 IMPLEMENTATION ET RESULTATS

Nous décrirons les résultats de quelques tests que nous avons effectués pour vérifier les avantages que nous avons cités dans le paragraphe 2.2. Ces tests comparent la méthode de Crow celle que nous proposons dans ce chapitre. Le tableau de la figure 3.13 résume les résultats obtenus pour les rayons réfléchis et réfractés. Les scènes 1, 2 et 3 ne contiennent que des unions. Dans ces scènes les objets AT sont des cubes et des cylindres et les objets TR des sphères. La positions et la taille des objets de la base AT sont définies de manière aléatoire.
Pour la scène 4, nous utilisons les opérations booléennes intersection et différence.
Pour la scène 5, la base de données AT a été conçue de façon à tirer profit au maximum des avantages cités dans le paragraphe 2.2 : elle contient 200 cylindres longs et inclinés de 45 degrés. Cet exemple montre clairement l'avantage de ne pas utiliser de boîtes englobantes puisque l'on est arrivé à un gain de 75% sur une base de données qui ne compte que 200 objets AT.

			Tracé de rayons	Algorithme d'atherton	méthode de Crow		notre méthode		
	# objets AT	# objets TR	temps CPU en secondes	temps CPU en secondes	# appels à fonction intersection avec un rayon	temps CPU en secondes	# appels à fonction intersection avec un rayon	temps CPU en secondes	pourcentage de gain en faveur de notre méthode
scène 1	50	10	216.2	200	9352	182.2	8582	98	46%
scène 2	150	10	411.4	274	13479	433.6	9419	212.1	51%
scène 3	400	10	1400.2	546	19357	1026.7	10607	390.1	62%
scène 4	170	17	386.8	97.2	77524	234.7	45202	110.3	53%
scène 5	200	10	1520,4	370	243229	1112.8	17039	300.4	73%

FIGURE 3.13

D'après ces tests on remarque que le gain augmente avec le nombre d'objets AT. D'autre part, dans ces exemples, nous n'avons utilisé que des objets (cube, sphère, cylindre) dont les calculs d'intersection avec un rayon ne sont pas très coûteux en temps. S'il existe dans la scène des objets dont les calculs soient coûteux, ce gain en temps se trouve amélioré. C'est le cas des objets déformés, des surfaces gauches,.... Cette amélioration vient du fait que la méthode de composition de rendu pour les réflexions et les transparences que nous proposons fait moins appel à la fonction de calcule de l'intersection d'un rayon avec un objet que la méthode de Crow comme le montre le tableau de la figure 3.13.

De plus, on remarque que le gain de temps CPU entre la méthode de Crow et le tracé de rayons classique (sans composition), qui illustre, en fait, le gain obtenu grâce à la méthode de composition des algorithmes telle qu'elle a été proposée par [Duff 80] et [Port 84], c'est à dire sans interactions entre bases de données, est moins important que le gain de temps CPU obtenu en utilisant notre méthode de composition de rendu à la place de la méthode de Crow.

Le tableau de la figure 3.14 résume les résultats obtenus pour les rayons d'ombre. Dans les scènes utilisées, les positions et les tailles des objets de la base AT dans le repère du monde sont obtenues de façon aléatoire. Le gain obtenu correspond au temps mis par le modeleur pour facettiser les sphères. Si on utilise, à la place de la sphère, une autre primitive (des cylindres par exemple), les deux méthodes, (celle de Crow et la

composition de rendu pour les ombres portées), donnent des résultats identiques du point de vu temps de calcul pour les rayons d'ombre.

	# objets AT	# objets TR	méthode de Crow temps CPU en secondes	notre méthode temps CPU en secondes	tracé de rayons temps CPU en secondes	algorithme d'Atherton temps CPU en secondes
scène 1	50	10	290	157	203.7	330
scène 2	150	10	410	232	519.4	503
scène 3	400	10	910	691.4	2747.8	1050

FIGURE 3.14

Les images 1 et 2 montrent repectivement les bases TR et AT ; l'image 3 montre la scène complète.

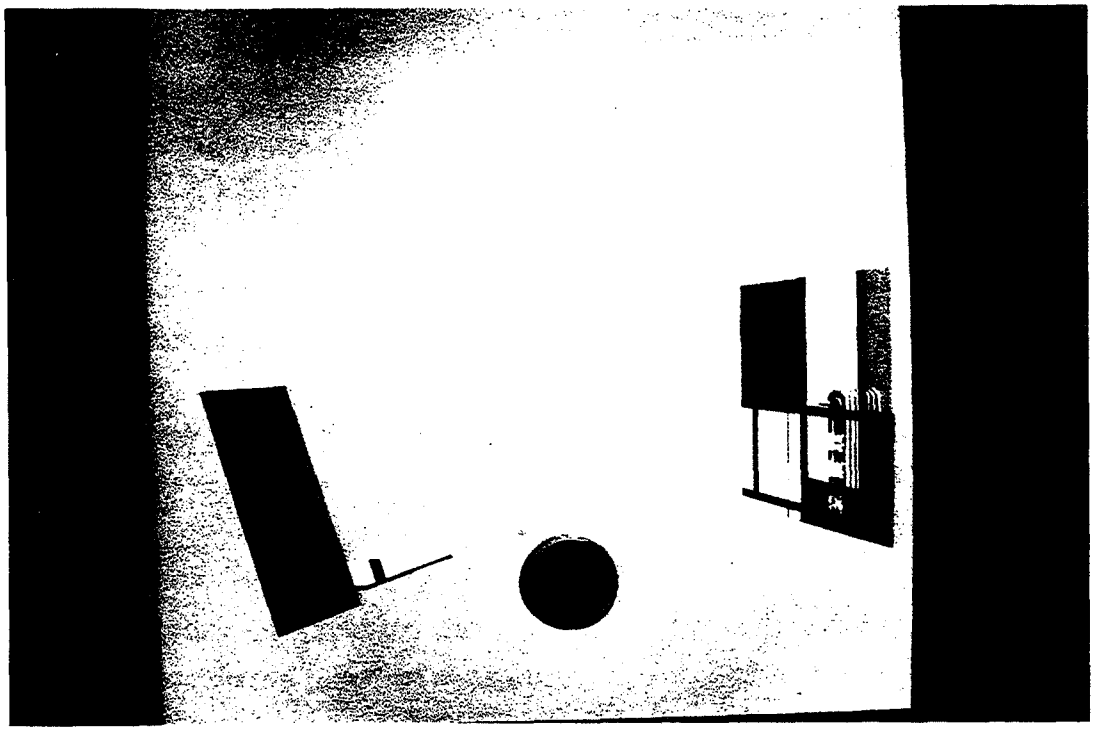


image de la base TR

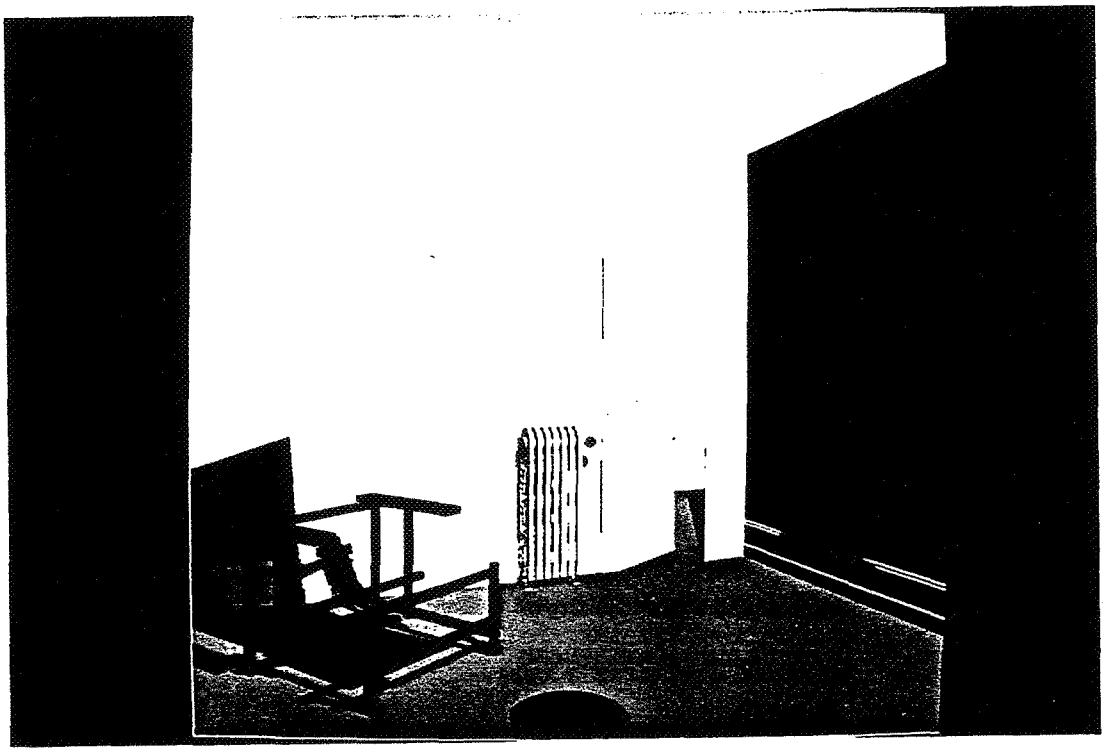


image de la base AT

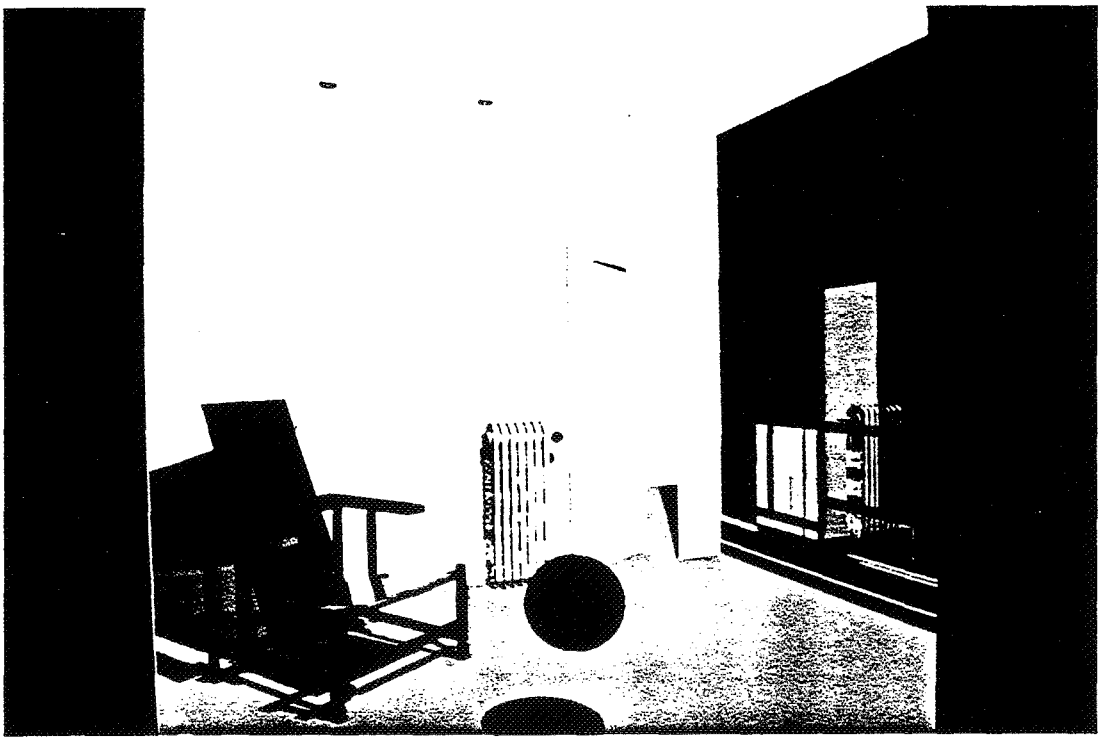


image de la scène complète

5 CONCLUSION ET PERSPECTIVES

La technique de composition est une méthode assez simple et efficace pour permettre le mixage d'algorithmes de calcul des parties cachées dans une même scène. Très tôt, on a proposé une méthode pour remédier aux problèmes d'aliassage qu'elle laisse apparaître. Par contre, le problème des interactions en réflexion, transparence et ombre portée, entre les bases de données - complètement séparées - n'a pas été traité. Dans ce chapitre, nous avons proposé des solutions à ce problème dans le cas particulier d'une composition entre l'algorithme de tracé de rayons et celui d'Atherton. Ces solutions ont l'avantage de permettre la communication d'informations entre les deux algorithmes. Ces communications ont permis d'éviter la facettisation des objets appartenant à la base TR, comme le fait Crow, et de réaliser une optimisation intéressante pour le calcul des rayons réfléchis et réfractés en tracé de rayons.

Un thème de développement possible est la généralisation de nos méthodes de composition de rendu à d'autres algorithmes d'élimination des parties cachées que les algorithmes d'Atherton et de tracé de rayon. Pour une composition entre le tracé de rayons et n'importe quel autre algorithme de visualisation, toutes les solutions que nous avons proposées dans le paragraphe 2 et 3 restent valables. Seule la solution pour économiser de la place mémoire ne peut s'étendre car elle est liée à la cohérence de ligne utilisée par les algorithmes de balayage ligne par ligne.

Enfin, les informations échangées par les algorithmes d'Atherton et du tracé de rayons, durant la composition de rendu pour les réflexions et les transparences, sont distinctes de celles échangées durant la composition de rendu pour les ombres portées. Une voie de recherche consistrait à essayer de tenir compte de ces informations dans l'autre algorithme, pour essayer d'optimiser les algorithmes que nous avons proposés.

Chapitre 4

La gestion des niveaux de détails

1 INTRODUCTION

Si les premiers systèmes de synthèse d'images ne permettaient d'obtenir que des représentations de scènes assez simples, avec un réalisme modeste, on arrive maintenant à fabriquer des images réalistes de haute qualité pour des scènes assez complexes. Mais, le coût de ces dernières , en temps de calcul, est souvent important. Cela constitue un des grands handicaps de la synthèse d'images réalistes. Pour résoudre ce problème, les chercheurs ont essayé de mettre au point des algorithmes de calcul de visibilité et des méthodes de rendu plus rapides, tout en restant aussi performants.

Il existe une autre piste, relativement peu abordée dans la littérature, qui peut permettre de réduire le temps de calcul d'une image réaliste. Cette piste consiste à diminuer la complexité de la scène en gérant les niveaux de détails dans cette scène : les entités qui doivent exister dans l'image sont uniquement celles que l'observateur arrive à distinguer. D'où l'idée d'avoir plusieurs représentations pour un même objet, chaque représentation correspondant à un certain niveau de détail. Par exemple, lorsqu'une maison est vue de très près, l'observateur distingue la profondeur des fenêtres, les sculptures sur la porte, les tuiles sur le toit. Plus il s'éloigne, moins il les distingue, jusqu'à ce que la maison ressemble pour lui à un simple cube. Lorsqu'on veut visualiser un objet, la représentation correspondant au niveau de détail nécessaire devrait être choisie. Supposons que nous voulions produire une séquence d'images dans laquelle l'observateur s'éloigne de plus en plus de la maison. Cette dernière passerait de la représentation la plus détaillée (avec les sculptures sur la porte), à la représentation d'un simple cube. De cette façon, on évite de garder la même description très détaillée, donc coûteuse en temps de calcul, pour toutes les images à produire.

De plus, dans un système de synthèse d'images flexible, qui permet le mixage d'algorithmes de visualisation et de méthodes de rendu dans une même scène, chacune des représentations est affectée à l'algorithme qui lui est le mieux adapté. Par exemple, si les sculptures sur la porte sont modélisées à l'aide de petites sphères, la porte sera affectée par le système à l'algorithme du tracé de rayons lorsque l'observateur est proche de la maison. Lorsqu'il s'éloigne jusqu'à ne plus arriver à distinguer ces sculptures, on utilise une nouvelle description de la porte qui ne contient plus les sphères. La porte, un simple parallélipipède, sera alors affectée par le système à un algorithme plus rapide travaillant sur des facettes. De cette façon, les descriptions utilisées pour visualiser la scène sont celles juste nécessaires pour convaincre l'observateur du réalisme de l'image, et les méthodes de

calcul de visibilité et de rendu utilisées pour une représentation donnée d'un objet sont celles dont le degré de performance est jugé nécessaire par le système.

Nous avons intégré dans le système de synthèse d'images ILLUMINES plusieurs outils et méthodes permettant la gestion des niveaux de détails. Dans le paragraphe 2, nous citerons les principaux travaux déjà réalisés pour la gestion des niveaux de détails. Dans le paragraphe 3, nous exposerons les techniques que notre modeleur fournit à l'utilisateur pour générer les différentes descriptions d'un même objet. Dans le paragraphe 4, nous décrirons une méthode pour remplacer un ensemble d'objets de petites dimensions sur l'écran par un seul objet plus simple associé à une texture. Dans le paragraphe 5, nous exposerons un outil qui permet de changer l'aspect associé à un objet donné. Enfin, nous conclurons dans le paragraphe 6.

2 RAPPEL DES SOLUTIONS DEJA PROPOSEES

Nous pensons que la gestion des niveaux de détail constitue un problème important en synthèse d'images. Son influence intervient aussi bien dans la modélisation que dans la visualisation.

Il existe une analogie entre la gestion des niveaux de détail et les *extracteurs* dans les applications qui possèdent d'importantes bases de données, comme les simulateurs de vol par exemple. Le rôle de ces programmes est d'extraire, depuis la base de données, la partie de la scène nécessaire à la visualisation lorsque l'observateur est dans une région donnée. Ils effectuent donc une gestion primitive des niveaux de détails : un objet existe ou n'existe pas dans la scène.

Clark [Clar 76] a été un des premiers chercheurs à avoir évoqué le problème des niveaux de détails. Il a suggéré de représenter un objet à l'aide d'un arbre : la racine est une représentation très grossière de cet objet. L'objet est divisé en parties de plus en plus détaillées qui correspondent aux branches de l'arbre. Ainsi, chaque noeud contient une partie de l'objet. Un noeud donné dans l'arbre possède une description plus complexe que celle de son nœud-père. En continuant avec l'exemple de la maison, cette dernière sera représentée par un arbre dont la racine est un cube qui est la description grossière de la maison. Les nœuds-fils de la racine constituent la seconde description, plus détaillée. Ils contiennent des primitives modélisant la porte, les fenêtres, le toit ... Chacun de ces nœuds peut avoir des fils qui constituent une version plus détaillée de ce nœud. Par exemple, les fils du nœud qui

décrit la porte introduisent les sculptures sur cette porte. Un algorithme de calcul de visibilité doit être capable de traverser l'arbre et d'afficher les parties ayant le bon niveau de détails. Des tests sont effectués pour décider si le nœud courant doit être affiché ou bien si en doit passer aux nœuds-fils. Cette organisation permet aux différentes parties d'un même objet d'être calculées avec différents degrés de complexité.

Pour représenter un objet, on peut décrire le volume occupé par ce dernier en utilisant les arbres octaux [Same 84]. Un tel arbre peut être considéré comme une hiérarchie de descriptions. La figure 4.1 montre un arbre octal décrivant un objet. Les figures (4.2 a) et (4.2 b) montrent deux niveaux de détails de cet objet (les nœuds indéterminés sont rendus pleins). Un algorithme d'élimination des parties cachées, travaillant sur un objet décrit par un arbre octal, doit pouvoir traverser l'arbre et s'arrêter au niveau de résolution qu'il estime suffisant. Pour cela, il effectue un test en chaque nœud indéterminé pour décider s'il le remplace ou non par un nœud plein.

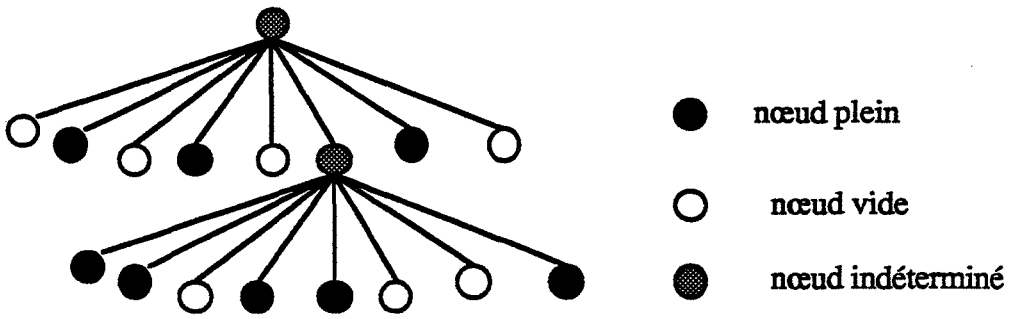


FIGURE 4.1

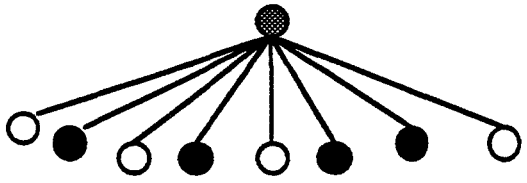


FIGURE 4.2 a



FIGURE 4.2 b

Rubin et Whitted [Rubi 80] ont développé une représentation hiérarchique basée sur une partition spatiale constituée de parallélépipèdes pour rendre plus rapide l'algorithme du tracé de rayons. Les seuls objets utilisés dans la modélisation sont des parallélépipèdes. Cette utilisation d'une hiérarchie de volumes englobants, pour réduire la dépendance de

l'algorithme du tracé de rayons vis à vis de la complexité de l'environnement, a fait l'objet de plusieurs publications [Wegh 84], [Fuji 86], [Arge 88a].

[Rubi 82] reprend la même hiérarchie développée par [Rubi 80] et l'utilise pour la gestion des niveaux de détails : comme le fait Clark [Clar 76], Rubin effectue un test pour chaque nœud de la hiérarchie pour décider s'il affiche les polygones définissant le parallélépipède attaché à ce nœud, ou bien s'il doit passer aux nœuds-fils de ce dernier. Par contre, lorsqu'il juge qu'un parallélépipède est trop petit, il ne le remplace pas tout de suite par son parallélépipède-père comme le fait Clark, mais il réalise un passage en douceur entre ces deux parallélépipèdes en mélangeant leurs couleurs pendant un certain temps.
Le test qu'il utilise pour décider s'il affiche un parallélépipède ou bien s'il passe à ses parallélépipèdes-fils consiste à calculer sa taille sur l'écran : si cette taille est plus petite que le tiers d'un pixel, ce parallélépipède est jugé trop petit. Ce critère - un tiers de pixel - se déduit du critère de Rayleigh : l'angle minimum entre deux objets pour que l'œil puisse les distinguer doit être:

$$1/\sin(1.22\mu/\partial)$$

où ∂ est le diamètre de la pupille et μ la longueur d'onde [Scha 48]. En utilisant ce critère, il a été décidé qu'un écran télévision a besoin de 1500 à 1800 ligne de balayage [Scha 48], [Lele 80]. Ainsi, pour un écran de 512*512, l'œil ne peut pas distinguer des objets plus petits qu'un tiers de pixel.

Williams [Will 83] a proposé une méthode pour traiter l'aliassage qui apparaît lorsqu'on met en perspective une texture (phénomène de moirés). Cette méthode effectue un pré-traitement qui consiste à extraire de la texture initiale de dimension 2^n*2^n, une suite d'images T_p de dimensions 2^p*2^p, p variant de n à 0. L'intensité de chaque pixel de T_{p-1} est la moyenne simple de 2*2=4 pixels correspondants de T_p. Une telle structure est appelée *mip map* ; le mot mip est l'abrégé d'un mot latin signifiant beaucoup de choses dans une petite place. Une image T_p correspond à des taux de compression de 2^{n-p} égaux dans les deux directions. Cette structure peut être considérée comme une structure pyramidale où la base de la pyramide est l'image de dimension 2^n*2^n, et les coordonnées verticales le taux de compression D. Pour obtenir l'intensité d'un pixel (i,j) de l'image finale, on détermine p tel que : $2^{n-p-1} \leq D \leq 2^{n-p}$, où D est le taux de compression présent en ce pixel ; on calcule les intensités correspondantes sur T_{p-1} et T_p notées I_{p-1} et I_p, à l'aide de huit lectures de pixels et deux approximations bilinéaires. La valeur de l'intensité I(i,j) du pixel (i,j) est obtenue par une approximation linéaire entre Ip-1 et Ip.

```
{
 pour chaque pixel (i,j) de l'image finale faire {
   u = f(i,j) ;          /* f est la transformation inverse entre l'espace de l'image */
   v = g(i,j) ;                    /*   et l'espace de la texture */
   Calculer le taux de compression D.
   déterminer p tel que 2^(n-p-1) ≤ D ≤ 2^(n-p) ;
   calculer I_(p-1)(u,v) par 4 lectures et 1 approximation bilinéaire sur T_(p-1) ;
   calculer I_p(u,v) par 4 lecture et 1 approximation bilinéaire sur T_p ;
   calculer I(i,j) par 1 approximation linéaire entre I_(p-1)(u,v) et I_p(u,v) ;
   }
}
```

L'ensemble des images $(T_p)_p$ constitue un intervalle *continu* de textures à différents niveaux de détails. Cet intervalle s'étend de la texture initiale la plus détaillée, à une couleur unique qui représente la moyenne des couleurs de cette dernière.

MacDouglas [Macd 84] a traité le problème, déjà évoqué par Rubin [Rubi 82], du passage en douceur entre deux descriptions différentes d'un même objet pendant une animation. En effet, une transition simple entre deux descriptions pourrait être visible par l'observateur. Supposons que, pendant une animation, un premier objet *obj1* cache un second objet *obj2* qui possède plusieurs descriptions. Une solution pourrait consister à effectuer le passage entre les descriptions de *obj2* lorsque *obj1* se situe devant lui. L'inconvénient est que cette méthode n'est applicable que si l'objet, qui doit changer de descritpion, est caché par un autre objet à un certain moment durant l'animation.

MacDouglas range les différentes représentations d'un objet dans un ordre de complexité décroissante. Pour réaliser la transition entre deux descriptions, il les mélange dans la même image pour pouvoir obtenir des transitions plus douces. Pour réaliser ce mélange, il a proposé deux méthodes : la première utilise la proportion du pixel qu'occupe chaque représentation d'un objet ; la deuxième utilise la transparence progressive de la description qui doit disparaître, et l'opacité progressive de celle qui doit la remplacer.

Ainsi, il calcule, pour chaque description, une taille critique sur l'écran ; pour une position donnée de l'œil, si la taille de la projection d'une description donnée sur l'écran est plus petite que sa taille critique, cette description doit disparaître et doit être remplacée par une autre moins précise. Il spécifie une zone, dite zone de transition, en fonction de la distance de l'objet à l'œil, pendant laquelle il mélange les deux descriptions.

Notons que pendant la transition, les deux descriptions sont présentes en même temps dans la scène à visualiser, ce qui implique un travail supplémentaire. L'inconvénient est que, pendant une animation, l'objet peut rester un long moment dans la zone de transition, comme, par exemple, une animation où l'observateur suit un objet : en effet, il se peut que la distance entre eux reste pendant une durée importante une distance critique qui nécessite le calcul des images pour les deux descriptions de l'objet. Ceci augmentera de beaucoup le temps de calcul.

Pour calculer la taille critique de l'objet sur l'écran, on peut utiliser la méthode de la force brutale qui consiste à projeter une description donnée de l'objet à différentes distances de l'oeil pour choisir une taille critique qui semble être acceptable pour cette description. MacDouglas propose une méthode qui consiste à prendre comme taille critique d'une description donnée la taille sur l'écran telle que le plus grand côté des polygones décrivant cette description ait une longueur plus petite qu'un pixel (dans le cas d'un objet à représentation polygonale) .

Enfin, il a exposé d'autres méthodes pour assurer la génération automatique des différentes représentations d'un objet décrit par des polygones. Ces méthodes partent de la description polygonale la plus détaillée et génèrent d'autres descriptions polygonales moins détaillées. Pour cela, il trie les arêtes suivant l'ordre croissant de leurs longueurs. Ensuite, Il commence par supprimer la plus petite arête en remplaçant les deux points qui la définissent par leur milieu. Il répète la même opération pour les arêtes dont les deux extrémités n'ont pas été modifiées. Chaque fois qu'il supprime une arête, il retrie la liste de nouveau.

Plemenos [Plem 87] reprend la technique utilisée par Clark [Clar 76] pour son modèle hiérarchique à attributs. Il a recours à l'intelligence artificielle pour réaliser l'assemblage des nœuds dans sa hiérarchie, et présente un algorithme de visualisation réaliste permettant de tirer profit de la structuration de la scène proposée pour accélérer le processus de visualisation.

Enfin, pour effectuer le choix entre plusieurs représentations d'un objet, Blake [Blak 87] calcule la taille de la boîte qui englobe chaque description de cet objet avec tous les mouvements possibles de ses différentes parties. Il l'affiche si cette taille est plus petite que sa distance à l'œil multipliée par l'angle sous lequel est vue cette description. Il justifie ses décisions par des calculs qu'il effectue dans le domaine fréquentiel.

3 GENERATION DES DIFFERENTES DESCRIPTIONS D'UN OBJET

Un des problèmes essentiels qui surgit lorsqu'on veut assurer la gestion des niveaux de détails dans un système de synthèse d'images, est que l'effort demandé à l'utilisateur pour fabriquer les différentes descriptions d'un même objet devient assez important, même pour une scène modeste. L'idéal serait de pouvoir générer de façon automatique les différentes représentations d'un objet : en conservant l'exemple de la maison, l'utilisateur fournirait la description la plus détaillée, puis, le système générerait seul les autres descriptions. Ce problème - l'automatisation de la génération des représentations d'un objet - est un problème très difficile qui n'a pas encore été résolu. Des travaux sont en cours à l'université de Metz où l'on propose de faire, dans un premier temps, une étude évaluative de la scène. Cette étude a pour but de découvrir les objets prédominants, soit par leur volume, soit par leur texture plus claire que d'autres, ou par leur emplacement isolé ou privilégié par rapport aux autres objets. Les éléments de la scène sont classés suivant un ordre croissant d'importance. Dans un deuxième temps, cet ordre est exploité pour filtrer les objets en supprimant certains d'entre eux ou bien en les remplaçant par d'autres objets. On pense d'utiliser l'intelligence artificielle pour intégrer des règles de décision simples en fonction de l'environnement d'une scène.

Dans notre modeleur, nous proposons à l'utilisateur deux possibilités, une première manuelle et une seconde automatique, pour qu'il puisse obtenir les différentes représentations d'un objet.

3.1 Génération manuelle des différentes descriptions d'un objet

Nous avons introduit une fonction *Version* qui permet à l'utilisateur de donner plusieurs descriptions, qu'il a lui-même définies, d'un même objet.

```
obj = Version(obj_def_precise, 1.,
              obj_def_moyenne, 5.,
              obj_def_grossière, 20.,
              FIN) ;
```

Dans cet exemple, l'objet *obj* possède trois représentations *obj_def_précise*, *obj_def_moyenne* et *obj_def_grossière*, modélisées par l'utilisateur pour décrire le même objet. Les nombres 1, 5 et 20 sont les tailles critiques respectives de l'objet sur l'écran qui indiquent le moment adapté pour faire un changement de description. Le choix de la taille critique d'une description n'est pas évident. Dans notre système, nous avons autorisé l'utilisateur à la fixer lui même. Pour cela nous avons fait appel aux rapport_seuils utilisés dans le chapitre 2. Dans cet exemple, l'utilisateur fixe les rapport_seuils à 1, 5 et 20 ; à partir de ceux-ci, la description correspondante est choisie :
si le rapport de *obj_def_précise* est inférieur ou égal à 1 (c'est à dire que la longueur de l'image est inférieure ou égale à une fois la *longueur* de l'objet), on utilise la définition la plus précise de l'objet ;
si le rapport de *obj_def_moyenne* est inférieur ou égal à 5 et plus grand que 1 (c'est à dire que la longueur de l'image est entre 5 fois et 1 fois la *longueur* de l'objet), on choisit la définition moyenne ;
si le rapport de *obj_def_grossière* est inférieur ou égal à 20 et plus grand que 5, on utilise la définition grossière de l'objet.
Dans le cas où le rapport de *obj_def_grossière* est plus grand que 20, la description grossière est quand même gardée.

Cependant, il est possible que le choix des tailles critiques devienne une tâche difficile qui alourdisse le travail demandé à l'utilisateur. Pour remédier à cet inconvénient, nous avons introduit un autre procédé pour décider du passage entre deux descriptions : le passage d'une description d'un objet à la description consécutive correspond à la suppression d'éléments. Nous souhaitons que la transition entre ces deux descriptions ait lieu lorsque l'observateur n'arrive plus à distinguer ces éléments, donc lorsqu'il n'arrive plus à voir les objets qui les modélisent. Lorsque la projection du plus grand de ces objets est inférieure à un pixel, nous sommes sûrs que nous pouvons effectuer cette transition. Par exemple, lorsque la projection sur l'écran des plus grand des objets modélisant les sculptures sur la porte est plus petite qu'un pixel, nous pouvons utiliser une description moins précise de cette porte qui ne contient plus ces sculptures.

L'utilisateur désigne un certain nombre d'objets, que nous appellerons objets de référence : lorsque leur projection a une taille inférieure à celle d'un pixel, ils indiquent que la transition entre les descriptions peut être effectuée.

obj = Version(obj_def_précise, obj_référence1, obj_référence2, FIN,
obj_def_moyenne, obj_référence3, FIN,

obj_def_grossière, obj_référence4, FIN,
FIN) ;

Obj_référence1 et *obj_référence2* sont deux objets de référence pour la description de *obj_def_précise ; obj_référence3* et *obj_référence4* sont ceux des descriptions de *obj_def_moyenne* et *obj_def_grossière* respectivement.

La fonction *Version* teste si chacune des projections de *obj_référence1* et de *obj_référence2* sur l'écran est plus petite qu'un pixel. Si ce n'est pas le cas, elle retourne *obj_def_précise* qui sera la description choisie. Sinon, elle effectue le même test pour *obj_référence3* ; elle retourne obj_def_moyenne si ce test s'avère négatif, sinon elle passe à la description suivante.

Remarquons que si la projection de *obj_référence1* et/ou celle de *obj_référence2* sur l'écran ne sont pas plus petites qu'un pixel, nous sommes sûrs que c'est encore le cas pour obj_référence3, puisque *obj_def_précise* est une description plus détaillée que *obj_def_moyenne*. Pour cette raison, si l'utilisateur échange les places de *obj_def_précise* et *obj_def_moyenne* en tant que paramètres de la fonction *Version*, des erreurs peuvent surgir : en effet, supposons que la position de l'œil soit telle que c'est la description la plus détaillée (*obj_def_précise*) qui doit être utilisée. Par conséquent, parmi tous les objets de référence de toutes les descritpions, aucun ne possède une projection plus petite qu'un pixel sur l'écran. La fonction *Version* commencera par tester la taille sur l'écran de *obj_référence3*. Lorsqu'elle trouvera que cette taille est plus grande que celle d'un pixel, elle retournera *obj_def_moyenne* comme description choisie de l'objet *obj*, sans prendre le temps de tester obj_référence1 ou *obj_référence2*.

Pour éviter ce problème, la fonction *Version* commence par calculer les tailles sur l'écran des différents objets de référence correspondants aux différentes descriptions. Ensuite, elle trie les descriptions de telle manière que les tailles des projections sur l'écran de leurs objets de référence soient dans un ordre croissant.

Quant au problème du passage en douceur entre deux descriptions d'un même objet, nous avons implémenté la méthode proposée par MacDouglas qui affecte une transparence progressive à la description qui doit disparaître et une opacité progressive à celle qui doit la remplacer (voir §5.3). Par ailleurs, un passage simple entre deux descriptions est cependant autorisé dans notre système. MacDouglas envisage un certain

nombre de cas où un tel passage est possible sans qu'il soit visible par l'observateur. Parmi ces cas on peut citer :
- l'utilisation de descriptions peu différentes l'une de l'autre, de façon à ce que ce passage ne nuise pas au réalisme de l'image ;
- la réalisation de ces passages lorsque l'objet est assez loin de l'observateur.

Pour décider du moment où la transition doit être réalisée entre deux descriptions d'un objet, l'utilisation des rapports-seuils (donc des boîtes englobantes dont les faces sont parallèles au plan du repère du monde) risque de produire des transitions brutales qui nuisent au réalisme souhaité si l'utilisateur choisit mal ses rapports-seuils, ou bien s'il ne se conforme pas aux cas cités plus haut où un tel passage entre descriptions est tolérable. Les objets de référence que nous avons introduits dans ce paragraphe, nous permettent d'effectuer des passages simples et doux, même si aucun des deux cas cités plus haut ne sont pas vérifiés. Il est donc souhaitable, pour ce type de transition, de les utiliser, surtout qu'il est beaucoup plus naturel pour l'utilisateur de repérer et de choisir des objets de référence que de choisir un rapport-seuil.

Dans le cas de l'utilisation des rapports-seuils, la fonction *Version* procède de la façon suivante : d'abord, elle trie les différentes descriptions dans l'ordre croissant de leurs rapports-seuils. Elle calcule le rapport de la première description dans la liste et le compare avec son rapport-seuil. S'il est supérieur à ce dernier (ce n'est pas la représentation qu'il faut choisir), elle passe à la description suivante et ainsi de suite ; sinon (c'est la représentation qu'il faut choisir), il existe trois cas qui servent à assurer le passage en douceur entre les descriptions :

- c'est la même description *rep1* qui a été utilisée pour le calcul de l'image précédente dans la séquence d'images à produire (on utilise un drapeau pour le savoir). Dans ce cas, cette description est retournée par la fonction *Version*.

- une autre description *rep2* a été utilisée pour le calcul de l'image précédente ; la fonction *Version* retourne alors les deux descriptions *rep1* et *rep2*. Cela permettra à l'algorithme de visualisation d'effectuer le passage en douceur entre ces deux représentations de l'objet ; pour cela, la fonction *Version* les baptise *objet-fondant* (cf § 5). D'autre part, elle calcule un intervalle dit intervalle de transition qui sera l'intervalle où le passage en douceur entre les deux descriptions aura lieu. Cet intervalle est calculé de la façon suivante : Supposons, pour fixer les idées, que l'observateur s'éloigne de l'objet. Pour calculer l'intervalle de transition associé à une description, on calcule la distance minimale D de l'œil à la boîte englobante de cette dernière. L'intervalle

de transition sera [D, D+ A] ; A est ou bien une constante positive possédant une valeur par défaut et qui peut être fixée par l'utilisateur ; ou bien c'est la fonction *Version* qui calcule la valeur de A en fonction des prochaines positions de l'œil, si on ne travaille pas en temps réel. Dans le cas où l'observateur s'approche de l'objet, l'intervalle de transition sera [D-A, D].

- la description *rep1* a été utilisée avec une autre ancienne description *rep2* pour le calcul de l'image précédente. Il s'agit maintenant de savoir quand est ce qu'il faut remplacer ces deux descriptions par la seule nouvelle description *rep1*. Les distances minimales *dmin* et maximales *dmax* de l'œil à la description *rep1* sont alors calculées. Si l'intervalle [dmin, dmax] est disjoint de l'intervalle de transition associé à l'ancienne description *rep2* (calculé dans cas précédant), seule la nouvelle description est retournée, sinon la fonction *Version* retourne encore une fois les deux descriptions.

Une fois que les différentes descriptions d'un objet et les rapports-seuils (ou objets de référence) correspondant ont été définis, la fonction *Version* est un outil simple qui permet à l'utilisateur de visualiser un objet avec le bon niveau de détails. Soient B et C deux objets possédant chacun deux descriptions :

B = Version (B1 - B2, nb1,
B1, nb2) ;

C = Version (C1 **U** C2, nb3,
C1, nb4) ;

B1, B2, C1, C2 sont des primitives ou bien des sous-arbres de construction ; nb1, nb2, nb3, nb4 sont des tailles critiques (exprimées en rapport-seuil).
Soit A l'objet construit à partir de l'union des objets B et C : A = B **U** C.
L'objet A possède une description seulement, qui est la réunion des objets B et C. ces derniers gérent eux même automatiquement leurs différentes descriptions. En conséquence, selon la position de l'observateur, l'objet A peut avoir une des expressions suivantes :

A = (B1 - B2) **U** (C1 **U** C2) ;
A = (B1 - B2) **U** C1 ;
A = B1 **U** (C1**U**C2) ;
A = B1 **U** C1 ;

Chaque objet gère donc ses propres descriptions et la définition des versions des différents objets est par suite simple et modulaire : au lieu de définir quatre descriptions pour l'objet A, nous avons défini deux descriptions pour chacun des objets B et C. Si A est un objet complexe, chaque partie de cet objet peut ainsi avoir plusieurs descriptions.

2.2 Génération automatique des différentes descriptions d'un objet

La fonction *Version* est un outil manuel offert à l'utilisateur pour gérer les niveaux de détails, qui lui demande de modéliser plusieurs descriptions du même objet. Cette fonction concerne les objets composites (groupement d'objets). Une première description peut contenir plus d'objets qu'une seconde (cf figure 4.3).

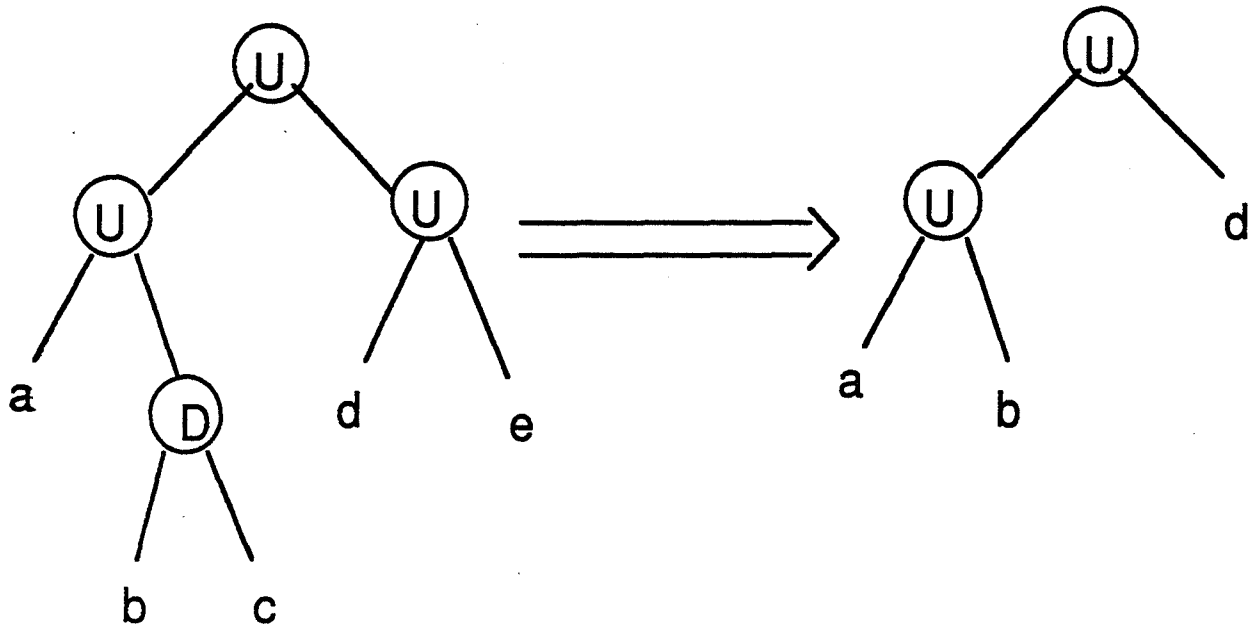


FIGURE 4.3

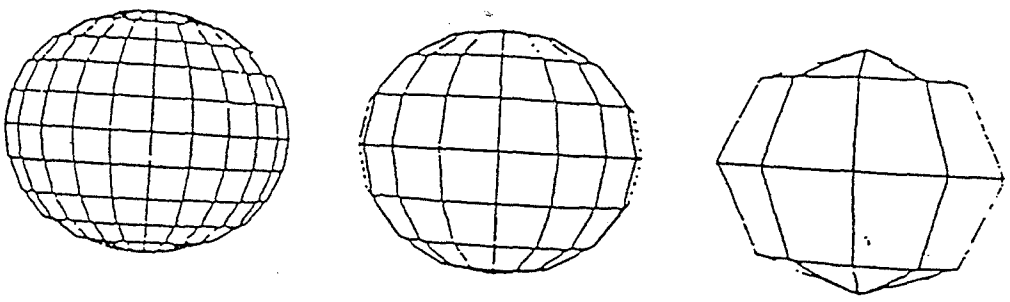


FIGURE 4.4

Pour un objet simple (une primitive de modélisation) facettisé, une génération automatique des différentes représentations est rendue possible dans notre modeleur. Deux représentations d'un tel objet diffèrent par le nombre de polygones qu'elles contiennent. Une description détaillée contient plus de facettes qu'une autre moins détaillée (cf figure 4.4). En effet, supposons qu'un objet soit modélisé par une sphère et qu'on veuille facettiser cette primitive pour pouvoir la visualiser plus tard. Plus cet objet est près de l'observateur, plus le degré de facettisation doit être élevé. Le degré de facettisation d'une primitive de modélisation doit dépendre de la taille de sa projection sur l'écran. Moins la taille de cette projection est importante sur l'écran (i.e. plus sa distance à l'oeil est grande), moins on a besoin de la facettiser. Une sur-facettisation d'une primitive entraîne un surcoût de calcul inutile. Chaque degré de facettisation utilisé pour cette sphère correspond à une des descriptions de cet objet servant à la gestion des niveaux de détails.

La génération de ces descriptions (i.e. le choix du degré de facettisation) se fait d'une façon automatique. En effet, nous avons introduit des fonctions qui facettisent les primitives de modélisation en prenant en compte leur taille sur l'écran. La génération automatique de deux représentations consécutives d'un objet fait varier assez peu le degré de facettisation de cet objet, de façon à ce que ces deux descriptions ne diffèrent que légèrement, et qu'un passage entre elles ne risque pas d'être visible par l'observateur. Cette approche n'est pas nouvelle ; Catmull [Catm 75] l'a utilisée dans son algorithme de subdivision d'un carreau bicubique qui utilise le nombre de pixels couverts par la projection sur l'écran d'un sous-carreau comme test d'arrêt. Bien que cette méthode ne soit pas destinée à gérer les niveaux de détails, elle possède cette propriété. On trouve dans la littérature des méthodes qui traitent de la facettisation adaptative des surfaces. En plus du critère de la taille de la surface sur l'écran qui sert à la gestion des niveaux de détails, elles utilisent d'autres critères plus sophistiqués comme test d'arrêt. Lane [Lane 80] a proposé une variante de l'algorithme de subdivision de Catmull dans laquelle il utilise la courbure du sous-carreau comme test d'arrêt. Il obtient beaucoup moins de polygones qu'avec l'algorithme de Catmull pour des carreaux peu courbés. [Herz 87] et [Parr 87] triangulent d'une façon adaptative les surfaces pour leur appliquer des déformations .

Pour notre problème de niveaux de détails, nous avons besoin de méthodes simples, rapides et efficaces pour réaliser la facettisation automatique des primitives de modélisation. Nous n'avons pas forcément besoin de les trianguler. Nous avons utilisé une fonction de facettisation différente par primitive de modélisation ; c'est l'approche utilisé dans le système Reyes [Cook 87]. Pour chaque primitive, nous fixons dès le départ la forme des polygones auxquels nous voulons aboutir (cf figures 4.5 a, b, c), et la méthode de facettisation permet d'obtenir des polygones de cette forme.

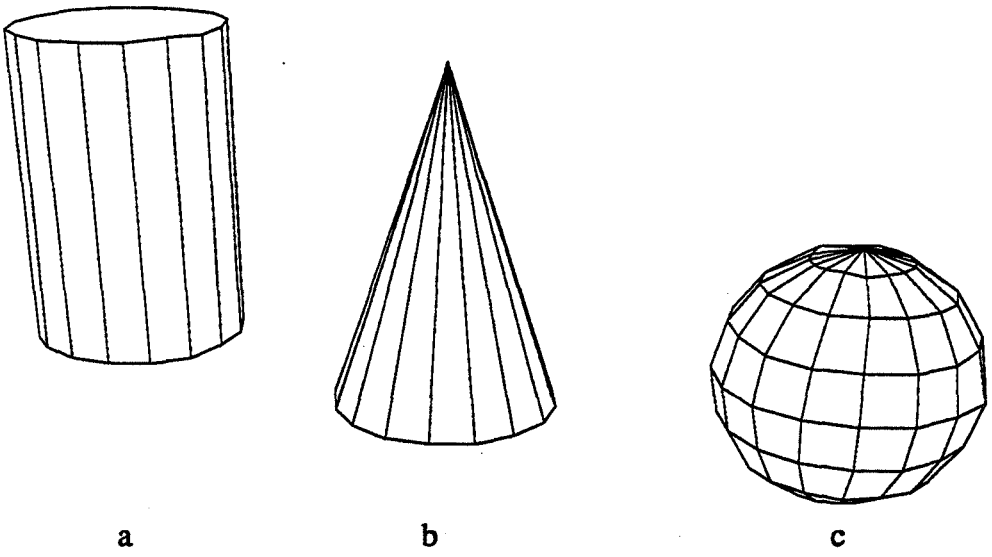


FIGURE 4.5

La facettisation automatique est une solution simplifiée du problème général, beaucoup plus difficile, qui consiste à générer automatiquement, à partir d'une description très détaillée d'un objet composite, toutes les autres descriptions.

4 L'UTILISATION DE LA TEXTURE

Supposons que la scène que nous voulons visualiser contienne un ensemble d'objets proches les uns des autres relativement à leur distance à l'observateur. Plus ce dernier s'éloigne, moins il distingue les éléments de l'ensemble ; à la place, il voit une forme qui ressemble à l'ensemble de ces éléments. Par exemple, si nous regardons de près un toit de tuiles rondes, ce dernier est perçu comme un véritable objet 3D constitué de beaucoup

d'éléments (les tuiles). Par contre, si nous le regardons de loin, il pourra être remplacé par un polygone peint avec une image des tuiles sans que nous nous en apercevions.

L'idée principale que nous proposons est de remplacer un ensemble d'objets par un seul objet que nous appellerons *objet-silhouette*. Ce dernier possède la même silhouette que cet ensemble d'objets, et une couleur appropriée de façon à ce que l'observateur ne remarque aucun changement. Cette couleur sera une texture que nous plaquerons sur *l'objet-silhouette*. Cette texture est une image calculée à partir de la définition 3D de l'ensemble des objets.

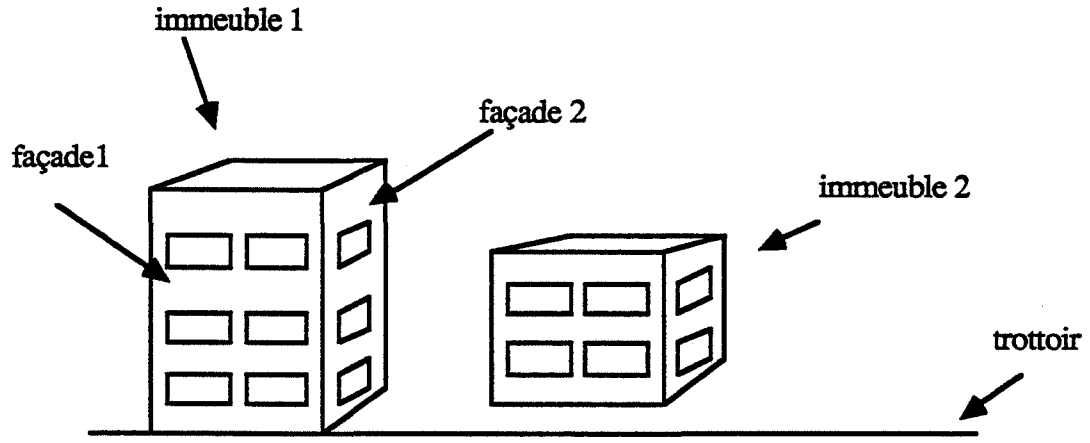


FIGURE 4.6

La figure 4.6 montre deux immeubles, le premier étant plus proche du trottoir que le second ; les rectangles sur les façades sont des balcons. Lorsque l'observateur est près de ces immeubles, leurs descriptions 3D sont utilisées. Supposons qu'il s'éloigne de façon à ce qu'il n'arrive plus à distinguer les profondeurs des balcons sur façade 1. cette dernière est alors remplacée par un *objet-silhouette* (un polygone par exemple) possèdant la textures de cette façade. Il en est de même pour les autres façades. Lorsque l'observateur s'éloigne encore plus, il n'arrive ni à distinguer la profondeur des immeubles, ni que immeuble 1 est plus près du trottoir que immeuble 2. L'ensemble des deux immeubles est alors remplacé par un *objet-silhouette* (un polygone par exemple) sur lequel on plaque l'image de ces immeubles. Dans les paragraphes suivants, nous allons préciser comment on génère la texture à plaquer sur l'*objet-silhouette*, et quels critères nous utilisons pour décider du moment où il faut remplacer un ensemble d'objets par un *objet-silhouette*.

Ce type de problème est au cœur des problèmes de la gestion des niveaux de détails. Posé de cette façon, en toute généralité, c'est un problème très difficile. Pour le

résoudre, nous avons été amenés à nous placer dans un cadre restrictif que nous définirons dans le paragraphe 4.1. D'autre part, l'automatisation, très souhaitable, de la définition de l'*objet-silhouette* se révèle très difficile, et demande beaucoup d'*intelligence* à l'ordinateur. Pour cette raison, nous avons renoncé à une telle automatisation et nous avons confié cette tâche à l'utilisateur qui doit lui même fabriquer ses *objets-silhouettes.*

Dans le paragraphe 4.1, nous discuterons de la définition de la texture. Dans le paragraphe 4.2, la façon dont on déterminera l'*objet-silhouette* sera exposée. Dans le paragraphe 4.3 nous expliquerons comment nous effectuons le passage entre la représentation de l'ensemble des objets 3D et la représentation avec l'*objet-silhouette* texturé. Nous discuterons des limitations de cette méthode dans le paragraphe 4.4.

4.1 La détermination de la texture

Comme nous l'avons signalé ci-dessus, la couleur de l'*objet-silhouette* sera celle d'une texture que nous plaquerons sur cet objet. Pour obtenir cette texture, on calcule une image de l'ensemble des objets que nous voulons remplacer par l'*objet-silhouette.* Cette image-texture, qui sera désignée par image de première passe dans la suite, doit vérifier deux propriétés indispensables :

- elle doit possèder suffisamment d'informations sur l'ensemble des objets, donc elle doit conserver une quantité suffisante de détails sur ces objets.

- elle ne doit pas présenter de déformation due à la projection des objets 3D sur un plan 2D.

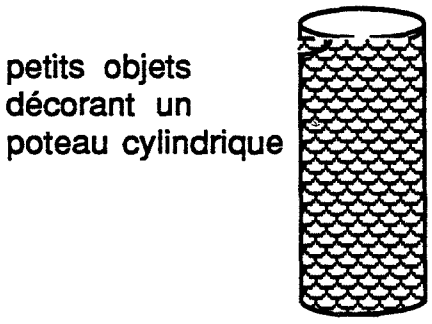


FIGURE 4.7

Supposons que les objets étudiés ne soient pas agencés d'une façon plane, mais qu'ils soient disposés sur une surface gauche. Une image de ces objets présentera forcément des déformations. Par exemple, la figure 4.7 présente un ensemble de petits objets décorant un poteau cylindrique. Une image de ces objets présentera des déformations, surtout sur le bord du cylindre. En conséquence, la seconde propriété ne sera pas satisfaite. Pour cela, nous émettrons une hypothèse restrictive pour l'application de notre méthode :

- les objets à remplacer par l'*objet-silhouette* texturé doivent être agencés d'une façon plutôt plane (comme s'ils reposent sur un plan), ou, au moins, doivent paraître tels à l'observateur lorsqu'il les voit de loin.

Dans l'exemple du toit, les tuiles sont agencées d'une façon plane ; donc nous pouvons utiliser notre méthode. Par contre, un ensemble d'immeubles, bordant une rue, ne possède pas généralement cette propriété (cf figure 4.6); notre méthode n'est donc pas applicable. Dans le cas où l'observateur regarde ces immeubles de loin, on peut considérer qu'ils sont disposés d'une façon plane et on pourra utiliser notre méthode.

Pour calculer une image d'un ensemble d'objets qui vérifie les deux propriétés énoncées, nous calculons l'image de la projection orthogonale de ces objets sur une face de leur boîte englobante. La projection orthogonale possède deux avantages par rapport à la projection perspective :

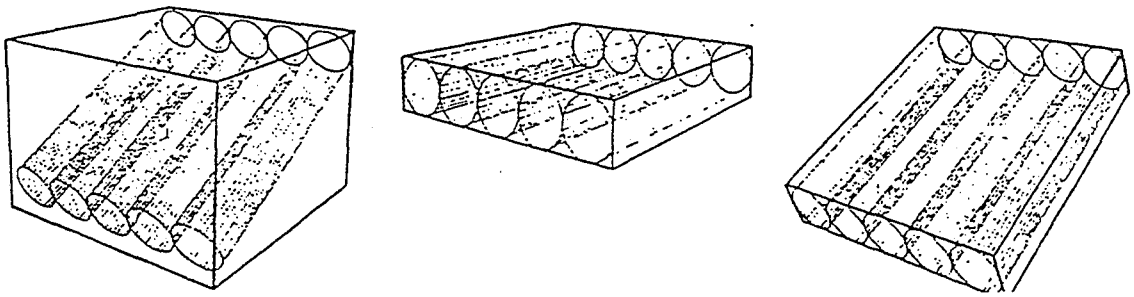
- l'image de la projection orthogonale d'un ensemble d'objets agencés d'une façon plutôt plane ne présente aucune déformation.
- cette image conserve tous les détails de cet ensemble d'objets.

Il faut noter que le second avantage n'est vrai que si la boîte utilisée pour effectuer la projection orthogonale approxime bien l'ensemble des objets. Sinon, il y a perte d'informations : en effet, si on utilise une boîte qui approche assez mal l'ensemble des objets, ces derniers n'occuperont qu'une partie de l'image fabriquée à partir de leur projection sur une face de cette boîte. Le reste de l'image sera occupé par un fond qui n'a aucun sens en tant que texture à plaquer sur l'*objet-silhouette*. Si la partie occupée par ce fond dans l'image est importante, l'information apportée par la texture par rapport à l'ensemble des objets risque d'être pauvre et par suite risque de nuire au réalisme de l'image

perspective de la scène finale (avec *objet-silhouette* texturé que nous appellerons image de deuxième passe dans la suite).

La boîte englobante recherchée doit donc avoir la propriété de bien approcher l'ensemble des objets de façon à ce que ces derniers occupent toute l'image. La détermination de cette boîte n'est pas immédiate et peut être assez difficile.

Nous avons laissé à l'utilisateur le soin de déterminer lui même cette boîte. La difficulté de cette tâche dépend de la nature de l'ensemble des objets. Supposons que ces derniers soient fabriqués, l'un à côté de l'autre, dans le plan xoy par exemple, et qu'ils soient ensuite placés ensemble dans la scène. Dans ce cas la détermination de la boîte est immédiate : il suffit de prendre la boîte englobante de ces objets lorsqu'ils sont dans le plan xoy, et de lui appliquer les mêmes transformations que celles qui permettent de mettre en place l'ensemble des objets dans la scène. Par exemple, si l'utilisateur veut modéliser un toit de tuiles, il peut le fabriquer dans le plan xoy d'abord, puis le mettre en place tout entier, au lieu de placer les tuiles une à une dans la scène. La figure 4.8 (c) montre la boîte englobante approximant d'une façon précise le toit lorsqu'il est en place dans la scène. Elle a été calculée lorsque le toit était dans le plan xoy comme le montre la figure 4.8 (b). La figure 4.8 (a) montre une mauvaise boîte du toit ayant ses faces parallèles aux plans du repère du monde.



a b c

FIGURE 4.8

Dans le cas où l'utilisateur est obligé de placer chaque objet tout seul dans la scène, la détermination de la boîte englobante devient plus difficile. On peut donner comme exemple un ensemble d'immeubles. C'est seulement lorsque l'observateur les regarde de

loin que ces immeubles paraissent agencés d'une façon plane ; de près, ce n'est pas le cas. En conséquence, l'utilisateur est obligé de les placer un à un dans la scène.

La détermination de la boîte englobant d'une façon précise l'ensemble des objets peut donc alourdir le travail de l'utilisateur. Ce dernier est le mieux placé pour se faciliter, si possible, ce travail, selon la manière suivant laquelle il modélise ces objets. Les deux façons suivant lesquelles on peut modéliser le toit et le mettre en place en est un exemple (placer tout le toit dans la scène une fois pour toutes, ou placer chaque tuile séparément).

L'inconvénient majeur de l'utilisation de la projection orthogonale de l'ensemble des objets pour fabriquer la texture à plaquer sur l'*objet-silhouette* est qu'elle nécessite la fabrication d'une boîte englobante très précise. Nous avons donné à l'utilisateur un autre choix qui consiste à faire une image de la projection perspective de l'ensemble des objets : l'œil regarde au centre de l'ensemble des objets, et est situé à une distance d de ce centre. Cette distance correspond à la position de l'œil la plus proche du centre de l'ensemble des objets dans toutes les images à produire. Avec ce choix de la position de l'œil, l'image perspective vérifie la seconde propriété exigée au début de ce paragraphe pour la texture à plaquer sur l'*objet-silhouette*. Quant à la première propriété, il y aura éventuellement des déformations à cause de la projection perspective. Ces déformations peuvent être plus ou moins importantes selon la manière dont les objets sont disposés. C'est à l'utilisateur de prévoir si ces déformations seront importantes et si elles nuiront au réalisme de l'image de deuxième passe de toute la scène avec l'*objet-silhouette* texturé. Dans ce cas, il sera amené à utiliser la projection orthogonale qui est le choix par défaut.

4.2 La détermination de l'objet-silhouette

L'ensemble des objets à remplacer par l'*objet-silhouette* étant agencés d'une façon plutôt plane (suite à l'hypothèse restrictive que nous avons imposée), l'*objet-silhouette* sera un objet 2D (polygone, cercle, ...). Celui-ci doit couvrir exactement tous les objets ; il ne doit pas déborder, sinon il risque de cacher d'autres objets derrière lui. Si l'*objet-silhouette* est un polygone, celui-ci peut être une des faces de la boîte englobante qu'on

utilise pour la projection orthogonale (cf §4.1) ; sa détermination pose donc les mêmes problèmes que ceux évoqués dans le paragraphe 4.2.

Etant données les difficultés que l'utilisateur peut rencontrer pour construire l'*objet-silhouette* 2D, il est possible, malgré tous les efforts, que ce dernier déborde de l'ensemble des objets 3D. En conséquence, il risque de cacher d'autres objets situés derrière lui que l'observateur est censé voir, nuisant ainsi au réalisme de l'image. Par exemple, imaginons que de petits objets soient disposés comme le montre la figure 4.9. L'utilisateur va donc être obligé de définir un *objet-silhouette* assez complexe (un polygone pour cet exemple), ce qui augmente encore les difficultés et le risque de commettre des erreurs.

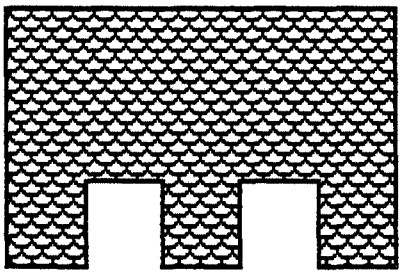


FIGURE 4.9

Pour éviter les erreurs possibles et pour alléger le travail de l'utilisateur, nous avons toléré que ce dernier fabrique un *objet-silhouette* de taille plus grande que l'ensemble des objets 3D. Les régions de l'*objet-silhouette* qui débordent seront détectées et déclarées complètement transparentes. Pour détecter ces régions, nous utilisons le test suivant : pour décider si un point vu sur l'*objet-silhouette* doit être transparent ou pas, on le transforme dans l'espace de l'image de première passe (qui est l'espace de la texture) ; s'il est à l'extérieur du champs de vue ou à l'intérieur de ce dernier mais qu'il se projette sur l'écran dans une région qui possède la couleur du fond, ce point est déclarée transparent.

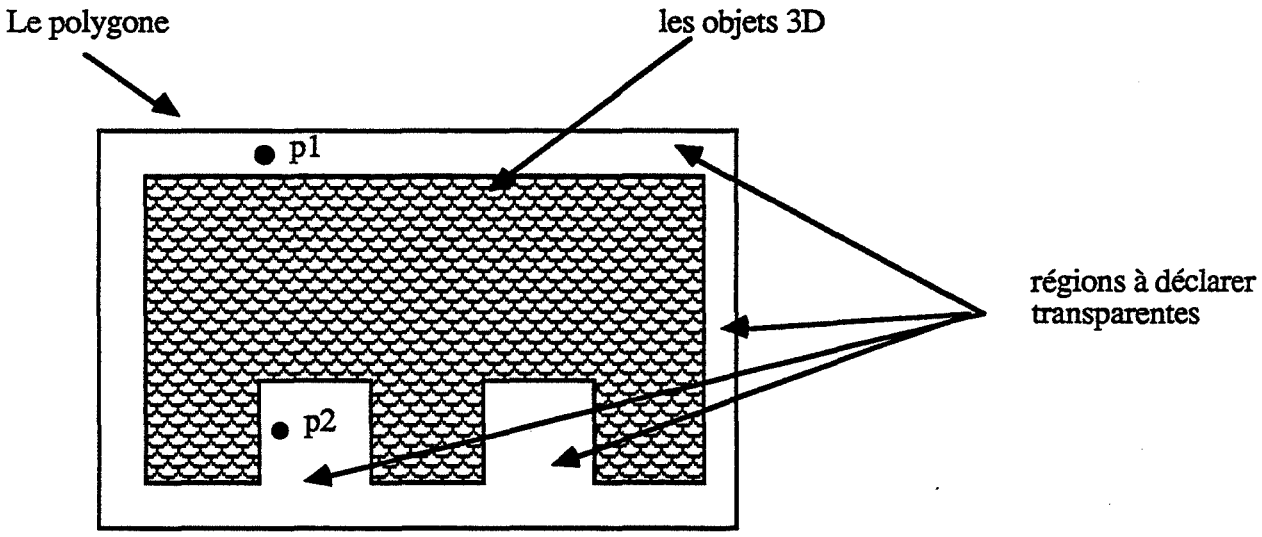


FIGURE 4.10

Par exemple, soient p1 et p2 deux points de l'*objet-silhouette* qui doit remplacer les objets de la figure 4.9, et qui se trouvent sur des régions débordant de l'ensemble de ces objets (cf figure 4.10). Supposons qu'en un pixel on voit le point p1, il s'agit de pouvoir décider s'il doit être déclaré transparent. Pour cela, on calcule les coordonnées de ce point dans le monde, puis on le projette dans l'espace de l'image de la première passe, qui est l'espace de la texture.

D'autre part, supposons que la boîte englobante de l'ensemble des objets 3D utilisée pour le calcul de cette image de première passe approche d'une manière très précise ces objets, de façon à ce que l'ensemble des objets occupe tout l'écran. Comme le point p1 déborde de l'ensemble des objets, sa projection dans l'espace de la texture (l'espace de l'image de première passe) sera forcément en dehors du champs de vue (cf figure 4.11). C'est de cette façon qu'on décide que le point p1 doit être complétement transparent.

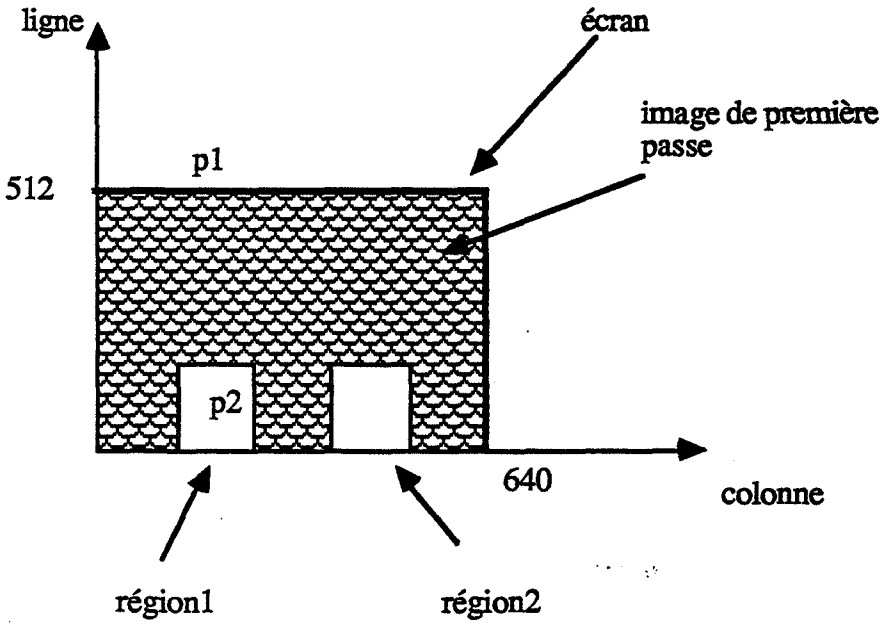


FIGURE 4.11

Dans le cas où la boîte n'approxime pas bien l'ensemble des objets, l'image de ces derniers n'occupera pas tout l'écran, mais il y aura des régions sur l'écran où on voit le fond. C'est le cas des régions *région1* et *région2* sur la figure 4.11. Le point p2 se projette à l'interieur de l'écran mais dans une région où on voit le fond (*région1*) ; c'est de cette façon qu'on décide qu'il doit être déclaré transparent.

Ainsi, tout se passe comme si les régions de l'*objet-silhouette* qui débordent de l'enveloppe des objets 3D n'existaient pas.

4.3 Critère de platitude

Dans notre méthode, nous remplaçons un ensemble d'objets 3D par une texture 2D. C'est d'autant moins gênant en synthèses d'images que, par ailleurs, de nombreuses approximations sont faites lors de la création d'une image. L'une des approximations est celle due à la discrétisation de l'image. Grâce à cette discrétisation, nous allons pouvoir définir un critère de *platitude* qui nous permettra de choisir entre une vue de l'ensemble des objets 3D et une vue de l'*objet_silhouette* texturé. En effet, il existe une distance à l'œil au-delà de laquelle la dimension de la projection de l'épaisseur de la boîte englobante des objets à remplacer par l'*objet_silhouette* texturé sera inférieure à la taille d'un pixel. A cette distance, on ne peut donc plus percevoir sur l'image synthétique la géométrie 3D des objets originaux, on peut donc les remplacer par une texture. Par exemple, selon l'ensemble des objets 3D qu'on souhaite remplacer par un *objet-silhouette*, cette épaisseur correspond à celle de la projection d'un balcon lorsqu'il s'agit d'une façade de la figure 4.6, à l'épaisseur de la projection des deux immeubles lorsqu'il s'agit de remplacer ces immeubles, ou de l'épaisseur de la projection du toit lorsqu'il s'agit de ce dernier (figure 4.12).

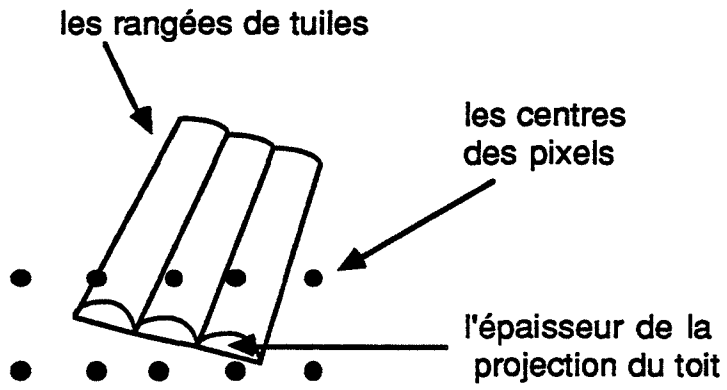


FIGURE 4.12

Il existe une analogie entre ce critère de platitude et les objets de référence (cf §2.1), qui sont utilisés par la fonction *Version* pour décider du moment où on peut réaliser une transition entre deux descriptions d'un objet. La boîte englobante de l'ensemble des objets à remplacer par l'*objet-silhouette* constitue l'objet de référence.

4.4 Le problème de l'éclairement

Notre méthode réalise deux images successives : une première, qui est la projection orthogonale de l'ensemble des objets 3D à remplacer sur les faces de leur boîte englobante, et une deuxième qui est la projection perspective de la scène contenant l'*objet_silhouette* texturé sur l'écran. Nous devons utiliser pour chacune de ces images une formule d'éclairement pour calculer la couleur de chaque pixel. Ces deux formules de calculs d'éclairement doivent être choisies de façon à obtenir des résultats égaux avec l'objet_silhouette et les objets 3D. Nous allons détailler ce problème par rapport aux formules de Phong que nous utilisons par ailleurs couramment pour réaliser nos images.

Rappelons que le calcul d'éclairement selon la formule de Phong combine par addition trois termes, respectivement nommés ambiant, diffus et spéculaire. Il faut que la somme des valeurs de ces trois termes soient les mêmes si l'on calcule l'image avec l'ensemble des objets 3D ou avec l'*objet_silhouette* texturé. Supposons pour le moment que l'ensemble des petits objets ne soient pas spéculaires. La formule de Phong se restreint alors à la somme des deux premiers termes. Cette somme est indépendante de la position de l'oeil et dépend seulement de la couleur propre de l'objet. Elle possède donc, en chaque pixel, la même valeur si on calcule l'image de la projection orthogonale des petits objets sur leur boîte englobante, ou si on calcule l'image de la projection perspective de ces objets. Par conséquent, lors de la visualisation de l'*objet_silhouette* texturé, il suffit de garder la couleur de la texture comme couleur des pixels sur lesquels se projette l'*objet_silhouette* en question. Autrement dit, on ne fait pas de calcul d'éclairement pour un pixel où on voit l'*objet_silhouette.*

Dans le cas où les objets 3D sont spéculaires, la formule de Phong contient une composante spéculaire qui dépend de la position de l'oeil. Elle n'aura donc pas la même valeur lorsqu'on calcule l'image de la projection orthogonale des objets 3D et l'image de leur projection perspective. Nous ne pouvons donc pas garder la valeur de la texture lorsqu'on visualise l'*objet_silhouette* qui remplace ces objets. Cette composante spéculaire est de la forme :

$$Ks\,(\vec{R}.\vec{V})^{n} \qquad (1)$$

Une solution possible consiste à ne calculer que la somme des deux premiers termes de la formule de Phong lorsqu'on calcule l'image de première passe et à stocker cette

valeur dans la texture. En même temps, on stocke le vecteur R pour chaque pixel. Plus tard, lorsqu'on calcule l'image avec l'*objet_silhouette* texturé, on lit dans ce fichier les valeurs de R et on calcule la quantité (1) qu'on ajoute à la valeur de la texture en ce point. Ainsi, on arrive à prendre en compte les effet spéculaires. L'inconvénient de cette technique est sans doute que nous aurons beaucoup d'informations à stocker.

Supposons que l'observateur reste dans une région de telle manière que le centre cc de cette région soit une approximation acceptable des positions de l'œil pendant les différentes images à produire. Dans ce cas, nous pouvons utilisé cc pour calculer la formule d'éclairement de Phong complète (avec la quantité (1)) pendant le calcul de l'image de première passe.

4.5 Résultats et perspectives

Nous présentons ici une application de notre méthode sur les toits d'une maison. Cette dernière a été modélisée par M. Frank Chopin avec notre modeleur CASTOR(cf chapite 1).

Trois différents points de vue ont été utilisés pour produire deux images de la maison dont l'une est avec un toit 3D et l'autre avec un polygone texturé. Les images 1(a) et 1(b) montrent respectivement une vue de l'ensemble des tuiles 3D, et une vue du polygone texturé. Ces deux images sont prises d'une même position de l'oeil, très près de la maison, telle que le critère de platitude ne soit pas satisfait. Nous remarquons que notre méthode aboutit à une image non réaliste dans ce cas. Les images 2(a) et 2(b) correspondent à une autre position de l'oeil, plus loin de la maison, telle que le critère de platitude ne soit pas satisfait non plus. On remarque que le réalisme de l'image obtenue par notre méthode est acceptable bien que le critère de platitude ne soit pas vérifié. Enfin, les images 3(a) et 3(b) correspondent à une position de l'œil telle que le critère de platitude soit satisfait. On remarque qu'il est difficile de distinguer entre l'images des toits avec tuiles et celle avec polygone texturé. La scène 3D contient 140 000 polygones, alors que la scène avec objet-silhouette texturé contient 22 000 polygones. Le gain du temps de calcul était de l'ordre de 75%.

Nous avons utilisé l'algorithme d'Atherton pour calculer nos images (cf chapitre 1). La méthode d'antialiassage utilisée pour la mise en perpective de la texture sur l'*objet-silhouette* 2D est la méthode de L. Williams [Will 83] (cf §4.2).

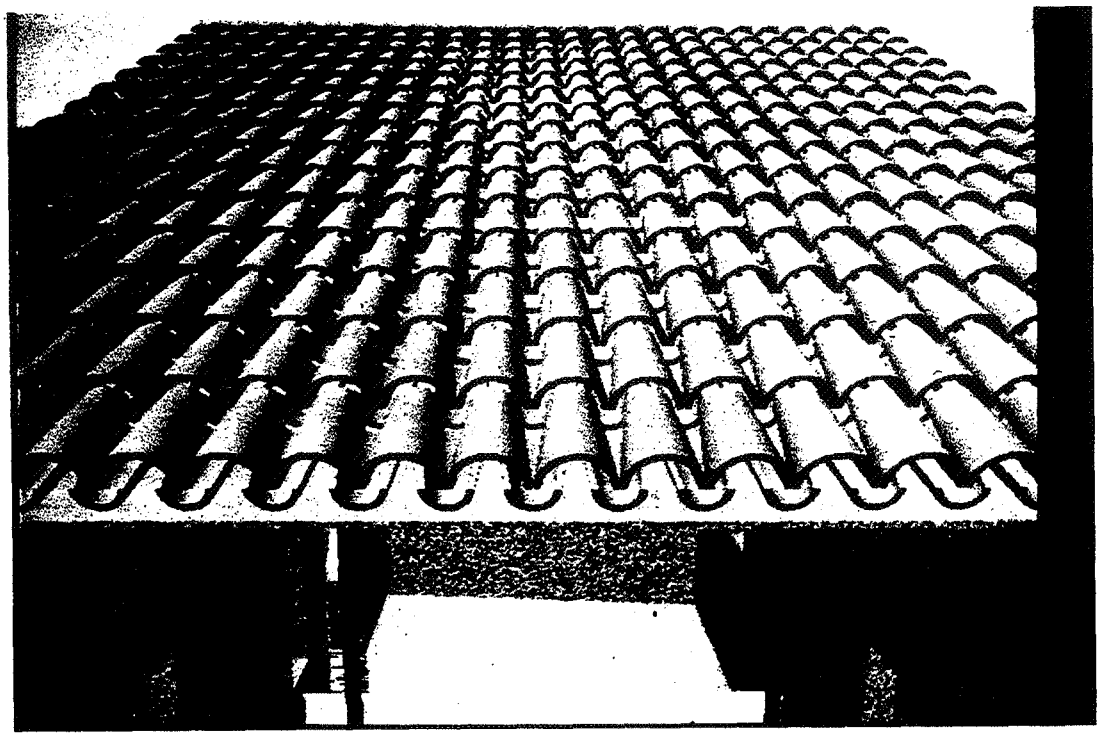


Image 1 (a) : vue de près du toit 3D



Image 1 (b) : vue de près du polygone texturé

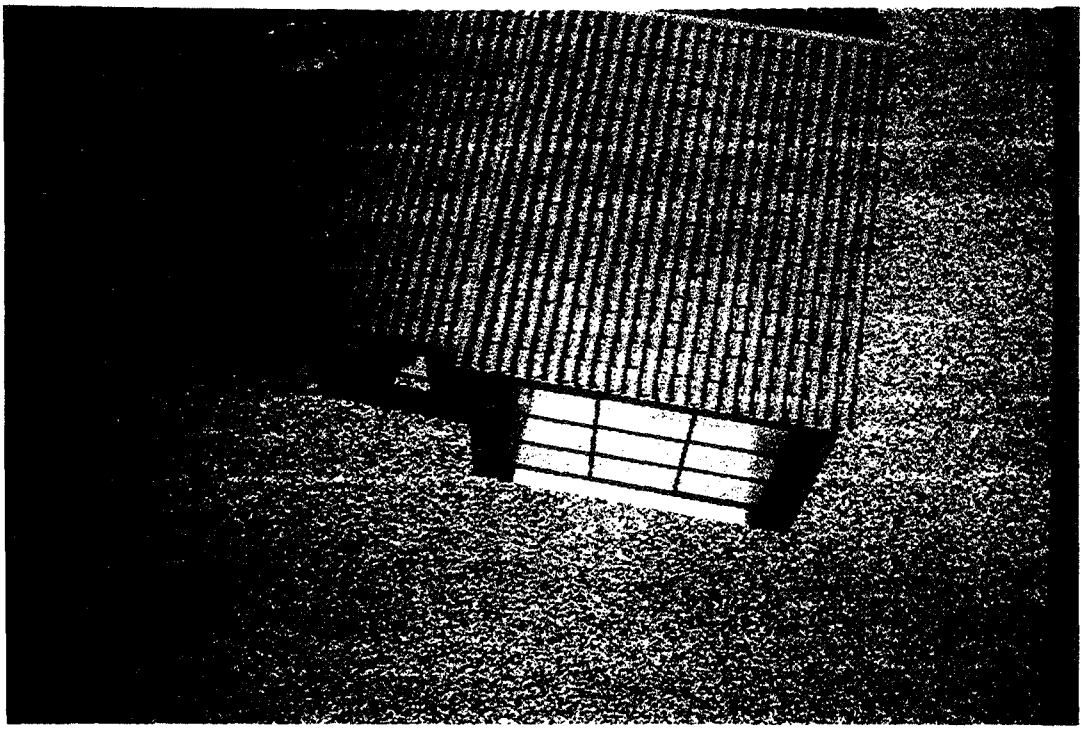


Image 2 (a) : vue du toit 3D, le critère de platitude n'est pas satisfait

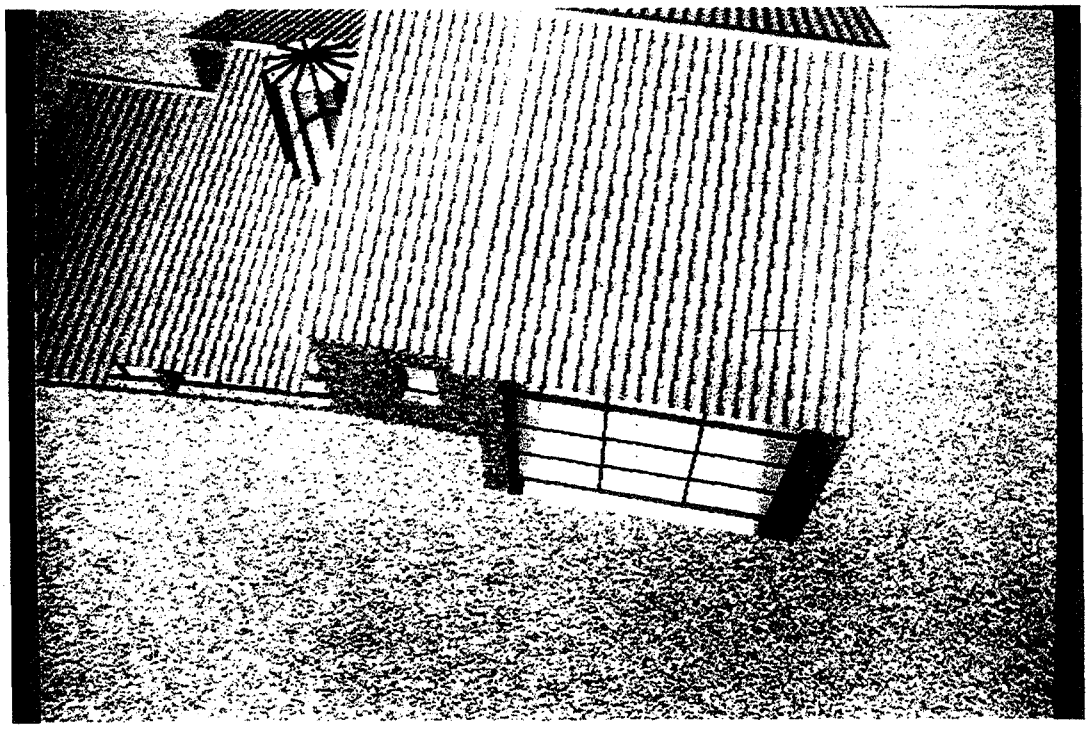


Image 2 (b) : vue du polygone texturé, le critère de platitude n'est pas satisfait

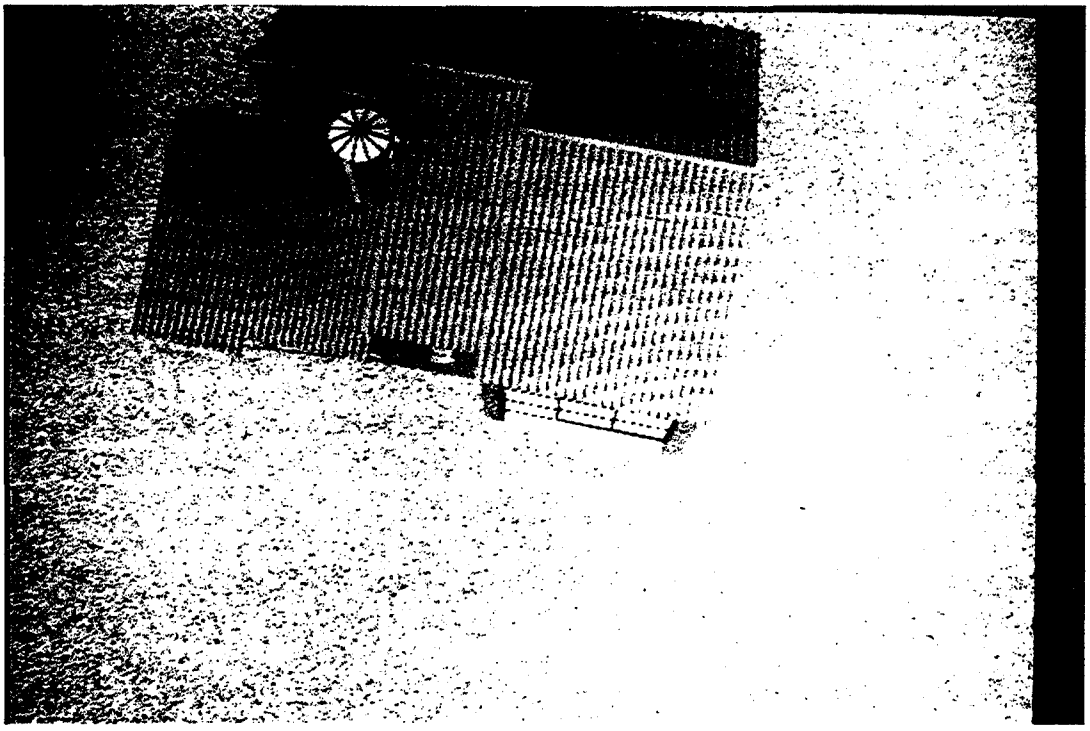


Image 3 (a) : vue du toit 3D, le critère de platitude est satisfait

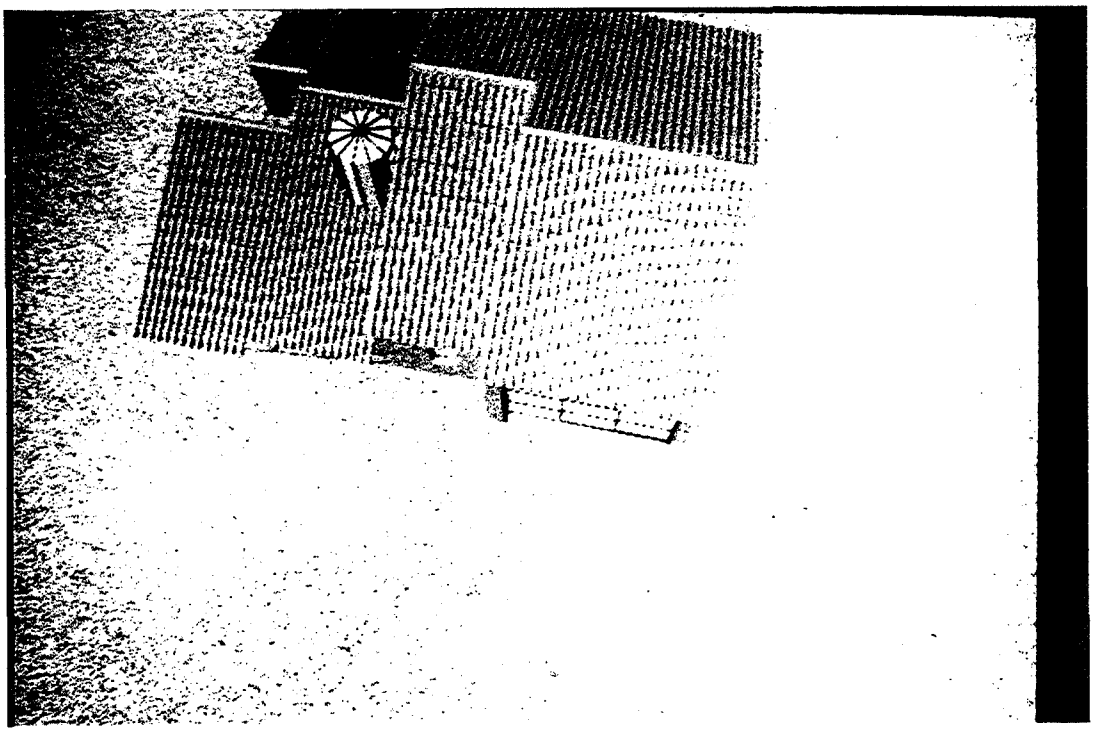


Image 3 (b) : vue du polygone texturé, le critère de platitude est satisfait

L'hypothèse restrictive que nous avons introduite dans le paragraphe 4.1 était essentiellement due au problème des déformations, qui seront présentes dans la texture, si les objets à remplacer par l'*objet-silhouette* sont disposés sur une surface gauche. Une solution pour éviter cette restriction serait d'utiliser une vignette prise sur l'image de ces objets et de plaquer cette vignette sur l'*objet-silhouette*. La vignette serait choisie de façon à ce que les objets y soient le moins déformés possible (cf figure 4.13). Dans ce cas, l'*objet-silhouette* n'est pas forcément un objet 2D mais peut être une primitive de modélisation 3D quelconque (un cylindre pour l'exemple de la figure 4.7).

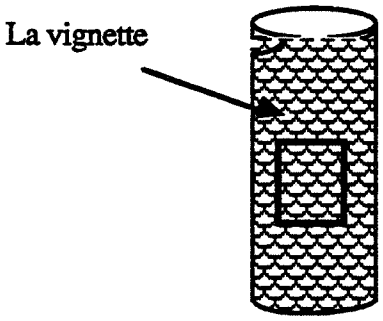


FIGURE 4.13

Le problème de cette démarche est qu'on perd la possibilité du test que nous utilisons pour repérer les points sur l'*objet-silhouette* qui doivent être transparents. En effet, le transformé d'un point sur l'*objet-silhouette* dans l'espace de la texture n'est plus nécessairement dans la vignette. Il faut pouvoir trouver d'autres tests.

5 LES OBJETS FONDANTS

5.1 Principe

Dans le paragraphe 3.1, nous avons introduit la fonction *Version* qui permet à l'utilisateur d'utiliser plusieurs descriptions différentes d'un même objet. Ces descriptions peuvent différer par leur géométrie, elles ne sont pas modélisées de la même façon. Dans ce paragraphe, nous allons introduire un outil qui permet d'obtenir plusieurs descriptions d'un objet qui gardent, cette fois-ci, la même géométrie mais qui changent d'aspect. L'aspect d'un objet est l'ensemble des paramètres suivants : sa couleur, un numéro de texture (à plaquer sur cet objet), les coefficients de transparence et de réflexion, l'indice de réfraction, etc.

L'idée générale consiste à affecter à l'objet un certain aspect tant qu'il est à une distance inférieure à une distance d1 de l'œil, puis à lui affecter un autre aspect lorsqu'il s'éloigne de plus d'une distance d2 de l'œil. Entre les deux distances d1 et d2, on interpole entre ces deux aspects, l'objet est alors appelé *objet-fondant*.. L'objet *fondant* est une solution pour réaliser le passage en douceur entre deux descriptions différentes de l'aspect d'un objet.

obj = FONDANT(aspect_proche,
aspect_fondant, distmin,distmax,
aspect_loin).

Aspect_proche est l'aspect de l'objet *obj* lorsqu'il est à une distance de l'œil plus petite que distmin.
Aspect_loin est l'aspect de l'objet *obj* lorsqu'il est à une distance plus grande que distmax.
Les distances distmin et distmax sont fixées par l'utilisateur.

Il faut remarquer que cette technique n'agit que sur la couleur du pixel, pas sur la forme des objets. Il s'agit donc d'une fonction d'ombrage qui travaille en mode pixel : en chaque pixel où est vu *obj*, on connaît la distance entre le point vu sur cet objet et l'œil ; on compare cette distance à celles données par l'utilisateur, et on utilise l'*aspect_proche*, l'*aspect_loin* ou l'interpolation entre ces deux aspects. Ainsi, l'objet *obj* n'est pas forcément visualisé entièrement avec le même aspect : une première partie peut être visualisée avec l'*aspect_proche*, une seconde partie avec l'*aspect _loin*, et une troisième partie par une interpolation entre ces deux aspects, selon la distance à l'œil du point vu en chaque pixel.

Dans ce qui suit, nous allons exposer la notion d'objet *fondant* lorsqu'on change la couleur et les coefficients spécifiant la matière de l'objet entre les deux aspects affectés à ce même objet.

5.2 Changement de la couleur entre deux descriptions

Supposons qu'il existe dans la scène un objet mince entre deux objets plus larges (joints, menuiseries, ...). Si cet objet est vu de loin, il risque de produire de l'aliassage sous forme de crénelé. Pour éliminer cet aliassage, nous utilisons les objets *fondants*. De près, cet objet mince possède une couleur propre ; de loin on veut qu'il ait la couleur des deux objets larges qui l'entourent afin d'éliminer l'aliassage ; entre les deux,

la couleur est obtenue par une interpolation linéaire entre ces deux couleurs.La figure 4.14 montre un objet long et mince, situé entre deux objets larges. Les parties de cet objet qui sont visualisées en utilisant l'*aspect_proche*, l'*aspect_loin*, ou en interpolant entre ces deux aspects, sont désignées sur la figure 4.15.

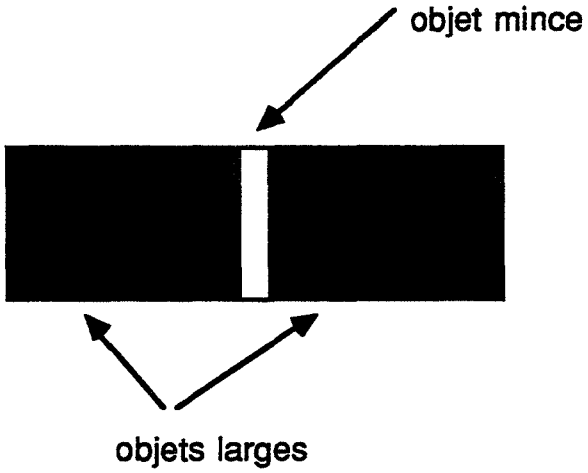


FIGURE 4.14

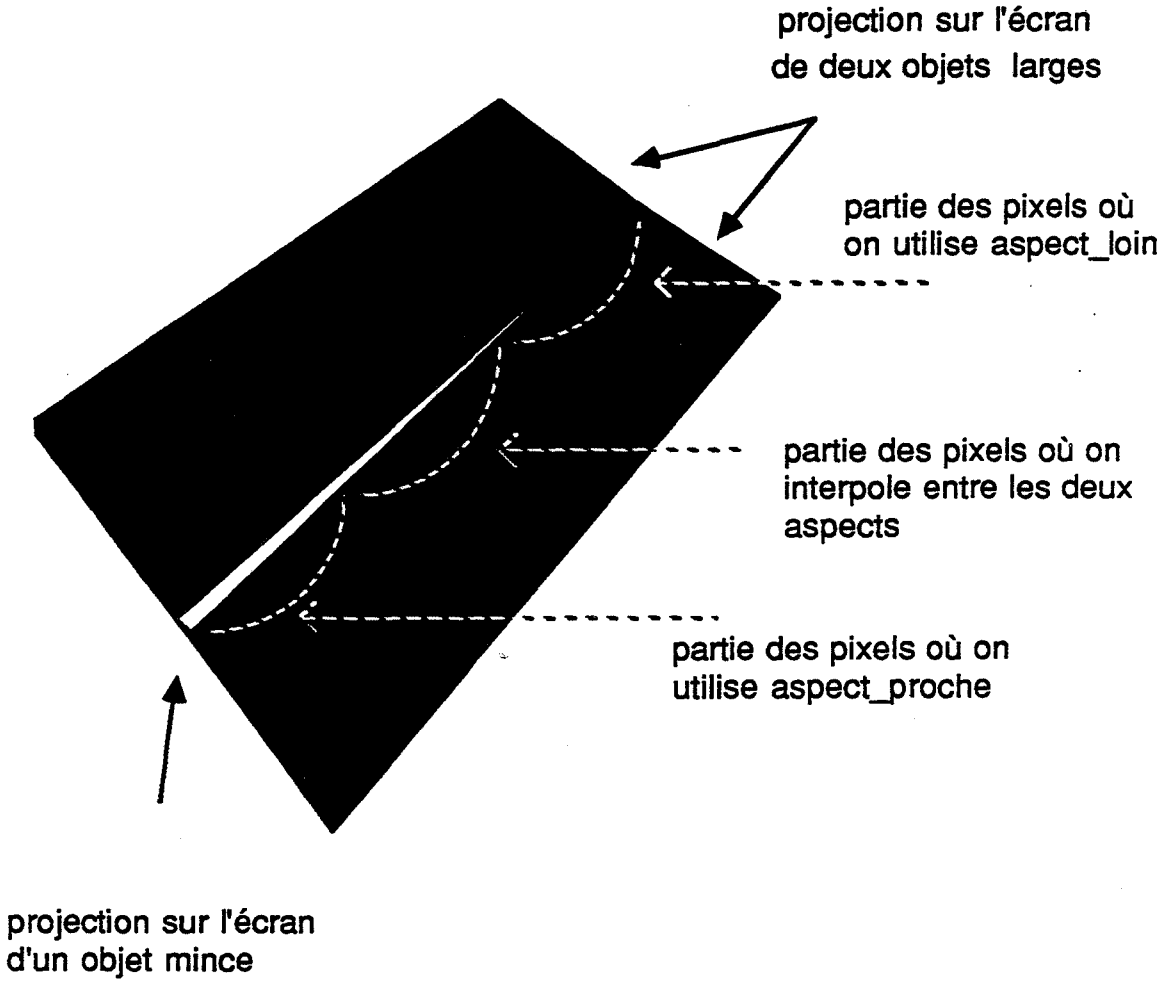


FIGURE 4.15

Ainsi, en donnant une nouvelle description d'un objet qui garde la même modélisation que la première description et change seulement d'aspect, nous arrivons à éviter les problèmes d'aliassage causés par les objets minces et longs.

5.3 Changement de la matière d'un objet entre deux descriptions

Lorsqu'un objet transparent et/ou réfléchissant est suffisamment loin, on peut supposer que l'observateur n'arrive plus à distinguer les objets qui se reflètent sur lui ou ceux qu'on voit à travers lui. Il est donc préférable de changer la matière dont il est constitué, le rendant opaque et diffus, pour économiser les calculs nécessaires aux réflexions et aux transparences que l'utilisateur juge inutiles pour une telle position de l'objet par rapport à l'observateur.

D'autre part, rappelons que pour réaliser un passage en douceur entre deux descriptions consécutives d'un même objet, MacDouglas [Macd 84] utilise la transparence progressive de la description qui doit disparaître, et l'opacité progressive de celle qui doit la remplacer (cf § 4.2). Un objet *fondant* qui change de matière entre deux positions n'est qu'une implémentation de cette méthode. En effet, pour avoir la transparence progressive pour la description qui disparaît, il suffit de la déclarer comme un objet *fondant* dans la zone de transition, le degré de transparence variant de 0 à 1 (0 -> opaque, 1 -> complètement transparent). De même, pour avoir l'opacité progressive de la nouvelle description, le degré de transparence de cette dernière passe de 1 à 0.

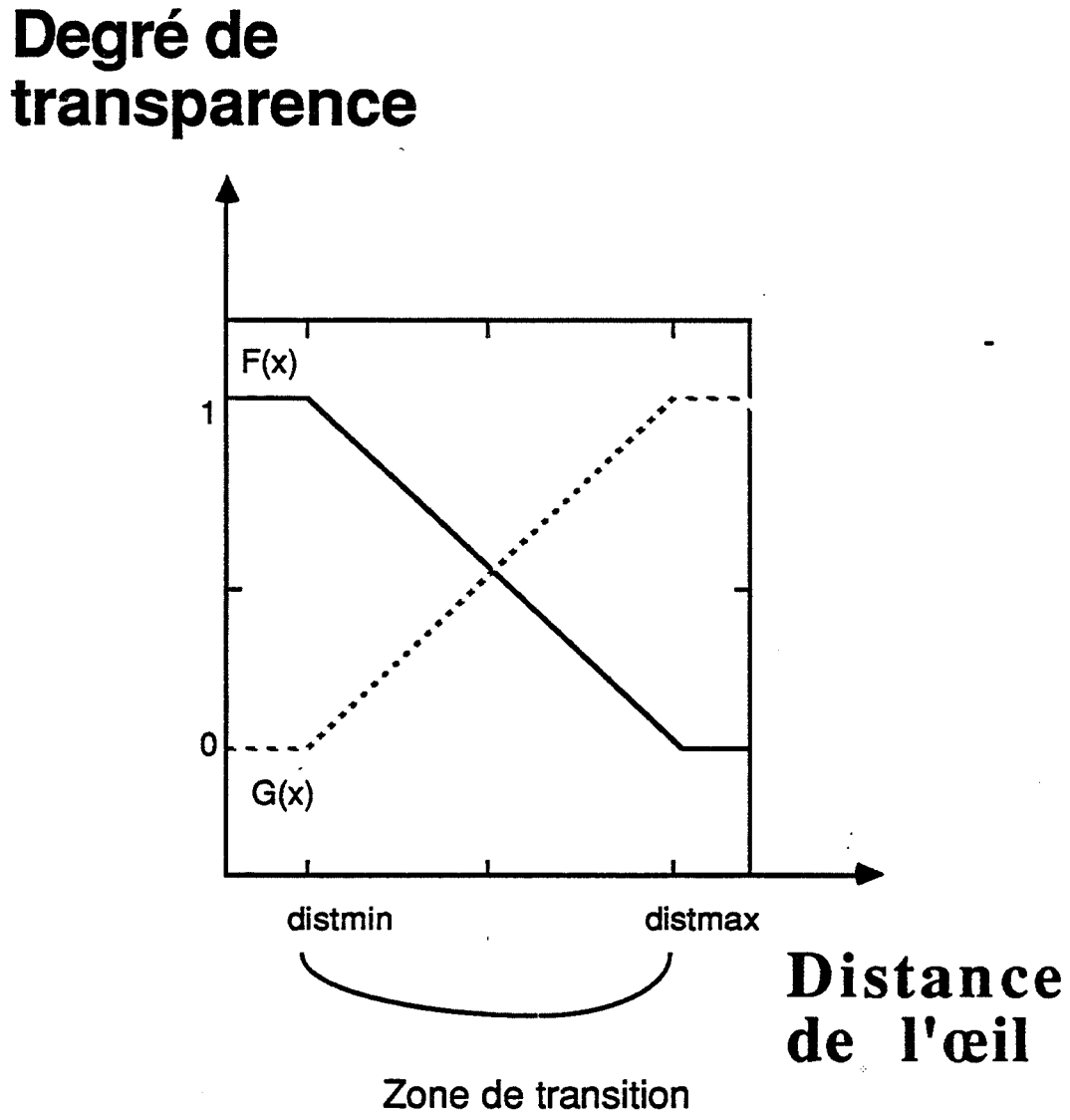


FIGURE 4.16

L'interpolation linéaire que nous avons utilisée pour le changement de couleur d'un objet *fondant* (§ 5.2) n'est pas suffisante pour le changement du degré de transparence introduit dans le but de réaliser le passage entre deux descriptions d'un même objet. En effet, soient F(x) et G(x) les fonctions présentées sur la figure 4.16, 1 - F(x) et 1 - G(x) sont respectivement les degrés de transparence de la description qui disparaît et de celle qui apparaît dans la scène. Au milieu de la zone de transition, les deux fonctions ont la même valeur. Les deux descriptions possèdent alors le même degré de transparence. Or la lumière traversant deux surfaces consécutives, ayant chacune un degré de transparence égal à 0.5, ne s'arrête pas forcément lorsqu'elle atteint la seconde surface, mais elle peut traverser cette dernière. Par suite, le fond de l'image sera visible à travers l'objet en transition, ce qui est

fortement indésirable. Pour résoudre ce problème, MacDouglas [Macd 84] a proposé d'utiliser les fonctions F(x) et G(x) de la figure 4.17. Ce choix des deux fonctions F(x) et G(x) résout le problème du fond soulevé plus haut ; cependant, il n'est pas sans inconvénient. En effet, il est indispensable que la nouvelle description soit devant l'ancienne, sinon, lorsque cette dernière, étant opaque, est enlevée de la scène à la distance distmax (cf figure 4.17), la nouvelle description sera visible soudainement. De plus, si la description à apparaître dans la scène ne couvre pas totalement l'ancienne description à la distance distmax, le passage entre ces deux descriptions à cet instant sera visible. Ainsi, ce choix des fonctions doit être utilisé dans le cas où la représentation de l'objet dans la scène devient de moins en moins complexe.

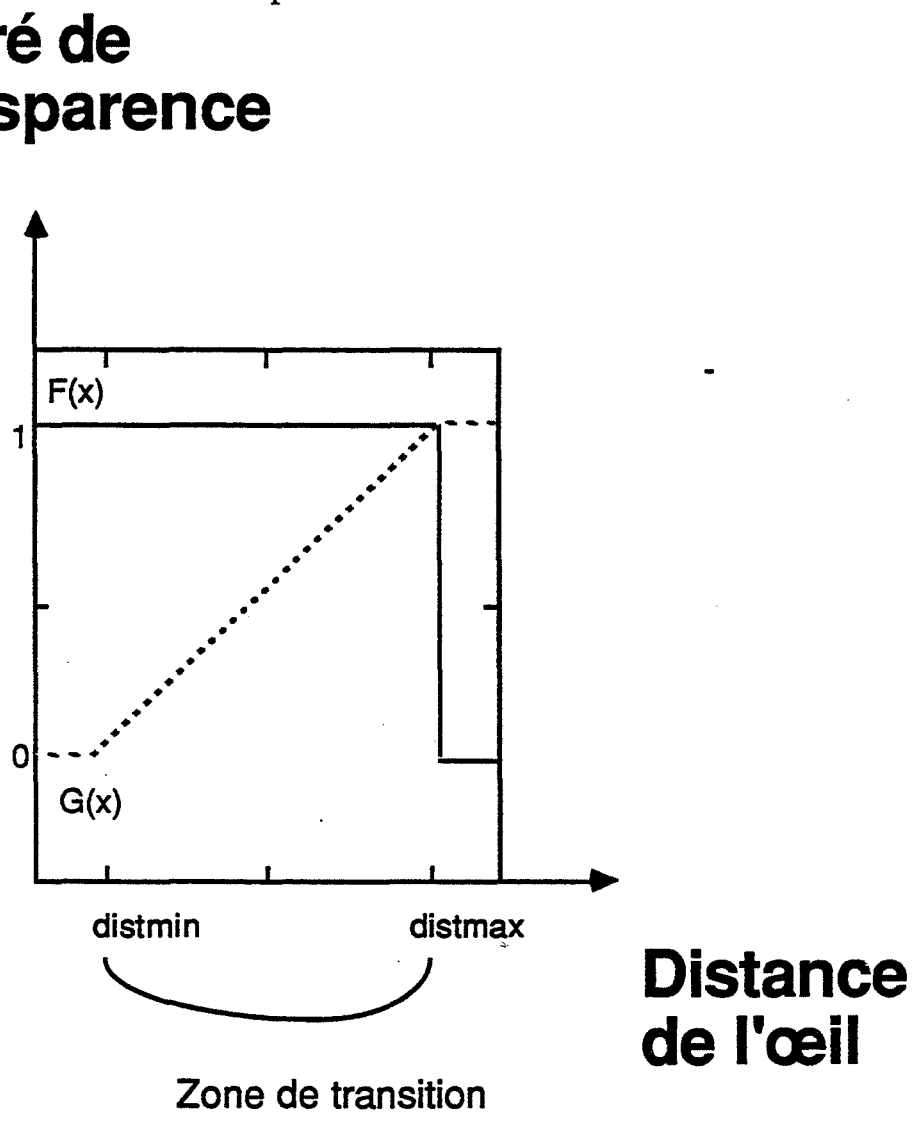


FIGURE 4.17

S'il est difficile de positionner correctement les deux descriptions, ou bien, si la description à apparaître dans la scène ne couvre pas l'ancienne description, MacDouglas propose les fonctions F(x) et G(x) de la figure 4.18, qui évitent ces problèmes et qui possèdent comme unique inconvénient l'élargissement de la zone de transition. C'est la solution que nous avons adoptée.

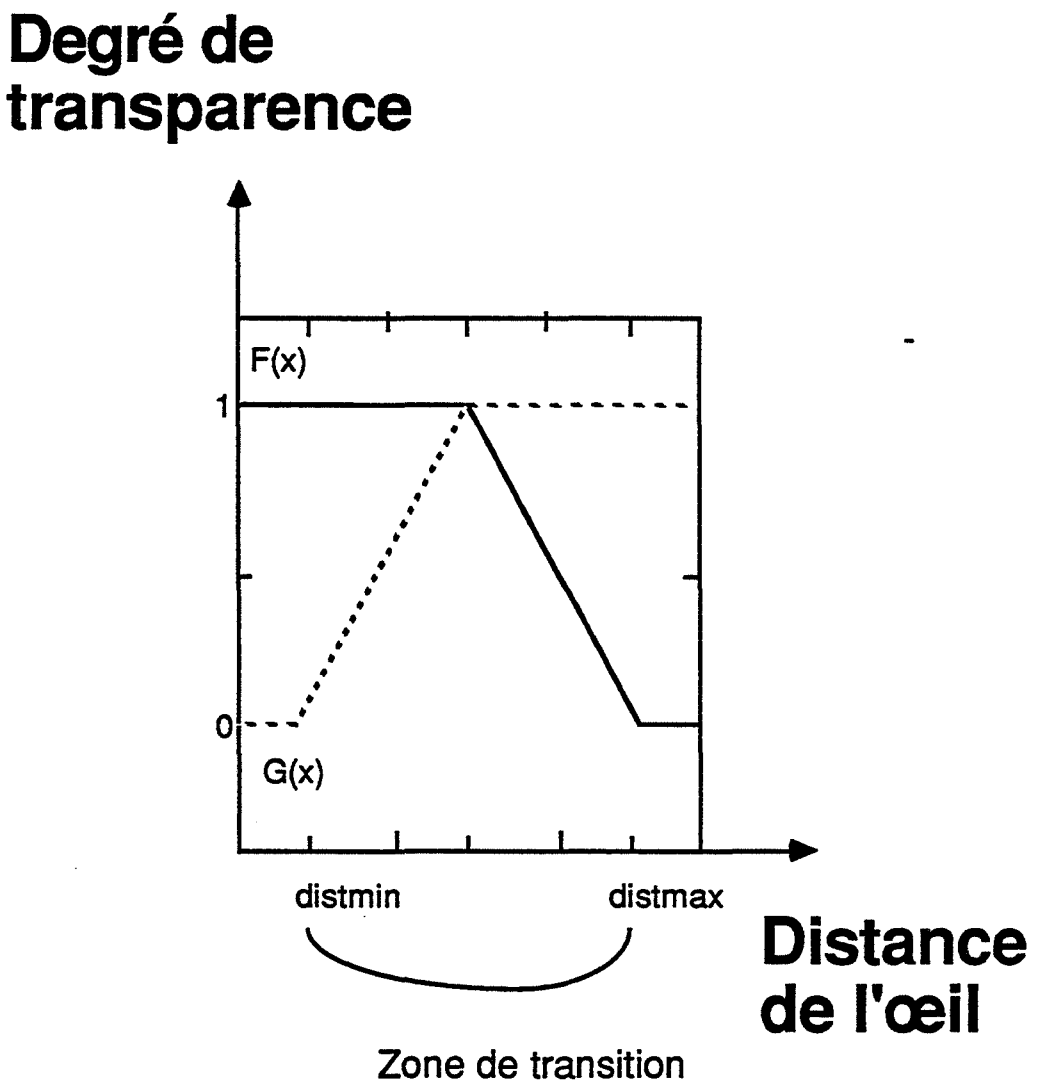


FIGURE 4.18

Ainsi, pour un objet *fondant* changeant de degré de transparence, il existe deux cas : s'il est introduit pour effectuer le passage entre deux descriptions d'un objet, les fonctions F(x) et G(x) de la figure 4.18 sont utilisées ; sinon, une interpolation linéaire est effectuée entre les deux degrés de transparence qui le définissent. Dans le dernier cas, c'est

l'utilisateur qui déclare cet objet comme *fondant* en précisant les deux valeurs entre lesquelles le degré de transparence de cet objet doit varier. Dans le premier cas, c'est la fonction *Version* qui déclare les deux descriptions entre lesquelles on souhaite effectuer un passage en douceur comme objets *fondants* (cf § 3.1), qui change de degré de transparence de 0 à 1 (pour la description qui disparaît) et de 1 à 0 (pour celle qui apparaît).

A première vue, les objets *fondants* ne permettent pas d'économiser de temps de calcul contrairement à ce que nous avons annoncé, dans l'introduction, pour les méthodes qui traitent le problème de niveaux de détails ; cependant leur utilisation pour traiter le cas d'un objet mince situé entre des objets épais (§ 5.2) permet de ne pas effectuer un sur-échantillonnage pour éviter l'aliassage sous forme de crénelé. Ainsi, on économise indirectement du temps de calcul. De même, changer la matière dont est constitué un objet transparent ou réfléchissant pour le rendre opaque et diffus lorsque l'observateur n'arrive plus à distinguer les effets optiques sur cet objet, permet de ne pas effectuer un travail inutile et par suite de gagner du temps.

Notons enfin qu'il existe une analogie entre la méthode de Williams [Will 83], pour traiter l'aliassage lors de la mise en perspective d'une texture, et la technique des objets *fondants* que nous avons décrite dans ce paragraphe. En effet, Williams effectue un pré-traitement sur la texture initiale qui consiste à extraire de cette dernière plusieurs autres textures à des niveaux de complexité inférieurs. Pour chaque pixel, selon le taux de compression présent (qui est lié à la distance du point vu à l'œil), il choisit deux textures correspondant à deux niveaux consécutifs. Il mélange les deux couleurs tirées de ces textures en faisant une interpolation linéaire entre elles. Cela est tout à fait semblable à un objet *fondant* changeant de textures entre deux positions de l'œil. La différence est qu'outre la distance du point vu à l'œil, il existe d'autres facteurs qui influent de façon importante sur la valeur du taux de compression, et par suite sur le choix des textures à utiliser.

6 Conclusion

Dans ce chapitre, nous avons proposé quelques solutions pour résoudre un certain nombre de problèmes qui se posent lorsqu'on essaie de gérer les niveaux de détail dans un système de synthèse d'images. Ce problème a été relativement peu abordé dans la littérature bien qu'il soit, à notre avis, un point important en synthèse d'images.

Plusieurs problèmes assez difficiles dans la gestion des niveaux de détails ne sont pas encore résolus et susciteront sans doute l'intérêt des chercheurs dans le proche avenir. Un de ces problèmes est l'automatisation de la génération des différentes descriptions d'un objet. Dans ce mémoire, l'utilisateur doit lui même définir les différentes descriptions d'un objet (fonction Version), ce qui lui demande un effort considérable. De même, pour l'utilisation de la texture pour la gestion des niveaux de détail (cf § 4), c'est l'utilisateur qui définit l'*objet-silhouette*.

S'il est très difficile de définir d'une manière automatique l'*objet-silhouette* lorsque celui-ci peut prendre toutes les formes possibles et imaginables, ce problème est quand même simplifié dans le cas, très fréquent, où l'objet-silhouette 2D est un polygone. Nous avons commencé à essayer d'automatiser la fabrication d'un polygone-silhouette : en effet, à partir des sommets des boîtes englobantes des différents objets à remplacer par ce polygone, nous pouvons trouver le plan le plus proche au sens de moindres carrés et le déplacer ensuite pour fabriquer la boîte précise souhaitée de l'ensemble des objets et ainsi obtenir le polygone-silhouette.

CONCLUSION

Nous nous sommes préoccupés, dans ce mémoire, de la globalité du système de synthèse d'images. Nous nous sommes intéressés aux différentes étapes du pipeline graphique dans le but d'augmenter et l'évolutivité du système et sa flexibilité aussi bien pour le concepteur que pour l'utilisateur.

L'utilisation de la programmation orientée objet en synthèse d'images n'est pas nouvelle. L'originalité de ce que nous avons présenté dans le chapitre 2 vient du fait que nous avons posé le problème du choix des classes et de la hiérarchie entre ces classes pour atteindre nos objectifs de flexibilité et d'évolutivité. Nous avons proposé des règles pour effectuer les choix des classes, et, nous avons mis en cause la hiérarchie classique des classes largement utilisée dans les systèmes de modélisation. Nous avons proposé une hiérarchie originale plus efficace pour l'évolutivité du système.

Alors que la plupart des systèmes de modélisation que nous connaissons utilisent la programmation orientée objet d'une façon plutôt superficielle, nous avons analysé dans le second chapitre de cette thèse l'adaptation de la programmation orientée objet à un système de synthèse d'images et nous avons utilisé cette analyse pour que ce nouveau style de programmation contribue véritablement à la réalisation d'un système de modélisation orienté objet évolutif et flexible.

La composition des algorithmes de visualisation dans une même scène, malgré ses nombreux avantages, laisse apparaître plusieurs problèmes comme l'aliassage et les interactions entre les différentes bases de données. Si on a proposé plusieurs solutions pour le premier problème, le deuxième problème, plus difficile, n'a pas fait l'objet de publications. Dans le chapitre 3 de cette thèse , nous avons abordé ce problème et nous avons proposé des solutions originales.

L'histoire de la synthèse d'image, bien que très récente, se caractérise par une recherche constante vers un plus grand réalisme des images obtenues. On est ainsi passé des images noir et blanc en fil de fer à des images très réalistes pouvant utiliser plusieurs millions de couleurs avec, entre autres, l'élimination des parties cachées, la création d'ombres portées, de textures, d'effet de transparence, de réflexion, de réfraction. Un des grands obstacles dans la quête du réalisme est le temps du calcul qu'une image réaliste nécessite. Parmi les domaines qui ont peu attiré l'attention des chercheurs pour remédier à ce problème figure la gestion des niveaux de détails dans une scène. Dans le chapitre 4, nous avons décrit plusieurs outils et méthodes originaux pour assurer la gestion des niveaux de détails. Le problème de l'automatisation de la gestion des niveaux de détail, qui est un problème très difficile, n'est pas abordé dans ce chapitre et constitue un des axes pour la continuation de ce travail.

Bibliographie

[Alth 81] J.C. Althoff, L.P. Deutsh, "The Smalltak - 80 system", Byte, vol. 6, N° 8, 1981.

[Amar 89] M. Amara, "Interface graphique pour le modeleur CASTOR", Rapport de DEA, Ecol des Mines de Saint Etienne.

[Ansa 85] S. Ansaldi, L. de Floriani, B. Falcidieno, "Geometric modeling of solid objects by using a face adjency graph representation", SIGGRAPH'85, ACM Computer Grapl 19 (3), p. 131-139.

[Appe 68] A. Appel, "Some techniques for shading machine renderings of solids", SJCC, 1968, 37-45.

[Arge 88a] J. Argence, "Algorithmes pour le tracé de rayons dans le cadre d'une modélisation] arbre de construction", Thèse, Ecole des Mines de Saint Etienne, Novembre 1988.

[Arge 88b] J. Argence, "Antialiasing for ray tracing using CSG modeling", proc. CG International'88, New Trends in Computer Graphics, N. Magnenat-Thalmann and D. Thalmann eds., Springer Verlag, 1988, p. 199-208.

[Athe 83] P. R. Atherton, "A scan-line hidden surface removal procedure for construtive s(geometry", Computer Graphics, 17(3), 73-82.

[Baum 72] B. Baumgardt, "Winged-edge polyhedron representation", Stanford Artificial Intelligence Report CS-320, 1972.

[Bart 86] P. S. Barth, "An object-oriented approach to graphical interfaces", ACM Transactior on Graphics, 5, 2, April 1986, p. 142-172.

[Beig 88] M. Beigbeder, "Un dévéloppement pour la modélisation et la visualisation en synthè: d'images : CASTOR", thèse, Ecole des Mines de Saint Etienne, 1988.

[Beig 90] M. Beigbeder, B.Peroche, "un système de synthèse d'images 3D : ILLUMINES", Actes journées Unix de Grenoble.

[Beek 89] E. Beeker, "Application de la programmation orientée objet à la CAO", MICAD (2), 1989, p. 121-131.

[Bert 88] C. Bertin, "Une introduction à C++", rapport de recherche numéro 88.9, décembre ! Ecole des Mines de Saint Etienne.

[Blak 87] E. H. Blake, "A mectric for computing adaptive detail in animated scenes using Obje Oreiented programming", EUROGRAPHICS'87, p. 295 - 307.

[Blin 76] J. F. Blinn, M. E. Newell, "Texture and reflection in computer generated images", Communications of the ACM, 19, (10), Octobre 1976, p. 542 - 546.

[Blin 77] J. F. Blinn, "Models of light reflection for computer synthesized pictures", ACM

Computer OR, 11(2), 1977, 192-198.

[Born 86] A. Borning, R. Duisberg, "Constraint-Based tools for building user interfaces", ACM Transactions on Graphics 5, 4, Oct. 1986, p. 345-347.

[Bour 83] S. R. Bourne, "The UNIX system", Reading, Addison-Wesley, 1983.

[Bouv 85] C. Bouville, Bounding ellipsoids for ray-fractal intersection, Computer Graphics, 19(3), 1985, 45-52.

[Bree 87] D.E.Breen, P.H.Getto, A.A.Apodaca, D.G.Schmidt, B.D.Sarachan "The CLOCKWORKS: an object-oriented computer animation system", Eurographics, 1987, 275-282.

[Bree 89] D.E.Breen, "Message based object - oriented interaction modeling", EUROGRAPHICS'89, p. 489-503.

[Bron 84] W. F. BRONSVOORT, J.J. VAN WISK and F.W. JANSEN, Two methods for improving the efficiency of ray casting in solid modelling, CAD, 16(1), 1984.

[Card 85] L. Cardelli, R. Pike, "Squeak : A language for communicating with Mice", ACM SIGGRAPH '85, 19, 3, July 1985, p. 199-204.

[Catm 75] E. Catmull, "Computer display of curved surfaces", IEEE Conference Proceedings on Computer Graphics, Pattern Recognition and Data Structures, mai 1975, p. 11.

[Chmi 89] M.Chmilar, B.Wyvill, "A software architecture for integrated modelling and animation", Proceedings of CG International, 1989, 257-276.

[Clar 76] James H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithmes", Communications of the ACM, 19, (10), Octobre 1976, p. 547 - 554.

[Cook 82] R.L.Cook, K.E.Torrance, "A reflectance model for computer graphics", ACM Transactions on Graphics, 1(1), 1982, 7-24.

[Cook 84] R. L. Cook, "Shade trees", SIGGRAPH'84, p. 223-230.

[Cook 87] R. L. Cook, L. Carpenter, E. Catmull, "The Reyes Image Rendering Architecture", SIGGRAPH'87, p. 95-102.

[Cout 88] J. Coutaz, "Interface Homme-Ordinateur : conception et réalisation", thèse, université Joseph Fourier, 1988.

[Crow 82] F. C. Crow, "A more flexible image generation environment", Computer Graphics, 16(3), 9-18.

[Duff 80] T. Duff, "The soid and roid manual", NYIT Computer Graphics Laboratory internal memorandum, 1980.

[Duff 85] T. Duff, "Compositing 3-D rendered images", Computer Graphics, 19(3), 41- 44.

[Exco 87] T. Excoffier, E. Tosan, "Une méthode d'optimisation du lancer de rayon (Ray-Casting)", Micad, p. 549-563.

[Fert 89] G. Fertey, B. Peroche, "Sources directionnelles de lumières en tracé de rayons", PIXIM 1989, p. 219-232.

[Fert 90] G. Fertey, B.Peroche, J. Zoller,"Creating 3D scenes with constraints", Eurographic Workshop on Object Oriented Graphics, Juin 1990, p. 65-87.

[Fium 87] E. Fiume, D. Tsichritzis, L. Dami, "A temporal Scripting Language for object-oriented Animation", EUROGRAPHICS, 1987, p.283-293.

[Flei 87] K.Fleischer, "Implementation of a modeling testbed", cours de Siggraph : Object - Oriented Geometric Modeling and Rendering, 1987.

[Fole 82] J. D. Foley, A. Van dam, "Fundamentals of interactive computer graphics", Reading, Addison-Wesley, 1982.

[Fuji 86] A. Fujimoto, T. Tanaka, K. Iwata, "ARTS : accelerated ray-tracing system", IEEE Computer Graphics and Applications, 6, 4, 1986, p. 16 - 26.

[Ghaz 85] Ghazanfarpour-Kholendjany D., "Synthèse d'images et anti-aliassage", thèse de docteur-ingénieur, Ecole des Mines de Saint Etienne.

[Ghaz 90] Ghazanfarpour-Kholendjany D., "Problèmes de discrétisation et de filtrage pour la visualisation d'images numériques", thèse, Ecole des Mines de Saint Etienne.

[Gora 84] C.M.Goral, K.E.Torrance, D.P.Greenberg, B.Baittaile, "Modeling the interaction o: light between diffuse surfaces", Computer Graphics, 18(3), 1984, 213-222.

[Gour 71] H.Gouraud, "Continous shading of curved surface", IEEE Trans. Compt., 20(6), 1971, 623-628.

[Guit 90] P. Guitton, C. Schlick, "An object-oriented approach for realistic image synthesis", Eurographics Workshop on Object Oriented Graphics, juin 1990, p. 221-236.

[Gran 86] E.Grant, P.Amburn, T.Whitted, "Exploiting classes in modeling and display software", IEEE Computer Graphics and Applications, 6, 11, 1986, p. 13-20.

[Habi 88] M. Habib, R. Ducournau, "Mechanisms and algorithms for multiple inheritance in object oriented systems", Laboratoire d'informatique de Brest, rapport N° 1/88 Avril 1988.

[Hall 86] R. Hall, "Hybrid techniques for rapid image synthesis", cour SIGGRAPH : Image rendering triks, 1986.

[Hano 88] G. Hanoteau, "Ombres portées", Rapport de DEA, Ecole des Mines de Saint Etienne

[Herz 87] Brian Von Herzen, A. H. Barr, "Accurate triangulations of deformed, intersecting Surfaces", Siggraph 87, p. 103 - 110.

[Hill 86] R. D. Hill, "Supporting concurrency, comminication, and synchronization in Humar Computer Interaction - The Sassafras UIMS", ACM Transactions on Graphics 5, 3, July 1986, p. 179-210.

[Jaha 90] G. Jahami, B. Peroche, "Mixage d'algorithmes en synthèse d'images", Rapport de recherhce à l'école des Mines de Saint Etienne, 1990.

[Kaji 83] J. T. Kajiya, New techniques for ray tracing procedurally defined objects, Computer Graphics, 17(3), 1983, 91-102.

[Kay 86] T. L. Kay, J. T. Kajiya, "Ray tracing complex scenes", ACM Siggraph'86, 20(4), 269-278.

[Lane 80] J. M. Lane, T. Whitted, J. F. Blinn, L. C. Carpenter, "Scan line methods for displaying parametrically defined surfaces", Communication of the ACM, 23, (1), janvier 1980, p. 23 - 34.

[Laun 89] P. Launay, "Euclid-Is data concepts : an object oriented CAD/CAM system", MICAD (1), 1989, p.173-195.

[Lele 80] W. J. Leler, "Human vision, anti-aliasing, and the cheap 4000 line display", Computer Graphics, 14, (3), 1980, 308-313.

[Lore 87] W.Lorensen, M.Barry, D.Mclachlan, B.Yamrom, "Object-oriented software development in a non-object-oriented environment", cours Siggraph : Object-Oreiented Geometric Modeling and Rendering, 1987.

[Macd 84] Paul D. MacDougal, "Generation and management of object description hierarchies for the simplification of image generation", Ph. D., department of Computer and Information Science of the Ohio State University.

[Magn 85] N. Magnenat-Thalmann, D. Thalmann, and M. Fortin, "Miranim : an extensible director-oriented system for the animation of realistic images", IEEE Computer Graphics and Applications 4, 3 (March 1985).

[Mant 82] M. Mantyla, R. Sulonen, "A solid modeler with Euler operators", IEEE Computer Graphics and applications 2(7), p. 17-31, 1982.

[Masi 90] G. Masini, D. Colnet, A. Napoli, D. Léonard, K. Tombre, "les langages orientées objet", InterEditions, 1990.

[Meye 88] B. Meyer, "Object-oriented software construction", Prentice Hall, 1988.

[Mich 89] D. Michelucci, "CastorC", communication intern.

[Mich 87] D. Michelucci, "Les représentations par les frontières : quelques constructions ; difficultés rencontrées", thèse, Ecole des Mines de Saint Etienne.

[Nada 87] T. Nadas, A. Fournier, "GRAPE : An environment to build display processes", SIGGRAPH'87, p. 75-83.

[Newm 79] W. M. Newman, R. F. Sproull, "Principles of interactive computer graphics", Reading, MrGRAW-HILL, 1979.

[Nie 89] Z. Nie, B. Peroche, "Déformations libres et arbres de construction", BIGRE N° 67, Journées AFCET GROPLAN, Décembre 1989, p. 98-109.

[Parr 87] Scott R. Parry, "Free - form deformations in a constructive solid geometriy modeling system", thèse, department of civil engineering Brigham Young University.

[Pero 88] B.peroche, J.Argence, D.Ghazanfarpour, D.Michelucci, "La synthèse d'images",

Editions Hermes, 1988.

[Phon 75] B.T.Phong, "Illumination for computer generated pictures", Comm. ACM, 18(6), 1975, 311-317.

[Plem 87] D. Plemenos, "Un modèle hiérarchique pour la représentation de scènes tridimentionnelles", MICAD 1987, p. 521-533.

[Port 84] T.Porter, T. Duff, "Compositing digital images", Computer Graphics, 18(3), 253-259.

[Reis 86] S. P. Reiss, "An object-oriented Framework for graphical programming", ACM SIGPLAN Notices 21, 10, OCT. 1986, p. 49-57.

[Requ 80] A. A. G. Requicha, "Representations for rigual solids : theory, methods and system Computing Surveys, 12 (4), december 1980, p. 437-464.

[Reyn 82] C. Reynolds, "Computer animation with scripts and actors", SIGGRAPH 1982, p. 289-296.

[Reyn 87] C. Reynolds, "Flocks, Herds, and Schools : a distributed behavioral model", SIGGRAPH 1987, p.25-34.

[Rubi 80] S. M. Rubin, T. Whitted, "A 3-dimensional represebtation for fast rendering of complex scenes", SIGGRAPH 80, p. 110 - 116.

[Rubi 82] S. M. Rubin, "The representation and display of scenes with a wide range of detail", Computer Graphics and Image Processing, 19, 1982, p. 291 - 298.

[Sabe 87] P.Sabella, I.Carlbom, "An object-oriented approach to solid modeling for empirical data", cours Siggraph : Object - Oriented Geometric Modeling and Rendering, 1987

[Same 84] H. Samet, "The quadtree and related hierarchical data structures", ACM Computing Surveys, 16 (2), 187-260, Juin 1984.

[Scha 48] O. Schade, "Electro-optical characteristics of television systems", RCA Rev., 9 (6), june 5-37.

[Stro 86] B.Stroustrup, "The C++ programming language", Addison-Wesley, Reading, Mass 1986.

[Suth 74] I.E.Sutherland, R.F.Sproull, R.A.Schumacker, "A characterisation of ten hidden surface algorithms", Computing Surveys, 6(1), 1974, 1-55.

[Watk 70] G. S. Watkins, "A real time visible surface algorithme", University of Utah, UTEC-CSC, 1977, 70-101.

[Wegh 84] H. Weghorst, G. Hooper, D. p. Greenberg, "Improved computational methods for tracing", ACM Transactions on graphics, 3(1).

[Whit 80] T. Whitted, "An improved illumination model for shaded display", Computer Graphics and Image Processing, 23(6), 343-349.

[Will 78] L. Williams, "Shadow algorithms for computer graphics", Computer Graphics, 12(3), 242-248.

[Will 83] L. Williams, "Pyramidal parametric", SIGGRAPH'83, p. 1-11.

[Wyli 67] C. Wylie, G. Romney, D. C. Evans, A. Erdahl, "Halftone perpective drawing by computer", proc. AFIPS FJCC, 31, 49-58.

[Yamr 87] B. Yamrom, "Ray tracing : an object-oriented design", cours Siggraph : Object - Oriented Geometric Modeling and Rendering, 1987.