Architecture logicielle pour la simulation des transferts radiatifs

David Chamont

Université Henri Poincaré - Nancy I

Département de Formation Doctorale en Informatique — École Doctorale IAE + M

Architecture Logicielle pour la Simulation des Transferts Radiatifs

THÈSE

présentée et soutenue publiquement le 13 Novembre 1997

pour l'obtention du

Doctorat de l'Université Henri Poincaré – Nancy I

(Spécialité Informatique)

par

David CHAMONT

Composition du jury

Président :	Claude GODART
Rapporteurs :	Kadi BOUATOUCH Pascal GUITTON
Examinateur :	Jean-Claude DERNIAME
Directeur de thèse :	Jean-Claude PAUL

Centre de Recherche en Informatique de Nancy
INRIA–Lorraine

Aux Cœurlequins du neuf chemin neuf
Aux Gobs trotteurs d'antan

Remerciements

Ce travail a été initié, encadré et soutenu sans relâche par Jean-Claude Paul. Son contact fût l'occasion d'un enrichissement intellectuel et d'échanges humains que j'espère poursuivre. Qu'il trouve ici le témoignage de mon estime et de mon amitié.

Le soutien financier d'Électricité de France m'a permis d'opérer dans les meilleures conditions. C'est une chance supplémentaire que d'avoir été accueilli au sein du groupe ODI, sous la férule de Jacques Allard, soucieux que toutes les parties y trouvent leur compte et que ma thèse soit une réussite.

J'exprime mes plus vifs remerciements aux différents membres de mon jury, qui ont bien voulu examiner mes résultats et contribuer à l'amélioration du mémoire.

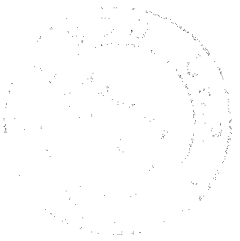
La réussite de ce travail reposait largement sur la bonne volonté, voire l'implication active de tous mes collègues passés et présents de l'équipe ISA. Merci à chacun d'entre eux pour la bonne humeur et l'énergie qu'ils ont sacrifiés à la cause commune.

La tâche la plus ingrate est sans doute celle de la relecture. Le mérite en revient principalement à Christine Chevrier, Karl Tombre, Gérald Masini, Laurent Alonso et Christophe Winkler. À charge de revanche.

J'ai le sentiment d'avoir bénéficié d'un environnement matériel et technique d'une qualité rare. Merci aux différents directeurs du CRIN et de l'INRIA-Lorraine de m'avoir accueilli dans l'enceinte du bâtiment Loria.

Enfin, les encouragements de Michelle Rumpf et les particules positives émises par l'équipe CMS de l'École Polytechnique m'ont bien aidé à boucler la boucle.

Merci à tous.



Résumé

La synthèse d'images est fondée depuis quelques années sur des bases physiques et mathématiques plus rigoureuses, notamment la modélisation des propriétés radiatives des surfaces et la simulation par éléments finis des transferts radiatifs entre ces surfaces. Une telle évolution permet d'envisager de nouvelles applications industrielles, en particulier en ingénierie de l'éclairage.

Ce mémoire présente une architecture logicielle à base d'objets, conçue pour soutenir la recherche dans les disciplines que nous venons d'évoquer et faciliter le développement de prototypes industriels. Elle repose d'une part sur une bibliothèque de classes abstraites tirées de l'équation de transfert, d'autre part sur un système de fichiers et de programmes calqués sur les étapes et les tâches d'une simulation.

À travers différents projets de recherche et plusieurs applications, nous montrons comment notre architecture a été mise en œuvre avec succès, en particulier pour évaluer de nouveaux modèles d'émission ou de réflexion de la lumière. Nous discutons également ses lacunes dans la représentation des algorithmes et la mise en œuvre du paradigme objet.

Mots-clés: illumination globale, radiosité, synthèse d'images, architecture logicielle

Abstract

For a few years, research in image synthesis is built on more formal physical and mathematical foundations. Our own team focus on the modeling of radiative properties of surfaces and on the simulation of radiative transfers between those surfaces, through the use of finite elements. This evolution opens some new application fields, especially in lighting engineering.

The present report exposes an object oriented framework, designed to facilitate research in the above fields and the production of software prototypes. The framework includes a library of abstract classes, deduced from the transfer equation. It also relies on a system of files and programs modelled from the tasks and steps of a simulation.

Through various research projects and several applications, we demonstrate the qualities of the framework, especially in the evaluation of new models of light emission and reflexion. Finally, we examine the defects in the representation of algorithms and in the use of the object paradigm.

Keywords: global illumination, radiosity, image synthesis, framework

Table des matières

Introduction

L'objet de cette thèse était de concevoir et développer une structure logicielle pour la génération d'images de synthèse réalistes, en poursuivant un double objectif :

- faciliter l'implémentation des modèles et algorithmes conçus dans notre équipe de recherche, afin de réduire le temps que les chercheurs sont obligés de consacrer au développement logiciel,
- faciliter le transfert de méthodes relativement générales vers des applications industrielles spécifiques.

Ce mémoire de thèse fait le bilan de trois ans de recherche dans cette perspective. En guise d'introduction, nous voulons montrer quelles sont les idées fortes qui ont jalonné notre parcours, et comment nos motivations et notre problématique scientifique ont évolué pendant ces trois ans. Nous allons aborder successivement :

- le *contexte scientifique initial* dans lequel étaient développées les recherches en synthèse d'images lorque nous avons commencé notre travail,
- le *banc d'essai de Cornell*, un système logiciel procédural inscrit dans ce contexte,
- le potentiel que recèle l'*orientation objets*, par opposition à l'approche logicielle procédurale,
- le *choix scientifique de notre équipe*, de fonder la génération d'images sur la simulation la plus précise possible du comportement et de la propagation de la lumière,
- la *plate-forme Graph'IS*, notre premier système à objets pour la simulation des transferts radiatifs,
- le *contexte scientifique actuel*, caractérisé par l'utilisation de méthodes stochastiques et de méthodes par éléments finis,
- l'*architecture VISION* de Philipp Slusallek, orientée objets et conforme au nouveau contexte,
- de *nouvelles propositions*, fruits de l'expérience acquise avec la plate-forme Graph'IS et de sa confrontation avec VISION.

Ce survol chronologique éclairera le lecteur sur notre démarche. La suite du mémoire est organisée de façon thématique. Nous en présenterons la structure générale.

Contexte scientifique initial

Pour calculer l'image d'une scène, il faut d'abord modéliser les surfaces qui la composent, puis simuler la propagation de la lumière depuis les sources lumineuses jusqu'à un observateur virtuel. Les programmes les plus simples ne prennent en compte que l'éclairage direct des surfaces par les sources : ils n'évaluent que l'*illumination locale*. Pour obtenir des images plus réalistes il est nécessaire de simuler également l'éclairage indirect, c'est-à-dire les réflexions lumineuses entre les surfaces, ce qui permet d'évaluer leur *illumination globale*. Il y a quelques années, on classait la plupart des méthodes d'illumination globale par rapport à deux algorithmes de références : le *lancer de rayons* [Whi80] et la *radiosité* [GTGB84].

Le lancer de rayons consiste à parcourir les rayons lumineux dans le sens inverse de la lumière, en partant de l'œil de l'observateur virtuel. À chaque fois que l'un de ces rayons intersecte une surface de la scène, on calcule l'éclairage direct dû aux sources lumineuses, et on évalue l'éclairage indirect en émettant de nouveaux rayons dans les directions les plus importantes, notamment dans la direction symétrique si la surface est très *spéculaire* (par exemple un miroir).

La radiosité est inspirée de travaux sur les échanges radiatifs de chaleur. Elle consiste d'abord à subdiviser la scène en éléments de surfaces. Si on considère que chaque élément est *diffus*, c'est à dire qu'il réfléchit uniformément la lumière, et qu'il possède une illumination constante sur toute sa surface, alors on peut construire et résoudre un système linéaire afin d'évaluer l'illumination des éléments. Pour visualiser la scène, il suffit alors de projeter ces éléments sur le plan de l'image.

On admet généralement que la radiosité respecte mieux les lois physiques de propagation de l'énergie lumineuse [CW93]. De plus, l'essentiel des calculs réside dans la résolution du système linéaire, et les résultats peuvent être exploités pour produire rapidement de nombreuses images pour des points de vues variés. Cette propriété est appelée *indépendance du point de vue*. L'atout du lancer de rayons est plutôt dans la diversité des effets qui peuvent être simulés, dont la réflexion spéculaire et les textures. Face à ce constat, les chercheurs se sont ingéniés à produire des extensions qui comblent les lacunes de l'une ou l'autre des approches, voire à construire des algorithmes mixtes qui alternent plusieurs passes de radiosité et de lancer de rayons [WEH89].

Système procédural typique

Plusieurs systèmes logiciels, plus ou moins modulaires, ont été proposés pour supporter les méthodes ci-dessus. Le banc d'essai de l'université Cornell nous a servi de repère principal [TLG91].

La figure 0.1 présente le fonctionnement global du banc d'essai, lequel est délimité dans la figure par les pointillés. Les données d'entrée sont fournies par des modeleurs géométriques externes. Le format neutre MID sert de passerelle entre ces modeleurs et les programmes de rendu. Les principaux programmes implémentés sont les classiques évoqués ci-dessus : un lancer de rayons, une méthode de radiosité et un algorithme à deux passes. Ces programmes sont implémentés à l'aide d'une bibliothèque de modules, dont les principaux représentants sont donnés dans la figure 0.2.

La conception du banc d'essai et son implémentation en langage C sont foncièrement procédurales. La couche consacrée aux primitives est à ce titre très caractéristique : au lieu de créer un module par type de primitive (sphère, cube, etc) les auteurs ont préféré créer un module par fonction (approximation polygonale, volume englobant, intersection avec un rayon, etc). Malgré la qualité et la flexibilité indiscutable du banc d'essai, nous avons rejeté l'idée de l'utiliser ou de

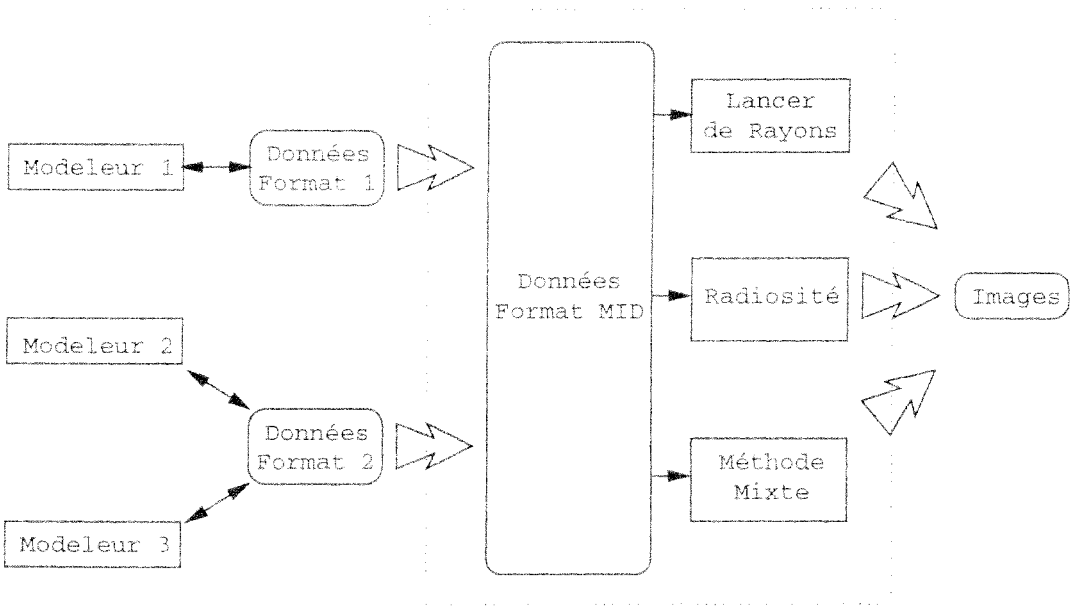


FIG. 0.1 – *Fonctionnement global du banc d'essai de Cornell*

développer par nous-même un système similaire. Ceci pour deux raisons essentielles :

- La décomposition des modules est directement calquée sur les étapes des algorithmes de référence, comme en témoigne certains noms de modules : `Optimisation Rayons`, `Affichage Radiosité`, etc. Cela les rend difficile à réutiliser dans des algorithmes foncièrement différents.
- Pour adapter un module à ses propres besoins sans pertueber les autres utilisateurs, il faut nécessairement en faire une version personnelle et recompiler tout le banc d'essai dans son propre espace de travail. On perd ainsi le bénéfice des améliorations qui pourraient être apportées ultérieurement au module central.

Avantages de l'orientation objets

Nous avons donc pris le parti d'utiliser une conception et une programmation par objets, réputées produire des programmes plus réutilisables (voir chapitre 2). Si l'on s'en tient aux langages de classes, le paradigme objet peut être réduit à quelques principes :

- *Classes et objets* : un *objet* est un ensemble de données et d'opérations pour les manipuler ; les objets qui possèdent la même structure de données peuvent partager les mêmes opérations ; ces traits communs sont définis par une *classe*, dont les objets sont les instances.
- *Héritage* : une nouvelle classe (*sous-classe*) peut être définie comme l'extension ou la redéfinition partielle d'une classe existante (*superclasse*).
- *Polymorphisme et liaison dynamique* : si elle est *polymorphique*, une variable peut référencer indifféremment les instances de plusieurs classes distinctes ; les appels d'opérations addres-

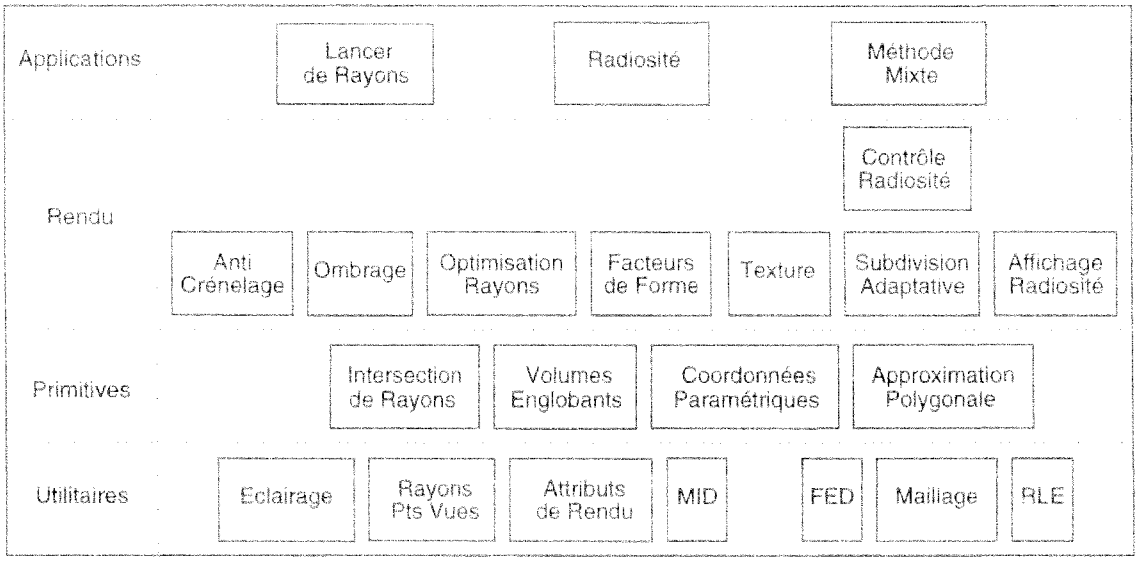


FIG. 0.2 – *Modules du banc d'essai de Cornell*

sés à cette variable sont résolus lors du l'éxécution du programme (*liaison dynamique*), selon la classe effective de l'objet référencé.

L'héritage fournit l'adaptabilité dont manquent les bibliothèques de modules. Par ailleurs, la pratique de programmation la plus intéressante est sans doute celle qui consiste à déclarer une opération sans en fournir immédiatement l'implémentation. Une classe qui possède au moins une opération non implémentée est dite *abstraite*. Ces classes incomplètes permettent notamment de spécifier des *interfaces standards*.

Une bibliothèque de classes riche en classes abstraites est qualifiée d'*architecture générique* (*framework*). C'est un système de ce genre que nous avons conçu et que nous présenterons dans ce mémoire.

Équation de transfert

Les premiers travaux de recherche sur le rendu d'images réalistes étaient surtout centrés sur la modélisation géométrique des éléments d'une scène et les calculs permettant de déterminer leur visibilité depuis une ou plusieurs sources lumineuses, et depuis un observateur virtuel donné. L'aspect de ces éléments était obtenu grâce à un modèle physique très simplificateur.

Une étape importante fut franchie quand Kajiya [Kaj86] formula l'équation de transfert, qui résume les échanges d'énergie lumineuse au voisinage d'une surface. Cette équation peut s'exprimer ainsi (voir chapitre 1) :

$$r(x,\vec{\omega}) = r_0(x,\vec{\omega}) + \int_{\Omega} \rho(x,\vec{\omega}',\vec{\omega}) r(x'(x,\vec{\omega}'),\vec{\omega}') cos\theta' d\vec{\omega}' \qquad (0.1)$$

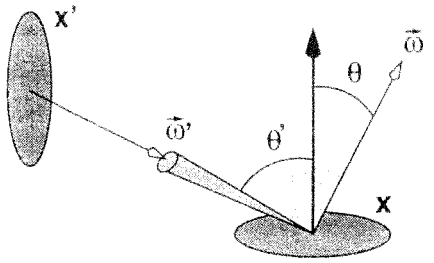


FIG. 0.3 – *Paramètres géométriques de l'équation de transfert*

avec (figure 0.3) :

- S : lieu formé par les surfaces de la scène,
- Ω : ensemble des directions,
- Λ : ensemble des spectres du domaine visible,
- $r(S \times \Omega) \mapsto \Lambda$: radiance totale,
- $r_0(S \times \Omega) \mapsto \Lambda$: radiance émise,
- $\rho(S \times \Omega^2) \mapsto \Lambda$: réflectance,
- $x'(S \times \Omega) \mapsto S$: visibilité.

L'équation de transfert a marqué un tournant important dans le domaine de la synthèse d'images. Les images ne sont plus que le sous-produit final d'une simulation physique du comportement et de la propagation de la lumière dans l'espace géométrique considéré.

Plate-forme Graph'IS

Les lacunes des systèmes existants nous ont amené à concevoir notre propre système, nommé *plate-forme Graph'IS*. Ses spécificités devaient être les suivantes :

- permettre de simuler le comportement et la propagation de la lumière à partir de modèles physiques les plus précis possibles,
- s'appuyer sur les principes de l'orientation objets,
- faciliter l'implémentation et l'expérimentation de nouveaux modèles et algorithmes de simulation,
- faciliter l'implémentation et l'expérimentation de prototypes industriels.

Pour identifier les classes principales de notre domaine et assurer une validité physique aux simulations mises en œuvre dans la plate-forme, nous sommes partis de l'équation de transfert, ce qui nous a amené à retenir les classes illustrées dans la figure 0.4 :

- `Scene` : simulation globale.

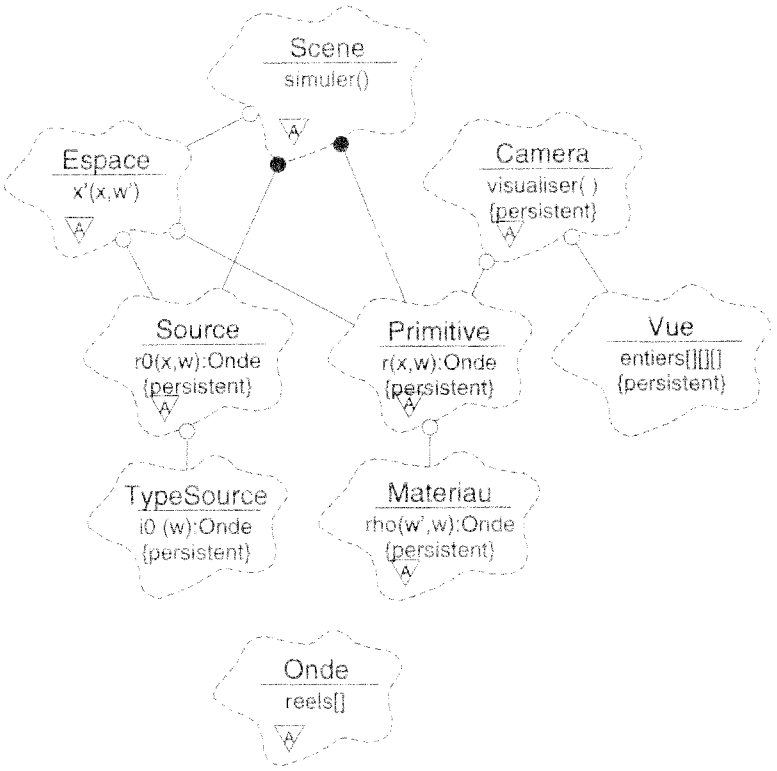


FIG. 0.4 – *Classes principales de la plate-forme Graph'IS*

- `Espace` : calcul de la visibilité x'.
- `Primitive` : primitives géométriques supportant les fonctions r et ρ.
- `Source` : sources lumineuses supportant la fonction r_0.
- `TypeSource` : modèles d'émission $i_0(\Omega) \mapsto \Lambda$
- `Materiau` : modèles de réflexion $\rho(\Omega^2) \mapsto \Lambda$.
- `Onde` : grandeurs spectrales $\in \Lambda$.
- `Camera & Vue` : observateurs virtuels et images.

Sur la base des applications que nous avons réalisées pour des architectes et des éclairagistes, nous avons également spécifié un ensemble de formats de fichiers et de programmes. La figure 0.5 illustre le fonctionnement de la plate-forme Graph'IS. Les programmes et les sous-ensembles de données sont les suivants :

- `ModPhoto` permet d'étudier le comportement physique local des matériaux et des sources. Il intègre dans une *bibliothèque photométrique* les mesures issues des laboratoires spécialisés.
- `ModGeo` récupère et nettoie les *descriptions géométriques* fournies par les modeleurs du commerce.

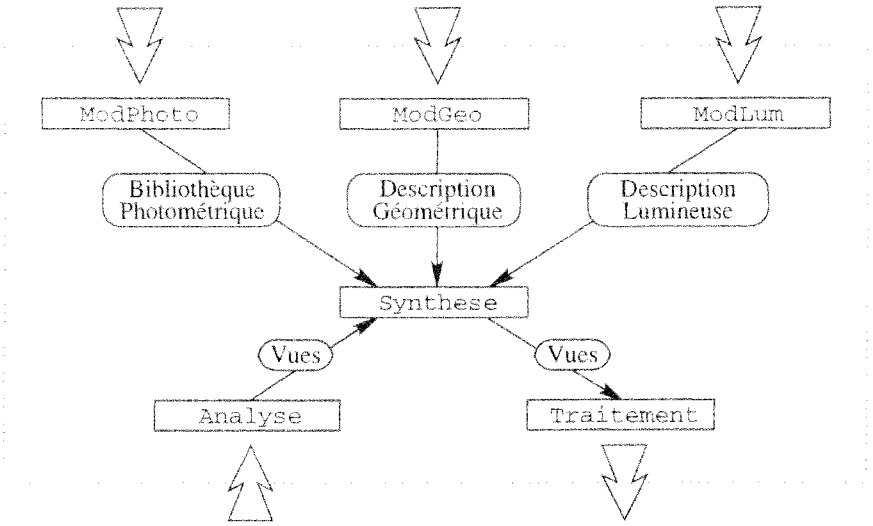


FIG. 0.5 – *Fonctionnement global de la plate-forme Graph'IS*

- **ModLum** permet de positionner et régler les sources d'une scène, que l'on regroupe dans une *description lumineuse.*

- **Synthese** est le programme de simulation proprement dit. Il produit des *vues* (combinaisons linéaires d'images).

- **Analyse** retrouve le point de vue d'une photographie pour le transmettre au programme de simulation.

- **Traitement** recombine les images d'une *vue* et sait les incruster dans une photographie.

L'ensemble de la plate-forme a été développé collectivement, en langage C++. Pour amorcer l'implémentation, chaque chercheur de l'équipe à contribué en développant quelques classes concrètes standard. Les classes abstraites et utilitaires ont été développées par moi-même et par les ingénieurs qui nous ont accompagné au gré des transferts industriels. Nous avons notamment mis en place des classes permettant la transformation d'un objet en *représentation ASCII*, plus simple à sauvegarder et à éditer.

Si l'on reprend aujourd'hui les objectifs que nous avions fixés, on peut raisonnablement considérer qu'ils ont été atteints. En effet :

- En nous appuyant sur l'équation de Kajiya, nous avons implanté des algorithmes fidèles aux lois de propagation de la lumière.

- Nous avons utilisé l'approche objets, y compris en ce qui concerne les formats d'échanges de données.

- Les chercheurs de l'équipe ont largement utilisé la plate-forme pour expérimenter leurs travaux [DMP94, DMCP94, CCDP94, FPPS94, San94, BPP95, CBP95, PDW95, Dev96, Che96, Fas96, SC96].

- Plusieurs prototypes logiciels ont été développés pour l'ingéniérie de l'éclairage, dont *Phostere*, utilisé à EDF pour simuler des projets d'illuminations [DCF95].

S'il est un point moins satisfaisant, c'est la diversité des algorithmes de calcul de propagation que nous avons intégrés dans la plate-forme. La plupart sont des algorithmes connus [CCWG88, Che90] que nous avons améliorés.

Par ailleurs, nous avons suffisamment progressé dans la maîtrise de la programmation objet pour comprendre que nous étions parfois trop extrémistes. A vouloir se différencier systématiquement de l'approche procédurale, nous sommes souvent tombés dans une conception statique, trop centrée sur les données et sur la hiérarchie des classes.

Nouveau contexte scientifique

En même temps que nous terminions l'expérimentation de la plate-forme Graph'IS, notre problématique scientifique évoluait. L'ancienne opposition entre méthodes de radiosité et méthodes de lancer de rayons avait laissé la place à une opposition entre méthodes stochastiques et méthodes à éléments finis ; une évolution qui s'inscrit dans un mouvement général vers une plus grande rigueur mathématique.

Nous avons nous-même réexprimé la simulation des transferts radiatifs selon la problématique plus générale des éléments finis. Voyons brièvement comment l'équation de transfert (0.1) peut être transformée en système linéaire de n équations à n inconnues. L'équation de départ peut d'abord être reformulée à l'aide d'opérateurs linéaires, ce qui permet d'obtenir une notation plus concise, et met en valeur la relation linéaire entre la radiance propre r_0 et la radiance totale r :

$$Mr = r_0 \tag{0.2}$$

avec

- $M \equiv I - KG$,
- I : opérateur identité,
- $(Kh)(x,\vec{\omega}) \equiv \int_\Omega \rho(x,\vec{\omega}',\vec{\omega}) h(x,\vec{\omega}') d\mu(\vec{\omega}')$ avec $d\mu(\vec{\omega}') \equiv cos\theta' d\vec{\omega}'$
- $(Gh)(x,\vec{\omega}) \equiv h(x'(x,\vec{\omega}),\vec{\omega})$

Pour ramener ce problème à une dimension finie, on définit un opérateur de projection, lequel permettra de transformer l'équation (0.2) en système linéaire. La projection repose sur :

- *Une base de fonctions* : on maille le support géométrique en éléments de surface finis, et on choisit un ensemble de fonctions $\{u_1, \ldots, u_n\}$, telle que la solution r puisse être approchée par :

$$r_n = \sum_{j=1}^{n} \alpha_j u_j \tag{0.3}$$

– *Des fonctionnelles linéaires* : afin d'ajouter n contraintes linéaires, on choisit un ensemble de fonctionnelles $\phi_{1\leq i\leq n}$ et on impose :

$$\phi_{1\leq i\leq n}(e_n) = \phi_i(Mr_n - r_0) = 0 \tag{0.4}$$

En combinant les formules (0.3) et (0.4), et en exploitant la linéarité de ϕ_i et de M, nous pouvons finalement exprimer notre problème sous forme matricielle :

$$\begin{bmatrix} \phi_1 Mu_1 & \phi_1 Mu_2 & \cdots & \phi_1 Mu_n \\ \phi_2 Mu_1 & \phi_2 Mu_2 & \cdots & \phi_2 Mu_n \\ \vdots & \vdots & \ddots & \vdots \\ \phi_n Mu_1 & \phi_n Mu_2 & \cdots & \phi_n Mu_n \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} \phi_1 r_0 \\ \phi_2 r_0 \\ \vdots \\ \phi_n r_0 \end{bmatrix} \tag{0.5}$$

La plupart des algorithmes publiés dans la littérature récente sont des cas particuliers de la formule (0.5). Chaque algorithme est caractérisé par la base de fonctions qu'il utilise, les fonctionnelles linéaires associées, le mode de calcul des cœfficients de la matrice, et la méthode de résolution du système.

Architecture VISION

Philipp Slusallek a adopté très tôt, au sein de son architecture VISION [Slu95], la distinction que l'on fait aujourd'hui entre les méthodes stochastiques et les méthodes par éléments finis. Son travail est proche du nôtre, puisqu'il cherche à concevoir une architecture à base d'objets pour la simulation de l'illumination globale dans un environnement 3D. Nous allons faire quelques comparaisons entre ses travaux et les nôtres.

Les classes principales de VISION sont présentées dans la figure 0.6 . En mettant de coté la présence de classes pour gérer les volumes participatifs (`EclairageVolumique`, `Echantillonneur`, `Transmetteur`), on peut faire les remarques suivantes :

– Comme dans la plate-forme Graph'IS, les auteurs sont partis de l'équation de Kajiya. Il fait également la distinction entre des objets représentant les caractéristiques locales des primitives (`Reflecteur`, `SourceLumineuse`, `Geometrie`) et un objet global chargé de calculer les transferts lumineux (`Eclairage`).

– Les différents aspects des primitives de la scène (géométrie, réflectance, radiance) sont dispersés sur différents objets, et c'est en fait la description de scène (`Divers`) qui fait le lien entre ces aspects. Cela vaut aussi pour les résultats de la simulation, qui sont stockés séparement de la géométrie.

– Malgré une description assez précise des méthodes par éléments finis dans la publication, on ne trouve pas de proposition de classes pour faciliter leur implémentation. Comme dans la plate-forme Graph'IS, on peut craindre que la classe globale (`Eclairage`) ne soit trop volumineuse.

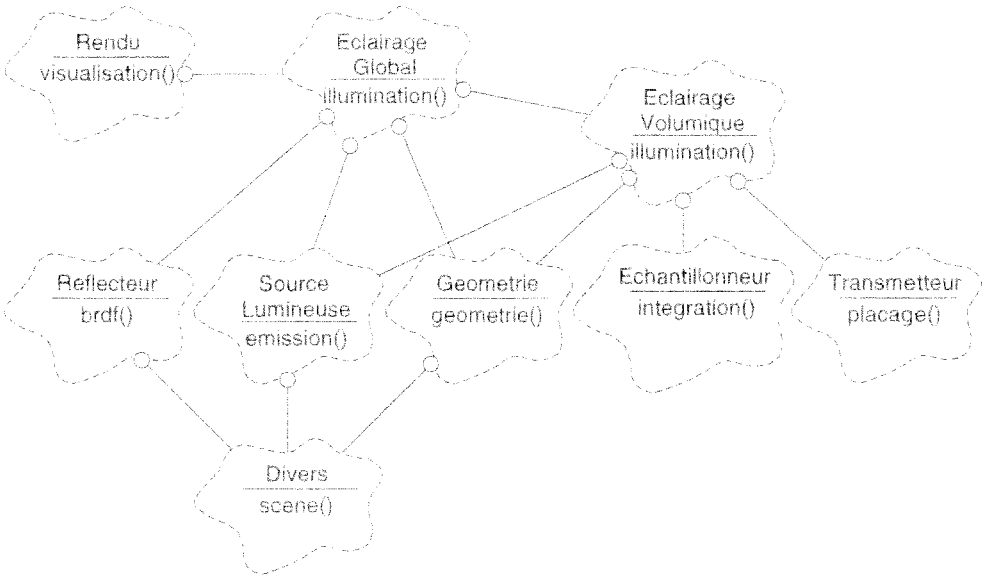


FIG. 0.6 – *Classes principales de l'architecture VISION*

– Le calcul de l'illumination par `Eclairage`, bien qu'indépendant du point de vue, est piloté ici par la visualisation (`Rendu`). En fait, la simulation est intégrée comme un précalcul, et les appels de la classe `Rendu` à la classe `Eclairage` ne font que déclencher la reconstruction de la solution au point voulu.

L'importation des données d'entrée de l'architecture VISION est basée sur l'interface RENDERMAN, déjà très utilisée dans la communauté de l'informatique graphique. Les auteurs proposent une extension de cette interface pour mieux supporter les algorithmes d'illumination globale [SPS95]. Le résultat est probablement l'interface la plus complète qui existe pour l'échange de données entre modeleurs et programmes de rendu. RENDERMAN semble cependant difficile à marier avec un mécanisme de persistance, comme on souhaiterait le faire dans une architecture à base d'objets.

En ce qui concerne l'implémentation de VISION, c'est également le langage C++ qui a été retenu. Pour faciliter la sélection des classes et piloter l'éxécution du système, les auteurs utilisent une extension objet du langage TCL [McL93, HSS94], ce qui permet de disposer d'un C++ presque interprété. En l'absence d'un langage de programmation répandu qui soit à la fois compilable et interprétable, le couple C++/TCL constitue un compromis intéressant.

Les auteurs de VISION ont prouvé la flexibilité de cette architecture en implantant un large éventail d'algorithmes d'illumination globale, aussi bien avec des méthodes par éléments finis qu'avec des méthodes stochastiques. Dans la plate-forme Graph'IS, nous avions au contraire mis l'accent sur les modèles de réflexion, de sources lumineuses, de colorimétrie, ou encore sur la visualisation d'images incrustées et de séquences d'images. Pourtant, nous avons souvent abouti à des classes similaires.

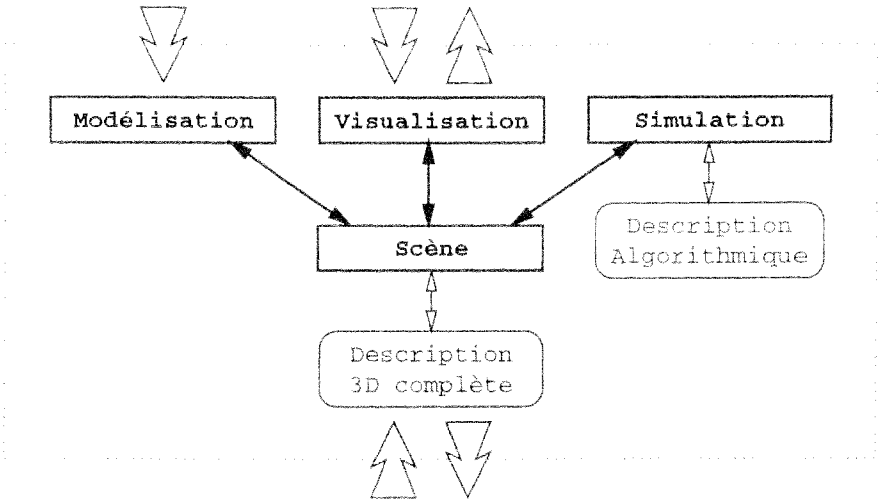


FIG. 0.7 – *Fonctionnement global de la nouvelle architecture*

Nouvelle architecture proposée

Le déplacement des centres d'intérêts de notre équipe (de la conception de modèles physiques à la conception d'algorithmes de calcul), les progrès du matériel graphique, ou encore le potentiel du calcul parallèle et les défauts que nous avons identifiés dans la plate-forme Graph'IS nous ont finalement amené à imaginer une nouvelle architecture.

Lors de l'exploitation de *Phostere*, logiciel applicatif issu de l'architecture Graph'IS, nous avons mesuré à quel point les systèmes logiciels actuels (y compris Graph'IS et VISION) ne sont pas faits pour s'insérer dans un environnement logiciel plus large, incluant par exemple des fonctionnalités de réalité virtuelle ou de CAO. Le cœur du problème est dans l'enchevêtrement des calculs de simulation et de visualisation, et dans la séparation de la description géométrique et de l'illumination qui est calculée. C'est pourquoi nous proposons aujourd'hui une architecture logicielle basée sur quatre composants, illustrés dans la figure 0.7. Leurs rôles seraient les suivants :

- `Scene` : gestion centralisée de la description de la scène, comprenant la géométrie mais également l'illumination globale déjà évaluée.
- `Modelisation` : définition ou modification des caractéristiques géométriques et photométriques des primitives et des sources lumineuses composant la scène.
- `Simulation` : évaluation de l'illumination globale de la scène, conformément à une *description algorithmique* donnée.
- `Visualisation` : production des images, par simple projection ou par lancer de rayons, en temps réel ou au prix d'un calcul très précis.

Pour implémenter cette nouvelle architecture, nous préconisons la boîte à outils *Open Inventor*, qui est conçue pour les applications graphiques 3D interactives. Les caractéristiques et la diffusion de ce produit en font un candidat tout à fait adapté à nos besoins, en particulier à l'implémentation du composant `Scene`.

L'architecture *Candela*, en cours de développement et d'expérimentation au sein de l'équipe, reprend une partie des propositions ci-dessus. Elle est décrite dans le mémoire de thèse de Slimane Merzouk [Mer97].

Contenu du mémoire

Notre mémoire se poursuit par un exposé sur les aspects physiques, mathématiques et algorithmiques de la *simulation des transferts radiatifs* (chapitre 1). Nous parlerons uniquement des méthodes de simulation étudiées dans notre équipe, à savoir celles qui sont à base d'éléments finis.

Dans un deuxième temps nous aborderons le *paradigme objet* (chapitre 2). Nous présenterons la terminologie et la notation utilisée dans le mémoire, puis nous discuterons de la réutilisation logicielle et du concept d'*architecture générique*.

Viendra ensuite un état de l'art des *systèmes logiciels* déjà proposés pour la simulation de l'illumination globale (chapitre 3). Nous y revenons un peu plus en détail sur le banc d'essai de Cornell et sur l'architecture VISION.

Après ces préliminaires indispensables, nous présenterons le système logiciel que nous avons conçu et expérimenté : la *plate-forme Graph'IS* (chapitre 4). Nous montrerons dans quelle mesure les objectifs fixés ont été atteints, d'un point de vue scientifique et industriel.

Enfin, la dernière partie du mémoire sera consacrée à l'évaluation critique de la plate-forme et aux nouvelles propositions que nous avons faites sur la base de cette analyse (chapitre 5).

Chapitre 1

Simulation des transferts radiatifs

Nous allons présenter ci-dessous l'approche selon laquelle notre équipe de recherche aborde la *synthèse d'images*. Cette approche est fondée sur la simulation des échanges d'énergie lumineuse entre les surfaces qui composent la scène à visualiser. Parce que les méthodes que nous utilisons sont souvent issues de travaux plus anciens en *thermodynamique* [SH81], et parce que ces méthodes peuvent être appliquées en dehors du seul spectre visible, nous définissons cette approche « *simulation des transferts radiatifs* ».

Après avoir présenté les lois qui régissent le comportement de la lumière (1), nous construisons un système linéaire à base d'*éléments finis* (2), dont la résolution nous permet de connaître la distribution de l'énergie lumineuse dans la scène (3). La dernière section concerne l'exploitation de ces résultats pour produire des images (4).

1 Physique de la lumière

A l'échelle macroscopique où nous opérons, les effets d'interférence et de diffraction lumineuse sont anecdotiques, et nous pouvons nous cantonner au formalisme de l'optique géométrique. Nous pouvons également postuler que chaque longueur d'onde est le siège de phénomènes lumineux indépendants de ceux des longueurs d'onde voisines.

L'environnement géométrique dans lequel nous voulons simuler ces phénomènes est une scène modélisée sous forme de surfaces. Le milieu dans lequel se propage la lumière est supposé vide. Sous ces hypothèses, l'équation de transfert proposée par Kajiya [Kaj86] exprime à la fois les échanges entre surfaces et les phénomènes locaux sur chacune de ces dernières. Nous allons la reformuler ci-dessous.

Dans un souci de simplification, nous ne considèrerons que les surfaces opaques.

Grandeurs physiques

L'énergie qui est à l'origine des phénomènes lumineux se propage par radiations sous forme d'ondes électromagnétiques. Le flux radiant Φ s'exprime à partir de l'énergie Q par :

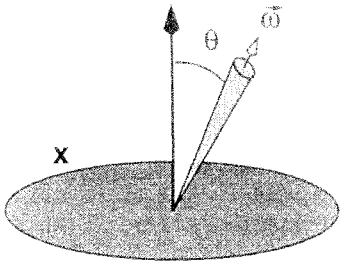


FIG. 1.1 – *Paramètres géométriques de la radiance*

$$\Phi = \frac{dQ}{dt} \tag{1.1}$$

On appelle *radiance* r le flux émis au point x dans la direction ω, par unité d'angle solide et par unité d'aire projetée :

$$r(x, \vec{\omega}) = \frac{d^2\Phi}{d\vec{\omega}\, cos\theta\, dx} \tag{1.2}$$

Le terme *radiance*, lorsqu'il est utilisé seul, fait référence à l'énergie qui quitte la surface au point x. On parlera de *radiance incidente* r_i si on applique la formule (1.2) au flux incident. On peut aussi décomposer la radiance totale r en *radiance émise* r_0, dont la source est la surface elle-même, et en *radiance réfléchie* r_r, qui est issue de la réflexion du flux incident.

On appelle *irradiance* i le flux incident par unité d'aire au un point x. On peut exprimer l'irradiance i en fonction de la radiance incidente r_i :

$$i(x) = \frac{d\Phi}{dx} = \int_\Omega cos\theta \frac{d^2\Phi}{d\vec{\omega}\, cos\theta\, dx} d\vec{\omega} = \int_\Omega cos\theta\, r_i(x, \vec{\omega})\, d\vec{\omega} \tag{1.3}$$

Si la surface n'est pas une source d'énergie lumineuse, tout le flux radiant est issu de la réflexion du flux incident. On caractérise cette réflexion grâce à la *réflectance bidirectionnelle*, qui exprime la contribution de l'énergie reçue de la direction ω_i à l'énergie réfléchie dans la direction ω_r :

$$\rho(x, \vec{\omega_i}, \vec{\omega_r}) = \frac{dr_r(x, \vec{\omega_e})}{cos\theta_i\, r_i(x, \vec{\omega_i})\, d\vec{\omega_i}} \tag{1.4}$$

Les grandeurs ci-dessus ont été définies en radiométrie, et englobent toutes les longueurs d'onde. Elles ne conviennent pas aux simulations qui veulent préserver les couleurs perçues par l'œil humain. Dans ce qui va suivre, nous considérerons les distributions spectrales correspondant à ces différentes grandeurs.

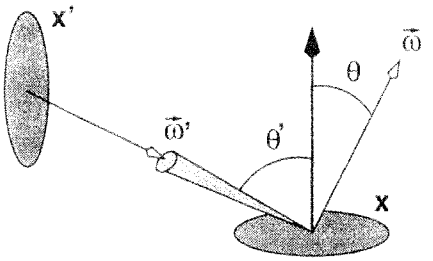


FIG. 1.2 – *Paramètres géométriques de l'équation de transfert*

Équation de transfert

L'équation de transfert a l'avantage de décrire en même temps les phénomènes locaux d'émission de lumière, de réflexion, d'absorption et la redistribution globale de l'énergie entre les surfaces. On utilise les espaces et les fonctions suivantes :

- S : lieu formé par les surfaces de la scène,
- Ω : ensemble des directions,
- Λ : ensemble des spectres du domaine visible,
- $r(S \times \Omega) \mapsto \Lambda$: radiance spectrale totale,
- $r_0(S \times \Omega) \mapsto \Lambda$: radiance spectrale émise,
- $\rho(S \times \Omega^2) \mapsto \Lambda$: réflectance spectrale.

La fonction r_0 correspond aux sources lumineuses de la scène. Les fonctions r_0 et ρ sont présumées connues, et on cherche à évaluer la fonction r. Si deux points x et x' se voient mutuellement, sous l'hypothèse d'un milieu vide, on peut écrire :

$$r(x,\vec{\omega}) = r_0(x,\vec{\omega}) + \int_{\Omega} \rho(x,\vec{\omega}',\vec{\omega}) r(x'(x,\vec{\omega}'),\vec{\omega}') cos\theta' d\vec{\omega}' \tag{1.5}$$

Dans cette équation, le point x' est par construction le point visible depuis x dans la direction $\vec{\omega}'$ (cf. figure 1.2).

2 Méthode de construction du système linéaire

Nous exposons ci-dessous un processus permettant de passer progressivement de l'équation de transfert (1.5) à un système linéaire de n équations à n inconnues, en se basant notamment sur la théorie des éléments finis.

Formulation à l'aide d'opérateurs linéaires

L'équation intégrale (1.5) appartient à la classe plus générale des équations d'opérateurs. Les équations d'opérateurs tendent a être plus concises que les équations intégrales équivalentes, tout en exprimant les propriétés algébriques essentielles. Ainsi, nous pouvons exprimer l'équation (1.5) sous la forme d'une équation d'opérateurs linéaires de seconde espèce :

$$r = r_0 + KGr \tag{1.6}$$

avec

- $(Kh)(x,\vec{\omega}) \equiv \int_\Omega \rho(x,\vec{\omega}',\vec{\omega})h(x,\vec{\omega}')d\mu(\vec{\omega}')$ avec $d\mu(\vec{\omega}') \equiv cos\theta' d\vec{\omega}'$
- $(Gh)(x,\vec{\omega}) \equiv h(x'(x,\vec{\omega}),\vec{\omega})$

Pour souligner la relation linéaire entre la radiance totale et la radiance propre, on peut exprimer (1.6) de façon encore plus compacte :

$$Mr = r_0 \tag{1.7}$$

avec

- $M \equiv I - KG$,
- I : opérateur identité.

Définition d'un opérateur de projection

Étant donnée une base de fonctions $\{u_1,\ldots,u_n\}$ définies dans l'espace χ. Si ces fonctions sont correctement choisies, on peut approximer χ par l'ensemble de leurs combinaisons linéaires. Nous appelerons χ_n l'espace approximé. Dans cet espace de dimension finie n, l'approximation r_n de r est déterminée par les cœfficients $\{\alpha_1,\ldots,\alpha_n\}$:

$$r_n = \sum_{j=1}^{n} \alpha_j u_j \tag{1.8}$$

Maintenant que nous avons un nombre fini d'inconnues $\{\alpha_1,\ldots,\alpha_n\}$, nous devons construire n équations linéaires à partir de la relation (1.7). Pour y parvenir, il faut choisir un ensemble de fonctionnelles linéaires $\{\phi_1,\ldots,\phi_n\}$, définies sur $\chi \mapsto \Re$, et qui minimisent l'erreur résiduelle e_n :

$$\phi_{1\leq i\leq n}(e_n) = \phi_i(Mr_n - r_0) = 0 \tag{1.9}$$

Les fonctionnelles linéaires et la base de fonctions définissent ce que l'on appelle un *opérateur de projection*. Chaque opérateur correspond à une conception spécifique de l'approximation de χ, et conduit à une fonction r_n spécifique.

Formulation matricielle

En exploitant la linéarité des fonctionnelles et des opérateurs linéaires, nous pouvons finalement aboutir au système suivant :

$$\begin{bmatrix} \phi_1 Mu_1 & \phi_1 Mu_2 & \cdots & \phi_1 Mu_n \\ \phi_2 Mu_1 & \phi_2 Mu_2 & \cdots & \phi_2 Mu_n \\ \vdots & \vdots & \ddots & \vdots \\ \phi_n Mu_1 & \phi_n Mu_2 & \cdots & \phi_n Mu_n \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} \phi_1 r_0 \\ \phi_2 r_0 \\ \vdots \\ \phi_n r_0 \end{bmatrix} \tag{1.10}$$

La matrice mise en évidence dans (1.10) n'est pas semblable à celles qu'on manipule généralement en analyse numérique. Elle est d'abord de très grande taille : n est égal au nombre d'éléments de surfaces multiplié par le nombre de fonctions de base définies sur chaque élément. De plus, l'évaluation de chaque cœfficient de la matrice comprend notamment le calcul de la visibilité (fonction x'), qui est un calcul compliqué. La résolution du système ne peut donc pas être directe, quelle que soit la puissance informatique dont on dispose.

3 Algorithmes de simulation

De nombreux algorithmes publiés dans la littérature récente sont des cas particuliers de la démarche présentée dans la section 2. Chacun d'entre eux est caractérisé par ses choix en ce qui concerne :

- la base de fonctions $\{u_1, \ldots, u_n\}$,
- les fonctionnelles linéaires $\{\phi_1, \ldots, \phi_n\}$,
- la méthode de calcul des cœfficients $\phi_i Mu_j$ du système linéaire (1.10),
- la méthode de résolution du système linéaire.

Chaque algorithme est un compromis plus ou moins réussi entre la précision du modèle physique sous-jacent, la précision des calculs de visibilité, et la rapidité de la résolution. Nous allons survoler quelques techniques typiques pour illustrer la diversité des solutions proposées.

Base de fonctions

Dans de nombreuses méthodes, en particulier les plus anciennes, on découpe uniformément les surfaces et on dote chaque élément d'une fonction constante. Seule la composante diffuse de la réflectance est prise en compte, et on a l'habitude de remplacer la radiance par son intégrale sur Ω, la *radiosité* [GTGB84, CG85]. Pour produire des images suffisamment réalistes à partir d'une telle base de fonctions, on procède souvent à un lissage lors de la visualisation finale, ce qui améliore le réalisme mais pas la précision intrinsèque de la simulation.

Pour une dimension n donnée, on peut aussi choisir d'utiliser des éléments de surfaces moins nombreux, mais dotés d'une base de fonctions plus élaborée (polynomiales, ondelettes [GSCH93]).

Quel que soit le type de fonctions que l'on utilise, on peut améliorer sensiblement la simulation en maillant les surfaces de façon non uniforme. On peut raffiner le maillage pendant la simulation, en s'adaptant progressivement à la répartition de l'énergie lumineuse [CCWG88], ou découper les éléments a priori et sur des critères purement géométriques, notamment avec un maillage hiérarchique [HSA91].

Fonctionnelles linéaires

La *collocation* a été largement utilisée en raison de sa simplicité. Elle consiste à choisir des points $\{x_1, \ldots, x_n\}$ tels que $det[u_j(x_i)] \neq 0$, et à définir les fonctionnelles par :

$$\phi_{1 \leq i \leq n} h \equiv h(x_i), \tag{1.11}$$

Le cœfficient ij du système linéaire a alors la forme suivante :

$$u_j(x_i) - (KGu_j)(x_i). \tag{1.12}$$

Dans le cas d'une base de fonctions constantes définies sur des éléments de surfaces planaires, la matrice a une diagonale unitaire, et des facteurs de forme [point/surface] en dehors de la diagonale. Ceci a été très largement utilisé dans les formulations de la méthode de radiosité [CG85]. En général, les méthodes fondées sur un nombre fini d'interactions [point/surface] sont des méthodes de collocation.

Une seconde technique est la méthode de Galerkin. Elle définit les fonctionnelles selon la formule ci-dessous, où la notation $\langle . | . \rangle$ représente l'intégrale du produit de deux fonctions :

$$\phi_{1 \leq i \leq n} h \equiv \langle u_i | h \rangle \tag{1.13}$$

Le cœfficient ij du système linéaire a alors la forme suivante :

$$\langle u_i | u_j \rangle - \langle u_i | KGu_j \rangle \tag{1.14}$$

La méthode de Galerkin fut utilisée pour la première fois dans [GTGB84] avec un maillage uniforme et une base de fonctions constantes. Dans une telle configuration, le second produit scalaire de l'équation (1.14) équivaut à un facteur de forme [surface/surface]. L'utilisation de fonctions d'ordre plus élevé a été explorée par [Zat93].

Méthode de calcul des cœfficients

L'opération la plus coûteuse de la simulation reste sans conteste l'évaluation des cœfficients de la matrice du système (1.10). La plupart du temps ils sont approximés. L'accélération des

calculs n'est pas la seule motivation de cette approximation : dans de nombreux cas, il n'existe pas de solution analytique exacte.

Le calcul d'un cœfficient peut englober l'évaluation d'une variété de facteur de forme, en particulier quand on utilise une base de fonctions constantes. De nombreuses techniques ont été proposées pour approximer le facteur de forme, dont l'hémi-cube [CG85] et le lancer de rayons [WEH89].

Lorsque l'on utilise une base de fonctions non constantes, on calcule un produit scalaire plutôt qu'un facteur de forme, ce qui implique des approximations différentes. Par exemple, avec l'approche de type Galerkin proposée dans [Zat93], chaque cœfficient comprend quatre intégrales, que l'on approche par des quadratures de Gauss. Les environnements non-diffus posent une difficulté similaire dans la mesure où le calcul des cœfficients requière la prise en compte des fonctions de réflectance dans les calculs d'intégration [PSV90].

Méthode de résolution du système linéaire

On peut reconnaître dans les méthodes de synthèse d'images diverses techniques d'analyse numérique, par exemple :

- l'élimination gaussienne [GTGB84],
- les méthodes itératives qui doivent converger complètement après un nombre fini d'itérations, comme Gauss-Seidel [CG85] ou Jacobi [HSA91],
- la relaxation de Southwell, qui fournit des solutions relativement satisfaisantes avant la convergence complète [CCWG88],
- certaines techniques de pré-conditionnement [LTG93] ou de structuration de la matrice en blocs [HSA91].

Quand la méthode doit converger complètement, on construit généralement la matrice à l'avance. Dans les autres cas, on préfère calculer les cœfficients au vol, ce qui doit permettre d'économiser le calcul de certains cœfficients inutiles.

Pour conclure ce survol des algorithmes de simulation, il nous semble instructif de récapituler toutes les sources potentielles d'erreur :

- la modélisation imparfaite des données d'entrée r_0 et ρ,
- la représentation surfacique initiale, éventuellement détériorée par une facettisation,
- l'approximation de χ par χ_n (opérateur de projection),
- l'approximation des cœfficients $\phi_i M u_j$,
- l'arrêt de la résolution avant la convergence.

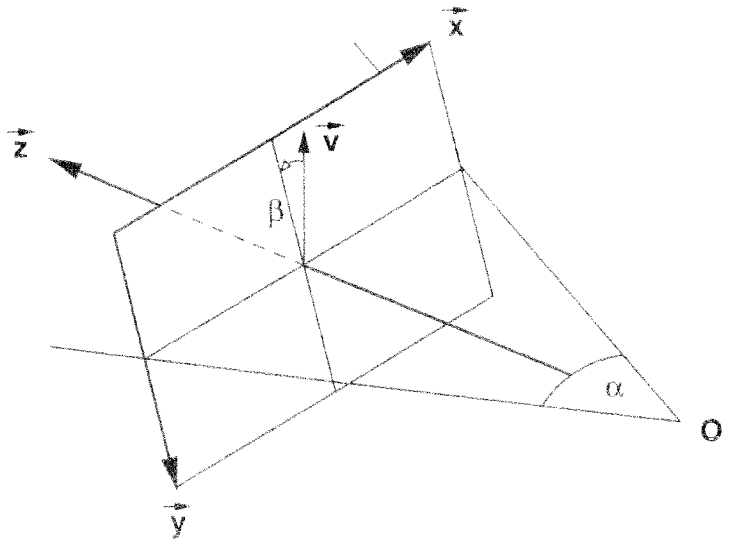


FIG. 1.3 – *Paramètres du point de vue*

4 Visualisation

Notre équipe de recherche a toujours privilégié les méthodes de synthèse d'images qui reposaient sur des résultats intermédiaires indépendants du point de vue, comme en témoigne les sections précédentes. L'objectif final reste tout de même de produire des images, et nous voulons terminer ce chapitre en évoquant brièvement les procédés de visualisation et les aspects colorimétriques.

Point de vue

On assimile la visualisation à la prise d'une photographie. Les propriétés optiques de l'appareil photographique virtuel sont le plus souvent modélisées comme une projection perspective parfaite, caractérisée par son origine O, son axe de vision $\vec{z}$, son angle d'ouverture horizontal α et son roulis β (rotation vis-à-vis de la verticale $\vec{v}$). Ces paramètres sont rappelés dans la figure 1.3, et constituent ce qu'on appelle le *point de vue*.

Avant de calculer une image, il faut choisir un point de vue, et choisir la résolution de l'image (nombre de pixels horizontaux et verticaux). On discrétise le plan de l'image en pixels, et l'objectif de la visualisation est de déterminer la radiance des surfaces perçues par le point O à travers chacun des pixels.

Z-buffer

La technique la plus courante consiste à projeter les surfaces les unes après les autres sur le plan de l'image, en stockant dans chaque pixel non seulement la radiance, mais également la coordonnée z du point projeté. Lorsque l'on veut stocker un nouveau point dans un pixel qui est déjà occupé, on doit d'abord vérifier que la coordonnée z du nouveau point est inférieur à celle de l'occupant. Cette technique à la fois simple et efficace est appelée *z-buffer* [FvDFH90].

Lancer de rayons

La visualisation par z-buffer se contente d'exploiter les informations fournies par une simulation préalable. Ces informations sont incomplètes. En particulier, elles ne prennent pas en compte la composante spéculaire parfaite de la réflectance ρ (effet de miroir). En effet, cette composante ne peut s'exprimer qu'à l'aide d'une distribution de Dirac, et il est impossible de la prendre en compte lors de l'évaluation des cœfficients du système linéaire.

Le *lancer de rayons* est une méthode qui intègre la composante spéculaire à ses calculs [Whi80]. Cela lui permet de produire des images assez spectaculaires, qui ont largement contribuées à son succès. Cet algorithme ne repose pas sur une simulation préalable. Il est capable de produire une image complète de façon autonome. Son principe est le suivant : on émet des rayons depuis O à travers tous les pixels, et à chaque intersection avec une surface, on ré-émet de nouveaux rayons dans les directions les plus importantes, en particulier vers les sources lumineuses et dans la direction de réflexion spéculaire. On applique ce principe de façon récursive, ce qui permet de construire pour chaque pixel l'arbre des rayons qui contribuent le plus à sa radiance. La radiance finale d'un pixel est calculée en parcourant son arbre et en calculant la contribution de chaque embranchement selon le modèle de reflexion associé à la surface.

L'algorithme prend en compte la réflectance spéculaire et calcule la fonction de visibilité x' de façon beaucoup plus précise que les algorithmes indépendant du point de vue. C'est particulièrement frappant en ce qui concerne les ombres projetées par les sources lumineuses primaires. Il souffre pourtant de deux inconvénients. D'une part, il est dépendant du point de vue : à chaque nouvelle image, on reprend tous les calculs de zéro. D'autre part, les inter-réflexions diffuses ne sont pas vraiment intégrées.

En fait, on peut réussir à simuler correctement toutes les composantes de la réflectance et à avoir des ombres projetées précises, en modifiant le lancer de rayons pour qu'il exploite les radiances évaluées dans une simulation préalable, tout en évaluant par lui-même les réflexions spéculaires et l'éclairage directe. On appelle ce type d'algorithme une *méthode à deux passes* [WEH89, SP89].

Colorimétrie

Lors de la présentation de l'équation de transfert (1.5), nous avions défini le résultat des fonctions r, r_0 et ρ comme des spectres. Nous avons ensuite occulté cet aspect afin de pas alourdir la présentation. Tout ce qui a été dit reste valable avec des grandeurs spectrales, à condition de considérer que les phénomènes lumineux se produisent indépendamment dans chaque longueur d'onde.

Dans la pratique, on sélectionne un sous-ensemble représentatif de longueurs d'onde de référence $\{\lambda_1, \ldots, \lambda_{n_\lambda}\}$, pour lesquelles tous les calculs sont menés de front. Les radiances et les réflectances manipulées sont des tableaux de n_λ réels associés aux différents λ de référence, et les additions ou les multiplications sont effectuées terme à terme entre les tableaux.

La base $\{\lambda_1, \ldots, \lambda_{n_\lambda}\}$ doit être choisie avant le début de la simulation, au même titre que l'opérateur de projection géométrique. Le nombre de longueurs d'onde n_λ a un impact très important sur la simulation, car les temps de calcul et la mémoire nécessaire lui sont directement

proportionnels. Plusieurs stratégies sont envisageables :

- Si l'on souhaite être extrêmement précis, on peut réaliser les calculs sur 80 longueurs d'onde uniformément réparties (les données d'entrées sont rarement plus précises) ;
- Gary Meyer a proposé une méthode permettant de sélectionner les longueurs d'onde les plus importantes vis-à-vis de la perception visuelle [Mey88b], en faisant l'hypothèse que les spectres sont continus ;
- Pascal Deville a proposé de prendre également en compte l'irrégularité possible des distributions spectrales et les raies d'émission en particulier [Dev96].

Le processus de visualisation que nous avons décrit précédemment produit un tableau de pixels dont on connaît la radiance spectrale sur les longueurs d'onde $\{\lambda_1, \ldots, \lambda_{n_\lambda}\}$. La dernière étape de la visualisation consiste, pour chaque pixel, à reconstruire le spectre complet et à le convertir dans l'espace colorimétrique RVB.

La plupart des systèmes de synthèse d'images effectuent encore tous leurs calculs dans l'espace RVB. Cette méthode n'est pas conforme à la réalité physique. Notamment, lors du produit d'un éclairement par une réflectance, ce qui devrait être l'intégrale d'un produit est remplacé par le produit de deux intégrales.

Chapitre 2

Orientation objets

Au début de notre travail, nous avons rapidement décidé d'adopter l'approche dénommée *orientation objets*, afin de faciliter la *réutilisation logicielle* au sein de notre équipe. La communauté de l'informatique graphique s'accorde aujourd'hui sur le progrès que constitue cette approche, même s'il est parfois conseillé de la compléter par d'autres techniques, notamment la programmation par contraintes [Wis96]. On pare l'orientation objets de nombreuses vertus : meilleure maîtrise des systèmes logiciels de grande taille, conception plus « naturelle », adaptabilité, maintenabilité, évolutivité... La pratique montre que l'on peut faire de mauvais programmes à objets, tout aussi facilement qu'avec les approches plus classiques, voire plus facilement du fait de concepts plus nombreux et plus complexes. La pratique montre aussi que les vertus sont réelles, si on consent un effort suffisant lors de la phase préliminaire de conception.

Le génie logiciel propose des méthodes pour mieux concevoir les systèmes logiciels : on développe un modèle du futur programme, que l'on documente à l'aide de diagrammes. Parmi les méthodes dédiées à l'orientation objets [JCJO93, WBWW90, CY91a, CY91b, RBP+91, Def92], notre préférence va à celle de Booch [Boo94] qui semble à la fois la plus souple et la mieux articulée avec le langage C++. Dans ce mémoire, nous utilisons la notation de Booch pour la plupart des figures illustrant un système logiciel. En ce qui concerne le langage de programmation proprement dit, nous nous sommes alignés sur le choix le plus courant, à savoir C++ [ES90, Lip91].

L'objectif de nos recherches était de construire un système logiciel *réutilisable* spécialement adapté à la simulation des transferts radiatifs. L'utilisation d'un langage à objets et d'une méthode de conception logicielle ne constituent qu'un premier pas dans cette direction. Pour aller plus loin, nous avons cherché des travaux de référence sur les architectures génériques (*frameworks*) [WBJ90] ou la programmation par composants logiciels [Cou96], sans réussir à en extraire une méthodologie claire et complète. Finalement, nous avons choisi de nous en tenir principalement au concept de *classe abstraite*.

Le chapitre présent comprendra d'abord deux sections de rappels sur les mécanismes des langages de classes (1), et la notation de Booch (2). L'ambition de ces rappels n'est pas de faire une présentation théorique rigoureuse, mais plutôt d'introduire la terminologie et les diagrammes que nous utiliserons tout au long du mémoire. Ensuite, nous aborderons la spécificité du langage C++ (3) et l'influence de ce choix sur nos travaux. Nous discuterons finalement des mécanismes et des pratiques qui peuvent favoriser la réutilisation logicielle (4).

1 Mécanismes des langages de classes

Plusieurs familles de langages permettent de mettre en œuvre le paradigme objet. Parmi eux, on trouve les *langages d'acteurs* et les *langages de représentation des connaissances centré objets* [MNC+91]. Ces langages sont spécialisés pour des catégories d'applications qui ne sont pas les nôtres, c'est pourquoi nous ne les avons pas étudiés plus avant.

La famille la plus utilisée est celle des *langages de classes*, parmi lesquels figurent *Smalltalk*, *CLOS*, *Eiffel* et *C++*. Les mécanismes que nous allons décrire ci-dessous concernent cette famille. Pour faciliter l'explication, nous nous appuierons sur les concepts des langages procéduraux (*Pascal*, *C*, *Fortran*), supposés connus. Pour une présentation plus formelle, nous renvoyons le lecteur à l'ouvrage de Grady Booch [Boo94].

1.1 Classes & objets

Dans un langage de programmation procédural, les principaux blocs de construction d'un programme sont les *procédures*. Ces procédures manipulent des *variables*, dont les valeurs appartiennent à des *types prédéfinis*. La première innovation des langages de classes est de permettre au programmeur de définir ses propres types, dénommés *classes*. Les éléments d'une classe sont appelés *objets* ou *instances*.

La façon la plus simple de construire une nouvelle classe est l'*agrégation*, qui consiste à assembler une structure de variables et des procédures pour manipuler ces variables. Chaque instance de la nouvelle classe possède son propre jeu de variables, nommées et organisées conformément à la structure définie par la classe. Lorsqu'on appelle l'une des procédures, on précise à quel objet on souhaite l'appliquer, et la procédure agit sur le jeu de variables de cet objet. Plus formellement, la description d'une classe comprend :

- La liste des variables représentant les données d'un objet, et leurs types. Les variables de cette sorte sont baptisées *attributs*. Leur valeur courante constitue l'*état* de l'objet.
- La description des procédures servant à construire, consulter, modifier et détruire les objets. Les procédures de cette sorte sont baptisées *méthodes*, et caractérisent le *comportement* de la classe et de ses instances.

Par ailleurs, en vertu du principe d'*encapsulation*, on compartimente les éléments d'une classe en une partie publique, l'*interface*, et une partie privée, l'*implémentation*. L'organisation la plus conforme à la philosophie objet consiste à ne garder dans l'interface de la classe que les interfaces des méthodes, c'est-à-dire leurs noms et la description de leurs arguments. Ainsi, les clients d'un objet ne peuvent agir sur ses données qu'indirectement, en invoquant les méthodes proposées dans l'interface.

Le programmeur est autorisé à se servir du même nom pour des méthodes différentes, à condition qu'elles appartiennent à des classes différentes.

1.2 Héritage

L'agrégation n'est pas le seul moyen de définir une nouvelle classe. On peut également avoir recours à l'*héritage*, qui consiste à étendre ou modifier une classe existante. Trois sortes de

manipulations sont possibles :

- ajout de nouveaux attributs,
- ajout de nouvelles méthodes,
- redéfinition de l'implémentation de certaines méthodes.

On dit que la nouvelle classe *hérite* de la classe d'origine. Dans le contexte de cette relation, on qualifie la première de *sous-classe*, et la seconde de *superclasse*.

L'héritage permet d'exprimer des relations de généralisation/spécialisation parmi les classes d'un système. D'un point de vue plus pragmatique, ce mécanisme est également un bon moyen de factoriser le code. En effet, la description d'une classe n'est pas dupliquée dans ses sous-classes : la relation d'héritage est stockée en tant que telle dans le système logiciel.

1.3 Liaison dynamique & polymorphisme

On agit sur un objet en adressant un appel de méthode à la variable qui le représente. L'instruction d'appel comprend le nom de la variable destinataire, le nom de la méthode souhaitée et la valeur de ses arguments. Nous avons vu que le même nom de méthode peut être utilisé dans plusieurs classes, et que l'implémentation d'une méthode peut être redéfinie par héritage. Donc, selon la classe de l'objet destinataire, la même instruction peut conduire à l'éxécution de méthodes différentes, voire de versions différentes de la même méthode. Le processus de détermination de la méthode et de l'implémentation à éxécuter pour résoudre un appel est appelé *liaison*. Si le langage de programmation est compilé, et si la liaison est réalisée lors de la compilation, on la dit *statique*. Au contraire, si la liaison est réalisée lors de l'éxécution des programmes, on la dit *dynamique*.

Un langage est doté du *polymorphisme* si une variable donnée peut désigner des objets de classes différentes lors de l'éxécution du programme. Un tel langage utilise nécessairement la liaison dynamique, car il est impossible de résoudre un appel de méthode tant que l'on ne connaît pas la classe du destinataire.

Polymorphisme et liaison dynamique sont des composantes essentielles du paradigme objet. Ils permettent de manipuler des objets indépendamment de leur classe et de leur appliquer des méthodes sans se soucier de la manière dont elles sont concrètement mises en œuvre. Les programmes ainsi bâtis sont plus faciles à réutiliser.

1.4 Notions complémentaires

Certains mécanismes et concepts ne sont pas présents dans tous les langages de classes, ou ne sont encore utilisés que dans le cadre des méthodes de conception. Il relèvent cependant de la même philosophie et méritent qu'on les aborde brièvement.

Métaclasses

Les langages les plus orthodoxes postulent que tous leurs éléments sont des objets, y compris les classes elles-mêmes. De même qu'un objet est une instance de classe, une classe est une

instance de *métaclasse*. Certains auteurs ont montré les avantages de cette couche d'instanciation supplémentaire, en particulier une simplification de la syntaxe et une meilleure homogénéité des concepts.

Puisqu'une classe est aussi un objet, elle peut posséder ses propres données stockées dans des *attributs de classe*, et des *méthodes de classe* que l'on invoque par des appels à la classe et non pas à ses instances.

Les méthodes servant à construire les objets d'une classe sont intrinsèquement des méthodes de classe : on ne peut pas les invoquer en adressant des appels aux objets, puisqu'elles servent précisément à les construire. Ainsi, même lorqu'un langage n'est pas doté de métaclasses, ses auteurs sont obligés d'accorder un statut particulier aux méthodes de construction.

Héritage multiple et répété

Lorsque l'*héritage multiple* est autorisé, une classe peut hériter de plusieurs autres à la fois. Anodin au premier abord, ce principe peut amener une classe à hériter plusieurs fois d'une même superclasse par des chemins différents, ce qui constitue un cas d'*héritage répété*.

A titre d'exemple, si `B` et `C` sont deux sous-classes de `A`, et si `D` hérite à la fois de `B` et de `C`, alors `D` hérite deux fois de `A`. Ce cas d'héritage est ambigu : les attributs de `A` doivent-ils être répétés deux fois au sein de `D` ? En cas de répétition, comment désigner les différentes copies ? Selon le contexte, le programmeur peut souhaiter des comportements variés.

Les langages de classes adoptent des stratégies différentes face à ce problème : interdiction pure et simple de l'héritage multiple, procédé unique imposé par le langage, ou décision déléguée au programmeur. On fait souvent un usage abusif de l'héritage multiple, ce qui amène certains auteurs à le déconseiller.

Persistance

Un objet occupe une certaine position et un certain volume en mémoire, pendant un laps de temps déterminé. Atkinson suggère qu'il existe une échelle de degrés d'existence pour les objets, depuis les objets les plus éphémères qui interviennent dans l'évaluation d'une expression, jusqu'aux objets qui résident dans une base de données [ABW+89]. Un objet est qualifié de *persistant* s'il peut survivre à la disparition du programme qui l'a créé et s'il conserve son identité quand on le déplace d'une position à une autre en mémoire.

La notion de persistance est essentiellement utilisée dans les méthodes de conception par objets. Elle n'est directement intégrée à aucun langage de classes, mais on commence à voir apparaître des composants logiciels commerciaux qui permettent d'ajouter ce type de fonctionnalité à un langage : en fin d'éxécution de programme, les objets sont automatiquement transférés sur disque, et rétablis tels quels lorsque le programme est ultérieurement relancé. Les systèmes de gestion de bases de données orientés objets constituent un moyen plus sophistiqué d'assurer la persistance des objets, avec en supplément une gestion des accès concurrents de plusieurs utilisateurs. Quel que soit le moyen retenu, le programmeur n'a plus besoin de gérer explicitement la sauvegarde de ses données : il lui suffit de déclarer certains objets comme persistants pour être assuré de les retrouver intacts lors d'une éxécution ultérieure de son programme.

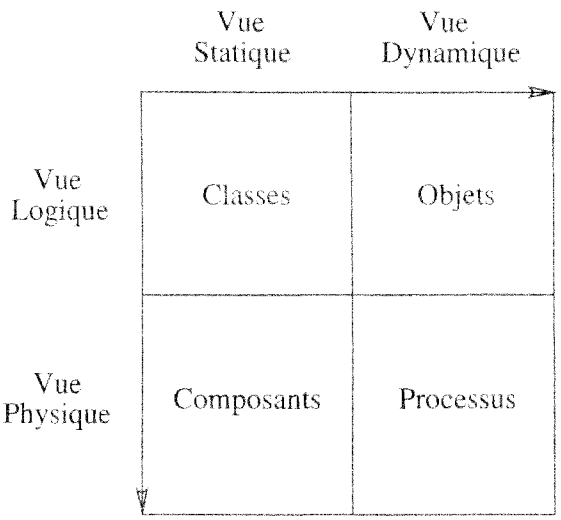


FIG. 2.1 – *Points de vue sur un système logiciel*

2 Notation de Booch

Dans la section précédente, nous avons présenté les caractéristiques des langages de programmation permettant de réaliser des systèmes logiciels à objets. Pour concevoir de tels systèmes, ou les présenter à des tiers, un langage de programmation ne constitue pas un support satisfaisant. Il est souvent préférable de disposer d'une notation plus graphique. Nous avons adopté celle de Booch, et nous allons en présenter maintenant les éléments principaux. La notation est composée d'un ensemble de diagrammes, chacun mettant en relief un aspect particulier du système logiciel à décrire. La figure 2.1 illustre les différents points de vue selon lesquels on peut observer le système, et les quatre familles de diagrammes correspondants :

- Les *diagrammes de classes* illustrent les relations statiques entre les classes.
- Les *diagrammes d'objets* illustrent les appels de méthodes entre les objets durant l'éxécution du programme. Chaque diagramme est l'expression d'un *scénario*.
- Les *diagrammes de composants* illustrent l'implantation physique des classes par des fichiers et des répertoires.
- Les *diagrammes de processus* illustrent la répartition des tâches entre les processeurs, en cas d'éxécution en parallèle.

2.1 Diagrammes de classes

La figure 2.2 montre l'icône représentant une classe (nuage pointillé) et les trois sortes de relations que l'on peut établir entre deux classes :

- Le disque blanc dénote une relation d'utilisation : la classe `A` utilise la classe `B` (comme variable interne, argument ou résultat d'une méthode de `A`).

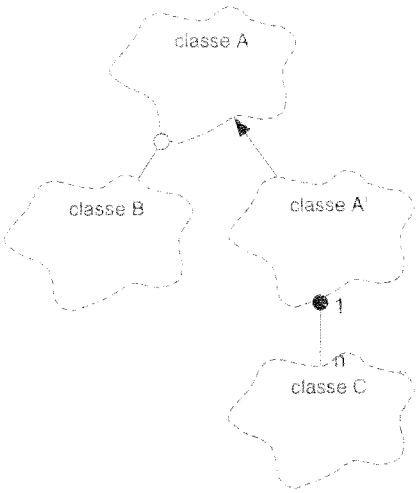


FIG. 2.2 – *Diagramme de classes*

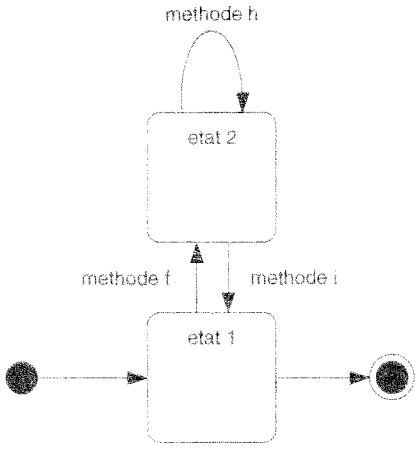


FIG. 2.3 – *Diagramme de transitions d'états de la classe* B

- La flèche dénote l'héritage : la classe `A'` hérite de `A` (des attributs et des méthodes sont ajoutées, et/ou des méthodes sont redéfinies).

- Le disque noir dénote l'agrégation, et des indications de cardinalité peuvent être ajoutées : chaque instance de `A'` contient un nombre illimité d'instances de `C`.

Il arrive que les instances d'une classe aient un comportement proche de celui d'une machine à états finis. Même si les langages de classes ne permettent pas d'implémenter explicitement les états finis, il peut être utile de compléter la documentation d'une classe par un *diagramme de transitions d'états*, montrant l'espace des principaux états apparents de la classe et les méthodes qui causent des transitions de l'un à l'autre. La figure 2.3 réprésente les états finis d'une instance de la classe `B`. L'opération `f` fait passer l'objet de l'état `1` à l'état `2`, et l'opération `i` ramène l'objet dans l'état `1`. Le disque noir permet de désigner l'état initial d'un objet, et le même disque entouré d'un cercle désigne l'état dans lequel se trouve normalement un objet lorsqu'il est

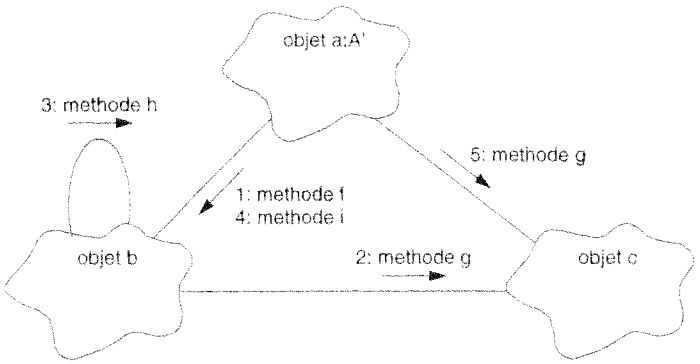


FIG. 2.4 – *Diagramme d'objets*

détruit.

2.2 Diagrammes d'objets

Un diagramme d'objets permet d'exprimer un extrait d'éxécution, ou *scénario*, d'un système logiciel. Le diagramme met en avant les objets qui sont impliqués dans le scénario et les appels de méthodes qu'ils échangent.

Dans la figure 2.4, on peut observer qu'un objet est représenté par la même icône qu'une classe, mais en trait plein. Des flèches représentent les appels de méthodes, et ces appels sont numérotés dans leur ordre d'émission. Le nom inscrit dans l'icône d'objet peut prendre plusieurs formes selon le besoin :

- « `c` » : nom de l'objet.
- « `:C` » : classe de l'objet.
- « `c :C` » : nom et classe de l'objet.

On peut résumer ainsi le scénario de la figure 2.4 :

1. l'objet `a` de classe `A'` appelle la méthode `f()` de l'objet `b`,
2. l'objet `b` appelle la méthode `g()` de l'objet `c`,
3. l'objet `b` appelle sa propre méthode `h()` puis rend le contrôle à l'objet `a`,
4. l'objet `a` appelle la méthode `i()` de l'objet `b`,
5. l'objet `a` appelle la méthode `g()` de l'objet `c`.

Il est parfois souhaitable de mettre plus en valeur l'ordre d'émission des messages. On remplace alors le diagramme d'objets par un diagramme d'interaction, tel que celui de la figure 2.5.

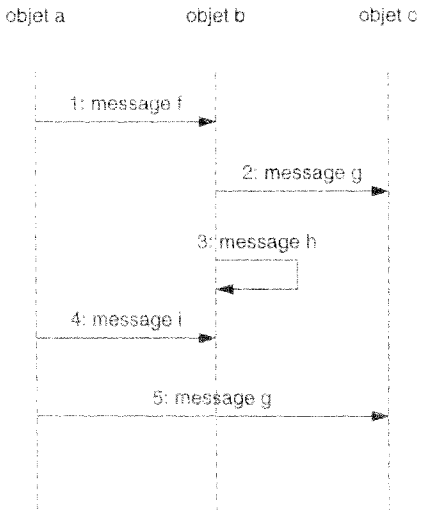


FIG. 2.5 – *Diagramme d'interaction*

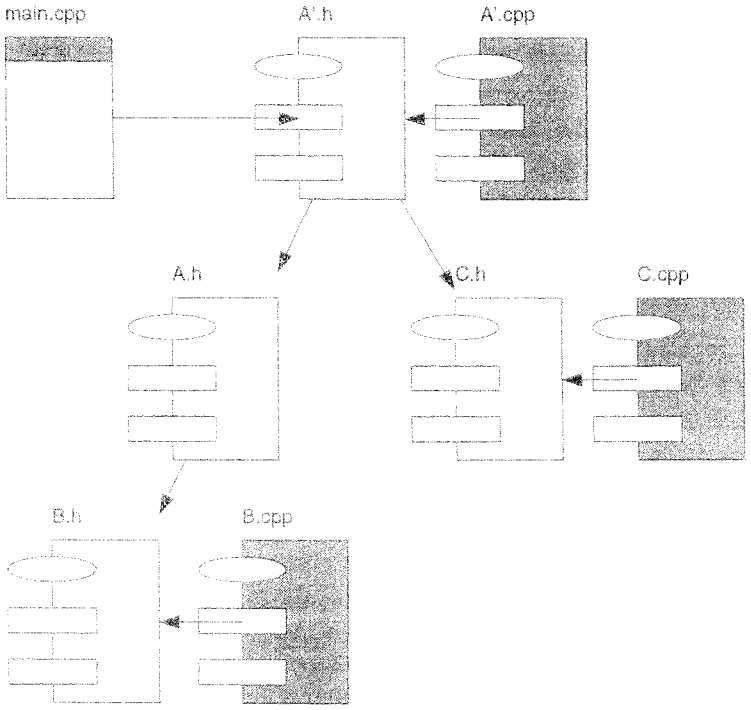


FIG. 2.6 - *Diagramme de composants*

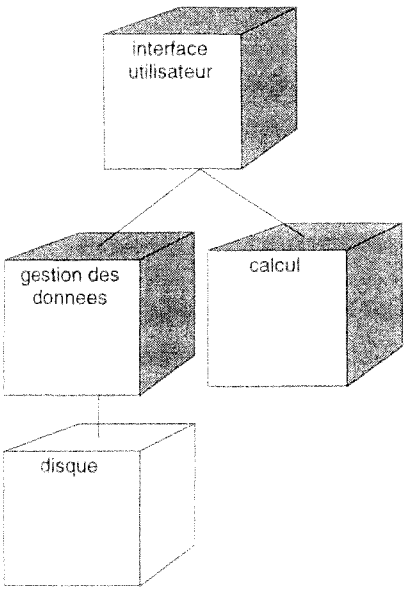


FIG. 2.7 – *Diagramme de processus*

2.3 Diagrammes de composants

Certains langages imposent la façon dont la description des classes est stockée dans le système de fichiers du système d'exploitation, ou prennent complètement en charge cet aspect. Dans le cas contraire, le concepteur d'un système logiciel peut spécifier l'organisation physique de son système à l'aide de diagrammes de composants. Les éléments centraux de ces diagrammes sont les *fichiers d'entête* et les *fichiers d'implémentation* qui contiennent respectivement la déclaration et la définition des éléments du système.

Pour un langage compilé, l'organisation la plus simple consiste à créer un couple de fichiers pour chaque classe. Le fichier d'implémentation peut-être compilé de façon autonome, et celui d'entête contient l'information nécessaire pour la compilation séparée des autres fichiers dans lesquels on utilise la classe.

La figure 2.6 montre les icônes de base qui permettent de représenter les différents sortes de fichiers. L'icône à fond blanc désigne un fichier d'entête. Celle à fond gris désigne un fichier d'implémentation. On notera également l'icône spéciale dédiée à l'éventuel fichier principal du programme.

La seule relation possible entre deux fichiers est la dépendance de compilation, représentée par une flèche. Le fichier d'implémentation d'une classe dépend généralement du fichier d'entête de cette classe et des fichiers d'entête de toutes les classes avec lesquelles elle a des relations.

2.4 Diagrammes de processus

Nous terminons ce survol des diagrammes de Booch par celui qui traite le parallélisme éventuel en illustrant la distribution des tâches sur les processeurs ainsi que les connexions entre processeurs et périphériques. La figure 2.7 contient un périphérique, représenté par le cube à

bords blancs, et trois processeurs, représentés par les cubes à bords gris. Chaque cube représente un élément matériel, et chaque ligne une connexion physique entre ces éléments. Les périphériques sont des éléments passifs, et leurs icônes portent simplement le nom où le type du périphérique. Par contre, les processeurs sont des éléments actifs, et chaque icône porte le nom de la tâche ou des tâches que le processeur éxécute.

3 Programmation en C++

Nous avons choisi de programmer en langage C++ parce qu'il était le langage le plus répandu et que notre équipe souhaitait collaborer activement avec l'industrie. Nous voulons maintenant éclaircir autant que possible l'impact de ce choix sur nos travaux.

Nous allons d'abord brosser le tableau des langages de classes les plus représentatifs afin de mieux situer C++ (3.1), puis nous détaillerons les caractéristiques spécifiques de ce dernier (3.2). Nous pourrons finalement discuter de l'adéquation de C++ à notre problématique (3.3).

3.1 Panorama des langages de classes

Le langage de classes le plus représentatif est sans doute *Smalltalk* [GR89], un des précurseurs dans le domaine (après Simula dont il s'est largement inspiré). C'est à la fois un langage et un environnement de programmation. Les auteurs ont cherché à privilégier la cohérence et l'homogénéité : tous les éléments du langage et de l'environnement sont des objets et s'utilisent à l'aide de méthodes. A tort ou à raison, Smalltalk souffre encore aujourd'hui d'une réputation d'efficacité médiocre, sans doute parce qu'il utilise un typage dynamique.

Dans une veine assez proche, Lisp a donné le jour à de nombreux dialectes orientés objets. *CLOS* [Ste90] est né d'un effort de standardisation de ces dialectes. Il devait reprendre le meilleur de chacun d'entre eux, être suffisamment puissant et flexible pour s'adapter à la plupart des domaines d'application, tout en restant extensible et sujet à de futures recherches en programmation par objets. Ce dernier trait semble dominer les autres.

Eiffel a été créé par Bertrand Meyer [Mey88b]. C'est un langage très tourné vers le génie logiciel et qui met en avant les principes d'une bonne conception logicielle : de bonnes spécifications de classes, un typage fort et la réutilisation à travers l'héritage et les classes génériques. Contrairement aux deux langages précédents, Eiffel utilise le typage statique, ce qui le rend a priori plus efficace. Malgré une excellente réputation, ce langage de choix reste trop peu utilisé et mal doté en outils de développement.

C++ a été conçu par Bjarne Strousup [ES90, Lip91]. Il s'agit d'une évolution du langage C, et fait office de mauvais petit canard : il ne possède ni la rigueur d'Eiffel ni l'orthodoxie de Smalltalk. Néanmoins, grâce à son affinité avec Unix et le C, il est sans doute aujourd'hui le langage à objets le plus répandu et celui pour lequel on trouve le plus de bibliothèques de classes clés-en-main.

3.2 Spécificité de C++

Le langage C++ reprend les caractéristiques habituelles d'un langage procédural (sémantique par valeur, typage fort, généricité, compilation) auquel il adjoint de nouveaux éléments liés

au paradigme objet (sémantique par référence, encapsulation, héritage, polymorphisme, liaison dynamique). Ces deux natures sont conciliées avec plus ou moins de bonheur. Dans l'exposé qui suit, nous n'utiliserons pas la terminologie spécifique du C++. Nous préférons éviter un foisonnement inutile de termes équivalents et nous en tenir à ceux qui ont été introduits dans la section 1.

Compatibilité avec le C

Il est toujours permis de déclarer des variables globales ou des procédures indépendantes. Les types prédéfinis (`int`, `float`, `char`) garde le même statut et ne sont pas considérés comme des classes à part entière. Il est toujours permis de définir des *structures* (agrégats d'attributs, sans méthodes et sans encapsulation). Enfin, un programme commence toujours par l'éxécution de la procédure principale `main`.

Ceci garantit une compatibilité immédiate avec les bibliothèques écrites en C, mais laisse la liberté au programmeur d'adopter un style de programmation hétérogène mêlant classes, variables globales et procédures.

Sémantique par valeur et par référence

Les langages procéduraux possèdent généralement une *sémantique par valeur*. Les variables contiennent directement les données qu'elles désignent, et n'existent plus que sous la forme d'une zone en mémoire après la compilation. Dans cette sémantique, l'affection d'une variable à une autre provoque une copie des données.

Les langages à objets possèdent nécessairement une *sémantique par référence*, selon laquelle les variables ne contiennent pas directement les données mais leurs adresses. Dans cette sémantique, l'affectation provoque une copie de l'adresse et les deux variables partagent les mêmes données. Ce mode de fonctionnement est indispensable pour permettre le polymorphisme.

Les variables normales de C++ fonctionnent selon une sémantique par valeur. On peut obtenir l'autre mode de fonctionnement en les déclarant comme des *références* ou comme des *pointeurs*. Dans ce dernier cas, l'indirection est explicite et on peut manipuler directement l'adresse des données. Le programmeur dispose donc de moyens variés pour mettre en œuvre des variables de toutes natures.

Classes et objets

La déclaration d'une nouvelle classe en C++ comprend ses attributs et l'interface de ses méthodes. On stocke généralement cette déclaration dans un fichier d'entête, et l'implémentation des méthodes est décrite séparement dans un fichier d'implémentation. Il eût été plus conforme à la philosophie objet de placer les attributs au sein de l'implémentation plutôt qu'au sein de l'entête, mais le compilateur a besoin de connaître la taille des instances pour la compilation séparée des fichiers clients (qui peuvent contenir des variables par valeur de la classe).

La notion d'objet n'a pas été généralisée aux classes et il n'y a pas de métaclasses. Il est tout de même possible de déclarer des attributs de classes et des méthodes de classes pour manipuler ces mêmes attributs. La construction et la destruction des objets se fait à l'aide de méthodes

spéciales appelées *constructeurs* et *destructeurs*.

L'unité de protection pour l'encapsulation est la classe et non l'objet : deux instances d'une même classe peuvent mutuellement accéder à leurs attributs privés. En revanche, le programmeur dispose d'un arsenal très riche pour préciser les droits d'accès. Chaque attribut ou méthode possède l'une des visibilités suivantes : *publique*, *protégée* ou *privée*. Le niveau protégé correspond à des éléments qui sont accessibles aux sous-classes mais pas aux classes clientes. Cela permet de créer dans les classes des méthodes utilitaires destinées à leurs sous-classes. En sus, il est possible d'attribuer des *amis* à une classe, c'est-à-dire des clients privilégiés qui ont accès à ses attributs privés. Ce dernier mécanisme est perçu par les uns comme une violation de l'encapsulation, et par les autres comme un raffinement complémentaire.

Typage

Le langage C++ est *fortement typé* : lorsque l'on déclare une variable, un attribut ou un argument, il faut préciser la classe de l'objet ainsi désigné. On peut également indiquer si la variable est une *référence*, un *pointeur* ou même une *constante*. L'ensemble de ces informations forme le *type* de la variable. La cohérence des déclarations est vérifiée par le compilateur : il s'agit d'un *typage statique*.

Ce typage permet au programmeur d'exprimer plus précisement le rôle des variables qu'il utilise et de mieux maîtriser la cohérence de son code. La compilation est l'occasion de détecter de façon précoce les erreurs de conception d'un programme et produit un code éxécutable beaucoup plus fiable.

De surcroît, le compilateur peut exploiter les déclarations de types pour produire des programmes plus compacts et plus rapides : on connaît par avance la taille des zones mémoire, et on peut réaliser une liaison statique des méthodes. Le typage statique nous semble indispensable pour assurer la fiabilité d'un programme de grande taille et avoir de bonnes performances.

Surcharge

Dans la description générale des langages de classes, nous avons déjà vu que le même nom peut être utilisé pour plusieurs méthodes, à condition qu'elles appartiennent à des classes différentes. C++ permet également d'utiliser plusieurs fois le même nom au sein d'une classe, si le type d'au moins un argument est différent. Ce procédé est appelé *surcharge*, et la méthode est dite *surchargée*.

La surcharge permet de simplifier les noms de méthodes et facilite l'écriture du code. En contrepartie, elle complique le procédé de liaison, qui doit s'appuyer sur le type des arguments pour déterminer la méthode désignée dans un appel.

Héritage

L'héritage de C++ permet naturellement d'ajouter des attributs, des méthodes, et de redéfinir l'implémentation des méthodes existantes. La redéfinition ne peut porter que sur le corps des méthodes : si les arguments sont différents en nombre ou en type, on considère qu'il s'agit d'une méthode différente, surchargée.

Une déclaration d'héritage peut-être *publique* ou *privée*. La première variante correspond à l'héritage classique, exprimant une relation de généralisation/spécialisation. La variante privée fait en sorte que les attributs et les méthodes héritées soient privées au sein de la sous-classe. Il s'agit d'un *héritage d'implémentation*, uniquement destiné à factoriser le code. Ce type d'héritage est parfois utilisé abusivement, et il convient de s'assurer au cas par cas si l'agrégation ne serait pas plus indiquée.

L'héritage multiple est autorisé. Lorsqu'il y a héritage répété, il y par défaut une répétition des attributs de la superclasse. Pour éviter cette répétition, le programmeur doit explicitement déclarer que la relation d'héritage est *virtuelle*.

Masquage

La combinaison de la surcharge et de la redéfinition des méthodes lors de l'héritage peut induire un grand nombre de méthodes de même nom dans un arbre de classes. Il devient alors difficile pour le programmeur de recenser toutes les variantes dont hérite une classe donnée. Pour réduire les risques de confusion, le langage C++ possède une règle de *masquage* : la redéfinition d'une méthode dans une classe *masque* toutes les méthodes de même nom dans ses superclasses. Si le programmeur veut à nouveau disposer de plusieurs méthodes surchargées dans la classe, il doit les redéfinir toutes explicitement, et seul ce jeu de méthodes sera transmis aux sous-classes du niveau suivant. La règle de masquage permet au programmeur de conserver une certaine visibilité sur les méthodes qu'il invoque.

Conversions automatiques

La possibilité de déclarer des variables, des références, des pointeurs et des constantes démultiplie le nombre de types possibles. En compensation, le compilateur est capable de procéder automatiquement à une grande variété de conversions (transformations en constante ou en référence, conversions entres les types numériques élémentaires), et le programmeur peut définir des conversions supplémentaires qui seront automatiquement appliquées par le compilateur. Lorsqu'on abuse de cette dernière possibilité, on peut aboutir à des conversions abérrantes sans s'en rendre compte.

Liaison statique

Comme la plupart des variables sont typées et ne sont pas polymorphes, la liaison C++ est majoritairement statique. Lorsqu'un appel à la méthode `m` est adressé à une variable de classe `C`, la liaison peut se résumer très schématiquement aux étapes suivantes :

1. *Résolution du masquage* : si `C` définit une ou plusieurs méthodes `m`, ces dernières sont considérées comme candidates ; sinon, les candidates sont les méthodes `m` héritées de la superclasse la plus proche.

2. *Résolution de la surcharge* : parmi les méthodes candidates, le compilateur détermine celle qui est la plus compatible avec les arguments de l'appel.

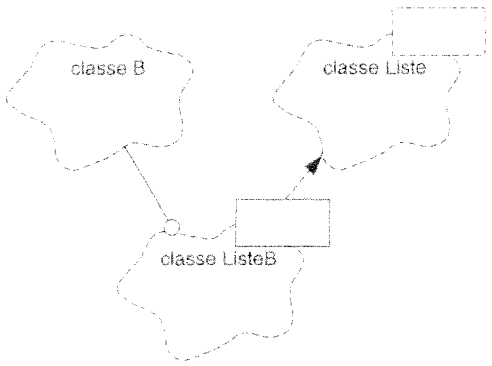


FIG. 2.8 – *Classes génériques dans les diagrammes de Booch*

3. *Conversion des arguments*: les arguments qui ne possèdent pas exactement le type approprié sont convertis à l'aide des méthodes de conversion prédéfinies où fournies par le programmeur.

Polymorphisme & liaison dynamique

Typage et polymorphisme sont apparemment incompatibles : une variable ne peut pas être déclarée avec une classe `C` puis se voir attribuer des objets de classe quelconque. C++ offre un compromis : les objets doivent être des instances des sous-classes de `C`. En effet, ces objets ont hérité des méthodes de `C` et peuvent donc se comporter comme des instances de cette dernière, sans déroger aux règles du typage. Ce genre de polymorphisme est qualifié de *restreint*, et il ne s'applique qu'aux références et aux pointeurs, qui seules obéissent à une sémantique par référence.

La liaison n'est pas dynamique pour autant : par défaut, elle reste statique et le compilateur se fonde sur le type de la variable pour résoudre un appel de méthode, même si cette variable est polymorphe. La liaison dynamique s'obtient au prix d'une instruction supplémentaire : il faut déclarer la méthode comme *virtuelle* dans la définition de la classe.

Donc, seuls les appels de méthodes virtuelles adressés aux références et aux pointeurs sont résolus dynamiquement, en fonction de la classe exacte de l'objet destinataire. Dans tous les autres cas, la liaison est statique.

Généricité

C++ est doté d'un mécanisme de *généricité*. On appelle *classe générique* une classe dont tous les types utilisés ne sont pas complètement définis. A chaque fois qu'on utilise une telle classe dans une instruction, il faut préciser la valeur des types manquants, ce qui est assimilé à une forme d'instanciation.

Les classes génériques peuvent être représentées dans les diagrammes de Booch. La figure 2.8 illustre une classe générique `Liste` et une instance de cette classe avec le paramètre `B`. La relation d'instanciation est représentée par la flèche pointillée. La plupart du temps, seule l'instance est représentée dans un diagramme.

On peut simuler la *généricité* à l'aide de l'héritage, mais au prix d'une grande verbosité et

d'une éxécution plus lente. La disponibilité de classes génériques simplifie la vie du programmeur en déplaçant la majorité du travail vers le compilateur. En outre, c'est une solution idéale pour implémenter les *collections* d'objets (tables, piles, listes, arbres, graphes...).

3.3 Adéquation de C++ à nos besoins

Le langage dispose de tous les mécanismes nécessaires pour pratiquer une programmation par objets. Il se prête donc bien au prototypage et à la mise en commun des programmes développés par les membres d'une équipe de recherche. On regrette toutefois que l'encapsulation ne soit pas obligatoire et que la liaison ne soit pas dynamique par défaut pour les variables polymorphes. On regrette aussi que certaines tâches restent à la charge du programmeur alors qu'elles pourraient être réalisées automatiquement, par exemple la destruction des objets inutilisés.

A l'inverse, la compatibilité avec le langage C, la sémantique par valeurs et le typage statique permettent de construire du code efficace lorsque c'est nécessaire. Cela nous intéresse pour transférer nos résultats à l'industrie et mener des projets en vraie grandeur : la simulation des transferts radiatifs est encore largement tributaire des temps de calcul.

C'est donc la force et la faiblesse du langage C++ que de permettre un style de programmation par objets aussi bien qu'un style plus efficace et de plus bas niveau. Il peut ainsi soutenir à la fois des activités de recherche et de transfert industriel. En contre-partie, la syntaxe est très complexe, et promet de le devenir plus encore dans les évolutions à venir. Cela ne poserait pas de problème si le langage se comportait par défaut conformément aux principes objets, et si les instructions les plus difficiles étaient réservées aux spécialistes chargés de produire les applications industrielles. On se trouve malheureusement confronté à la situation inverse.

Les lacunes d'un langage peuvent être amoindries en instaurant des règles de programmation communes au sein d'un groupe de collaborateurs. Dans le cas de C++, cela s'avère indispensable. Au sein de notre équipe, nous avons adopté un style de programmation susceptible de corriger les défauts stigmatisés ci-dessus, en nous inspirant en partie de ce qu'ont préconisé les laboratoires suédois Ellemtel [HN92]. Pour l'essentiel, les principes sont les suivants :

- uniformiser le nommage des différents éléments, la mise en page et l'implantation sous Unix,
- mettre en garde contre les usages abusifs ou erronés de l'héritage, de la surcharge et des pointeurs,
- décourager certaines pratiques issues du C et proposer des alternatives (remplacement des variables globales et des procédures libres par des classes statiques),
- exhorter au respect de l'encapsulation (les attributs des classes doivent être privés),
- encourager l'usage du polymorphisme et de la liaison dynamique (utilisation de références ou de pointeurs, et déclaration de méthodes virtuelles),
- expliquer la problématique liée à la sémantique par référence (assignations, copies, destructions),
- arbitrer les dilemmes entre agrégation, héritage et généricité,

- identifier les pratiques délicates à réserver aux utilisateurs avertis (redéfinition des opérateurs, gestion des exceptions).

Le but de ces règles n'était pas tant de brider les spécialistes que de guider les débutants. Il s'agissait de leur proposer une variante de C++ plus simple et plus conforme à l'orientation objets.

4 Réutilisation logicielle

Comme nous l'avons signalé en début de chapitre, notre adoption de l'orientation objets ne répond qu'à une motivation : réduire le travail de programmation nécessaire pour expérimenter un nouvel algorithme de simulation ou le transférer à l'industrie. Pour y parvenir, il faut fournir aux chercheurs un système logiciel à partir duquel ils puissent bâtir rapidement leurs programmes. Nous allons comparer plusieurs genres de systèmes, en les jugeant sur la base de deux critères complémentaires :

- *Évolutivité* : possibilité de modifier le système sans perturber les programmes qui l'utilisent.
- *Adaptabilité* : possibilité de modifier le système dans le cadre d'un programme donné sans perturber les autres programmes.

Les modifications apportées à un système peuvent porter sur de multiples aspects. Nous en retiendrons trois :

- *Type des paramètres.*
- *Structure des données.*
- *Implémentation des procédures.*

Après avoir mis en évidence certaines limites des approches procédurale et modulaire, nous allons donner quelques pistes sur les avantages de la programmation par objets. On peut s'en faire une idée plus complète en consultant [Mey88a] et [Cou96]. Nous évoquerons également les limitations qui persistent, ce qui nous conduira finalement au concept d'*architecture générique*.

4.1 Bibliothèques de procédures

Les bibliothèques de procédures sont les premiers systèmes logiciels réutilisables à avoir remporté quelques succès, notamment dans le domaine de l'analyse numérique. Malheureusement, peu de domaines d'application se prêtent à une approche aussi simple : il faut notamment que les problèmes du domaine puissent être exprimés simplement, sans utiliser trop de paramètres, et qu'ils soient relativement indépendant les uns des autres.

Si ces conditions ne sont pas réunies, la tentation est grande d'avoir recours à des variables globales pour éviter la multiplication des paramètres ou contenir les données partagées par plusieurs procédures. Ces variables sont particulièrement nuisibles à la réutilisation :

- la structure des données globales est exposée aux programmes, et on ne peut plus la modifier sans devoir les mettre à jour ;

- les accès aux variables sont dispersés parmi toutes les procédures, et il est difficile de maîtriser leur cohérence ;
- il est impossible de détecter les accès des programmes aux variables, à plus forte raison de s'assurer de la validité des valeurs qui leur sont affectées.

Quand elle ne possède pas de variables globales, une bibliothèque de procédures peut être considérée comme évolutive, dans la mesure où on peut corriger l'implémentation des procédures sans générer de mise à jour des programmes, à part une éventuelle recompilation.

Par contre, l'utilisateur dispose de peu de moyens pour adapter la bibliothèque à ses besoins spécifiques, puisque ces moyens se réduisent aux valeurs des paramètres des procédures. S'il souhaite revoir l'implémentation d'une procédure, l'utilisateur est généralement obligé de faire une copie du code source. Il travaille alors avec sa propre version, et perd le bénéfice des améliorations apportées ultérieurement à la bibliothèque.

4.2 Bibliothèques de modules

Les désavantages liés à la visibilité des variables globales peuvent être réduits en interposant des *procédures d'accès* entre ces variables et les programmes. On a également intérêt, au sein même de la bibliothèque, à identifier clairement et à restreindre le nombre de procédures accédant à chaque groupe de variables globales. On est ainsi amené à décomposer la bibliothèque en *modules* (*packages*), chaque module englobant un groupe de variables globales et les procédures qui leur sont liées. On considère que la décomposition est réussie si les modules ont une grande cohésion interne et sont faiblement couplés les uns aux autres.

Certains langages intègrent la notion de module, où permettent de placer des variables dans des zones dont l'accès est restreint à certaines procédures. On peut ainsi organiser une bibliothèque en modules et faire en sorte que leur encapsulation soit respectée.

Les bibliothèques de modules sont applicables à la plupart des domaines d'applications, des plus simples aux plus compliqués. Si l'encapsulation des modules est respectée, l'évolutivité est bonne à la fois pour l'implémentation des procédures et pour les structures de données. En outre, les utilisateurs comme les créateurs de la bibliothèque localisent plus facilement ce qu'ils cherchent.

Malheureusement, une bibliothèque de modules est aussi inadaptable qu'une bibliothèque de procédures, même si le découpage en modules améliore un peu la situation. En effet, si un utilisateur est contraint de dupliquer du code pour en modifier le contenu, il peut limiter son intervention à quelques modules, et continuer à bénéficier des évolutions des autres modules.

4.3 Bibliothèques de classes

La notion de classe peut être vue comme la continuité de celle de module. La seule différence se situe dans le mécanisme d'instanciation et apparaît lors de l'éxécution : alors qu'un module possède un seul exemplaire de ses variables, les attributs d'une classe sont dupliqués autant de fois qu'il y a d'objets. Dans les deux cas, l'encapsulation des données assure l'évolutivité des bibliothèques. Par contre, les mécanismes supplémentaires de l'orientation objet ouvrent enfin la voie de l'adaptabilité.

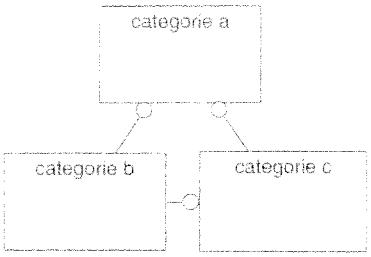


FIG. 2.9 – *Catégories dans les diagrammes de Booch*

Grâce à l'héritage, un utilisateur peut étendre et redéfinir les méthodes d'une classe sans dupliquer cette classe, donc sans modifier la bibliothèque d'origine, et il pourra toujours bénéficier des évolutions futures de cette bibliothèque. Par ailleurs, si les paramètres de la bibliothèque sont stockés dans des variables polymorphes avec une liaison dynamique, alors l'utilisateur peut également susbtituer aux paramètres initialement prévus des objets dont la classe hérite de celle déclarée par la bibliothèque.

Les limites d'adaptabilité d'une bibliothèque de classes sont liées aux règles de l'héritage : on peut redéfinir une méthode, ajouter des méthodes et des attributs, mais on ne peut pas en supprimer. Dans la mesure où les attributs sont encapsulés, et appartiennent à l'implémentation d'une classe plutôt qu'à son interface, il est dommage de ne pas pouvoir les redéfinir.

4.4 Architectures génériques *(frameworks)*

Nous venons de voir les bénéfices immédiats qu'apportent l'orientation objets, et ce qui nous manquent encore pour faciliter l'adaptation d'une bibliothèque de classes. Nous décrivons ci-dessous deux nouveaux mécanismes qui permettent d'aller plus loin, et qui conduisent finalement à la notion d'*architecture générique*,

Catégories

L'encapsulation généralisée des données a pour effet secondaire de démultiplier le nombre de méthodes et de classes, au point qu'il devient souhaitable de créer des groupes de plus haut niveau. Booch appelle ces groupes de classes des *catégories*. Les règles de cohésion et de couplage de la modularité s'appliquent aussi au découpage en catégories.

La figure 2.9 montre l'icône représentant une catégorie dans les diagrammes de Booch. Le seul type de relation possible entre deux catégories est l'utilisation : une catégorie en utilise une autre si au moins l'une de ses classes a une relation d'utilisation, d'agrégation ou d'héritage avec au moins l'une des classes de la catégorie utilisée. Dans la figure 2.9, la catégorie `a` utilise les catégories `b` et `c`, et la catégorie `c` utilise la catégorie `b`.

Les catégories peuvent être à leur tour classées en couches logicielles, de telle sorte que les catégories d'une couche donnée n'utilisent que les catégories des couches inférieures. Cette nouvelle hiérarchie révèle les différents niveaux d'abstraction d'un ensemble de classes.

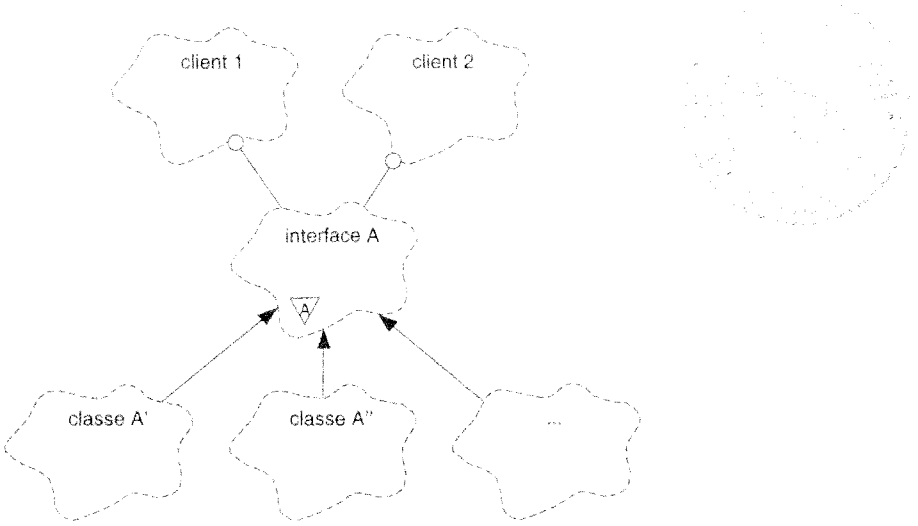


FIG. 2.10 – *Classe abstraite*

Classes abstraites

Certains langages donne la possibilité au programmeur de définir l'interface d'une méthode sans en fournir l'implémentation. Une classe dont au moins une méthode ne possède pas d'implémentation est dite *abstraite*, et ne peut pas avoir d'instance tant qu'elle n'a pas été complétée dans une sous-classe, que l'on qualifiera par opposition de *classe concrète*.

Une classe abstraite peut-être utilisée comme type pour une variable polymorphe à liaison dynamique. En effet, les objets que l'on affectera à la variable seront par définition des instances de sous-classes concrètes, possèdant des implémentations de toutes les méthodes, et la liaison se fera sans difficulté. On entrevoit donc déjà le rôle que peuvent jouer les classes abstraites : représenter des interfaces standards.

Dans la notation de Booch, les classes abstraites sont décorées à l'aide d'une lettre « A » dans un triangle, comme l'est la classe `A` dans la figure 2.10. D'après cette même figure, les deux classes clientes n'ont besoin que de connaître `A` pour que leurs instances sachent utiliser indifféremment toute instance de `A'`, `A''`, ou de toute nouvelle sous-classe qui serait créée ultérieurement.

En C++, il est possible de déclarer une méthode comme *purement virtuelle*, ce qui indique au compilateur que l'implémentation de la méthode n'est pas définie. Les classes C++ qui ont au moins une méthode purement virtuelle ne peuvent pas être instanciées.

Vers une conception réutilisable

Nous avons regretté un peu auparavant l'impossibilité de redéfinir les attributs d'une classe lors de l'héritage, ce qui limite l'adaptabilité d'une bibliothèque. Le créateur d'une bibliothèque peut pallier cette lacune en exploitant le mécanisme de classe abstraite. Là où il aurait normalement défini une seule classe, il lui suffit d'en ajouter une seconde, superclasse de la première, ne possédant aucun attribut (et donc aucune implémentation de méthode). Ainsi, l'utilisateur qui veut redéfinir son propre jeu d'attributs peut choisir d'hériter de la superclasse abstraite, et toutes les adaptations deviennent possibles.

Pour les besoins de la démonstration, nous nous sommes focalisés sur l'adaptabilité et sur la redéfinition des attributs. Il faut aussi considérer que le créateur d'un système réutilisable ne connaît pas forcément en détail tous les objets qui seront manipulés, et que les classes abstraites lui fournissent le moyen de définir des classes incomplètes. De plus, le principe proposé ci-dessus ne doit pas être appliqué de façon radicale. Il serait déraisonnable de dédoubler toutes les classes, et le créateur peut vouloir imposer certains attributs et certaines méthodes. Il doit donc sélectionner au cas par cas les classes et les méthodes qu'il veut rendre abstraites.

Au terme de cette section sur la réutilisation logicielle, il faut souligner que nous avons glissé de la réutilisation de code à la réutilisation de conception. En effet, comme le souligne Rebecca Wirfs-Brock [WBJ90], une classe abstraite peut être vue comme la conception de ses sous-classes concrètes, et une collection de classes abstraites peut être vue comme la conception d'un système logiciel. C'est pourquoi, quand une bibliothèque de classes comprend un taux élevé de classes abstraites, ses auteurs préfèrent le terme d'architecture générique (*framework*) à celui de bibliothèque (*library*). Rebecca Wirfs-Brock ajoute encore qu'une architecture générique est généralement spécialisée pour un domaine d'application donné, et que l'on peut quasiment la considérer comme une formalisation de ce domaine. Plus que dans la réutilisabilité des implémentations, la puissance d'une architecture générique réside dans la réutilisabilité des interfaces et des regroupements de méthodes.

C'est un système de ce genre que nous avons cherché à créer pour notre domaine d'application.

5 Post scriptum

On peut s'étonner de trouver autant de rappels et de discussions sur l'orientation objets dans une thèse consacrée à la simulation des transferts radiatifs. Le fait est que cette approche logicielle, si elle est prometteuse, n'est pas encore totalement défrichée. Ce qui devait n'être qu'un outil est devenu pour une part l'objet de nos recherches, en particulier la problématique des architectures génériques.

Malgré l'investissement de temps que cela suppose, nous encourageons ceux qui s'intéressent aux aspects logiciels de la synthèse d'images à utiliser l'orientation objets. Après plusieurs années de satisfactions ou de frustrations, nous restons convaincus que cette approche est bénéfique et indispensable. Par ailleurs, les perspectives sont plutôt encourageantes :

- Trois des plus grands ténors de la modélisation objet, *Booch*, *Rumbaugh* et *Jacobson*, viennent de proposer un langage unifié pour décrire les systèmes logiciels à objets.
- Une norme ANSI/ISO pour le C++ sera bientôt finalisée, et on trouve d'ores et déjà des compilateurs revendiquant le qualificatif ANSI.
- Internet et objets font bon ménage. Le succès de la toile numérique devrait accélérer la diffusion de l'orientation objets, comme en témoignent Java et Corba.

Nous espérons que les spécialistes nous aurons pardonné les raccourcis de ce chapitre. Au delà du strict minimum nécessaire pour comprendre les termes et les diagrammes de ce mémoire, nous tenions à donner quelques pistes aux chercheurs de notre domaine qui envisagent de « passer à l'objet ».

Chapitre 3

Systèmes logiciels pour la simulation de l'illumination globale

Ce chapitre est consacré à la littérature abordant les aspects logiciels de la synthèse d'images. Il est principalement ciblé sur les méthodes que nous avons présentées dans le chapitre 1, mais les travaux sur ce thème sont relativement peu nombreux. Afin d'élargir notre champ d'investigation, nous avons considéré l'ensemble des systèmes logiciels capables de simuler l'illumination globale, et nous avons également intégré dans notre analyse quelques produits à caractère industriel. Les différents systèmes ont été évalués à la fois sur leur capacité à produire des images et sur leur capacité à servir de banc d'essai pour de nouvelles méthodes. Plus précisément, nous nous sommes appuyés sur les critères suivants :

- *Adaptabilité du système* : facilité de modifier les structures de données, les algorithmes et les types de paramètres.

- *Validité physique* : respect des lois qui régissent les transferts radiatifs, précision et disponibilité des grandeurs physiques calculées.

- *Généralité des scènes* : capacité à traiter des scènes de la vie réelle, comprenant beaucoup de primitives géométriques ainsi que des processus physiques variés et complexes.

- *Aspects pratiques* : rapidité des calculs, simplicité de d'utilisation, possibilité d'interface avec des systèmes externes de CAO ou de visualisation.

Après un tour d'horizon général de tous les travaux qui ont retenu notre attention (section 1), nous décrirons plus précisément les deux systèmes dont la structure nous semble la plus représentative : le banc d'essai de l'université Cornell (section 2), en tant que système procédural, et le projet VISION (section 3), en tant que système à objets. Pour terminer le chapitre, nous apporterons un éclairage complémentaire en présentant un environnement logiciel développé pour le domaine de la vision artificielle (section 4).

1 Tour d'horizon

Dans la revue suivante, nous avons choisi de répartir les systèmes en trois catégories, correspondant à trois étapes de la recherche en synthèse d'images. Dans un premier temps, les méthodes proposées par les chercheurs traitaient uniquement l'éclairage direct de la scène par les sources lumineuses (section 1.1). On dit que ces méthodes sont basées sur un *modèle local* d'illumination. L'architecture logicielle par excellence de cette catégorie est le *flot de données* (*dataflow*). Dans un deuxième temps, le réalisme fut amélioré de façon significative en ajoutant l'éclairage indirect dû aux échanges de lumière entre les éléments de la scène (section 1.2). Ce modèle d'illumination fut qualifié de *global*. Il donna lieu à de nombreuses bibliothèques de fonctions, ainsi qu'aux premiers systèmes à objets. Enfin, le besoin d'un réalisme encore meilleur a poussé les chercheurs à se fonder sur une simulation plus précise des propriétés physiques de la lumière (section 1.3). Ce mouvement vers la simulation des processus naturels a constitué une incitation supplémentaire, s'il en était besoin, à utiliser la programmation par objets.

Indépendamment de ce classement, les systèmes que nous allons citer évoluent entre deux tendances, selon l'usage auquel ils sont principalement destinés :

- *Essais d'algorithmes :* limités en ce qui concerne la complexité de leurs données, ces systèmes se distinguent par leur architecture interne modulaire [TLG91, Gla91, SS95].
- *Production d'images :* afin de pouvoir traiter une grande diversité de scènes, ces systèmes possèdent souvent des interfaces génériques pour décrire les modèles d'émission et de réflexion de la lumière ; c'est surtout à ce titre qu'ils nous intéressent [Ups90, War94, War95].

Pour conclure cette section, nous tenterons de faire le point sur la situation présente et nous justifierons le choix des systèmes qui seront présentés plus en détail dans les sections 2 et 3.

1.1 Modèle local

La première génération de logiciels de synthèse d'images s'est attachée à produire un rendu réaliste des objets modélisés par la CAO. Le défi principal résidait dans la multiplicité des primitives géométriques à représenter, et c'est cette multiplicité que les premiers systèmes se proposaient de traiter, à l'image du banc d'essai des laboratoires Bell [Whi82].

L'architecture REYES [CCC87], développée par la société *Pixar*, est un outil de production d'images. Elle est adaptée aux scènes de grande complexité et peut mettre en œuvre des modèles d'émission et de réflexion évolués. Dans la continuité de cette architecture, Pixar a ensuite spécifié l'interface RENDERMAN [Ups90], qui a instauré la scission entre les processus de modélisation et de rendu, et standardisé le dialogue entre ces deux processus.

Une autre approche envisagée très tôt est celle du *flot de données* (*dataflow*). Dans ce type de système logiciel, l'utilisateur construit son application en connectant des blocs de traitement par des bus de données. Il peut s'agir d'un simple *pipeline Unix*, comme dans FRAMES [PH87] ou de graphes plus complexes avec des connexions multiples, à la façon de GRAPE [NF87]. Ces systèmes offrent une certaine souplesse à l'utilisateur, et sont très utilisés dans les domaines de la visualisation scientifique et du traitement d'images [WRH92]. Ils excellent dans les applications où il faut appliquer des transformations successives à de gros volumes de données homogènes. Par

contre, ils conviennent moins à la structure itérative ou récursive des algorithmes d'illumination globale.

1.2 Lancer de rayons et radiosité

La prise en compte de l'illumination globale est apparue avec la méthode du lancer de rayons [Whi80], et fut l'objet de nombreux systèmes logiciels. Nous retiendrons notamment le premier banc d'essai de l'université Cornell [HG83], conçu pour aider les recherches sur les modèles d'illumination et de réflexion, le rendu des surfaces paramétriques, la propagation de la lumière et le plaquage de textures.

Le second banc d'essai de Cornell [TLG91] ajoute les méthodes de radiosité à son catalogue d'algorithmes, et constitue sans doute une des bibliothèques logicielles les plus complètes pour les calculs d'illumination globale. Ce système est particulièrement modulaire, donc très intéressant pour nos recherches, mais il utilise le langage *C* et souffre des lacunes d'adaptabilité propres à l'approche procédurale. Nous y reviendrons plus en détail dans la section 2.

Le *Ray Tracing Kernel* [KA88] est une des premières véritables architectures à objets. On y voit l'utilisation de classes abstraites pour uniformiser le traitement de différents types d'objets géométriques et permettre le choix de différents outils de rendu. Cet article a inspiré beaucoup de travaux ultérieurs, dont le *Ray Tracing Framework* [SSB91], qui intégre le lancer de rayons avec la méthode de Monte-Carlo.

Nous complèterons cette seconde série de systèmes avec l'architecture SPECTRUM proposée par Andrew Glassner [Gla91]. Comme la plupart des systèmes qui lui sont contemporains, SPECTRUM se focalise sur le lancer de rayons, la radiosité, et les méthodes hybrides. L'architecture est orientée objets et riche en classes abstraites. En dépit de tous ces atouts, sa flexibilité est compromise par une délégation excessive des calculs aux éléments de la scène. L'algorithme global de simulation s'en trouve dispersé et difficile à remplacer. Par ailleurs, la validité physique ne figure pas dans les objectifs de l'auteur, contrairement aux systèmes qui vont suivre.

1.3 Simulation physique

La dernière catégorie que nous voulons aborder, et celle qui nous concerne le plus, est celle des systèmes fondés sur une simulation physique des transferts lumineux. La méthode de radiosité constituait déjà un pas dans ce sens, à condition de la compléter par des modèles plus sophistiqués d'émission et de réflexion de la lumière. Dans cette évolution de la synthèse d'images vers une simulation plus rigoureuse des processus physiques, la méthode de radiosité a été replacée dans le cadre plus général de la *théorie des éléments finis*. En parallèle, le lancer de rayons a été couplé à des techniques de type Monte-Carlo, aussi appellées *méthodes stochastiques*.

Le logiciel RADIANCE [War94, War95] était initialement un outil de recherche, mais il a été utilisé avec succès dans de nombreux projets d'ingénierie de l'éclairage. Comme l'architecture de *Pixar*, RADIANCE repose sur un seul algorithme de rendu, mais cette fois l'algorithme est physiquement valide et prend en compte les interactions lumineuses entre surfaces. De plus, comme dans le cas de RENDERMAN, le langage de description de scène est suffisamment général pour pouvoir être utilisé avec les scènes les plus complexes, ce qui peut expliquer la grande popularité de RADIANCE. Le code source du logiciel est public et facile à obtenir, mais rien ne semble prévu pour faciliter la modification de l'algorithme global de simulation.

Enfin, le système VISION [SS95] constitue la proposition la plus récente, et celle qui répond le mieux aux critères que nous avions fixés. Totalement orienté objets, ce système accepte une grande diversité d'algorithmes. Aux méthodes stochastiques sont ajoutées les méthodes par éléments finis, qui généralisent la méthode de radiosité. Par ailleurs, la même équipe de chercheurs propose une extension de RENDERMAN permettant d'adapter cette interface à l'illumination globale [SPS95]. Ainsi, VISION s'avère également extensible en ce qui concerne la description des scènes.

1.4 Bilan

Les systèmes à flots de données possèdent l'architecture la plus appropriée pour un rendu ne considérant que l'éclairage local. Dans le cadre plus complexe de l'illumination globale, ces systèmes peuvent encore être utilisés avec profit pour la visualisation finale, ou pour le calcul de la visibilité.

Le lancer de rayons et la radiosité, qui ont introduit la prise en compte de l'illumination globale, ont aujourd'hui évolué vers les méthodes stochastiques et les éléments finis. Le système VISION incarne cette évolution de la synthèse d'images vers la simulation des processus physiques, ainsi que celle de l'informatique vers l'orientation objets. Nous reviendrons plus en détail sur VISION dans la section 3. Lors de la lecture ultérieure du chapitre 4, décrivant les travaux que nous menions à la même époque sur la plate-forme Graph'IS, le lecteur pourra apprécier la similitude de notre démarche avec celle des auteurs de VISION, et la similitude de nos résultats.

Auparavant, nous voulons revenir sur le second banc d'essai de Cornell [TLG91], qui nous semble être le système le plus complet et le plus modulaire, parmi ceux qui n'utilisent pas l'orientation objets.

2 Banc d'essai de l'université Cornell

Le banc d'essai présenté par Trumbore et al. [TLG91] a été utilisé pendant plusieurs années à l'université Cornell pour tester diverses méthodes de synthèse d'images. À l'époque de la publication, les auteurs voyaient l'avenir de la discipline dans l'illumination globale, la réflectance bidirectionnelle, les textures procédurales, et le parallélisme. Le banc d'essai a été conçu pour intégrer ou évoluer vers ces méthodes.

La figure 3.1 résume le fonctionnement d'ensemble du système. Les auteurs décomposent la production d'une image en deux phases successives, à la façon de RENDERMAN : modélisation géométrique puis rendu réaliste. La préparation des données géométriques est laissée à la charge de modeleurs externes, qui disposent de leurs propres structures de données et de leur propres modules de rendu en temps réel (*PHIGS+, structure d'affichage*). Lorsque le modèle géométrique est prêt et que l'on souhaite en faire un rendu de qualité, les données sont converties au format MID (*Description Indépendante du Modeleur* dans la figure, ou *Modeler Independent Description* dans l'anglais original) et transmises au banc d'essai. Le format MID sert de passerelle unique entre les modeleurs et les programmes de rendu appartenant au banc d'essai. Dans un deuxième temps, les programmes de rendu calculent des images et les stockent dans un format également unique et imposé.

La réalisation des programmes de rendu est basée sur une bibliothèque de modules logiciels qui

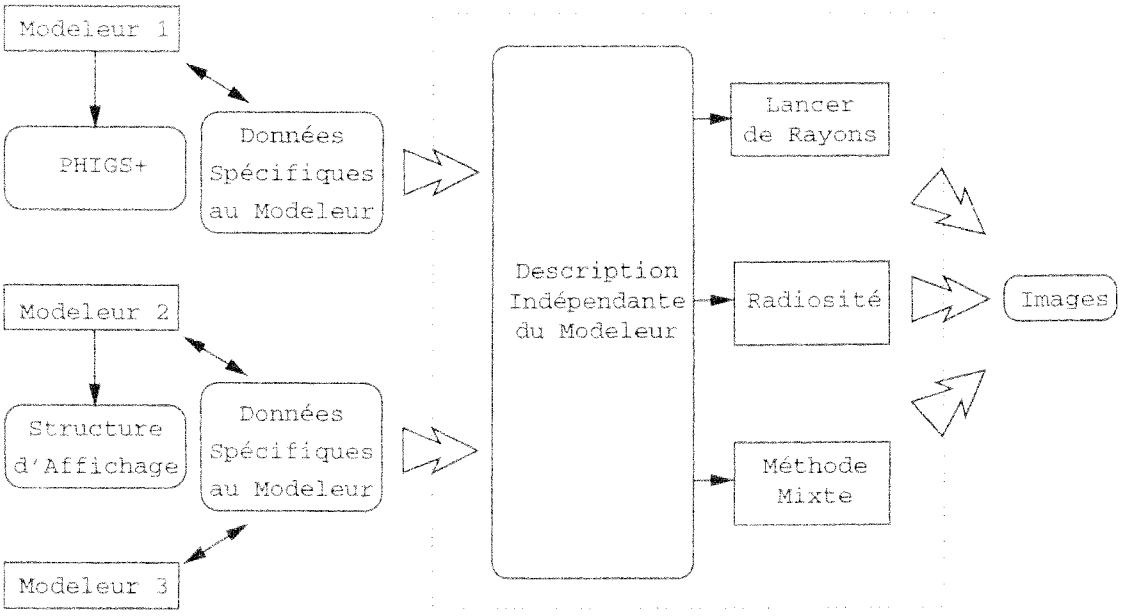


FIG. 3.1 – *Fonctionnement global du banc d'essai de Cornell*

couvrent la majorité des fonctions nécessaires. Les exemples d'implantation donnés par les auteurs reflètent l'état de l'art du début des années 90 : un algorithme de radiosité standard [CCWG88], un algorithme de lancer de rayon, et un algorithme hybride en deux passes. L'ensemble du banc d'essai regroupe environ 10000 lignes de code en langage *C*.

Les deux sections suivantes sont consacrées aux deux éléments clés du banc d'essai : le format MID (2.1), et la structure logicielle interne à base de modules (2.2). Nous discuterons ensuite les limites du système (2.3).

2.1 Format MID

Un fichier au format MID est un fichier ASCII contenant un ensemble de primitives, lesquelles peuvent être combinées à l'aide d'opérateurs d'union, d'intersection ou de différence.

Chaque primitive est décrite par une étiquette de type et une liste d'attributs : des couples [nom/valeur] dont la valeur peut être un nombre scalaire, une chaîne de caractères, ou le nom d'un fichier externe (dans un format différent). Pour chaque type de primitive, trois sortes d'attributs sont prédéfinis par le banc d'essai :

- données géométriques (rayon d'une sphère, sommets d'un objet facettisé, etc),
- transformations (passage du repère de la primitive au repère de la scène),
- informations de rendu (propriétés d'émission et de réflexion).

Un module spécialisé de l'architecture permet de lire les fichiers MID. À partir de ces informations de base, les modules de rendu peuvent construire leurs données propres, très variables

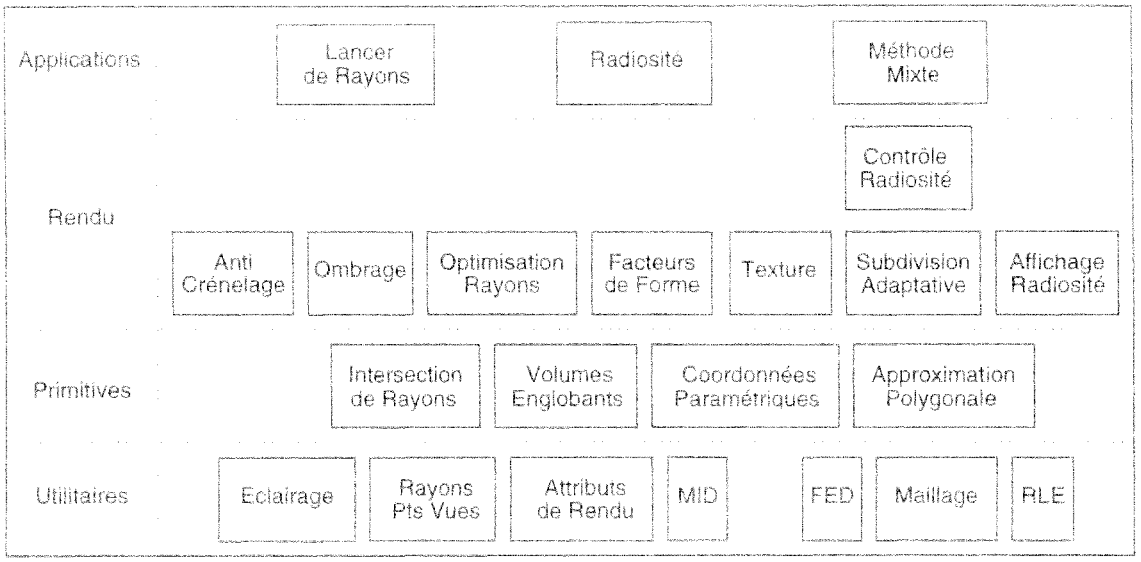


FIG. 3.2 - *Modules du banc d'essai de Cornell*

d'un algorithme à l'autre. Lorsqu'un nouvel algorithme est implanté, on définit au besoin de nouveaux attributs. MID étant simple et extensible, on peut utiliser de vieilles bases de données avec de nouveaux programmes de rendu, et vice versa. Lorsqu'un module de rendu parcourt une description MID, il ignore simplement les attributs qu'il ne reconnaît pas.

Pour que la taille des scènes ne soit pas limitée par la mémoire de l'ordinateur, il faut que les procédures qui interprètent les fichiers MID procèdent de façon séquentielle, en ne conservant en mémoire que les données en cours d'interprétation. Quand des modèles géométriques peuvent être définis et réutilisés n'importe quand, il faut que ces modèles résident en mémoire durant toute la lecture des fichiers. Pour cette raison, la grammaire MID ne permet pas que les primitives déjà lues puissent servir de modèles aux primitives suivantes. Les seules données réutilisables sont celles qui ont été reléguées dans des fichiers externes, dont c'est justement la principale utilité.

2.2 Bibliothèque de modules

Les modules sont répartis en trois couches, auxquels il faut ajouter la couche des applications proprement dites. Les modules d'une couche donnée peuvent utiliser les modules de la même couche et des couches inférieures. Les trois couches sont les suivantes :

- `Utilitaires` : ces modules facilitent la manipulation des structures de données, et apportent des fonctionnalités mathématiques de base. Il s'agit de fonctionnalités indispensables et standards, pas spécialement susceptibles d'évoluer ou de faire l'objet de recherches. Pour chaque fonctionnalité on dispose d'une seule implantation optimisée.

- `Primitives` : c'est ici que sont implantées les fonctionnalités que doivent remplir les différents types de primitives. Au lieu d'utiliser un module pour chaque type de primitives, les auteurs utilisent un module pour chaque fonctionnalité. Il est prévu de créer de nouvelles fonctionnalités plutôt que de nouvelles primitives, ce qui justifie leur choix.

- `Rendu` : il s'agit des fonctionnalités de synthèse d'images proprement dites. Les interfaces de ces modules sont spécifiées très soigneusement, afin de pouvoir choisir facilement entre plusieurs implémentations d'une même fonction. Ce sont ces modules dont on veut pouvoir comparer les performances et les résultats.

2.3 Discussion

Les auteurs ont bien souligné la nécessité de définir soigneusement un format de description de scène générique. D'autre part, le découpage en couches logicielles introduit des distinctions intéressantes, entre une couche utilitaire nécessaire mais pas forcément évolutive, une couche dédiée aux primitives de la scène, et une couche très évolutive contenant les algorithmes de rendu. Ces mêmes distinctions pourraient être appliquées aux classes d'une architecture à objets.

Le découpage en modules est également instructif. Il permet d'identifier certaines tâches que devront accomplir les objets. Ce découpage est basé sur les étapes des algorithmes et les fonctions à calculer, ce qui est légitime lorsqu'on utilise un langage de programmation procédural, mais conduit à des modules fortement couplés les uns aux autres. Par ailleurs, nous avons expliqué dans le chapitre 2 que les bibliothèques modules ne sont pas adaptables (il faut dupliquer un module pour pouvoir le modifier sans gêner les autres utilisateurs).

Enfin, soulignons la séparation nette entre le processus de modélisation et le processus de rendu, qui est destiné à produire des images de haute qualité pour un modèle figé. Il n'est pas question d'animation ou d'interactivité, mais seulement de réalisme. Nous tenterons d'écarter cette restriction dans le chapitre 5.

Après avoir illustrer l'approche procédurale et modulaire, nous allons maintenant présenter un système logiciel plus récent, utilisant l'orientation objets.

3 VISION

L'architecture VISION, proposée par Philipp Slusallek et Hans-Peter Seidel [SS95], est conçue pour supporter les algorithmes d'illumination globale les plus divers, depuis Monte-Carlo jusqu'aux éléments finis. La démarche des auteurs est assez voisine de celle que nous avons nous-même suivie, comme en témoignent leurs objectifs initiaux :

- *Validité physique* : les valeurs manipulées par les algorithmes doivent correspondre à des grandeurs physiques cohérentes ; les anciens algorithmes doivent être restreints aux cas qui ne constituent pas des violations de la loi de conservation de l'énergie.
- *Contrôle de la précision* : l'architecture ne doit pas limiter intrinsèquement la précision des calculs, mais fournir à l'utilisateur le moyen de la contrôler, afin de pouvoir s'adapter aux exigences différentes de la simulation et de la production d'images.
- *Flexibilité* : il est souhaitable de pouvoir choisir entre plusieurs algorithmes pour résoudre un sous-problème donné, dans le but de les comparer ou de mieux s'adapter à la spécificité du sous-problème.
- *Modularité* : afin de fournir la flexibilité requise, l'architecture doit être décomposée en sous-parties autonomes ; pour limiter l'inter-dépendance de ces sous-parties, la décomposition

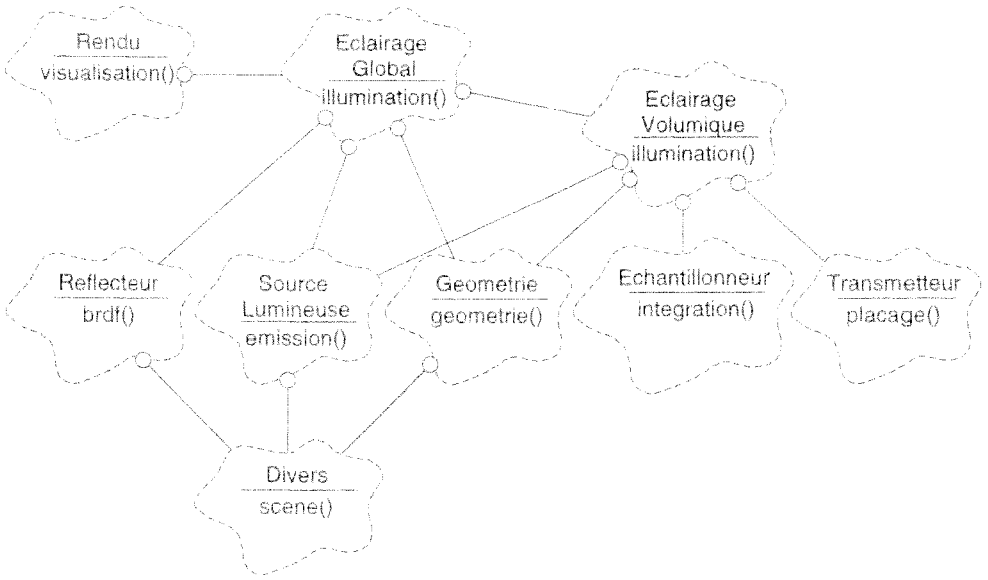


FIG. 3.3 – *Sous-systèmes de VISION*

doit reposer sur les processus physiques plutôt que sur les algorithmes.

- *Architecture générique* : plus qu'une simple collection d'algorithmes, VISION doit fournir une architecture prédéfinie, dont les sous-parties sont spécifiées par leurs responsabilités et leur interfaces ; la définition des interfaces ne doit pas restreindre le champ des implantations possibles, ce qui amène parfois à choisir des interfaces plus génériques et moins performantes.

- *Facilité d'utilisation* : l'architecture VISION n'est pas une fin en soi ; pour être un soutien aux programmeurs comme aux utilisateurs finaux, son usage doit être simple.

- *Productivité* : il doit être possible de traiter des scènes de grande taille et de grande complexité, ainsi que de produire des images de bonne qualité dans des temps raisonnables.

Afin de concilier ces objectifs parfois divergents, les auteurs ont utilisé l'orientation objets. En se fondant sur les processus physiques de propagation de la lumière, ils ont définis des sous-systèmes que nous allons présenter (3.1). La diversité des algorithmes mis en œuvre est très convaincante (3.2). La facilité d'utilisation et la productivité du système sont moins clairement établies. Après quelques informations concrètes sur l'implantation de VISION (3.3), nous discuterons les limites de ce système (3.4).

3.1 Sous-systèmes principaux

L'architecture VISION, comme toute architecture à objets, repose sur un ensemble de classes et sur leurs interfaces respectives. Les classes qui concourent à remplir la même tâche sont regroupées en sous-parties que les auteurs appellent *sous-systèmes*. Les principaux sous-systèmes de VISION sont présentés dans la figure 3.3.

L'architecture se caractérise par une distinction entre les sous-systèmes locaux, chargés de fournir les caractéristiques locales des éléments de la scène (`Reflecteur`, `SourceLumineuse`, `Geometrie`), et un sous-système global (`Eclairage`), chargé de combiner les informations locales afin de calculer les transferts d'énergie. On peut également remarquer la distinction entre le calcul de l'illumination globale (`Eclairage`) et le calcul des images proprement dites (`Rendu`), ainsi que la présence de trois sous-systèmes pour la prise en compte des volumes participants (`ElairageVolumique`, `Echantillonneur`, `Transmetteur`).

SourceLumineuse

Ce sous-système sait évaluer le flux lumineux dû à une source et fournir les caractéristiques de ce flux à l'endroit où il est émis ou bien à l'endroit où il est reçu. Selon l'approche générale adoptée pour simuler l'illumination, on peut utiliser l'une des trois interfaces possibles du sous-système :

- Calcul de l'effet : les méthodes de cette interface fournissent l'éclairement produit par la source en un point donné de la scène, en faisant l'hypothèse que cette source voit parfaitement le point.
- Échantillonnage : la source est représentée par un ensemble de points et de directions d'émission accompagnées de leurs densités de probabilité, ce qui permet d'exploiter cette source dans les méthodes stochastiques.
- Élements finis : la source est représentée sous forme d'éléments de surface et de leurs radiosités, ce qui permet de l'intégrer directement dans une structure d'éléments finis.

Ajoutons que pour la première interface, les méthodes ne renvoient pas directement l'éclairement mais une liste d'instances de la classe `EchantillonLumiere`. On pourra tester séparement la visibilité de chaque échantillon avant de prendre en compte son éclairement. La présence de ces objets intermédiaires augmente le nombre de classes, mais permet de simplifier l'interface de `SourceLumineuse`. Ils ajoutent un degré de flexibilité dans l'architecture, et peuvent abriter des précalculs améliorant les performances.

Reflecteur

Ce sous-système est responsable du calcul de la réflectance bidirectionnelle. Comme dans le cas précédent, on dispose de trois interfaces :

- BRDF : les méthodes fournissent directement la réflectance bidirectionnelle en fonction des directions de réception et de réémission passées en paramètres.
- Échantillonnage : pour une direction de réception donnée, un ensemble de directions de réflexion privilégiées est établi et retourné avec les réflectances et les densités de probabilité correspondantes.
- Éléments finis : la réflectance est approximée par une moyenne ou un ensemble de valeurs avec leurs dérivées partielles, indépendamment de la base de fonctions de l'élément de surface.

Geometrie

Ce sous-système est responsable de la représentation géométrique des surfaces dans la scène. Trois types de requêtes peuvent être adressées à une surface, à travers trois interfaces :

- Encombrement : le volume englobant d'une primitive peut être obtenu et utilisé pour accélérer certains calculs géométriques.
- Échantillonnage : cette interface fournit des points échantillons, et toutes les informations associées comme les coordonnées spatiales et paramétriques, les vecteurs tangents et normaux, ...
- Subdivision : la primitive est représentée comme un ensemble d'éléments de surface, éventuellement hiérarchisés ; cette fonctionnalité complexe est primordiale pour les algorithmes par éléments finis.

Dans la seconde interface, il serait peu pratique de fournir toutes les informations à la fois, mais inefficace de les fournir séparément (il y a beaucoup de redondance dans le calcul de ces informations). Comme dans la première interface de `SourceLumineuse`, l'utilisation d'un objet intermédiaire encapsulant certains précalculs, ici `InfoSurface`, constitue un compromis réussi.

Eclairage

Il s'agit du sous-système global, responsable du calcul de la propagation de la lumière. Après avoir fourni à `Eclairage` l'ensemble des éléments composant la scène, on l'active en lui demandant la valeur de l'éclairement en un point donné de la scène. Il peut également être indirectement sollicité par les sous-systèmes locaux lors du calcul des inter-réflexions. Pour permettre ces requêtes, `Eclairage` possède une interface semblable à celle des sources lumineuses, et retourne donc des instances d'`EchantillonLumiere`.

Une phase optionnelle de précalcul est définie dans l'interface d'`Eclairage`. Ce précalcul pourrait être caché dans les autres méthodes, mais le faire apparaître explicitement dans l'interface permet d'accorder un meilleur contrôle à l'utilisateur.

On retrouve dans VISION certaines pratiques facilitant le traitement des grandes scènes et l'accélération des calculs :

- la description de la scène peut-être subdivisée en sous-parties, afin de permettre un calcul réparti de l'illumination,
- il est possible de déclarer certaines surfaces comme passives, auquel cas elles ne rediffusent pas la lumière,
- une méthode est chargée de fournir une représentation de la scène adaptée aux cartes graphiques 3D.

Enfin, pour supporter la modification incrémentale de la scène, le sous-système fournit une interface pour l'ajout, la suppression ou la modification des éléments de la scène. Les calculs d'illumination déjà effectués ne sont pas automatiquement remis à jour. Afin de préserver les performances, cette mise à jour n'est réalisée que sur une demande explicite de l'utilisateur.

Divers

Terminons cette revue des principaux sous-systèmes de VISION par `Divers`, qui est notamment chargé de fournir la description de scène utilisée par les autres sous-systèmes.

Les données sont initialement disponibles sous forme de fichiers, dans une version étendue du format RENDERMAN, supportant les besoins de l'illumination globale [SPS95]

Dans cette description, les surfaces sont organisées en arbre, et chacune d'entre elles est associée à un objet `Reflecteur` et optionnellement à un objet `SourceLumineuse`.

3.2 Algorithmes implantés

En utilisant l'architecture ci-dessus, les auteurs ont implanté les algorithmes d'illumination globale les plus connus et les plus récents. Dans tous ces algorithmes, ils évaluent la visibilité à l'aide d'un lancer de rayon, seule méthode capable de prendre facilement en compte les éventuels volumes participants. Les algorithmes sont regroupés en deux familles : les méthodes Monte-Carlo et les méthodes par éléments finis. Après quelques mots sur les méthodes Monte-Carlo, nous allons détailler le déroulement d'une méthode par éléments finis.

Méthodes de type Monte-Carlo

Les trois algorithmes implantés sont le *Path Tracing* [Kaj86], le *Bidirectionnal Estimators* [VG94] et l'*Irradiance Caching* [WRC88, WH92, War94]. La dernière méthode (empruntée à RADIANCE) combine une approche de Monte-Carlo pour l'éclairage direct et un système de cache pour l'éclairage indirect. Elle tire particulièrement parti de la modularité de l'architecture.

Méthodes par éléments finis

La figure 3.4 donne un exemple de scénario pour un calcul par éléments finis. Ces méthodes se décomposent toujours en cinq étapes :

1. le choix d'une base de fonctions pour représenter l'illumination d'une surface,
2. le maillage de la scène en éléments de surfaces,
3. la construction du système linéaire représentant les relations entre les éléments de surface,
4. la résolution du système,
5. la reconstruction de l'illumination pour les points visibles sur les images.

Selon les variantes, ces différents calculs sont intégralement effectués, ou bien ils sont évalués par raffinements successifs. Le maillage, par exemple, pourra être réalisé une fois pour toute en début de processus ou être précalculé de manière grossière puis affiné là où cela s'avère nécessaire. Un sous-système particulier, `Subdivision`, est chargé de la gestion de ce maillage.

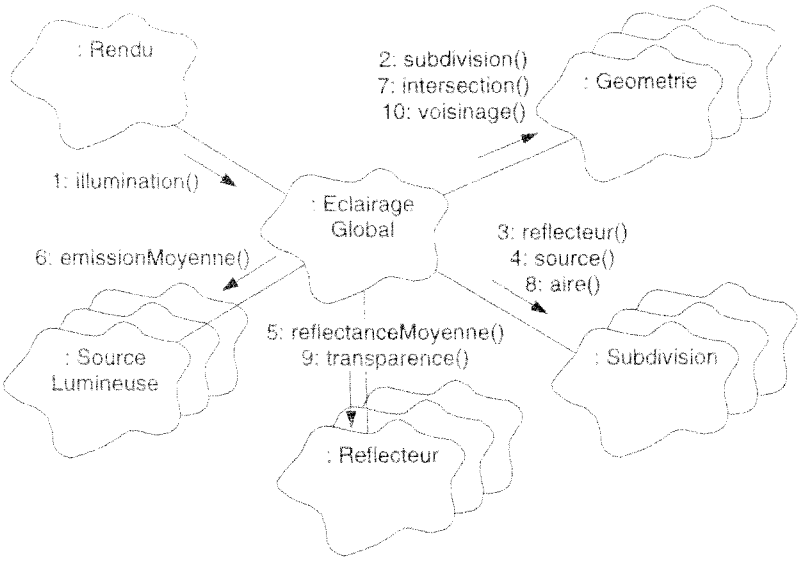


FIG. 3.4 – *Étapes d'une méthode par éléments finis*

Lors de l'initialisation du système linéaire, les sources lumineuses surfaciques sont remplacées par des éléments finis équivalents, dont les radiances sont évaluées par moyennage ou échantillonnage. Les radiances initiales des autres éléments sont le fruit de l'éclairage direct des sources non surfaciques.

Par ailleurs, contrairement à la pratique courante, les radiosités ou les éclairements calculés ne sont pas stockés avec les primitives, mais dans une structure de données séparée, au sein du sous-système *Eclairage*. En effet, la nature et la structure de ces données est spécifique à chaque algorithme de simulation.

Trois algorithmes ont été implantés : la radiosité hiérarchique [HSA91], la radiosité par ondelettes [GSCH93] et la radiance par ondelettes [Sch94].

3.3 Le système logiciel VISION

L'implantation de tous les algorithmes ci-dessus, en langage C++, a permis de construire un système logiciel comprenant environ 250 classes, ainsi réparties :

- classes abstraites : 40 ;
- sous-systèmes d'illumination : 140 ;
- interface avec RENDERMAN : 40 ;
- utilitaires : 30.

La hiérarchie des classes forme une « forêt » (Il n'y a pas de classe racine *Object* en C++). Orthogonalement à cette forêt, le système est découpé en couches : utilitaires généraux, utilitaires d'illumination, classes abstraites d'illumination, classes concrètes d'illumination, classes RENDERMAN, et classes du langage de configuration.

La flexibilité de VISION aurait pu alourdir son interface, car il faut donner à l'utilisateur le moyen de choisir parmi tous les algorithmes disponibles. Pour contourner cette difficulté, les auteurs ont ajouté à VISION un langage de configuration, basé sur le langage TCL [HSS94].

3.4 Discussion

L'utilisation des technologies objets donne à VISION une flexibilité largement supérieure à celle du banc d'essai de Cornell. Au lieu de réaliser des modules logiciels prêts à l'emploi, les auteurs de VISION ont porté leur effort sur la définition d'interfaces génériques, en se basant sur les processus physiques qu'ils cherchent à simuler.

Par opposition à SPECTRUM, où toutes les responsabilités sont déléguées aux éléments de la scène, VISION fait la distinction entre les sous-systèmes locaux et le sous-système global qui les coordonne. L'algorithme global se trouve moins dispersé, il est donc plus facile de le remplacer.

En implantant des algorithmes nombreux et variés, les auteurs ont démontré la souplesse de leur architecture. En ce qui concerne la description de la scène, ils ont proposé une extension au langage RENDERMAN, pour l'adapter aux besoins de l'illumination globale.

Nous avons dans l'ensemble une excellente opinion de VISION. Nous émettrons seulement quelques réserves :

- Il y a somme toute peu de classes consacrées aux tâches algorithmiques essentielles, comme le calcul de la visibilité.
- Malgré la présence de nombreux algorithmes indépendants du point de vue, le sous-système qui déclenche l'ensemble des calculs reste celui qui est chargé de produire les images finales.
- VISION s'inscrit dans le schéma introduit par RENDERMAN, selon lequel le rendu est une étape venant après le gel de la modélisation. Nous aimerions évoluer vers un schéma où modélisation et rendu seraient plus interactifs.
- Les auteurs de VISION n'ont pas prévu de format de sortie standard pour décrire les primitives d'une scène et leurs luminances, et on ne peut pas tirer pleinement profit des résultats. On aimerait par exemple pouvoir alimenter un système de réalité virtuelle, ou permettre à un utilisateur de naviguer dans la scène et de consulter interactivement les éclairements.

4 Image Understanding Environment

L'utilisation de l'orientation objets pour développer un système logiciel flexible n'est pas réservée au seul domaine de la synthèse d'image. Il nous semble intéressant d'évoquer maintenant les efforts menés au sein du projet IUE.

L'agence de défense américaine ARPA a initié en 1990 ce qui allait devenir IUE (*Image Understanding Environment*), un projet de grande envergure visant à créer un environnement logiciel pour la recherche en vision artificielle. Deux ans de spécifications, associant des chercheurs américains, européens et japonais [M+92], ont permis de préciser les structures de données et

les opérations que l'environnement IUE devait supporter. Les principaux points retenus sont les suivants :

- manipulation interactive d'images nombreuses et de grande taille,
- utilisation de pixels de tous types (entiers, réels, complexes, triplets colorimétriques, spectres),
- gestion de séquences d'images de longueur quelconque,
- modélisation de tous types de capteurs (caméras perspectives ou stéréos, satellites),
- utilisation de modèles 3D volumiques ou surfaciques,
- importation et exportation des modèles 3D au format IGES.

L'environnement IUE est développé depuis trois ans par plusieurs universités et par la société *Amerinex Applied Imaging* [KM94, D+96, Ler97]. Les moyens et les objectifs du projet sont ambitieux. IUE doit notamment améliorer :

- la *productivité de la recherche*, grâce à la spécification d'une interface à objets pour le développement et le partage des logiciels de vision artificielle,
- le *transfert de technologie*, à travers une plate-forme démontrant les bénéfices des algorithmes de vision artificielle dans différents contextes applicatifs,
- l'*enseignement*, en fournissant un format cohérent et standard pour décrire les algorithmes.

Nous ne décrirons pas ici les classes d'IUE qui sont spécifiques à la vision artificielle (`Images`, `Spatial Objects`, `Image Features`, `Coordinate Systems and Transforms`, `Spatial Indices`, `Sensors`). Nous présenterons plutôt quelques éléments de conception générale, et nous tenterons d'en tirer des enseignements pour nos propres travaux.

4.1 Gestion des données

La plupart des applications de vision artificielle incluent la manipulation de données nombreuses et complexes (images, modèles en trois dimensions). On ne sera donc pas surpris de trouver dans IUE des fonctionnalités qui évoquent les systèmes de gestion de bases de données (SGBDs).

Par exemple, les auteurs des spécifications ont exprimés le besoin de ne jamais figer la structure de données des objets, et de pouvoir leur adjoindre de nouveaux attributs à tout moment, y compris pendant l'exécution d'une application. Cette gestion dynamique des attributs tend à détériorer les performances et le contrôle du typage, mais autorise en contre-partie l'implementation générique de fonctionnalités puissantes. Notamment, dans un graphe d'objets, on peut définir des règles de valeurs par défaut et d'héritage d'attributs. On peut également implémenter la recherche des objets répondant à des critères portant sur la valeur de leurs attributs. Cette dernière fonctionnalité, apparentée au traitement des requêtes dans les SGBDs, peut être accélérée avec les procédés utilisés dans ce domaine.

Les concepteurs d'IUE ne s'intéressent pas au *partage* des données, tel qu'il est habituellement proposé dans les SGBDs. Par contre, ils accordent une grande importance à l'*échange* de ces données. Ils veulent pouvoir sauvegarder dans un fichier l'état des objets à un instant donné, puis pouvoir les reconstruire dans un contexte différent. Pour y parvenir, ils ont choisi d'utiliser des fichiers textuels et une syntaxe proche de LISP. les fichiers doivent être autosuffisants, c'est-à-dire contenir toute l'information nécessaire pour reconstruire intégralement une structure d'objets et leurs attributs.

4.2 Représentation des algorithmes

On qualifie de *tâche* un algorithme qui agit sur un nombre significatif de pixels. Par exemple, une fonction appliquant un noyau de convolution à une image est une tâche, alors qu'une fonction appliquant un cœfficient multiplicatif à un pixel isolé ne l'est pas. Chaque tâche est représentée par un objet à part entière plutôt que par une simple méthode. Plusieurs arguments sont avancés pour soutenir ce choix.

D'une part, les algorithmes de vision ont des structures de données internes complexes, ce qui est déjà une raison amplement suffisante pour en faire des classes. Par ailleurs, ces algorithmes ont également de nombreux paramètres, et une grande partie des recherches en vision artificielle consiste à explorer les valeurs de ces derniers. Il est plus facile de manipuler plusieurs jeux de paramètres si ces derniers constituent les attributs d'un objet représentant l'algorithme.

D'autre part, la recherche en vision artificielle implique beaucoup de calculs et la génération de beaucoup de résultats. Il est important de pouvoir contrôler l'historique des calculs. Un objet de type tâche est un lieu approprié pour stocker des informations sur l'éxécution de l'algorithme correspondant.

Enfin, la notion de tâche permet de représenter un processus de vision sous la forme d'un *flot de données*. Il est en effet aisé de traduire un processus de vision artificielle par un flot de données dont les blocs correspondent aux tâches du processus.

4.3 Recouvrement avec la synthèse d'images

La vision artificielle fait de plus en plus appel à une modélisation en trois dimensions des scènes visualisées et des capteurs à l'origine des images. Les spécifications d'IUE sur ce thème sont classiques et présentent relativement peu d'intérêt pour nous.

Une classe `Scene` permet de gérer un arbre CSG d'objets ainsi qu'une liste de sources lumineuses et de capteurs. La classe `Renderer` permet de demander l'énergie émise dans une ou plusieurs directions, avec une intégration sur l'angle solide du pixel et éventuellement sur les longueurs d'ondes. Une sous-classe a été prévue pour chaque type de rendu : filaire, faces cachées, ombré, avec *z-buffer*, par lancer de rayon, ...

4.4 Langages et outils de programmation

Les deux langages de programmation retenus sont CLOS, comme langage général, et C++ pour la performance et la portabilité. CLOS est jugé intéressant parce qu'il bénéficie de la flexibilité de tous les langages derivés de LISP. Il est relativement simple d'implanter le mécanisme

d'attributs dynamiques dont nous avons déjà parlé. Deux méthodes `put` et `get` peuvent être implantées une fois pour toute dans la classe `IUEObject` dont héritent toutes les classes d'IUE. Par ailleurs, l'utilisation de méta-classes permet l'implantation unique de quelques opérations fondamentales comme les duplications d'objets ou les tests d'égalité. L'utilisation de C++ est normalement réservée aux cas suivants :

- calculs intensifs,
- interfaçage avec le système d'exploitation, les bibliothèques graphiques, les matériels spécialisés,
- récupération de code C++ déjà disponibles.

Initialement, il était prévu de débuter le développement en CLOS, puis d'y inclure progressivement des portions de C++. Dans la pratique, tous les développements actuels ont été réalisés en C++, et l'utilisation de CLOS devient de plus en plus hypothétique. Ce revirement peut s'expliquer par la progression du C++ dans le milieu industriel ou par les difficultés d'interconnexion entre CLOS et C++ (problèmes de gestion de mémoire).

L'abandon de CLOS est compensé par l'utilisation d'un outil de génération automatique du C++ à partir des spécifications. Cette approche pourrait marier la souplesse de l'interprétation (coté spécifications) et les performances de la compilation (coté C++), mais l'ergonomie et la robustesse de ce type d'outils restent à vérifier.

Par ailleurs, l'environnement IUE se veut complet, couvrant toutes les couches d'un système logiciel depuis la programmation de bas niveau jusqu'à l'interface avec l'utilisateur, et couvrant aussi tous les outils susceptibles d'aider à la préparation et à l'exploitation de ce système : éditeurs, debogueurs, kits de construction d'interfaces, analyseurs de performances. L'utilisation d'outils commerciaux soigneusement choisis est jugée préférable au développement interne d'outils sur mesure.

4.5 Discussion

Les chercheurs impliqués dans le projet IUE ont consacré beaucoup de temps à la spécification formelle des classes. Grâce à l'outil de traduction automatique en C++, ces spécifications se répercutent intégralement sur le code, et sont censées évoluer facilement. L'utilisation de spécifications détaillées permet de définir les interfaces caractéristiques du domaine de la vision artificielle, et constituent une alternative intéressante à l'utilisation de classes abstraites (cf. chapitre 2). Cette pratique permet d'utiliser des mécanismes indépendants et plus élaborés que ceux du langage de programmation, mais impliquent l'apprentissage d'une terminologie supplémentaire. Par ailleurs, l'évolution des spécifications et du code correspondant reposent en grande partie sur l'outil de traduction automatique et ces outils ne sont pas réputés comme étant fiables. Nous avons vu que CLOS, initialement prévu comme langage principal, n'est toujours pas utilisé. On retrouve néanmoins une saveur LISP prononcée dans les règles de spécifications ou la notion d'attributs dynamiques. En fin de compte, un besoin se dégage assez clairement : celui d'un langage standard de haut niveau pour décrire les interfaces d'un système logiciel. Dans la plate-forme Graph'IS, nous avons choisi préféré pour le moment nous en tenir à l'utilisation de classes abstraites C++.

L'architecture générale d'IUE est proche du concept de *flots de données* que nous avions abordé dans le tour d'horizon. Cela se traduit par une séparation assez nette entre deux catégories de classes : celle qui encapsulent des données et celles qui encapsulent des algorithmes (tâches). Le contexte de la synthèse d'images réalistes semble différent. Il s'agit de simuler un processus physique, il paraît donc approprié de créer des classes simulant les éléments qui interviennent dans ce processus. Cela conduit à une architecture plus proche de la philosophie objet, où données et opérations sont très mélangées. Il faut pourtant admettre les bénéfices du concept de tâche. Parce que les algorithmes existent en tant que tel dans le système logiciel, il est plus aisé de les modifier et de les contrôler.

Comme dans tous les projets où les chercheurs se soucient du transfert industriel, l'environnement IUE est doté de nombreux outils logiciels complémentaires, ainsi que d'un standard pour l'échange de données. L'échange de données est la première étape vers la collaboration de chercheurs et d'industriels dans le domaine logiciel. Ce fut un des axes principaux d'IUE pendant la première année de développement.

Actuellement, la société *Ameinex* et ses collaborateurs universitaires ont réalisé les classes de base de leur environnement et travaillent sur l'implantation des algorithmes standards de vision artificielle. Ils ne semblent pas en retard sur le planning établi il y a plusieurs années, ce qui est suffisamment inhabituel pour être souligné.

5 Conclusion

Les systèmes de synthèse d'images à structure monolithique sont très productifs. Ils possèdent souvent de nombreuses options qui leur permettent de traiter une très grande diversité de scènes, mais ces options ne permettent pas d'agir en profondeur sur les algorithmes. Les systèmes par flots de données, beaucoup plus flexibles, s'avèrent inadaptés au calcul de l'illumination globale. Le banc d'essai de Cornell constitue le premier système véritablement flexible, et reste un cas exemplaire de structure modulaire. Les seules limites de ce banc d'essai sont imputables à sa conception procédurale. En s'appuyant sur l'orientation objet, l'architecture VISION constitue une avancée sensible. C'est cette même approche que nous avons adoptée pour nos recherches, et qui nous a conduit indépendamment à des résultats comparables (chapitre 4).

L'utilisation de la programmation objet semble aujourd'hui aller de soi. Toutes les tentatives faites en ce sens s'avèrent prometteuses. Il reste cependant plusieurs questions sans réponse. Au cœur du débat se trouve le choix des classes et la répartition des responsabilités. SPECTRUM est fondé sur une délégation maximale aux éléments qui composent la scène, alors que VISION maintient une classe centrale pour gérer les échanges lumineux. L'utilisation de classes pour les algorithmes bouscule l'idée que l'on se fait habituellement de la programmation par objets. Nous pensons que la représentation des algorithmes par des classes est indispensable si on souhaite étudier, comparer et changer ces algorithmes, sans que cela constitue une « régression » vers la programmation procédurale.

VISION, le banc d'essai de Cornell, l'environnement IUE sont maintenant implantés en C++. Ce langage nous semble incontournable pour la réalisation d'un système opérationnel que l'on souhaite diffuser. Cependant, C++ est notoirement trop permissif et de trop bas niveau pour permettre une conception robuste et la spécification des interfaces. Plusieurs solutions ont été adoptées par les uns et les autres : la notion de classe abstraite, un langage de plus haut niveau, un langage de spécification ou encore un outil de modélisation graphique. Personnellement, nous

nous en sommes tenu aux classes abstraites, limitées par le langage de programmation mais mieux intégrées au système.

Tous les chercheurs qui se soucient du transfert industriel ont pris soin de créer des environnements logiciels complets et capables de traiter des scènes complexes. La standardisation de l'échange des données est toujours considérée comme cruciale et passe le plus souvent par un format de fichier ou un langage évolué de description de scène : MID à Cornell, RENDERMAN étendu pour VISION. L'évolutivité de ce format conditionne pour une grande part celle du système logiciel entier.

Vis-à-vis des critères que nous avions définis au début de ce chapitre, la disponibilité des grandeurs physiques calculées et l'ouverture des systèmes restent insatisfaisantes. D'une part, l'interaction avec les modeleurs est à un seul sens. D'autre part, même lorsqu'ils sont fondés sur une simulation physiquement valide et indépendante du point de vue, les systèmes que nous avons passés en revue ne donnent pas directement accès aux résultats de leur simulation. Ils ne savent produire que des images, et ne peuvent pas servir à alimenter un outil de navigation en 3D. Ces réserves s'appliquent tout autant au premier système que nous avons nous-même réalisé, et qui fait l'objet du prochain chapitre (4). Nous proposerons dans le dernier chapitre (5) une architecture plus appropriée.

Chapitre 4

Plate-forme Graph'IS

L'intitulé initial de cette thèse était le suivant : « recherche et développement d'un prototype logiciel de synthèse d'image, fondé sur un algorithme général de simulation des transferts lumineux, appliqué aux scènes réelles ». Il s'agissait de concevoir un système logiciel qui permette de mieux partager et réutiliser les développements des membres de notre équipe de recherche et qui facilite le transfert de nos résultats à Electricité de France, qui a co-financé et co-encadré ce travail. Les termes utilisés dans l'intitulé traduisent les préoccupations du moment au sein de l'équipe :

« recherche et développement d'un prototype logiciel »

> La thèse ayant une vocation industrielle, elle devait se concrétiser à travers une application, en particulier par un logiciel d'aide à la conception destiné aux éclairagistes d'Électricité de France.

« fondé sur un algorithme général »

> Par « algorithme général » nous entendions un algorithme qui soit suffisamment flexible pour traiter une grande variété de situations avec une grande variété de méthodes. Cette recherche de flexibilité est à l'origine de notre choix d'utiliser les méthodes et les outils *à base d'objets*.

« de simulation des transferts lumineux »

> Notre démarche était de respecter avant tout les lois physiques de transmission de la lumière, et de privilégier ainsi la validité physique de tous les calculs intermédiaires plutôt que le seul réalisme des images finales. Afin de disposer de résultats indépendants du point de vue, nous nous sommes tournés vers les méthodes basées sur la radiosité plutôt que vers celles inspirées du lancer de rayons.

« appliqué aux scènes réelles »

> Notre équipe collabore étroitement avec un laboratoire d'architecture, et nos logiciels devaient pouvoir être utilisés par des architectes sur des projets et des scènes en vraie grandeur, lesquelles sont généralement plus complexes que les cas d'école utilisés pour les démonstrations..

Nous avons effectivement commencé par développer un prototype logiciel monolithique, sur lequel collaborait toute l'équipe. Cependant, une fois passée la fabrication des briques logicielles

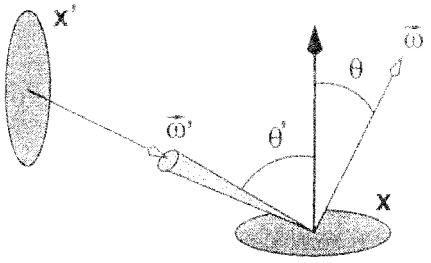


FIG. 4.1 – *Paramètres géométriques de l'équation de transfert*

de base, les besoins des chercheurs et des ingénieurs chargés du transfert industriel ont rapidement divergé, au point de nécessiter la création de systèmes séparés. C'est pourquoi, au delà de la conception d'un prototype unique (tel que c'est formulé dans l'intitulé précédent), notre travail s'est alors orienté vers la mise au point d'un ensemble de règles et d'éléments logiciels permettant de maintenir la cohérence d'ensemble des différents projets de l'équipe.

Pour y parvenir, nous avons pris le parti d'utiliser la programmation par objets. Comme nous l'avons souligné dans le chapitre 3, les systèmes logiciels proposés dans la littérature de l'époque n'étaient pas des systèmes à objets ou étaient trop centrés sur les méthodes de lancer de rayons. C'est ce qui nous a encouragé à concevoir un système logiciel nouveau, que nous avons nommé *plate-forme Graph'IS*.

L'architecture de la plate-forme Graph'IS résulte de deux démarches. D'une part, en partant de l'équation de transfert de la lumière, nous avons défini un ensemble de classes standards, regroupées dans une bibliothèque (section 1). D'autre part, en étudiant les différentes étapes d'un projet de synthèse d'images, nous avons défini des formats de fichiers ainsi qu'un ensemble de programmes standards (section 2). Nous exposerons à travers plusieurs exemples comment cette architecture a facilité nos travaux de recherche et nos transferts industriels (section 3). L'exploitation intensive de la plate-forme nous a aussi permis de mettre en évidence ses limites, que nous discuterons en fin de chapitre (section 4).

1 Bibliothèque de classes

Le cœur de la plate-forme Graph'IS est un ensemble de classes réunies dans une bibliothèque précompilée. Il s'agit d'une part de *classes utilitaires* prêtes à l'emploi, comme on peut en trouver dans une bibliothèque classique, et d'autre part de *classes abstraites* (cf. chapitre 2).

Afin d'asseoir la validité physique de la plate-forme, nous avons défini les classes principales en nous fondant sur l'équation de transfert ci-dessous (cf. chapitre 1) :

$$r(x,\vec{\omega}) = r_0(x,\vec{\omega}) + \int_{\Omega} p(x,\vec{\omega}',\vec{\omega}) r(x'(x,\vec{\omega}'),\vec{\omega}') cos\theta' d\vec{\omega}' \qquad (4.1)$$

avec (figure 4.1) :

– S : lieu formé par les surfaces de la scène,

- Ω : ensemble des directions,
- Λ : ensemble des spectres du domaine visible,
- $r(S \times \Omega) \mapsto \Lambda$: radiance totale,
- $r_0(S \times \Omega) \mapsto \Lambda$: radiance émise,
- $\rho(S \times \Omega^2) \mapsto \Lambda$: réflectance,
- $x'(S \times \Omega) \mapsto S$: visibilité.

On peut immédiatement reconnaître dans l'équation les deux constituants de base de la scène que l'on souhaite visualiser : les *primitives géométriques* qui supportent les fonctions r et ρ, et les *sources lumineuses* qui supportent la fonction r_0. Les classes `Primitive` et `Source` seront les briques de base de la bibliothèque. Par ailleurs, on sait que la fonction x' est particulièrement coûteuse, et nous avons choisi de confier son évaluation à la classe dédiée `Espace`. Au delà de ces classes directement issues de l'équation, nous avons ajouté les suivantes, moins directement tangibles mais tout aussi importantes :

`Scene`

Les fonctions ρ et r_0 sont des caractéristiques *locales* et supposées connues des primitives et des sources. Au contraire, la fonction r est inconnue et s'obtient par la simulation *globale* des transferts radiatifs. Un objet de classe `Scene` sera chargé de contenir la description globale de la scène, et mettra en œuvre la simulation.

`TypeSource`

On peut généralement définir des distributions de lumière standard pour les sources lumineuses, en particulier lorsqu'il s'agit de sources artificielles (luminaires, projecteurs). Chaque distribution standard $i_0(\Omega) \mapsto \Lambda$ est confiée à une instance de la classe `TypeSource`. Les objets de classe `Source` utilisent des objets de classe `TypeSource` afin d'évaluer r_0 à partir d'une distribution standard i_0.

`Materiau`

La fonction ρ n'est pas vraiment déterminée par le point x pour lequel on demande sa valeur, mais plutôt par le matériau de la surface qui porte ce point. Chaque matériau de la scène détermine une fonction $\rho(\Omega^2) \mapsto \Lambda$ qui sera fournie par une instance de la classe `Materiau`.

`Onde`

les fonctions spectrales r, r_0, i_0 et ρ ont un résultat dans Λ. Un élément de cet espace sera représenté par une instance de la classe `Onde`, chargée de la représentation des spectres pendant la simulation.

`Camera & Vue`

Les objets de classe `Camera` jouent le rôle d'observateurs virtuels, et sont chargé de la visualisation de la scène. Le résultat d'une visualisation est un objet de classe `Vue`.

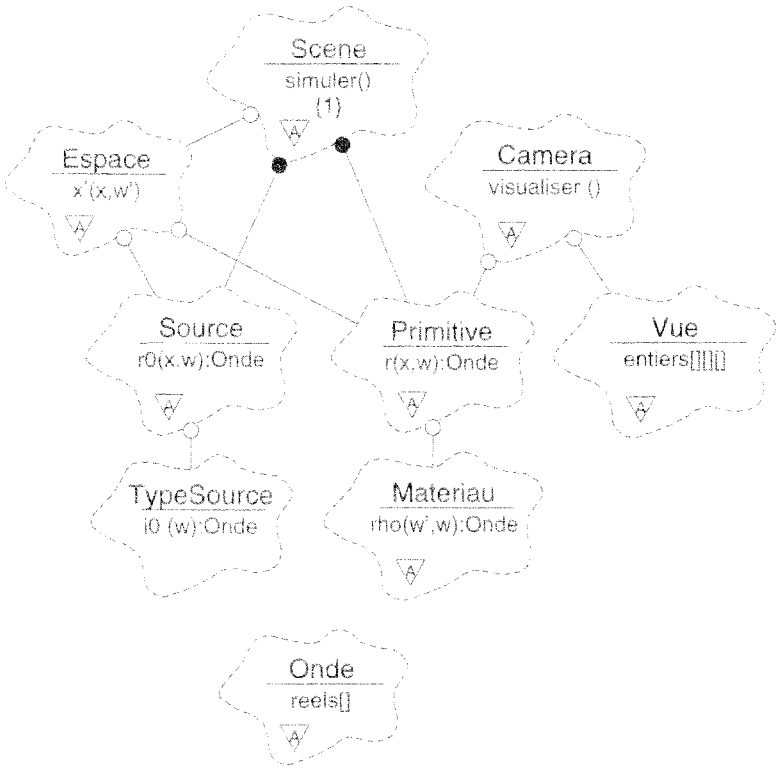


FIG. 4.2 – *Classes principales de la bibliotheque*

Nous obtenons ainsi neuf classes principales qui constituent l'ossature de la plate-forme Graph'IS. Elles sont récapitulées dans la figure 4.2, avec les principales fonctions dont elles sont responsables. Les relations vers la classe `Onde` n'ont pas été représentées pour ne pas surcharger la figure, mais pratiquement tous les objets du système utilisent des objets de classe `Onde`.

Les classes de la figure 4.2 ne fonctionnent pas seules et la bibliothèque contient en réalité beacoup plus de classes. Il convient de les organiser en *catégories*, qui sont illustrées dans la figure 4.3. Les quatres catégories inférieures contiennent en majorité des classes utilitaires de bas niveau. Nous y reviendrons en partie dans la section 2. Les quatre catégories supérieures se partagent les classes principales que nous avons décrites précédemment. Leur découpage repose sur les relations entre les classes ainsi que sur les spécialités scientifiques et techniques concernées. Nous allons maintenant les passer en revue.

1.1 Catégorie `colorimetrie`

Nous regroupons dans cette catégorie tout ce qui a trait à la couleur et à la manipulation de grandeurs spectrales. Nous avons expliqué dans le chapitre 1 que notre stratégie consiste à choisir un base commune de longueurs d'onde $\{\lambda_0, \ldots, \lambda_{n_\lambda}\}$ et à représenter chaque grandeur spectrale par la liste de ses valeurs pour ces longueurs d'onde.

Comme plusieurs modèles colorimétriques sont envisageables, conduisant à des bases de longueurs d'onde différentes, la classe `Onde` est abstraite et définit seulement les services que l'on

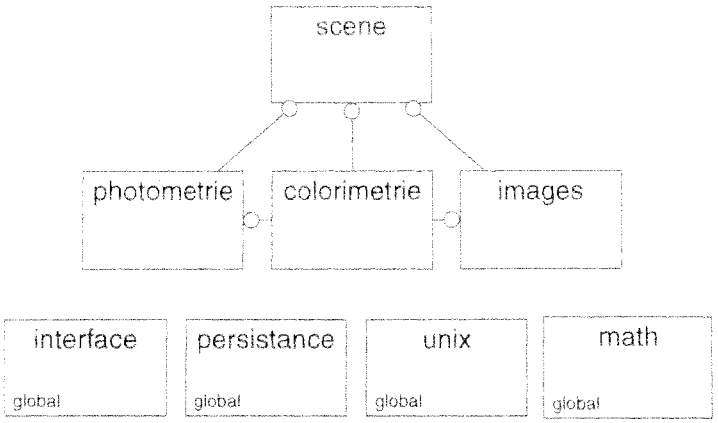


FIG. 4.3 – *Catégories de la bibliothèque de classes*

attend de chaque sous-classe :

- construction d'un objet à partir d'un spectre,
- opérations linéaires entre les objets,
- calcul des flux radiatif et lumineux,
- conversion en triplet XYZ.

On dispose généralement en entrée de spectres relativement précis. En sortie, on préfère convertir les résultats dans l'espace colorimétrique XYZ, qui ne dépend pas des caractéristiques particulières d'un écran. La construction initiale et la conversion finale dépendent fondamentalement du modèle colorimétrique qui a été choisi, c'est pourquoi ces opérations sont toutes à la charge de la classe `Onde`.

La figure 4.4 illustre les classes de la catégorie `colorimetrie` et donne une vision plus précise des différentes représentations qui sont successivement utilisées pour les grandeurs spectrales :

1. `Spectre`

   Données d'entrée sur une émission r'_0 ou une réflectance ρ' : liste de couples [λ/valeur], directement issus des appareils de mesure.

2. `Onde`

   Représentation utilisée pendant la simulation : liste de valeurs correspondant aux $\{\lambda_0, \ldots, \lambda_{n_\lambda}\}$ communs.

3. `Xyz`

   Valeur d'un pixel dans l'espace colorimétrique XYZ, obtenu par conversion de la radiance correspondante.

4. `RvbReel`

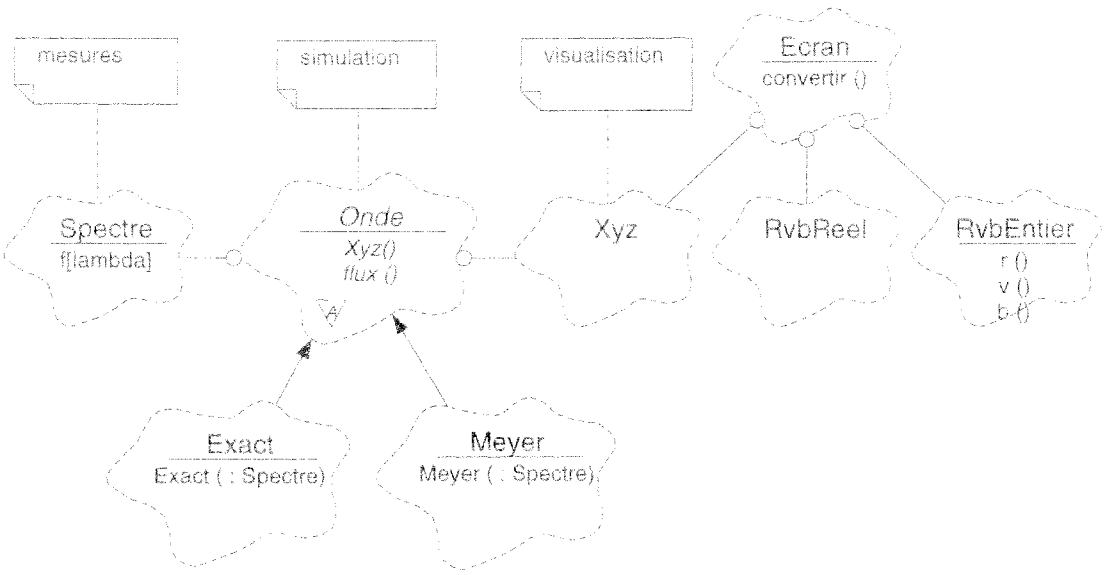


FIG. 4.4 – *Classes de la catégorie* `colorimetrie`

Valeur du même pixel dans l'espace RVB, avant la conversion gamma. L'espace RVB dépend de l'écran à qui est destiné l'image. Un objet de classe `Ecran` est donc chargé de la conversion. La première étape repose sur les coordonnées chromatiques des luminophores de l'écran et produit une instance de `RvbReel`.

5. `RvbEntier`

   Résultat de la dernière étape de conversion, qui inclue la correction gamma et la transformation des trois coordonnées [rouge/vert/bleu] en valeurs entières (valeurs de 0 à 255).

Deux sous-classess de `Onde` sont proposées dans la bibliothèque : la classe `Exacte` qui utilise 80 longueurs d'ondes, et la classe `Meyer` basée sur le modèle du même nom [Mey88b].

1.2 Catégorie `photometrie`

La photométrie est la discipline étudiant la mesure des flux lumineux. Certains laboratoires sont spécialisés dans cette activité, et nous voulions exploiter au maximum les données dont ils disposent. La catégorie présente est dédiée aux classes qui font le lien entre les mesures et les fonctions nécessaires à la simulation.

Les classes sont récapitulées dans la figure 4.5. On distingue deux groupes, liés respectivement aux fonctions i_θ et ρ. La structure de ces groupes est similaire. Elle repose sur une classe racine qui supporte la méthode principale (`TypeSource`/`Materiau`), une classe contenant la composante spectrale de l'entité mesurée (`Lampe`/`RepSpec`), et une classe contenant la composante spatiale (`Solide`/`RepSpat`). Les deux composantes sont dissociées parce qu'elles sont issues de deux catégories distinctes d'appareils, et le rôle de la classe racine est justement de les recomposer. Nous allons examiner les spécificités de chacun des deux groupes.

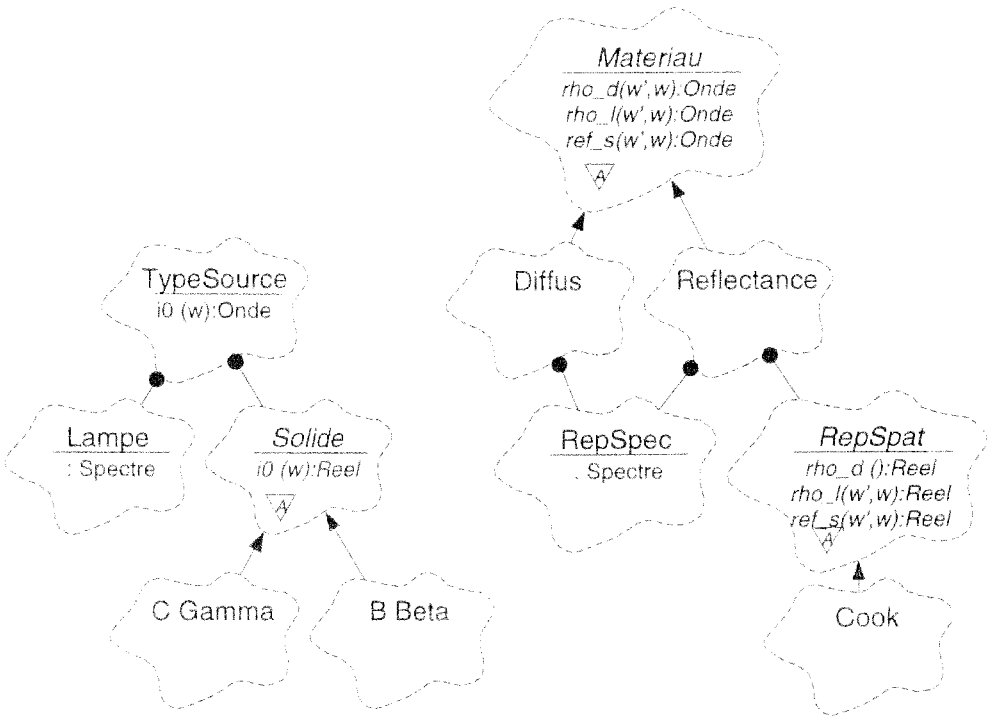


FIG. 4.5 – *Classes de la catégorie* `photometrie`

Types de sources lumineuses

Une source lumineuse peut faire l'objet de deux mesures : la distribution spectrale de la lampe seule, réalisée à l'aide d'une *sphère intégrante*, et la distribution de la source complète, réalisée à l'aide d'un *gonio-photomètre*. L'ensemble repose sur l'hypothèse de ponctualité de la source (les surfaces éclairées doivent être à une distance de la source supérieure à six fois sa largeur). Nous nous sommes cantonné à cette hypothèse, et nous avons prévu une seule classe `TypeSource`, dont la méthode `i0` ne renvoie pas une *radiance* mais une *intensité*.

La distribution spectrale mesurée est stockée dans une simple instance de `Spectre`, elle-même encapsulée dans une instance de `Lampe`.

La distribution spatiale est appelée *solide photométrique*. Elle peut être organisée selon différents systèmes de coordonnées. On peut aussi vouloir préparer « manuellement » des solides simplifiés, en l'absence de mesures ou pour réaliser des tests. Pour toutes ces raisons, nous proposons une classe abstraite `Solide`, et deux sous-classes pour les deux systèmes de coordonnées les plus répandus : [C Gamma] et [B Beta].

La méthode `i0` de `Solide` renvoie un réel. C'est en combinant ce réel avec le spectre de la lampe que la méthode `i0` de `TypeSource` renvoie finalement une valeur spectrale (objet de classe `Onde`).

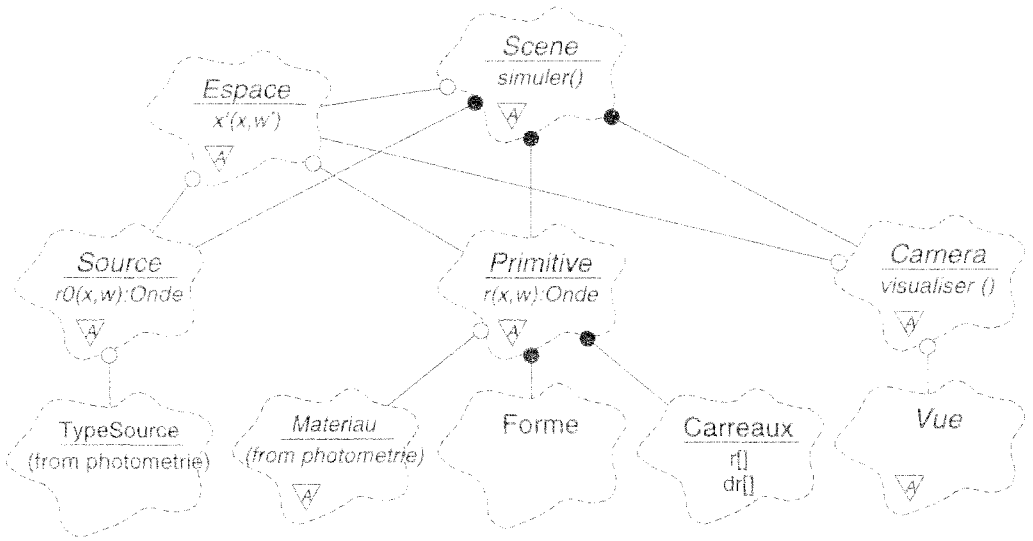


FIG. 4.6 – *Classes de la catégorie* `scene`

Matériaux

La réflectance des matériaux fait l'objet de beaucoup moins de mesures que les sources lumineuses. En effet, la dépendance bi-directionnelle complique singulièrement l'opération. L'exploitation d'un *gonio-réflectomètres* n'est possible que sur des échantillons au sein d'un laboratoire. Pour des mesures in situ, on se contente d'une mesure par *spectro-photomètre*, qui fournit le spectre de réflexion à la verticale du matériau. Cette mesure est stockée dans une instance de `Spectre`, elle-même encapsulée dans un objet `RepSpec`.

Pour compenser le manque de mesures sur la composante spatiale de la réflectance, on a recours à différents modèles physiques et mathématiques, dont aucun ne semble dominant. Pour permettre de choisir facilement entre ces modèles, nous avons défini une classe abstraite `RepSpat`.

Les classes `Materiau` et`RepSpat` possèdent trois méthodes chacunes, correspondant à la décomposition classique de la réflectance : réflectance diffuse, lobe spéculaire et pic spéculaire. Le pic spéculaire ne peut s'exprimer qu'à l'aide d'une fonction de Dirac, c'est pourquoi nous préférons le remplaçer par une *réflectivité* et utiliser la nom de méthode `ref` plutôt que `rho`.

Deux modèles prédéfinis sont implémentés dans la bibliothèque. Le modèle diffus uniforme repose sur la seule classe `Diffus`. Le modèle de Cook et Torrance reposent sur la classe `Reflectance` qui implémente la recomposition d'un spectre et d'une réflectance réelle, et sur la classe `Cook` qui implémente la composante spatiale de ce même modèle.

1.3 Catégorie `scene`

Cette catégorie centrale contient les éléments directement mis en jeu dans la simulation des transferts radiatifs (figure 4.6). On distingue trois types d'éléments : les sources de lumière, les primitives géométriques transférant la lumière, et finalement les capteurs de lumière. Ces trois types d'éléments sont respectivement représentés par les classes `Source`, `Primitive` et `Camera`. Ils définissent complètement la scène que l'on veut simuler.

Il nous semble important de justifier la distinction que nous faisons entre les sources de lumière et les primitives. D'un point de vue mathématique, les fonctions r et r_0 peuvent partager le même support et rien n'oblige à les répartir entre des éléments de natures différentes. Dans la pratique, plusieurs motifs militent pour la séparation :

- primitives et sources d'une scène ne sont modélisées ni au même moment ni par les mêmes personnes,
- les mesures photométriques qui décrivent les sources artificielles font l'hypothèse que la source est ponctuelle et non pas surfacique,
- on distingue souvent dans une simulation les échanges de source à primitive (illumination directe) et les échanges entre primitives (illumination indirecte),
- les modifications d'une source ou d'une primitive en cours de simulation n'ont pas le même impact.

Les différents éléments de la scène ne se connaissent pas les uns les autres, et ne sont responsables que de caractéristiques *locales*. C'est la classe `Scene` qui regroupe la connaissance d'ensemble et qui est chargée de simuler les transferts radiatifs. Il nous a semblé important de regrouper dans `Scene` tout ce qui constituait une connaissance *globale*, et d'y centraliser autant que possible l'algorithme de simulation. Toutefois, une tâche essentielle de la simulation, le calcul de la visibilité, est sous-traitée à la classe `Espace`. Il s'agit d'une tâche suffisamment spécifique et séparable pour la confier à une classe spéciale.

Enfin, la visualisation est confiée à la classe `Camera`, qui produit une image de la scène et l'intègre à un objet de type `Vue`. Un objet `Vue` contient une copie de l'objet `Camera` dont il est issu, en partie pour garder la trace de sa construction. Comme les deux classes se connaissent mutuellement, elle doivent être rangées dans la même catégorie. C'est ce qui explique la présence de `Vue` dans la catégorie `scene`, alors qu'elle serait plus à sa place dans la catégorie `images` que nous allons décrire maintenant.

1.4 Catégorie `images`

Cette dernière catégorie est dédiée au traitement des images et aux opérations en deux dimensions. Elle exploite les espaces de couleurs proposés dans la catégorie `colorimétrie`, et elle est évidemment utilisée par les caméras de la catégorie générale `scene`. Une grande partie de la catégorie `images` sert à réaliser des combinaisons linéaires d'images. En effet, la propagation de l'énergie lumineuse est linéaire et on peut utiliser cette propriété pour calculer certaines images par parties. Par ailleurs, il nous est également nécessaire de combiner des images lorsque nous souhaitons incruster un objet de synthèse dans une photographie. Pour répondre à ces besoins, nous organisons nos objets en trois étages :

- La classe `Image` décrit un tableau de pixels. C'est une photographie numérisée ou le résultat d'une visualisation d'une scène de synthèse. Il s'agit d'une image telle qu'on l'entend couramment, avec les seules informations habituellement stockées dans un fichier : la résolution et les coordonnées [rouge/vert/bleu] des différents pixels.

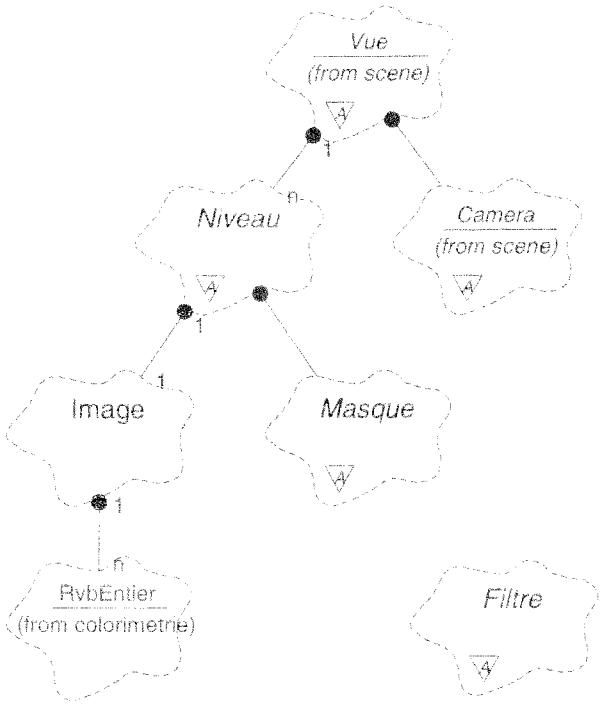


FIG. 4.7 – *Classes de la catégorie* `images`

- La classe `Niveau` permet de compléter les informations sommaires de la classe `Image` par tout ce qui est nécessaire au différents traitements qu'on peut vouloir lui appliquer. Quand ces informations concernent les différents pixels isolément, elles sont stockées sous forme d'objets `Masque`. Étant donné la diversité des traitements et informations possibles, les classes `Niveau` et `Masque` sont abstraites.

  La classe `Vue` définit une combinaison linéaire d'images. Elle contient la description du point de vue commun (objet `Camera`), la liste des images d'entrée avec leurs masques respectifs (objets `Niveau`), et les opérateurs à appliquer aux images d'entrée pour former l'image de sortie.

Au lieu de calculer immédiatement le résultat d'une combinaison linéaire d'images, nous préférons stocker sa formule dans un objet `Vue` qui conserve toute l'information disponible. Il est ainsi possible d'ajuster n'importe quand les coefficients linéaires ou d'appliquer de nouveaux filtres aux images d'entrée. Lorsque l'image composée est satisfaisante et que l'on veut minimiser l'encombrement mémoire, on demande simplement à l'objet `Vue` de produire un objet `Image` contenant l'image composée de sortie, et on peut alors détruire l'objet `Vue` et éventuellement les images et les masques intermédiaires.

2 Données et programmes

Dans la section 1, nous avons présenté les catégories et les classes qui constituent la bibliothèque précompilée de la plate-forme Graph'IS. Cette bibliothèque permet aux membres de notre

équipe de partager du code.

Au delà de cet apport à la programmation, nous voulons également améliorer le test et l'exploitation des logiciels que nous développons. Pour ce faire, il nous a semblé nécessaire d'uniformiser la gestion des données et l'utilisation des programmes de la plate-forme. En faisant l'hypothèse qu'un utilisateur de la plate-forme possède une culture minimale en `Unix` et en `C++`, nous avons établi quelques règles et quelques objectifs basiques :

- Chaque programme aura pour interface un interpréteur de commandes semblable à un interpréteur `Unix`.
- Les fonctionnalités les plus couramment utilisées dans nos projets doivent être réparties parmi un ensemble de programmes prédéfinis.
- Les différents programmes échangeront leurs données d'entrée et de sortie à travers des fichiers de syntaxe uniforme. Les fichiers doivent être lisibles, doivent pouvoir être préparés à l'aide d'un simple éditeur de texte, et le contenu des fichiers doit refléter la structure objet de la bibliothèque.
- Les données les plus couramment utilisées dans les projets de l'équipe doivent être organisées selon un ensemble de formats de fichiers prédéfinis.

Pour déterminer les données et programmes à prédéfinir, nous nous sommes fondés sur les étapes habituelles des projets que menons en collaboration avec des éclairagistes et des architectes. De façon schématique, nous pouvons distinguer cinq activités dans la poursuite d'un projet :

- La saisie des caractéristiques géométriques de tous les éléments de la scène. Nous appellerons cette activité *modélisation géométrique*.
- La prise de mesures photométriques sur les matériaux et les luminaires utilisés dans le projet. Nous appellerons cette activité *modélisation photométrique*.
- La saisie du projet d'éclairage (type et position des luminaires) et la saisie des points de vue privilégiés. Nous appellerons cette activité *modélisation lumineuse*.
- La *synthèse* des images proprement dite (l'activité qui nous concerne le plus directement).
- La préparation finale des images : composition, incrustation éventuelle dans une photographie ou une séquence vidéo.

En étudiant l'enchaînement des activités et les informations qui sont échangées de l'une à l'autre, nous avons choisi de prédéfinir cinq programmes et nous avons baptisé les blocs d'information qui transitent entre ces programmes, tel que cela est présenté dans la figure 4.8. Dans cette figure, les flèches larges représentent les données extérieures, les rectangles représentent les programmes, et les ovales représentent les blocs de données internes. Dans les trois sous-sections suivantes, nous revenons plus en détail sur les données et les programmes de cette figure.

Nous allons commencer par exposer la méthode générale qui nous a permis de marier la notion de *persistance* avec le système de fichiers Unix (2.1). Dans un second temps, nous présenterons les spécifications obtenues en appliquant cette méthode à notre problématique (2.2). Nous terminerons la section par une revue des programmes prédéfinis (2.3).

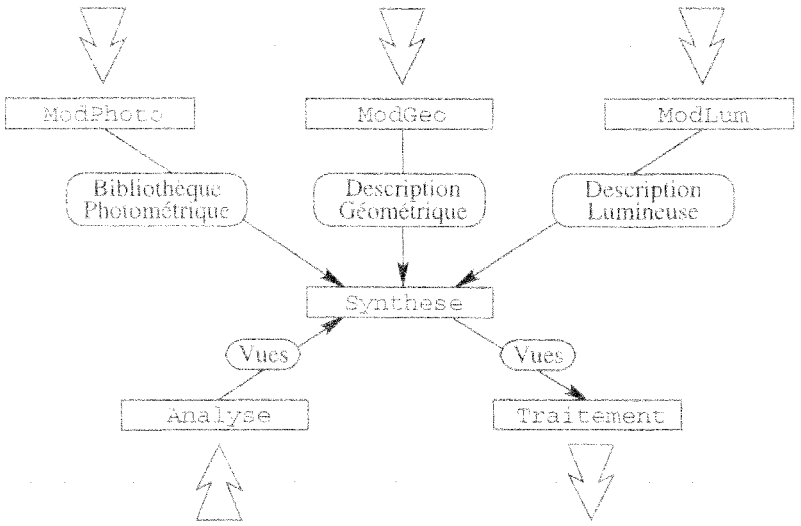


FIG. 4.8 – *Données et programmes de la plate-forme Graph'IS*

2.1 Persistance & système Unix

La gestion des entrées/sorties sur disque correspond dans le paradigme objet à la notion de persistance. On qualifie de *persistant* un objet dont la durée de vie excède celle du programme en cours d'éxécution, ce qui suppose que cet objet est stocké lorsque le programme s'achève et qu'il est reconstruit tel quel lorsque le programme est relancé.

Dans le cadre de la plate-forme Graph'IS, nous tenions d'une part à respecter le paradigme objet, d'autre part à utiliser des fichiers ASCII simples et lisibles pour le stockage de nos données. Pour marier ces objectifs, nous avons décidé que certaines classes de la plate-forme seraient considérées comme persistantes, et qu'à ce titre elles devraient pouvoir manipuler pour chaque objet une représentation ASCII équivalente. Pour normaliser nos entrées/sorties, nous avons donc défini quelles étaient les classes persistantes, quelle était la représentation ASCII de leurs instances, et dans quels fichiers seraient stockées ces représentations.

Nous définissons ci-dessous les principes généraux qui ont guidé notre travail sur la persistance, et la catégorie qui facilite leur implémentation. L'application effective de ces principes à la simulation des transferts radiatifs ne sera abordée que dans la sous-section suivante (2.2).

Le concept de champ

Nous appellons *champ* la représentation ASCII équivalente d'un objet persistant. Chaque champ commence par une *étiquette* (mot encadré par les caractères « < » et « > »), destinée à illustrer la nature des données qui lui succèdent. Par ailleurs, afin de permettre la représentation d'objets complexes, les champs peuvent être imbriqués les uns dans les autres. Lorsque le contenu d'un champ est une liste de sous-champs imbriqués, cette liste est close par l'étiquette vide « <> ». Cette structure peut se décrire par des règles BNF :

```
champ       ::=  ETIQUETTE contenu
contenu     ::=  liste_champs | liste_mots
```

```
liste_champs ::= { champ } FIN
liste_mots   ::= { MOT }

ETIQUETTE       ::= '<' { LETTRE | CHIFFRE } '>'
FIN             ::= '<>'
MOT             ::= [ '+' | '-' ] { LETTRE | CHIFFRE | '.' }
LETTRE          ::= 'A' ... 'Z' | 'a' ... 'z' | '_'
CHIFFRE         ::= '0' ... '9'
```

Cette syntaxe à l'avantage d'être extrêmement simple, et de pouvoir rendre compte en partie de la structure objets sous-jacente. Tous les fichiers de la plate-forme Graph'IS sont composés d'une liste de champs, éventuellement entrecoupés de commentaires. À titre d'exemple, voici un extrait de fichier contenant la description d'une caméra, d'une source et de deux primitives :

```
# fichier : exemple.scene
# auteur : Isabelle Fasse
# date : 26 février 1995
# version : 1.6
<camera>
    <nom> c1
    <position>
        <origine> 0.0 5.0 0.0
        <mire> 1.0 1.0 0.0
        <>
    <ouverture> 50          # angle d'ouverture horizontal.
    <resolution> 200 100    # résolution de l'image
    <>
<source>                    # source lumineuse
    <nom> s1
    <position>
        <origine> -1.0 -1.0 3.0
        <mire> 0.0 0.0 0.0
        <>
    <type> spot12
    <>
<couleur> 1                 # couleur des primitives suivantes.
<facette>                   # primitive géométrique plane.
    <contour>
        1.00 0.00 1.00      # contour extérieur polygonal,
        7.00 0.00 1.00      # décrit par ses sommets donnes
        7.00 6.00 1.00      # dans l'ordre trigonometrique
        1.00 6.00 1.00
    <trou>
        3.00 2.00 1.00      # trou polygonal,
        3.00 4.00 1.00      # décrit par ses sommets donnés
        6.00 2.00 1.00      # dans l'ordre trigonometrique
    <>
```

```
<facette>                    # seconde primitive
    <contour>
        0.00 0.00 0.00
        0.00 7.00 0.00
        8.00 7.00 0.00
        8.00 0.00 0.00
    <>
<>
```

Lorsque l'on relit un fichier composé de champs, c'est l'enchaînement des étiquettes qui permet de détecter les imbrications : quand deux étiquettes se suivent, la deuxième étiquette désigne un sous-champ imbriqué ; l'étiquette vide désigne la fin d'une imbrication. Quant à la mise en page des fichiers, nous avons choisi de ne rien imposer aux programmeurs et utilisateurs, mais il est certain qu'un minimum de mise en page est nécessaire pour préserver la lisibilité des fichiers.

Formats de champs

Notre façon de gérer les entrées/sorties dans la plate-forme Graph'IS consiste donc essentiellement à définir certaines classes comme persistantes, et à doter chacune de ces classes de deux méthodes : l'une pour produire un champ à partir d'un objet, l'autre pour reconstruire un objet à partir d'un champ. Pour chaque classe, il est nécessaire de spécifier un *format de champ* : quelles seront les étiquettes, les imbrications éventuelles de sous-champs, les données qui seront stockées. Nous avons examiné chaque classe isolément, en adoptant toujours un format de champ qui soit facile à parcourir visuellement. Nous ne voulions pas théoriser sur la persistance et la manière de stocker des systèmes d'objets dans des fichiers. Du reste, il est sans doute impossible de traduire de façon satisfaisante la structure et les relations complexes établies entre les classes dans la syntaxe simpliste des champs. Nous pouvons néanmoins livrer quelques principes généraux (les exemples font référence à l'extrait de fichier donné précédemment) :

- L'étiquette principale du champ correspond parfois au nom de la classe (ex : `<camera>`), parfois à l'usage qui est fait de l'objet si la classe est trop générale (ex : `<ouverture>` pour le réel designant l'angle d'ouverture d'une caméra).
- Les variables de types prédéfinis de base (int, float, etc) sont regroupées autant que possible (ex : les résolutions en x et en y d'une `Camera` sont regroupées en un seul champ `<resolution>` ; les coordonnées des sommets d'une facette sont regroupées dans le seul champ `<contour>`).
- Les relations d'agrégation entre les objets se traduisent par des imbrications de champs (ex : le repère inclus dans une caméra ou une source devient un champ `<position>` imbriqué dans les champs `<camera>` et `<source>`).
- Lorsqu'un objet est partagé par plusieurs autres, il est nommé et décrit séparément, et seul son nom est cité dans les champs des objets clients (ex : la source `s1` fait référence au type `spot12`, dont la description est donnée ailleurs).
- Nous n'avons pas réussi à dégager une règle générale pour traiter les relations d'héritage, qui sont utilisées à des fins trop diverses.

Nous avons toujours cherché à alléger nos fichiers, pour simplifier la tâche de l'utilisateur qui voudrait les éditer directement. Lorsqu'un champ contient une liste de mots, il est rare que sa taille soit précisée. Il est ainsi plus facile d'ajouter ou de supprimer des éléments dans la liste. Toujours pour éviter les contraintes inutiles, l'ordre des sous-champs imbriqués est généralement libre, et des valeurs par défaut peuvent être prévues (ex : le format du champ <camera> inclus un sous-champ <roulis> dont la valeur par défaut est 0, et qui peut donc être omis). Il est parfois permis de redéfinir une valeur par défaut au niveau d'imbrication supérieur, ce qui a l'inconvénient de rendre l'ordre des champs signifiant (ex : la <couleur> de l'exemple sert de valeur par défaut aux <facettes> suivantes).

Les formats d'entrées/sorties constituent toujours un obstacle important pour l'évolution d'un logiciel, et nous voulions que la plate-forme Graph'IS facilite ce genre d'évolution tout en préservant la pérennité de nos bases de données. Quelques mécanismes ont été adoptés pour y parvenir :

- Les étiquettes des formats de champs peuvent avoir plusieurs variantes, ce qui permet de les changer plus facilement, ou de disposer d'une version anglaise.

- Lors de l'interprétation d'un champ, les sous-champs inconnus sont simplement ignorés.

- La méthode de reconstruction d'une classe est capable de comprendre tous les formats de champs encore en pratique. Par contre, la méthode de traduction d'un objet en champ utilise systématiquement le dernier format officiel, ce qui favorise l'évolution des bases de données vers ce format.

Comme nous l'avons déjà souligné, aucune des règles données ci-dessus n'a été appliquée systématiquement. Nous avons spécifié cas par cas chaque représentation de classe, sans chercher à découvrir une méthode générale hypothétique. Nous espérons cependant avoir convaincu le lecteur que nous avons respecté nos objectifs : obtenir des fichiers simples et faciles à manipuler, tout en reflétant la structure objet sous-jacente et en facilitant l'évolution des formats de données.

Formats de fichiers

Au delà des formats de champs, nous avons aussi spécifié comment ces champs devaient être distribués dans les fichiers. Un certain nombre de formats de fichiers ont été prédéfinis, qui seront présentés dans la sous-section 2.2.

Chaque format précise les classes des objets présents, et définit éventuellement une liste de répertoires Unix où les fichiers peuvent être rangés. Le nom des fichiers porte toujours une extension qui signale leur format. Dans ce mémoire nous utiliserons la notation `*.ext` pour désigner indifféremment les fichiers au format `ext` ou le format lui-même.

Pour terminer ces quelques informations sur les fichiers de données, ajoutons qu'il est permis d'insérer des commentaires dans les fichiers : les caractères compris entre un caractère « # » et la fin de ligne sont ignorés lors de la lecture du fichier.

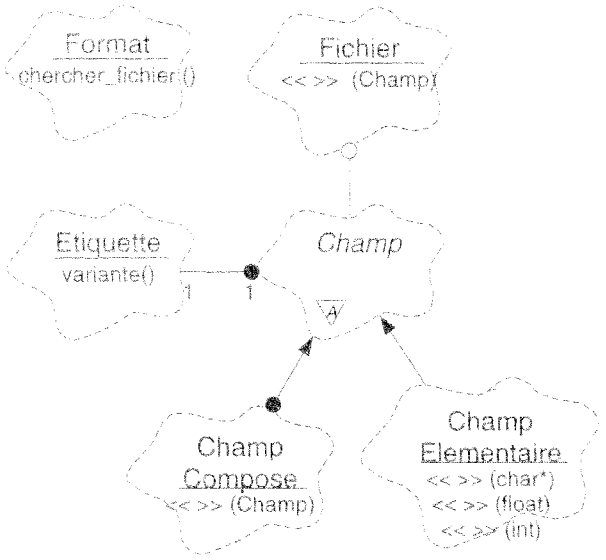


FIG. 4.9 – *Classes de la catégorie* `persistance`

Catégorie `persistance`

Pour faciliter l'implémentation d'entrées/sorties sur disque conformes aux principes ci-dessus, nous avons ajouté à la bibliothèque une catégorie utilitaire dont les classes sont présentées dans la figure 4.9.

La classe `Champ` constitue le noyau de cette catégorie. Elle permet de grouper des données et de leur adjoindre une étiquette (`Champ Elementaire`), ou éventuellement d'imbriquer des champs déjà constitués dans un champ de plus haut niveau (`Champ Compose`). En complément, certaines méthodes permettent d'extraire les sous-champs ou les données contenues dans l'objet. L'ensemble de ces opérations est implanté sous forme d'opérateurs d'insertion « ≪ » et d'extraction « ≫ ». L'interface de ces classes est ainsi semblable à celle de la bibliothèque standard d'entrées/sorties du langage C++.

Les objets de classe `Fichier` savent enregistrer le contenu d'un objet de classe `Champ` dans un fichier Unix, en respectant une mise en page régulière et lisible. Leur interface est également à base d'opérateurs « ≪ » et « ≫ ».

Chaque classe persistante doit contenir une méthode pour traduire une de ses instances en objet `Champ`, et une autre méthode pour reconstruire une instance à partir d'un objet `Champ`. Dès lors, il devient facile de faire persister un objet : celui-ci est transformé en objet `Champ`, inséré dans un fichier Unix via la classe `Fichier`, et restitué ultérieurement par les opérations inverses.

La dernière classe de la figure, `Format`, est une classe globale et statique qui contient la liste des formats de fichiers et leurs caractéristiques. Elle propose entre autres une méthode facilitant la recherche automatique des fichiers dans les répertoires prédéfinis.

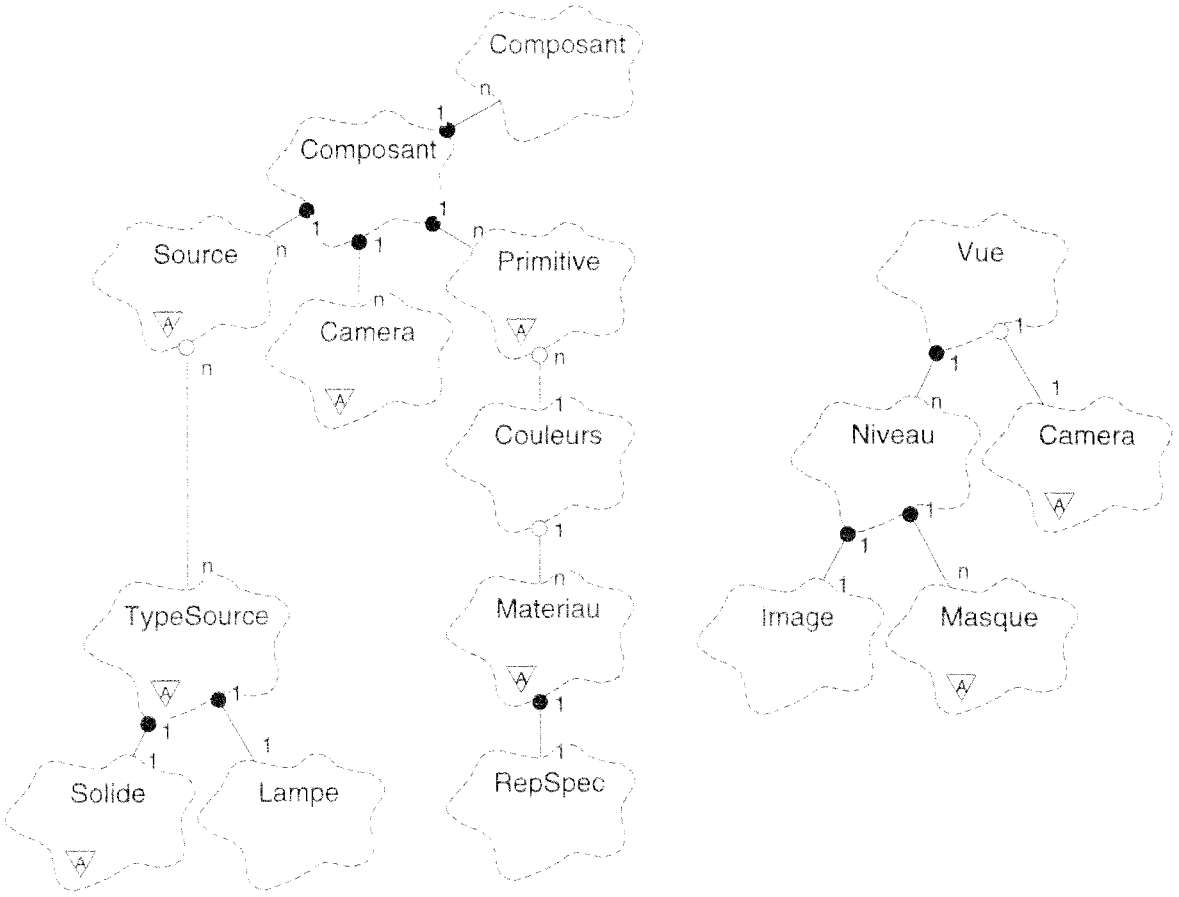


FIG. 4.10 – *Classes persistantes de la bibliothèque*

2.2 Formats de données de la plate-forme

Nous avons expliqué comment notre souci de rester fidèle à la philosophie objet nous a amené à laisser de côté les formats de fichiers traditionnels (nff, explore, renderman), pour préférer une démarche centrée sur la persistance. Nous avons expliqué comment cette démarche repose sur trois étapes : choisir quelles sont les *classes persistantes*, définir pour chacune d'entre elles le *format de champ* associé, et finalement spécifier la distribution des objets parmi les fichiers Unix à travers des *format de fichiers*.

Nous exposons ci-dessous le résultat de cette démarche, appliquée à notre problème de simulation. Nous voulions éviter au lecteur la description rébarbative des formats de champs. Nous allons donc nous limiter à lister les classes qualifiées de persistantes, puis nous passerons en revue les différents formats de fichiers. Nous avons organisé ces formats conformément aux blocs de la figure 4.8, c'est-à-dire en *bibliothèque photométrique*, en *descriptions géométriques et lumineuses*, et enfin en *vues*.

Classes persistantes

La figure 4.10 présente les principales classes de la bibliothèque que nous avons définies comme persistantes, et que nous avons donc dotées de méthodes pour transformer leurs objets en champs. L'arborescence de gauche constitue la description d'une scène, c'est-à-dire l'ensemble des informations d'entrées du programme `Synthese`. L'arborescence de droite correspond au résultat de la simulation, c'est-à-dire les informations de sortie du programme `Synthese`.

Dans la description de la scène, on reconnaît les émetteurs de lumière (`Source`), les transmetteurs (`Primitive`) et les récepteurs (`Camera`). On reconnaît également les objets qui contiennent les mesures photométriques (`Solide`, `Lampe`, `RepSpec`).

Etant donnés les projets sur lesquels travaillait notre équipe, en particulier pour l'illumination des monuments, nous devions couramment manipuler des milliers de primitives et des centaines de sources pour une même scène. Afin de faciliter cette manipulation nous avons ajouté la classe `Composant`, qui permet d'organiser la scène sous forme d'un graphe acyclique. Chaque composant porte un nom et correspond en général à une sous-partie architecturale de la scène, comme une façade, un étage, une colonne, etc.

On peut enfin remarquer la présence d'une nouvelle classe `Couleurs` entre `Primitive` et `Materiau`. Cette classe introduit une indirection entre les primitives et leurs matériaux. Au lieu d'attribuer directement un matériau à chaque primitive, on lui attribue une couleur, qui sera ultérieurement associée à un matériau. Grâce à cette indirection, la géométrie peut être préparée indépendamment des matériaux, et il est plus facile de tester différentes combinaisons de matériaux pour une scène donnée.

Selon la figure 4.8, la préparation des données constituant la scène est répartie entre plusieurs programmes, et nous avons choisi de répartir de même les objets parmi plusieurs groupes de fichiers, selon leur classe :

- Bibliothèque photométrique : `TypeSource`, `Solide`, `Lampe`, `Materiau`, `Fresnel`.
- Description géométrique : `Composant`, `Primitive`.
- Description lumineuse : `Source`, `Couleurs`, `Camera`.
- Vues : `Vue`, `Niveau`, `Camera`, `Image`, `Masque`.

Bibliothèque photométrique

Nous regroupons sous ce label l'ensemble des données disponibles sur le comportement photométrique des sources lumineuses et des matériaux. Il s'agit essentiellement de mesures réalisées en laboratoire, et permettant de construire les objets de la catégorie `photometrie`. Nous utilisons le terme « bibliothèque » parce ces informations peuvent être réutilisées dans tous les projets, et que chaque nouvelle campagne de mesures vient enrichir la collection des modèles de sources et des matériaux disponibles.

Nous avions initialement créé une seule bibliothèque globale, partagée par tous les utilisateurs et tous les projets. Les membres de l'équipe qui collaboraient avec des laboratoires de photométrie étaient chargés d'alimenter cette bibliothèque. Pour faciliter sa gestion, nous avions décidé de

créer un fichier pour chaque modèle de source ou de matériaux, et de regrouper l'ensemble dans un répertoire commun.

Dans la pratique, il s'est avéré que nous ne disposions jamais de toutes les mesures nécessaires pour un projet, et nous devions compenser en créant « à la main » certains fichiers. Dès lors, nous avons commencé à avoir plusieurs représentations des mêmes matériaux ou des mêmes sources, selon le projet ou l'utilisateur concerné. En conséquence, nous continuons à présent de maintenir une bibliothèque centrale, mais elle est complétée par d'autres bibliothèques liées aux utilisateurs ou aux projets, et stockées dans d'autres répertoires. Par défaut, lorsqu'un fichier photométrique est recherché, on inspecte successivement les répertoires `./BibPhoto`, `$PROJET/BibPhoto`, `$USER/BibPhoto` et `$GRAPHIS/BibPhoto`, qui sont autant de répertoires susceptibles de supporter une bibliothèque et de contenir le fichier.

En ce qui concerne plus précisément les matériaux, nous ne savons aujourd'hui mesurer que le facteur de Fresnel, ce qui explique que la classe `RepSpat` n'a pas été déclarée persistante. Nous avons choisi de créer un fichier `*.mat` pour chaque matériau, et d'y stocker son facteur de Fresnel. L'objet `RepSpec` correspondant est bien représenté par un champ dans le fichier, par contre l'objet `Materiau` est représenté par le fichier `*.mat` lui-même. Les méthodes de persistance de la classe `Materiau` ne transforment pas ses instances en `Champ` mais directement en objet `Fichier`.

En ce qui concerne les types de sources lumineuses, nous disposons de deux familles de mesures : les mesures de lampes et les mesures de solides photométriques. Pour un même solide photométrique, il existe plusieurs lampes possibles. Pour pouvoir combiner solides et lampes sans dupliquer les données, nous avons choisi d'utiliser trois formats de fichiers :

`*.lampe` : contient les champs représentant un objet de type `Lampe`, à savoir un spectre et la valeur du flux nominal.

`*.optique` : contient un champ `BBeta` ou `CGamma`, et quelques informations supplémentaires comme le rendement optique ou l'aspect sous lequel doit apparaître la source dans une image.

`*.source` : fichier décrivant une instance de `TypeSource` , à travers le nom d'un fichier **.lampe* et le nom d'un fichier **.optique*.

La bibliothèque photométrique déroge à la règle consistant à associer un champ à chaque objet persistant. Nous avons plus volontiers associé les objets à des fichiers, ce qui facilitait la constitution et la maintenance des bibliothèques (il est plus rapide de manipuler des fichiers par des commandes Unix que d'éditer leur contenu).

Description géométrique

Nous appelons *description géométrique* le résultat de la *modélisation géométrique*, qui englobe la saisie de toutes les caractéristiques géométriques des éléments de la scène. De nombreux logiciels puissants permettent de faire cette modélisation, c'est pourquoi nous avons voulu créer un format de fichier réservé à la géométrie, noté `*.geo`. Ce format sert de passerelle avec les modeleurs commerciaux, et nous avons évidemment développé des convertisseurs vers les plus utilisés, comme *Arc—* et *Autocad*, ou vers certains formats neutres comme *DXF*.

Nous nous sommes limités au cas où les éléments de la scène sont fixes, le contenu d'un fichier `*.geo` est donc à la base une simple liste de primitives dont on décrit la forme. Comme la modélisation géométrique est réalisée séparément des mesures photométriques, on ne sait pas par avance quel sera l'échantillon de matériaux disponibles (dans une scène réelle, il y a une infinité de nuances de matériaux). Il est donc impossible d'attribuer des matériaux aux primitives dès le stade de la modélisation géométrique. On peut cependant prédéfinir quelles sont les primitives d'aspect semblable (murs, boiseries, vitres, etc), ce qui nous conduit à la notion de *couleur*. Dans la description géométrique, chaque primitive se voit attribuer une couleur. L'association de matériaux à ces couleurs ne fait pas partie de la description géométrique.

Une scène contient couramment plusieurs milliers de primitives : il est impossible dans la pratique de décrire une scène à l'aide d'un simple fichier contenant une simple liste de primitives. Plusieurs ajouts nous permettent de gérer cette complexité :

- Un mécanisme d'inclusion de fichiers permet de répartir les primitives dans plusieurs fichiers au lieu d'un seul (le champ `<fichier_geo>` permet d'inclure un fichier).
- La classe `Composant` permet de créer un graphe acyclique direct de composants architecturaux, parmi lesquels on peut répartir les primitives.
- Il est possible de déclarer une couleur par défaut dans la description d'un composant, ce qui évite de la répéter dans les primitives qui viennent ensuite.

Lorsqu'un composant est inclus dans un composant père, on décrit sa position dans le repère du père. Un même composant peut être utilisé plusieurs fois avec des positions différentes, ce qui permet de partager certaines descriptions standards de primitives.C'est la présence de composants partagés qui donne sa structure de graphe à la description géométrique de la scène. Etant donné ce partage potentiel de tout composant, et la profondeur illimitée de l'arbre des composants, nous ne pouvions évidemment pas imbriquer les champs `<composant>` les uns dans les autres : le champ qui représente le composant père contient seulement le nom du composant fils et sa position, et le composant fils est décrit séparément.

En ce qui concerne la répartition des composants dans les fichiers `*.geo`, nous avons choisi de ne pas imposer de règle figée, comme le serait par exemple l'utilisation systématique d'un fichier pour chaque composant. Cette liberté permet de choisir une granularité de fichiers adaptée à la scène en cours de modélisation, mais cela créé deux structures concurrentes (composants et fichiers) qu'il est parfois délicat de concilier lors de la lecture ou la sauvegarde des données.

Description lumineuse

La création, le positionnement et le réglage des sources lumineuses est souvent un aspect négligé de la synthèse d'images. Dans la mesure où nous travaillions souvent avec des éclairagistes, il était nécessaire d'en faire une activité aussi importante que la modélisation géométrique. Nous avons nommé cette activité *modélisation lumineuse*, et son résultat la *description lumineuse* de la scène. Nous englobons également dans cette activité la saisie des points de vue de référence (objets de type `Camera`) et la saisie des correspondances entre couleurs et matériaux. La modélisation lumineuse vient juste avant la simulation dans l'organisation des projets, et elle est supposée

définir toutes les données qui n'ont pas été établies lors des modélisations photométrique et géométrique précédentes. La description s'appuie sur trois formats de fichiers :

- `*.lumiere` : un fichier décrit l'ensemble des sources lumineuses pour une scène.
- `*.cameras` : un fichier décrit l'ensemble des caméras prédéfinies pour une scène.
- `*.couleurs` : un fichier contient l'ensemble des couples (couleur/matériau) pour une scène.

Dans les trois cas, les données ne semblaient pas suffisamment nombreuses pour nécessiter plusieurs fichiers par scène, comme c'est le cas pour la géométrie. De plus, et particulièrement dans nos collaborations avec l'industrie de l'éclairage, il est courant d'avoir plusieurs projets d'illumination pour une même scène. Il est donc pratique de représenter chaque projet par un fichier `*.lumiere` séparé.

Malgré le nombre relativement faible de sources lumineuses par comparaison avec les primitives, certains projets d'illumination peuvent tout de même compter plusieurs centaines de sources. Il nous a paru souhaitable de les organiser, ainsi que les caméras, à l'aide des composants déjà définis pour la géométrie. Les composants architecturaux, d'abord créés pour la géométrie, sont en fait un moyen commode pour organiser toute la description de la scène. Nous considérons donc la scène comme un graphe de composants, lesquels contiennent à la fois des primitives, des sources lumineuses et des caméras. Cette idée simple a posé deux problèmes :

- On ne retrouve pas dans la description des sources lumineuses des modèles répétitifs comme c'est le cas pour la géométrie. Les composants partagés que nous avons évoqués dans la description géométrique ne contiennent pas forcément les mêmes sources lumineuses. Du point de vue des sources et des caméras, il faudrait donc dupliquer tout ces composants, et la structure sous-jacente n'est pas un graphe mais un arbre.
- Normalement, dans notre approche de la persistance, un objet de type `Composant` est censé être transformé en champ, lequel contient les sous-champs imbriqués décrivant ses primitives et ses sources, et le champ en question doit être stocké dans un fichier. Cette description unitaire des composants est en fait incompatible avec la répartition des primitives, sources et caméras dans trois types de fichiers différents.

Pour concilier nos différents objectifs, nous avons décidé que la description des objets de type `Composant` serait répartie. Contrairement aux autres classes persistantes de la plate-forme, la classe `Composant` est dotée de trois opérations pour produire des champs, selon le type du fichier destinataire. De même, elle est dotée d'un constructeur d'objet qui utilise les champs lus dans les fichiers `*.geo`, et de deux autres opérations pour compléter cette construction à l'aide des champs issus des fichiers de type `*.lumiere` et `*.cameras`. De plus, le nom des composants utilisés dans la description lumineuse comprend le nom de tous les composants pères jusqu'au composant racine : il ne s'agit pas d'une adresse de nœud dans le graphe, mais plutot dans l'arbre déplié équivalent.

Au bout du compte, les fichiers `*.geo` contiennent la description d'un *graphe* de composants et des primitives attachées à ces composants, alors que les fichiers `*.lumiere` et `*.cameras` contiennent la description d'un *arbre* de composants équivalent et des sources et caméras attachées. Cette répartition des données entre un graphe et un arbre équivalent a bien fonctionné

lors de l'utilisation de la plate-forme dans les différents projets, mais il faut reconnaître qu'elle a été délicate à programmer, surtout si l'on considère le conflit supplémentaire avec les inclusions de fichiers.

Vues

Les objets de classe `Vue` sont les seules informations qui transitent entre le programme central de simulation et les programmes qui manipulent les images. Comme nous le verrons à nouveau dans la section 2.3 sur les programmes, les vues ne contiennent pas seulement les résultats de la simulation et du rendu, elles peuvent aussi servir à véhiculer le point de vue extrait d'une photographie (cf. figure 4.8).

Chaque vue est représentée par un fichier `*.vue`, qui peut être incomplet et constitué tour à tour l'entrée ou la sortie des programmes `Synthese`, `Analyse` et `Traitement`. Nous avons décidé de créer un fichier par vue pour que ces dernières soit autonomes au sein du système Unix, et qu'il soit aussi facile de les déplacer que les fichiers images traditionnels.

Etant donné que plusieurs vues peuvent inclure une même image, et étant donné l'encombrement de ces dernières, nous avons dû renoncer à inclure une copie des images utilisées dans chaque vue. Il était donc nécessaire de créer également un fichier pour chaque image. Cette approche maintient l'autonomie de chaque vue, mais complique son déplacement : lorsque l'on déplace un fichier `*.vue`, il faut également copier les images qui sont utilisées par la vue.

En toute rigueur, nous aurions dû créer un format `*.image` pour les objets de classe `Image`, qui auraient contenu une liste de champs représentant les valeurs des pixels. Cela nous a semblé inadapté de représenter une image par un fichier ASCII, qui serait de toute façon inexploitable, et nous avons préféré nous en tenir au format binaire traditionnel `*.ppm`. Nous avons donc exceptionnellement toléré un format binaire externe au sein des formats objets de la plate-forme.

Récapitulation

Nous récapitulons ci-dessous les formats prédéfinis pour la plate-forme Graph'IS, avec les classes des objets persistants qu'ils représentent ou contiennent :

Bibliothèque Photométrique

`*.materiau` : `Materiau`, `Fresnel`.

`*.source` : `TypeSource`.

`*.optique` : `Solide`.

`*.lampe` : `Lampe`.

Description Géométrique

`*.geo` : `Composant`, `Primitive`.

Description Lumineuse

`*.lumiere` : `Composant`, `Source`.

`*.cameras` : `Composant`, `Camera`.

```
*.couleurs : Couleurs.
```

Vues

```
*.vue : Vue, Niveau, Masque.
*.ppm : Image.
```

L'établissement d'une correspondance entre les classes persistantes de la bibliothèque d'une part, et les blocs d'informations de la figure 4.8 d'autre part, nous a demandé de nombreux aménagements. Nous avons ajouté de nouvelles classes (`Composant`, `Couleurs`) et nous avons admis la distribution d'un objet sur plusieurs champs (`Composant`) ou encore l'introduction d'un format externe (`*.ppm`). Il nous a également semblé naturel de traduire certains objets par un fichier plutôt que par un champ, lorsque ces objets étaient susceptibles d'être gérés au niveau du système d'exploitation. Nous obtenons ainsi :

- un découpage des données compatible avec les étapes et les activités d'un projet d'illumination,
- un ensemble de fichiers ASCII lisibles, que les utilisateurs peuvent éditer directement lors de leurs tests,
- un contenu organisé plus ou moins conformément à la structure objet sous-jacente, ce qui permet aux programmeurs d'implanter la gestion des entrées/sorties de façon très modulaire, en attribuant à chaque classe la gestion des champs qui la concerne.

Ce mariage entre le paradigme objet et la structure plus traditionnelle d'un système de fichiers n'a pas toujours été simple à implémenter. Dans la présentation des résultats (section 3) nous aborderons certains des problèmes auxquels nous avons été confrontés, et nous tenterons de faire le point dans la discussion finale (section 4).

2.3 Programmes

Au début de la section 2, nous avons identifié un certain nombre d'activités plus ou moins indépendantes, dont nous avons tiré des programmes et des blocs de données, résumés dans la figure 4.8. Après avoir présenté les formats de données, nous allons maintenant décrire les responsabilités des différents programmes.

Nous avons placé au centre du système le programme qui nous concerne le plus directement, `Synthese`, lequel est chargé de simuler les transferts radiatifs dans la scène. Les programmes placés en haut de la figure 4.8 (page 72) sont chargés de préparer les données d'entrée de `Synthese`, et les programmes placés en bas sont chargés de traiter ses résultats. Tous ces programmes ont été dotés d'un interpréteur de commandes plus ou moins similaire à celui d'Unix, afin de présenter une syntaxe déjà familière aux utilisateurs. Un ensemble de classes, réunies dans la catégorie `interface`, et intégrées à la bibliothèque de classes, facilite la mise en place des interpréteurs. Avant de revenir plus en détail sur la distribution des responsabilités et des commandes parmi les programmes, nous allons décrire cette catégorie.

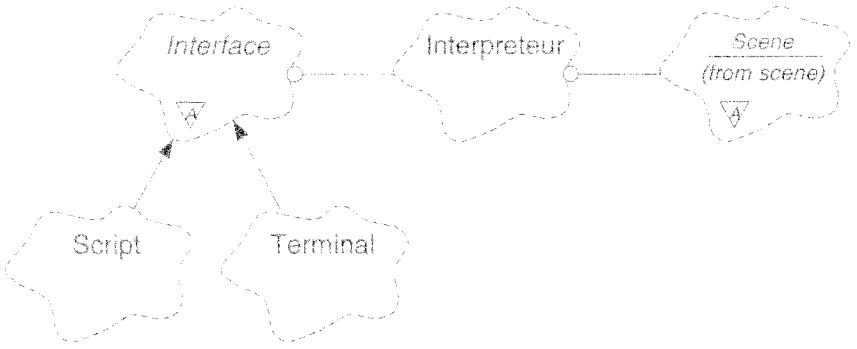


FIG. 4.11 – *Classes de la catégorie* `interface`

Catégorie `interface`

La figure 4.11 illustre le dialogue entre l'utilisateur et le système logiciel. À gauche, un objet de classe `Interface` reçoit les commandes de l'utilisateur. À droite, l'objet de classe `Scene` sert d'intermédiaire avec tous les autres objets du système. Au centre, l'interpréteur transforme les commandes transmises par l'interface en appels de méthodes à l'adresse de la scène.

Pour chaque programme, la classe `Interpreteur` est instanciée une fois. On confie à l'objet instancié la liste des noms de commandes et une liste de fonctions associées. Lorsqu'il reçoit une commande, l'interpréteur effectue d'abord une analyse syntaxique, à la façon d'un interpréteur Unix (*shell*), et identifie ainsi le nom de la commande, ses options et ses arguments. L'interpréteur transmet le tout à la fonction qui est associée à la commande, et cette fonction appelle les opérations appropriées de l'objet `Scene`.

La classe `Interface` est donc chargée des interactions avec l'utilisateur. Elle transforme les interventions de ce dernier en commandes pour l'interpréteur. La superclasse, abstraite, définit simplement la relation avec l'interpréteur, et déclare la seule méthode publique, `run`, qui constitue la boucle de contrôle principale du programme. Nous fournissons dans la bibliothèque deux sous-classes :

- `Terminal` : interface de base, proche d'un interpréteur Unix, utilisant le clavier comme entrée et comme sortie le terminal où le programme a été lancé. Cette interface saisit les commandes de l'utilisateur et les transmet directement à l'interpréteur. Elle est également capable de fournir une aide en ligne.

- `Script` : interface spécialisée dans le traitement des fichiers de commandes à exécuter en tâche de fond ; les sorties sont redirigées dans un fichier « trace » ; l'objet de type `Script` ne traite pas certaines commandes qui sont inutiles dans ce type d'utilisation, et produit surtout un affichage adapté pour la relecture ultérieure de la trace d'éxécution.

Si un utilisateur veut doter un programme d'une interface graphique, il est prévu qu'il crée une nouvelle sous-classe de la classe abstraite `Interface`. En rendant obligatoire le passage par le langage de commande et l'interpréteur, nous avons voulu isoler l'interface utilisateur de la partie orientée objets de nos programmes, et garder ainsi toute liberté de faire évoluer nos classes, en ne mettant à jour que l'implémentation des commandes et en préservant la validité des interfaces.

La procédure principale d'un programme bâti à l'aide de la catégorie `interface` est toujours constituée de la même façon :

1. Création d'un objet de classe `Scene`.
2. Création d'un objet de classe `Interpreteur`, à qui on fournit la liste des noms de commandes, la liste des procédures associées, et l'adresse de la scène.
3. Selon les options positionnées lors de l'appel du programme, création d'une interface de type `Terminal` ou `Script`, à qui l'on donne l'adresse de l'interpréteur.
4. Lancement de l'interface (méthode `run()`).

Voyons à présent les programmes prédéfinis qui constituent la plate-forme Graph'IS.

Programmes de modélisation

Les trois programmes de modélisation (dont le nom commence par `Mod`) pourraient sembler secondaires. Nous avons tenu à en dire quelques mots pour plusieurs raisons : la préparation des données est nécessaire si l'on veut obtenir des images d'une bonne qualité et travailler avec des scènes complexes ; par ailleurs, ces programmes peuvent également proposer des précalculs qui accéléreront le travail ultérieur de `Synthese`.

Le premier programme, `ModPhoto`, est essentiellement dédié à la physique. Il est utilisé pour mettre au point la bibliothèque photométrique et les modèles implantés dans les catégories `photometrie` et `colorimetrie`. Pour analyser ces modèles, et en particulier évaluer les associations [matériau/source], nous avons doté le programme d'un module de rendu spécialisé, à base de lancer de rayons, qui ne peut traiter que des scènes géométriquement très simples mais avec une meilleur précision physique que `Synthese`. Les principales fonctionnalités de `ModPhoto` sont :

- *visualisation de la couleur d'un matériau*, éclairé d'une lumière blanche, selon différents modèles colorimétriques,
- *visualisation de la réflectance d'un matériau*, en trois dimensions,
- *visualisation de la couleur d'un type de source*, selon différents modèles colorimétriques,
- *visualisation du solide photométrique d'un type de source*, en trois dimensions,
- *visualisation d'une scène prédéfinie*, composée d'une primitive dont on peut changer le matériau et d'une source dont on peut changer le type.

Le second programme, `ModGeo`, n'est pas censé être un modeleur géométrique proprement dit. Il y a suffisamment de modeleurs commerciaux de qualité pour les utiliser dans nos projets. `ModGeo` est donc plutôt une passerelle permettant de récupérer et d'optimiser des données géométriques externes. La problématique de la simulation de la lumière est différente de celle de la modélisation géométrique, et les données importées sont le plus souvent partiellement inadaptées

à la simulation. Le rôle de `ModGeo` est de nettoyer, structurer, enrichir la description géométrique d'une scène. Les principales fonctionnalités prévues sont :

- *chargement/sauvegarde* d'une description géométrique,
- *affichage d'informations* : nombre total de primitives, taux de facettes rectangulaires,
- *nettoyage* : suppression des points alignés sur les contours polygonaux, suppression des primitives de surface quasi nulle,
- *réorganisation* des composants architecturaux.

Enfin, le programme `ModLum` doit permettre de créer ce que nous avons appelé la description lumineuse d'une scène, à savoir l'ensemble des sources, des caméras, et des couples [couleur/matériau]. Cette fois, aucun logiciel commercial ne nous a paru approprié, et nous nous sommes résolus à développer des fonctionnalités de modélisation proprement dite. Dans l'interface du programme, la scène est présentée à l'utilisateur comme une hiérarchie de type Unix, où les composants seraient les répertoires et les sources et caméras seraient les fichiers. L'utilisateur peut circuler dans les composants qui constituent la scène comme il circule dans les répertoires Unix, et manipuler sources et caméras qu'ils contiennent. Les principales fonctionnalités de `ModLum` se résument à :

- *chargement d'une description géométrique,*
  chargement/sauvegarde d'une description lumineuse,
- *circulation dans l'arbre des composants,*
- *création/modification/duplication de sources,*
- *création/modification/duplication de caméras,*
- *création/modification/duplication de correspondances [couleur/matériau],*
- *affichage d'informations* : nombre total de sources, nombre de types de sources utilisés, etc.

`ModPhoto`, `ModGeo` et `ModLum` sont donc les trois programmes «secondaires» dédiés à la manipulation des données d'entrée de la plate-forme Graph'IS. Ces programmes prennent toute leur importance quand il s'agit d'appliquer nos travaux à des projets d'architectures en vraie grandeur.

Programme de simulation

Le programme `Synthese` exploite les informations préparées par les programmes précédents pour simuler les transferts radiatifs dans une scène complexe. Par scène «complexe», nous entendons une scène comprenant une très grande quantité de primitives et de sources. Pour simuler ce type de scène, nous pensons que la résolution numérique doit obligatoirement être itérative, et nous avons défini ainsi la liste des commandes de base :

- *charger* les descriptions géométriques et lumineuses,

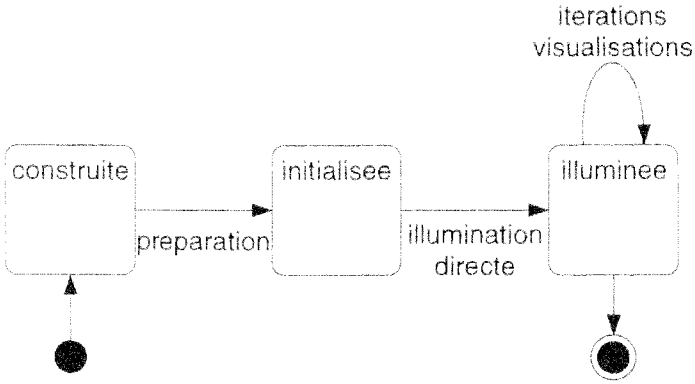


FIG. 4.12 – *États de la classe* scene

- *préparer* les primitives,
- *simuler* l'illumination directe,
- *itérer* la simulation de l'illumination indirecte,
- *visualiser* la scène.

Avant de pouvoir simuler les transferts radiatifs, il est nécessaire de ramener les primitives à une représentation surfacique et discrète : c'est le rôle de la commande de préparation des primitives. Par ailleurs, suite à notre distinction entre primitives et sources lumineuses, nous séparons également le calcul des illuminations directes et indirectes. L'illumination directe est toujours intégralement évaluée, ce qui permet d'initialiser les primitives. Cette première étape est suivie par la simulation itérative de l'illumination indirecte, dans laquelle il n'est plus nécessaire de prendre en compte les sources. La visualisation de la scène peut être demandée quelle que soit l'avancée de l'illumination indirecte, ce qui constitue une façon de suivre l'évolution des calculs. Les commandes d'itération de l'illumination indirecte ou de visualisation peuvent être requises aussi souvent que nécessaire par l'utilisateur.

L'enchaînement de toutes ces commandes et les états successifs de la scène sont résumés dans la figure 4.12. Lorsque l'on construit la scène, on utilise la représentation habituelle par composants architecturaux, mais dès que la préparation de la scène est entamée, les composants sont oubliés et l'ensemble des sources et des primitives est transmis à l'objet de classe Espace, chargé des calculs de visibilité. On transite ainsi d'une organisation de la scène adaptée à la modélisation vers une organisation plus efficace pour la simulation. Voici ci-dessous un exemple de script de commandes, semblable à ceux que nous avons utilisés :

```
charger louvre.geo
charger louvre.lumiere
charger louvre.cameras
charger louvre.couleurs

composant /louvre/facade_nord
```

```
mailler 0.12 2
subdiviser 6 6 6 8

illuminer

iterer 50
visualiser /louvre/camera_face vue1
iterer 50
visualiser /louvre/camera_face vue2
```

Les groupes de commandes de ce script sont conformes à l'enchaînement [construction⇒ préparation⇒illumination⇒itérations]. Certaines étapes réclament quelques commentaires :

- *Construction* : la bibliothèque photométrique n'est pas chargée explicitement par l'utilisateur ; les fichiers nécessaires sont recherchés automatiquement par la catégorie `photometrie` en fonction des matériaux et des types de sources réclamés par les objets `Primitive` et `Source`.
- *Préparation* : dans cet exemple, l'utilisateur limite la simulation à une sous-partie de la scène (la façade nord), lance le maillage des primitives, et lance également la préparation de l'objet `Espace` (commande `subdiviser`).
- *Itérations* : l'utilisateur demande deux séries de 50 itérations, et calcule à chaque étape une vue depuis la caméra nommée `camera_face`.

L'organisation de la scène par composants n'est pas exploitée par les calculs. Elle permet cependant de désigner une sous-partie de la scène sur laquelle l'utilisateur veut restreindre la simulation. Si ce dernier sait découper la scène en sous-parties indépendantes entre lesquelles les échanges lumineux sont négligeables, alors il peut calculer ses vues par morceaux, et les recomposer ultérieurement à l'aide du programme `Traitement`. C'est une façon supplémentaire de réduire la complexité des calculs, tout en préservant la validité physique. Nous allons à présent aborder les programmes liés à la manipulation des vues.

Programmes liés aux Images

Lors de nos collaborations avec l'industrie, il est rapidement apparu nécessaire de faire appel à des méthodes issues du domaine de la vision, en particulier pour l'incrustation des images de synthèse dans des photographies numérisées. Les deux derniers programmes de la figure 4.8 sont chargés de fournir les fonctionnalités de ce type, et toutes les méthodes de manipulation d'images en général.

Le programme `Analyse` est chargé de l'analyse des photographies ou des films pour fournir des informations à `Synthese`. Il vient en amont du programme de simulation. Ses fonctionnalités principales sont :

- *reconnaître un composant architectural* ou une de ses parties dans une image (par détection de contours),

- *retrouver le point de vue d'une image*, en fonction des éléments reconnus,
- *calculer les masques* définissant les différentes zones de l'image.

Le programme `Traitement` vient au contraire en aval de la simulation. Il sert à recomposer les images quand la scène a été simulée par morceaux, et à incruster le tout dans une photographie de fond. L'application de filtres accroît l'adéquation entre les photos et les images de synthèse. Les fonctionnalités principales sont :

- *modification des caractéristiques colorimétriques* d'une image (cœfficient gamma, coordonnées chromatiques),
- *combinaison linéaire* d'images de synthèse de même point de vue,
- *incrustation* dans une photographie,
- *application de filtres.*

Traducteurs

Pour conclure cette revue des programmes prédéfinis de la plate-forme, nous voulons signaler que nous avons finalement choisi d'implémenter dans des petits programmes séparés l'importation des données externes. Nous avons normalisé l'ensemble de formats de fichiers manipulés dans la plate-forme, mais il est évidemment nécessaire de pouvoir récupérer les données exterieures à cette dernière. Dans un premier temps, nous avions intégrer des fonctionnalités d'importation dans les différents programmes de modélisation de la plate-forme. Ce choix a posé quelques problèmes, notamment :

- l'ajout des fonctionnalités d'importation/exportation alourdit des programmes déjà trop chargés,
- l'intégration d'un nouveau format nous oblige à intervenir sur les programmes principaux.
- le découpage des informations externes est parfois différent du nôtre, et la traduction d'un type de fichier difficile à attribuer à un programme plutôt qu'à un autre.

Nous avons donc finalement implanté l'importation et l'exportation de données externes dans des programmes séparés et développés au cas par cas.

3 Expérimentation et résultats

Dans la première partie de ce chapitre, nous avons présenté ce qui constitue la structure de la plate-forme Graph'IS : un ensemble de classes abstraites pour uniformiser l'ossature de nos logiciels (section 1), ainsi que des formats de fichiers et des programmes prédéfinis pour uniformiser le flot des données (section 2). À présent, nous voulons montrer comment cette structure a été utilisée par les chercheurs de notre équipe et par les ingénieurs successivement chargés du transfert industriel.

Après trois ans de modifications tous azimuths par quelques dizaines de programmeurs aux profils variés, il est devenu délicat de décrire simplement l'état final de la plate-forme sinon qu'elle regroupe environ 300 classes et 200000 lignes de C++ et de commentaires. Nous avons choisi d'articuler la présentation ci-dessous autour de quelques utilisateurs et de quelques projets typiques. Dans chaque cas, nous illustrerons ce qui a été réutilisé avec succès, ainsi que les obstacles rencontrés et les ajouts qui ont permis de les contourner.

3.1 Recherche en simulation des transferts radiatifs

Chaque chercheur, en fonction de son thème de recherche, s'est vu confié l'implémentation de certaines classes. Il s'agissait le plus souvent de fournir une sous-classe standard et robuste pour une classe abstraite qui le concernait directement (colorimétrie de Meyer, réflexion diffuse, facettes, sources lumineuses ponctuelles, radiosité progressive). Les classes utilitaires ou d'intérêt général ont été développées par moi-même ou par un ingénieur à l'occasion d'un transfert industriel. Nous avons ainsi disposé assez rapidement d'un jeu de classes permettant de réaliser des simulations complètes. Une fois passée cette première étape, chacun a pu se concentrer sur sa problématique particulière, dont voici ci-dessous quelques exemples.

Propriétés radiatives des sources lumineuses

Pascal Deville a mené des travaux de recherche sur la modélisation des sources lumineuses et notamment sur les modèles colorimétriques [Dev96]. Le modèle de Meyer [Mey88b], un des plus reconnus, s'était avéré insuffisant parce qu'il reposait sur l'hypothèse que les spectres étaient continus. Or, pour une illumination extérieure, les éclairagistes utilisent régulièrement des lampes dont le spectre est discontinu et concentré en quelques raies. Pascal Deville a proposé de découper le domaine spectral en bandes, choisies de telle sorte que l'hypothèse de continuité soit respectée dans chaque bande prise séparément.

Ce nouveau modèle colorimétrique a pu être implémenté en tant que sous-classe de la classe abstraite `Onde`. la nouvelle classe `Bandes` possède une différence importante avec sa concurrente : les longueurs d'ondes de référence ne sont pas prédéfinies par le modèle mais évaluées à partir d'une analyse des spectres des lampes utilisées dans la scène. Si de nouvelles lampes sont ajoutées ultérieurement, elles ne seront pas prises en compte dans la découpe des bandes, ce qui s'ajoute à la perte d'information liée à la conversion des instances de `Spectre` en instances d'`Onde`. Pour limiter cette perte, on essaie de prédéfinir l'ensemble des lampes possibles et on repousse autant que possible la conversion des instances de `Spectre` en `Onde`, jusqu'au commencement effectif de la simulation.

L'implémentation du nouveau modèle nous a posé un problème inattendu. Les méthodes de simulation manipulent des objets de type `Onde` comme elles manipuleraient des réels ou des entiers. Il y a une très grande quantité d'instructions demandant la construction d'une nouvelle instance d'`Onde`, disséminées partout dans le code. Il est inconcevable de les corriger toutes à chaque fois que l'on souhaite changer de modèle, et le polymorphisme ne peut intervenir qu'*après* la construction de ces objets, donc n'englobent pas les constructeurs. La parade consiste à implémenter, pour chaque sous-classe de `Onde`, une classe supplémentaire dont la seule tâche est de construire des objets conformes au nouveau modèle. Une instance de ce type de classe est souvent appelée une *fabrique* (*factory*). En fournissant une fabrique aux méthodes de simulation, celles-ci deviennent capables de créer elles-mêmes des instances de la nouvelle classe développée

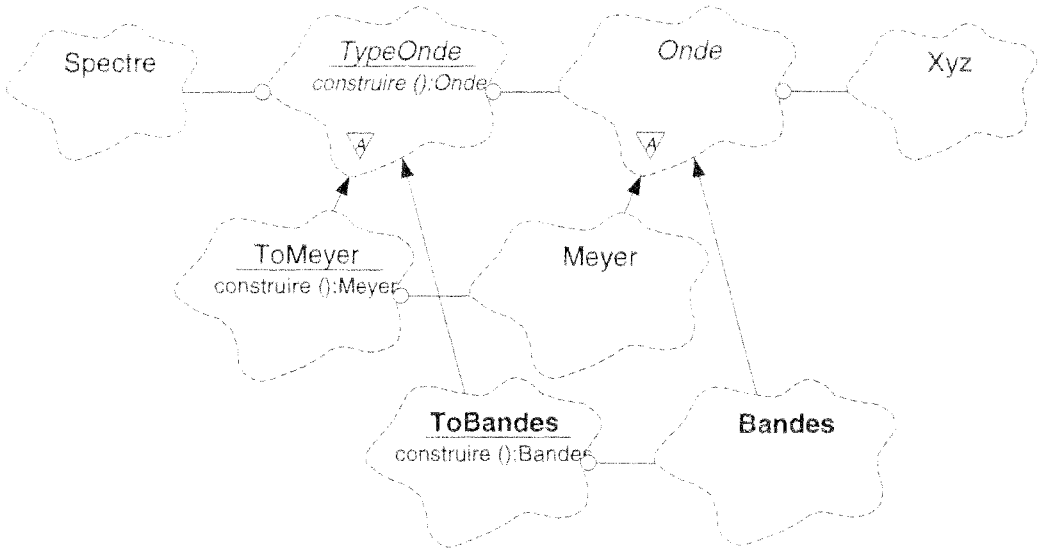


FIG. 4.13 – *Nouveau modèle colorimétrique*

par le programmeur.

La figure 4.13 met en évidence les nouvelles classes qui ont été créées. `TypeOnde` joue le rôle de fabrique abstraite et a été ajoutée à la bibliothèque, ainsi que sa sous-classe standard `ToMeyer`. Tous les appels au constructeur `Meyer` dans la bibliothèque ont dû être remplacés par des appels à la méthode virtuelle de construction de la classe `TypeOnde`. Pour sa part, Pascal Deville a créé les classes `ToBandes` et `Bande` et a pu réutiliser directement toutes les classes des autres catégories pour produire des images avec son nouveau modèle.

Propriétés radiatives des surfaces

La section précédente traitait de la problématique des sources lumineuses. Des travaux complémentaires ont été menés par Dorothée Schulz sur les propriétés radiatives des surfaces [SC96]. L'objectif de ces travaux étant pour une grande part une comparaison et une évaluation des modèles de réflexion de la littérature, Dorothée Schulz a énormément exploité la plate-forme Graph'IS et plus particulièrement la catégorie `photometrie` et le programme `ModPhoto`.

La figure 4.14 illustre une partie des classes qui ont été implémentées pour expérimenter les différents modèles. Le modèle de Ward [War92], comme celui de Cook et Torrance [CT82], est implanté à l'aide de la classe `Reflectance` et d'une sous-classe de `RepSpat`. Par contre, l'implantation du modèle de He et al. [HTSG91] nécessite une classe supplémentaire `He` héritant de `Reflectance`. En effet, il est impossible d'isoler une composante purement spatiale dans la formulation de ρ_l, et donc impossible de se conformer complètement à l'interface de `RepSpat`. Il est également nécessaire d'utiliser une instance supplémentaire de `RepSpec`. Grâce à l'héritage, Dorothée Schulz a réutilisé en partie la classe `Reflectance`, et redéfini le strict nécessaire, en particulier le calcul de ρ_l.

La présence de plusieurs niveaux de classes abstraites rend possible de nombreux assemblages d'objets, y compris des assemblages aberrants qui ne correspondent à aucun modèle physique.

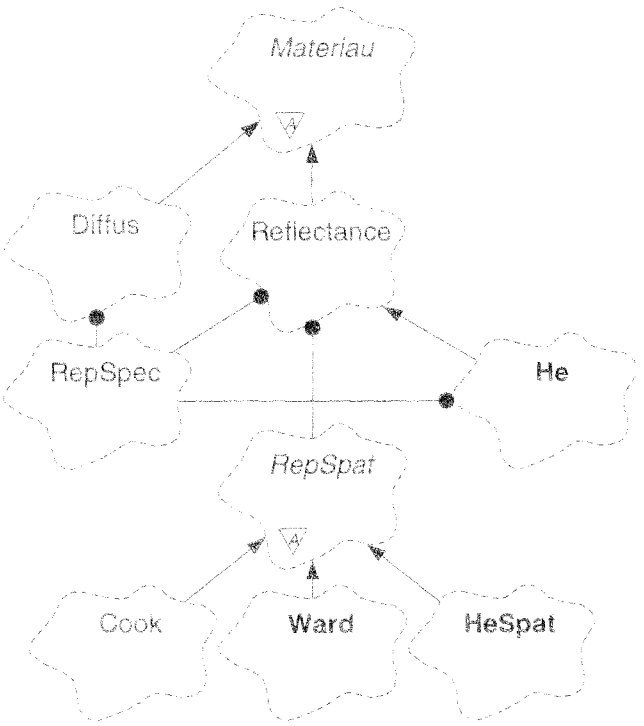


FIG. 4.14 – *Classes implémentant les différents modèles de réflectance*

On pourrait par exemple aboutir à un objet de classe `He` combiné à un objet de classe `Ward`. Les assemblages correspondant aux différents modèles de réflectance sont présentés dans la figure 4.15. Il convient d'être vigilant lors de la construction pour éviter les assemblages illicites. Cette construction avait été initialement implémentée sous la forme d'une méthode statique de `Materiau`. Comme dans le cas des modèles colorimétriques, il devint nécessaire d'insérer une *fabrique* abstraite pour que la méthode de construction puisse être librement redéfinie par les utilisateurs.

La possibilité de mal combiner les classes est fortement liée au fait que nous avons doté à tort la classe `Reflectance` de constructeurs. En effet, cette classe a été initialement prévue pour faciliter le codage des modèles, et non pour être directement instanciée. En l'absence de constructeurs, Dorothée Schulz aurait été amenée à créer une sous-classe de `Reflectance` pour chaque nouveau modèle, comme dans la figure 4.16. Dans une telle implémentation, la construction des différents objets liés à un modèle peut être prise en charge par la classe principale du modèle. Le rôle de la fabrique se limite alors à invoquer le constructeur de la classe principale, et il est plus facile d'ajouter de nouveaux modèles.

Nous voulons finalement souligner les difficultés liées à la nature à la fois *abstraite* et *persistante* de la classe `Materiau`. Dans la description des formats de données, nous avons expliqué que toutes les données caractérisant un matériau donné étaient regroupées dans un fichier unique. La création des nouveaux modèles de réflectance a impliqué une pénible mise à jour de tous les fichiers `*.materiau`. L'édition des fichiers `*.materiau` peut difficilement être effectuée dans les programmes de la plate-forme Graph'IS, car chaque sous-classe ne détient qu'une partie de la description du matériau. Nous reviendrons sur cette problématique dans la section suivante.

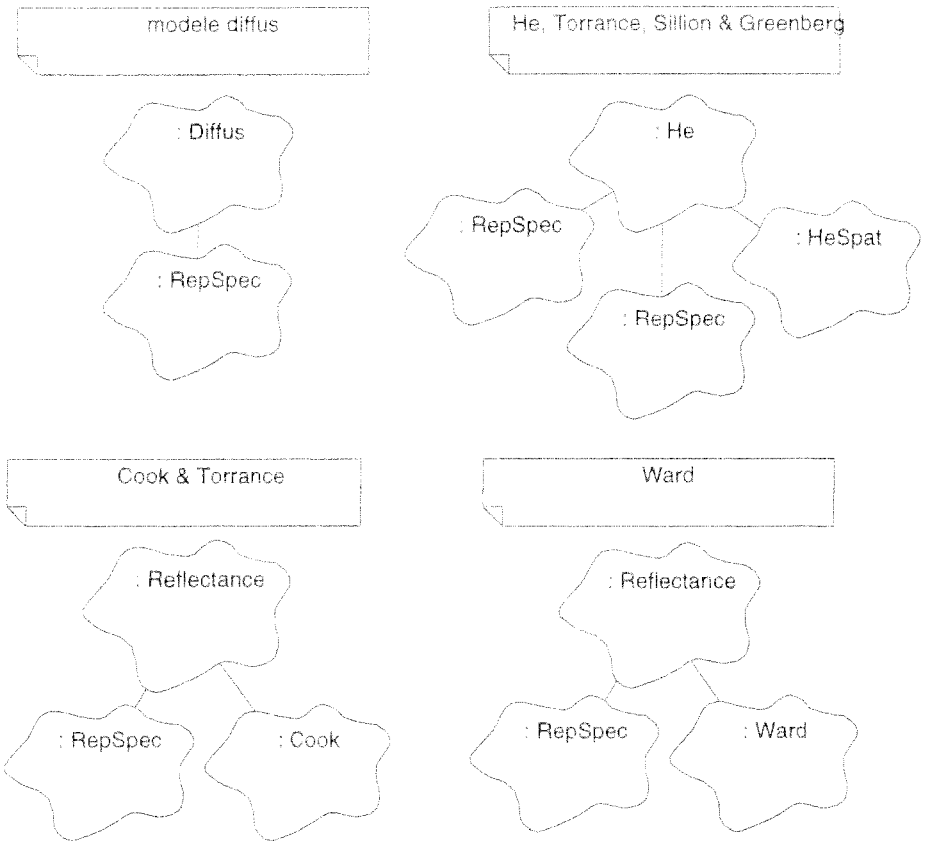


FIG. 4.15 – *Assemblages d'objets pour les différents modèles de réflectance*

Génération de séquences composées d'images de synthèses et d'images vidéo

La thèse de Christine Chevrier a porté sur «la mise au point, le test et l'intégration de diverses méthodes d'analyse et de synthèse pour la composition réaliste d'images de synthèse et d'images vidéo dans le cadre de la simulation de projets d'architecture et de projets urbains» [Che96].

Christine Chevrier a plus particulièrement exploité les catégories `images` et `scene` ainsi que les programmes `Synthese`, `Analyse` et `Traitement`. Pour pouvoir calculer et manipuler des séquences vidéo, elle a ajouté le programme `Sequence` et la classe du même nom. La même ossature `[Sequence-Vue-Niveau-Image]` est utilisée dans chaque programme, mais avec chaque fois des classes spécialisées pour les besoins particuliers du programme. La majorité de ces classes est illustrée dans la figure 4.17, où l'on peut identifier le programme associé à une classe à l'aide du suffixe accolé à son nom : `An` pour `Analyse`, `Tt` pour `Traitement`, `Dyn` pour le programme `Sequence`. Deux classes supplémentaires, `VueSyn` et `NiveauSyn`, contiennent la partie commune aux classes en `Tt` et aux classes en `Dyn`. Nous avons ici affaire à une réutilisation des implantations plutôt que des interfaces ; l'héritage sert à étendre des classes existantes plutôt qu'à les redéfinir. Suite à ce constat, les classes abstraites de la catégorie `images` ont été transformées en classes utilitaires contenant une implantation figée des fonctionnalités de base.

Les problèmes soulevés par ces travaux sont issus de l'extension des classes plutôt que de

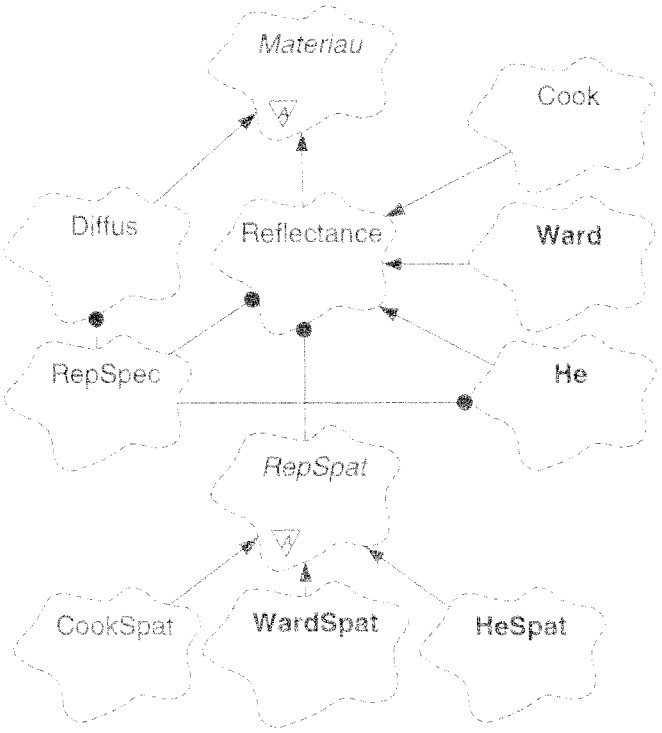


FIG. 4.16 – *Autre implémentation des modèles de réflectance*

l'usage relativement limité du polymorphisme. En particulier, le fait qu'un niveau donné possède des attributs différents selon le programme en cours complique singulièrement sa sauvegarde. Pour préserver toutes les informations liées à un niveau, nous nous sommes appuyés sur les *champs* et leurs *étiquettes*. Nous avions déjà prévu à l'origine qu'un constructeur puisse ignorer les champs qu'il ne reconnaît pas lors de l'interprétation d'un fichier. Dans le cas des instances de `Niveau`, les champs inconnus ne sont pas pris en compte mais nous avons néanmoins fait en sorte qu'ils soient stockés dans l'objet, et restitués tels quels lorsque l'objet est sauvegardé. Ce mécanisme, initialement créé pour préserver l'intégrité des objets de classe `Niveau`, s'est finalement avéré indispensable pour tous les objets persistants qui sont susceptibles d'être modifiés puis resauvegardés.

Le second blocage est venu du découpage des programmes. Le calcul des séquences d'images repose en grande partie sur l'interpolation des pixels, mais certains d'entre eux ne peuvent être évalués qu'à l'aide d'une méthode de visualisation classique. Christine Chevrier a choisi de placer dans le programme `Synthese` tous les calculs de visualisation effectués en trois dimensions, et de placer dans `Sequence` les interpolations en deux dimensions. Ce souci de respecter le découpage en programme que nous avions spécifié lui a sans doute posé des problèmes, en l'obligeant à fragmenter ses algorithmes. Un découpage plus adapté à ses besoins consisterait à avoir d'une part un programme de simulation des transferts radiatifs, capable de sauvegarder une carte en trois dimensions des éclairements de la scène, et d'autre part un programme de visualisation qui se chargerait des calculs d'images aussi bien que des calculs de séquences.

Pour terminer par un problème de compilation, ajoutons que l'inclusion d'une instance de `Camera` dans l'objet `Vue` nous a obligé a placé la classe `Vue` dans la même catégorie que la classe

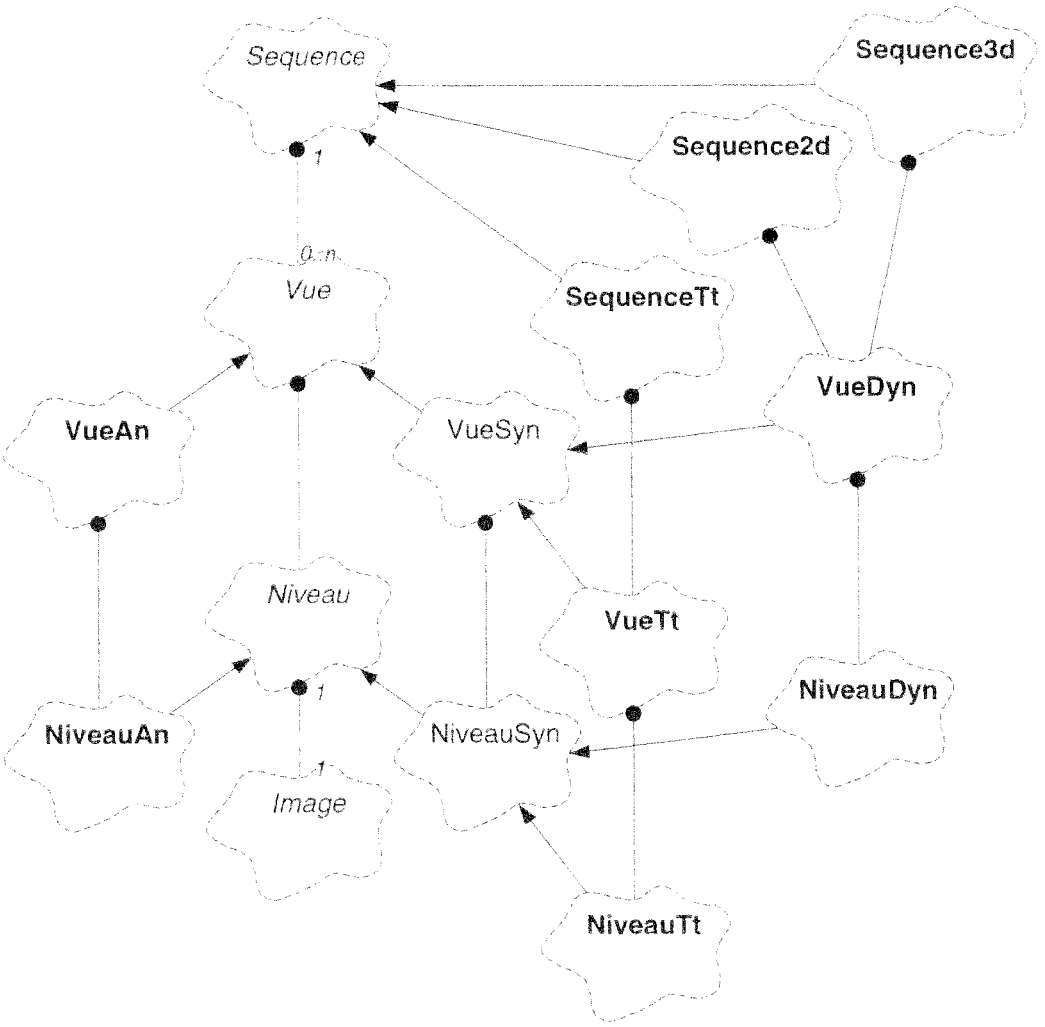


FIG. 4.17 – *Sous-classes pour l'incrustation d'images*

`Camera`, à savoir `scene`. En effet, placer la classe `Vue` dans sa catégorie naturelle, `images`, aurait créé une dépendance de compilation mutuelle entre les deux catégories. Ce problème est lié au fait que nous avons intégré dans la seule classe `Camera` les attributs de point de vue et l'algorithmie de visualisation. Pour y remédier, il suffirait de créer une classe séparée pour le point de vue et de substituer une instance de cette classe à la caméra actuellement incluse dans `Vue`.

Algorithmes de simulation

Le principal algorithme de simulation que nous avons implémenté est celui que proposa Wallace [WEH89]. Il a été intégré à la classe `Scene`, à l'exception des calculs de visibilité qui furent délégués à la classe `Espace` et optimisés à l'aide d'un arbre BSP.

Lorsque Slimane Merzouk a commencé ses travaux sur de nouvelles méthodes de simulation et sur leur parallélisation, il a regretté l'absence de classes spécifiques pour son sous-domaine de

recherche. Cette lacune est liée à plusieurs facteurs :

- les premiers travaux de l'équipe portaient sur les propriétés radiatives locales plutôt que sur l'algorithme global,
- l'utilisation de l'orientation objets nous a amené à intégrer les algorithmes dans les classes simulant les éléments statiques de la scène,
- la généralisation que nous avons présentée dans le chapitre 1 n'était pas encore suffisamment formalisée pour servir de support à la phase d'identification des classes clés (c'est l'équation de transfert qui nous a servi de support).

Vu les changements profonds qu'il fallait apporter à la plate-forme pour répondre à ses besoins, Slimane Merzouk a préféré commencer un nouveau système logiciel [Mer97], en reprenant le meilleur de Graph'IS et les propositions dont nous ferons état dans le chapitre 5.

3.2 Recherche appliquée à l'architecture

Isabelle Fasse, architecte de formation, étudiait au sein de notre équipe l'utilisation de la synthèse d'images pour la prise en compte de la lumière en architecture. Ses résultats sont publiés dans [Fas96]. Elle était l'utilisatrice principale de nos prototypes logiciels, et nous a permis d'expérimenter nos recherches sur des projets en vraie grandeur. Rappellons brièvement ce qui fait l'adéquation de la plate-forme à la simulation d'illumination :

- L'utilisation de mesures et de modèles physiques renforce la crédibilité des résultats.
- La découpe des programmes et des données facilite la collaboration entre les différents intervenants (architectes, éclairagistes, infographistes).
- Les composants architecturaux permettent de mieux organiser les primitives géométriques et les sources lumineuses d'une scène comprenant plusieurs milliers d'éléments.

La plate-forme Graph'IS, malgré une convivialité bien inférieure à celle des logiciels de synthèse d'images du commerce, a été effectivement utilisée par les architectes avec qui nous collaborons, parce qu'elle produisait des images plus conformes à la réalité. La palette de fonctionnalités disponibles, d'abord sommaire, a été facilement enrichie au fur et à mesure des nouveaux besoins, grâce à la structure d'objets sous-jacente. En fait, chaque nouveau projet a amené son lot de problèmes spécifiques. L'évolution de la plate-forme a été rythmée par l'enchaînement des projets, dont les plus représentatifs sont :

1. *La cour carrée du Louvre* : le tout premier défi, à partir duquel la structure fondamentale de la plate-forme a été imaginée.
2. *La place Stanislas à Nancy* : contrairement à la cour carrée du Louvre, l'espace n'est pas fermé, et il a fallu avoir recours à l'incrustation de photographies pour obtenir le réalisme souhaité.
3. *Les ponts de Paris* : notre souhait de réaliser un film a engendré des travaux importants sur l'incrustation d'objets virtuels dans une séquence vidéo.

4. *La cité interdite* : les fresques omniprésentes ont requis l'ajout d'un mécanisme de textures, et les images de jour ont requis l'intégration des sources lumineuses naturelles.

Au fil de ces projets successifs, nous nous sommes rapidement rendu compte que l'extension et l'évolution des programmes et de la bibliothèque de classes devaient s'accompagner d'une extension et d'une évolution des bases de données elles-mêmes. Certains des mécanismes implémentés dans la classe `Champ` ont facilité la réutilisation des données anciennes, mais très peu leur conversion aux nouveaux formats. Pour cette conversion, nous avons surtout tirer profit de l'attirail des outils d'édition Unix.

La flexibilité de la biliothèque de classes et celle des fichiers de données nous ont donc permis de franchir la majorité des obstacles. Il reste notamment à mieux exploiter la hiérarchie des composants architecturaux. Les bases de données géométriques que nous avons récupérées étaient rarement pourvue d'une structure hiérarchique que l'on puisse traduire en instances de `Composant`. Au mieux disposions-nous d'un découpage en fichiers, à partir desquels nous pouvions reconstruire une structure sommaire. Les utilisateurs de la plate-forme auraient aimé avoir la possibilité d'éditer ces composants et de réorganiser les primitives. Ils auraient aussi aimé visualiser les normales des primitives et pouvoir les retourner en cas d'erreur. Pour ces deux cas, nous n'avons pas pu les satisfaire.

Par ailleurs, l'effort que nous avons consenti sur le programme `ModLum` pour faciliter le placement et le réglage des sources lumineuses n'a pas été très rentable. En effet, il a apparu clairement que cette activité exige une interface graphique tout aussi évoluée que celle dont bénéficie les logiciels de modélisation géométrique. En l'absence d'une telle interface, les utilisateurs ont souvent préféré l'édition directe des fichiers `*.lumiere` plutôt que l'utilisation de `ModLum`.

En fait, toutes les fonctionnalités qui manquent à la plate-forme Graph'IS ou qui n'ont pas pleinement satisfait les utilisateurs sont celles qui exigent une interface graphique, voire une visualisation en trois dimensions. Le prototype industriel que nous avons développé avec Électricité de France a pallié ces insuffisances.

3.3 Transfert industriel

La plate-forme Graph'IS a servi de point de départ à plusieurs prototypes industriels, dont nous allons présenter les deux plus représentatifs. Le travail des ingénieurs responsables des transferts a essentiellement porté sur la conversion des données fournies par l'industriel, sur l'amélioration de l'interface et sur l'optimisation des performances. Cette optimisation s'est faite le plus souvent au détriment de la réutilisabilité, notamment en court-circuitant les classes abstraites jugée inutiles pour un domaine d'application donné. Les prototypes sont ainsi devenus incompatibles avec la plate-forme, mais leur réalisation a eu tout de même de nombreuses retombées pour les chercheurs :

- optimisation des classes utilitaires qui ont le plus d'impact sur les performances,
- implémentation de fonctionnalités secondaires pour la recherche mais indispensables pour mener une réalisation concrète à son terme,
- réalisation de campagnes de tests permettant de mettre à jour les bogues les plus enfouis,
- constitution d'une importante base de données.

Phostere : simulation d'illumination

Au delà de la production et de la distribution de l'électricité, *Électricité de France* poursuit quelques activités complémentaires dans le domaine de l'éclairage public, notamment les actions ponctuelles et prestigieuses du *Mécénat Technologique et Scientifique*, ou les travaux plus quotidiens du *Service Eclairage Public*. Ces activités ont amené EDF à collaborer activement avec le *CRAI* (*École d'Architecture de Nancy*) et avec notre équipe.

Nos premiers développements logiciels étaient intégrés dans un seul environnement, directement transmis à nos collaborateurs d'EDF. Après les premières versions, il devint nécessaire de dupliquer l'environnement en deux variantes indépendantes. La première, destinée aux chercheurs, privilégie la réutilisabilité ; elle fut baptisée *plate-forme Graph'IS*. La seconde, destinée à EDF, privilégie les performances ; c'est le prototype logiciel *Phostere*, dont nous allons parler à présent.

Ce prototype industriel fut d'abord l'occasion de mener une campagne de tests beaucoup plus conséquente que les précédentes, et de mettre ainsi à jour quelques bogues insoupçonnés. Certains aspects « secondaires », comme la gestion des affichages et des erreurs, ont ensuite été revisités, pour aboutir finalement à un produit sensiblement plus robuste et rigoureux que la plate-forme initiale.

L'optimisation des performances a notamment été obtenue par une spécialisation des classes géométriques basée sur les caractérisitiques des scènes qui sont soumises à *Phostere* : des édifices publics ou des monuments historiques, presque entièrement réductibles à des facettes planes, voire à des rectangles. La figure 4.18 donne un aperçu des multiples classes qui ont été créées à cette occasion. Héritage et polymorphisme y sont intensivement exploités pour gérer tous les cas particuliers. Pour compléter cette optimisation globale des calculs géométriques, la classe `Espace`, chargée des calculs de visibilité, a été réécrite avec le plus grand soin.

Par ailleurs, il est apparu que la classe `Onde` avait un impact important sur les performances, car les nombreuses manipulations de grandeurs physiques se font toutes à travers des instances de cette classe. En analysant les implémentations des différents modèles colorimétriques, nous avons constaté que les sous-classes de `Onde` étaient très semblables, à l'exception du nombre de longueurs d'ondes et de la conversion finale en triplet XYZ. En déplaçant cette conversion dans les sous-classes de `TypeOnde`, nous avons pu faire de la classe `Onde` une classe concrète ordinaire. Ainsi, les sous-classes de `TypeOnde` ne sont plus vraiment des fabriques mais plutôt des convertisseurs, et les instances de la classe `Onde` peuvent être manipulées comme des objets ordinaires, sensiblement plus vite.

Ensuite, au lieu de conserver l'ancienne allocation dynamique de mémoire, nous avons préféré doter chaque onde d'une zone mémoire de taille fixe et capable de convenir à n'importe lequel des modèles colorimétriques disponibles. Nous avons ainsi réduit d'un facteur dix le temps de création et de destruction de ces objets, mais en multipliant jusqu'à vingt fois l'espace occupé en mémoire. Pour corriger ce dernier défaut, nous avons également créé une classe `TabOnde` qui se présente comme un tableau d'ondes classiques mais qui gère plus intelligemment l'espace mémoire, de façon dynamique et globale. Au bout du compte, seuls les objets solitaires restent très volumineux, et il s'agit en majorité d'objets éphémères qui apparaissent le temps d'évaluer une expression. Grâce à une arithmétique de pointeurs plutôt scabreuse, mais encapsulée, l'interface de `TabOnde` est semblable à celle d'un tableau d'instances de la classe `Onde`. Un autre langage que C++ n'aurait sans doute pas permis d'obtenir une telle efficacité, c'est pourquoi nous tenions à

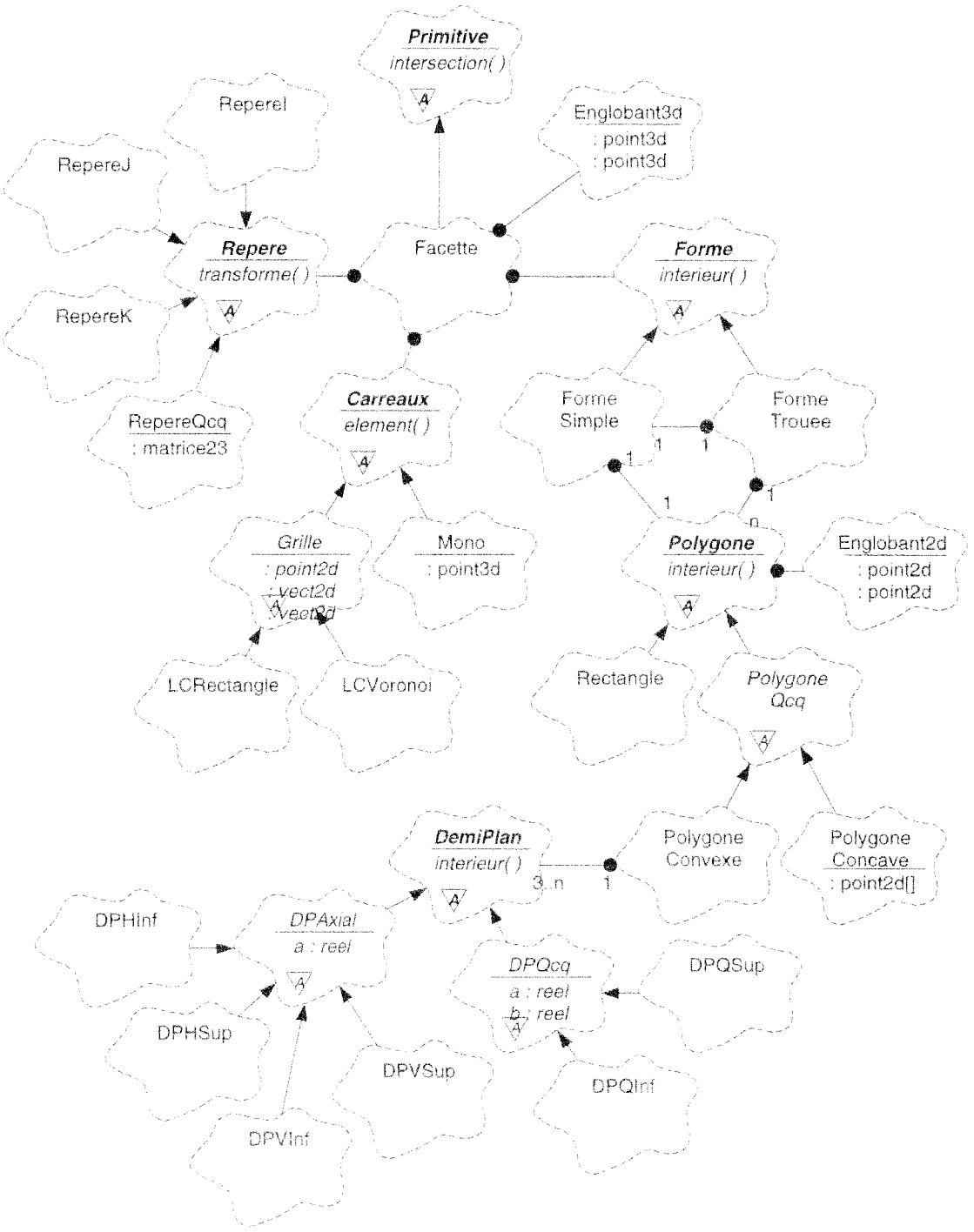


FIG. 4.18 – *Classes géométriques spécialisées*

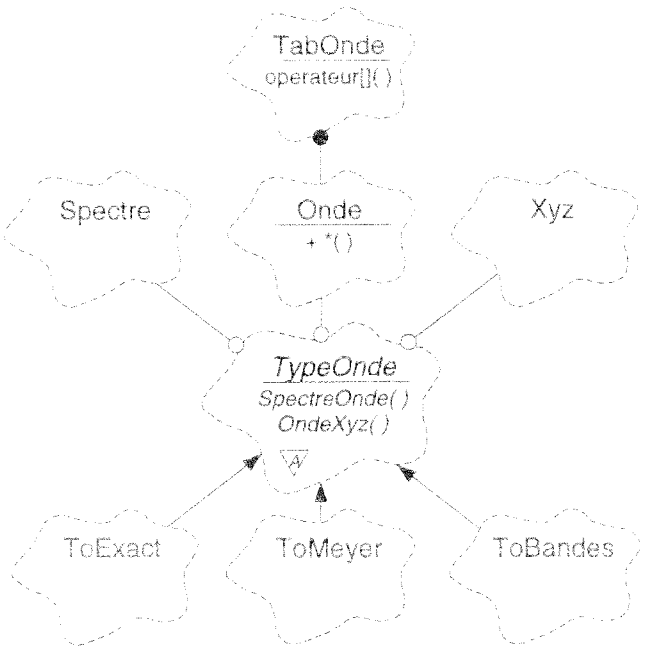


FIG. 4.19 – *Classes colorimétriques optimisées*

exposer ces détails. La nouvelle organisation des classes liées à la colorimétrie est résumée dans la figure 4.19.

En dehors des tests et des optimisations, le plus gros chantier a été l'ajout d'une interface graphique, qui fut sous-traitée à une société extérieure. En théorie, nous avions préparé cette évolution : il « suffisait » de créer une nouvelle sous-classe d'`Interface`. Dans la pratique, plusieurs aménagements supplémentaires ont été indispensables :

- De nombreuses commandes de bas niveau ont été ajoutées à l'ancien programme `ModLum`.
- Certaines commandes de haut niveau, qui n'avaient de sens que pour une interface textuelle, ont été déplacées vers les classes `Terminal` et `Script`.
- Un objet `Inspecteur` a été créé, permettant de lire le contenu de la scène sans passer par les commandes, afin de pouvoir afficher la scène dans la fenêtre graphique 3D avec des délais raisonnables.
- Tous les programmes sauf `Synthese` ont été regroupés en un seul programme `Interaction` qui a reçu l'interface graphique, alors que `Synthese` est resté un programme isolé et destiné à s'exécuter en tâche de fond sur un ordinateur différent.
- Les algorithmes de recherche automatique des fichiers dans les répertoires Unix ont été retirés de la bibliothèque de classes et modifiés.

Les constats les plus importants sont la trop grande lenteur d'un langage de commandes pour le dialogue avec la fenêtre graphique 3D, et la remise en cause du flot de données que nous avions spécifié. L'équipe chargée de réaliser l'interface graphique souhaitait regrouper toutes les

activités interactives dans un seul et même programme, et souhaitait organiser différemment les fichiers et les répertoires. Nous sommes aujourd'hui convaincus que l'organisation des fichiers et des répertoires relève également de l'interface utilisateur, et que les classes de la bibliothèque centrale doivent se limiter à la manipulation de champs.

Phostere a été mis en œuvre sur plusieurs des projets que nous citions à propos des travaux d'Isabelle Fasse. Son interface graphique et ses performances en font un produit à la fois plus convivial et plus efficace que la plate-forme Graph'IS. Grâce à Phostere, EDF a d'ores et déjà réalisé de nombreuses simulations d'illuminations, et largement exploité les images obtenues pour communiquer autour de ses projets de prestige.

Une des évolutions envisagées concerne la classe `Scene`, qui est chargée de gérer les éléments qui la composent et de mettre en œuvre la simulation. Comme elle est de surcroît le seul intermédiaire entre l'interpréteur de commandes et le reste du système, cette classe a progressivement atteint une taille déraisonnable. Un des moyens de l'alléger est d'augmenter le nombre d'intermédiaires avec l'interpréteur de commandes et d'externaliser la simulation dans une classe séparée, comme nous l'avons déjà envisagé dans la section sur les algorithmes de propagation.

La seconde évolution importante sera la définition d'un format permettant de décrire la distribution de l'énergie lumineuse dans la scène, en cours ou à la fin de la simulation. Il était déjà possible de sauvegarder l'état courant de la scène, mais sous une forme brute et binaire, difficile à exploiter. L'absence d'un format de fichier similaire à ceux des données d'entrée s'est fait ressentir lorsque nous avons voulu ajouter à Phostere la possibilité d'interroger interactivement les éclairements de la scène, ou lors des tentatives de connexions avec des outils de réalité virtuelle.

3EI : simulation de réflecteurs

En 1995, la société *3 E Internationale*, fabricant de luminaires, a décidé de produire elle-même les réflecteurs optiques qu'elle avait l'habitude de sous-traiter. Un réflecteur est un objet à la forme complexe, qui doit permettre d'éclairer uniformément une route sans pour autant éblouir les usagers. Pour concevoir un nouveau réflecteur, les fabricants procèdent généralement par essais successifs. A chaque étape, un prototype du réflecteur est réalisé et on mesure l'éclairement qu'il produit. Tant que le résultat n'est pas satisfaisant, on corrige la forme du réflecteur et on produit un nouveau prototype. L'ensemble du processus de conception est coûteux et peut durer jusqu'à dix mois. Les progrès récents en simulation des transferts radiatifs ont amené les responsables à évaluer comment la simulation numérique pouvait réduire leurs coûts et leurs délais de conception. Nous leur avons proposé de remplacer leurs prototypes par un logiciel capable de calculer en temps quasi-instantané l'illumination produite par un réflecteur sur une route.

Le type de simulation à réaliser ressemble plus à ce que fait le programme `ModPhoto` plutôt que `Synthese` : une géométrie minimale mais une simulation physique précise. En outre, la scène à simuler, composée d'un réflecteur maillé très finement et d'une route, est majoritairement spéculaire, ce qui est assez nouveau par mémoire à nos travaux avec *Phostere*. Dans le cas de *Phostere*, les scènes sont majoritairement diffuses, et l'éclairement des primitives est stocké indépendamment de la direction de la lumière incidente. Au contraire, dans le cas du logiciel *3EI*, les carreaux spéculaires ont une influence prédominante sur l'illumination globale, et il est nécessaire de stocker la distribution angulaire des éclairements de ces carreaux.

Une autre nouveauté réside dans la possibilité d'agir sur la géométrie et d'obtenir aussitôt

la nouvelle illumination. Etant donnée la relative simplicité de la géométrie et des calculs de visibilité, la simulation numérique peut être suffisamment rapide pour laisser à l'utilisateur une certaine sensation d'interactivité. La difficulté de cette fonctionnalité n'est pas tant dans l'efficacité de la simulation numérique que dans le développement d'une interface ergonomique pour manipuler la forme du réflecteur.

Pour développer le logiciel *3IE*, l'ingénieur chargé du projet (Hervé Dumortier) a largement réutilisé les catégories `colorimétrie` et `photométrie` de la plate-forme, ainsi que le programme `ModPhoto` avec quelques classes supplémentaires implantées par Dorothée Cazier. Par contre, il a été nécessaire de refaire presque intégralement la simulation des transferts radiatifs. Nous ne disposions pas de primitives qui stocke la distribution directionnelle des éclairements, et l'interface de la classe abstraite `Primitive` ne convenait pas, car elle ne permettait pas la transmission des directions des flux lumineux. Suite à la nécessité de refaire la classe `Primitive`, les programmeurs ont également modifié la classe `Scène`, pour ne lui conserver que la responsabilité de gérer les éléments qui la composent, et confier les fonctionnalités algorithmiques à d'autres classes spécialisées : `Illumination` et `Radiosité` (nous avions justement regretté l'intégration de ces fonctionnalités à la seule classe `Scene` dans *Phostere*).

Le programme *3EI* a été doté d'une interface graphique, permettant notamment de manipuler la forme du réflecteur. Par ailleurs, la configuration matérielle retenue étant une station de travail *Silicon Graphics*, les concepteurs du logiciel ont cherché à tirer le meilleur parti des cartes graphiques intégrées à la station, qui sont optimisées pour le traitement des instructions *OpenGL*. La bibliothèque *OpenGL* a été utilisée pour le rendu final et pour tous les calculs de visibilité.

Pour clore cette courte présentation du programme *3EI*, soulignons que ce programme est susceptible d'alimenter les bibliothèques photométriques de la *plate-forme Graph'IS* ou de *Phostere*. En effet, les résultats de la simulation du réflecteur peuvent être exprimés conformément aux formats `*.source`, `*.optique` et `*.lampe`. Il y a tout de même une perte d'information, car ces formats ont été conçus selon une modélisation ponctuelle des sources. En généralisant les formats aux sources surfaciques, ce qui ne présente a priori pas de difficulté particulière, nous pourrions profiter plus pleinement des résultats de la simulation des réflecteurs. Cette simulation pourrait même s'avérer meilleure que les mesures des laboratoires, interprétées selon le modèle ponctuel.

4 Discussion

Après trois ans de bons et loyaux services, la plate-forme Graph'IS s'est essoufflée. L'accumulation d'optimisations et d'extensions parfois divergentes a rendu le système très complexe, au point d'empêcher toute évolution profonde. Avant de faire le bilan général de cette expérience, nous voulons débattre de certaines questions de fond et tenter d'en dégager quelques enseignements.

À propos du choix des classes

Les manuels d'introduction à l'orientation objets présentent souvent ce paradigme comme étant basé sur les *données*, par opposition à une programmation « classique » qui serait basée sur les *procédures*. Il serait sans doute plus juste de déclarer que la conception objet est basée sur

les *services* et les *comportements*. En effet, l'interface d'une classe correctement encapsulée doit exhiber les services qu'on peut lui demander, et non pas les données qu'elle contient.

Nous sommes nous-mêmes tombés dans le travers consistant à trop se focaliser sur les données, par conséquent sur la structure statique du système logiciel, et à vouloir exagérement nous différencier de la programmation procédurale. Nous avons ainsi négligé l'étude du comportement des objets pendant l'éxécution des programmes, et commis des erreurs sans doute courantes parmi les nouveaux convertis à l'orientation objets :

1. le découpage en classes est trop fondé sur les *données* et pas assez sur les *comportements*,
2. les *entités du problème initial* sont bien représentées, mais peu de classes sont dédiées aux *entités supportant la solution*,
3. les phases de *construction* et de *destruction* des objets ne sont pas suffisamment réfléchies.

La classe `Onde` est une bonne illustration de la première dérive. En se basant sur les structure de données, nous avions clairement identifié le besoin d'une classe qui décrirait la valeur d'une certaine grandeur physique pour un ensemble de longueurs d'ondes prédéterminées. Nous avons donc conçu la classe `Onde`, et utilisé ses instances pour représenter les radiances, les éclairements, les réflectances, et les radiosités. Comme nous avions besoin d'additionner et de multiplier les uns par les autres, nous avons doté la classe de tous les opérateurs linéaires appropriés. Du point de vue des structures de données, le résultat paraissait satisfaisant. Pourtant, si on analyse l'utilisation que l'on peut faire de ces objets, il est évident que toutes les instances ne s'utilisent pas de la même façon, selon les grandeurs qu'elles représentent. Par exemple, on ne multiplie jamais une radiance par une autre radiance. Il aurait donc été souhaitable de créer une hiérarchie de classes, avec des interfaces définissant quel type d'objet peut être additionné et multiplié avec quel autre type d'objet. Avec la seule classe `Onde`, nous avons fourni à l'utilisateur une entité logicielle qui lui permet de gagner du temps pour coder. Avec une hiérarchie de classes, nous aurions en plus capturé une partie des lois optiques, et contraint l'utilisateur à écrire des expressions physiquement valides.

Revenons à présent sur la seconde critique. Il est clair que la nature de notre problème initial, à savoir la *simulation d'un phénomène naturel*, encourage la création de classes correspondant aux entités que nous simulons. L'orientation objets s'y prête particulièrement bien. Il est pourtant tout aussi clair que c'est insuffisant et qu'il faut également concrétiser par des classes le processus de simulation lui-même. Lorsque les algorithmes de simulation sont implantés sous forme de méthodes attachées aux autres classes, ils sont trop peu tangibles. De plus, si le système logiciel est destiné à tester, comparer et évaluer ces algorithmes, il faut identifier les entités algorithmiques standards et en faire des classes abstraites. La première modification à entreprendre dans ce sens est évidemment la création d'une classe abstraite `Simulation`, ce qui déchargerait la trop lourde classe `Scene`.

À propos de la construction des objets

Le troisième défaut énoncé ci-dessus, sur la construction des objets, découle toujours de notre relatif désintérêt quant au comportement des objets pendant l'éxécution du code. Lors de l'expérimentation, nous avons mesuré à quel point la flexibilité apportée par les classes abstraites s'en trouvait amoindrie. La figure 4.20, qui représente de façon très schématique la construction

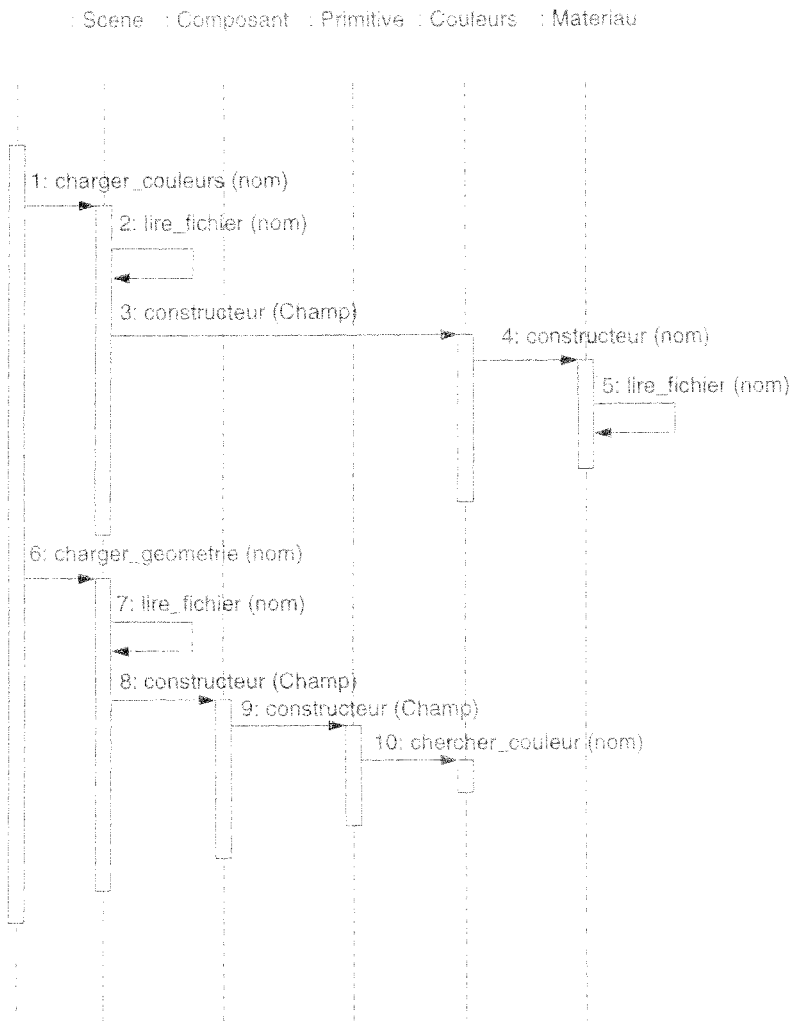


FIG. 4.20 - *Construction d'une scène*

des couleurs, des matériaux et des primitives, va nous servir d'illustration. Il s'agit du processus implémenté dans la bibliothèque de classes. Le principe de base consiste à déléguer l'interprétation des fichiers et de leurs champs aux classes directement concernées : l'instance de `Scene`, à qui sont adressées les requêtes initiales, se charge d'extraire les champs des fichiers, puis construit ses propres éléments en transmettant les données appropriées aux constructeurs appropriés ; ces constructeurs interprètent l'information et appellent à leur tour d'autres constructeurs. Seul le constructeur de `Materiau` ne reçoit pas ses données sous forme d'instance de `Champ`, mais détermine par lui-même le répertoire et le fichier où il trouvera ses caractéristiques.

Nous avons vu que Dorothée Cazier avait développé de nouvelles sous-classes de `Materiau`. Du fait que ces classes héritent de la classe abstraite, leurs instances peuvent être directement associées à des couleurs et exploitées par les primitives d'une scène, sans codage supplémentaire. Encore faut-il que l'on fasse appel à ces nouvelles classes pour construire les instances de `Materiau`. Pour qu'il en soit ainsi, Dorothée a dû créer une sous-classe de `Couleurs` dont le constructeur fasse appel à ses nouveaux matériaux, puis une sous-classe de `Scene` dont la méthode `charger_couleurs` fasse appel à la nouvelle sous-classe de `Couleurs`. Notre choix de déléguer

la construction a donc un effet secondaire très désagréable : lorsque que l'on crée une nouvelle sous-classe de bas niveau, il faut également faire hériter récursivement toutes les classes qui instancient cette première. Pour résoudre ce problème, on peut décider d'exclure toute construction de la bibliothèque, et laisser à la charge des programmes externes la création et l'assemblage initial des objets. Une solution plus élégante consiste à associer à chaque classe abstraite une autre classe abstraite exclusivement chargée de l'instanciation. C'est le concept de *fabrique*, que nous avons déjà évoqué plusieurs fois. Dans notre exemple, si nous faisons l'hypothèse que la classe `Couleurs` passe par une fabrique pour construire ses matériaux, et que cette fabrique peut être changée par l'utilisateur de la bibliothèque, alors il n'est plus nécesssaire de créer des sous-classes de `Couleurs` et de `Scene` pour faire usage des nouveaux matériaux. Il suffit de créer une fabrique appropriée et de la transmettre à l'instance de `Couleurs`.

L'exemple ci-dessus n'est qu'un aperçu des problèmes variés que nous avons pu rencontré lors de l'implémentation de la plate-forme Graph'IS, et qui auraient pu être anticipés en partie. Pour éviter ce travers, il faut probablement aborder la programmation par objets non pas comme une alternative à la programmation procédurale, mais plutôt comme une approche d'un peu plus haut niveau, dont les aspects dynamiques restent de première importance et rejoignent souvent la problématique procédurale.

À propos des programmes

Une des caractéristiques de la plate-forme est sa spécification assez « complète », couvrant à la fois une bibliothèque de classes, des programmes et des fichiers prédéfinis. Nous avons ainsi gagné beaucoup de temps pour l'implémentation initiale et la production des premiers prototypes, en fournissant des directives claires à tous ceux qui ont bien voulu collaborer. Cette approche a été possible parce que notre équipe était de taille modérée, qu'il fallait tout coder depuis zéro, et que nous avions tous un objectif commun et une volonté de travailler en groupe. Passé ce premier stade, chaque chercheur a pu se consacrer a sa problématique personnelle. Alors, ce qui constituait une aide est parfois devenu une gêne.

Dans la section 3, nous avons souvent relevé les limites que nous imposait le découpage prédéfini des programmes. Les voici rappelés, avec quelques cas supplémentaires :

- Les travaux de Christine Chevrier sur la génération de séquences se sont heurtés à la séparation des calculs en trois dimensions, attribués à `Synthese`, et de ceux en deux dimensions, attribués à `Traitement`. Il aurait également été plus adroit de répartir sur deux programmes séparés les activités de simulation et de visualisation, ce qui aurait permis à Christine de cantonner ses interventions au seul programme de visualisation.
- L'implémentation de la *radiosité incrémentale* [Che90] a buté sur la séparation de `ModLum` et de `Synthese`. Pour pouvoir agir sur les sources en cours de simulation, des commandes ont dû être ajoutées à `Synthese`. Elles sont redondantes avec celle de `ModLum`, et de surcroît les modifications ainsi faites ne peuvent pas être sauvegardées.
- La société chargée de développer une interface graphique pour Phostere a demandé une répartition des commandes très différente de la nôtre, basée entre autres sur le caractère « interactif » ou non des commandes.
- Nos expériences de connexion avec des modules de réalité virtuelle auraient été grandement facilitées si nous avions séparé dès le début les activités de « simulation » et de

« visualisation ».

Tous ces blocages plaident pour l'abandon d'une structure prédéfinie de programmes. Nous ne renonçons pas à l'idée de spécifier les tâches d'un système de synthèse d'images, mais il nous faut trouver un autre moyen pour y parvenir, avec une granularité plus fine.

À propos des interfaces utilisateurs

Pour compléter la discussion sur les programmes, il est intéressant de faire état de l'évolution des langages de commandes. Initialement, chaque programme avait ses propres commandes, implémentées sous forme de procédures. Quand ces procédures avaient quelques informations à partager, elles le faisaient à travers de rares variables globales. Lorsque nous avons commencé à bousculer l'organisation prédéfinie, notamment pour Phostere, de nombreuses procédures ont dû être dupliquées ou déplacées. Les variables globales sont devenues plus nombreuses et il devînt difficile de maîtriser leur utilisation. Nous avons alors organisé les commandes les plus courantes en petits groupes autonomes, que nous avons réécrits sous forme de classes statiques. L'étape suivante consista à supprimer le caractère statique de ces classes, ce qui nous a permis de les faire hériter d'une classe abstraite `GroupeCommandes` et de standardiser la manipulation de ces instances par l'interpréteur. De fil en aiguille, sans vraiment le vouloir, nous avons rejoué l'histoire récente du logiciel, qui va de la programmation procédurale à la programmation par objets, en transitant par la programmation modulaire !

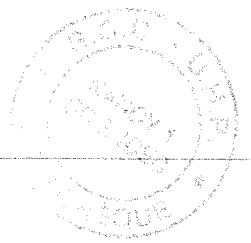
Au bout du compte, l'utilisateur agit toujours sur un programme à travers une collection de commandes, mais il suffirait de quelques aménagements pour disposer en bonus d'une interface de programmation (API), constituée de sous-classes de `GroupeCommandes`. Les classes les plus utilisées pourraient même être intégrées à la bibliothèque. En outre, elles pourraient servir à spécifier les activités principales que nous ne voulons plus représenter par des programmes. L'ancienne collection de programmes serait alors remplacée par une collection d'objets apparents aux utilisateurs, plus nombreux et plus faciles à réarranger selon les besoins particuliers de chaque utilisateur.

À propos des fichiers

Notre découpage en fichiers a été chahuté au même titre que le découpage en programmes. En voici quelques exemples :

- Contrairement à notre attente, les architectes ont souhaité diviser en plusieurs fichiers les descriptions lumineuses, de la même façon que les descriptions géométriques.
- La société chargée de doter Phostere d'une interface graphique a choisi une autre organisation des répertoires Unix que la nôtre, et regroupé dans un même fichier plusieurs matériaux ou plusieurs types de sources.
- Nous avons regretté à diverses occasions l'absence d'un format Graph'IS pour sauvegarder la distribution de la lumière dans une scène après une simulation.

La modification de la structure des fichiers s'est avérée plus délicate que celle des programmes, à cause du code dispersé au sein de la bibliothèque de classes. Nous estimons aujourd'hui que la



structure des fichiers est surtout l'affaire de l'interface utilisateur et ne doit pas interférer avec le noyau de simulation. En conséquence, nous avons enlevé de la bibliothèque toute référence aux fichiers, pour ne plus utiliser que des instances de `Champ`, et la manipulation des fichiers incombe complètement aux programmes.

Tout ce qui constitue le lien avec le système d'exploitation Unix, pour les données comme pour les programmes, se trouve donc finalement relégué à l'extérieur de la bibliothèque de classes, et ne fait plus l'objet d'une découpe prédéfinie. C'est une des principales évolutions de la plate-forme vis-à-vis de l'implémentation initiale.

À propos de la persistance

Notre implémentation de la persistance s'est sensiblement étoffée par mémoire à sa première version. Initialement, nous avions décidé d'assurer la persistance des objets en dotant leurs classes de deux méthodes : l'une pour transformer l'objet en *champ*, une représentation ASCII facile à stocker ; l'autre pour reconstruire ultérieurement l'objet à partir du champ. Nous pensions créer un *format de champ* spécifique pour chaque classe dont les instances sont persistantes, et implémenter la reconstruction des objets de la façon suivante :

1. extraction des champs d'un fichier.
2. pour chaque champ :
   (a) lecture de l'étiquette principale et identification du format.
   (b) appel du reconstructeur de la classe associée au format.

Cette première idée a vite été dépassée, quand l'analyse de notre problème nous a amené à définir des *classes abstraites* dont les sous-classes ont des *instances persistantes*, en particulier les classes représentant les éléments de la scène simulée : `Primitive`, `Source`, `Materiau`. Le cas de `Primitive` n'est pas vraiment problématique : chaque sous-classe (`Facette`, `Sphere`,...) possède sa propre structure de donnée et peut être associé à un format de champ spécifique. Deux instances de classes différentes incarnent à priori deux éléments différents de la scène d'origine. Le cas de `Materiau` est plus délicat, car le même matériau d'origine peut donner lieu à des instances de classes différentes selon le modèle retenu par l'utilisateur. Les différentes classes ont des données en commun, et nous voulions décrire chaque matériau de la bibliothèque photométrique par un seul et même champ, regroupant toutes les informations disponibles sur ce matériau pour l'ensemble des modèles. Nous avons donc spécifié un format commun qui n'était attaché à aucune classe en particulier. Le procédé de reconstruction a alors été enrichi d'une étape de sélection du modèle, et chaque reconstructeur a été implémenté de façon à ignorer les informations qui ne les concernaient pas. Grâce à cette dernière faculté des reconstructeurs, on peut créer de nouvelles sous-classes de `Materiau` et enrichir les bibliothèques photométriques avec de nouvelles données, sans perturber les sous-classes existantes.

On peut déjà ressentir que la forme persistante des matériaux a acquis une certaine autonomie vis-à-vis des classes. C'est encore plus vrai dans le cas des vues. Lors de l'exposé des expérimentations, nous avons constaté qu'une même vue était une instance d'une classe différente selon le programme dans lequel nous nous trouvions : `VueAn` au sein du programme `Analyse`, `VueTt` dans le programme `Traitement` ou `VueDyn` dans `Sequence`. Comme pour les matériaux, une même

vue peut être représentée par des instances de classes différentes, mais avec une difficulté supplémentaire : les classes ne permettent pas seulement de consulter les vues, mais également de les modifier. De ce fait, la faculté d'ignorer les données inconnues n'est plus suffisante. Pour préserver l'intégrité des vues, nous avons modifié les reconstructeurs afin qu'ils stockent les informations qu'ils ne reconnaissent pas, et nous avons fait en sorte de restituer ces informations lorsqu'une instance est retransformée en champ.

En fin de compte, nous aboutissons à un système où certains objets ont une forme persistante plus ou moins autonome, conforme à un format de champ donné, dont la mise à jour est assurée collectivement par un ensemble de classes concrètes. Ce système, que nous avons adopté spontanément, permet aux utilisateurs d'étendre les structures de données en toute liberté. On peut craindre que cette liberté excessive favorise une divergence des utilisateurs et une baisse de la réutilisation des données, puis du code. L'autre solution consisterait à recréer une classe pour chaque format de champ, par exemple `MateriauData` et `VueData`, mais il deviendrait alors très difficile de changer ou d'étendre la structure de données. L'utilisation d'un composant commercial pour la persistance ou d'un système de gestion de bases de données orienté objets s'apparenteraient plutôt à la seconde option. Ces outils commerciaux, rares au début de nos recherches, deviennent aujourd'hui plus nombreux et plus matures. Il serait souhaitable de les mettre à profit, si on peut faire en sorte de ne pas trop figer les structures de données.

Si nous devions continuer à améliorer la classe `Champ` et la catégorie `persistance` dans son ensemble, quelques modifications majeures s'imposeraient :

- généralisation du mécanisme d'inclusion de fichiers,
- généralisation de l'usage de valeurs par défaut, et de la possibilité de les redéfinir dans les niveaux supérieurs d'un arbre de champs,
- création d'une forme binaire pour les champs, équivalente à la forme ASCII, avec la possibilité de passer librement de l'une à l'autre selon le besoin.

À propos de l'implémentation en C++

Le caractère mixte du langage n'a pas posé autant de problèmes que ce que l'on pouvait craindre. Un peu de pédagogie et la démonstration par l'exemple ont suffi à convaincre les différents contributeurs de se défier des variables globales, des procédures libres, et de toutes les pratiques venant de la programmation en C.

Le seul obstacle prévisible que nous n'avons pas réussi à contourner est celui de la gestion dynamique de la mémoire. En effet, pour tirer parti du polymorphisme, il est indispensable d'utiliser des pointeurs. De ce fait, la création et la destruction des objets est à la charge du programmeur, et comme dans tout développement en C++, nous avons affronté les multiples erreurs d'éxécution liées à une mauvaise gestion de la mémoire. Des outils commerciaux performants existent aujourd'hui pour instrumenter et analyser un programme pendant son éxécution. Ils permettent d'éliminer la plupart des erreurs. Une autre solution, que nous n'avons pas expérimentée, consisterait à utiliser un composant logiciel commercial de gestion de mémoire (*garbage collector*).

Au titre des problèmes imprévus, quelques-uns sont particulièrement retors et nous ont fait

perdre beaucoup de temps. Ils sont apparus le plus souvent au sein de nos classes mathématiques :

- La déclaration d'un nombre excessif de constructeurs et d'opérateurs de conversion peut amener le compilateur à faire des conversions automatiques absolument insoupçonnables, et bien sûr absolument fausses.
- Les instances constantes figurant dans les déclarations de classes (vecteurs unitaires, matrice identité) sont initialisées avant l'éxécution de la procédure `main()`, dans un ordre quasi-aléatoire, ce qui conduit à des valeurs fausses lorsqu'une constante se construit à partir d'une autre qui n'est pas encore initialisée.

Par ailleurs, nous avons évidemment essayé les classes génériques (*templates*), qui peuvent simplifier considérablement l'écriture d'un programme. Malheureusement, la compilation devient beaucoup plus compliquée, et chaque compilateur applique une stratégie personnelle pour instancier ces classes. Devant la complexité et la diversité des implémentations, nous avons préféré temporiser. L'effet ne s'est pas fait attendre : en l'absence de recommandations sur l'implémentation des collections, chacun des programmeurs a utilisé ses propres structures pour gérer les listes. Il y a autant d'implémentation de listes qu'il y a eu de programmeurs sur la plate-forme, et cela complique singulièrement la maintenance. Pendant ces dernières années, nous avons vu émerger la bibliothèque STL, mais la situation ne semble pas évoluer en ce qui concerne la compilation. Aujourd'hui, nous sommes d'avis qu'il vaut mieux utiliser les classes génériques et affronter les problèmes de compilation associés, plutôt que de laisser se développer de multiples variantes de collections.

La même remarque vaut pour le mécanisme de gestion des exceptions. A présent que ce mécanisme est fourni par les compilateurs, son usage s'impose, surtout dans la perspective d'un transfert industriel. Les pratiques consistant à saupoudrer le code d'appels à `assert()`, `abort()`, où à faire des affichages sauvages sur le terminal sont autant de temps perdu quand on veut ajouter une interface utilisateur plus conviviale.

À propos des statistiques sur l'éxécution des algorithmes

Étant donné le rôle de banc d'essai que doit assurer la plate-forme Graph'IS, nous devions faciliter la prise d'information lors de l'éxécution des programmes. Pour y parvenir, nous avons créer quelques classes, notamment des compteurs et un chronomètre, et ajouter des appels à ces classes dans les méthodes supportant les algorithmes.

Ces appels sont particulièrement pénalisant pour les performances, et nous ne pouvions pas les conserver pour l'utilisation de la plate-forme dans de vrais projets. Au lieu de créer plusieurs versions du code, nous avons préféré nous appuyer sur le préprocesseur C++ et placer ces instructions dans des zones conditionnées au mode de compilation. Ce système n'est pas totalement satisfaisant, car la clarté du code est brouillée par toutes ces instructions additionnelles.

À propos de l'optimisation des performances

La plate-forme Graph'IS a été trop optimisée. Dans l'optique de produire des prototypes industriels, nous avons cédé à la tentation d'optimiser nos algorithmes de façon un peu incontrôlée. L'effet était prévisible : en spécialisant les algorithmes et les structures, nous avons réduit la

lisibilité du code et sa flexibilité. Cette erreur tient pour une bonne part dans la lourdeur actuelle atteinte par la plate-forme.

Nous n'avons pas trouvé de moyen idéal pour bien localiser et signaler les optimisations dans un logiciel. Nous ne pouvons pas non plus nous résoudre à bannir toute optimisation, car c'est un de nos prérequis que de pouvoir traiter des projets de grande taille. Pour un nouveau système logiciel, nous pourions seulement préconiser un usage modéré et contrôlé des optimisations. La meilleure façon consiste à toujours utiliser par défaut l'implémentation la plus simple et la plus évolutive (donc la plus lente), puis à utiliser un outil commercial d'analyse de performance pour identifier les goulots. Il faudrait que les programmeurs renoncent aux optimisations qu'ils appliquent à l'aveuglette. Avec les progrès récent des compilateurs, ces pratiques sont au mieux obsolètes, au pire préjudiciables pour le bon fonctionnement des méthodes d'optimisation automatique du compilateur.

5 Conclusion

Dans ce chapitre, nous avons exposé les caractéristiques principales de la plate-forme Graph'IS : un ensemble de classes abstraites tirées de l'équation de transfert, des représentations ASCII pour les objets persistants, un système de fichiers et de programmes adaptés aux étapes d'une simulation. À travers différents travaux de recherche et de transfert industriel, nous avons montré comment la plate-forme Graph'IS a été exploitée avec succès.

Nous avons aussi mis en évidence certains défauts de conception et d'implémentation, en particulier la trop faible attention accordée aux algorithmes de propagation. Les écueils les plus visibles auraient pu être évités en suivant les règles suivantes :

- calquer les classes sur les comportements standards plutôt que sur les structures de données,
- créer des classes abstraites pour les algorithmes,

  concevoir le comportement dynamique du système, en particulier le processus de construction des objets,
- adjoindre des fabriques abstraites aux classes abstraites,
- être attentif à l'évolutivité des données, au même titre que celle du code,
- ne pas prédéfinir les fichiers et les programmes,
- doter le noyau de simulation d'une interface de programmation,
- utiliser les classes génériques pour implémenter les collections d'objets,
- utiliser le mécanisme de gestion des exceptions.

En 1995, nos réflexions ont été relancées par la publication de Philipp Slusallek [Slu95] sur l'architecture VISION, qui se donnait des objectifs parfois proches des nôtres. Par ailleurs, après le renouvellement d'une grande partie des chercheurs de notre équipe, les centres d'intérêt scientifiques se sont sensiblement déplacés. Nous avons saisi ces prétextes pour réfléchir à une nouvelle architecture, qui corrige les erreurs identifiées dans la plate-forme Graph'IS et qui couvre un

champ scientifique un peu plus large. Nous livrons le fruit de cette réflexion dans le chapitre 5, sous forme de propositions.

Chapitre 5

Quelques propositions

Les précédents chapitres nous ont permis de présenter l'état de l'art en ce qui concerne les plates-formes logicielles pour la synthèse d'images, ainsi que notre tentative en la matière, la plate-forme Graph'IS. Fort de cette expérience, nous pouvons faire quelques nouvelles propositions, plus en accord avec les dernières évolutions des domaines de la synthèse d'images et de l'orientation objets. Certaines de ces propositions sont actuellement expérimentées dans notre équipe de recherche.

1 Description algorithmique

De même que l'équation de transfert nous a servi de support pour identifier les classes clés de Graph'IS, nous pourrions exploiter la présentation de la section 3 pour identifier les éléments algorithmiques du processus de simulation, et leur associer des classes abstraites. La figure 5.1 en est un exemple simple.

Par ailleurs, si de nombreux algorithmes sont possibles pour une tâche donnée, il faut que l'utilisateur dispose d'un moyen simple pour énoncer ses choix. Cette configuration pourrait être réalisée à travers l'interface utilisateur du programme de simulation, mais cela implique que l'ajout d'un nouvel algorithme impose une mise à jour de l'interface du programme. Une solution sans doute plus évolutive et plus souple serait de considérer le choix des algorithmes comme une donnée d'entrée à part entière. Si l'on reprend la terminologie utilisée dans la plate-forme Graph'IS, ces informations pourraient être qualifiées de *description algorithmique* et faire l'objet d'un format de fichier.

2 Quatre composants distribués

Les architectures que nous avons présentées précédemment ont toutes le même objectif applicatif : produire une image la plus conforme possible à la réalité. Les phases de modélisation et de rendu sont complètement séparées, et les données ne transitent que dans un sens : du modeleur vers le programme de rendu. Cette approche ne nous semble pas en accord avec les défis actuels. Parmi les différents obstacles que nous avons rencontrés avec le prototype logiciel *Phostere*, les

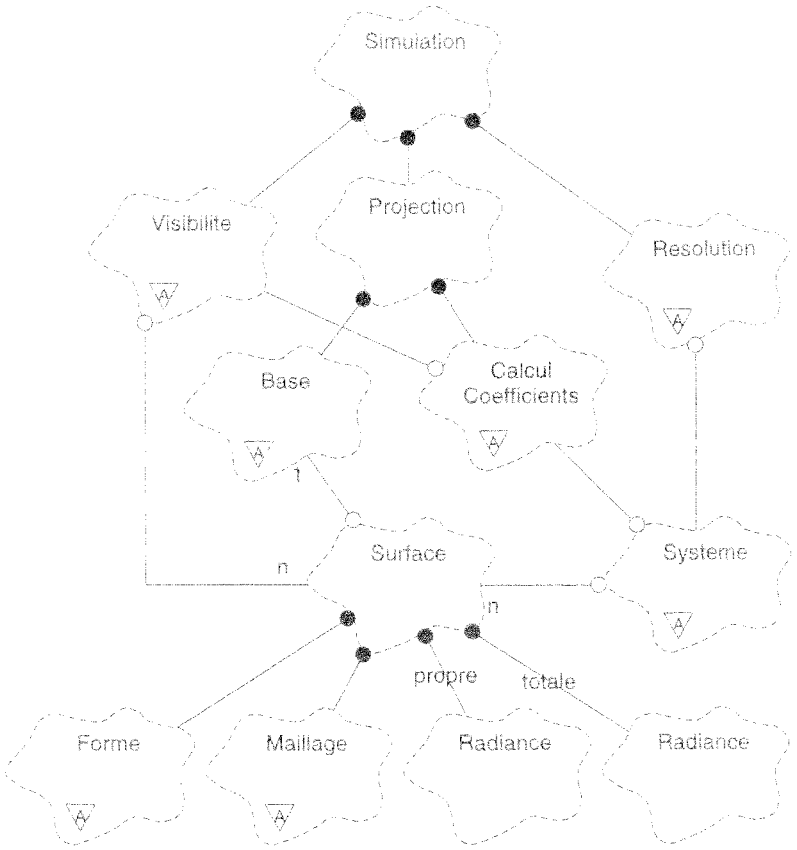


FIG. 5.1 – *Exemple de classes pour la simulation*

quatre suivants sont révélateurs :

- Pour implémenter l'algorithme de radiosité incrémentale, capable de mettre à jour l'illumination lorsque l'utilisateur agit sur les sources lumineuses, nous avons dû réécrire pour le programme `Synthese` un grand nombre des commandes de modélisation de `ModLum`.
- Quand certains collaborateurs d'EDf ont souhaité réutilisé les résultats de Phostere dans leurs outils de réalité virtuelle, nous avons dû « bricoler » une commande pour sauvegarder une description 3D des radiances d'une scène.

  Lorsque les éclairagistes ont souhaité avoir les valeurs numériques des radiances de la scène, nous n'avons pu leur proposer que des images en fausses couleurs.
- Lorsque nous avons déporté sur une machine distante le calcul de l'illumination globale, nous avons également dû déporter le calcul des images, trop imbriqué avec le premier.

Afin de mieux séparer les calculs dépendants et indépendants du point de vue, et afin de pouvoir intégrer l'illumination globale dans un environnement de modélisation ou de navigation, nous proposons d'adopter une structure globale composée de quatre composants (figure 5.2) :

- *La base de données*: description de la scène, contenant les informations nécessaires aux trois autres blocs, et associée à un format 3D.

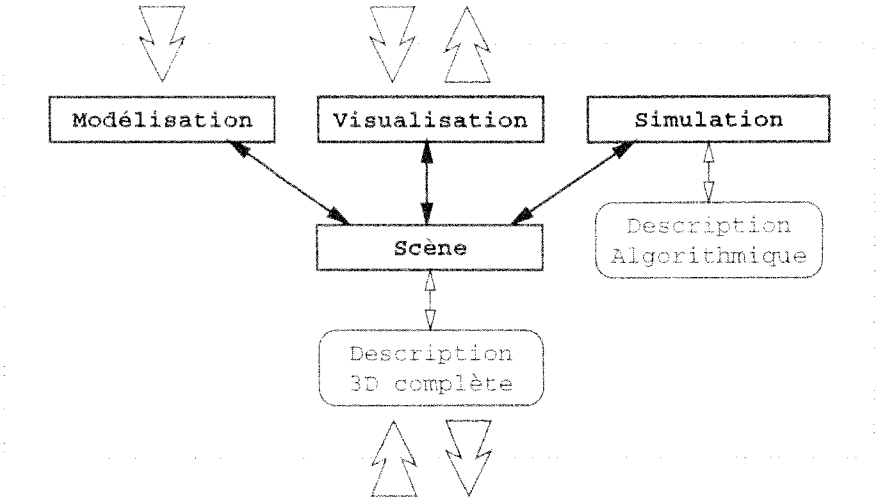


FIG. 5.2 – *Nouveau découpage en composants*

- *La modélisation* : agit sur les primitives et les sources de la scène.
- *La simulation* : diffuse la lumière à travers la scène, c'est-à-dire calcule l'illumination globale, indépendamment du point de vue.
- *La visualisation* : produit des images, en tirant parti des calculs de simulation.

Il y a ici plusieurs différences avec ce qui avait été spécifié dans la plate-forme Graph'IS. D'abord, nous proposons une nouvelle répartition des fonctionnalités. Ensuite, la même structure de données en 3D sert à la fois d'entrée et de sortie aux algorithmes de simulation. Enfin, nous ne parlons plus de «programmes» mais de «composants». Ce dernier terme est utilisé aujourd'hui pour désigner des morceaux de code éxécutable que l'on peut recomposer librement au moment de l'éxécution. Nous avions évoqué la difficulté de marier la philosophie d'Unix et celle de l'orientation objets. Au lieu de raisonner en termes de programmes échangeant des fichiers, il nous semble à présent plus approprié d'utiliser le formalisme des composants. Par ailleurs, l'idée de composant logiciel va de pair avec celle de calcul distribué. L'utilisation d'un environnement conforme au standard *Corba* permettrait d'éxécuter le composant de simulation sur une machine parallèle distante, alors que les composants de modélisation et de visualisation pourraient s'éxécuter sur le poste de l'utilisateur.

3 Protocoles

Lors de l'implémentation de certains algorithmes à base de *shooting*, il est apparu que l'émetteur défini à chaque itération était toujours utilisé de la même façon, qu'il s'agisse d'une source ou d'une surface. Pour traduire le fait que la surface se comporte comme une *source secondaire*, nous avons envisagé de créer un lien d'héritage entre les deux classes, mais cette solution ne nous satisfaisait pas, dans la mesure où la surface se conduit comme une source *seulement lors de son utilisation en tant qu'émetteur*. Cet exemple illustre notre besoin d'un mécanisme pour exprimer les *rôles standards* d'un système.

Définition

On peut exprimer un *rôle standard* à l'aide d'une classe abstraite, de même qu'on utilisait déjà ce type de classe pour exprimer les *entités standards* d'un système. Cependant, les classes représentant des rôles sont encore plus incomplètes que leurs consœurs, car elles n'expriment qu'une facette des objets. Pour différencier ces deux utilisations des classes abstraites, nous appelerons *interfaces* celle qui représentent les entités du systèmes, et *protocoles* celles qui représentent les rôles de ces entités.

L'utilisation de protocoles va de pair avec l'héritage multiple. De même qu'un objet peut jouer plusieurs rôles dans un système, une classe abstraite peut être constituée par héritage de plusieurs protocoles. En réexaminant tous les endroits où nous avons envisagé d'utiliser l'héritage multiple dans la plate-forme Graph'IS, nous avons constaté que la plupart servaient justement à exprimer ce genre de relations.

Grady Booch a également identifié le besoin d'avoir des classes décrivant les *rôles*. Parce qu'elles sont destinées à l'héritage multiple, il les qualifie de *fusionnantes*. De son coté, le nouveau langage de programmation *Java* [AG96] abonde dans le même sens, en introduisant une sorte de classe semblable aux protocoles, et en restreignant l'héritage multiple à cette seule sorte de classe.

Exemple

L'utilisation de protocoles est illustrée dans la figure 5.3, qui reprend l'exemple des primitives et des sources lumineuses. Dans cette figure, nous introduisons trois protocoles :

- `Emetteur` : capacité de produire un éclairement en tout point de la scène. Sources et primitives peuvent jouer ce rôle.
- `Recepteur` : capacité de stocker un éclairement reçu. Seules les primitives peuvent être subdivisées en éléments et enregistrer l'énergie qu'on leur transmet.
- `Visible` : capacité d'intercepter les rayons lumineux. La luminance d'une source est dûe à son énergie propre. Celle d'une primitive résulte de l'énergie qu'elle a reçue en tant que récepteur.

La figure met également en évidence un autre aspect très important : lorsque des protocoles sont disponibles, les clients d'une classe utilisent ces protocoles comme types plutôt que la classe elle-même. Ce procédé réduit le couplage entre classes et favorise donc la réutilisation.

Optimisation

Lorsque l'on veut définir une classe qui implémente un protocole donné, on l'exprime normalement par une relation d'héritage, auquel cas la classe fournit elle-même l'implémentation des méthodes définies par le protocole. Une autre solution consiste à interposer une classe qui sert d'intermédaire. En C++, les opérateurs de conversion automatique permettent de réaliser cette interposition sans qu'il soit nécessaire de modifier le code des clients. Le procédé est illustré dans la figure 5.4.

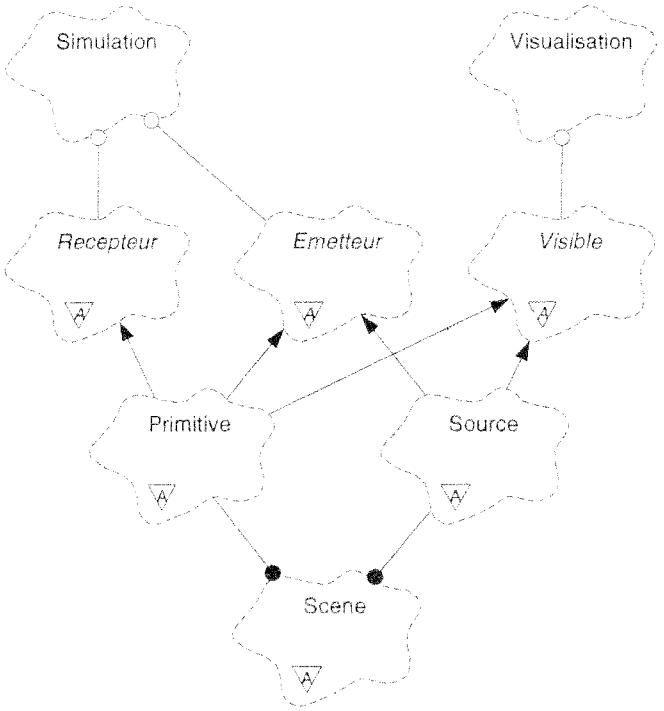


FIG. 5.3 – *Utilisation de protocoles*

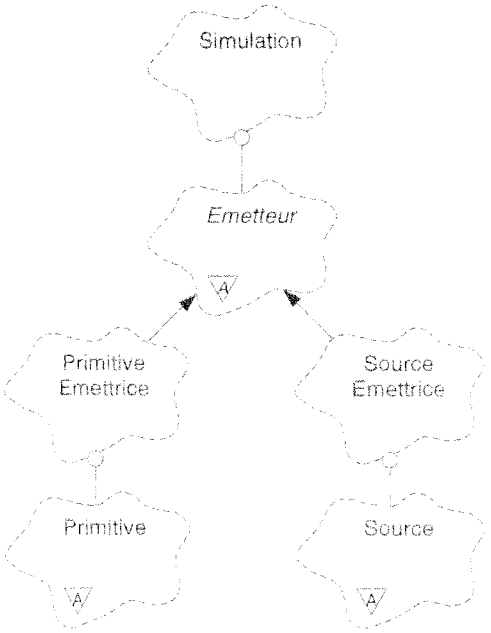


FIG. 5.4 – *Optimisation des protocoles*

Le principal intérêt d'un objet intermédiaire, c'est que ce dernier peut être le siège de précalculs adaptés aux méthodes du protocole. Il s'agit essentiellement d'une technique d'optimisation. Ainsi, lorsque l'objet de type `Simulation` veut utiliser une certaine primitive comme émetteur, l'objet de type `PrimitiveEmettrice` qui est créé peut précalculer les luminances de la primitive correspondante.

L'idée des classes intermédiaires est à rapprocher de celle des *itérateurs* de la STL (*Standard Template Library*), ou des *smart objects* de VISION.

4 Implémentation basée sur Open Inventor

Open Inventor [Wer94a, Wer94b, B+94] est une boîte à outils pour la manipulation de scènes en trois dimensions. Derrière le terme « boîte à outils », il faut comprendre une architecture générique, une large collection de classes concrètes prêtes à l'emploi, et des utilitaires logiciels. Par « manipulation de scènes en trois dimensions », on entend la possibilité de naviguer en temps réel dans une scène et de pouvoir agir sur ses éléments.

Afin d'illustrer comment cette boîte à outils facilite le développement d'applications de *modélisation* ou de *visualisation en temps réel*, nous allons présenter brièvement ses principaux constituants :

- les *nœuds* permettant de décrire une scène,
- les *kits* qui aident à la construire,
- les *manipulateurs* grâce auxquels l'utilisateur interagit,
- le *format de fichier*,
- les *composants* de l'interface graphique.

Grâce à son architecture générique, Open Inventor est également très extensible. On peut raisonnablement envisager de lui ajouter un module de simulation des transferts radiatifs ou un module de visualisation plus précise.

Nœuds

Une scène est modélisée par un ou plusieurs graphes acycliques, dont les éléments sont appelés des *nœuds*. Les principales classes représentant les nœuds appartiennent aux catégories suivantes :

- *Formes* : différentes primitives géométriques,
- *Propriétés* : modèles de réflexion et d'éclairage, textures,
- *Groupes* : organisation du graphe,
- *Moteurs* : animations des nœuds,
- *Détecteurs* : réactions automatiques à certains évènements.

Il est possible d'appliquer des *actions* à tous les nœuds d'une scène. Le parcours des graphes est pris en charge par le système. Plusieurs actions sont fournies en standard, comme le calcul d'une boîte englobante, la visualisation d'une scène, ou sa sauvegarde. Le programmeur peut définir par héritage de nouveaux nœuds et de nouvelles actions.

Kits

Les kits de construction sont des assemblages de nœuds dont la structure est prédéfinie. Ils facilitent la construction des graphes en fournissant des structures et des valeurs par défaut que l'utilisateur n'a plus qu'à compléter. Le programmeur peut définir ses propres kits ou mettre en œuvre ceux qui sont fournis.

Manipulateurs

Un *manipulateur* est un type d'objet particulier qui réagit aux événements de l'interface graphique et sur lequel l'utilisateur peut directement agir. Il possède généralement une représentation dans la scène, ce qui fournit un moyen de traduire les événements en modifications de cette scène.

Par exemple, la *boîte de maniement* est une boîte englobante que l'on peut manipuler par ses coins et ses arêtes. Lorsque l'utilisateur agit sur cette boîte, les modifications sont appliquées à la fois à la boîte et à l'objet englobé.

Les manipulateurs fournissent un moyen simple pour incorporer des interactions graphiques directes dans une application.

Format de fichiers

Tous les éléments d'une scène, y compris les manipulateurs et les senseurs, peuvent être intégralement sauvegardés dans un fichier. Ce fichier peut être binaire ou ASCII, selon que l'on souhaite réduire son encombrement ou pouvoir l'éditer facilement.

La structure d'un fichier reflète fidèlement le graphe de nœuds d'origine, et contient toutes les informations permettant de reconstruire ultérieurement la même scène. On peut considérer le mécanisme de sauvegarde d'Inventor comme une forme légère de persistance.

Les attributs dont la valeur est celle par défaut ne sont pas sauvegardés, ce qui réduit la taille du fichier et accroît sa lisibilité. La gestion des nœuds nouveaux ou inconnus a également été envisagée, ce qui facilite l'évolution des fichiers et du format.

Composants d'interface

Des classes sont proposées pour faciliter le dialogue avec le système de fenêtrage, l'intégration sous X11 et le développement d'une interface utilisateur graphique. Les principaux éléments fournis sont les suivants :

- une zone de rendu 3D,

- une zone de visualisation et d'édition des nœuds,
- un traducteur d'évènements X11 en événements Inventor (transmis aux manipulateurs),

  des procédures d'initialisation et de lancement de la boucle principale du système de gestion des fenêtres.

Les composants comprennent toujours une zone de rendu et une interface utilisateur. Ils sont utilisés pour éditer individuellement les nœuds du graphe (matériaux, sources lumineuses, transformations) ou pour visualiser la scène selon différents points de vues et différentes métaphores de navigation.

Adéquation à notre problématique

Comme le laisse deviner le tableau que nous venons de brosser, Open Inventor est un excellent candidat pour implémenter le processus `Scene` que nous avons proposé dans la première section et assurer la persistance des objets. De plus, cette boîte à outils possède de nombreux atouts qui coïncident avec les améliorations que nous souhaitions apporter à la plate-forme Graph'IS :

- un mécanisme de gestion de la mémoire se charge de récupérer les objets inutilisés (*garbage collecting*),
- une fabrique abstraite centralisée prend en compte les nouvelles classes définies par le programmeur,
- les kits facilitent la construction initiale et l'assemblage des objets,
- les fichiers d'entrées-sorties peuvent basculer d'une forme ASCII lisible à une forme binaire plus compacte.

L'adéquation d'Open Inventor aux divers problèmes que nous avons évoqués dans les chapitres précédents est due à la convergence naturelle des besoins des utilisateurs et de l'offre industrielle. Si le domaine d'application ciblé n'est pas exactement le nôtre, l'extensibilité de l'architecture nous donne les moyens de nous approprier ce produit. Par ailleurs, l'avenir d'Open Inventor est bien plus assuré que celui de n'importe quel système strictement dédié à la synthèse d'image. Si ce n'est déjà fait, on peut raisonnablement espérer qu'Open Inventor ou un produit directement dérivé s'impose comme standard, au même titre qu'Open GL et VRML, promus par le même constructeur informatique.

Conclusion

La perspective d'une longue collaboration entre notre équipe de recherche et Électricité de France, sur le thème de la synthèse d'image appliquée à l'ingénierie de l'éclairage, a amené les deux parties à mettre en place une convention CIFRE spécialement dédiée aux aspects logiciels. Ce travail m'a été confié, et j'ai conçu une architecture qui devait satisfaire à la fois les chercheurs de l'équipe et les ingénieurs chargés de produire des prototypes industriels. L'implémentation de cette architecture a été collective et a donné lieu à plusieurs systèmes logiciels, avec de très nombreuses variantes. La préparation de ce mémoire a surtout consisté à analyser les multiples expérimentations et à en extraire les traits communs. Nous avons parfois abouti à des conclusions que nous avions déjà lus par le passé sans les comprendre vraiment. Il est à craindre que toute équipe qui se lance dans le développement d'un système à objets dédié à son domaine soit obligée de passer par les mêmes erreurs. J'espère tout de même que ce mémoire permettra à ses lecteurs de prendre quelques raccourcis. La récapitulation ci-dessous reprend l'essentiel de chaque chapitre.

Simulation des transferts radiatifs

En exprimant les échanges lumineux à l'aide de son *équation de transfert*, Kajiya a contribué à redonner un fondement physique à la synthèse d'images. En plus de la géométrie fournie par les modeleurs de CAO, nous nous sommes appliqués à utiliser les *mesures photométriques* que peuvent établir certains laboratoires spécialisés :

- distribution spatiale des luminaires,
- distribution spectrale des lampes,
- facteur de fresnel des matériaux.

Notre équipe s'intéressait en priorité aux méthodes de simulation qui produisent des résultats indépendant du point de vue, en particulier aux méthodes de radiosité, que nous avons replacées dans le cadre plus formel des *éléments finis*. Pour traiter l'équation de transfert selon cette approche, on la transforme en système linéaire par le biais d'un opérateur de projection. Chaque méthode de simulation est caractérisée par les choix suivants :

- mode de discrétisation et base de fonctions du domaine spectral,
- mode de discrétisation des surfaces et des directions, et base de fonctions pour exprimer la radiance,

- base de fonctionnelles linéaires,
- mode de calcul des cœfficients du système linéaire,
- mode de résolution du système linéaire.

Les approximations faites à ces différents niveaux introduisent des erreurs dans la simulation. La maîtrise de l'erreur globale est particulièrement difficile. Lorsque le système linéaire est résolu, il reste encore à produire des images, par une projection ou une variante de lancer de rayons.

Architectures logicielles génériques

Le système logiciel que nous voulions développer se devait en priorité d'être évolutif et adaptable. Ce n'est pas le cas des bibliothèques de modules, que les utilisateurs ne peuvent modifier qu'en dupliquant les modules.

Devant les lacunes de l'approche procédurale, nous avons pris le parti d'utiliser la programmation par objets. Le mécanisme d'héritage fournit l'adaptabilité dont manque les bibliothèques de modules. Par ailleurs, si une bibliothèque de classes fait usage du polymorphisme et de la liaison dynamique, il est également possible de substituer de nouveaux types de paramètres à ceux qui sont utilisés par défaut.

Ce n'est toutefois pas suffisant : il est parfois nécessaire de redéfinir les structures de données internes des classes, ce que ne permet pas l'héritage. Pour éviter de figer les structures de données, il faut définir des superclasses sans attributs, mais il est alors impossible de fournir l'implémentation des méthodes. Dans cette optique, les langages de classes permettent de spécifier une méthode sans en fournir l'implémentation. Les classes qui possèdent de telles méthodes sont dites *abstraites*. Dans le cas extrême, une classe abstraite se réduit à une liste de méthodes non implémentées, et sert à exprimer une *interface standard*. Les classes abstraites sont incomplètes et ne peuvent être instanciées. Pour pouvoir construire des objets, il faut d'abord créer une sous-classe qui ajoute les implémentations manquantes.

Une bibliothèque riche en classes abstraites est particulièrement adaptable et s'utilise d'une manière spécifique (la phase de construction et d'assemblage des objets devient essentielle). Pour traduire cette spécificité, on ne parle plus de bibliothèque mais d'*architecture générique* (*framework*). C'est ce type de système logiciel que nous avons cherché à concevoir pour la simulation des transferts radiatifs.

Systèmes de synthèse d'images

Nous avons évoqué l'évolution des systèmes logiciels de synthèse d'images, passant progressivement du *modèle local* au *modèle global*, puis à la *simulation des phénomènes physiques*. Nous avons également évoqué la transition des bibliothèques de modules aux bibliothèques de classes, et leur souci commun de bien s'interfacer avec les outils commerciaux de modélisation géométrique.

Le banc d'essai de Cornell est un exemple typique de bibliothèque de modules pour le modèle global. Il récupère en entrée les données des modeleurs géométriques, et produit des images en sortie. Le format neutre MID permet de mieux articuler la connexion entre modélisation et

rendu. Le banc d'essai inclut les trois algorithmes de sa génération : un lancer de rayons, une méthode de radiosité, et un algorithme à deux passes. Ces programmes sont implantés à l'aide d'une bibliothèque de modules, organisés en trois couches :

- *Utilitaires* : dialogue avec le système d'exploitation, manipulation des structures de données.
- *Primitives* : fonctions à appliquer aux primitives de la scène.
- *Rendu* : sous-tâches des programmes de synthèse d'images.

La conception et l'implémentation en C sont foncièrement procédurales. Malgré l'abondance des modules et la finesse de la décomposition, celle-ci est trop liée aux algorithmes déjà implantés. Elle s'est mal accordée aux méthodes qui ont été proposées par la suite. De plus, le banc d'essai souffrait du manque d'adaptabilité intrinsèque à un programme C.

Le système VISION, au contraire, s'appuie sur une architecture à objets et une conception fondée sur les grandeurs physiques du problème. Philipp Slusallek se démarque tout de même d'une mise en œuvre basique du paradigme objet, qui voudrait que l'on délègue toutes les tâches aux classes représentant les éléments de la scène simulée, et que toutes les données associées à un élément de la scène soient réunies sous un seul et même objet. Au lieu de cela, l'auteur préconise la création d'une classe chargée des tâches impliquant une connaissance globale de la scène, et la séparation des données géométriques et des radiances calculées. On obtient ainsi une architecture à quatre classes principales :

- `Geometry` : description géométrique de la scène,
- `Shader` : réflectance locale d'une surface,
- `LightSource` : distribution lumineuse locale d'une source,
- `Lighting` : pilotage de la simulation globale et stockage des radiances calculées.

Un large éventail d'algorithmes a été implémenté dans VISION, aussi bien dans la branche des méthodes stochastiques que dans celle des méthodes par éléments finis. Les choix d'implémentation sont également intéressants : l'extension de RENDERMAN aux besoins de l'illumination globale plutôt que la création d'un nouveau format, le choix de C++, l'extension de TCL pour l'interface utilisateur.

Pour terminer le tour d'horizon des systèmes existants, nous avons dit quelques mots du projet IUE, intéressant à plus d'un titre. Ce projet vise à construire un environnement à objets dédié au domaine de la vision artificielle et dispose de moyens importants pour y parvenir. Nous en retiendrons les points suivants :

- l'utilisation d'un langage de spécification doublé d'un générateur de code,
- la représentation des algorithmes par des classes de type « tâche »,
- le besoin d'un mécanisme de persistance élaboré,
- le délaissement de CLOS au profit de C++.

Architecture proposée

Dans le chapitre 4, nous avons proposé notre propre système logiciel, constitué d'une *bibliothèque de classes* en majorité abstraites, ainsi que d'un ensemble de *formats de fichiers* et de *programmes prédéfinis*.

Les classes principales de la bibliothèque reflètent directement les termes de l'équation de radiance, à l'exception des classes de visualisation et de la classe `Scene` qui pilote l'ensemble du système. Elles sont organisées en *catégories* :

- `colorimetrie` : `Spectre`, `Onde`, `Xyz`, `Ecran`, `RvbReel`, `RvbEntier`,
- `photometrie` : `TypeSource`, `Lampe`, `Solide`, `Materiau`, `RepSpec`, `RepSpat`,
- `scene` : `Source`, `Primitive`, `Forme`, `Carreaux`, `Scene`, `Espace`, `Camera`,
- `images` : `Image`, `Niveau`, `Masque`, `Filtre`, `Vue`.

Parmi ces classes, certaines appartiennent aux entrées/sorties du système et ont donc été qualifiées de *persistantes*. Nous avons fourni un certain nombre d'utilitaires permettant de construire une représentation ASCII lisible des objets persistants, et regroupé ces utilitaires dans la catégorie `persistance`. Certains mécanismes facilitent la réutilisation des vieilles données quand la structure des classes évoluent.

Les formats de fichiers et les programmes prédéfinis sont inspirés de nos collaborations avec les architectes et les éclairagistes. Nous avons défini les formats suivants, organisés en quatre sous-ensembles :

- *Bibliothèque Photométrique* : `*.materiau`, `*.source`, `*.optique`, `*.lampe`,
- *Description Géométrique* : `*.geo`,
- *Description Lumineuse* : `*.lumiere`, `*.cameras`, `*.couleurs`,
- *Vues* : `*.vue`, `*.ppm`.

Les programmes prédéfinis vont de pair avec les sous-ensembles ci-dessus. Trois programmes sont destinés à la collecte et à l'enrichissement des données d'entrée nécessaires à la simulation : `ModPhoto`, `ModGeo`, et `ModLum`. `Synthese` est le noyau du système, chargé de produire les vues à partir de toutes les données d'entrée. `Analyse` et `Traitement` complètent le dispositif par des fonctionnalités de combinaison et d'incrustation d'images.

En guise d'interface utilisateur, nous avons décidé de doter chaque programme d'un langage de commande, servant d'intermédiaire entre l'utilisateur et l'objet Scene. Des utilitaires facilitant l'implémentation des langages de commandes sont regroupés dans la catégorie `interface`.

Expérimentation & résultats

Pour amorcer l'implémentation, chaque chercheur de l'équipe a développé quelques classes concrètes standard dans la catégorie qui le concernait le plus. Les classes d'intérêt général ont été

développées par moi-même ou par les ingénieurs qui nous ont accompagnés au gré des transferts industriels. Passé les premières versions, nous avons surtout fait évolué deux produits : la plate-forme Graph'IS en soi, où les chercheurs laissaient libre court à leur imagination débridée, et le prototype *Phostere*, spécialisé pour la simulation de projets d'illumination d'édifices publics, et destiné aux éclairagistes d'EDF.

Au sein de la plate-forme Graph'IS, les centres d'intérêts des utilisateurs nous ont surtout amené à réfléchir sur l'amélioration des classes supportant les modèles physiques locaux, comme en témoigne les différents travaux que nous avons présentés :

- Pascal Deville a travaillé sur les sources lumineuses et les modèles colorimétriques, faisant emergé la problématique des classes abstraites de bas niveau, et le besoin d'utiliser des *fabriques abstraites* pour les instances de la classe `Onde`.
- Dorothée Schulz a manipulé des modèles de matériaux à plusieurs niveaux, posant le problème de l'assemblage correct des objets, et soulignant la difficulté de maintenir une représentation persistante commune à tous les modèles de réflectance.
- Christine Chevrier, pour les besoins des séquences d'images, a étendu les interfaces de la bibliothèques de classes, en butant parfois sur la structure des programmes prédéfinis.
- Slimane Merzouk a rencontré beaucoup de difficulté à remplacer l'algorithme à deux passes initial, ce qui a motivé la création d'un système de deuxième génération.

A l'exception du dernier point, le fruit de ces travaux a été intégré avec succès dans la plate-forme Graph'IS. L'ensemble a été appliqué à des projets en vraie grandeur, dans le cadre des recherches d'Isabelle Fasse sur la simulation de la lumière pour l'architecture. Au fil des projets, de multiples fonctionnalités «secondaires» ont été ajoutées :

- Cour Carrée du Louvre : premier prototype logiciel d'assitance à la conception d'éclairage.
- Place Stanislas : simulation incrémentale et incrustation dans des photographies.
- Ponts de Paris : génération de séquences d'images et incrustation dans un film vidéo.
- Cité Interdite : ajout de textures et prise en compte de l'éclairage naturel.

Cette mise en œuvre sur de vrais projets nous a permis de valider la robustesse des algorithmes implémentés, mais aussi de mesurer l'importance de faire évoluer les bases de données au même rythme que le code. Nous avons également constaté que certaines fonctionnalités exigeaient une interface graphique élaborée.

Les nouveautés de la plate-forme Graph'IS ont été transférées au prototype Phostere au fur et à mesure de leur mise au point et en fonction de leur utilité. En plus de ces transferts, nous avons essayé d'optimiser les performances en déformant le moins possible l'architecture d'origine. Deux pratiques sont représentatives :

- l'exploitation intensive de l'héritage pour traiter les cas géométriques particuliers et tirer parti de la grande quantité de facettes rectangulaires dans les scènes architecturales,

- l'utilisation des mécanismes les plus retords de C++ pour concilier encapsulation, rapidité, encombrement mémoire et flexibilité au niveau des grandeurs spectrales.

L'ajout d'une interface graphique, que nous avions anticipée, a néanmoins suscité de nombreux réaménagements. Au delà d'une gestion plus rigoureuse des affichages et des erreurs, nous avons dû procéder à des corrections imprévues :

- extraire de la bibliothèque tout le code de manipulation des fichiers et des répertoires, et laisser cette responsabilité à l'interface utilisateur,
- ajouter un accès direct à la scène pour le module de navigation en 3D, à cause de la trop grande lenteur du langage de commande,
- réorganiser la répartition des commandes sur deux programmes, l'un interactif, l'autre étant éxécuté en « tâche de fond » sur une machine distante.

Nous avons plusieurs fois regretté l'absence d'un format de fichier qui regroupe la description géométrique d'une scène ainsi que les éclairements calculés par Phostere. L'absence d'un tel format a singulièrement compliqué la connexion de Phostere aux outils de réalité virtuelle d'EDF, ainsi que la réalisation d'un module de navigation 3D et de consultation interactive des éclairements, demandée par les utilisateurs.

Discussion

L'expérience acquise autour de la plate-forme Graph'IS et du prototype Phostere, ainsi que la confrontation de notre architecture à celle du système VISION, nous ont permis de dégager quelques idées dominantes :

- notre conception est trop fondée sur les données, et pas assez sur les comportements,
- nous n'avons pas suffisamment réfléchi à la construction et à l'assemblage des objets,
- le découpage en fichiers et en programmes est trop spécifique à chaque problèmatique pour être prédéfini,
- les langages de commandes ont introduit une surcouche trop décalée du reste du système,
- notre mécanisme de persistance tend à introduire des structures de données sous-jacentes, non typées et non encapsulées,
- il faut absolument uniformiser l'implémentation des collections et la gestion des erreurs,
- l'optimisation des performances tend à contrecarrer l'architecture générique et doit intervenir le plus tard possible.

Pour une nouvelle architecture, nous préconisons d'aller plus loin dans l'utilisation des technologies à objets :

- créer des classes pour les algorithmes et spécifier des protocoles.

- ne plus raisonner en terme de programmes mais en terme de composants logiciels, éventuellement distribués,
- envisager l'usage d'une véritable base de données orientée objets,
- disposer d'un langage de script qui serve à la fois à construire, assembler et utiliser les objets.

Par ailleurs, pour faciliter le transfert industriel et multiplier les domaines d'application de nos méthodes, il nous semble indispensable d'abandonner le paradigme selon lequel le *rendu* succède à la *modélisation*, pour lui préférer un scénario où les trois composants *modélisation*, *simulation* et *visualisation* collaborent à travers un composant *scene* qui contienne à la fois la géométrie de la scène et son illumination globale. Open Inventor(SGI) semble une boîte à outils idéale pour supporter une architecture de ce type.

Suite

Pendant que s'élaborait (trop lentement) ce mémoire, une plate-forme logicielle de « deuxième génération » a été développée dans l'équipe, en corrigeant les défauts les plus flagrants de Graph'IS et en prenant en compte la plupart des recommandations que nous avons présentées ici. La nouvelle plate-forme, baptisée *Candela*, est présentée dans le mémoire de thèse de Slimane Merzouk [Mer97], que nous encourageons à consulter pour prolonger cette lecture.

Annexe A

Quelques images

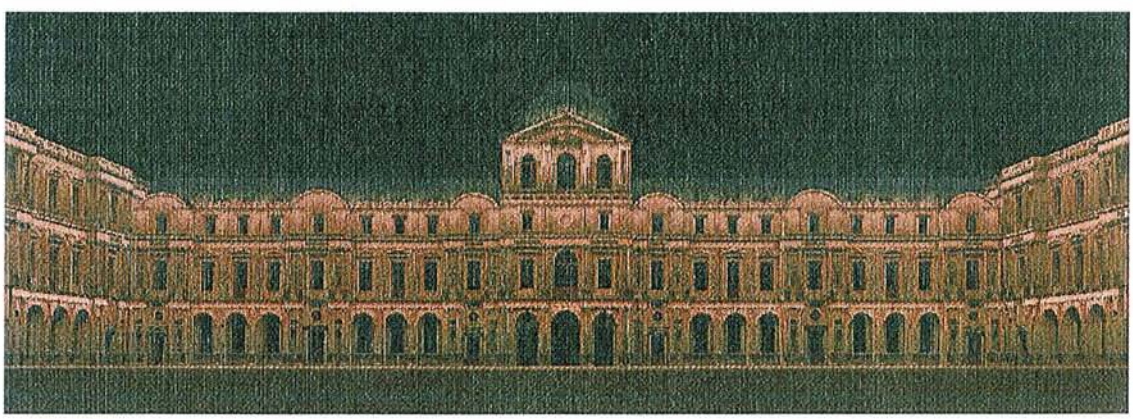


FIG. A.1 – *Cour carrée du Louvre / Paris*

FIG. A.2 – *Place Stanislas / Nancy*

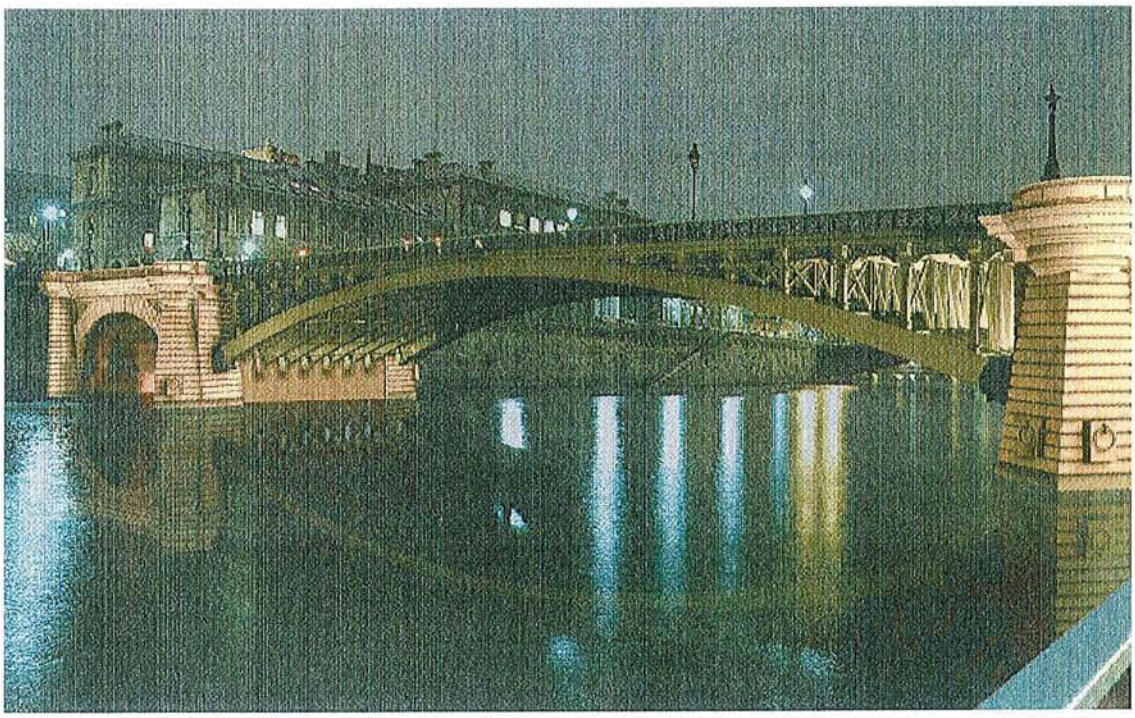


FIG. A.3 – *Pont Notre-Dame / Paris*

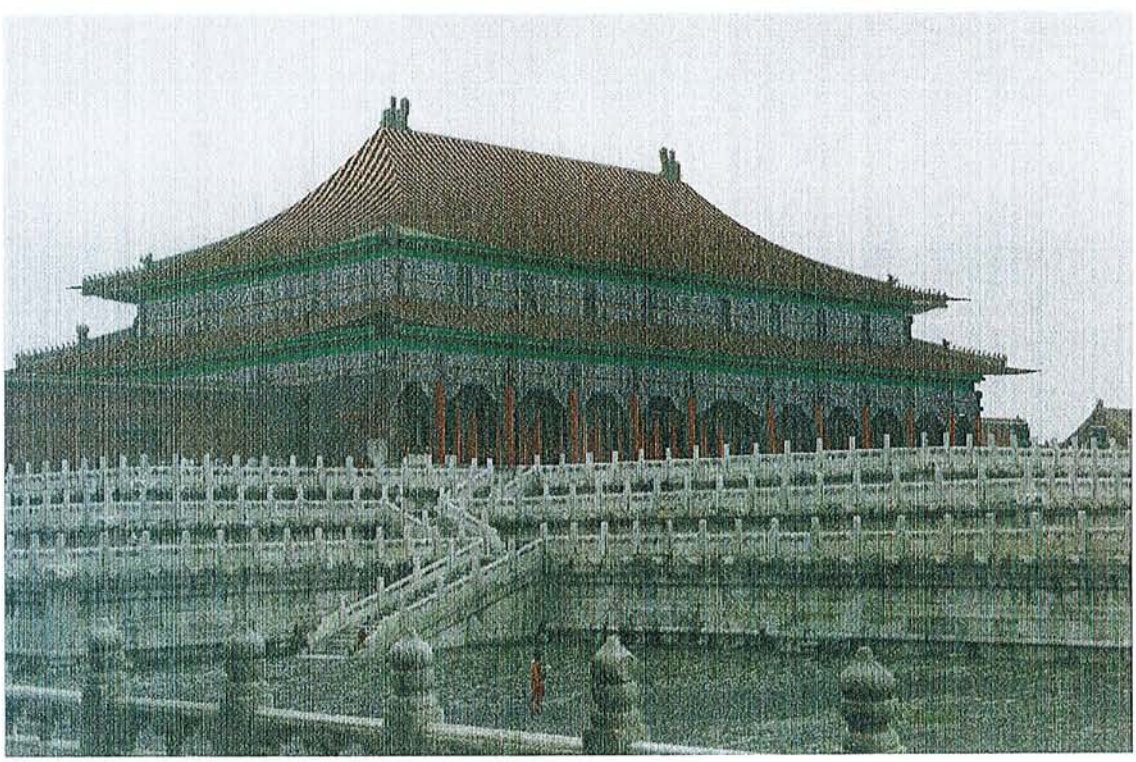


FIG. A.4 – *Cité Interdite / Pékin*

Bibliographie

[ABW+89] M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[B+94] G. Bell et al. *Open Inventor C++ Reference Manual.* Addison Wesley, Reading, Menlo Park, New York, 1994.

[Boo94] G. Booch. *Analyse et Conception Orientée Objets.* Addison Wesley, 2nd edition, 1994.

[BPP95] S. Belblidia, J. P. Perrin, and J. C. Paul. Multi-resolution rendering of architectural models. In *Computer-Aided Architectural Design Futures'95 International Conference*. School of Architecture, National University of Singapore, September 1995.

[CBP95] C. Chevrier, S. Belblidia, and J. C. Paul. Compositing computer generated images and video films. In R. A. Earnshaw and J. A. Vince, editors, *Computer Graphics: Developments in Virtual Environments*, pages 115–125. Academic Press, Leeds, England, June 1995.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987.

[CCDP94] D. Cazier, D. Chamont, P. M. Deville, and J. C. Paul. Modeling characteristics of light : A method based on measure data. In *Proceedings of the Second Pacific Conference on Computer Graphics and Applications*, pages 113–128, August 1994.

[CCWG88] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. In J. Dill, editor, *Computer Graphics (Siggraph'88 proc.)*, pages 75–84, August 1988.

[CG85] M. F. Cohen and D. P. Greenberg. The hemi-cube : A radiosity solution for complex environments. In B. A. Barsky, editor, *Computer Graphics (Siggraph'85 proc.)*, volume 19, pages 31–40, July 1985.

[Che90] S. E. Chen. Incremental radiosity : An extension of progressive radiosity to an interactive image synthesis system. In F. Baskett, editor, *Computer Graphics (Siggraph'90 proc.)*, pages 135–144, August 1990.

[Che96] C. Chevrier. *Génération de Séquences Composées d'Images de Synthèse et d'Images Vidéo*. Rapport de thèse, Université Henri Poincaré, 1996.

[Cou96] B. Coulange. *Réutilisation du Logiciel*. Masson Paris, 1996.

[CT82] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Transaction on Graphics*, 1(1) :7–24, January 1982.

[CW93] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, Inc., 1993.

[CY91a] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Englewood Cliffs, NJ : Prentice-Hall, 2nd edition, 1991.

[CY91b] P. Coad and E. Yourdon. *Object-Oriented Design*. Englewood Cliffs, NJ : Prentice-Hall, 1991.

[D+96] J. Dolan et al. Solving diverse image understanding problems using the image understanding environment. In *Proceedings of the ARPA Image Understanding Workshop*, 1996.

[DCF95] C. Donizeau, D. Chamont, and G. Fertey. Phostere : Simulation of lighting projects. In *Proceedings of the 23rd of the CIE*, pages 308–309, New Delhi, India, November 1995.

[Def92] P. Defray. *Ingénierie des objets*. Masson Paris, 1992.

[Dev96] P. M. Deville. *Modélisation et Simulation des Propriétés Radiatives des Sources Lumineuses*. Rapport de thèse, Université Henri Poincaré, 1996.

[DMCP94] P. M. Deville, S. Merzouk, D. Cazier, and J. C. Paul. Spectral data modeling for lighting applications. *The International Journal of The Eurographics Assocication, Computer Graphics Forum*, 13(3) :97–106, September 1994. Blackwell Publishers.

[DMP94] P. M. Deville, S. Merzouk, and J. C. Paul. Modeling light source for accurate simulations. In *Proceedings of the International Conference Computer Graphics International'94*, 1994.

[ES90] M. Ellis and B. Strousup. *Annotated C++ Reference Manual*. Addison Wesley, 1990.

[Fas96] I. Fasse. *Simulation d'Illumination d'Edifices Architecturaux en Image de Synthèse*. Rapport de thèse, Université Henri Poincaré, 1996.

[FPPS94] I. Fasse, J. C. Paul, J. P. Perrin, and S. Santopaulo. Accurate synthesis images for architectural design. In *Proceedings of the First Symposium Multimedia for Architecture and Urban Design*, pages 229–243, May 1994.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice*. The System Programming. Addison-Wesley, Reading, Massachusetts, second edition, 1990.

[Gla91] Andrew Glassner. Spectrum : a proposed image synthesis architecture. In *SIGGRAPH '91 Frontiers in Rendering course notes*. ACM, July 1991.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. MA : Addison Wesley, 89.

[GSCH93] S. J. Gortler, P. Schröder, M. F. Cohen, and P. Hanrahan. Wavelet radiosity. In J. T. Kajiya, editor, *Computer Graphics (Siggraph'93 proc.), Annual Conference Series*, pages 221–230, August 1993.

[GTGB84] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. In H. Christiansen, editor, *Computer Graphics (Siggraph'84 proc.)*, pages 213–222, July 1984.

[HG83] R. A. Hall and D. P. Greenberg. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications*, 3(8) :10–20, November 1983.

[HN92] M. Henricson and E. Nyquist. Programming in c++, rules and recommendations. Technical report, Ellemtel Telecommunication Systems Laboratories, Alvsjo, Suede, 1992.

[HSA91] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. In T. W. Sederberg, editor, *Computer Graphics (Siggraph'91 proc.)*, volume 25, pages 197–206, July 1991.

[HSS94] W. Heidrich, P. Slusallek, and H. P. Seidel. Using c++ class libraries from an interpreted language. *Technology of Object-Oriented languages and systems*, pages 397–408, 1994.

[HTSG91] X. D. He, K. E. Torrance, F. X. Sillion, and D. P. Greenberg. A comprehensive physical model for light reflection. In T. W. Sederberg, editor, *Computer Graphics (Siggraph'91 proc.)*, volume 25, pages 175–186, July 1991.

[JCJO93] I. Jacobson, M. Christeron, P. Jonsson, and G. Overgaard. *Le Génie Logiciel Orienté Objet*. Addison-Wesley France, 2nd edition, 1993.

[KA88] David Kirk and James Arvo. The ray tracing kernel. In *Proceedings of Ausgraph '88*, pages 75–82, 1988.

[Kaj86] J. T. Kajiya. The rendering equation. In D. C. Evans and R. J. Athay, editors, *Computer Graphics (Siggraph'86 proc.)*, volume 20, pages 143–150, August 1986.

[KM94] C. Kohl and J. Mundy. The development of the image understanding environment. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1994.

[Ler97] R. Lerner. The image understanding environment program : Progress since iuw'96. In *Proceedings of the 1997 DARPA Image Understanding Workshop*, 1997.

[Lip91] S. B. Lippman. *C++ Primer*. MA : Addison-Wesley, 2nd edition, 1991.

[LTG93] D. Lischinski, F. Tampieri, and D. P. Greenberg. Combining hierarchical radiosity and discontinuity meshing. In J. T. Kajiya, editor, *Computer Graphics (Siggraph'93 proc.), Annual Conference Series*, pages 199–208, August 1993.

[M+92] J. Mundy et al. The image understanding environment program. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1992.

[McL93] M.J. McLennan. [incr tcl] : Object-oriented programming in tcl. In *Proceedings of the Tcl/Tk Workshop*, 1993.

[Mer97] S. Merzouk. *Architecture Logicielle et Algorithmes pour la Résolution de l'Équation de Radiance*. Rapport de thèse, Université Henri Poincaré, july 1997.

[Mey88a] B. Meyer. *Object-Oriented Software Construction*. NY : Prentice Hall, 1988.

[Mey88b] G. W. Meyer. Wavelength selection for synthetic image generation. *Computer Vision, Graphics and Image Processing*, 41 :57–79, 1988.

[MNC+91] G. Masini, A. Napoli, D. Colnet, D. Leonard, and K. Tombre. *Object Oriented Languages*. Academic Press, London, 1991.

[NF87] Tom Nadas and Alain Fournier. GRAPE : An environment to build display processes. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 75–84, July 1987.

[PDW95] J. C. Paul, P. M. Deville, and C. Winkler. Modelling radiative properties of light sources and surfaces. *The Journal of Visualization and Computer Animation*, 6(4) :231–240, October 1995.

[PH87] M. Potmesil and E. M. Hoffert. FRAMES : Software tools for modeling, rendering and animation of 3D scenes. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 85–93, July 1987.

[PSV90] C. Puech, F. X. Sillion, and C. Vedel. Improving interaction with radiosity-based lighting simulation programs. In R. Riesenfeld and C. H. Séquin, editors, *Computer Graphics (Symposium'90 on Interactive 3D Graphics)*, volume 24, pages 51–57, March 1990.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. E. Lorensen. *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ : Prentice-Hall, 1991.

[San94] J. Dos Santos. Etude d'une méthode de radiosité incrémentale et amélioration de l'interactivité. Rapport de dea, Université Henri Poincaré, 1994.

[SC96] D. Schulz-Cazier. *Modélisation Physique et Simulation des propriétés radiatives des surfaces*. Rapport de thèse, Université Henri Poincaré, 1996.

[Sch94] P. Schröder. *Wavelet Algorithms for Illumination Computations*. PhD thesis, Princeton University, November 1994.

[SH81] R. Siegel and J. R. Howell. *Thermal Radiation Heat Transfer*. Hemisphere Publishing Corporation, 79 Madison Avenue, New York, New York 10016, second edition, 1981.

[Slu95] P. Slusallek. *Vision - An Architecture for Physically Based Image Synthesis*. PhD thesis, Computer Graphics Group, University of Erlangen, Erlangen, Germany, 1995.

[SP89] F. X. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. In J. Lane, editor, *Computer Graphics (Siggraph'89 proc.)*, volume 23, pages 335–344, July 1989.

[SPS95] P. Slusallek, T. Pflaum, and H.-P. Seidel. Using procedural renderman shaders for global illumination. *Computer Graphics Forum (Eurographics '95)*, 14(3), Septem ber 1995.

[SS95] P. Slusallek and H. P. Seidel. Vision - an architecture for physically based image synthesis. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):77–96, March 1995.

[SSB91] Peter Shirley, Kelvin Sung, and William Brown. A ray tracing framework for global illumination systems. In *Proceedings of Graphics Interface '91*, pages 117–128, June 1991.

[Ste90] Guy L. Steele. *Common Lisp the Language*. Digital Press, 2nd edition, 1990.

[TLG91] B. Trumbore, W. Lytle, and D. P. Greenberg. A testbed for image synthesis. In Werner Purgathofer, editor, *Eurographics '91*, pages 467–480. North-Holland, September 1991.

[Ups90] S. Upstill. *The Renderman Companion*. Addison Wesley, 1990.

[VG94] E. Veach and L. Guibas. Bidirectionnal estimators for light transport. In *eurow94*, pages 147–162, Darmstad, June 1994.

[War92] G. J. Ward. Measuring and modeling anisotropic reflection. In E. E. Catmull, editor, *Computer Graphics (Siggraph'92 proc.)*, volume 26, pages 265–272, July 1992.

[War94] Gregory J. Ward. The RADIANCE lighting simulation and rendering system. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 459–472. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[War95] G. Ward. Making global illumination user friendly. In P. M. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95 (Proceedings of the Eurographics Workshop in Dublin, Ireland, June 12-14, 1995)*, pages 104–114, New York, 1995. Springer-Verlag.

[WBJ90] R. J. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[WBWW90] R. J. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, NJ : Prentice-Hall, 1990.

[WEH89] J. R. Wallace, K. A. Elmquist, and E. A. Haines. A ray tracing algorithm for progressive radiosity. In *Computer Graphics (Siggraph'89 proc.)*, volume 23, pages 315–324, July 1989.

[Wer94a] J. Wernecke. *The Inventor Mentor*. Addison Wesley, Reading, Menlo Park, New York, 1994.

[Wer94b] J. Wernecke. *The Inventor Toolmaker*. Addison Wesley, Reading, Menlo Park, New York, 1994.

[WH92] Gregory J. Ward and Paul Heckbert. Irradiance gradients. *Third Eurographics Workshop on Rendering*, pages 85–98, May 1992.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[Whi82] T. Whitted. A software testbed for the development of 3d raster graphics systems. *ACM Transaction on Graphics*, 1(1):43–58, January 1982.

[Wis96] P. Wisskirchen, editor. *Object-Oriented and Mixed Programming Paradigms*. Springer-Verlag, 1996.

[WRC88] G. J. Ward, F. M. Rubinstein, and R. D. Clear. A ray tracing solution for diffuse interreflection. In J. Dill, editor, *Computer Graphics (Siggraph'88 proc.)*, volume 22, pages 85–92, August 1988.

[WRH92] C. Williams, J. Rasure, and C. Hansen. The state of the art of visual languages for visualization. In *Proceedings of Visualization '92*, pages 202–209, 1992.

[Zat93] H. R. Zatz. Galerkin radiosity : A higher order solution method for global illumination. In J. T. Kajiya, editor, *Computer Graphics (Siggraph'93 proc.), Annual Conference Series*, pages 213–220, August 1993.

Nom : **CHAMONT**

Prénom : **David**

DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 24 NOV 1997 n°109

Le Président de l'Université

J.P. FINANCE

3C-F9

UNIVERSITÉ HENRI POINCARÉ - NANCY 1
24-30, rue Lionnois B.P.3069 54013 Nancy cedex
Tél : 83 32 81 81 Fax : 83 32 95 90

Résumé

La synthèse d'images est fondée depuis quelques années sur des bases physiques et mathématiques plus rigoureuses, notamment la modélisation des propriétés radiatives des surfaces et la simulation par éléments finis des transferts radiatifs entre ces surfaces. Une telle évolution permet d'envisager de nouvelles applications industrielles, en particulier en ingénierie de l'éclairage.

Ce mémoire présente une architecture logicielle à base d'objets, conçue pour soutenir la recherche dans les disciplines que nous venons d'évoquer et faciliter le développement de prototypes industriels. Elle repose d'une part sur une bibliothèque de classes abstraites tirées de l'équation de transfert, d'autre part sur un système de fichiers et de programmes calqués sur les étapes et les tâches d'une simulation.

À travers différents projets de recherche et plusieurs applications, nous montrons comment notre architecture a été mise en œuvre avec succès, en particulier pour évaluer de nouveaux modèles d'émission ou de réflexion de la lumière. Nous discutons également ses lacunes dans la représentation des algorithmes et la mise en œuvre du paradigme objet.

Mots-clés: illumination globale, radiosité, synthèse d'images, architecture logicielle