

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PARIS

MÉMOIRE

présenté en vue d'obtenir le

DIPLÔME D'INGÉNIEUR C.N.A.M.

en

INFORMATIQUE

par

Alexandre TOPOL

VRML : étude, mise en œuvre et applications

soutenu le 6 Juillet 2001

Jury

Président : Stéphane Natkin (CNAM)

Membres : Thomas Baudel (Ilog Views)

Pierre Cubaud (CNAM)

Cédric Guiard (Mende13D / DURAN-DUBOI)

Dan Vodislav (CNAM)

Résumé : Les interfaces en 3 dimensions, autorisant une manipulation en temps réel du point de vue et des objets, se multiplient ces dernières années. Alors qu'il y a encore peu de temps elles étaient réservées à des applications de C.A.O. et de simulation s'exécutant sur des serveurs graphiques spécialisés, elles sont maintenant présentes sur des ordinateurs personnels, ouvrant ainsi une large palette de domaines dans lesquels la 3D est potentiellement utilisable et, plus important, potentiellement utile. Le frein majeur au développement d'applications 3D était jusqu'alors dû à la puissance de calculs requise. La présence dans les ordinateurs d'entrée de gamme de cartes graphiques accélératrices ainsi que d'instructions spécialisées pour les calculs matriciels dans les nouveaux microprocesseurs permettent, désormais, d'envisager de nouveaux domaines d'utilisation de la 3D.

La diffusion d'applications 3D depuis le concepteur jusqu'à l'utilisateur final peut se faire selon plusieurs vecteurs. Tout d'abord, par des applications exécutables. Pour celles-ci, le programmeur pourra utiliser des bibliothèques de fonctions 3D (API 3D) disponibles sous la forme de librairies dynamiques. Elles offrent le double avantage de ne pas à avoir à programmer les différentes étapes du pipeline graphique 3D et de profiter de l'accélération matérielle. En effet, pour les API 3D les plus couramment utilisées (comme Direct3D ou OpenGL), le matériel présent est directement mis à contribution pour la prise en charge de certaines fonctions 3D. Les fonctions non supportées par le matériel seront alors émulées par le microprocesseur. Des drivers faisant le lien entre l'API 3D et la carte graphique permettront donc au programmeur de faire abstraction du matériel présent. Il n'aura donc pas à programmer explicitement les registres de la carte graphique ni à écrire des fonctions bas-niveau.

La seconde manière de diffuser un contenu 3D est d'utiliser un ensemble modéleur et moteur de rendu. Ceux-ci doivent être capables d'échanger leurs données par un format de fichier commun. Pratiquement tous les derniers modéleurs disposent d'un filtre pour exporter les scènes construites au format VRML. Ces fichiers peuvent ensuite être interprétés et affichés par un co-navigateur (*plugin*). D'autres formats de fichier associés à leur *plugin* sont également utilisables pour afficher une scène stockée sur un serveur HTTP.

Sont couverts dans ce rapport, les aspects importants de la 3D temps réel ; aussi bien les fondements, que la présentation des API et des *plugins*. Et parmi ces derniers, nous étudierons plus en profondeur VRML qui est la seule norme publique actuellement. Pour illustrer de manière pratique les concepts de VRML, un navigateur, s'appuyant aussi bien sur OpenGL que sur Direct3D, a été développé. En marge de l'écriture de ce rapport et de ce *plugin* maison, nous présentons également quelques travaux parallèles.

Mots-clés : Synthèse d'images en temps-réel, API 3D, VRML, Navigation 3D, Interaction 3D.

Keywords : Real-time 3D rendering, 3D API, VRML, 3D browsing, 3D interaction.

REMERCIEMENTS

Avant tout, mille mercis à Pierre (Cubaud) pour son extrême patience. Faire précéder « patience » de l'adjectif « extrême » est à peine exagéré dans mon cas ! Il a su attendre deux longues années – malgré une insoutenable envie de lire mon mémoire et, plus matériellement, le droit mérité de toucher ses primes d'encadrement – sans pour autant me faire ressentir son impatience. Plus sérieusement, je tiens à dire qu'il m'a parfaitement guidé et qu'il a su me pousser juste comme il le fallait quand il le fallait. En marge de ses qualités évidentes d'encadrant, je souligne, ici, ses talents dans les rapports humains.

Je remercie également les membres de mon jury pour cette même raison de délai. La plupart ont été contactés il y a un moment déjà et ont accepté aussitôt de faire partie de ce jury. Ils n'ont jamais manifesté non plus leur impatience, ce qui m'a permis de finir ce rapport sereinement.

Une pensée revient aux membres de ma famille. Je leur promets de trouver un nouveau sujet de conversation pour animer les repas familiaux. Il paraît qu'il y a une thèse à rédiger cette année également ; suffirait-il de remplacer « alors ce mémoire ? » par « alors cette thèse ? » ? J'espère que non.

Je n'oublie pas et j'embrasse ceux qui m'ont poussés à suivre les cours du Conservatoire National des Arts et Métiers à un moment où je n'avais plus le goût des études. En particulier, ma mère, mon père, Catherine, Michel, mes grands-parents, Pierre et Rachel, Fran' et Jean'P, veuillez considérer ce mémoire comme une preuve de vos bons conseils. Le CNAM, vers lequel vous m'avez aiguillé, a su recycler l'élève médiocre que j'étais dans le circuit universitaire.

Enfin, une pensée particulière à Nathalie qui, bon gré mal gré, m'a poussé à donner le coup de rein final pour boucler ce mémoire et tourner cette page de ma vie...

TABLE DES MATIERES

INTRODUCTION	7
1 INTRODUCTION A LA 3D	9
1.1 GENERALITES SUR LA 3D	9
1.1.1 <i>Qu'est-ce que la 3D ? Pourquoi l'utiliser ?</i>	9
1.1.2 <i>Deux méthodes de rendu 3D : différences et utilités</i>	12
1.1.3 <i>Les bases de la 3D temps réel</i>	16
1.1.4 <i>Optimisation possible du pipeline 3D</i>	23
1.2 POSITIONNEMENT DES DIFFERENTS ELEMENTS DANS UNE ARCHITECTURE 3D	27
1.2.1 <i>Couche Accélération Matérielle</i>	27
1.2.2 <i>Couche Interface Logicielle</i>	30
1.2.3 <i>Couche Emulation</i>	31
1.2.4 <i>Couche Application</i>	31
1.3 PRESENTATION DE L'ARCHITECTURE 3D SOUS LINUX	32
2 QUELQUES API 3D DISPONIBLES.....	37
2.1 QUICKDRAW3D	38
2.2 OPENGL – MESA	40
2.2.1 <i>Les bibliothèques d'outils</i>	42
2.2.2 <i>Squelette de code OpenGL</i>	43
2.2.3 <i>Avantages et inconvénients de l'utilisation d'OpenGL</i>	48
2.3 DIRECT3D.....	49
2.3.1 <i>Création d'une application</i>	51
2.3.2 <i>Avantages et inconvénients</i>	56
2.4 JAVA3D	57
2.4.1 <i>Le graphe de scène</i>	58
2.4.2 <i>Construction d'une application en Java3D</i>	61
2.4.3 <i>Avantages et inconvénients de la solution Java3D</i>	62
2.5 FAHRENHEIT	63
3 LES OUTILS 3D ET LE WEB.....	67
3.1 QUELQUES MODELEURS	68
3.2 LES PRINCIPAUX MOTEURS 3D POUR LE WEB	70
3.2.1 <i>VRML</i>	73
3.2.2 <i>Superscape – Viscap</i>	79
3.2.3 <i>Scol et Blaxxun</i>	79
3.2.4 <i>MPEG4</i>	80
3.2.5 <i>Metastream</i>	83
3.3 DESCRIPTION DE LA NAVIGATION 3D DANS CES OUTILS	85
3.4 ETUDE PLUS APPROFONDIE DE VRML.....	86
3.4.1 <i>Les nœuds de regroupement</i>	87
3.4.2 <i>Les nœuds enfants</i>	88
3.4.3 <i>Les nœuds dépendants</i>	90
3.4.4 <i>Le format de fichier</i>	92
4 UN NAVIGATEUR VRML	97
4.1 LES DIFFERENTES ETAPES	98
4.1.1 <i>Analyse lexicale avec Lex</i>	98
4.1.2 <i>Analyse syntaxique avec Yacc</i>	100

4.1.3	<i>Traduction des nœuds VRML en appels de fonctions 3D</i>	106
4.2	LES DIFFERENTES APPROCHES POUR LA LECTURE DES FICHIERS VRML	108
4.2.1	<i>L'approche récursive</i>	108
4.2.2	<i>Solution avec une table des fichiers</i>	111
4.3	CONTROLES SUR LE RENDU ET AIDES A LA NAVIGATION	111
4.3.1	<i>Options de visualisation</i>	111
4.3.2	<i>Manipulation de la caméra</i>	113
4.3.3	<i>Aides à la navigation</i>	119
4.4	TRANSFORMATION DE L'APPLICATION EN UN PLUGIN VRML	120
4.4.1	<i>Les fonctions de gestion globale</i>	121
4.4.2	<i>Les fonctions de gestion des instances</i>	122
4.5	TRAVAIL RESTANT A FAIRE SUR LE PARSEUR	123
5	APPLICATIONS, TRAVAUX FUTURS	127
5.1	APPLICATIONS	127
5.1.1	<i>Simulateur Urbain</i>	127
5.1.2	<i>Interfaces 3D pour les bibliothèques numériques</i>	130
5.1.3	<i>Un bureau 3D</i>	139
5.2	TRAVAUX FUTURS	143
5.2.1	<i>Comportements 3D</i>	143
5.2.2	<i>Intégration du son spatialisé</i>	145
	CONCLUSION	147
	GLOSSAIRE	149
	TABLE DES FIGURES	155
	REFERENCES	157

INTRODUCTION

Depuis déjà quelques années, les évolutions technologiques en micro-informatique se succèdent à une vitesse impressionnante pour répondre principalement au besoin toujours croissant de puissance, requise pour les applications multimédias et notamment celles relativement récentes exploitant une interface 3D. En effet, non seulement les machines actuelles ont des microprocesseurs permettant d'effectuer de nombreuses opérations en virgules flottantes nécessaires à la 3D (pour les transformations, pour les projections, ...) mais aussi, ils embarquent des cartes graphiques bon marché dans lesquelles les fonctions 3D les plus courantes sont câblées (c'est-à-dire prises en charge par les cartes elles-mêmes). Jusqu'à présent, les performances de ces cartes ont quasiment été doublées tous les ans et les premières cartes de troisième génération, intégrant un module pour l'éclairage et les transformations, n'infléchissent pas, loin s'en faut, cette tendance. Les innovations dans le domaine de l'accélération 3D concernent même les microprocesseurs eux-mêmes qui intègrent des instructions spécifiques à la 3D pour les calculs matriciels par exemple.

Pour avoir un ordre d'idée de l'évolution des ordinateurs, entre 1945, date à laquelle l'armée américaine utilisait le premier ordinateur électronique (l'ENIAC – *Electronic Numerical Integrator And Computer*) pour calculer des tables d'artillerie, et 1999, les performances ont doublé (à prix égal) tous les dix-huit mois. Cela correspond à une multiplication par cent des performances tous les dix ans à prix égal. Ainsi, une calculatrice de poche coûtant vers les cent francs est aujourd'hui cent fois plus performante que le calculateur de l'ENIAC. Le progrès constant des microprocesseurs doublé de celui des cartes graphiques font de l'image de synthèse en temps-réel le domaine qui connaît le plus fort accroissement de puissance (300% tous les ans pour un prix égal).

Cette augmentation de puissance pour le traitement de la 3D ainsi que l'existence de bibliothèques de procédures de haut niveau tirant partie de ces possibilités rendent accessibles et facilitent l'utilisation de techniques d'interaction 3D. A l'heure actuelle, ces nouvelles possibilités dégagées par un ordinateur personnel permettent d'envisager non plus uniquement des interfaces 3D pour les simulateurs, les jeux ou les modélisateurs, mais également pour des applications courantes. Depuis 1996, le langage VRML, couplé à un co-navigateur ou *plugin* (une application associée à un navigateur HTML pour gérer un type de données spécifique) sachant l'interpréter, autorise la mise en ligne de scènes 3D réactives. Le but premier de ce langage était de mettre à disposition sur Internet un environnement 3D multi-utilisateur. Or, après quelques années, il s'avère que VRML ne remplit pas du tout ce rôle pour lequel il a été créé. On constate qu'il permet avant tout de décrire des composants génériques réutilisables dans n'importe quelle application exploitant une interface 3D. Et, grâce à cette utilisation beaucoup moins restrictive, VRML embrasse une palette d'applications beaucoup plus large que celles initialement visées. Dès lors, il nous a semblé naturel d'étudier en profondeur ses possibilités. Cette étude s'est déclinée en deux parties indissociables. La première visait à évaluer quantitativement les possibilités du langage par l'écriture complète d'un navigateur VRML. Il existe bien évidemment de nombreux navigateurs VRML. Cependant, le développement d'un navigateur propriétaire permet de prendre facilement des mesures de performances pour valider l'utilisation de la 3D pour la visualisation d'informations (dans le sens le plus général possible). Le second aspect de cette étude vise à déterminer les utilités de ce langage. Notre navigateur peut en effet être utilisé pour intégrer des objets VRML dans une application autonome.

Ce mémoire d'ingénieur fait suite à une spécialisation valeur C « conception des applications multimédias » dont le sujet était l'étude de faisabilité et la réalisation partielle d'un simulateur urbain. A partir du relief et du plan d'occupation des sols d'un lieu donné, notre programme fournissait un environnement en VRML dans lequel l'utilisateur pouvait librement se déplacer. Ce mémoire vient également après le D.E.A. médias et multimédia [TOP98] et intervient parallèlement à mon travail de thèse.

Avec le simulateur urbain nous nous sommes aperçus que la navigation dans une scène 3D, décrite en langage VRML et visualisée avec un *plugin* associé, était acceptable en terme de fluidité dès lors que la scène 3D comportait un nombre raisonnable de polygones et de textures. Dans le cadre du D.E.A., nous avons voulu élargir ce principe à la visualisation d'un fonds ancien en pensant que l'espace 3D permettait d'organiser les ouvrages pour, d'une part, réduire les mouvements de l'utilisateur et, d'autre part, classer les livres selon certains critères de recherche. S'agissant de livres anciens dont la tranche est une information essentielle pour aider les spécialistes dans leurs recherches, nous devons modéliser les livres en autant de faces texturées, ce qui causait un effondrement rapide des performances avec l'augmentation du nombre de livres [CUB98]. Bien que dans ce contexte particulier la puissance requise pour l'utilisation d'une interface 3D n'ait pas été suffisante et contrariait ce projet, l'aspect d'organisation des informations dans l'espace nous a séduit d'où la poursuite de ce travail par la proposition de ma thèse portant plus généralement sur les interactions 3D dans les systèmes d'informations réparties.

Alors que la thèse vise plus particulièrement à effectuer une étude théorique des interfaces 3D et de leurs applications, ce mémoire a permis d'étudier plus spécifiquement les outils de la 3D. Il se veut donc être un état de l'art sur les techniques les plus connues offertes pour construire une interface 3D. Les outils que nous présentons sont des moteurs 3D, disponibles sous forme de *plugin*, mais surtout, des bibliothèques de procédures 3D (API¹ 3D). En effet, que ce soit pour le manque d'informations sur le fonctionnement de leur moteur 3D ou pour les interactions trop générales, l'utilisation des *plugins* peut ne pas être satisfaisant dans certain cas. L'apprentissage des API 3D en vue de leur utilisation pour des expérimentations sur mesure (plutôt que de persévérer dans des solutions comme VRML, qui s'avéreront nécessairement insuffisantes par la suite, dans l'état actuel de ce langage), est par conséquent, nous en sommes convaincus, une tâche importante pour la suite de ma thèse.

Pour continuer néanmoins à suivre l'évolution du langage VRML et surtout pour étudier un exemple pratique d'utilisation des API 3D, nous verrons la mise en œuvre d'un *plugin* VRML. Il nous permettra de récupérer des mesures sur les performances des différentes API 3D sur différents systèmes et de tester les possibilités offertes. Bien qu'il en existe un bon nombre pour les ordinateurs fonctionnant sous Windows 95/98/NT/2000/ML, sous Linux ce projet était à l'origine de ce mémoire l'aspect le plus novateur puisqu'il n'en existait pas pour ce système d'exploitation. Cependant, depuis lors, la bibliothèque VRMLLib est apparue sur ce système d'exploitation.

La première partie de ce rapport introduit les généralités sur la 3D et décrit les différentes couches d'une architecture 3D générique. La deuxième présente, les principales librairies de procédures (API) autorisant un programmeur, soit à créer une application 3D « sur mesure », soit à intégrer des fonctions 3D à un logiciel. Puis nous verrons les outils, s'appuyant généralement sur les API précédentes, utilisés pour l'édition et/ou le rendu d'une scène 3D. Ces outils sont les modeleurs qui aident à la création d'un « monde » 3D et les *plugins*, capables d'interpréter certains fichiers décrivant une scène et d'afficher leur contenu. Dans cette partie, nous verrons de façon plus approfondie le format de fichier et les moteurs VRML. La connaissance du format de fichier VRML et des API 3D nous permettront ensuite de comprendre le fonctionnement de l'interpréteur (*parser*) de fichier VRML et la traduction des nœuds VRML en des appels de fonctions des API 3D pour afficher le contenu du fichier. Les deux chapitres suivants expliquent les utilisations que l'étude de VRML et le traducteur VRML-API 3D développé ont permis : un co-navigateur VRML pour Netscape Navigator et quelques travaux parallèles.

¹ *Application Program Interface* : bibliothèques de fonctions accessibles depuis un programme par chargement de sa librairie (dynamique le plus souvent).

1 INTRODUCTION A LA 3D

Résumé Nous revenons sur les fondements de la 3D. D'une part, nous y voyons les intérêts de son utilisation et les domaines principaux dans lesquels ces interfaces sont utilisées. D'autre part, nous rappellerons comment sont obtenues les images sur un écran 2D à partir d'une scène décrite en coordonnées 3D. Nous présentons ensuite une architecture 3D générique donnant la position des différents éléments 3D dont certains seront étudiés en détail dans la suite de ce rapport. L'architecture 3D de Linux est ensuite décrite pour fournir un exemple concret.

1.1 Généralités sur la 3D

1.1.1 Qu'est-ce que la 3D ? Pourquoi l'utiliser ?

La plupart des opérations d'entrée/sortie à partir des périphériques standards se font en deux dimensions : les entrées par des dispositifs 2D comme la souris ou le *joystick* qui comptent deux degrés de liberté (deux translations) et les sorties sur imprimante ou moniteur qui ne peuvent que coucher des images à plat. Des périphériques spécialisés existent cependant pour interagir réellement en 3D ou pour obtenir une sortie 3D. Ces périphériques 3D se sont en effet multipliés et l'on compte désormais parmi eux les gants de réalité virtuelle et les souris 3D offrant les six degrés de liberté (trois translations et trois rotations). La plupart des dispositifs de visualisation 3D sont cependant encore réservés aux domaines de la recherche, des technologies de pointe ou des simulateurs militaires. C'est le cas notamment des systèmes de représentation holographique comme le *MIT holovideo display* [WAT95], des caves équipées d'écrans offrant une vue panoramique, des *workbenches* [OBE96] où les images sont projetées sur une table et des imprimantes 3D créant des volumes.

Toutes les configurations actuelles bien que ne possédant pas ces périphériques embarquent cependant des possibilités 3D. 3D étant l'abréviation de trois dimensions, comment expliquer que sur un écran en deux dimensions nous pouvons obtenir une sortie 3D ? C'est à ce point que la définition de la 3D pour l'informatique est quelque peu différente de celle du monde physique puisqu'elle intègre également, en plus de la manipulation d'objets 3D, la manipulation d'images 2D (projections à l'écran d'objets décrits dans l'espace) donnant l'impression de volume. Les techniques utilisées pour créer cette impression seront présentées plus loin dans ce rapport. Mais précisons dès à présent que toutes les méthodes de création et de visualisation d'un contenu 3D dont nous parlerons utilisent toutes ces techniques et ne permettent donc que de donner à l'utilisateur l'illusion qu'il évolue dans un « monde 3D ». La plus grande immersion à un prix abordable est obtenue avec les casques de réalité virtuelle ou les lunettes grâce au rendu stéréoscopique. Dans les deux cas, deux images sont calculées, une par œil. La seule différence entre ces deux images provient d'une légère translation de la prise de vue sur la scène pour simuler l'écartement des yeux et donc la vision depuis chacun des yeux. Pour le casque, les deux images sont affichées simultanément sur deux écrans à cristaux liquides différents (un par œil). Avec les lunettes, le rendu est effectué sur le moniteur. L'effet stéréoscopique est obtenu par l'affichage alterné des images calculées. Pour percevoir le relief, les lunettes servent d'obturateur. Elles sont, en effet, synchronisées avec l'affichage et lorsque l'image pour un œil est affichée, l'autre œil est obturé pour ne pas la voir. Une fréquence double – 50 images par seconde – est nécessaire pour satisfaire la tolérance individuelle de chacun des yeux.

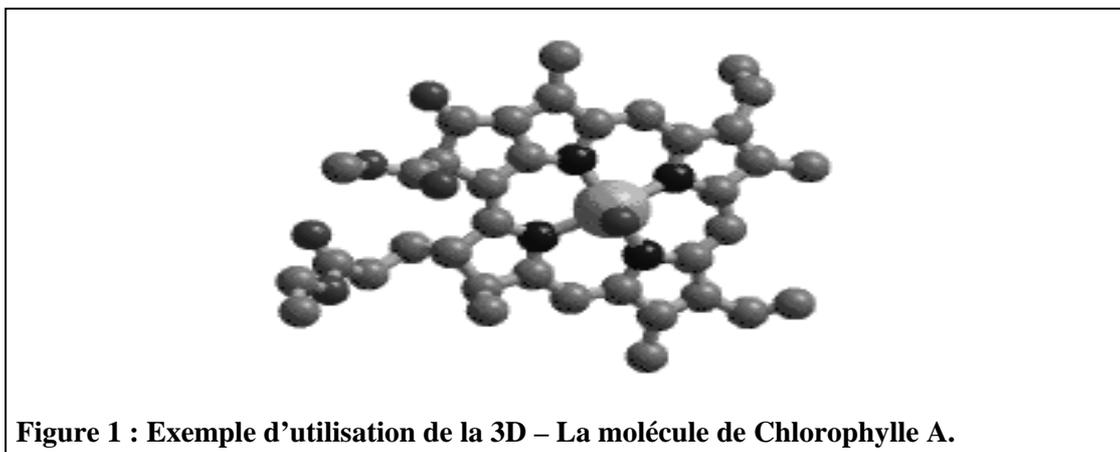
La principale différence entre le monde physique que nous percevons en 3D ou la représentation holographique et la fausse 3D plus communément utilisée est l'immobilité des objets lorsque nous bougeons la tête en face de l'écran. Ces systèmes (mis à part lorsque l'on utilise un casque avec capteurs de position) ne permettent pas, par une translation du regard, de découvrir plus d'information sur le volume d'un objet. Cependant, l'illusion est suffisante pour donner à l'utilisateur

le sentiment d'être plongé dans un univers 3D ; d'autant plus s'il utilise un environnement immersif visuel et sonore.

Tous les modes de représentation et les techniques de conception basés sur la 3D servent dans la plupart des applications à fournir une compréhension accrue d'un modèle comportant en réalité les trois dimensions. Leur but principal est de rendre l'environnement de travail (autrement dit l'interface entre l'homme et la machine) plus réaliste et visuel. Pour la simulation et la conception assistée par ordinateur (C.A.O.), les deux premières utilisations industrielles de la 3D, le but est de permettre la manipulation et la simulation de comportements selon certains paramètres. Dans le cas de la C.A.O., les pièces mécaniques sont créées, assemblées et testées avant leur mise en fabrication. Les modèles 3D des pièces, associés aux paramètres physiques de simulation (vitesses de rotation, frottements par exemple) permettent d'obtenir les usures au bout d'une durée donnée ou leur temps théorique de bon fonctionnement.

En ce qui concerne l'utilisation de la 3D dans les films, le but recherché est de créer un maximum d'émotions chez le spectateur en le plongeant directement dans une action irréaliste ou difficile à mettre en œuvre par des effets spéciaux physiques ou mécaniques. Pourtant, la 3D n'a pas qu'un rôle ludique puisqu'elle est utilisée dans les laboratoires de recherche, les centres de conception, pour toutes les études de design, en architecture et également de plus en plus dans l'art numérique.

Comme on peut le constater en regardant toute chronologie des événements ayant marqués l'IHM, un des premiers domaines dans lequel la 3D a été employée est la simulation et la visualisation de phénomènes réels. Dans les laboratoires de chimie, la 3D est utilisée pour obtenir des représentations de molécules complexes manipulables à souhait. La 3D est un outil très puissant permettant la représentation de l'infiniment petit. Même si une molécule peut être perçue au microscope, l'étude de sa structure est rendue bien plus simple par sa représentation 3D. Son utilisation devient obligatoire si l'on étudie des objets encore plus petits, impossibles à voir, même avec les plus puissants microscopes. C'est par exemple le seul moyen de représenter le produit d'une collision entre deux particules à l'intérieur d'un accélérateur.



En astronomie, règne de l'infiniment grand, la 3D est également largement utilisée. Il est possible de suivre le mouvement des astres ou tout simplement d'obtenir une carte animée du ciel vue d'un point quelconque du temps et de l'espace. On peut ainsi faire des promenades virtuelles dans le système solaire ou dans l'univers. Par simulation, il est possible de visualiser les dégâts occasionnés par une météorite. Une des premières animations 3D était pour visualiser l'orbite d'un satellite artificiel autour de la terre par E. Zajac au BTL (*Bell Telephone Laboratory*). A encore plus grande échelle, il est également possible de visualiser les collisions de galaxies.

La 3D est aussi utile en cartographie pour représenter les informations relevées par les satellites ou par les topographes sur le terrain. A partir des fichiers de données de ces relevés

topographiques accessibles à tous (par les fichiers DEM (*Digital Elevation Model*) ou bientôt sur le site *Web* de l'Institut Géographique National), il est possible de présenter les différentes informations géographiques comme dans le projet de J.M. Le Gallic [LEG99] ou d'effectuer une simulation dans un milieu en construction [TOP97].

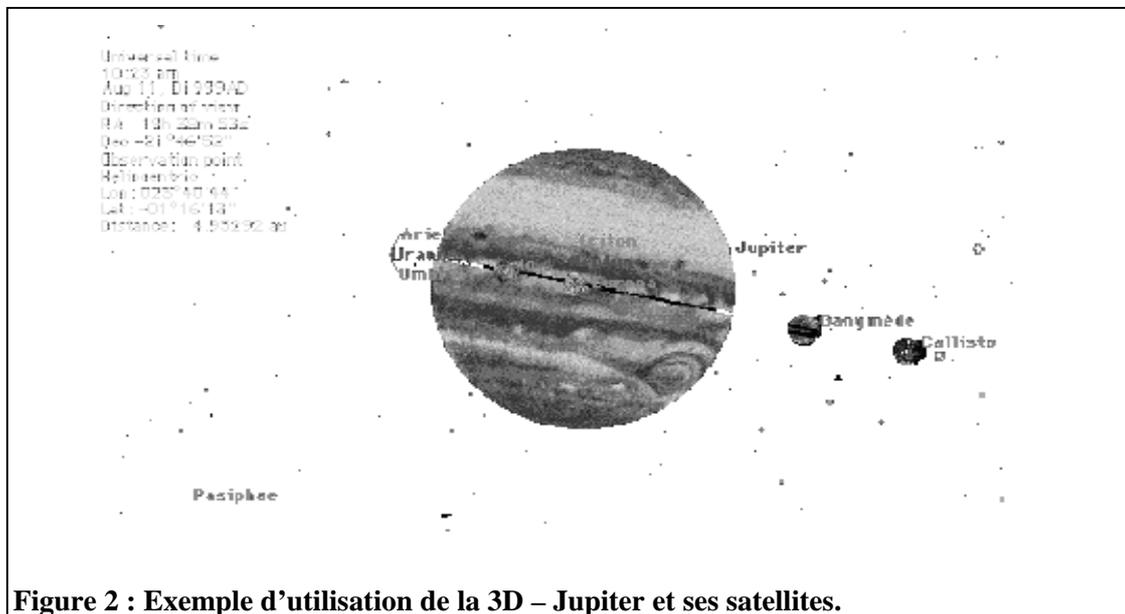


Figure 2 : Exemple d'utilisation de la 3D – Jupiter et ses satellites.

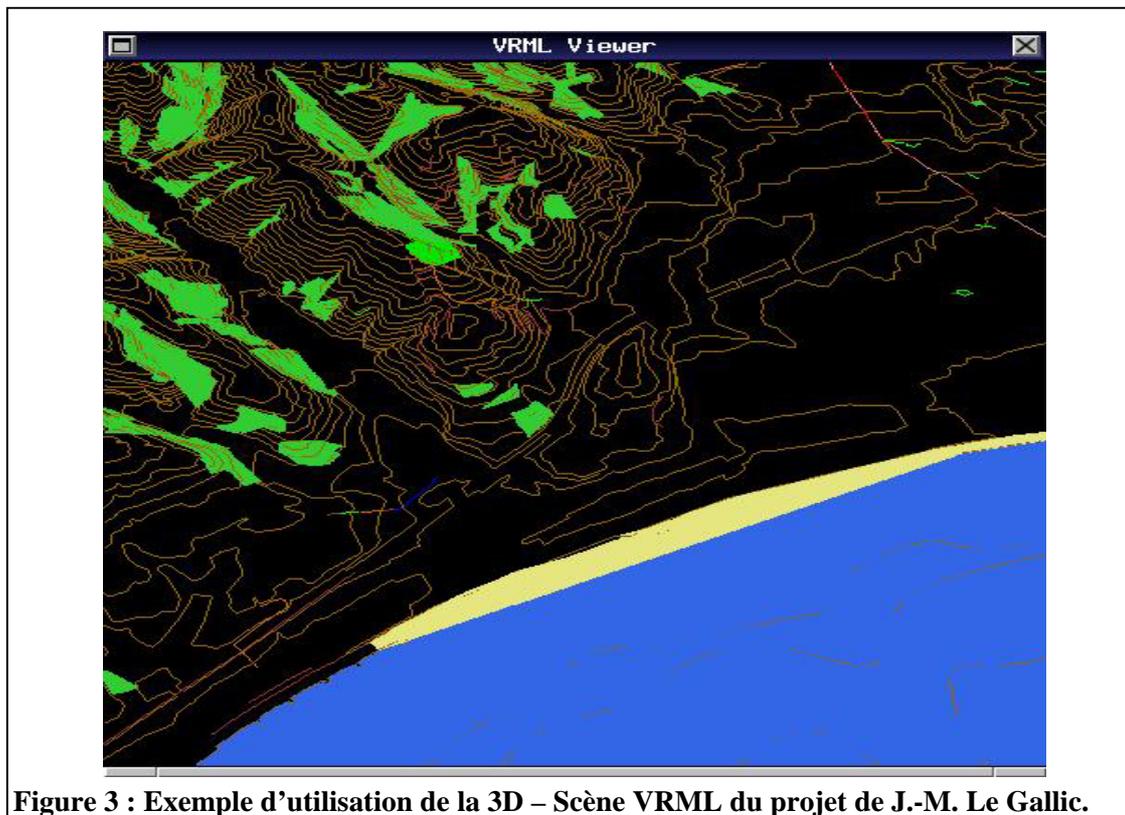


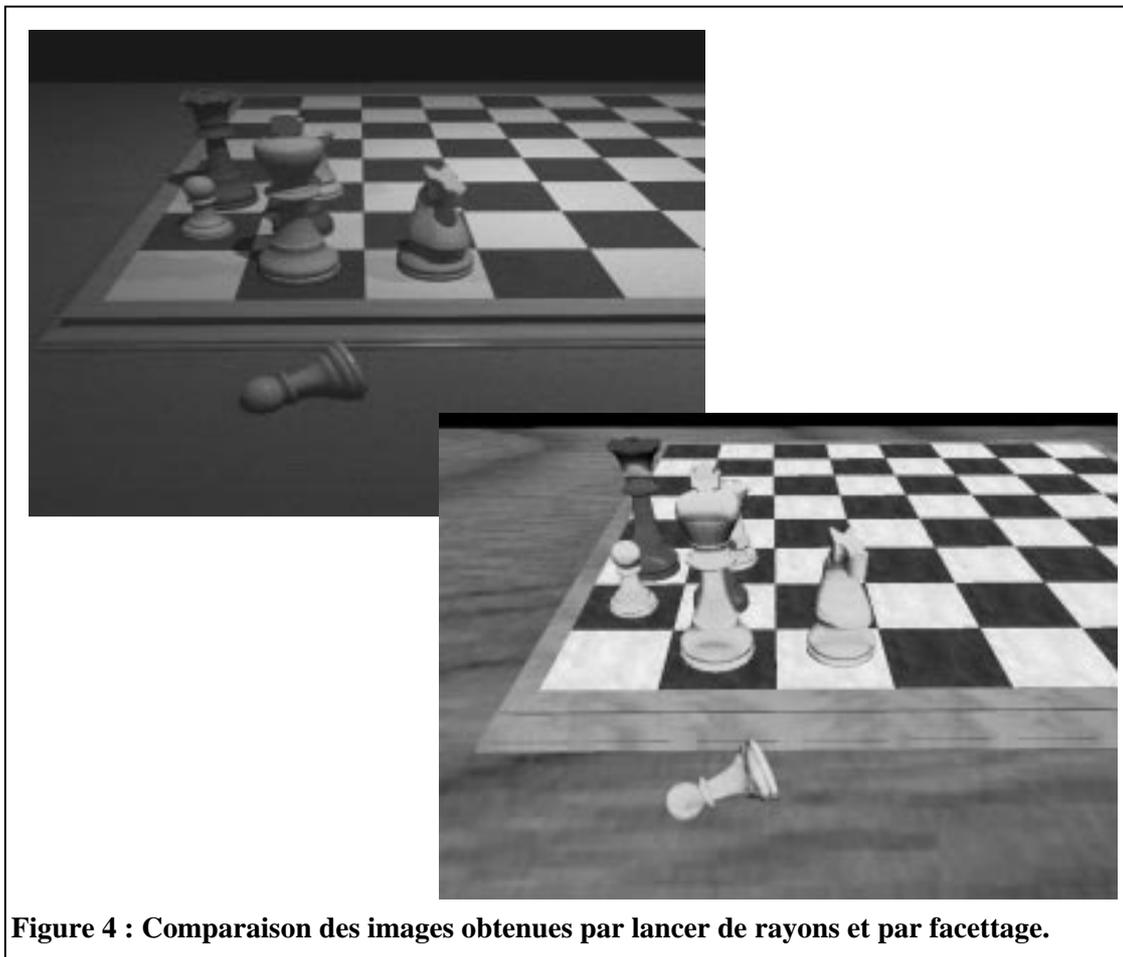
Figure 3 : Exemple d'utilisation de la 3D – Scène VRML du projet de J.-M. Le Gallic.

La 3D temps réel commence également à être employée pour permettre aux utilisateurs d'interagir avec de nombreuses données présentées de façon organisée [ROB93a][CAR96]. Les interfaces de type WIMP (*Windows, Icons, Menus, Pointers*), gérant des fenêtres recouvrantes

(*overlapping windows*), montrent leurs limites du point de vue ergonomique. Pour la visualisation par exemple, il est difficile de soumettre un nombre important de données avec une interface 2D. Nous pensons aussi que le matériel actuel permet d'envisager dans un avenir proche la réalisation d'interfaces graphiques en trois dimensions pour les systèmes d'exploitation [ROB2000]. Cependant, cela suppose auparavant de déterminer les possibilités et les besoins de telles interfaces et de spécifier les interactions possibles.

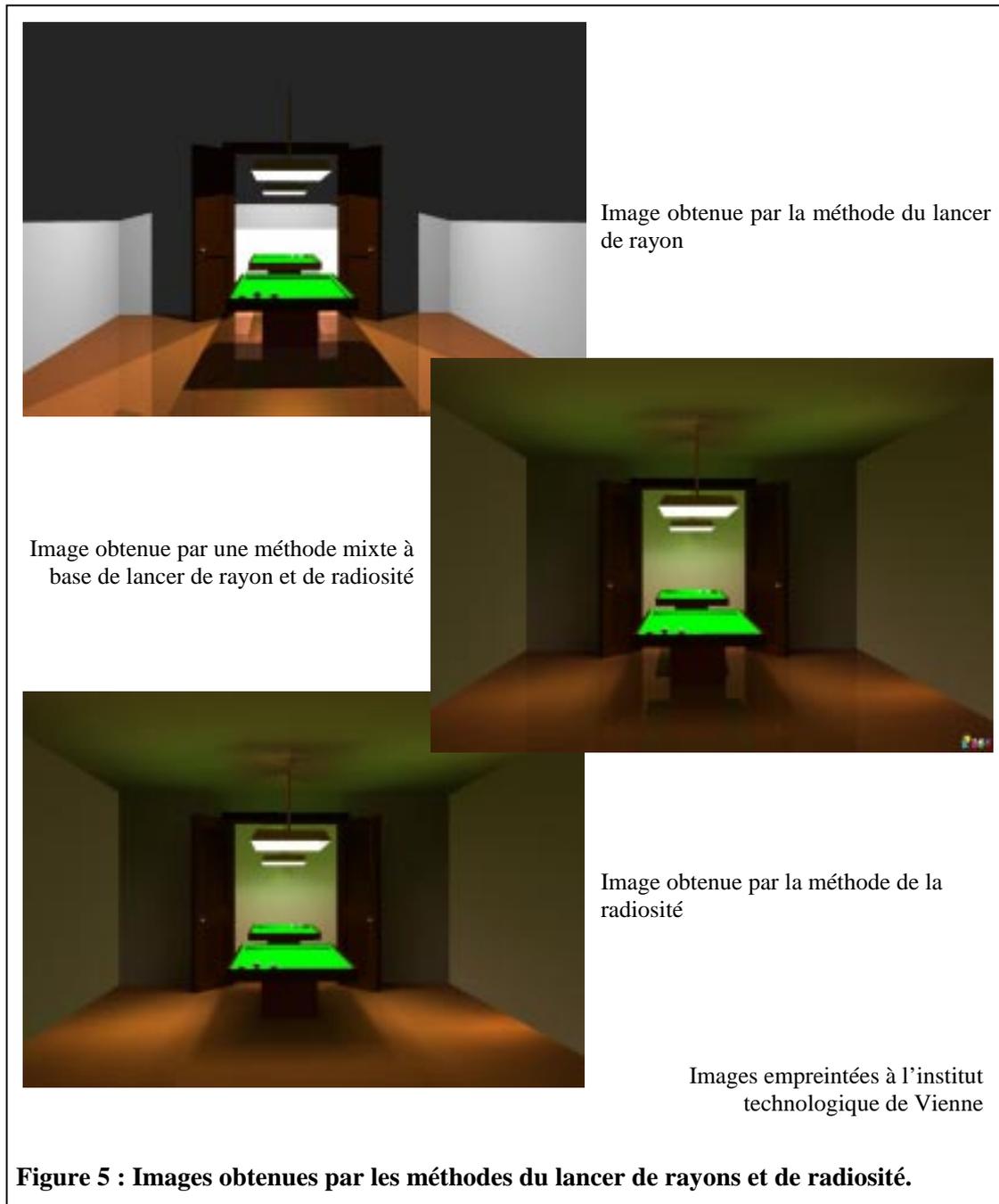
1.1.2 Deux méthodes de rendu 3D : différences et utilités

Il existe différentes manières de créer des images de synthèse à partir de la spécification d'une scène en 3 dimensions. Chronologiquement, est d'abord apparu un modèle à bases de polygones et plus précisément de triangles. Un second modèle, présenté en 1976 par James Blinn, est le lancer de rayons. Enfin, les méthodes utilisant la radiosité et celles dites mixtes sont apparues plus récemment. Bien que la suite de ce rapport s'intéresse à la première méthode qui comme nous le verrons permet une interaction en temps réel, nous nous proposons cependant d'étudier ici succinctement les différences entre ces approches et leurs avantages et inconvénients qui ont mené à notre choix.



Nous verrons en détail comment sont obtenues les images par *raytracing* (lancer de rayons) dont les méthodes cherchent à reproduire le trajet des rayons de lumière depuis l'œil de l'observateur jusqu'aux objets de la scène, éclairés par des sources lumineuses. Nous parlerons plus particulièrement de *raytracing* arrière (*backward raytracing*) qui consiste à suivre les rayons depuis la caméra jusqu'aux objets. Il existe également des méthodes de *Forward Raytracing* où le trajet est fait à l'envers depuis les différents points de l'écran vers la caméra (comme dans la réalité). La méthode dite de radiosité, quant à elle, calcule la luminosité et non la couleur de chacun des points. Ces calculs de

luminosité sont basés, contrairement à ceux du lancer de rayons, sur des modèles physiques rigoureux de distribution d'énergie dans un environnement. Ces méthodes s'approchent de celles employées dans la synthèse sonore basée sur une modélisation physique des ondes (par opposition au modèle psycho-acoustique). La technique consiste en un lourd calcul d'éclairage qui peut durer plusieurs jours selon la complexité. Une fois le modèle éclairé, le calcul d'une image est relativement rapide. Enfin, comme nous pouvons le voir dans la figure suivante, il existe des méthodes mixtes utilisant le *raytracing* et la radiosité.



Pour les deux méthodes de rendu 3D que nous décrivons ici (*raytracing* et rendu par facettes), les premières étapes de la chaîne de production menant de la description de la scène à l'image finale affichée à l'écran sont les mêmes. Cette chaîne débute par la partie conception de l'image. Pour un rendu sur écran, seulement cette étape est véritablement 3D puisque le créateur de la scène décrit le positionnement des objets dans l'espace. Chacun des objets composants la scène est alors habillé par

une couleur (dans le cas le plus simple) ou d'une texture qui représente à l'utilisateur le matériel de l'objet, voire également sa rugosité et son relief. Puis, sont définies les différentes lumières éclairant la scène et les caméras desquelles sont prises les vues.

La deuxième étape vise à transformer la description de la scène en 3 dimensions en une image en 2 dimensions en fonction des différents éléments de la scène et du point de vue duquel elle est regardée. C'est pour cette étape que le rendu diffère entre les deux méthodes.

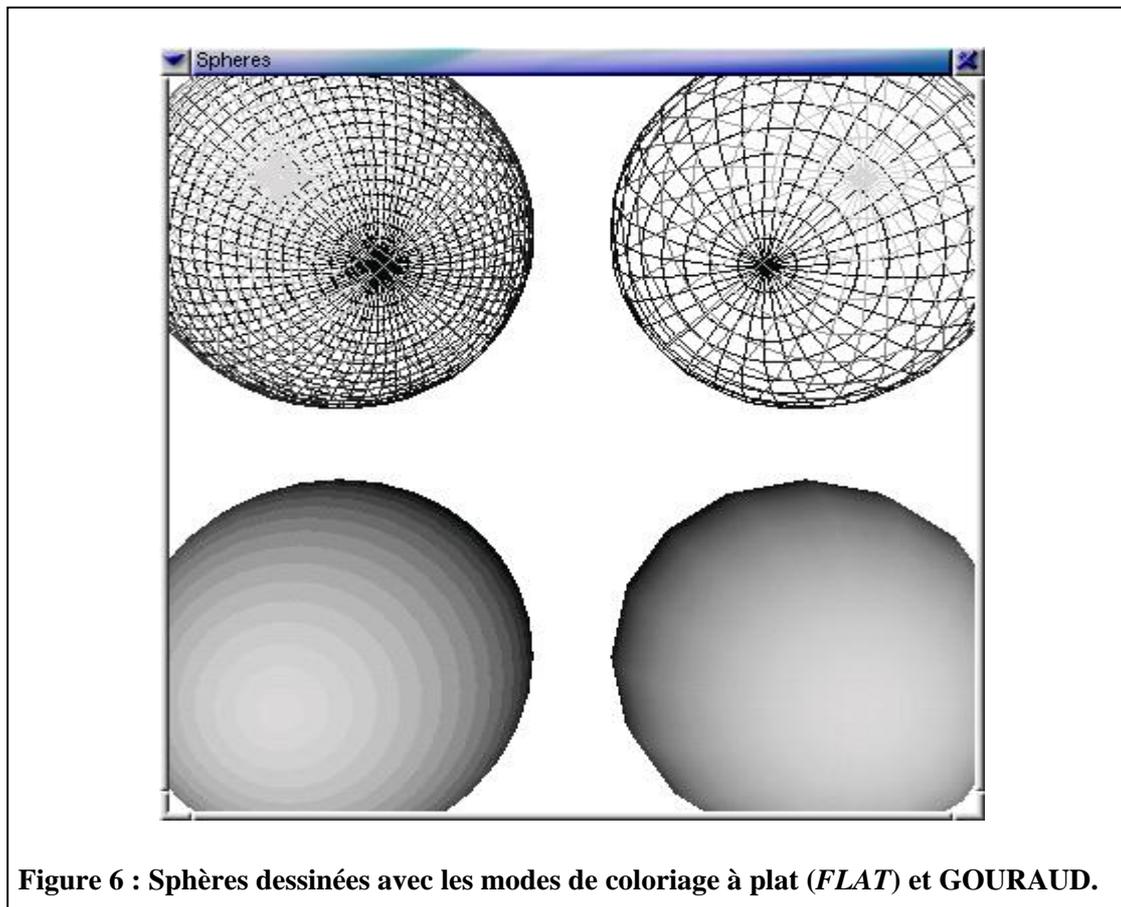


Figure 6 : Sphères dessinées avec les modes de coloriage à plat (FLAT) et GOURAUD.

Pour le modèle à base de triangles, il convient d'abord de transformer les différents éléments en ensembles de facettes. Tous les objets de la scène, y compris ceux qui sont courbes seront donc constitués de triangles qui sont le point d'entrée du pipeline graphique dont nous parlerons plus loin. Afin que l'aspect courbe d'un objet soit respecté, deux solutions sont envisageables : décomposer l'objet en un assez grand nombre de facettes afin d'obtenir un ensemble convainquant ou utiliser une méthode de lissage par apparence telle que celle de Gouraud ou de Phong (cette dernière méthode n'est cependant pas encore implémentée dans la plupart des API 3D). La Figure 6 illustre ces deux méthodes. Les sphères, en mode filaire, montrent le nombre de facettes utilisées. Les deux sphères du bas, quant à elles, nous donnent le résultat obtenu. Les sphères à gauche sont coloriées avec un modèle à plat (*flat*). Nous constatons que le nombre de facettes est élevé pour obtenir une apparence correcte. Il en découle une chute de performances car chacun des triangles doit subir différentes transformations. Donc, plus il y a de triangles, plus l'ensemble de ces calculs de rendu est long. La deuxième méthode, à droite, utilise moins de facettes et l'apparence finale obtenue par le modèle de coloriage de Gouraud est très satisfaisante.

Le pipeline graphique de la méthode par facetage effectue de nombreuses opérations (pour les transformations, la détermination des faces visibles, etc.) et les modèles de lumière, de coloriage ou de texturage sont souvent plus rudimentaires que pour le lancer de rayons. En effet, à l'heure actuelle ce mode de rendu par triangles est utilisé pour permettre une interaction temps réel. Ainsi, la nécessité de

pouvoir calculer et afficher une image rapidement après modification d'un paramètre de la scène interdit l'utilisation de méthodes trop coûteuses en temps. Cependant, grâce à des fréquences rapides d'affichage des images successives, l'œil humain tolère les défauts générés par cette méthode.

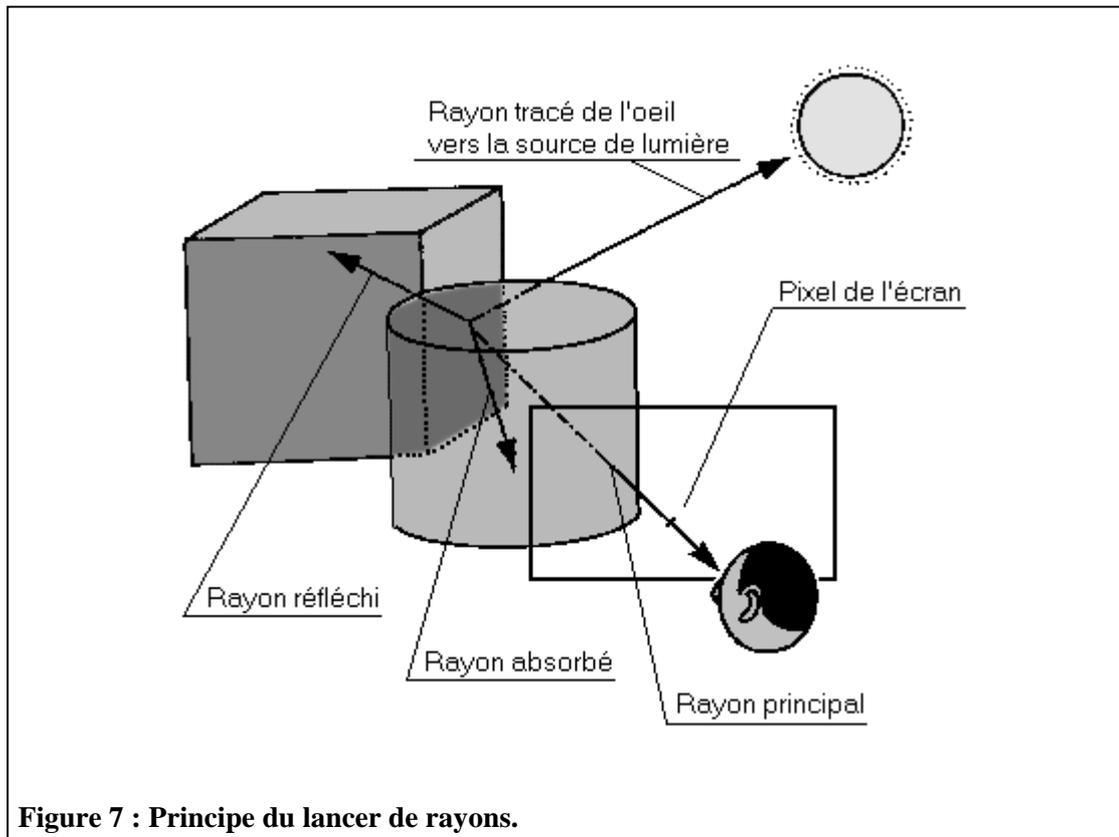


Figure 7 : Principe du lancer de rayons.

De l'autre côté, les méthodes de lancer de rayons ne permettent pas de réaliser des images en un temps suffisamment court pour permettre une interactivité avec l'utilisateur. Cependant, la qualité des images obtenues, fixes ou animées (par calculs d'images successives), est parfois stupéfiante. L'avenir de la 3D en temps réel réside probablement dans l'utilisation de cette méthode avec des ordinateurs bien plus puissants, embarquant un matériel spécialisé, capables de calculer une image en un temps inférieur au 25^{ème} de seconde. Cette fréquence d'échantillonnage étant connue comme celle à partir de laquelle l'œil humain ne perçoit plus les changements d'images.

L'algorithme de rendu par la méthode du lancer de rayons est beaucoup plus simple que celui de la méthode par facettes. On appelle cette méthode ainsi car son principe est de former un rayon partant de l'œil de l'observateur (considéré comme une caméra ponctuelle) et passant par un point précis de l'écran que l'on place virtuellement entre l'observateur et la scène. La couleur du pixel de l'écran par lequel passe le rayon sera alors de la couleur du matériel du premier objet frappé par le rayon. La détermination des objets traversés par le rayon se fait par des calculs d'intersection peu complexes mais assez coûteux, ce qui explique la lenteur de ce procédé. Pour obtenir l'image complète de la scène vue depuis un point précis, il suffit de « balayer » tous les pixels de l'écran et de faire le calcul précédent pour chacun des rayons formés. La Figure 7 schématise le parcours d'un rayon depuis la caméra jusqu'à la scène. Les deux rayons réfléchis et absorbés servent à calculer respectivement les réflexions et les objets vus en transparence. Le rayon vers la source lumineuse sert, quant à lui, à calculer les ombres influant sur la couleur de l'objet.

Le temps de rendu n'ayant pas une grande importance pour un logiciel de lancer de rayons, la qualité peut donc être augmentée même si cela s'accompagne d'un temps de calcul plus important. Ainsi, des modèles de lumières, d'ombrages, de réflexions, de matériaux beaucoup plus complexes pourront être utilisés. Citons parmi les améliorations possibles du lanceur de rayons de base que nous

avons vu :

- le coloriage d'un objet en fonction des lumières et les ombres sont obtenus en « tirant » des rayons secondaires depuis le point d'intersection (entre l'objet frappé et le premier rayon) jusqu'aux sources de lumières ponctuelles. La couleur de notre point à afficher sera alors calculée d'après un modèle de coloriage en fonction de la lumière ambiante et des sources lumineuses non cachées par un autre objet de la scène. Dans ces modèles, si tous les rayons secondaires ont une intersection avec un autre objet de la scène (c'est-à-dire que les sources lumineuses sont cachées par d'autres objets et donc qu'elles n'éclairent pas le point) alors la couleur sera uniquement calculée en fonction de la lumière ambiante, créant ainsi une impression d'ombre. Pour obtenir des zones de pénombre il sera nécessaire d'utiliser des sources de lumière non ponctuelles.
- les réflexions des objets réfléchissants et la réfraction des objets transparents sont obtenues par récursivité de l'algorithme de base. Le premier rayon frappant un objet de la scène en un point il suffit de créer deux rayons, l'un réfléchi, l'autre absorbé par l'objet et de continuer les calculs avec les nouveaux rayons de cette façon jusqu'à une profondeur de récurrence donnée. La direction des deux rayons supplémentaires après impact est calculée d'après la loi de Descartes. Quant aux intensités des nouveaux rayons, ils sont calculés en fonction de leur longueur.
- L'amélioration des temps nécessaires pour le calcul des intersections entre objets et rayons par l'utilisation de volumes englobants ou par les « octrees » qui permettent de définir rapidement quels objets sont dans un volume particulier de l'espace.

Cette manière d'afficher les scènes 3D est la plus réaliste des deux méthodes puisqu'elle s'inspire de lois physiques. On peut considérer le lancer de rayons comme la méthode qualitative alors que la méthode par facettage est quantitative. Bien que les calculs soient plus longs, ils sont néanmoins plus simples à mettre en œuvre (par exemple, il n'y a pas de matrices de transformations ou de projection à gérer). Avec une vitesse de calcul permettant une interactivité entre l'utilisateur et la scène, cette méthode semblerait idéale, par exemple, pour définir quel objet est manipulé par le pointeur de l'utilisateur puisqu'il suffirait de mémoriser le premier objet frappé par le rayon passant par la coordonnée à l'écran du pointeur. Dans notre prototype d'environnement 3D [TOP2000], nous avons utilisé ce principe pour déterminer quelle fenêtre est sélectionnée.

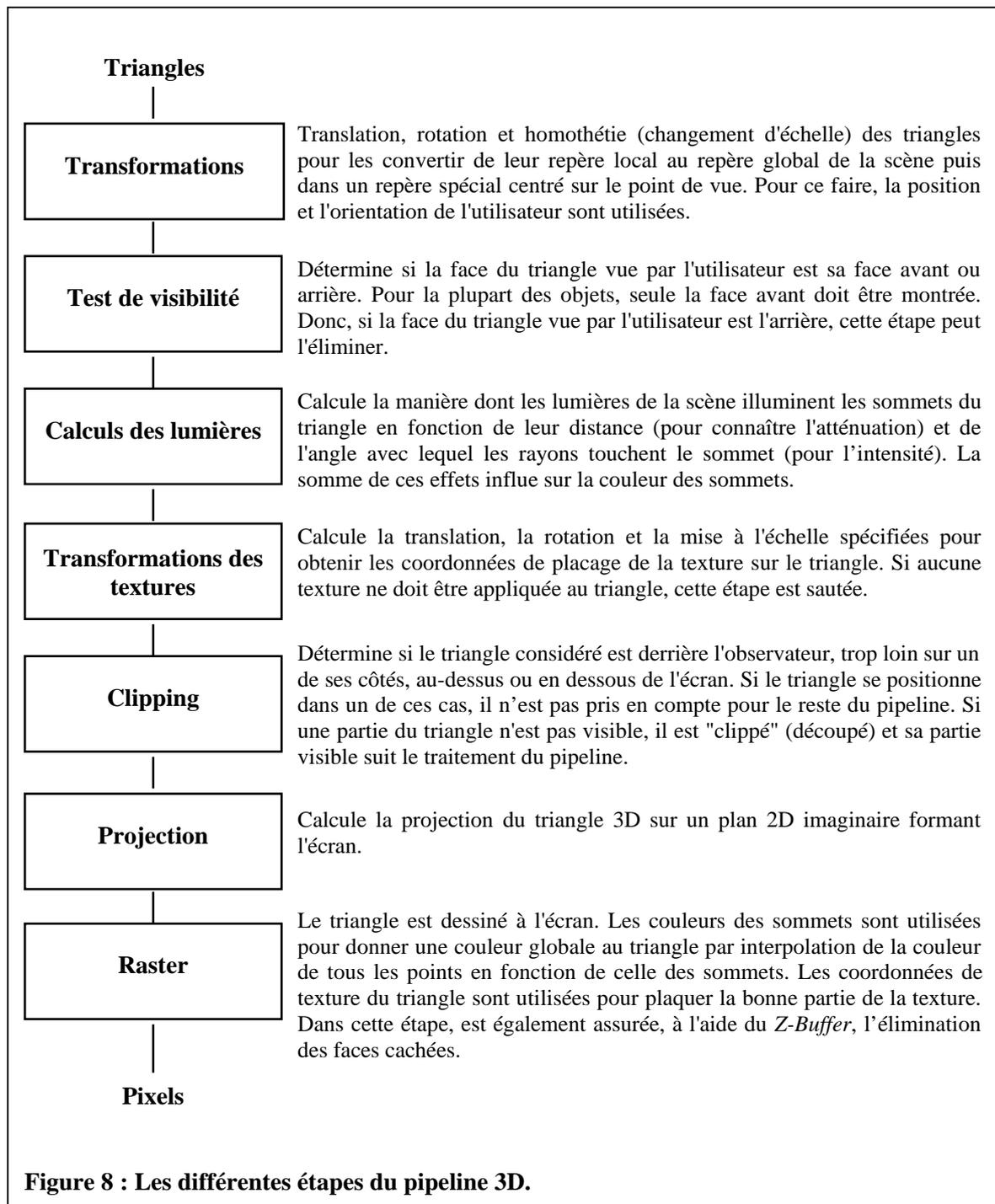
1.1.3 Les bases de la 3D temps réel

Avant de présenter les différentes manières de créer et d'afficher une scène 3D, nous souhaitons rappeler les fondements de la 3D temps-réel. Ils sont rassemblés dans la chaîne de production d'une image ou pipeline graphique (ou encore pipeline 3D) dont nous allons parler très souvent. Il convient donc de le présenter, ainsi que les différentes étapes de celui-ci, avant que nous le référencions.

Le pipeline 3D est l'ensemble des opérations nécessaires pour afficher une scène constituée d'objets 3D regardés depuis une position et avec une orientation données. A chaque fois que l'état de la scène change (c'est-à-dire à chaque mouvement de la caméra, à chaque déplacement d'un objet, etc.), elle doit être redessinée. Pour cela, la description en mémoire vive des objets de la scène doit être traduite en points 2D à l'écran. Ce processus est le travail du pipeline 3D (plus communément appelé moteur 3D). Quel que soit le moteur 3D utilisé, *plugin* ou API, un pipeline 3D effectue les différents calculs nécessaires à l'affichage d'une scène 3D sur un écran.

Les éléments d'entrée de ce pipeline sont des triangles plus pratiques que les quadrilatères ou autres polygones pour les calculs car ils ne peuvent être ni vrillés (dont les sommets ne sont pas coplanaires) ni concaves – donc plus faciles à tracer. Cependant, afin de nous autoriser à créer des polygones complexes, la plupart des moteurs 3D effectuent une triangulation des différentes faces

avant de les envoyer au pipeline graphique. Ainsi, ils nous permettent de réfléchir en terme de polygones et non uniquement en terme de triangles mais évitent néanmoins le problème de remplissage des faces concaves. La Figure 8 illustre les différentes étapes de calculs menant à l’affichage d’un triangle.



Les pipelines graphiques, contrairement à une chaîne de synthèse sonore fonctionnent généralement sur le mode "AFAP" (*As Fast As Possible*). Ils calculent la scène le plus rapidement possible sans prendre en compte le temps écoulé entre l'affichage de deux trames successives. Cette différence avec le son s'explique par la plus grande tolérance de l'œil aux changements d'échantillonnage. En effet, si le nombre d'images par secondes tombe par exemple de 100 à 30, l'œil humain ne voit pas la différence puisqu'il ne peut en percevoir qu'au maximum 25. Si la fréquence des

images tombe en dessous de 25 images par seconde, l'utilisateur percevra le ralentissement ; mais cela n'engendrera pas de gêne physique mais plutôt une gêne au niveau de l'interaction. Avec un son, non seulement le phénomène sera perçu mais il sera également très désagréable puisque la qualité d'un son dépend de sa fréquence d'échantillonnage.

Avec une scène de petite taille, ou optimisée, le pipeline calcule le rendu efficacement et effectue un rafraîchissement rapide. On obtient par conséquent une animation fluide. Au contraire, avec une scène comportant de nombreux objets ou pour une scène non optimisée, le pipeline risque de bloquer dans une, voire plusieurs étapes. Il existe cependant des pipelines graphiques garantissant 25 images par seconde quelque soit la scène ; ceci en arrêtant leurs traitements au-delà du 25^{ème} de seconde si cela est nécessaire. D'autres techniques, moins radicales, existent également.

Nous expliquons ici succinctement les calculs effectués à chacune des étapes du pipeline graphique afin que le lecteur non habitué aux techniques sous-jacentes à la 3D puisse comprendre le coût du calcul d'une image. Cependant, pour obtenir de plus amples informations sur le fonctionnement d'un moteur 3D, on pourra se reporter aux ouvrages de référence [BRE88], [FOL90] et [WAT93] ou pour un tour d'horizon moins technique [COU95].

Avant tout, il convient de parler du mode de représentation utilisé pour placer un point dans l'espace. Les calculs nécessaires pour la transformation et la projection d'un point dépendent de cette représentation. Le système de coordonnées que l'on utilise pour la 3D est cartésien. Selon les outils utilisés, ce référentiel sera « main droite » (comme avec OpenGL ou VRML) ou « main gauche » (comme avec Direct3D). Comme nous pouvons le voir dans la Figure 9, que le système soit direct ou indirect, pour chacun d'eux les axes X et Y sont les mêmes (X de gauche à droite et Y de bas en haut de l'écran). La façon la plus naturelle de se représenter un système de coordonnées cartésien est d'utiliser ses deux mains (d'où les noms « main droite » et « main gauche ») comme cela est schématisé dans la figure. Pour chacune des mains, il suffit de faire pointer le pouce (donnant l'axe X) vers la droite, l'index (l'axe Y) vers le haut. Le majeur, perpendiculaire aux deux doigts précédents (et à la paume de la main) nous donne automatiquement la direction positive de l'axe Z.

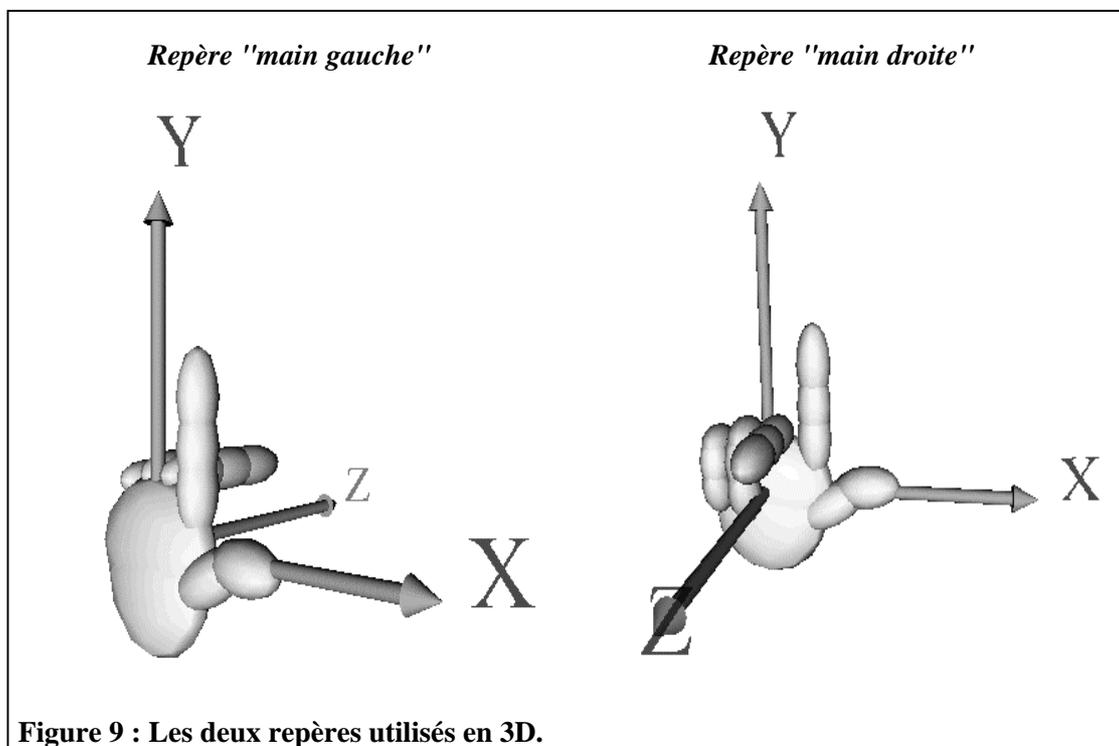


Figure 9 : Les deux repères utilisés en 3D.

La transformation des coordonnées d'un système de coordonnées à l'autre se fera en prenant

l'opposée de la coordonnées Z. Quant à la différence d'interprétation que l'on peut donner à cette opposition de l'axe Z, pour le repère « main gauche » on parlera de profondeur pour la coordonnées Z d'un point alors que, pour le repère « main droite », on y verra plus volontiers une altitude.

Toute primitive graphique complexe est d'abord transformée en un ensemble de triangles. L'apparence obtenue par approximation en un nombre plus ou moins de triangles permet d'avoir soit un rendu plus rapide soit un rendu de meilleure qualité. Chacun des triangles subit alors les différentes étapes du pipeline graphique dont nous donnons pour les plus importantes le détail des calculs :

- Transformations : opérations matricielles donnant le transformé d'un point par une ou plusieurs des matrices suivantes :

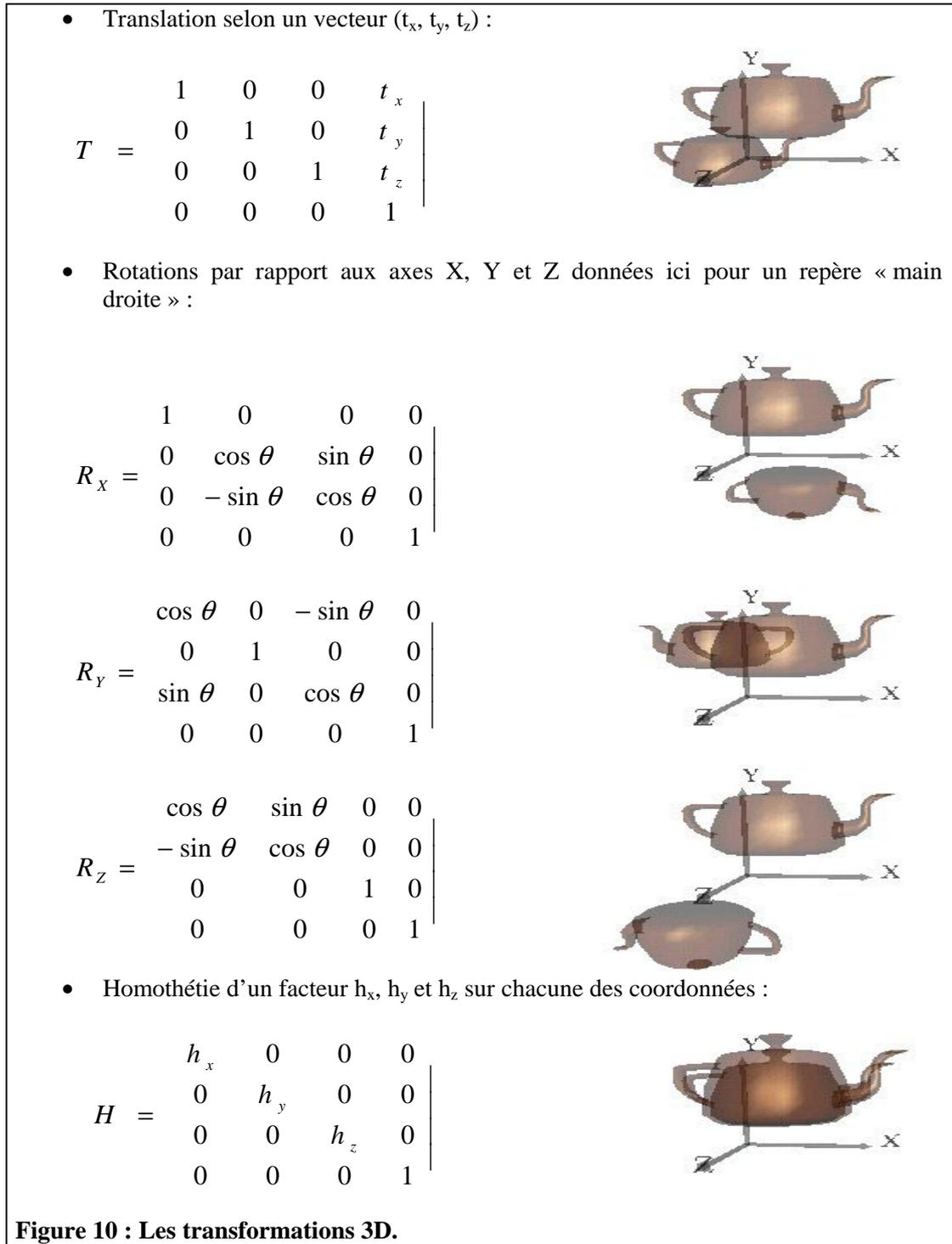
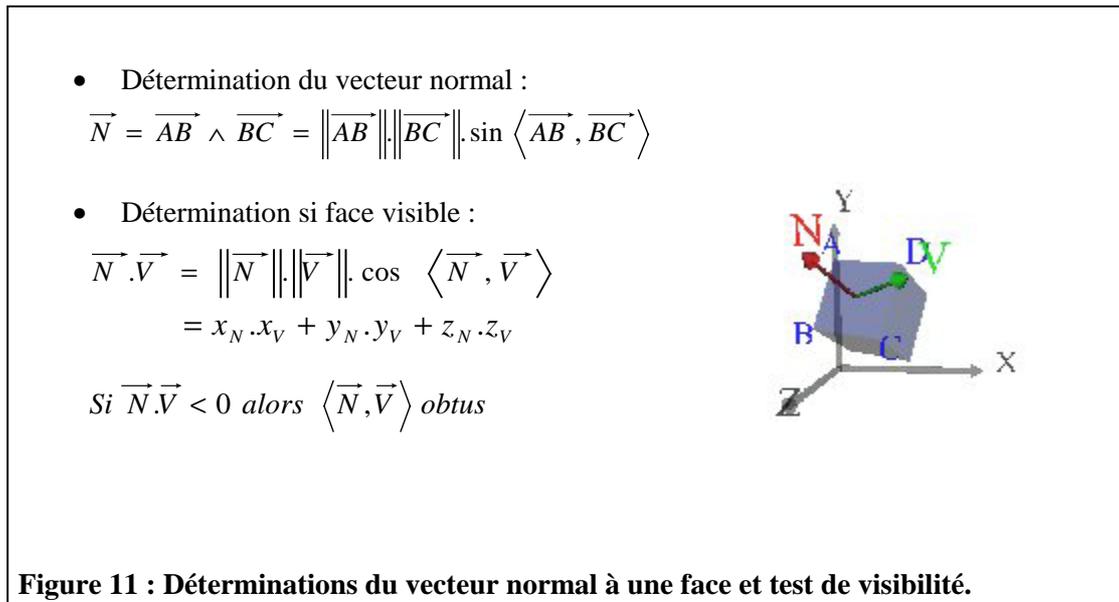


Figure 10 : Les transformations 3D.

- Test de visibilité : le calcul de visibilité d'un triangle se fait grâce à son vecteur normal donnant sa face avant. Si l'angle formé entre le vecteur normal et le vecteur de vision (allant de la face à l'œil) est aigu alors la face est visible sinon elle est invisible. Ce calcul se fait en utilisant le produit scalaire. Si celui-ci est négatif alors cela signifie que le cosinus de l'angle entre les deux vecteurs est négatif et donc que l'angle est obtus. Si nécessaire, le vecteur normal est calculé en faisant le produit vectoriel entre le vecteur formé par les deux premiers points et celui formé par les deuxième et troisième points du polygone. Par convention, tous les moteurs 3D attendent des polygones dont les sommets sont donnés dans l'ordre trigonométrique comme dans la figure suivante. Par conséquent le vecteur normal résultant du produit vectoriel est bien extérieur à la face.



- Calculs des lumières : utilisation de lois physiques simplifiées (pour des raisons de performance) comme la loi de Lambert donnant l'intensité de la lumière réfléchiée en fonction du matériel d'un objet. La somme des intensités des différentes lumières frappant la face donne un coefficient global d'éclaircissement de la face. Cette valeur est alors utilisée lors du coloriage. La loi de Descartes donnant les angles de réflexion et de réfraction en fonction du matériel de l'objet n'est pas utilisée comme dans la technique du lancé de rayon car les lumières réfléchies ne sont pas prises en compte.
- Transformations des textures : cette étape permet de transformer les textures avant qu'elles ne soient appliquées au triangle (dans l'étape de *Rasterization*). Ce sont des transformations 2D sur les images qui sont un cas simplifié des transformations 3D précédentes.
- *Clipping* : élimination des triangles ne faisant pas partie du volume de vue et découpage de ceux en partie visibles selon leurs intersections avec le volume de vue.
- Projection : la projection d'une scène 3D sur l'écran 2D comprend en fait deux étapes différentes : la conversion dans le repère de l'observateur et la projection sur l'écran (plan de projection). Dans la Figure 12 sont données les opérations pour ces deux étapes et un schéma explicatif. Dans le schéma ainsi que dans les calculs nous utilisons un repère « main droite » dont l'origine est O. La position de l'observateur est C (pour caméra) et il regarde le long de l'axe Z négatif (vers O). Son repère local est donc « main gauche » comme nous pouvons également le constater sur le schéma. Le plan de projection est perpendiculaire à OC à une distance D de l'observateur.

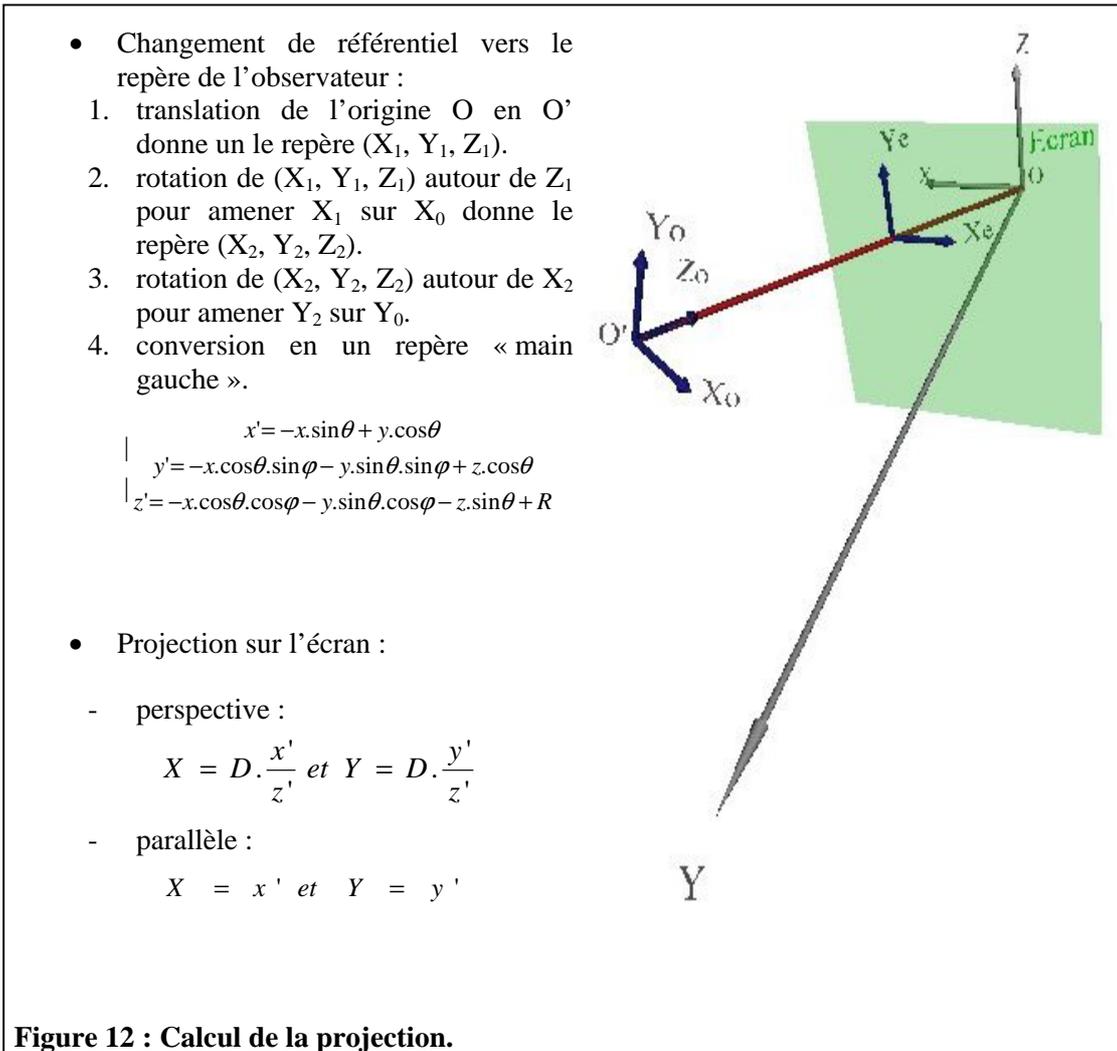


Figure 12 : Calcul de la projection.

- *Rasterization* : l'étape transformant les formes géométriques 3D en des *pixels* sur l'écran.

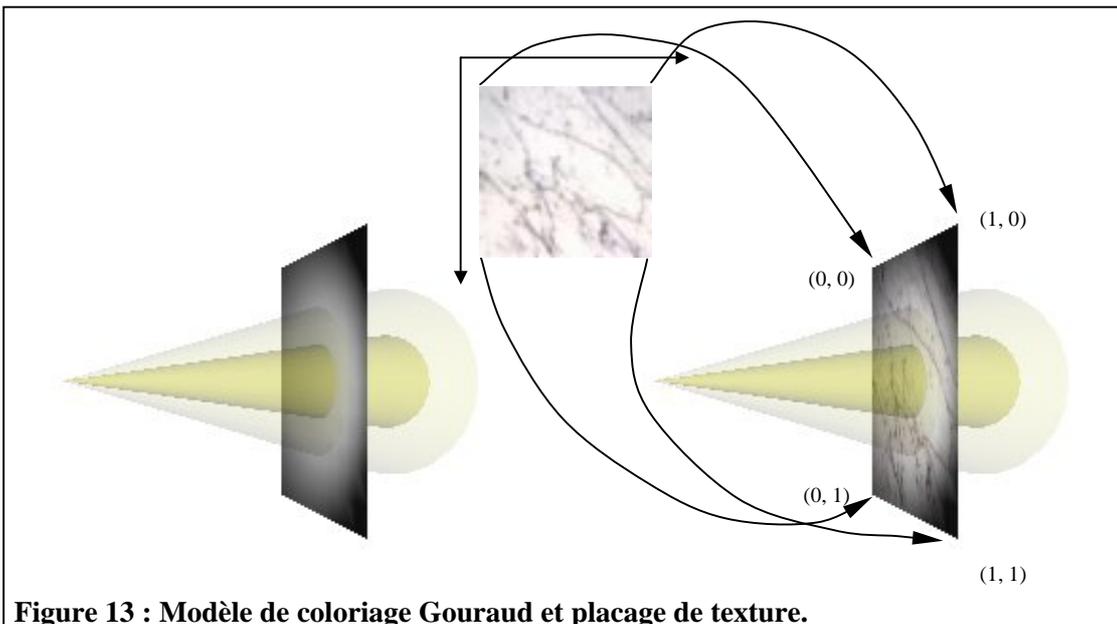


Figure 13 : Modèle de coloriage Gouraud et placage de texture.

Sur chacun des points visibles du triangle, déterminés par l'algorithme du *Z-Buffer*, est appliqué un modèle de coloriage. Il peut s'agir simplement d'une couleur unie calculée en fonction de la couleur de la face et de la direction de la lumière par rapport à la normale à la face (modèle de coloriage *Flat*). Avec un coloriage de type *Gouraud* (comme sur la Figure 14), la couleur d'un pixel est interpolée en fonction des distances le séparant des sommets du triangle dont il fait partie. Lorsqu'une texture doit être plaquée sur le triangle, le pixel prend la couleur du point de l'image lui correspondant. Les modèles à base de couleur peuvent être combinés avec le placage d'une ou de plusieurs textures.

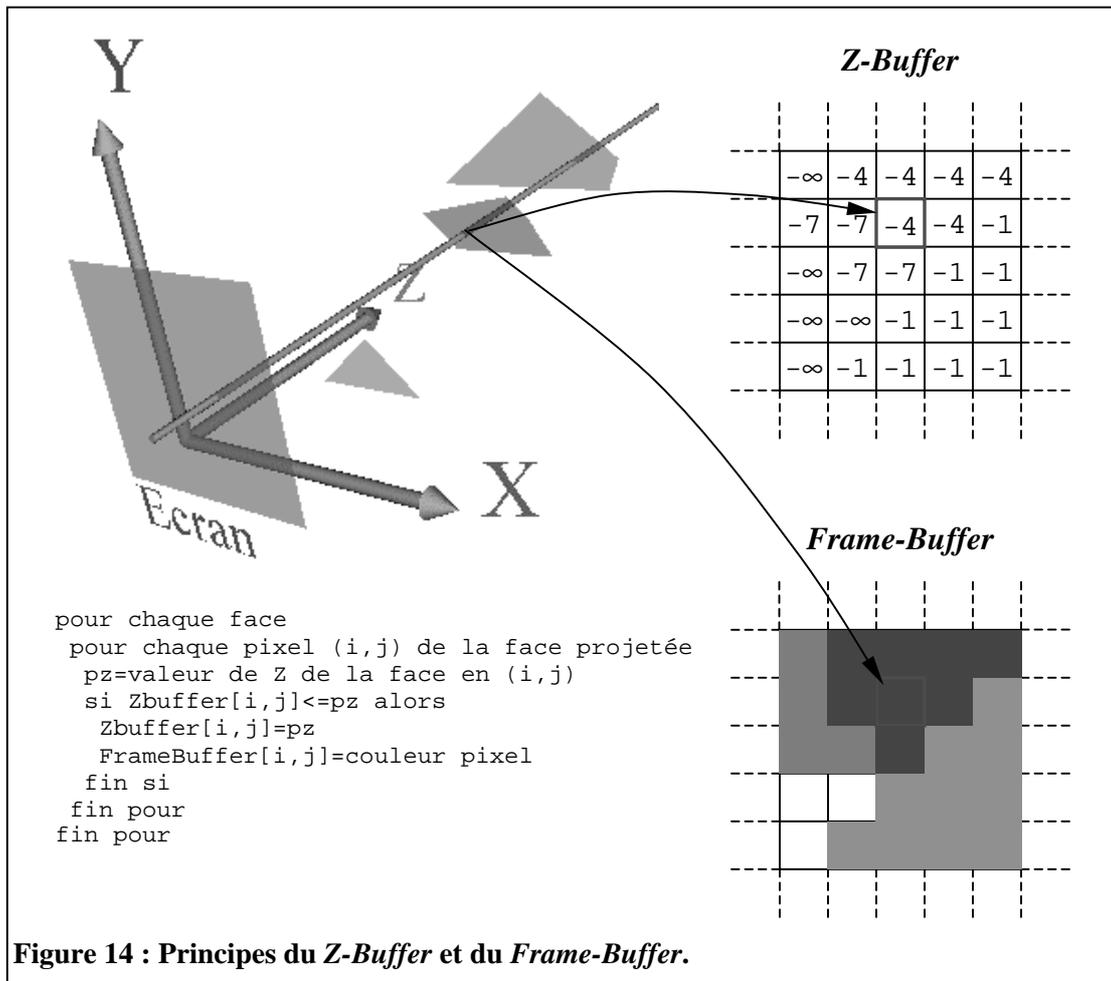


Figure 14 : Principes du Z-Buffer et du Frame-Buffer.

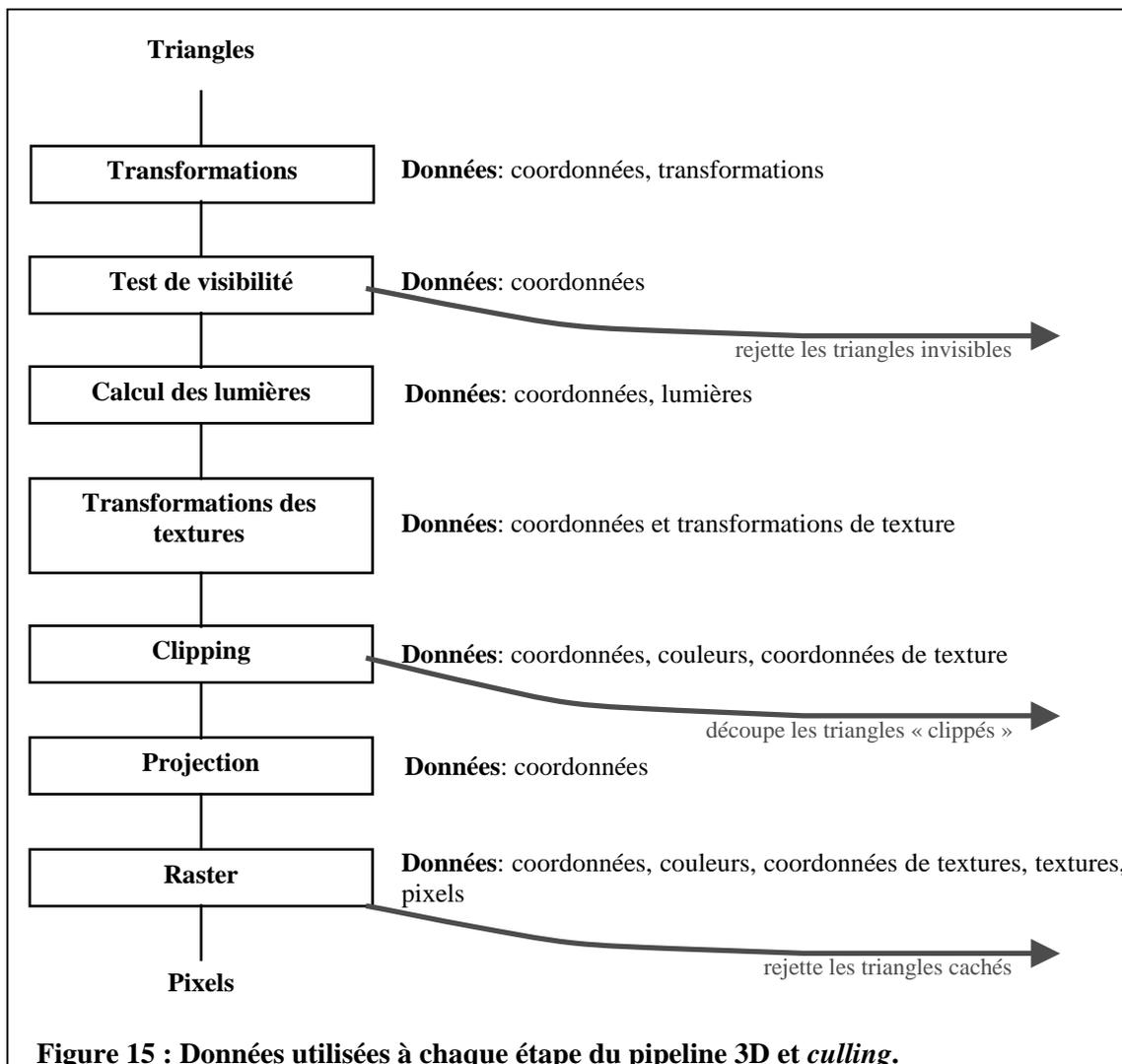
Deux tableaux de même taille sont utilisés lors de l'étape de *rasterization*. Le premier dont nous avons déjà parlé est le *Z-Buffer* et il sert à résoudre le problème des faces cachées. Ce tableau stocke la profondeur (coordonnée Z) d'une portion d'une face la plus proche du point de vue. Il sert de table des Z minimums pour chaque point projeté. Pour chaque face, on compare la valeur Z de chaque point projeté avec la valeur stockée dans le *Z-Buffer* dans la case correspondant à l'endroit où est projeté ce point. Si la valeur du *Z-Buffer* est inférieure ou égale à cette coordonnée Z alors le point considéré est plus proche du point de vue que le point précédemment stocké. La coordonnée Z de ce point est alors stockée dans le *Z-Buffer* pour la suite des comparaisons.

Dans le même temps, la couleur du point (s'il est plus proche du point de vue que le point sauvegardé précédemment) est stockée dans le *Frame-Buffer*. Ainsi, la couleur des pixels dans ce tableau correspond toujours à celle des points des faces les plus proches du point de vue et dont la profondeur est stockée dans le *Z-Buffer*.

1.1.4 Optimisation possible du pipeline 3D

Avoir un taux de rafraîchissement supérieur ou égal aux 25 images par seconde (le maximum que l'œil humain puisse discerner) est d'une extrême importance pour percevoir une navigation de manière fluide. Lorsque le nombre d'images par seconde n'est pas suffisant, il existe quelques techniques pour réduire le temps d'affichage des trames. Ces techniques, particulièrement bien décrites dans le cours présenté à la conférence SIGGRAPH'2000 [SIG2000], visent à optimiser soit la scène pour que son contenu soit affiché plus rapidement, soit le code de l'application afin qu'il effectue des optimisations sur toutes les scènes.

Chaque étape du pipeline 3D peut être caractérisée par le type de données sur lequel elle passe le plus de temps à faire ses traitements. Par exemple, l'étape de transformation, qui calcule la nouvelle position de chacun des sommets des triangles, manipule les coordonnées des sommets des triangles ainsi que les données de transformation que sont la translation, la rotation et l'homothétie. La Figure 15 reprend chacune des étapes du pipeline 3D avec les données qu'elle manipule. Lorsque l'on connaît les données manipulées par le pipeline, nous pouvons alors déterminer pourquoi elles peuvent le saturer.



Quand le moteur 3D dessine trop lentement la scène, cela signifie que le pipeline a trop de données à traiter pour pouvoir calculer une image en un temps acceptable. Pour optimiser une scène, il convient donc de trouver le type de données qui ralentit le pipeline 3D et cela passe par certaines manipulations. La suppression d'un certain nombre de ces données pourra accélérer le pipeline et donc

la fluidité de la navigation et la réponse aux interactions dans la scène.

Le rendu d'une scène dépend en premier lieu du nombre de pixels à dessiner à l'écran. C'est pourquoi, les fabricants de cartes graphiques, devant l'importance croissante que prend la 3D, ont inclus des fonctions câblées accélérant la génération de pixels et réduisant donc les problèmes d'entonnoir dus à ceux-ci. La cause du ralentissement est alors transférée sur un autre type de données, les coordonnées ou les textures.

Quoiqu'il en soit, la cause du ralentissement peut tout à fait se déplacer avec l'observateur. Considérons, par exemple, une scène représentant une planète de façon très détaillée. Si l'on se trouve très loin de cette planète, elle n'est plus qu'un point sur l'écran mais le nombre de coordonnées pour calculer cet unique point est élevé. La scène provoque alors un ralentissement à cause des coordonnées. Si l'utilisateur se rapproche, l'image à afficher comporte alors de plus en plus de points à calculer, ce qui peut provoquer un ralentissement à cause des points à calculer en plus des coordonnées à traiter. L'origine du problème s'est donc bien légèrement déplacée en même temps que l'utilisateur.

Pour optimiser une scène il faut rechercher les étapes du pipeline qui sont saturées. Nous appelons cette méthode la recherche des entonnoirs. En fonction des résultats obtenus, la scène pourra être modifiée en conséquence. Il est possible de trouver le type d'entonnoirs d'une scène 3D en effectuant les tests suivants :

- ① Tracer la même trame et déterminer le nombre obtenu en un temps donné. La scène doit être la même durant tout le test. Si un paramètre change (mouvement de la caméra, déplacement d'un objet) les données saturant le pipeline peuvent également changer. Le temps minimum pour un tel test devra être de deux secondes pour que l'erreur d'horloge n'excède pas 1%.

Réduire la taille de la fenêtre de navigation. En naviguant, si l'écran se rafraîchit plus vite alors le pipeline est saturé par les pixels, puisqu'il y en a moins à dessiner. Pour les scènes qui sont ralenties par une autre cause, la réduction de la taille de la surface de rendu n'aura pas ou peu d'effets.

Si la scène utilise des textures, les enlever. Si la fluidité est accrue lors de la navigation sans textures, c'est que le pipeline est saturé par les textures puisqu'il ne les gère plus dans l'étape de rasterisation.

Si les manipulations précédentes ne révèlent pas d'accélération, c'est que le pipeline est saturé par les coordonnées. La scène comporte probablement trop d'objets pour être dessinée rapidement.

La troisième étape de la recherche des entonnoirs dans le pipeline 3D consiste à supprimer les textures. Cette étape n'est envisageable que pour savoir si la scène est saturée par le nombre ou la taille trop importante des textures. En aucun cas cette étape ne sera utilisée pour opérer des optimisations agressives (c'est-à-dire supprimer les textures) puisqu'il est de première importance de garder les textures dans la plupart des cas. Alors, la seule optimisation possible sera de réduire la taille des textures ou, pour réduire le temps de rendu dans l'étape de rasterisation, d'utiliser des techniques de filtrages comme le *MipMapping*².

Par rapport aux optimisations précédentes qui visent à réduire le temps de rendu d'une scène précise, la modification du code effectuant le rendu a l'avantage d'être une solution globale. Les méthodes employées sont essentiellement celles dites de *culling*³ permettant de supprimer rapidement les triangles non visibles du pipeline graphique afin de l'accélérer. La plupart des API prennent en

² Voir le glossaire des termes 3D page 149.

³ *to cull* signifiant « sélectionner dans un group ».

compte les opérations de *culling* les plus efficaces dans la gestion de leur pipeline 3D. La Figure 15 schématise quels triangles sont éliminés par quelles étapes du pipeline.

Dans le cas d'un triangle non visible depuis le point d'observation ou « clippé », il est respectivement éliminé ou re-découpé durant la troisième et la cinquième étape : « Test de visibilité » et « clipping ». Dans ces deux cas, quel que soit l'habillage du triangle, l'élimination se fait au même point du pipeline et en un même nombre d'opérations. Dans le cas d'un triangle caché par un autre, le *Z-Buffer* l'élimine durant la dernière étape de *rasterization*.

Les API les plus récentes intègrent des techniques permettant de supprimer plus rapidement un ensemble de triangles en fonction du point de vue de l'utilisateur. Les boîtes ou les sphères englobantes permettent, par exemple, d'éliminer des branches entières du graphe de scène lorsqu'elles ne sont pas dans le champ de vision de l'utilisateur. La Figure 16 illustre ce principe. Sa partie supérieure montre ce que voit l'utilisateur depuis un point d'observation donné. L'arbre, dans la partie inférieure, schématise le graphe de la scène visualisée et l'intersection des sphères englobantes avec le cône de vue. Différents objets peuvent être rassemblés dans une seule sphère englobante s'ils sont proches dans l'espace. Si la sphère n'est pas dans le cône alors tous les triangles constituant tous les objets présents dans la sphère englobante peuvent être supprimés du pipeline pour le calcul de l'image courante. Même si les sphères ne permettent que d'éliminer un seul objet à la fois du pipeline, comme c'est le cas dans la figure, cela autorise tout de même la suppression des étapes de transformation et de test de visibilité pour tous les triangles le composant (puisque de toute façon ils auraient été supprimés à cette deuxième étape du pipeline). Le gain pour une scène dont les boîtes englobantes sont correctement calculées (dont les objets sont correctement répartis et englobés), peut être très appréciable.

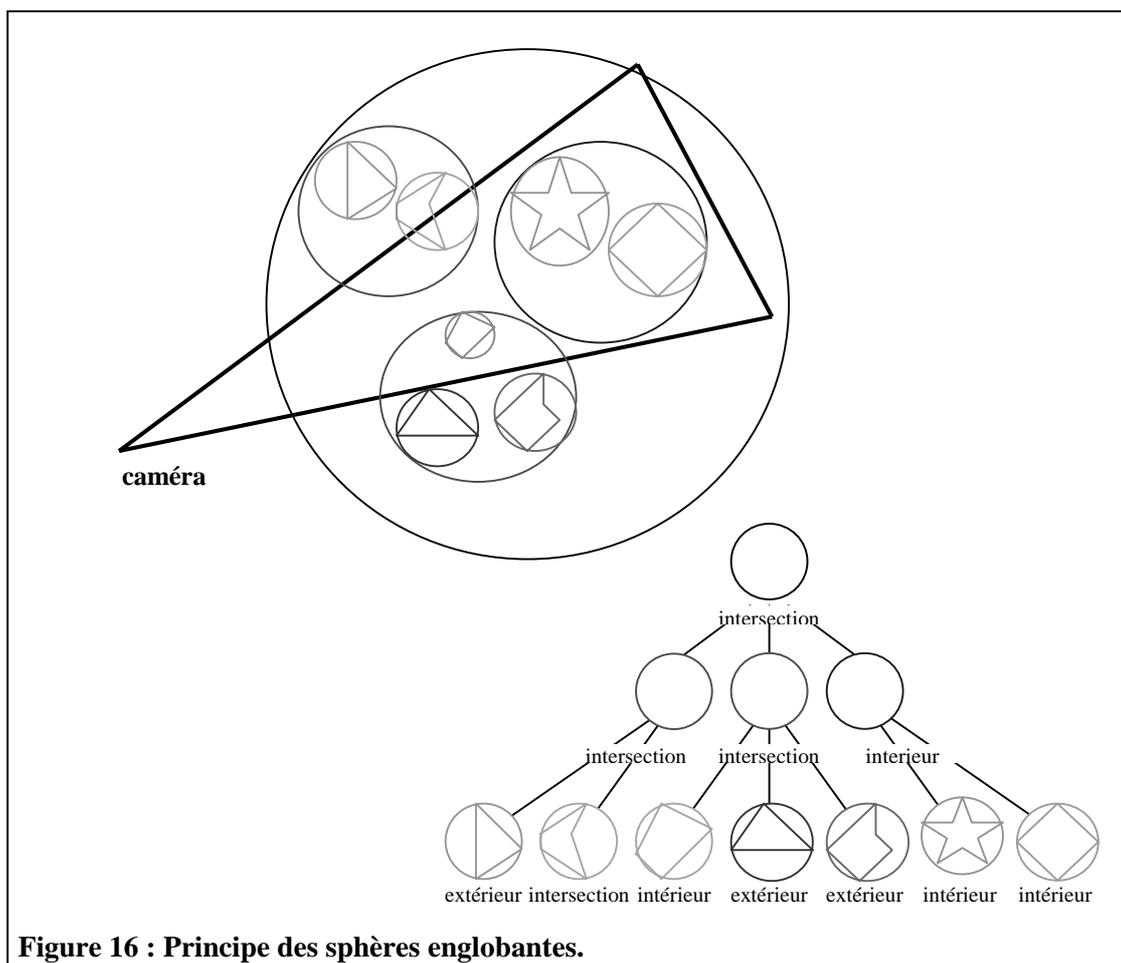
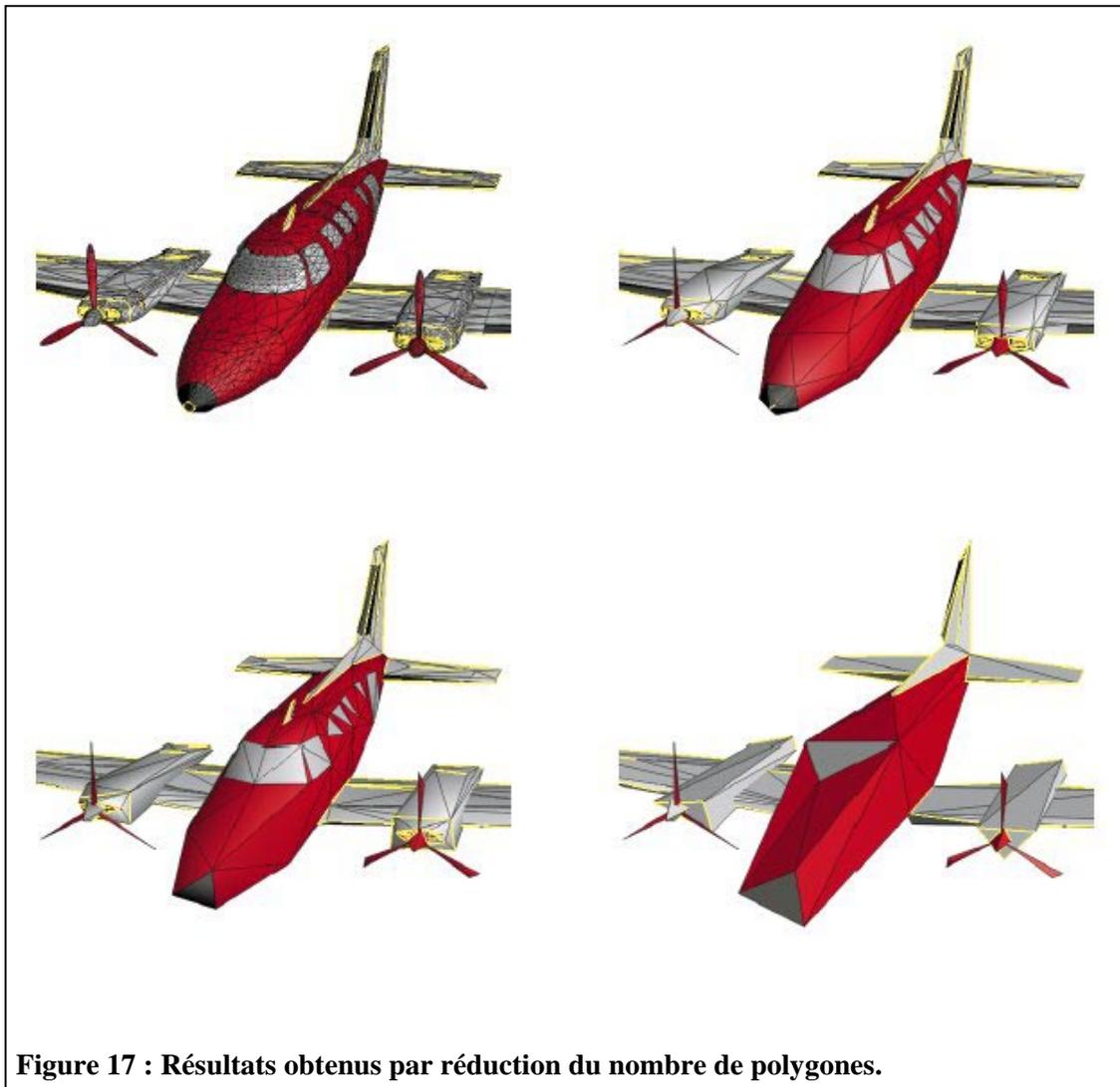


Figure 16 : Principe des sphères englobantes.

Une autre méthode est l'utilisation de rendus différents selon la distance séparant un objet de la caméra. Ces changements d'apparence d'un objet en fonction de la distance peuvent être spécifiés dans le fichier décrivant la scène (par les nœuds *LOD* en VRML97). La simplification d'un objet peut non seulement être faite sur son coloriage (suppression de la texture par exemple) mais aussi sur sa forme. Dans ce cas, certaines techniques de réduction du nombre de polygones pourront être utilisées. Ces techniques sont également de plus en plus utilisées en temps réel. Ainsi, les optimisations sont opérées par le navigateur pour toutes les scènes chargées.



1.2 Positionnement des différents éléments dans une architecture 3D

Le processus menant de la description d'une scène (par un langage prévu à cet effet ou par une série de commandes d'une bibliothèques 3D) à son affichage à l'écran est complexe. Le pipeline 3D présenté précédemment n'est qu'une partie des calculs nécessaires et des ressources utilisées pour l'obtention d'un résultat visuel. En aucun cas, le pipeline n'explique comment des scènes consécutives sont affichées à une fréquence telle qu'elle permet une interactivité. Pour cela, nous devons étudier l'ensemble du processus de production qui passe par différentes phases ; chacune d'elles prise en charge par une couche spécifique. L'ensemble de ces couches forment une architecture 3D complète qui est illustrée dans la Figure 18.

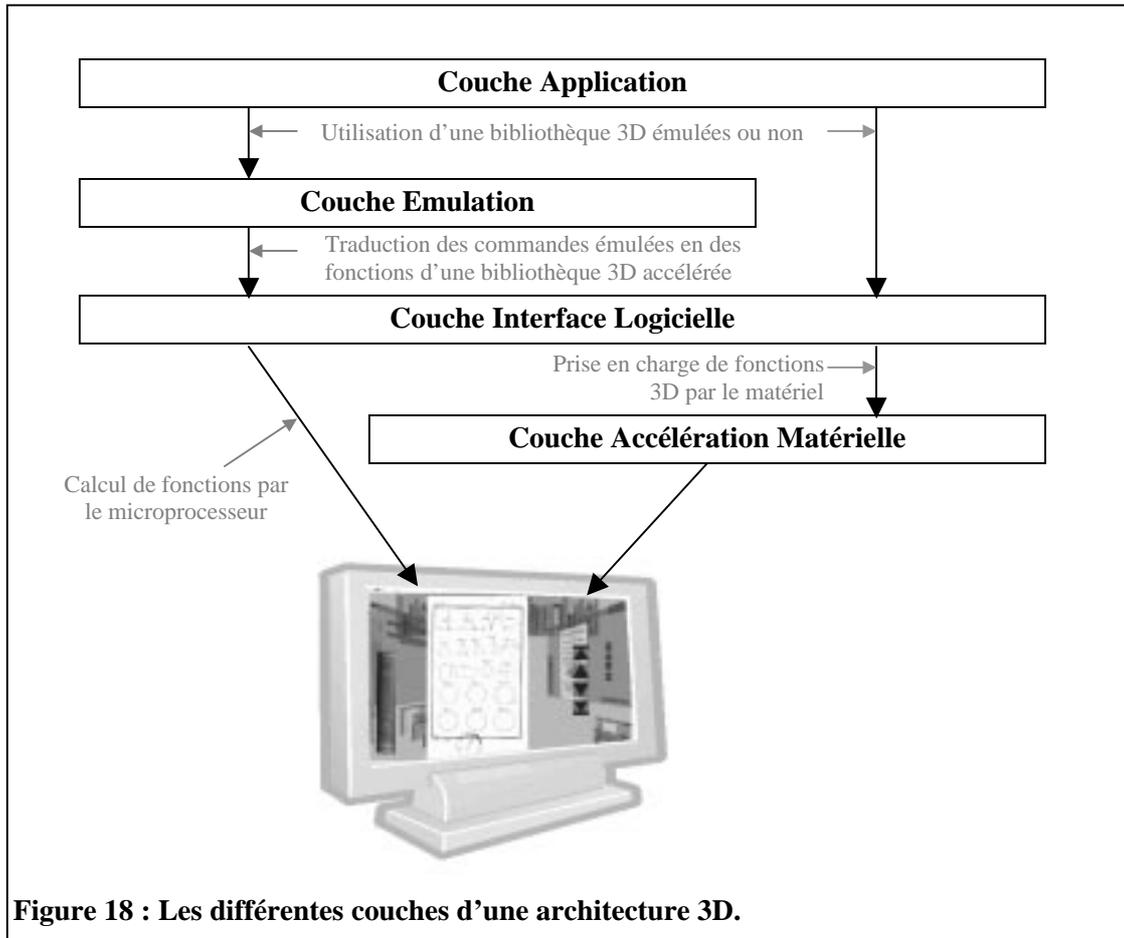


Figure 18 : Les différentes couches d'une architecture 3D.

1.2.1 Couche Accélération Matérielle

La carte graphique, composant essentiel de cette couche accélératrice, a pour but premier de gérer l'affichage des données à l'écran. Depuis vingt ans, plusieurs normes graphiques se sont succédées : CGA, EGA, VGA, SVGA et XGA. Grâce à cette progression, les cartes vidéos ont pu offrir une définition de plus en plus fine et un choix de couleurs de plus en plus important. En général, une carte vidéo est composée des éléments suivants :

- un processeur graphique (GPU – *Graphic Processir Unit*), qui exécute rapidement diverses opérations graphiques (rotation, translation, coloriage, remplissage...) à la demande du microprocesseur,
- une zone de mémoire vidéo à accès direct et dédiée à l'image affichée à l'écran (composée de puces de mémoire dont le niveau de performance varie: DRAM, EDO DRAM, VRAM,

SDRAM, SGRAM),

- un contrôleur graphique, qui reçoit les informations numériques en provenance du processeur central et les transforme en une matrice de points qu'il écrit dans la zone de mémoire dédiée à l'affichage et d'un convertisseur numérique-analogique (ou RAMDAC, pour *RAM Digital-Analog Converter*), qui permet de convertir la matrice de points en signaux analogiques qui vont piloter le canon à électrons du moniteur lors de l'affichage.

Les cartes 2D utilisent le même principe depuis leur création. Chaque processeur possède de nombreux circuits câblés contenant toutes les informations pour réaliser la plupart des fonctions simples telles que le déplacement des blocs ou du curseur de la souris, le tracé des lignes et des polygones plans ainsi que leur remplissage, etc. Maintenant que toutes les cartes gèrent ce genre de fonctions et disposent d'une mémoire suffisante pour atteindre des résolutions importantes, on assiste à une stagnation des performances 2D. Ces performances dépendent désormais uniquement de l'architecture des cartes ainsi que de la qualité de leur driver. Le type de mémoire embarquée a également une importance majeure pour la vitesse d'affichage. Pour exemple, l'emploi de mémoire EDO à 60 ns plutôt que des SGRAM ou WRAM (mémoires vidéo spécifiques) à 10 ns peut faire chuter les performances d'une façon étonnante. Enfin, la capacité de la mémoire vidéo et la fréquence du RAMDAC ne permettent en aucune façon l'accélération de l'affichage 2D mais autorisent simplement un plus grand confort grâce à une palette de couleurs plus importante ainsi qu'un taux de rafraîchissement élevé même dans les hautes définitions.

Avec l'arrivée de nombreux logiciels et jeux vidéos ayant recours aux graphiques en 3D, des circuits câblés spécifiques ont été ajoutés aux fonctions 2D déjà présentes. C'est en 1995 qu'apparaît le premier accélérateur 3D grand public disponible sous la forme d'une carte fille reliée à la carte vidéo principale. Par la suite, les accélérateurs 3D ont pris place directement sur la carte vidéo principale. Aujourd'hui, la quasi totalité des cartes graphiques disposent d'un accélérateur 3D afin de décharger le processeur des tâches affectées au rendu. L'accélération 2D n'étant plus un enjeu majeur pour les fondeurs de cartes graphiques, ce sont les performances des fonctions 3D qui ne cessent d'être améliorées.

Le nombre et le type de fonctions 3D accélérées dépendent de la carte graphique utilisée. Comme cela est illustré dans la Figure 19, toutes les générations de cartes graphiques n'accélèrent pas les mêmes étapes du pipeline 3D. Comme nous pouvons le voir, et cela rejoint notre propos du paragraphe précédent, les cartes 3D ne cessent d'étendre, depuis leur apparition jusqu'à aujourd'hui, leur prise en charge des calculs nécessaires à la 3D.

Les cartes 2D ne gèrent aucun aspect propre à la 3D. Les dernières cartes uniquement 2D du marché permettent cependant le transfert rapide de zone mémoires. Ainsi, il est possible d'effectuer des animations fluides grâce à une gestion du *double-buffer*⁴. Avec une prise en charge totale des calculs 3D par le microprocesseur, ces cartes suffisent pour afficher des scènes simples avec un mode de coloriage peu coûteux.

En plus des zones mémoires où sont stockées les images à afficher à l'écran, la première génération de cartes 3D prend en charge les calculs de rendu des scènes 3D. Une zone mémoire supplémentaire est ajoutée à la carte graphique pour y stocker les textures. Depuis lors, l'utilisation de la mémoire embarquée sur les cartes 3D a toujours été la même : une moitié de la mémoire pour l'affichage et l'autre moitié pour les textures utilisées lors du rendu. Les premières fonctions 3D câblées ont été le plaquage de texture et la gestion du *Z-Buffer*.

Les cartes 3D de deuxième génération ont étendu les possibilités des cartes précédentes en offrant des fonctions plus avancées pour le rendu des objets telles que l'*antialiasing* des scènes, le

⁴ Voir le glossaire page 149.

filtrage bi et tri linéaire. La partie supérieure du pipeline 3D (transformations, *clipping*, testes de visibilité, gestion des sources lumineuses) est cependant toujours prise en charge par le microprocesseur.

La troisième génération de cartes 3D intègre un nouveau module que l'on nomme *Transform and Ligthning* (T&L ou TnL). Ces nouvelles fonctions permettent à une telle carte d'effectuer les projections parallèle et orthogonale, les transformations géométriques, le *clipping* et les éclairages et ainsi traiter seule le processus entier de conversion des triangles en des pixels d'une image.

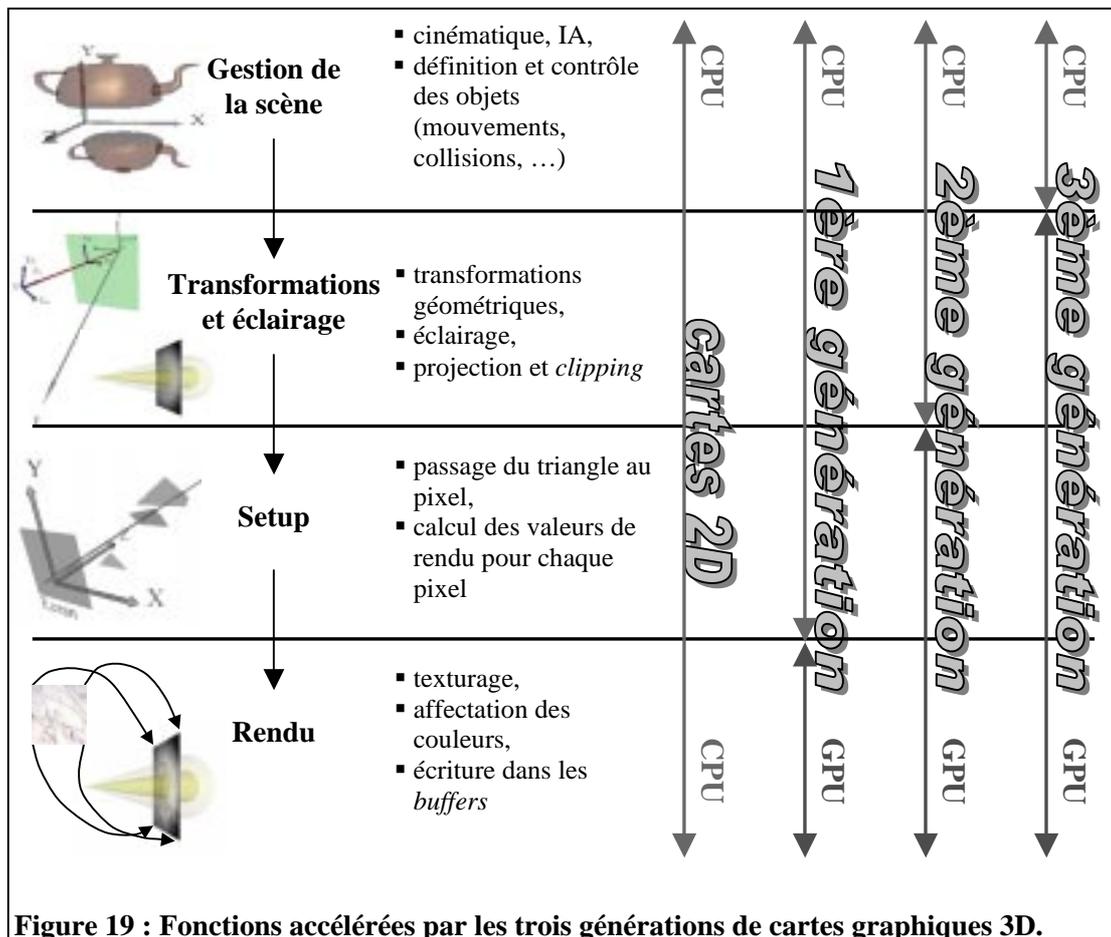


Figure 19 : Fonctions accélérées par les trois générations de cartes graphiques 3D.

Le microprocesseur ne gère plus que les aspects inhérents aux mouvements de l'utilisateur (c'est-à-dire du point de vue sur la scène) et aux mouvements des objets. Dans une scène animée et/ou hautement interactive, cette part du travail peut toutefois représenter 50% du temps de calcul total nécessaire. Sachant qu'une carte 3D de troisième génération effectue les opérations du pipeline graphique au pire en deux fois moins de temps qu'un microprocesseur récent alors l'utilisation d'une telle carte offre au moins un gain de 25% par rapport à l'utilisation unique d'un microprocesseur.

Depuis l'apparition du Pentium MMX (*Matrix Math eXtension*) d'Intel, les microprocesseurs disposent également d'instructions spécialisées pour les applications multimédias. Avec les instructions MMX, aucune accélération 3D n'était cependant possible car les 57 nouvelles instructions ne permettaient que d'effectuer des opérations sur les entiers (alors que les calculs inhérents à la 3D sont essentiellement sur des flottants). Pour combler cette lacune, AMD a étendu le jeu d'instructions MMX par 21 instructions supplémentaires permettant des opérations sur les flottants. Ces nouvelles instructions, baptisées 3DNow!, permettent d'accélérer notablement les applications 3D. Devant ce succès, Intel a emboîté immédiatement le pas à AMD en proposant les instructions SSE (*Streaming SIMD Extensions*) puis SSE2 pour prendre en charge les applications nécessitant de nombreux calculs

sur les flottants. Une synthèse très complète a été effectuée sur ce sujet par H. Doornbos pour son oral probatoire du CNAM [DOO2000].

On peut se demander, après avoir vu la Figure 19, si les instructions spécialisées des microprocesseurs servent encore à quelque chose puisque les nouvelles cartes accélèrent la majorité des calculs nécessaires à l'affichage d'une scène 3D. Justement, les cartes 3D ne s'occupent que de l'affichage et ne prennent absolument pas en charge les à-côtés. Ainsi, les interactions, les mouvements, les collisions et les autres aspects qui rendent une application 3D interactive, devront être pris en charge par le microprocesseur. Il suffit de se rappeler que ces calculs peuvent représenter 50% des traitements dans certaines applications 3D pour se convaincre de l'apport de ses nouvelles instructions spécialisées.

1.2.2 Couche Interface Logicielle

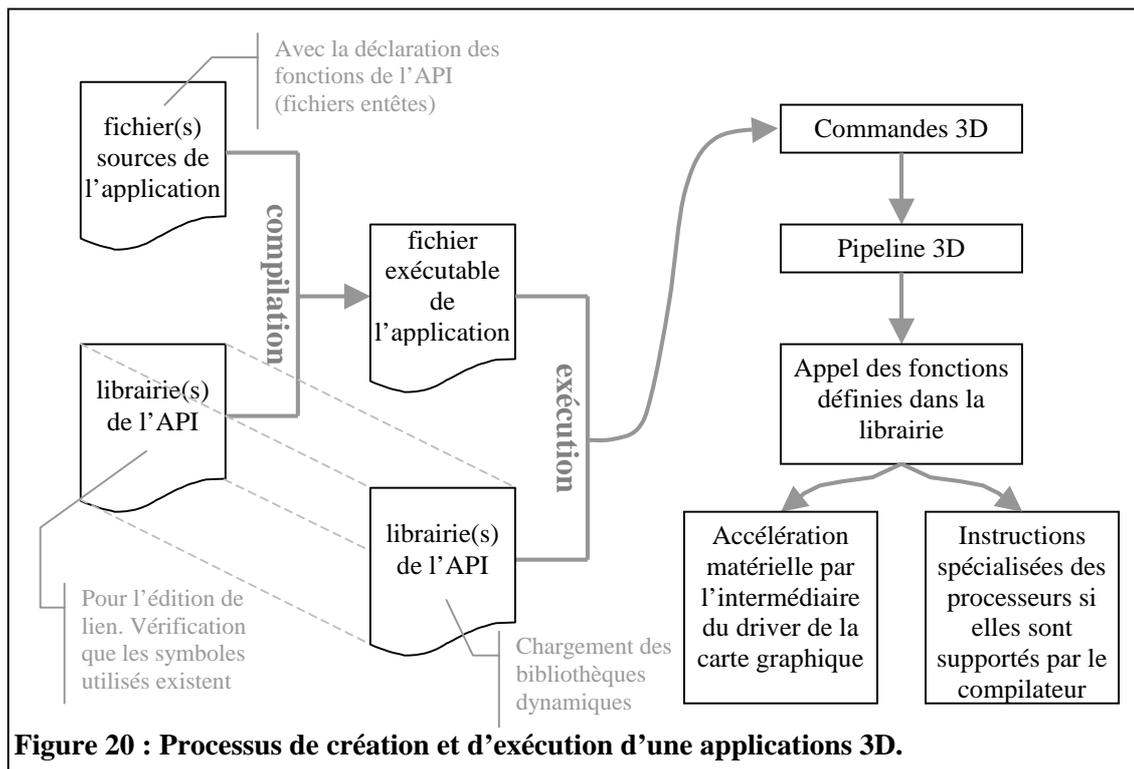
Dans cette couche logicielle sont regroupées les bibliothèques de fonctions rendant l'accès à une accélération 3D plus aisé. Sans ces API, il serait néanmoins possible d'afficher un contenu 3D mais toutes les opérations du pipeline 3D devraient être explicitement programmées. Lorsqu'on programme une application en s'appuyant sur une API 3D, il suffit de donner une série de commandes (définies par l'API) permettant de décrire la scène que l'on souhaite afficher. Ces commandes définissent la forme géométrique et l'apparence des objets manipulés et les options de rendu. Les différentes étapes du pipeline sont alors entièrement prises en charge et de manière transparente par l'API lorsqu'on lui demande d'afficher la scène. En aucune façon le développeur d'une application 3D n'aura besoin de se soucier de la programmation explicite du matériel présent. C'est là que la couche logicielle revêt toute son importance. En effet, s'il fallait programmer les registres d'une carte 3D pour accéder aux fonctions accélérées, il serait évident que la plupart des applications ne seraient vendues que pour quelques configurations précises.

Les cartes 3D sont fournies avec leurs drivers. Ceux-ci se présentent sous la forme de bibliothèques dynamiques effectuant le lien entre les commandes des API 3D et les fonctions câblées sur la carte. Les API supportées sont généralement OpenGL de Silicon Graphics et Direct3D de Microsoft. Alors que pour Direct3D toutes les fonctions sont généralement supportées, pour OpenGL c'est rarement le cas. Les cartes d'entrée de gamme ne proposent en effet qu'une accélération partielle. Deux options s'offrent alors aux fabricants de cartes 3D :

- Écrire un ICD (*Installable Client Driver*) : une implémentation complète (*TnL + Rasterization*). Le fabricant de carte 3D peut optimiser le driver pour chaque type de CPU (SSE, SSE2 et 3DNow!) afin d'augmenter les performances. Dans un ICD, toutes les fonctions OpenGL doivent être implémentées. Si l'une d'entre elles ne peut pas être effectuée par la carte, le driver passe alors en mode logiciel (la fonction est opérée par le microprocesseur).
- Écrire un MCD (*Mini Client Driver*) : une version allégée de l'ICD dans laquelle n'est développée que l'étape de *rasterization* (Setup+Rendu).

Quelque soit l'API utilisée, cette couche logicielle doit pouvoir déterminer si une opération peut être effectuée par le matériel ou si elle doit être émulée par le microprocesseur. Pour cela, toute API dispose d'un mécanisme permettant de questionner le matériel sur ses possibilités. En fonction des résultats, elle aiguillera l'exécution des commandes vers l'un ou l'autre des composants.

Deux familles d'éléments composent donc cette couche logicielle : les API 3D (elles-mêmes constituées d'une bibliothèque et d'un fichier de déclaration de leurs fonctions pour être utilisées depuis un langage de programmation) et les drivers des cartes graphiques (qui effectuent le lien entre les fonctions de l'API et la carte graphique). Le processus menant d'un code source à une application utilisant une API3D est schématisé dans la Figure 20.



1.2.3 Couche Emulation

La couche émulation est présente sur certaines architectures lorsque l'API 3D utilisée par le programmeur est traduite en une autre API. C'est le cas notamment de Java3D et de Fahrenheit dont les différents éléments de leur graphe de scène sont traduits soit en Direct3D soit en OpenGL. De cette manière, ces graphes de scène dans lesquels sont stockées toutes les commandes 3D bénéficient d'un rendu accéléré par l'intermédiaire de deux autres API. Sans cette traduction, ces API ne sont pas accélérées puisque les fonctions 3D des cartes graphiques ne sont pas câblées pour elles.

On différencie ici les API 3D qui sont traduites en une autre API 3D en vue d'être accélérée et les applications qui utilisent directement une API. Tous les *bindings* permettant d'accéder aux fonctions OpenGL depuis un langage de programmation (autre que ceux accédant directement aux bibliothèques dynamiques d'OpenGL) peuvent être considérés comme étant des API 3D « émulées ». En effet, le programmeur accèdera aux fonctions d'un *binding* qui lui-même accèdera aux véritables fonctions OpenGL. Etant donnée cette définition d'un *binding* OpenGL, on considèrera donc que Java3D en est un. Pour le langage Java, il en existe de nombreux autres comme : GL4Java, Magician, YAJOGLB (*Yet Another Java OpenGL Binding*), Jogl, Tree (interface Java pour Mesa 3D), JavaOpenGL. Mais le langage Java n'est pas le seul à bénéficier des *bindings* OpenGL puisque l'on peut également en trouver pour les langages Perl, Fortran, Ada et Python.

Les bibliothèques de fonctions qui n'émulent pas entièrement l'API cible peuvent également être considérées comme des API « émulées ». Les bibliothèques d'utilitaires pour OpenGL par exemple, qui regroupent des fonctions permettant de créer des objets complexes, utilisent OpenGL pour construire ces objets. Ces différentes bibliothèques se placent bien entre l'application et l'API 3D accélérée par la carte graphique.

1.2.4 Couche Application

Dans la couche application de notre architecture 3D nous regroupons tous les logiciels dans lesquels une interface est implémentée à l'aide d'une API 3D. Toutes les applications n'utilisent pas

nécessairement une API 3D (accélérée ou non, émulée ou non). Les opérations 3D peuvent en effet être calculées par l'application elle-même et ne nécessitent alors que des primitives mathématiques et 2D (pour tracer les points). Cependant, dans la majorité des cas et par soucis de performance la quasi-totalité des applications actuelles utilisent un rendu accéléré et c'est pourquoi nous ne considérons que celles-ci dans notre architecture.

Les différents moteurs 3D du Web que nous verrons en détail dans le chapitre 3 page 67 font partie de cette couche application puisqu'ils utilisent généralement les API OpenGL ou/et Direct3D.

C'est au niveau de l'application qu'un développeur doit interroger une API pour connaître les options de rendu qui sont accélérées. De cette manière, il sera possible de ne pas utiliser certaines options afin de favoriser les performances et, au contraire, d'enrichir la scène si toutes les fonctions sont accélérées.

1.3 Présentation de l'architecture 3D sous Linux

Linux est un système d'exploitation encore très récent. A l'origine, c'est l'œuvre d'une seule personne : Linus Torvalds. Passionné par le système d'exploitation MINIX, un UNIX simplifié, il décida en 1991 d'en créer un plus complet pour remplacer, sur son PC, le DOS, qui souffrait de quelques limitations (mono-tâche, mono-utilisateur, problème de gestion de la mémoire et des disques de grandes capacités, ...). En 1994, son travail fut récompensé par la sortie du noyau Linux version 1.0. Depuis lors, l'intérêt porté à ce système ouvert n'a cessé de croître. De nombreux programmeurs ont participé à l'enrichissement de Linux en proposant de nouveaux drivers et de nouvelles applications. C'est grâce à cette communauté que nous pouvons installer aujourd'hui des distributions Linux très complètes.

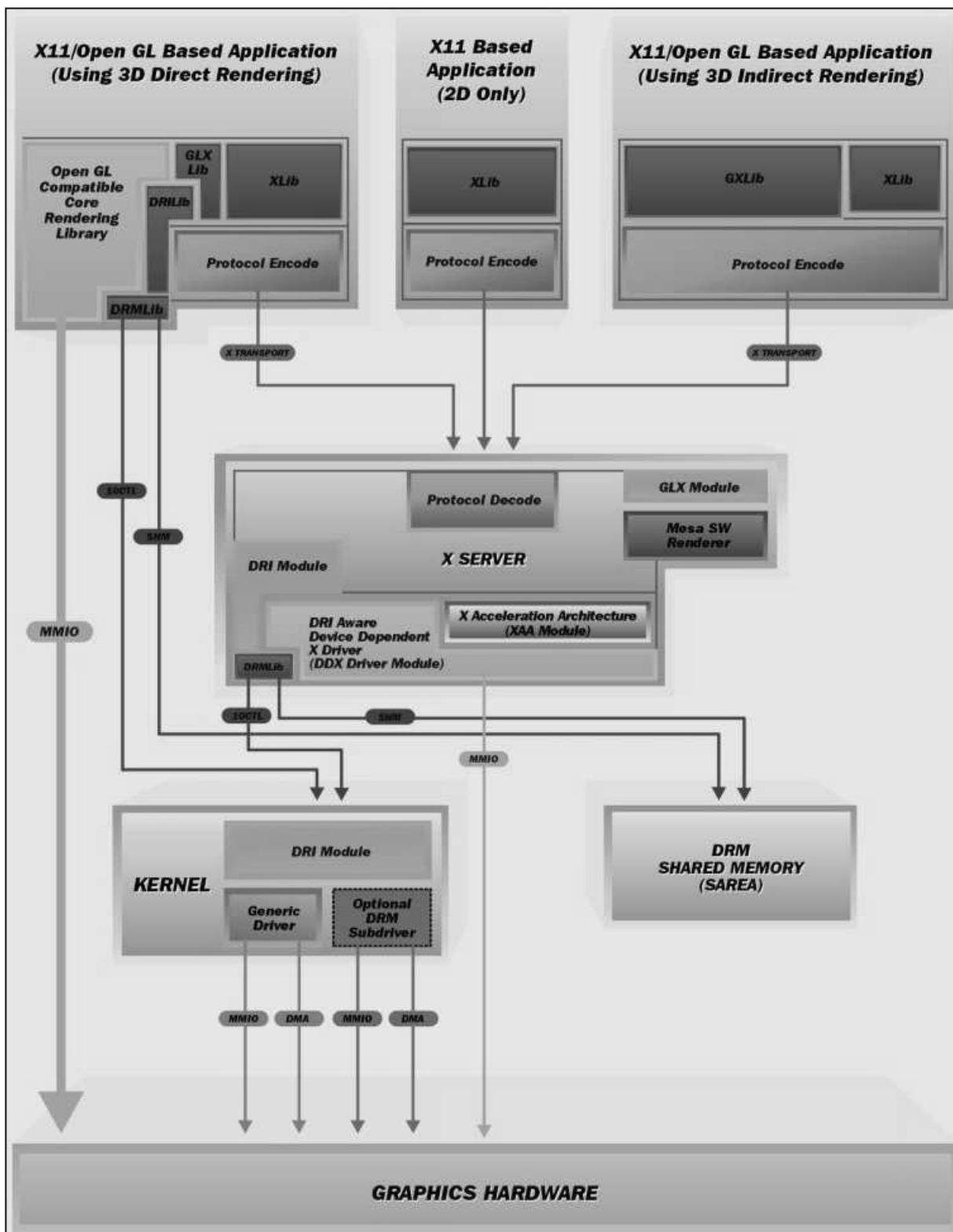
Le succès de la 3D sous Linux ressemble étrangement à celui de Linux lui-même. En effet, la principale avancée est l'œuvre d'un seul homme. Brian Paul a mis à disposition sa librairie Mesa⁵, utilisant une syntaxe et un automate semblables à ceux d'OpenGL (avec l'autorisation de Silicon Graphics). Créé en 1993, ce clone d'OpenGL a également connu un intérêt croissant qui a mené à de nombreux ajouts et de nombreux portages sur différentes plates-formes. Et récemment, les plus grands progrès de Linux et de Mesa concernent la prise en charge de l'accélération matérielle par différentes cartes graphiques. Linux est ainsi devenu une plate-forme idéale pour le développement d'applications 3D.

L'accélération des fonctions 3D sous Linux revêt aujourd'hui deux formes. La première, et la plus ancienne, utilise une interface permettant d'effectuer un rendu accéléré d'une application 3D utilisant Mesa dans une fenêtre X11. Cette interface, appelée *Accelerated GLX*, est une mise à jour de GLX (*Graphic Library eXtension*) pour prendre en charge l'accélération offerte sur certaines cartes graphiques. Ce travail a été possible grâce à Silicon Graphics, encore une fois, qui a rendu publics les sources de GLX. Il suffit, pour en profiter, de trouver un driver GLX pour la carte graphique installée.

La deuxième forme d'accélération 3D sous Linux nécessite l'utilisation de la DRI (*Direct Rendering Infrastructure*) de Precision Insight. Cette solution est en fait un complément de la première. Elle intègre en effet le module GLX (accéléré ou non ; cela dépend de la carte graphique) directement dans le serveur X mais offre, en plus, un chemin optimisé pour envoyer les ordres de rendu 3D directement aux cartes graphiques supportées. Le serveur XFree86 4.0⁶ intègre cette architecture de rendu direct qui est schématisée dans la Figure 21. Bien évidemment, cette architecture ne pourra être utilisée que pour un rendu local (le client et le serveur OpenGL étant la même machine). Pour le rendu sur une machine distante, la carte graphique de la machine effectuant les calculs ne peut pas être utilisée.

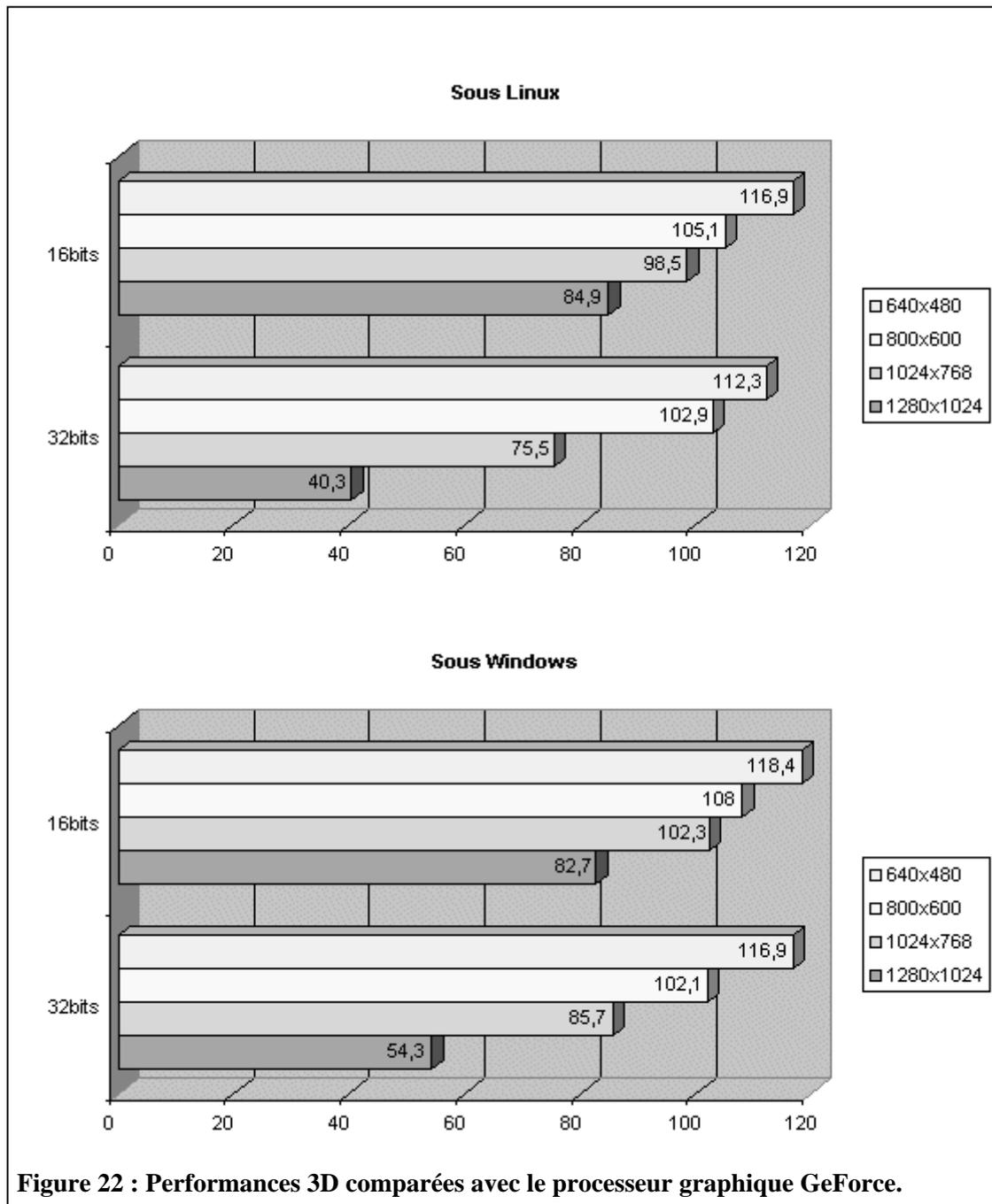
⁵ Téléchargement et documentation à l'URL <http://www.mesa3d.org>.

⁶ Le serveur XFree86 4.0 est disponible, ainsi que sa documentation à l'URL <http://www.xfree.org>.



Source : Precision Insight

Figure 21 : L'architecture DRI de XFree86 4.0.



Cette nouvelle architecture souffre encore de quelques petits problèmes. Pour la plupart, ils proviennent de sa relative jeunesse et du fait que son développement soit *Open Source*. Les drivers et applications libres de droits requièrent généralement un léger temps d'adaptation pour parvenir à leur maturité. Actuellement, XFree86 4.0 n'est fourni dans les distributions qu'à titre d'environnement expérimental. La version 3.3.6 de XFree86 fait office de serveur X par défaut. Les problèmes de XFree86 4.0 sont cependant répertoriés et en voie d'être réglés dans sa prochaine version :

- le setup utilisant une interface graphique n'est pas implémenté,
- la configuration est encore difficile, s'adressant aux utilisateurs experts,
- l'accélération ne fonctionne pas encore avec toutes les cartes graphiques,

- le rendu direct ne fonctionne pas encore avec toutes les applications,
- les drivers DRI sont plus lents que les drivers GLX de la version 3.3.6 dans certains cas et pour certaines cartes.

Malgré ces problèmes techniques, l'architecture DRI nous promet, dans sa version définitive, de grandes performances. Sa version bêta nous donne déjà de très bons résultats. La Figure 22 nous donne les performances 3D obtenues sous Windows et sous Linux avec XFree86 4.0 pour une carte embarquant le processeur GeForce de NVidia. Le *benchmark* (mesureur de performances) utilisé est Quake III (avec OpenGL) dans son mode qualité maximum. Nous constatons que les deux plateformes se valent pour les basses résolutions alors que l'accélération OpenGL sous Windows compte déjà plus de quatre ans d'existence contre quelques mois en phase bêta pour la DRI. Linux accuse encore un léger retard pour les hautes résolutions avec des couleurs codées sur 32 bits. Cependant, ces résultats nous laissent envisager de très bonnes performances de la DRI en version mûre et stable. Linux n'aurait alors plus rien à envier à Windows pour le développement d'applications 3D.

2 QUELQUES API 3D DISPONIBLES

Résumé Les API (*Application Program Interface*) présentées ici sont les plus utilisées par les applications exploitant une interface 3D temps réel (répondant instantanément aux interactions de l'utilisateur). Nous les décrivons en détail, donnons leurs avantages et inconvénients et proposons un squelette d'application pour les utiliser.

L'association d'un fichier décrivant une scène et d'un moteur permettant de la charger et d'y évoluer peut dans certains cas être insuffisant. En effet, certaines fonctions peuvent manquer ou au contraire être trop générales, bref, ne pas être satisfaisantes. En particulier, les différentes interactions gérées par le navigateur ne sont pas nécessairement les plus appropriées pour certaines scènes. La seule alternative pour créer un moteur 3D "sur mesure" est de programmer nous même ses différents comportements. Cependant, les fonctions de projection, de *Z-Buffering* ou de plaqué de textures par exemple restent les mêmes quelles que soient les fonctionnalités voulues. C'est pourquoi il est fort intéressant, pour économiser un temps précieux dans le développement d'un moteur et pour tirer partie de l'accélération matérielle du rendu, d'utiliser une API 3D regroupant toutes ces opérations élémentaires en 3D.

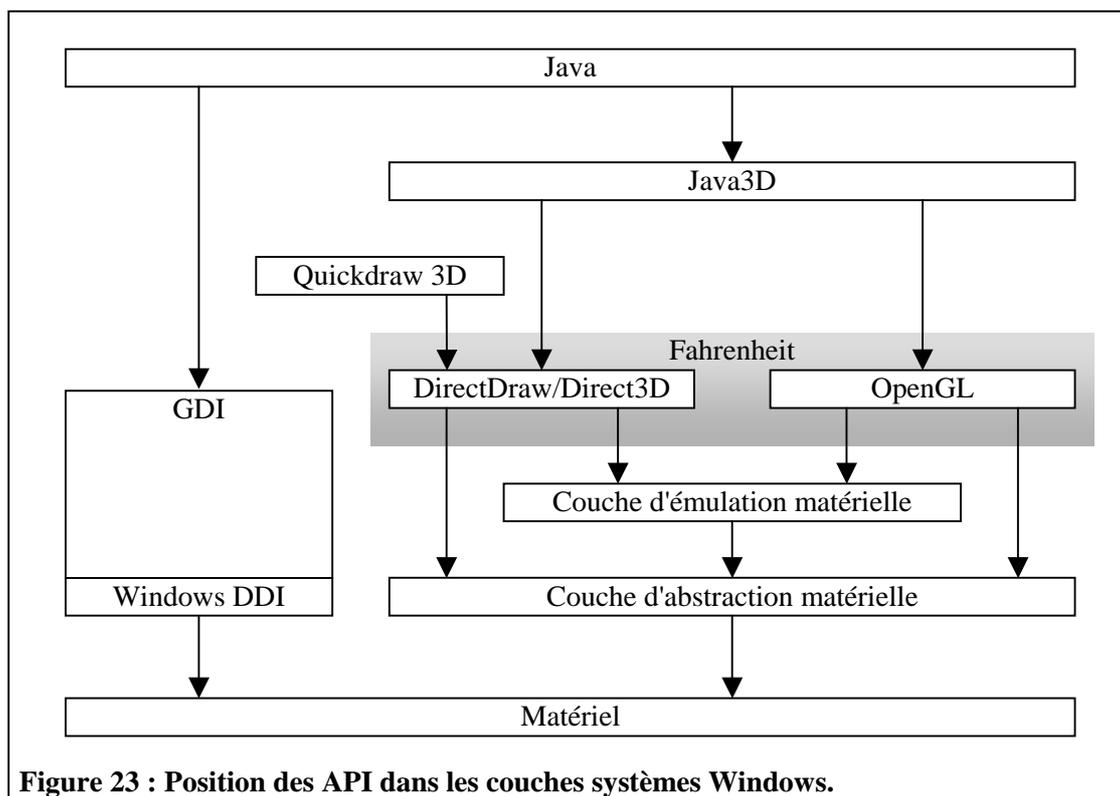


Figure 23 : Position des API dans les couches systèmes Windows.

Les API que nous présentons permettent toutes de créer des programmes interactifs qui produisent des images en couleur d'objets en trois dimensions mobiles. Il existe de nombreuses API 3D différentes⁷. Notre choix parmi ce grand nombre a été motivé par les critiques portant sur certaines d'entre elles et sur la quantité des fragments de code existant avec ces bibliothèques témoignant du soutien dont elles bénéficient. Ainsi, se sont dégagées de cette recherche trois API de bas niveau et

⁷ D'après [ISA], 643 au 26/03/2001.

deux API de haut niveau. Les trois premières (OpenGL, Direct3D et Quickdraw 3D) sont des interfaces entre l'application et le matériel graphique alors que les deux autres (Java3D et Fahrenheit) sont des interfaces entre l'application et les deux premières API précédentes. La position des API par rapport au système d'exploitation et au matériel est schématisé dans la Figure 23. Ce schéma est assez sommaire, les architectures logicielles des différentes API étant reprises plus en détails dans la suite du rapport.

Les API ont pour but de fournir des fonctions graphiques 3D de haut niveau tirant partie des possibilités du matériel présent sans devoir le programmer explicitement. Par conséquent, un seul programme fonctionnera avec toutes les cartes graphiques. Les fonctions graphiques prises entièrement en charge par le matériel (carte graphique et instructions spécialisées du microprocesseur) passent par la couche d'abstraction matérielle HAL (*Hardware Abstraction Layer*) de Windows alors que celles non supportées (ou non entièrement supportées) sont prises en charge par la couche d'émulation matérielle HEL (*Hardware Emulation Layer*). Les fonctions gérées par la carte graphique déchargeront en plus le travail du microprocesseur qui pourra se cantonner aux calculs physiques (pour la gestion de la cinématique des objets par exemple).

L'API DirectX de Microsoft regroupe DirectDraw (pour les surfaces et graphiques 2D), Direct3D (pour les fonctions 3D) et d'autres bibliothèques pour la gestion du son, des périphériques d'entrée et de sortie et la synchronisation entre plusieurs utilisateurs. C'est à cette API que l'on doit le développement du concept de HAL pour les cartes graphiques d'entrée de gamme offrant des fonctions 3D puisque les fonctions OpenGL étaient accélérées jusqu'à récemment que par des cartes haut de gamme pour les professionnels. Aujourd'hui, pratiquement toutes les cartes graphiques sont vendues également avec des drivers pour OpenGL. Quant à QuickDraw3D, l'API d'Apple, elle exploite le même principe aussi bien sur Macintosh que sur un PC sous Windows en passant par Direct3D.

Une application 3D effectuera bien évidemment des appels aux fonctions 3D de ces API mais, également, pour la gestion des fenêtres ou des événements par exemple, des appels aux fonctions de l'interface graphique par l'intermédiaire de son API, la GDI⁸, qui est liée au matériel par la couche DDI⁹. La physionomie de l'ensemble Java et Java3D est quasiment identique puisque Java gère le côté 2D et Java3D tout ce qui concerne la 3D.

2.1 Quickdraw3D

Avant d'étudier de façon plus approfondie OpenGL, Direct3D et Java3D, nous nous penchons tout d'abord sur QuickDraw3D et il convient, dans un premier temps, d'expliquer pourquoi nous ne l'étudions pas en détails.

Depuis quelques mois déjà, la *mailing-list* QuickDraw3D tenue par Apple est dominée par un sujet qui suscite des discussions houleuses, parfois même avec véhémence, animosité ou fanatisme de la part des participants. Cette agitation fait suite à l'annonce d'Apple, quelques temps après avoir finalisée la version 1.6 de QuickDraw3D, de l'abandon du développement de leur bibliothèque graphique 3D lui préférant OpenGL qui fait donc officiellement son entrée sur plate-forme Macintosh. Bien qu'y ayant consacré quelques jours de recherche, il nous a donc semblé inutile de présenter longuement cette API qui est en passe de devenir obsolète malgré de nombreuses signatures contre cet abandon¹⁰. Cependant, puisque, jusqu'à ce jour, elle fait encore partie de la version actuelle du *SDK* (*System Development Kit* (boîte à outils de développement)) QuickTime¹¹, nous allons en dire quelques mots.

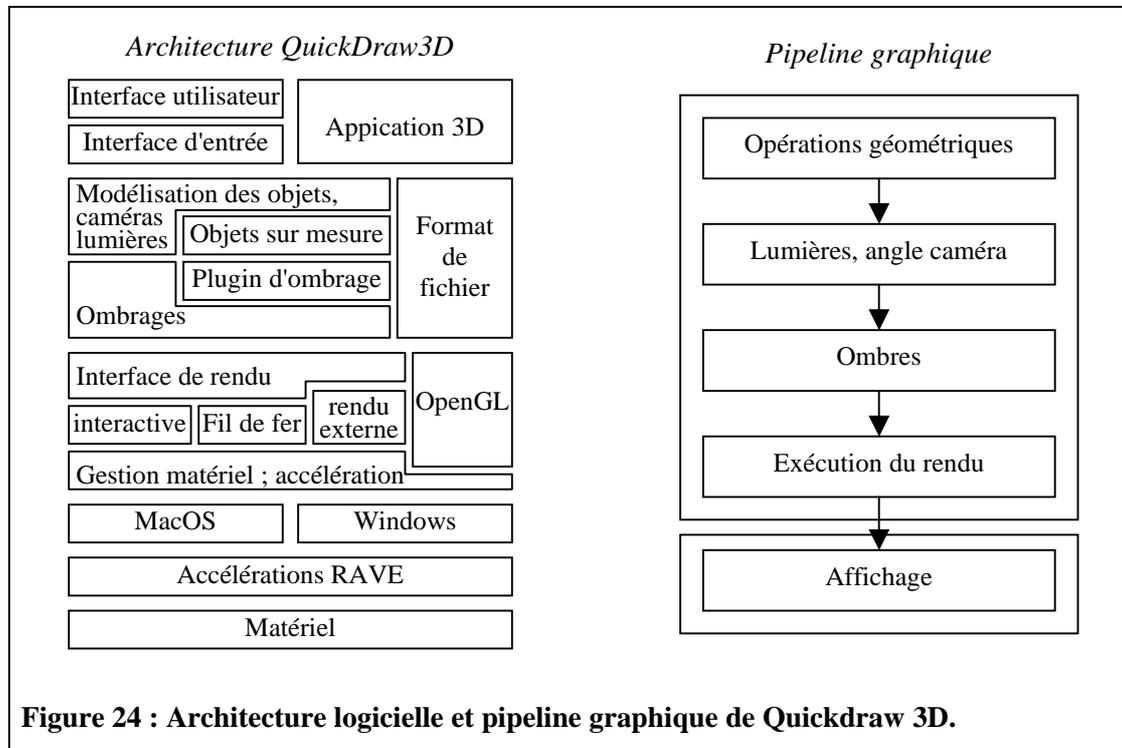
⁸ Graphic Device Interface.

⁹ Device-Driver Interface.

¹⁰ Voir la pétition en ligne à l'URL <http://www.designcommunity.com/qd3d/petition.html>.

¹¹ A l'URL <http://www.apple.com/techpubs/quicktime> on pourra télécharger QuickTime 4 pour Windows ou MacOS.

QuickDraw3D est un environnement complet pour traiter les graphiques en trois dimensions. Le niveau supérieur de l'API autorise les développeurs à créer et à manipuler des objets 3D mais, également, à lire et à écrire des données 3D dans un fichier. Cette couche haute communique avec le pipeline graphique qui exécute les commandes graphiques. A son tour, le pipeline communique avec le niveau inférieur qui est, comme pour Direct3D, une couche d'abstraction matérielle connue sous le nom de QuickDraw3D RAVE (*Renderer Acceleration Virtual Engine*). L'architecture logicielle complète de QuickDraw3D et son pipeline graphique sont donnés dans la figure suivante :



L'architecture illustre la grande souplesse de la bibliothèque et en particulier les possibilités de créer des objets graphiques complexes et une interface d'entrée sur mesure sans que leurs liaisons au reste de l'architecture soit un problème. Quant au pipeline graphique, il a une structure classique puisqu'il effectue les opérations que tout pipeline graphique effectue. Cependant, on notera que, parce que les objets utilisent l'héritage de la programmation orientée objets, l'état du pipeline est mis à jour automatiquement entre les opérations précédant le rendu et le rendu lui-même.

QuickDraw3D supporte comme toutes les autres API que nous présentons ici, un mode immédiat et un mode retenu. Le mode immédiat est identique, nous le verrons, à celui d'OpenGL dans le sens où c'est à l'application de donner les commandes graphiques au pipeline qui doit les traiter le plus rapidement possible au fur et à mesure qu'elles arrivent. Le mode retenu s'apparente plus à celui de Java3D car le programmeur manipule une hiérarchie de structures qui contiennent les éléments de la scène et qui servent à son rendu. Le paradigme de programmation orientée objets par lequel on fait appel aux fonctions de cette bibliothèque apporte tous les bienfaits qu'on lui connaît, comme la présence d'un constructeur qui permet d'initialiser par défaut les différents attributs ou l'héritage qui apporte une souplesse pour la maintenance de la scène. Le développeur aura le droit de choisir entre un des deux modes ou un mélange des deux si la situation le requiert.

QuickDraw3D, particulièrement quand elle est utilisée dans son mode retenu, est une bibliothèque de fonctions adaptée aux programmeurs novices en 3D qui souhaitent créer une application 3D. L'API offre, en effet, de nombreux objets élémentaires (comme les lignes, les sphères, les cônes, ...) qui permettent de prototyper facilement et rapidement une scène. Elle offre également une interface permettant de visualiser et d'éditer les objets d'une scène, de les sauvegarder et de les charger à l'aide du format de fichier multi plate-forme *3DMF (3D Metafile)*. Quant aux programmeurs

qui comprennent et dominent plus en avant les détails des concepts techniques de la 3D, ils pourront utiliser le mode immédiat qui offre plus de contrôle.

2.2 OpenGL – MESA

Le système graphique OpenGL, de Silicon Graphics Inc. (SGI), s'est imposé comme la bibliothèque de référence pour réaliser des projets de synthèse d'images en temps réel dans le milieu professionnel. Face à son grand succès sur les stations Silicon Graphics, plate-forme sur laquelle elle peut être considérée comme succédant à la bibliothèque IRIS-GL, cette librairie s'est maintenant généralisée sur de nombreux systèmes d'exploitation. Des clones de cette fameuse librairie sont d'ailleurs disponibles sur certains systèmes d'exploitation avant que l'originale ne soit portée. Le plus connu de ces clones est Mesa de Brian Paul, disponible sous Linux, Windows et plus récemment sur Macintosh.

S'appuyant sur leur expérience des bibliothèques de fonctions 3D, SGI a voulu proposer un produit facile à comprendre et à utiliser. Par rapport à IRIS-GL, leur ancienne API, de nettes améliorations ont été apportées :

- Elle est multi plate-forme par sa conception.
- Inspirée par l'interface graphique XWindow, elle propose un modèle client-serveur sur réseau hétérogène où le serveur effectue les calculs et le client l'affichage. Ainsi, le serveur peut être un supercalculateur qui exécute des calculs complexes de simulation alors que le client est un poste de travail principalement utilisé pour ses capacités de visualisation. Cependant, il est bien sûr possible que tout soit traité sur la même machine.

En plus d'être multi plate-forme, la librairie OpenGL peut être utilisée depuis d'autres langages de programmation que le langage C : Fortran, Ada, C++ et Java. Cette caractéristique fait d'OpenGL la librairie graphique utilisable sur le plus grand nombre d'environnements différents.

Comparée à Quickdraw3D et Direct3D, OpenGL n'est pas une API de haut niveau dans le sens où elle ne fournit pas de fonctions pour effectuer des tâches de haut niveau comme l'édition d'objet ou la gestion de fichiers. Ces tâches sont laissées à la charge du programmeur. Pour manipuler une API de haut niveau exploitant OpenGL, le meilleur choix est d'écrire une application utilisant *Silicon Graphics Open Inventor*. Cette API propose en effet une architecture 3D orientée objets donnant accès à des fonctions de haut niveau. Cette API, de ce fait, est fortement conseillée pour les novices en programmation 3D et ceux qui préfèrent la structuration des programmes orientés objets.

Un point très important dans le développement de cette bibliothèque est que les cartes graphiques avec fonctions 3D câblées prennent désormais en charge au niveau matériel les quelques 250 commandes OpenGL. Jusqu'à récemment de telles cartes coûtaient très cher et étaient uniquement disponibles sur certaines stations de travail UNIX dont les Silicon Graphics. Actuellement, grâce à la volonté d'ouverture de SGI, toutes les cartes d'entrée de gamme supportent désormais en plus des commandes Direct3D, celles d'OpenGL. L'effort des constructeurs va même jusqu'à proposer des drivers pour Linux, preuve qu'ils croient à la fois au développement d'OpenGL et à celui de Linux. Pour les cartes à base de processeurs graphiques *Voodoo* de *3Dfx Interactive*, première famille de carte à avoir proposé un driver sous Linux, cette accélération se fait par l'intermédiaire de la librairie *Glide* qui sert d'interface vers les registres de la carte.

La Figure 25 nous montre d'une part l'architecture logicielle d'OpenGL et, d'autre part, l'ordre des opérations dans son pipeline graphique. Puisque l'architecture d'OpenGL est de type client-serveur, les calculs et l'affichage de la scène pourront soit être faits sur une même machine, soit répartis selon les possibilités des machines. Dans ce second cas, le pipeline graphique sera géré par le serveur qui recevra les données du client et lui renverra la scène calculée. C'est dans le pipeline graphique que le système détermine si une commande est accélérée par le matériel. Selon le cas,

l'exécution de cette commande sera aiguillée vers un module matériel ou un module d'émulation logicielle.

OpenGL compte deux modes de rendu différents : un mode immédiat où les commandes sont envoyées une à une au serveur que celui-ci exécute le plus rapidement possible et un mode retenu où les commandes sont stockées dans une liste et envoyées en bloc au serveur. Ce dernier mode compte deux avantages importants :

- pour afficher un objet complexe, il suffit de référencer la liste au lieu de redonner toutes les commandes,
- les informations sur un objet sont ainsi envoyées rapidement sur le réseau.

Le seul inconvénient de ce mode retenu est lorsque l'on souhaite modifier l'objet. Il faudra pour cela supprimer la liste actuelle et en créer une autre contenant l'objet modifié. Avec OpenGL, il est cependant possible de mélanger des commandes immédiates et des exécutions de liste de commandes, rendant ainsi possible la gestion des objets animés en mode immédiat et ceux fixes en mode retenu.

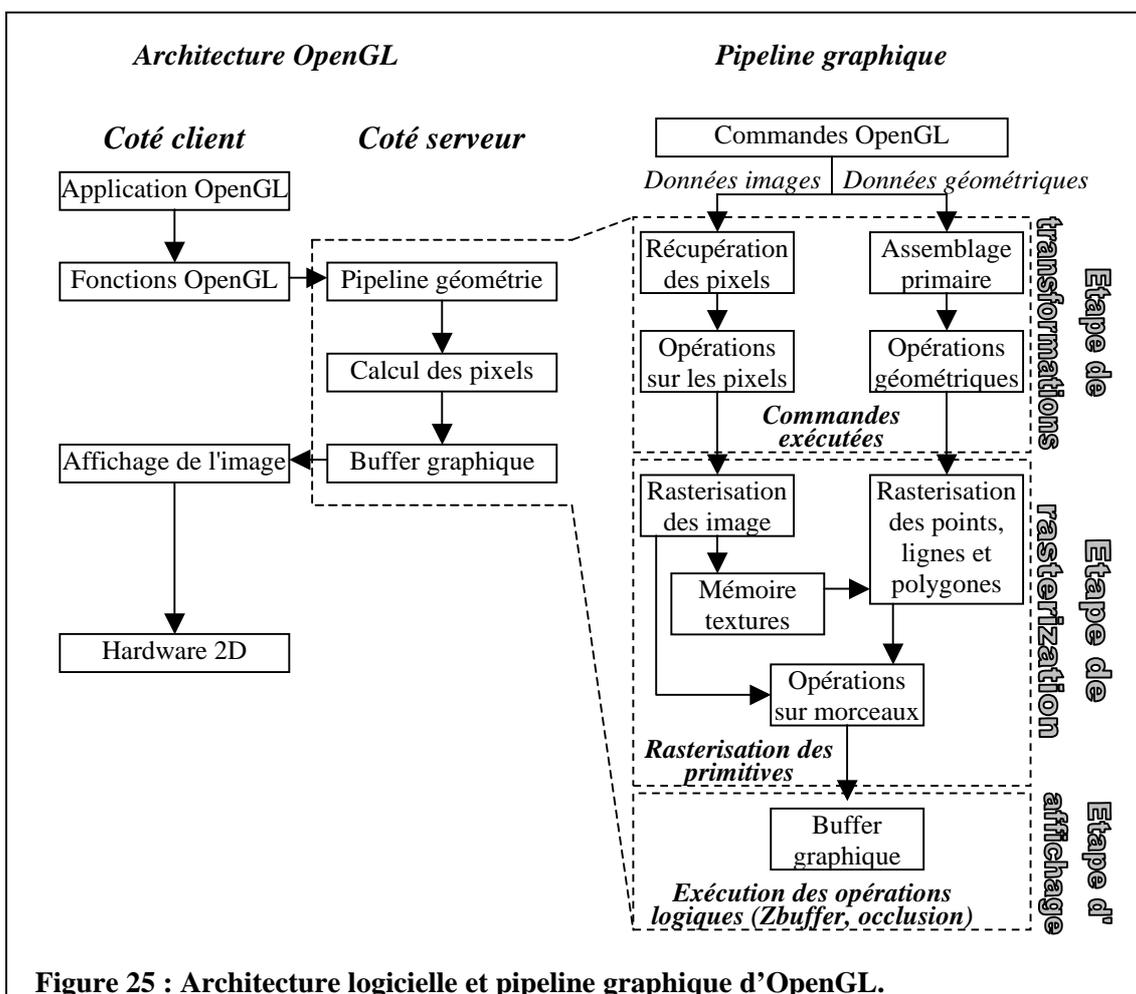


Figure 25 : Architecture logicielle et pipeline graphique d'OpenGL.

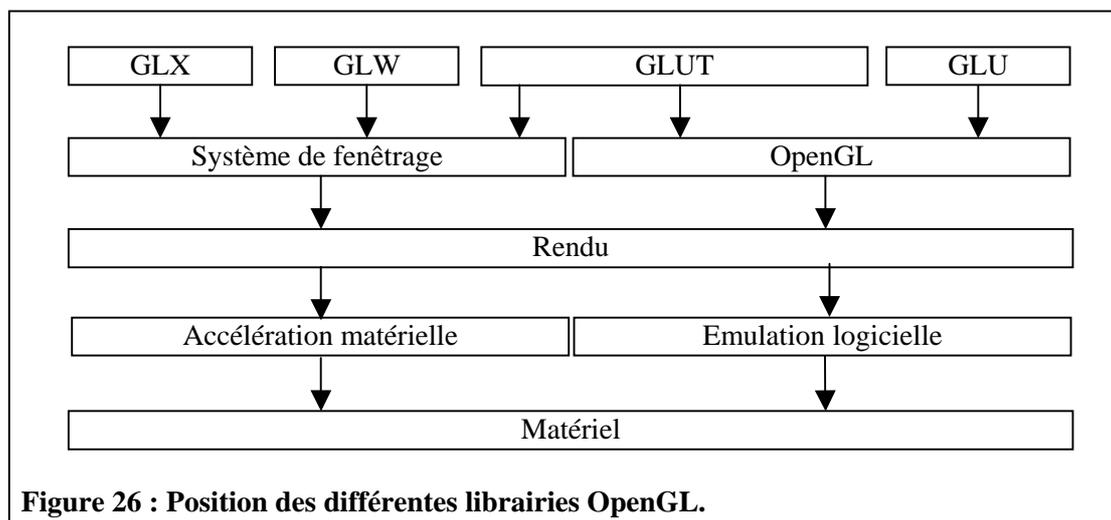
OpenGL semble pour toutes ces qualités une API de choix pour la création d'une application 3D. Les seuls points négatifs par rapport à ses concurrents est le manque d'architecture orientée objets et le fait que les textures utilisées doivent avoir des dimensions de 2^n pixels en largeur et de 2^m pixels en hauteur pour les raisons évidentes d'un rendu plus rapide car les images peuvent ainsi être parcourues pixel à pixel par décalages au lieu de multiplications plus coûteuses.

2.2.1 Les bibliothèques d'outils

Contrairement à l'ancienne API de SGI, IRIS-GL, OpenGL a été créée afin de donner accès à des fonctions graphiques indépendantes du matériel et donc indépendantes du système d'exploitation utilisé. C'est pourquoi, cette bibliothèque ne propose pas de fonctions permettant la gestion des fenêtres ; cette tâche étant dépendante du système d'exploitation utilisé et plus précisément de son interface graphique. Cependant, plusieurs bibliothèques d'utilitaires peuvent s'ajouter à OpenGL pour prendre en charge cet aspect (ces bibliothèques fonctionnent également avec Mesa) :

- La bibliothèque GLUT¹² (*OpenGL Utility Toolkit*), considérée comme la remplaçante de la bibliothèque AUX (*Auxiliary library*), gère de façon transparente par procédures de type *callback* les événements et le rendu dans des fenêtres. Elle donne la possibilité de créer des programmes pouvant être compilés et exécutés sous n'importe quel système d'exploitation possédant cette bibliothèque de fonctions. D'autre part, GLUT fournit des routines permettant de créer facilement des objets 3D complexes (des cylindres, des sphères, ...) qu'il n'est pas trivial de créer avec les quelques primitives géométriques OpenGL (points, lignes et polygones).
- La bibliothèque GLX (*OpenGL eXtension*) permet de lier un programme utilisant des fonctions OpenGL avec le système graphique XWindow.
- La bibliothèque GLW (*OpenGL extension for Windows*) est équivalente à GLX mais pour les plates-formes Win32. Elle est toutefois très peu par rapport à GLUT et AUX.

Il existe également la bibliothèque GLU (*OpenGL Utility Library*) qui comme la bibliothèque GLUT donne accès à des routines permettant de créer des objets 3D complexes. Cependant, GLU ne prend absolument pas en charge le système de fenêtrage. Les ensembles {GLX, GLU}, {GLW, GLU} ou {AUX, GLU} correspondront donc, à peu près, aux fonctionnalités de la bibliothèque GLUT seule. GLU est néanmoins plus complète que GLUT pour la création d'objets complexes puisqu'elle intègre, par exemple, la tassélation.



Pour le navigateur VRML que nous allons créer, la bibliothèque GLUT multi plates-formes convient très bien puisque le même code peut être compilé sous différentes architectures. Cependant, cette bibliothèque ne convient que pour le navigateur autonome pour lequel nous n'avons pas besoin de descripteur de fenêtre que GLUT ne fournit pas. En effet, pour écrire un *plugin*, nous devons fournir à Netscape Navigator, par l'intermédiaire de fonctions de son kit de développement, un pointeur vers la

¹² écrite par Mark J. Kilgard de SGI.

fenêtre ou les fenêtres de rendu. Il sera alors nécessaire d'utiliser GLX pour écrire un *plugin* Netscape sous XWindow et GLW ou AUX sous les plates-formes Win32. Pour avoir le *plugin* sur les deux plates-formes (Linux et Win32), il faudra donc deux programmes, l'un utilisant la librairie GLX et l'autre la librairie GLW ou AUX.

2.2.2 Squelette de code OpenGL

Les programmes OpenGL utilisant la librairie GLUT pour prendre en charge le système de fenêtrage quel que soit le système d'exploitation, auront toujours la même structure. Cette structure est donnée afin d'expliquer à quoi servent les différentes fonctions et pour montrer où nous devons ajouter le code créant les objets et le code pour les animations et les événements.

Avec GLUT

Le programme suivant explique comment utiliser la librairie OpenGL pour décrire des objets en trois dimensions et comment utiliser les fonctions de la librairie GLUT pour gérer le système de fenêtrage, les interactions et générer des objets complexes.

Cet exemple utilise le mode immédiat. Pour le mode retenu, il suffit d'initialiser une liste de commande (*display list*) dans la fonction *init* et de l'exécuter dans la fonction *display*. L'initialisation se fait tout d'abord par l'appel à la fonction *glGenLists* à qui l'on doit passer en argument le nombre de listes voulues. Puis il faut remplir les listes en faisant appel aux commandes (OpenGL ou GLUT ou GLU) entre les appels à la fonction *glNewList* (ayant comme paramètre le numéro de la liste à remplir) et à la fonction *glEndList*. L'utilisation de cette liste se fait dans la fonction *display* en appelant la commande *glCallList* avec le numéro de la liste à exécuter en paramètre. Un programme exemple utilisant une liste de commandes sera donné dans le paragraphe concernant GLX.

```

1 #include <GL/gl.h>                /* Déclaration des fonctions d'OpenGL */
2 #include <GL/glut.h>             /* Déclaration des fonctions de GLUT */
3
4 void init(void) {
5     glClearColor(0.0, 0.0, 0.0, 0.0); /* La couleur d'effacement */
6     glShadeModel(GL_FLAT);          /* Le style de rendu */
7 }
8
9 void display(void) {               /* Fonction dessinant chaque trame */
10  /* effacement du frame-buffer et du Z-buffer */
11  glClear(GL_COLOR_BUFFER_BIT, GL_CLEAR_DEPTH_BUFFER);
12  glColor3f(1.0, 1.0, 1.0);        /* La couleur pour dessiner */
13  /* Initialisation de la matrice de transformation */
14  glLoadIdentity();
15  /* Transformation des objets par rapport au point de vue */
16  glTranslatef(0.0, 0.0, (GLfloat) zoom);
17  /* dessin des objets */
18  glBegin(GL_POLYGON);
19  glVertexf(0.25, 0.25, 0.0);
20  glVertexf(0.75, 0.25, 0.0);
21  glVertexf(0.75, 0.75, 0.0);
22  glVertexf(0.25, 0.75, 0.0);
23  glEnd();
24  glutSolidSphere(1.0, 16, 16);
25  glSwapBuffers();                 /* Affichage de l'image calculée */
26 }
27
28 void reshape(int w, int h) {      /* Fonction de redimensionnement */
29  glViewport(0, 0, w, h);
30  glMatrixMode(GL_PROJECTION);
31  glLoadIdentity();
32  gluPerspective(45.0, w/h, 1.0, 20000.0);

```

```

33  glMatrixMode(GL_MODELVIEW);
34  glLoadIdentity();
35 }
36
37 int main(int argc, char **argv) {
38  glutInit(&argc, argv);
39  glutInitMode(GLUT_DOUBLE, GLUT_RGB);
40  glutInitWindowSize(500, 500);
41  glutInitWindowPosition(100, 100);
42  glutCreateWindow(argv[0]);
43  init();
44  glutDisplayFunc(display);
45  glutReshapeFunc(reshape);
46  glutMainLoop();
47  return 0;
48 }

```

Afin d'utiliser les fonctions et constantes des bibliothèques OpenGL et GLUT, nous devons déclarer l'utilisation des fichiers entêtes "GL/gl.h" et "GL/glut.h" (lignes 1 et 2). Pour la compilation, les bibliothèques dynamiques devront être liées pour que les symboles soient résolus et non seulement déclarés.

La fonction *main* (ligne 37) qui est appelée lorsque l'on exécute le programme, initialise tout d'abord la fenêtre de rendu à l'aide des cinq fonctions de la bibliothèque GLUT créées à cet effet. Les appels à ces fonctions se situent entre la ligne 38 et la ligne 42. Les fonctions effectuent les opérations suivantes :

- *glutInit(int *argc, char **argv)* initialise GLUT et prend en charge les options passées en paramètre (par exemple pour XWindow, cela peut être une option *-display* ou *-geometry*). Cette fonction doit toujours être appelée avant toute autre fonction GLUT.
- *glutInitDisplayMode(unsigned int mode)* spécifie, à l'aide de constantes GLUT, si l'on veut utiliser un modèle de couleur RGBA (triplet RGB plus une valeur Alpha de transparence) ou un modèle de couleurs indexées (par palette). Dans le cas où l'on souhaiterait utiliser le deuxième, il faudra remplir la palette à l'aide d'appels à la fonction *glutSetColor* qui définit une couleur. Dans la fonction *glutInitDisplayMode*, on peut également spécifier si l'on souhaite utiliser une fenêtre avec un double *buffer* pour les animations ou un simple *buffer* pour un seul rendu. Enfin, cette fonction permet de spécifier si l'on veut un *buffer* pour la profondeur (un tampon qui garde en mémoire la profondeur de chaque pixel, le *Z-Buffer*), un *stencil buffer* (un tampon qui garde en mémoire les portions de l'écran sur lesquelles le rendu ne doit pas être effectué), un *buffer* d'accumulation (généralement utilisé pour les opérations d'*anti-aliasing*).
- *glutInitWindowPosition(int x, int y)* spécifie la position de la fenêtre de rendu.
- *glutInitWindowSize(int width, int height)* spécifie la taille de la fenêtre de rendu.
- *glutCreateWindow(char *string)* crée la fenêtre de rendu dont le nom est passé par l'argument *string*. La fonction renvoie un identificateur unique de type entier pour la fenêtre nouvellement créée. Tant que l'appel à la fonction *glutMainLoop*, que nous expliquerons dans un instant, n'a pas été effectué, la fenêtre ne sera pas visible.

A la ligne 43 (dans la fonction *main*), après les initialisations nécessaires de la bibliothèque GLUT, nous faisons appel à une fonction d'initialisation d'OpenGL. Cette fonction *init* (lignes 4 à 7) définit la couleur d'effacement de la fenêtre de rendu entre chaque trame (la couleur du fond de la scène) à l'aide de la fonction *glClearColor* qui prend en paramètres quatre flottants définissant la valeur du

quadruplet RGBA. La deuxième commande d'initialisation d'OpenGL sert à définir le type de rendu que l'on souhaite, à l'aide de la fonction *glShadeMode* qui prend en paramètre une constante OpenGL correspondant à un type de rendu. Dans notre cas, nous avons spécifié un rendu de type FLAT (c'est-à-dire que la géométrie aura une couleur uniforme parce que la couleur d'un sommet est dupliquée pour les autres sommets), nous utilisons donc la constante *GL_FLAT* correspondante. L'autre valeur possible est *GL_SMOOTH* qui correspond au cas où la couleur de chaque sommet de la géométrie est traitée individuellement et la couleur des points intermédiaires est interpolée en fonction de celle des sommets.

Les lignes 44 à 46 définissent les fonctions de type *callback* de GLUT qui prennent en charge les événements. Généralement, ces fonctions servent à programmer les interactions possibles avec la scène. La mise à jour de la scène après une interaction se fait par la fonction *glutPostRedisplay*. Si l'on veut que la scène soit dessinée, les deux fonctions de type *callback* qu'il est obligatoire de faire apparaître dans un programme utilisant GLUT sont :

- *glutDisplayFunc(void(*func)(void))* dont le paramètre est une fonction sans argument, spécifie à GLUT la fonction à appeler pour effectuer le rendu (*display* dans notre exemple). Dès que GLUT juge qu'il est nécessaire de redessiner la fenêtre il doit appeler la fonction de rendu. Il est donc obligatoire de lui fournir.
- *glutMainLoop(void)* fait apparaître toutes les fenêtres créées et le rendu vers ces fenêtres devient effectif. L'appel à cette fonction fait également démarrer l'écoute des événements.

D'autres fonctions de gestion des événements peuvent être ajoutées mais ne sont pas obligatoires :

- *glutReshapeFunc(void(*func)(int w, int h))* dont le paramètre est une fonction avec deux arguments de type entier, spécifie à GLUT la fonction qu'il doit appeler lorsque la taille de la fenêtre est modifiée. Cette fonction est utilisée dans notre petit exemple ligne 44.
- *glutKeyboardFunc(void(*func)(unsigned char key, int x, int y))* dont le paramètre est une fonction avec trois paramètres (l'un récupérant la touche du clavier frappée, les deux autres les coordonnées de la souris lors de la frappe), spécifie à GLUT la fonction à appeler lorsqu'un événement clavier est reçu.
- *glutMouseFunc(void(*func)(int button, int x, int y))* dont le paramètre est une fonction avec trois paramètres (l'un récupérant le bouton de la souris appuyé, les deux autres les coordonnées de la souris lors du clic de souris), spécifie à GLUT la fonction à appeler lorsqu'un événement souris est reçu.
- *glutMotionFunc(void(*func)(int x, int y))* dont le paramètre est une fonction avec deux paramètres donnant les coordonnées de la souris, spécifie à GLUT la fonction à appeler lorsque la souris est en mouvement et qu'un de ses boutons est appuyé.

Les fonctions utilisées dans la fonction *display* qui effectuent le rendu sont principalement des fonctions de la librairie OpenGL. Ces fonctions définissent les points, les normales, les coordonnées pour appliquer les textures, les couleurs, etc. Toutes ces fonctions définissant des options de rendu doivent être appelées entre un appel à la fonction *glBegin* qui repère le début d'une série de commandes OpenGL et un appel à la fonction *glEnd* qui en marque la fin. Les fonctions valides sont au nombre de 24¹³. Au lieu de décrire les formes géométriques plus compliquées à l'aide de ces 24 fonctions de base, nous pouvons utiliser des fonctions GLU ou GLUT de plus haut niveau. Par exemple une fonction de la librairie GLUT pour tracer un cube en fil de fer est *glutWireCube* ou pour

¹³ On peut trouver toutes ces fonctions page 46 du « red book » [OpenGLa].

tracer une sphère pleine *glutSolidSphere*. Dans ce cas, nous n'avons pas besoin d'appeler *glBegin* et *glEnd*. Dans les deux cas, nous utilisons cependant les routines de GLUT pour la gestion des événements et des fenêtres.

La fonction *reshape* est appelée automatiquement par les fonctions de GLUT lorsque l'utilisateur agrandit ou diminue la fenêtre de rendu. Cette fonction définit la position de la caméra, son angle de vue ou le type de projection (parallèle ou perspective) en fonction de la taille de la fenêtre.

Avec GLX ou GLW

Comme nous l'avons dit précédemment, pour utiliser OpenGL dans un *plugin* Netscape, il faut fournir à ce dernier un descripteur de fenêtre (*handle*) dans laquelle le *plugin* effectue ses sorties écran. GLUT ne pouvant nous fournir cette information car tout est géré avec ses propres descripteurs de fenêtres et non avec ceux du système de fenêtrage, nous utilisons donc GLX ou GLW.

Ces deux bibliothèques fonctionnent de la même manière puisque GLW est un clone de GLX pour les plates-formes Win32. Avec ces bibliothèques, les événements sont récupérés de la même manière que si l'on programmait une interface 2D ; c'est-à-dire pour GLX et XWindow par exemple, dans une boucle infinie comme le montre le code suivant.

Le programme effectue la même tâche que celui utilisant GLUT : tracer un polygone et une sphère. Les différences avec le programme précédent sont :

- l'utilisation de la bibliothèque GLX pour gérer l'interface graphique XWindow,
- l'utilisation d'une liste de commandes contenant la scène à afficher.

On utilise toujours la fonction *glutSolidSphere* de la bibliothèque GLUT pour tracer la sphère.

```
1 #include <X11/Xlib.h>
2 #include <GL/glx.h>
3 #include <GL/glut.h>
4
5 /* Initialisation de GLX : couleurs 32bits, Zbuffer 16bits et double
   buffering */
6 static int attributeList[] = { GLX_RGBA, GLX_DEPTH_SIZE, 16,
7 GLX_DOUBLEBUFFER, None };
8
9 /* variable contenant le numéro de la liste de commandes */
10 GLuint theScene;
11
12 [...]
13
14 void reshape(int w, int h) {
15     glViewport(0, 0, w, h);
16     glMatrixMode(GL_PROJECTION);
17     glLoadIdentity();
18     gluPerspective(45.0, w/h, 1.0, 20000.0);
19     glMatrixMode(GL_MODELVIEW);
20     glLoadIdentity();
21 }
22
23 void draw_scene(Display *dpy, Window win) {
24     /* Effacement des pixels et du Zbuffer */
25     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
26     /* Effectue les transformations nécessaires */
27     glPushMatrix();
28     glTranslatef(posx, posy, posz);
```

```

29  glRotatef(rot_y+spin_y, -1.0, 0.0, 0.0);
30  glRotatef(rot_x+spin_x, 0.0, -1.0, 0.0);
31  /* Appel de la liste de commandes à dessiner */
32  glCallList(theScene);
33  glPopMatrix();
34  /* Affichage du buffer rempli */
35  glXSwapBuffers(dpy, win);
36 }
37
38 int main(int argc, char **argv) {
39  Display *dpy;
40  XVisualInfo *vi;
41  Colormap cmap;
42  XSetWindowAttributes swa;
43  Window win;
44  GLXContext cx; XEvent event;
45
46  /* Initialisation du client X */
47  dpy = XOpenDisplay(0);
48  vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributeList);
49  /* création d'un contexte X */
50  cx = glXCreateContext(dpy, vi, 0, GL_TRUE);
51  /* Création de la fenêtre de rendu */
52  cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen), vi->visual,
AllocNone);
53  swa.colormap = cmap; swa.border_pixel = 0;
54  swa.event_mask = StructureNotifyMask;
55  win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 100, 100,
0, vi->depth, InputOutput, vi->visual,
CWBorderPixel|CWColormap|CWEventMask, &swa);
56  XSelectInput(dpy, win, ExposureMask | ButtonPressMask |
ButtonReleaseMask | ButtonMotionMask | StructureNotifyMask);
57  XMapWindow(dpy, win);
58  /* Connexion du contexte GLX à cette fenêtre */
59  glXMakeCurrent(dpy, win, cx);
60  /* Création et remplissage d'une liste de commandes */
61  theScene = glGenLists(1);
62  glNewList(theScene, GL_COMPILE);
63  glColor3f(1.0, 0.0, 0.0);
64  glBegin(GL_POLYGON);
65  glVertex3f(0.25, 0.25, 0.0);
66  glVertex3f(0.75, 0.25, 0.0);
67  glVertex3f(0.75, 0.75, 0.0);
68  glVertex3f(0.25, 0.75, 0.0);
69  glEnd();
70  glPushMatrix();
71  glTranslatef(2.0, 0.0, 0.0);
72  glColor3f(0.0, 1.0, 0.0);
73  glutSolidSphere(1.0, 16, 16);
74  glPopMatrix();
75  glEndList();
76  /* Initialisations OpenGL */
77  glEnable(GL_DEPTH_TEST);
78  glDepthFunc(GL_LESS);
79  glClearDepth(1.0);
80  glClearColor(0.0, 0.0, 0.0, 0.0);
81  reshape(100, 100);
82  /* Traitement des événements dans une boucle infinie */
83  while (1) {
84      XNextEvent(dpy, &event);
85      switch (event.type) {

```

```

86     case ButtonPress:           /* bouton de souris appuyé */
87         [...]
88         break;
89     case ButtonRelease:        /* bouton de souris relâché */
90         [...]
91         break;
92     case MotionNotify:         /* souris déplacée */
93         [...]
94         needRedraw = GL_TRUE;
95         break;
96     case ConfigureNotify:      /* changement de taille */
97         reshape(event.xconfigure.width, event.xconfigure.height);
98         needRedraw = GL_TRUE;
99         break;
100 }
101 if (needRedraw) {
102     needRedraw = GL_FALSE;
103     draw_scene(dpy, win);
104 }
105 }
106 }
107

```

2.2.3 Avantages et inconvénients de l'utilisation d'OpenGL

OpenGL comporte comme nous l'avons vu quelques avantages de taille :

- son modèle client-serveur permettant d'effectuer des rendus de scène sur un ordinateur ayant une puissance de calculs insuffisante mais disposant d'un affichage rapide,
- sa conception multi plate-forme indépendante du système d'exploitation et l'association avec la librairie GLUT autorisent le programmeur à produire le même code pour différentes familles de machines,
- ses listes de commandes utilisent peu de mémoire et par conséquent se transmettent et s'exécutent très vite,
- l'équivalent d'une commande OpenGL est trois ou quatre appels de fonctions pour les autres API bas niveau. Le code OpenGL est donc plus petit et les applications s'écrivent plus vite.

Par rapport aux autres librairies que nous présentons, OpenGL compte cependant quelques aspects qui peuvent paraître des inconvénients :

- pas de paradigme de programmation orientée objets et par conséquent pas de structuration des programmes qui va avec,
- pas de fonctions de haut niveau permettant de créer des objets complexes sans appeler des fonctions d'autres bibliothèques,
- accélération matérielle douteuse : les cartes graphiques d'entrée de gamme même si elles sont supposées accélérer les fonctions OpenGL semblent ne pas les accélérer autant que celles de Direct3D et ne donnent, en tous cas, aucune spécification claire.

Cependant, certains aspects négatifs peuvent devenir, lorsqu'on les regarde de plus près, des avantages plus que des inconvénients :

- taille des textures ($2^n * 2^m$) : obligation de déformer les textures n'ayant pas la bonne taille

mais accélération du rendu grâce à l'utilisation de décalages de bits au lieu de multiplications pour parcourir les pixels de l'image ; il est vrai, cependant, que se soucier de telles petites optimisations paraît inutile sur les machines actuelles,

- le manque de fonctions permettant par exemple le chargement des fichiers textures permet au programmeur un plus grand contrôle des opérations à effectuer et particulièrement la mise à l'échelle des textures.

En conclusion, OpenGL semble être une bibliothèque de fonctions 3D idéale pour les programmeurs ayant une expérience dans le domaine. Elle permet le plus grand contrôle sur les options de rendu et également une gestion plus facile des animations dans le mode immédiat.

2.3 Direct3D

Direct3D est l'API 3D de Microsoft. C'est un des derniers ajouts à la famille d'API multimédia pour les plates-formes Win32, DirectX, actuellement à la version 8.0. Pour Windows 95 et 98, ces bibliothèques dynamiques (DLL) font parties des fichiers installés alors que pour Windows NT, la version 3.0 est disponible et elle fait partie du *Service Pack 5* de Microsoft. La version 8.0 a fait son apparition dans Windows 2000, le système d'exploitation unique succédant à Windows NT et Windows 98.

Cette librairie 3D a été conçue principalement pour les développeurs de jeux. En effet, elle remplace l'ancienne librairie WinG (*Windows Games*) qui fournissait uniquement les DIB (*Device Independant Bitmap*) capables d'être affichés rapidement (et donc utile pour le double buffering et les animations) chose que la GDI Windows n'est pas capable de faire. En aucun cas, WinG ne fournissait de fonctions 3D ; le pipeline 3D entier devait alors être pris en charge par le programmeur, rendant le développement d'une application 3D très coûteuse et ne tirant pas partie des possibilités du matériel sans programmation explicite de chaque composant.

L'important développement de cette librairie 3D est lié à l'adhésion des constructeurs de cartes graphiques 3D qui ont rapidement livré leur matériel avec des drivers Direct3D. Ainsi, non seulement Direct3D donne accès à des fonctions 3D gérant les différentes étapes du pipeline 3D mais en plus ces fonctions sont accélérées par le matériel si celui-ci en est capable.

Cette propriété, bien qu'existant également pour les librairies OpenGL et Quickdraw3D, est plus importante avec Direct3D vu le nombre de matériel supporté. La prise en charge du matériel sous Direct3D se fait par une couche d'abstraction matérielle HAL (*Hardware Abstraction Layer*). Cette couche permet au programmeur d'obtenir des informations et caractéristiques sur le matériel présent et également les performances de celui-ci. Parce que certains systèmes n'ont pas de matériel accélérant certaines fonctions 3D, une couche d'émulation HEL (*Hardware Emulation Layer*) prend en charge ces opérations particulières. Ainsi, toutes les applications peuvent bénéficier des apports de Direct3D, même si les performances s'en ressentent. L'architecture d'une application utilisant Direct3D ainsi que le pipeline graphique sont schématisés dans la Figure 27.

Comme les autres API, Direct3D propose un mode immédiat et un mode retenu pour gérer les opérations. Le mode retenu ressemble à celui de Quickdraw3D dans le sens où il offre une interface de haut niveau orientée objets pour la manipulation des objets graphiques. Dans ce mode, des fonctions permettent, en particulier, de charger et de sauvegarder des fichiers comportant des données sur les objets, sur les textures ou sur les animations.

Le mode immédiat de Direct3D peut se concevoir de deux manières différentes. Le premier est d'envoyer les commandes qui sont traitées une à une. Le second peut être considéré comme un mode retenu bis puisqu'il ressemble à s'y méprendre au mode retenu d'OpenGL. En effet, il utilise des listes de commandes comme celles d'OpenGL. Même si le nom de ces listes diffère (*display list* pour OpenGL contre *execute buffer* pour Direct3D) leur fonctionnement est néanmoins le même. A

l'initialisation, les différentes commandes (les polygones et la façon dont ils doivent être traités) sont placées dans un tampon. Lors du rendu, le programme référence ce tampon au lieu de devoir spécifier toutes les commandes qu'il contient. La différence majeure entre les deux types de liste est leur taille. Celle d'OpenGL utilise moins de mémoire et se transmet facilement d'un client à un serveur alors que celle de Direct3D a tendance à saturer la mémoire vidéo.

Comme dans les modes immédiats de la plupart des API présentées, celui de Direct3D ne fournit aucune fonction permettant de créer des objets complexes ou de charger ou sauvegarder un fichier. Ces opérations, dans ce mode, sont à la charge de l'application et plus précisément du programmeur. Il est donc particulièrement adapté aux programmeurs ayant leur propre moteur 3D mais qui souhaitent accéder à une accélération matérielle.

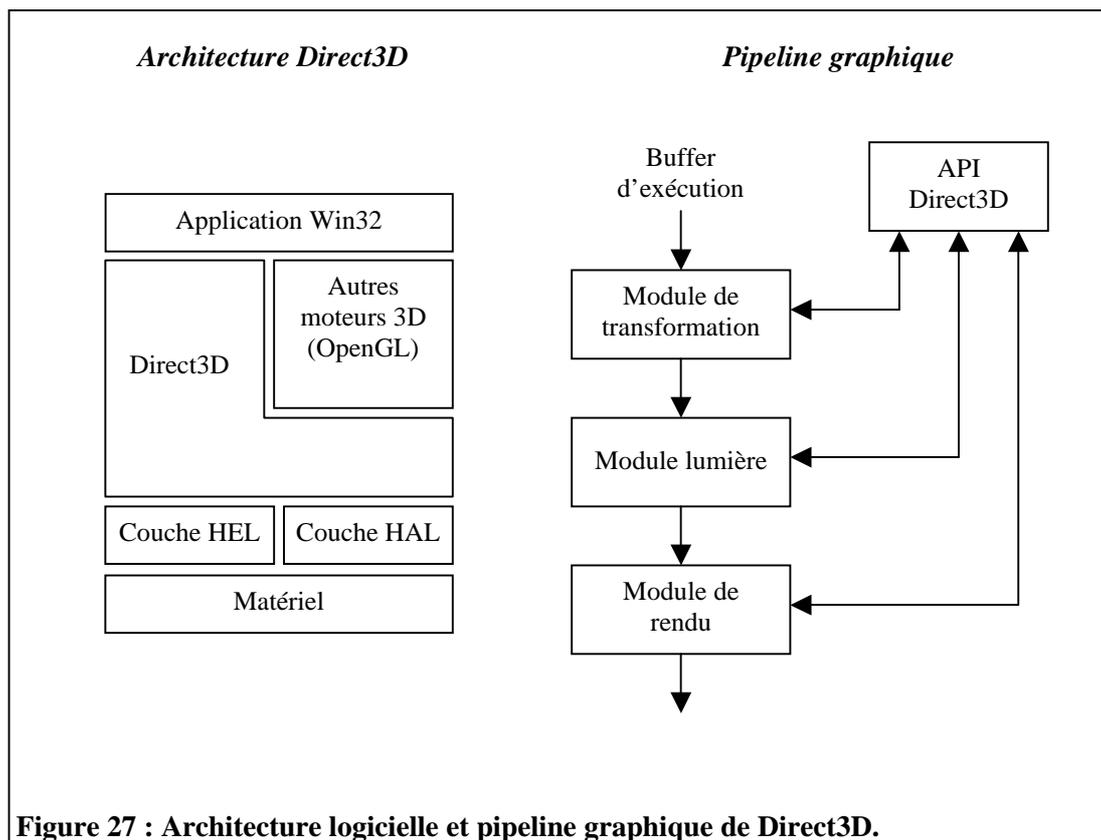


Figure 27 : Architecture logicielle et pipeline graphique de Direct3D.

Le pipeline graphique de Direct3D attend, en entrée, un *buffer* d'exécution comme ceux mis en place dans le mode immédiat. Lorsque le programmeur travaille en mode retenu, la hiérarchie des objets constituant la scène est alors traduite en une liste d'exécution pour être passée au pipeline. Cette liste de commandes passe alors par différents modules effectuant, ensemble, le rendu 3D complet. Chacun de ces trois modules que nous allons présenter est indépendant des autres. Ils sont appelés indépendamment depuis l'application Direct3D. Ainsi, le programmeur peut choisir d'utiliser ou non un de ces modules, lui préférant peut-être un module spécial qu'il aura écrit pour gérer certaines optimisations. Cependant, les modules standards sont particulièrement bien optimisés et donc la plupart du temps ce sont eux qui seront utilisés.

- Le module de transformation est le point de départ du pipeline. Il transforme, en fonction de la caméra, les polygones dans le buffer d'exécution. Ce module détermine aussi quels sommets sont dans la vue (*clipping*), éliminant ceux d'entre eux qui n'y sont pas (ceux qui sont invisibles) et qui ne doivent donc pas subir le rendu final. Cette étape revêt par ce dernier aspect une importance capitale dans l'obtention des meilleures performances possibles pour l'ensemble du pipeline.

- Le module gérant les lumières garde une trace de toutes les lumières présentes dans la scène et détermine en fonction d'elles la couleur de chacun des sommets des polygones. Il existe en fait deux modules différents pour gérer les lumières :
 - le module *RGB* où les lumières peuvent être colorées puisqu'elles sont spécifiées par les trois canons rouge - vert – bleu. Ce mode offre bien sûr un rendu réaliste mais est également notablement moins performant si les lumières RVB ne sont pas prises en charge par le matériel.
 - Le module *Ramp* permet d'utiliser des lumières monochromes. Ce mode est évidemment moins réaliste mais plus économique en terme de puissance. Cependant, pour une scène comportant uniquement des lumières blanches ou grises, il convient parfaitement.

Mais si le matériel supporte les couleurs RVB, il est préférable d'utiliser ce mode puisque les fonctions accélérées sont plus rapides que les fonctions émulées. Enfin, une dernière alternative est de définir un module spécial pour les applications dont le centre d'intérêt est un modèle de lumière particulièrement pointu.

- Le module de rendu (*rasterization*) affiche la scène à partir des sommets calculés par le module de transformation et des lumières du module lumière. Il existe un grand nombre d'options pour obtenir le rendu souhaité.

2.3.1 Création d'une application

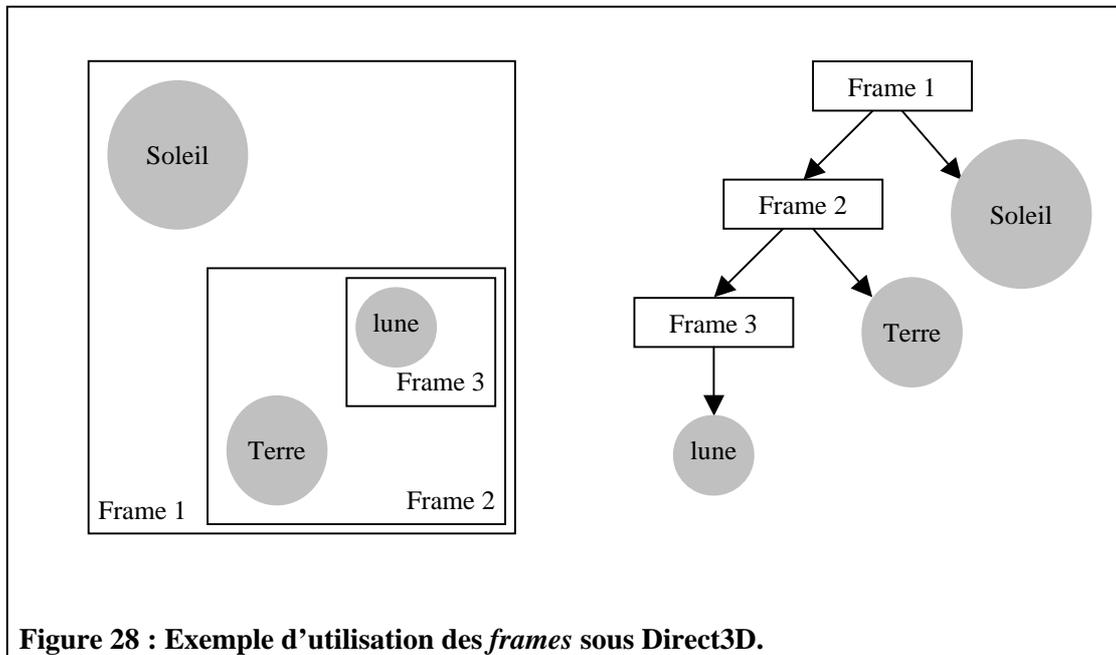
Avant de commencer, précisons que pour Direct3D le système de coordonnées est "main gauche" contrairement au repère d'OpenGL, de VRML ou de QuickDraw3D. La seule différence entre ces deux repères est l'inversion de l'axe Z (Voir la Figure 9 page 18).

En mode retenu

Pour créer une scène avec le mode retenu de Direct3D, il est nécessaire d'initialiser différents objets ou composants qui serviront, ensemble, à construire, à transformer et à afficher la scène :

- L'objet Direct3D sert à créer et à détruire tous les autres objets nécessaires.
- Le dispositif de visualisation de Direct3D ressemble à un objet DirecDraw¹⁴ plus qu'à un objet Direct3D. Il représente la destination du rendu ou la surface d'affichage. Les différentes options de rendu sont spécifiées à l'aide de cet objet. On peut utiliser plusieurs objets de ce type pour les applications gérant différentes scènes ou plusieurs vues à la fois.
- Les *frames* sont simplement des entités qui donnent les transformations à opérer sur les objets contenus. Le terme *frame* peut paraître déconcertant car ce n'est pas, comme l'on peut si attendre, d'un objet graphique, que l'on peut voir ou manipuler comme en HTML. Il s'agit d'un conteneur d'objets ou d'autres *frames* qui subissent les mêmes transformations. C'est avec elles que se met en place une arborescence d'objets (comme avec les objets *Group* de Java3D). La Figure 28 montre un exemple d'utilisation des *frames* pour simuler les orbites de la lune autour de la Terre, et de la Terre autour du soleil. Dans cet exemple, on applique à la *frame 2* une rotation automatique par rapport à la *frame 1* pour simuler l'orbite de la Terre autour du Soleil. De la même manière, la *frame 3* tournera autour de la *frame 2* avec une vitesse de rotation différente. La lune subira donc les deux rotations à la fois.

¹⁴ API graphique 2D de la suite DirectX.



- L'objet *Viewport* définit comment et où la scène sera affichée. Cet objet tient essentiellement le rôle de fenêtre par laquelle est vue la scène. Il suffit pour cela de lui donner une position et une orientation et de l'attacher à un dispositif de visualisation dans lequel la scène est affichée.

Pour Direct3D, l'initialisation est la moitié du travail. Nous devons créer une instance de chacun des objets vus précédemment. Cela n'est pas difficile mais prend un temps certain. Cependant puisque cette procédure est toujours la même, elle pourra être utilisée dans tous les autres programmes Direct3D par simple copier-coller avec changements des paramètres de rendu si nécessaire.

Les étapes pour créer une application Direct3D sont les suivantes. Chacune des fonctions données renvoie D3DRM_OK si tout c'est bien passé. Les lignes de code sont données à titre indicatif pour avoir une idée du travail à effectuer pour créer une scène. Tous les objets et méthodes utilisés sont référencés dans la documentation en ligne du mode retenu de Direct3D [D3Da].

1. Création de l'interface Direct3DRM (*Direct3D Retained Mode*) contenant tous les objets :
`Direct3DRMCreate(&pD3DRetainedModeAPI);`
2. Création de la *frame* racine contenant toute la scène :
`pD3DRetainedModeAPI->CreateFrame(NULL, &pD3DSceneAPI);`
3. Création et positionnement de la caméra :
`pD3DRetainedModeAPI->CreateFrame(pD3DSceneAPI, &pD3DCameraAPI);`
`pD3DCameraAPI->SetPosition(pD3DSceneAPI, 0.0f, 0.0f, 0.0f);`
4. Création du clipper et liaison avec la fenêtre de rendu :
`DirectDrawCreateClipper(0, &pDDClipperAPI, NULL);`
`pDDClipperAPI->SetHWND(0, hWnd);`
5. Création du dispositif de visualisation et du *Viewport* :
`pD3DRetainedModeAPI->CreateDeviceFromClipper(pDDClipperAPI, NULL,`
`width, height, &pD3DRMDeviceAPI);`
`pD3DRetainedModeAPI->CreateViewport(pD3DRMDeviceAPI, pD3DCameraAPI,`
`0, 0, pD3DRMDeviceAPI->GetWidth(), pD3DRMDeviceAPI->GetHeight(),`
`&dD3DViewportAPI);`

6. Initialisation de la profondeur jusqu'à laquelle les objets graphiques sont affichés :
`pD3DViewportAPI->SetBack(5000.0f);`
7. Construction de la scène.

La scène se construit également très rapidement en connaissant la liste des sommets formant les polygones. Les sommets ainsi que les normales doivent être stockés dans un tableau de flottants. Les trois premiers flottants de ces tableaux correspondront à chacune des coordonnées du premier point ou vecteur, puis les trois suivants le deuxième, etc.

Les faces, constituées de ces sommets, sont stockées dans un tableau d'entiers. Ces entiers représentent les index des différents éléments que l'on doit récupérer dans les tableaux de flottants. Pour chacune des faces, le premier entier représente le nombre de sommets. L'entier suivant correspond à l'index dans le tableau des sommets du premier sommet constituant le polygone et le deuxième entier est l'index du vecteur normal à ce sommet. Les deux suivants sont les indexes du deuxième sommet et du deuxième vecteur normal, etc. Si le tableau des faces est correctement construit alors, lorsque l'on a parcouru, pour une face, le nombre de sommets qui était donné par le premier entier, alors on doit se trouver sur le nombre de sommets de la face suivante. Ce principe est schématisé dans la Figure 29.

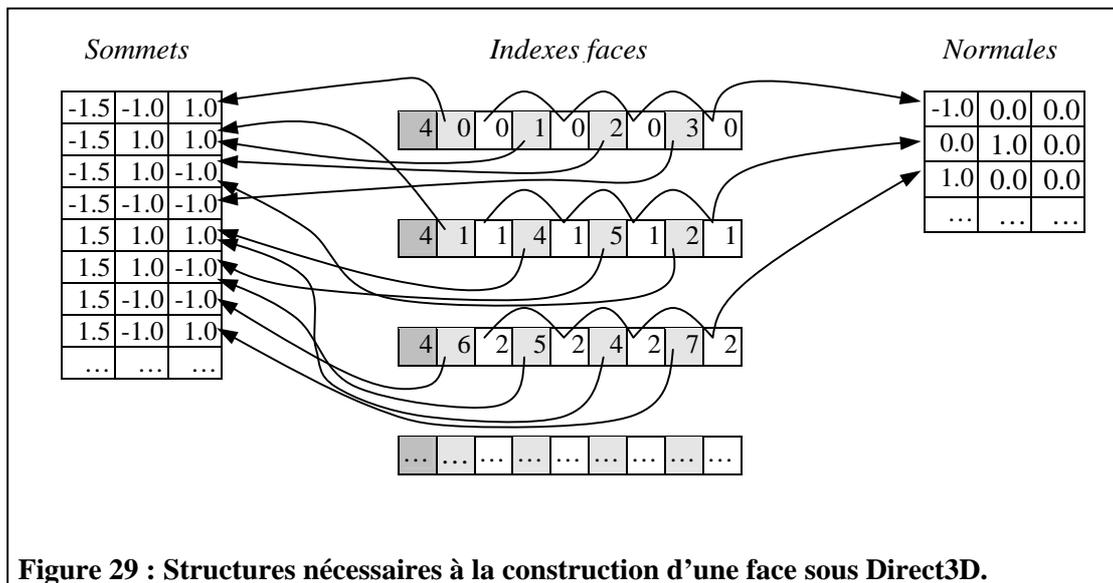


Figure 29 : Structures nécessaires à la construction d'une face sous Direct3D.

Dans le cas où nous n'avons pas besoin de vecteurs normaux (pas de gestion de lumière, pas de rendu avec un coloriage de type gouraud) alors il suffit d'omettre les index des vecteurs normaux dans le tableau des faces et de ne pas spécifier de tableau des normales lors de l'appel à la méthode *AddFaces* de l'objet *MeshBuilder*.

Un autre cas particulier concerne l'utilisation ou non d'une texture pour habiller une face. Avec Direct3D, nous sommes obligés de dupliquer les sommets et les normales dans les tableaux de flottants afin qu'il n'y ait pas deux mêmes valeurs dans le tableau des index. De cette manière, les coordonnées de texture pourront être associées à un seul sommet d'une seule face. L'exemple accompagnant les différentes étapes de la construction d'une scène ne gère pas les textures. Pour cela, il aurait fallu dupliquer les sommets et normales pour qu'ils ne servent qu'à une seule face, charger une texture, l'appliquer à notre objet *MeshBuilder* et définir les coordonnées de texture.

Les différentes étapes pour construire un objet sont les suivantes :

1. Définition des options de rendu (couleur du fond, type de rendu, lumières, ...) :

```

PD3DretainedMode->CreateLightRGB(D3DRMLIGHT_DIRECTIONAL, 0.4f,
0.4f, 0.4f, &pDirectionnal);
pD3DSceneAPI->AddLight(pDirectionnal);
pD3DSceneAPI->SetSceneBackground(D3DRGB(0, 0, 0));
pD3DRMDeviceAPI->SetQuality(D3DRMLIGHT_ON, D3DRMFILL_SOLID,
D3DRMSHADE_GOURAUD);

```

2. Création de l'objet *MeshBuilder* qui renferme les polygones à afficher :

```
pD3DretainedModeAPI->CreateMeshBuilder(&pMeshBuilder);
```
3. Chargement des polygones à partir du tableau de sommets, du tableau de normales et du tableau d'index :

```
pMeshBuilder->AddFaces(23, vertices, 23, normales, (unsigned
long*)indexesFace, NULL);
```
4. Choix de la couleur de l'objet :

```
pMeshBuilder->SetColor(D3DRGB(0, 0, 0));
```
5. Création de la *frame* qui contiendra l'objet que l'on veut ajouter :

```
pD3DretainedModeAPI->CreateFrame(pD3DSceneAPI, &pFaceFrame);
```
6. Ajout de l'objet *MeshBuilder* à la *frame* :

```
pFaceFrame->AddVisual(pMeshBuilder);
```
7. Positionnement et orientation¹⁵ de la nouvelle *frame* par rapport à sa *frame* parent :

```
pD3DCameraAPI->SetPosition(pD3DSceneAPI, 0.0f, 0.0f, -7.0f);
pD3DCameraAPI->SetOrientation(pD3DSceneAPI, 0.0f, 0.0f, 1.0f, 0.0f,
1.0f, 0.0f);
```
8. Libérations des interfaces que l'on n'utilisera plus : celle de la *frame* et l'objet *MeshBuilder* :

```
pFaceFrame->Release();
pMeshBuilder->Release();
```

La gestion des événements et, en particulier, des interactions de l'utilisateur avec la scène se fait, comme pour tout programme Windows, grâce à la définition de fonctions de type *callback* après avoir énuméré les cas que l'on veut gérer :

- WM_PAINT pour la mise à jour de la scène,
- WM_SIZE pour le changement de la taille de la fenêtre de rendu,
- WM_KEYDOWN pour la récupération des touches du clavier appuyées,
- WM_LBUTTONDOWN, WM_LBUTTONUP, WM_RBUTTONDOWN, WM_RBUTTONUP, WM_MOUSEMOVE pour gérer les boutons et mouvements de la souris.

Lorsque le programme est fermé (récupération de l'événement WM_DESTROY) il faut libérer tous les objets créés (et non déjà libérés) à l'aide de leur méthode *Release*.

En mode immédiat

Le mode immédiat de Direct3D est un peu plus difficile à utiliser que le mode retenu. Cependant, comme nous l'avons dit, ses possibilités peuvent être intéressantes pour un programmeur d'une application 3D qui n'est pas rebuté à l'idée d'utiliser cette couche de niveau inférieure.

¹⁵ L'orientation est spécifiée en donnant les axes Z (profondeur) et Y (haut).

Plutôt que de préparer une scène sous forme d'un graphe de scène et d'effectuer le rendu plusieurs fois avec seulement quelques modifications possibles, le mode immédiat propose un plus grand contrôle du pipeline graphique et des animations. Il existe deux manières (que l'on peut utiliser conjointement) pour envoyer les ordres de rendu à chaque trame : en utilisant des appels à la fonction *DrawPrimitive* pour envoyer directement les commandes au pipeline graphique ou en utilisant les *buffers* d'exécution. Pour cette deuxième alternative, le principe est le même qu'en mode retenu : utiliser une structure qui sauvegarde les différentes informations servant au rendu et que l'on référence au lieu de donner pour chaque affichage toutes les commandes élémentaires. Quelle que soit la solution choisie, en mode immédiat, le rendu de la scène peut être optimisé en tenant compte de notre connaissance de la situation. Certaines options non présentes dans le mode retenu sont :

- format plus flexible des sommets : possibilité de rendu de sommets pré-transformés et/ou où la lumière est pré-calculée, jusqu'à 8 coordonnées de textures par sommets,
- multi-texturage,
- plus grand contrôle des animations,
- création de modules "sur mesure".

Cependant, le mode immédiat présuppose une connaissance plus approfondie des algorithmes 3D de base. En effet, le programmeur devra prendre lui-même en charge les points suivants :

- les matrices de transformation,
- le calcul des normales aux sommets,
- utiliser uniquement des triangles (le mode immédiat de Direct3D ne sait, comme la plupart des moteurs 3D optimisé pour les performances, que projeter des triangles pour éviter les problèmes de rendu des polygones concaves ou vrillés (aux sommets non coplanaires) et ne dispose pas de routines pour créer des objets complexes formés de triangles),
- gérer entièrement la boucle de rendu.

Nous n'allons pas rentrer dans les détails de ce mode comme nous l'avons fait pour le mode retenu et surtout nous ne donnerons pas d'exemple de programme. En effet, les différentes commandes de haut niveau du mode retenu n'étant pas disponibles, il faut les prendre en charge par plusieurs instructions de bas niveau. Le meilleur moyen pour appréhender ce mode est de se reporter au tutoriel de Microsoft [D3Da]. Cependant, voici un exemple illustrant l'utilisation des commandes du mode immédiat effectuant la même opération qu'une commande du mode retenu. Il s'agit d'ajouter dans un buffer d'exécution les coordonnées des sommets d'un triangle et leurs normales, l'équivalent de la méthode *AddFace* de l'objet *MeshBuilder* vue précédemment. Cette partie de code vient après la définition des lumières, des matériaux et des transformations à utiliser et également après l'initialisation du *buffer* d'exécution en fonction de la taille des commandes qu'il reçoit.

```

1 // Remplissage du buffer d'exécution avec un triangle.
2 // Pour cela, on utilise un pointeur sur le champ lpData
3 // de notre buffer d'exécution d3dExeBufDesc
4
5 lpVertex = (LPD3DVERTEX)d3dExeBufDesc.lpData;
6
7 // Premier sommet
8 // Position en coordonnées du modèle
9 lpVertex->dvX = D3DVAL( 0.0);
10 lpVertex->dvY = D3DVAL( 1.0);
11 lpVertex->dvZ = D3DVAL( 0.0);
12 // Définition de la normale

```

```

13 lpVertex->dvNX = D3DVAL( 0.0);
14 lpVertex->dvNY = D3DVAL( 0.0);
15 lpVertex->dvNZ = D3DVAL(-1.0);
16 // coordonnées de texture (même si non utilisée)
17 lpVertex->dvTU = D3DVAL( 0.0);
18 lpVertex->dvTV = D3DVAL( 1.0);
19
20 // Deuxième sommet
21 lpVertex++;
22 // Position en coordonnées du modèle
23 lpVertex->dvX = D3DVAL( 1.0);
24 lpVertex->dvY = D3DVAL(-1.0);
25 lpVertex->dvZ = D3DVAL( 0.0);
26 // Définition de la normale
27 lpVertex->dvNX = D3DVAL( 0.0);
28 lpVertex->dvNY = D3DVAL( 0.0);
29 lpVertex->dvNZ = D3DVAL(-1.0);
30 // coordonnées de texture
31 lpVertex->dvTU = D3DVAL( 1.0);
32 lpVertex->dvTV = D3DVAL( 1.0);
33
34 // Troisième sommet
35 lpVertex++;
36 // Position en coordonnées du modèle
37 lpVertex->dvX = D3DVAL(-1.0);
38 lpVertex->dvY = D3DVAL(-1.0);
39 lpVertex->dvZ = D3DVAL( 0.0);
40 // Définition de la normale
41 lpVertex->dvNX = D3DVAL( 0.0);
42 lpVertex->dvNY = D3DVAL( 0.0);
43 lpVertex->dvNZ = D3DVAL(-1.0);
44 // coordonnées de texture
45 lpVertex->dvTU = D3DVAL( 1.0);
46 lpVertex->dvTV = D3DVAL( 0.0);

```

Après avoir ajouté un élément au *buffer* d'exécution, on définit quelles transformations il doit subir en lui assignant une matrice de transformations. On doit également définir avec quelles options de rendu le nouvel objet doit être dessiné.

2.3.2 Avantages et inconvénients

Les avantages et inconvénients des deux modes de rendu de Direct3D sont donnés en vrac ici. Pour le mode immédiat :

- plus grande complexité d'apprentissage et de programmation,
- plus grande flexibilité pour effectuer des effets spéciaux,
- peut faire des opérations que le mode retenu ne peut pas faire,
- offre généralement un rendu plus rapide,
- peut subir des changements dans les prochaines versions.

Pour le mode retenu :

- bien plus facile à prendre en main,

- peut faire des opérations simples qui sont complexes avec le mode immédiat,
- rendu plus lent,
- moins sensible aux futures mises à jour dans les nouvelles versions.

A notre avis, l'utilisation du mode immédiat à l'intérieur d'une application en mode retenu est la meilleure chose à faire. C'est la solution qui offre le meilleur compromis entre rapidité, flexibilité, complexité et temps de développement. Cette approche nécessite un peu de travail et de temps pour comprendre comment gérer les procédures de type *callback* mais cela ajoute de grandes libertés de programmation jointes à la clarté de la structuration sous forme de graphe de scène du mode retenu.

2.4 Java3D

Java3D est une nouvelle API développée par Sun Microsystems apparue au début de l'année 1998 et faisant partie de la suite *JavaMedia*. Cette bibliothèque de classes donne aux programmeurs des constructeurs de haut niveau permettant de créer et de manipuler une scène 3D et de construire les structures permettant de spécifier comment elle doit être affichée. Le principal atout de cette API est de transmettre la caractéristique de tout code écrit en Java dite « *write once, run anywhere* »¹⁶. D'autres avantages peuvent cependant paraître tout aussi importants, comme fournir un ensemble de classes de haut niveau. Java3D est également fort bien intégrée à Internet puisque les applications et *applets* écrites avec Java3D ont accès à toutes les autres classes Java.

Les bases de Java3D viennent d'API graphiques existantes et de nouvelles technologies. Pour la partie bas-niveau, elle synthétise les meilleures idées des API comme Direc3D, OpenGL, QuickDraw3D et XGL¹⁷ tout en s'appuyant dessus. De même, pour la manipulation des scènes, elle synthétise les meilleurs algorithmes issus des techniques de gestion des graphes. Enfin, Java3D introduit des aspects qui ne sont typiquement pas considérés comme faisant parties des environnements graphiques, comme le son spatial 3D, mais qui immergent l'utilisateur encore plus dans la scène.

Les buts fixés pour Java3D lors de la spécification de l'API furent par ordre de priorité :

- de hautes performances : les décisions entre deux solutions possibles pour une même opération penchent toujours vers la solution la moins coûteuse ; favorisant ainsi le temps d'exécution.
- un riche choix de fonctions : la difficulté fut de trancher entre les fonctions permettant de créer des scènes intéressantes et celles qui paraissaient non essentielles voire obscures à l'utilisation.
- programmation orientée objets : offrir un paradigme de POO de haut niveau qui permet aux programmeurs de développer rapidement des applications et des *applets* sophistiquées. Une application consistera à créer différents objets graphiques et à les connecter entre eux pour former une structure arborescente : le graphe de scène.
- chargement des fichiers 3D : fournir des fonctions de chargement de divers formats de fichier de description de scène 3D, autorisant ainsi l'utilisation d'un grand nombre de fichiers du monde de la CAO.

Java3D propose plusieurs mécanismes pour le rendu des scènes : le mode immédiat, le mode

¹⁶ Une seule écriture pour une exécution multi plates-formes.

¹⁷ API graphique 2D et 3D pour Solaris.

retenu et le mode retenu compilé. Chacun de ces modes offre, dans l'ordre de citation, de plus en plus de liberté au moteur Java3D pour les optimisations internes lors de l'exécution de l'application.

- Comme pour les API précédentes, le mode immédiat sert à donner des primitives de base que le moteur 3D doit traiter de suite. Même si la place réservée normalement aux optimisations dans ce mode est petite, Java3D accélère cependant le rendu de chacun des objets, grâce à un rehaussement du niveau d'abstraction. En effet, il est permis dans ce mode d'utiliser ou non un graphe de scène et d'utiliser des fonctions des deux modes retenus. Pour les deux modes suivants, l'utilisation du graphe de scène est obligatoire puisque c'est cette structure qui renferme (qui retient) les objets graphiques à dessiner.
- Le mode retenu est pratiquement aussi flexible que le mode immédiat mais il augmente néanmoins sensiblement les temps de rendu. Le graphe de scène est entièrement manipulable. L'ajout, l'effacement et l'édition d'un nœud sont des opérations permises dont la mise à jour est immédiate à l'écran.
- Le mode retenu compilé autorise le moteur Java3D à lancer une série d'optimisations complexes comme la compression des objets graphiques, l'aplanissement du graphe de scène, le groupement des objets, ... Une fois compilés, le programmeur a un accès minimal aux données et au graphe lui-même. Il peut accéder à certaines propriétés uniquement si les variables booléennes présentes à cet effet ont été initialisées comme il le faut.

2.4.1 Le graphe de scène

Le modèle de programmation orientée objets de Java3D, basé sur les graphes de scènes, fournit un mécanisme simple et flexible pour représenter et pour afficher tous les éléments d'une scène. Cette API autorise en effet l'utilisateur à penser en terme d'objets plutôt qu'en ensembles de triangles et également à la composition de la scène plutôt qu'à la façon d'en effectuer le rendu. C'est en ça que Java3d est une API de niveau supérieur.

Le graphe de scène est constitué d'objets Java 3D, appelés nœuds, arrangés dans une structure arborescente. L'utilisateur de l'API Java 3D crée un ou plusieurs sous-graphe de scène et les attache à un univers virtuel (objet *VirtualUniverse*). La seule restriction de Java 3D est de ne pas supporter des cycles dans le graphe. Les graphes et sous-graphes d'une scène Java 3D seront donc des graphes directs acycliques (DAG) tout comme dans une hiérarchie de nœuds VRML.

Dans Java 3D, les nœuds sont séparés en deux catégories, les groupes et les feuilles, comparables aux classes *container* et *component* de Java. Les groupes contiennent un ou plusieurs nœuds fils mais ne peuvent avoir qu'un seul nœud parent. Les feuilles, quant à elles, contiennent les définitions des objets (géométries, lumières, sons, etc.) présents dans la scène. Ces nœuds feuilles ne peuvent en aucun cas contenir des nœuds fils mais peuvent être rattachés à plusieurs parents (tant qu'ils n'introduisent pas de cycles), faisant alors office de macros réutilisables.

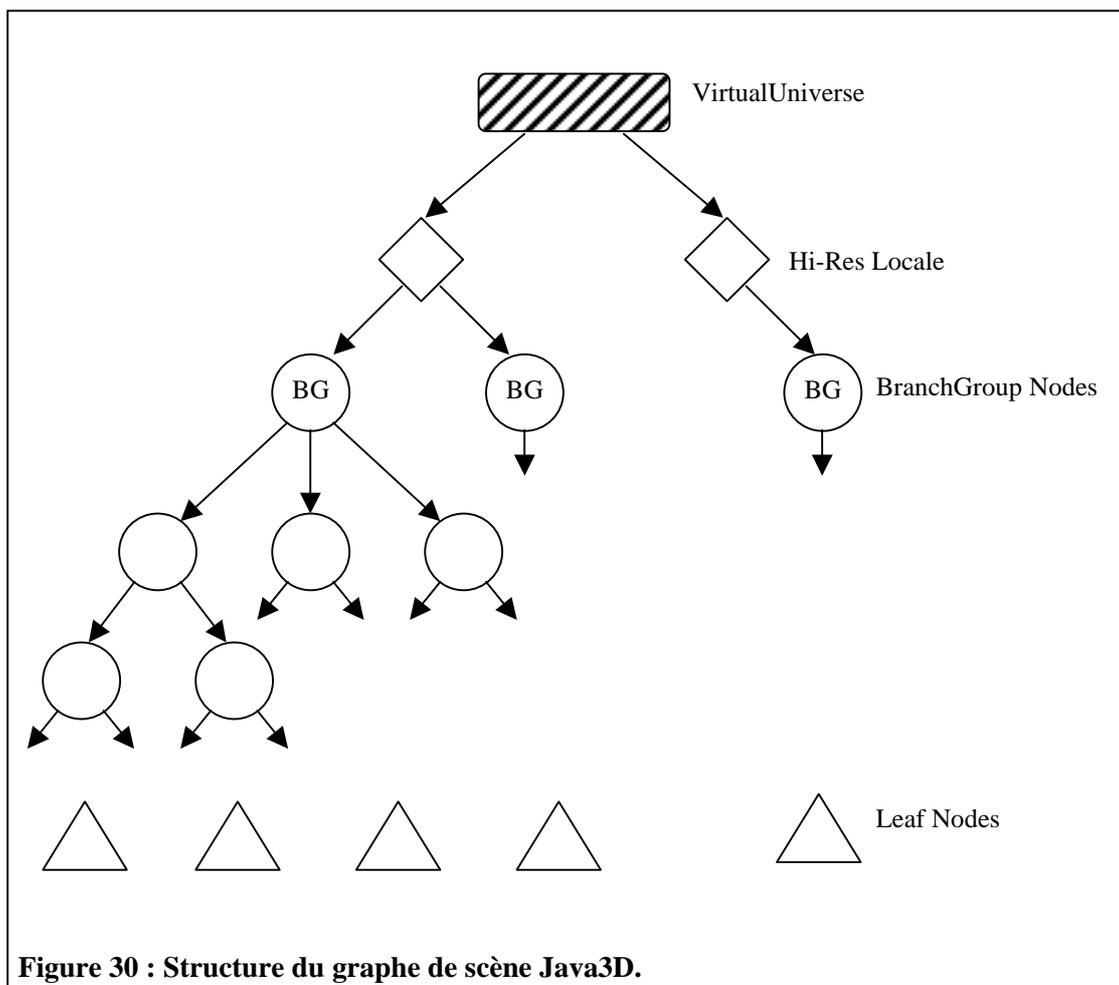


Figure 30 : Structure du graphe de scène Java3D.

Structure d'un graphe de scène

Le graphe de scène organise et contrôle le rendu des objets le constituant. Le moteur Java3D peut dessiner les différents objets indépendamment des autres ; ceci grâce à la forme du graphe de scène et au fait qu'il ne peut pas partager le même état entre plusieurs branches d'un arbre. Cette caractéristique permet d'effectuer les calculs en parallèle.

L'organisation hiérarchique du graphe de scène autorise naturellement le groupement spatial des objets graphiques se trouvant aux feuilles du graphe. Tous les nœuds internes jouent ainsi un rôle de regroupement de leurs nœuds enfants mais également celui de frontière spatiale englobant tous les objets géométriques définis par ses descendants. Cette dernière propriété est d'une grande utilité pour l'implémentation efficace de certaines opérations comme la détection de proximité, la détection de collision, ...

L'état d'une feuille est défini par tous les nœuds se trouvant sur le chemin direct menant de cette feuille au sommet du graphe de scène. Parce que le contexte graphique d'une feuille ne dépend que de ce chemin, le moteur Java3D peut décider de parcourir le graphe de scène dans un ordre quelconque (profondeur d'abord, largeur d'abord ou même en parallèle). Les seules exceptions sont pour les nœuds aux effets non dépendant de leur position dans l'espace comme les lumières ou le brouillard. Cette propriété est cependant un changement par rapport à des API plus vieilles comme PHIGS ou SGI Inventor avec lesquelles le changement d'état d'un nœud pouvait affecter tous les autres nœuds dans l'arbre.

Les objets du graphe de scène

Comme dit précédemment, le graphe de scène est un ensemble de nœuds ou objets Java3D connectés entre eux. Chacun de ces objets est construit en créant une nouvelle instance de la classe de l'objet désiré et est manipulé par ses méthodes *get* et *set*. Lorsqu'un objet est créé et connecté à d'autres objets dans le graphe de scène pour former un sous-graphe, ce dernier peut être attaché à un objet *univers virtuel* (classe *VirtualUniverse*) par l'intermédiaire d'un objet localisation haute résolution (classe *HiResLocale*), rendant la totalité *live* (que le moteur Java3D peut afficher). Avant d'attacher ce sous-graphe, il est cependant possible de le compiler dans un format interne optimisé.

Une caractéristique importante commune à tous les objets du graphe de scène est qu'ils ne peuvent pas être accédés ou modifiés durant la création du graphe de scène sauf si cela a été explicitement autorisé. Les accès à la plupart des méthodes *set* et *get* des objets qui font parties d'un graphe de scène *live* ou compilé sont limités. Cela permet au moteur Java3D de procéder à l'optimisation du graphe de scène avec la certitude que celui-ci ne sera pas modifié et donc que les optimisations seront faites une fois pour toute. Cependant, chaque objet dispose d'un masque de bits servant à marquer les différentes autorisations de modification de certains attributs via les méthodes *get* et *set* quand l'objet est *live* ou compilé. Par défaut, tous ces bits sont à zéro (modification impossible). Il suffit alors, pour pouvoir modifier un attribut, de mettre à 1 le bit correspondant.

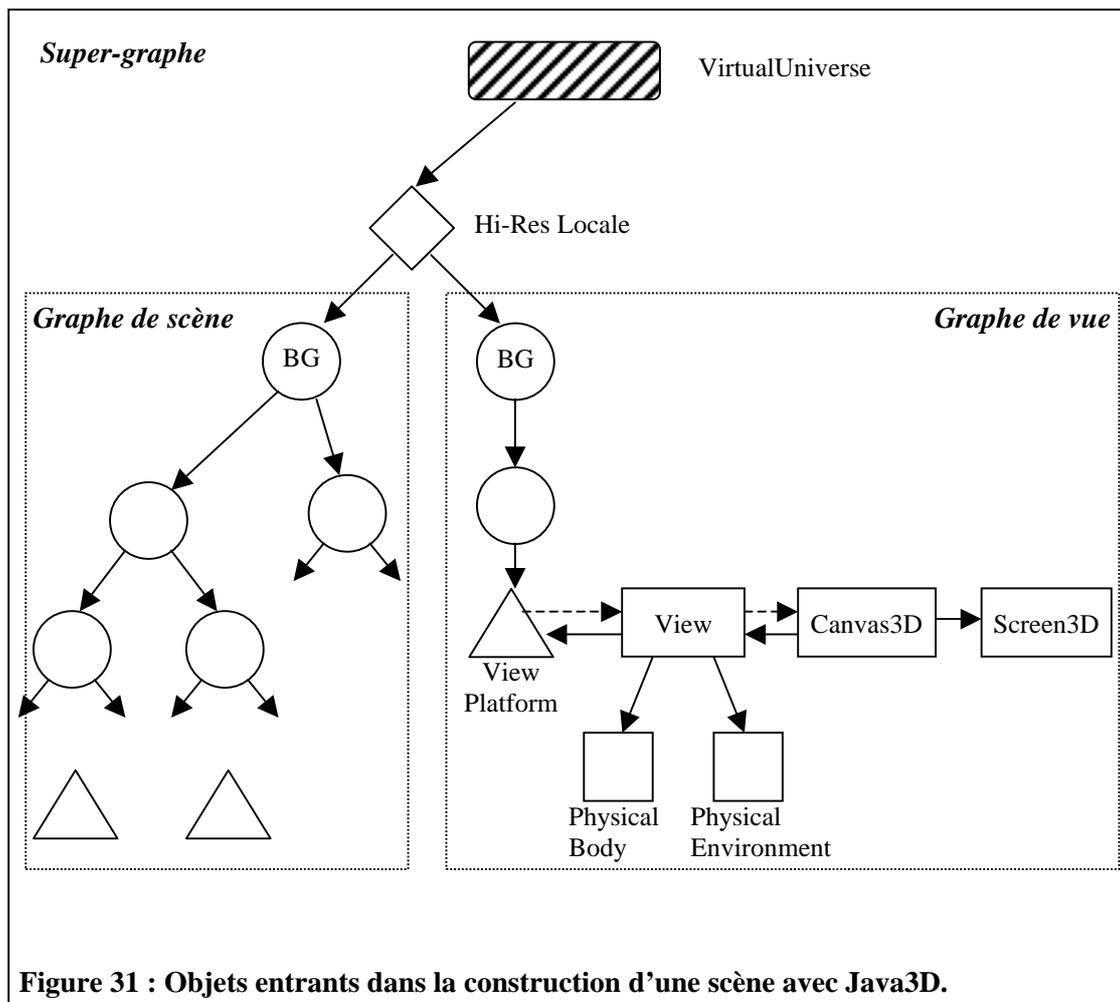
Les objets de la super-structure de graphe

Dans la partie précédente, nous avons introduit les deux objets « univers virtuel » et « localisation haute résolution ». Ce sont les deux objets formant la superstructure du graphe de scène. L'objet *VirtualUniverse* est une liste d'objets *HiResLocale*. Ces derniers contiennent une collection de graphes de scène et servent à définir l'origine dans l'espace des différentes coordonnées rencontrées dans les objets des graphes de scènes qu'ils renferment.

Les objets de visualisation de la scène

Java3D définit cinq objets qui ne font pas partie du graphe de scène proprement dit mais qui permettent de spécifier les paramètres de visualisation et de donner un lien vers le monde physique (c'est-à-dire vers les périphériques d'entrée-sortie). La Figure 31 montre les relations entre ces différents objets.

- L'objet *Canvas3D* contient tous les paramètres associés à la fenêtre dans laquelle la scène est affichée. Quand un tel objet est attaché à un objet *View*, le moteur Java3D effectue le rendu de la vue décrite dans la fenêtre spécifiée. Plusieurs objets *Canvas3D* peuvent pointer sur la même vue et ainsi afficher la scène du même point de vue.
- L'objet *Screen3D* contient les paramètres de l'écran contenant l'objet *Canvas3D*, tels que la résolution horizontale et verticale, la dimension physique de l'écran, ...
- L'objet *View* spécifie les informations nécessaires au rendu de la scène. C'est l'objet central qui lie les différents éléments intervenant dans la visualisation. Tous les paramètres de visualisation sont contenus soit directement dans l'objet *View* soit dans un objet pointé par lui. Java3D permet d'activer simultanément plusieurs objets *View* (plusieurs périphériques de sorties) pouvant eux-mêmes afficher une scène dans différents objets *Canvas3D* (dans différentes fenêtres).
- L'objet *PhysicalBody* contient des paramètres physiques intervenant dans le rendu, tels que la position de la tête, position des yeux, ...
- L'objet *PhysicalEnvironment* encapsule les paramètres de l'environnement physique comme la calibration des casques ou des gants pour la réalité virtuelle.



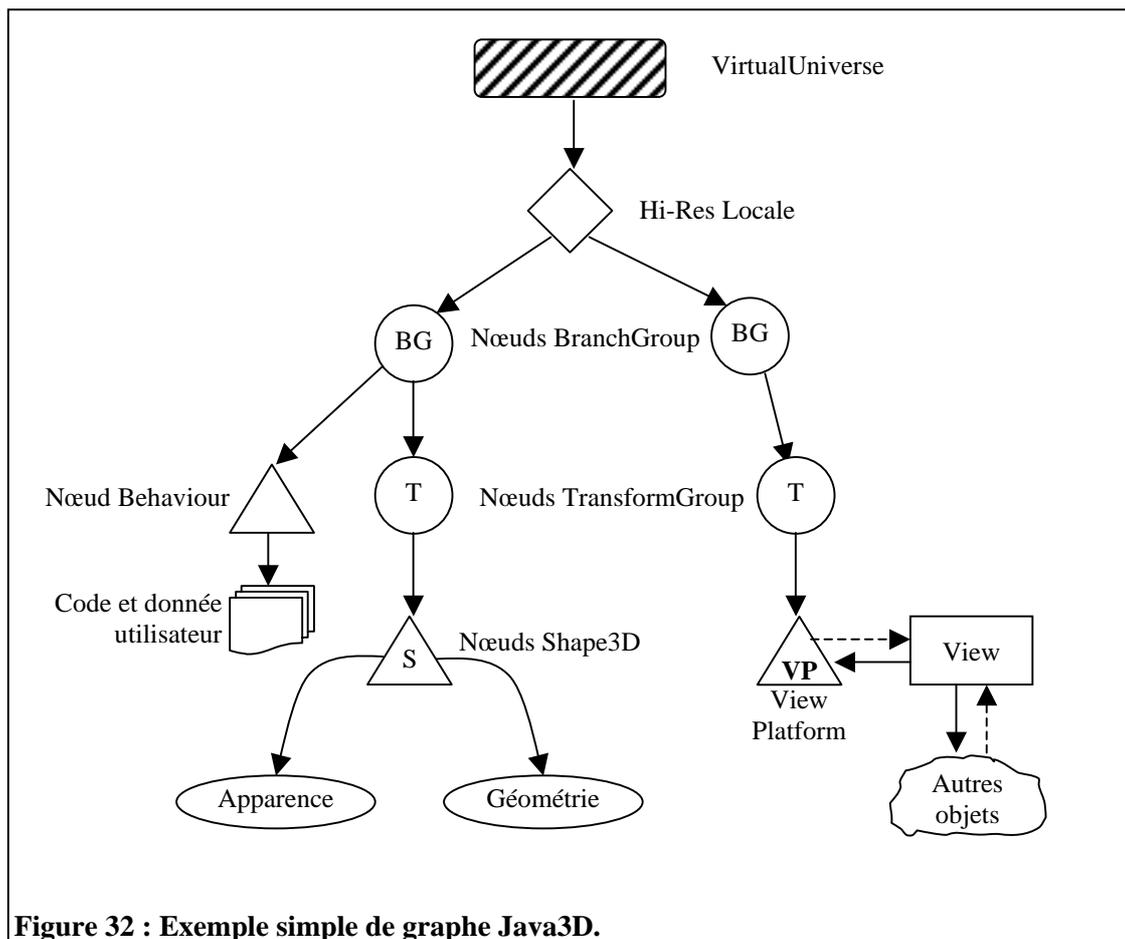
2.4.2 Construction d'une application en Java3D

Connaissant les différents éléments constituant une scène en Java3D, nous pouvons maintenant aborder les différentes étapes menant à la création d'un super-graphe. Ces étapes sont, la plupart du temps, toujours les mêmes, quelle que soit la complexité de la scène que l'on désire gérer.

1. Création d'un objet *Canvas3D* que l'on ajoute à un objet *Panel* d'une *Applet* ou d'une application autonome.
2. Création d'un nœud *BranchGroup* servant à regrouper les nœuds constituant la scène. Ces nœuds incluent les nœuds *Behaviour* qui servent, comme les scripts en VRML, à définir des fonctions utilisateurs pouvant modifier certains attributs des nœuds.
3. Construction des objets de la scène avec des objets *Shape3D* regroupés en fonction des transformations qu'ils doivent subir dans des nœuds *TransformGroup*.
4. Création des nœuds *Behaviour* et des codes utilisateurs modifiant les nœuds *TransformGroup*. Pour une rotation automatique, on pourra par exemple utiliser un nœud *RotationInterpolator* qui effectue des rotations d'un angle donné tous les intervalles de temps donnés.
5. Construction de tous les nœuds entrant dans la composition du graphe de vue et du super-graphe :
 1. Création de l'objet *VirtualUniverse* et du ou des objets *Locale*.

2. Création des objets *PhysicalBody*, *PhysicalEnvironment*, *View* et *ViewPlatform*.
3. Création de l'objet *BranchGroup* auquel on attache un nœud *TransformGroup* qui contient lui-même le graphe de vue créé précédemment.
4. Attachement de ce nœud *BranchGroup* à l'objet *HiResLocale*.
6. Liaison du graphe de scène à l'objet *Locale*.

Le graphe complet, pour un exemple tel que *HelloUniverse* ne comportant qu'un seul objet graphique, sera le suivant :

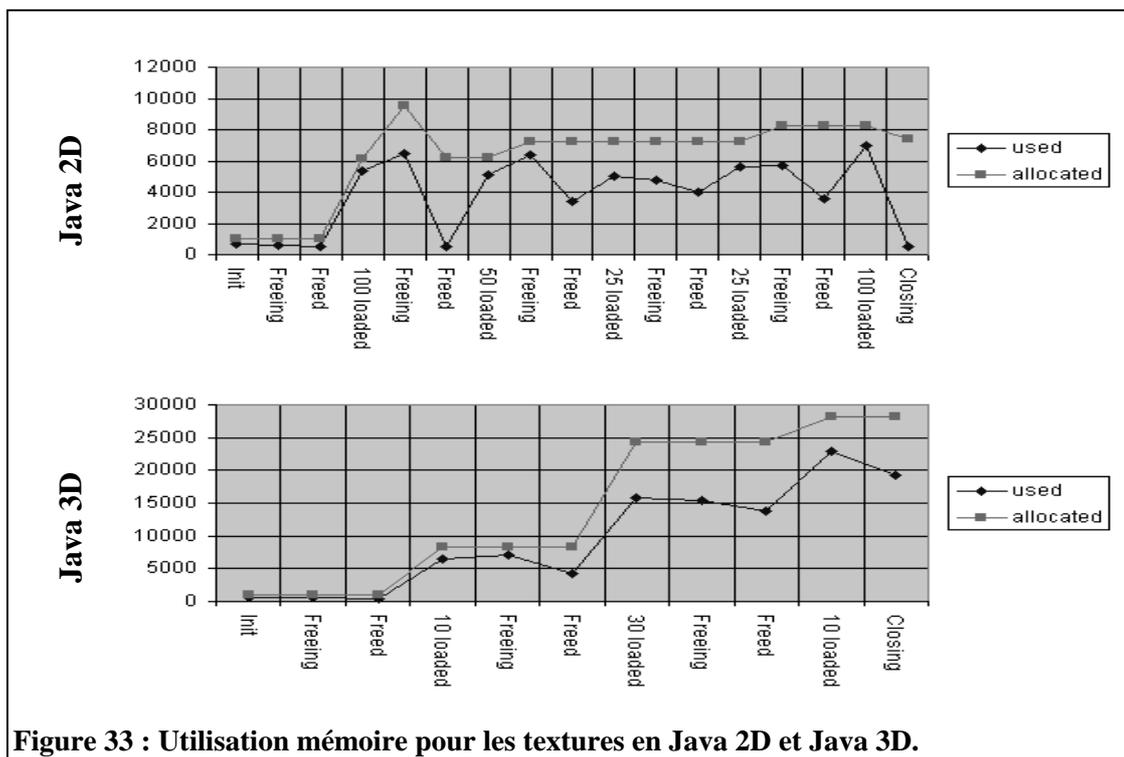


2.4.3 Avantages et inconvénients de la solution Java3D

Il existe également pour Java3D un paquetage regroupant diverses fonctions pour charger (« parser ») et afficher un fichier VRML97. Les classes de ce paquetage permettent de mettre en œuvre rapidement un navigateur VRML relativement complet. C'est un avantage de taille et surtout un gain de temps de développement précieux puisque l'interpréteur VRML n'est pas à écrire complètement.

Les autres avantages inhérents à une solution Java3D viennent directement des spécifications de départ données en début de cette partie. Quant aux inconvénients bien qu'étant d'apparence peu importants, ils suffisent néanmoins à nous faire préférer une solution utilisant le langage C avec l'API OpenGL.

- les seules plates-formes pour lesquelles est disponible l'API sont Solaris et Windows 95 ou NT,
- le peu de documentation et d'exemples,
- dû à sa caractéristique d'API haut niveau, Java3D ne rend pas accessibles aux programmeurs les détails du pipeline de rendu. Les programmeurs ne peuvent donc pas le contrôler comme dans une API graphique de bas niveau alors que certaines tâches l'exigeraient,
- Java3D effectue le rendu par liaison avec des API 3D de bas niveau (OpenGL ou Direct3D). Ce mode de fonctionnement ne se marie pas très bien avec d'autres composants Java tels que Swing, diminuant ainsi les possibilités offertes pour construire une interface graphique chiadée,
- enfin parmi les bogues¹⁸ relevées, un nous gêne particulièrement : le ramasse miette de la machine virtuelle Java ne désalloue pas la mémoire vive utilisée par des objets que l'on a détruits. Lorsque les objets graphiques d'une scène changent, que l'on doit en supprimer certains et en créer d'autres, ceux que l'on libère ne rendent donc pas la mémoire qu'ils utilisaient. Cela se ressent particulièrement lorsque les scènes sont chargées en textures comme le montre le graphique suivant. En effet, on y voit l'utilisation de la mémoire utilisée pour l'affichage des textures en 2D et en 3D et le fait que Java3D ne libère jamais la mémoire allouée aux textures alors que c'est le cas en 2D.



2.5 Fahrenheit

SGI et Microsoft ont formé un groupe de travail pour définir une architecture graphique qui sera probablement l'avenir des API graphiques 3D. Ce projet, au nom de Fahrenheit, a pour but de

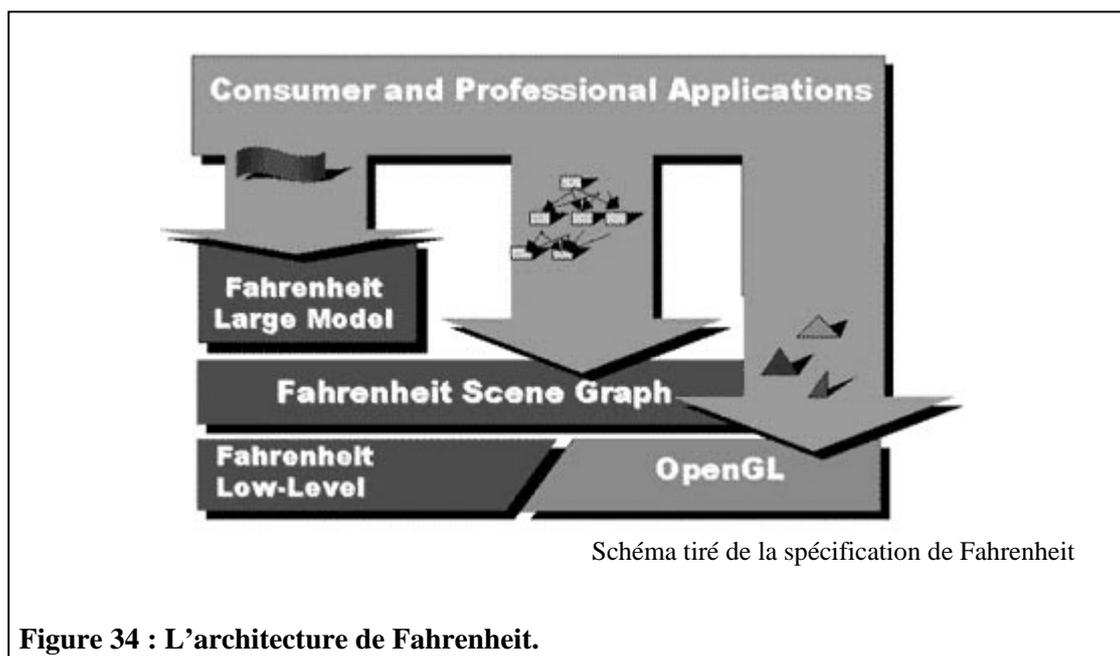
¹⁸ Voir sur le site de Sun les pages Java3D.

fournir aux développeurs un outil puissant permettant de créer facilement des applications 3D sur les plates-formes SGI, Windows et HP-UX.

L'API Fahrenheit se compose de 3 couches :

- L'API FFL (*Fahrenheit Low Level*) est la librairie de rendu bas niveau supportant l'accélération du matériel graphique. Cette API travaille avec les primitives graphiques de base (lignes, points, triangles). Les programmes qui s'appuient sur cette API devront, comme avec OpenGL et le mode immédiat de Direct3D, spécifier au système comment dessiner une scène avec une série de primitives graphiques et leurs propriétés. FFL remplacera l'API Direct3D sous Windows, se plaçant à côté d'OpenGL qui sera conservée. En attendant la sortie de cette API dans le courant de l'année 2000, les API bas niveau utilisées seront Direct3D et OpenGL.
- L'API FSG (*Fahrenheit Scene Graph*) est l'API qui fournit aux programmeurs un plus grand niveau d'abstraction en s'appuyant sur une couche inférieure (OpenGL, Direct3D ou FFL). Elle permet de décrire une scène sous forme de graphe composé d'objets décrivant les formes géométriques, les textures, les lumières, etc. Les applications utilisant cette API devront décrire ce qu'il faut dessiner plutôt que comment dessiner. Il sera également possible (et c'est ce qui est nouveau par rapport aux graphes Java3D) de spécifier quelles sont les préférences de l'application : la qualité d'image ou un taux de rafraîchissement constant. Cette dernière possibilité offre de nouvelles perspectives pour l'utilisation de la 3D afin de présenter des informations évoluant en temps réel. Le graphe de scène, géré par FSG, est optimisé et les performances devraient être étonnantes.
- L'API FLM (*Fahrenheit Large Model*) est une extension de FSG permettant de gérer et de visualiser des scènes de taille imposante. Elle autorise également à spécifier des constructions typiques de la CAO comme les surfaces paramétrées NURBS (*Non Uniform Rational B-Splines*).

La Figure 34 montre l'architecture de Fahrenheit et illustre comment les différentes couches peuvent être utilisées dans une même application. Cela permet, d'une part, d'utiliser une spécification haut-niveau pour une scène, tout en ajoutant des appels de fonctions bas niveau pour des raisons spécifiques. Pour les applications actuelles, cela permet également d'adopter petit à petit les API FSG et FLM plutôt que de devoir les réécrire en entier.



Le composant le plus important dans l'architecture Fahrenheit est l'API FSG. Elle permet de supporter un grand nombre de configurations matérielles et de répondre aux préférences d'une application. Ses points forts sont les suivants :

- L'abstraction des primitives graphiques tout en maintenant de très bonnes performances. Les programmeurs d'interfaces 3D pourront se concentrer sur les fonctionnalités de leurs applications plutôt que de résoudre les problèmes liés à l'utilisation d'une API de bas niveau.
- L'ajout ou la modification des données du graphe de scène en temps réel.
- Le support des machines multiprocesseurs sans devoir inclure un code spécifique. Différents processus peuvent accéder et modifier le graphe de scène simultanément. Les tâches courantes comme le rendu, les calculs d'intersections et de collisions bénéficient automatiquement de la présence de plusieurs processeurs. Là aussi, Fahrenheit supporte donc l'abstraction du matériel présent.

3 LES OUTILS 3D ET LE WEB

Résumé Nous parlons ici de l'alternative à une programmation directe d'une scène à l'aide de l'une des bibliothèques de fonction 3D présentées précédemment. Le couple {modeleur, navigateur 3D} permet en effet de créer et de naviguer dans une scène 3D sans effort de programmation. Les deux différents types de modeleurs (l'un à utilisation professionnelle, l'autre pour la création et la mise en ligne rapide) sont tout d'abord présentés. Puis nous répertorions les co-navigateurs (*plugins*) 3D les plus utilisés avant de nous pencher plus en détails sur le langage VRML.

Les logiciels offrant la possibilité de créer et/ou de visualiser des scènes 3D se sont multipliés avec l'explosion des performances 3D des micro-ordinateurs. Ils ne sont plus uniquement réservés aux professionnels et leur prix a considérablement diminué. Ces logiciels existent désormais aussi bien sous forme de programmes libres de droits d'utilisation que payants mais se regroupent à présent en deux familles principalement :

- les modeleurs qui autorisent la création et l'édition des scènes,
- les moteurs 3D dont le rôle est le rendu des scènes. Ils se différencient des fenêtres de rendu en temps réel des modeleurs les plus récents en proposant, d'une part, le meilleur compromis possible entre affichage de haute qualité et vitesse de rafraîchissement et, d'autre part, la gestion de techniques d'interaction plus avancées.

La manière la plus simple de créer rapidement des scènes 3D interactives, sans avoir besoin de connaissances informatiques hormis celles d'un utilisateur de niveau intermédiaire ou expert selon les cas, est d'utiliser un modeleur 3D. En effet, la plupart de ceux-ci permettent désormais, même si ce n'était pas leur raison d'être au départ, de générer des fichiers décrivant une scène dont le format fait partie des filtres d'import de moteurs 3D. Ainsi, ces derniers seront capables d'interpréter les données des fichiers et par-là même de présenter la scène à l'utilisateur qui pourra interagir avec.

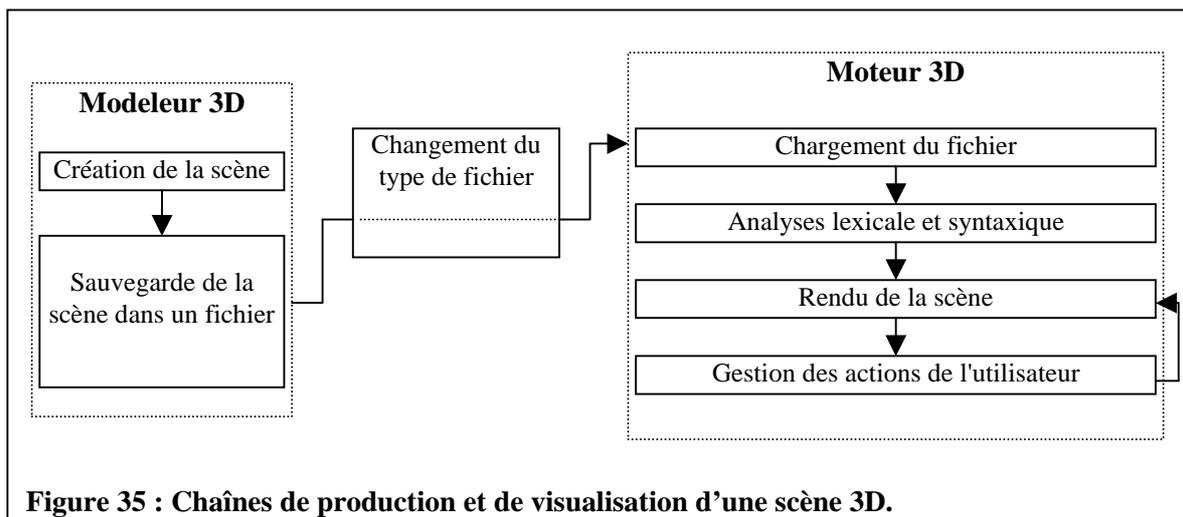


Figure 35 : Chaînes de production et de visualisation d'une scène 3D.

Les différentes étapes conduisant de la création d'une scène à l'aide d'un modeleur à notre but final qui est d'interagir avec elle par l'intermédiaire d'un moteur 3D sont schématisées dans la figure précédente. Les différentes techniques utilisées par les modeleurs 3D pour aider à la création et la manipulation des scènes, ainsi que les différentes fonctions prises en charge par le moteur 3D, sont couvertes plus loin dans ce rapport. La seule explication nécessaire à ce point concerne la boîte

intermédiaire dans le schéma. Sa présence sert à montrer que les formats de fichier en sortie du modelleur et ceux en entrée du moteur ne sont pas nécessairement les mêmes et qu'il est possible, dans ce cas, d'utiliser un certain nombre d'utilitaires de conversion entre formats de fichier de description de scènes 3D.

3.1 Quelques modelleurs

Selon le domaine pour lequel les modelleurs sont destinés, ils intègrent des fonctions et possibilités éloignées. La Figure 36 montre deux captures d'écran illustrant la grande différence entre les deux principaux types de modelleurs. Nous donnons, par la suite, une liste non exhaustive de modelleurs permettant de créer des fichiers VRML. En plus de ceux-ci, citons les plus célèbres que sont AutoCAD, Amapi 3D, Bryce 3D, ...

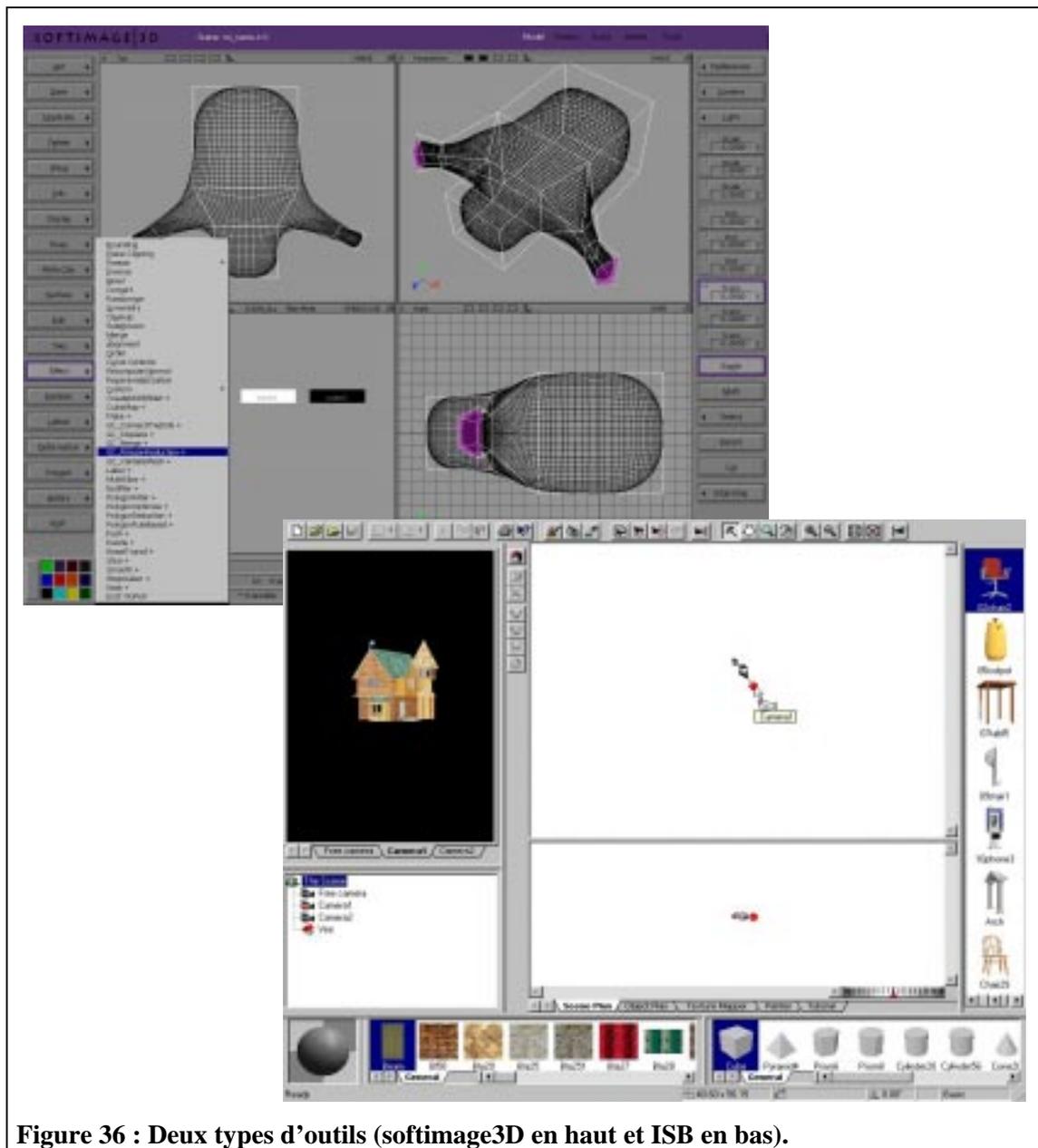


Figure 36 : Deux types d'outils (softimage3D en haut et ISB en bas).

Pour les professionnels de la C.A.O. et les architectes, la précision de placement et de taille des objets 3D est prépondérante. Pour cela, quatre vues permettent l'édition de l'objet ou de la scène

entière depuis n'importe quel côté. La visualisation offerte par les modeleurs répondant à ces critères, comme celui présenté à gauche dans la Figure 36, est le plus souvent de type lancer de rayons (pour rappel, c'est une technique plus précise, plus réaliste, mais également beaucoup plus lente et n'offrant donc pas de possibilités d'interaction). Bien que ces modeleurs ne soient pas conçus au départ pour créer un contenu exploitable dans un moteur 3D temps réel, la plupart de ces logiciels offrent néanmoins des filtres d'exportation de fichier pour permettre cela.

A l'inverse, sur la capture de droite, les modeleurs pour les créateurs de contenus 3D pour le World Wide Web ont pour but d'optimiser les fichiers en vue de la réduction du nombre de faces à afficher. Cette optimisation vise à offrir une plus grande fluidité d'interaction et un plus court temps de transfert de la scène 3D à travers le réseau. Ces modeleurs, d'aspect plus ludique que les précédents, permettent de créer en quelques minutes et quelques clics de souris un monde 3D texturé et sonore. Ils ne nécessitent pas d'apprentissage spécial contrairement aux modeleurs professionnels. La plupart exportent des fichiers VRML en vue de leur visualisation dans un *plugin* associé que nous verrons plus loin.

Cette deuxième famille de modeleurs apparus en nombre récemment, avec l'apparition de VRML, ne permet pas de créer un contenu aussi précis qu'avec les « vrais » modeleurs. En effet, bien qu'ils respectent les grandes phases de la création d'une scène 3D (modélisation à base de primitives ou d'extrusion, habillage par un matériel), ils ne permettent pas d'effectuer ces opérations avec la précision des autres modeleurs et en particulier les opérations de positionnement et de dimensionnement. C'est pourquoi, pour être plus exacts, nous préférons le terme d'outils VRML pour ces modeleurs.

Le nombre d'outils permettant de créer des scènes VRML97 est en constante augmentation. Le tableau suivant en donne quelques-uns. Ils sont aussi bien payants et destinés aux professionnels que *freeware* et à l'intention des concepteurs de sites en 3D.

Nom du modeleur	Créateur	Plates-formes	Description et URL
3DAnywhere	<i>Monfort Software Engineering Ltd</i>	Win32	Le dernier outil pour créer des présentations 3D consultables sur Internet. http://www.3danywhere.com/
3dsMax	<i>Discreet</i>	Win32	Un des modeleurs les plus utilisés sur PC. http://www2.discreet.com/products/d_products.html?prod=3dsmax
3D Virtual Character	<i>W Interactive SARL</i>	Sur tous les navigateurs HTML avec Java	Permet la création de modèles 3D d'un faciès à partir de photographies. http://www.winteractive.fr/
AvatarMaker 3D 1.0	<i>Sven Technologies Inc.</i>	Win32	Le premier outil de création d'avatars humains. http://www.sven-tech.com/products/avatarmaker/avatarmaker.html
Avatar Studio	<i>Canal +</i>	Win32	Création de personnages animés. http://www.avatarstudio.com/
Beyond 3d	<i>Uppercut Software</i>	Win32	Outil de création de scènes VRML97 incluant un navigateur. http://www.beyond-3d.com/beyond

Canoma	<i>Metacreations Inc.</i>	Win32, Mac	Assistant de création de scènes 3D à partir de photographies. http://www.metacreations.com/products/canoma/
Clayworks		Win32	Modeleur VRML http://members.aol.com/luther2000/clay.htm
Community Place Conductor	<i>Sony</i>	Win32	Création et manipulation d'objets 3D. Utilisé conjointement à inspire3D ou 3D Studio Max, cet outil permet de créer facilement des environnements 3D. http://www.community-place.com/product/conductor.html
DesignSpace	<i>DesignSpace, ANSYS</i>	Win32	Outil s'intégrant aux modeleurs professionnels et permettant d'obtenir des scènes en VRML97 afin de pouvoir faire des tests de performances avant la mise en ligne. http://www.designspace.com/
Dimension3D Scanner	<i>Micrografx</i>	Win32, SGI	Création de modèles 3D à partir d'une caméra évoluant en orbite autour d'un objet. http://www.dimension-3d.com/
Internet Character Animation	<i>ParallelGraphics</i>	Win32	Permet de créer des avatars animés. http://www.parallelgraphics.com/ica
Internet Scene Assembler	<i>ParallelGraphics</i>	Win32	Permet d'assembler différentes scènes VRML statiques ou dynamiques. http://www.parallelgraphics.com/isa
Internet Space Builder 3.0	<i>ParallelGraphics</i>	Win32	Création et édition de scènes VRML. http://www.parallelgraphics.com/isb
mjbWorld	<i>Martin Baker</i>	Java	Edition de scènes VRML. http://www.martinb.com/
Platinum VRCreator	<i>Platinum Technology</i>	Win32	Création de scènes VRML. http://www.platinum.com/products/appdev/vream/vrc_ps.htm

3.2 Les principaux moteurs 3D pour le Web

Les moteurs 3D pour le Web sont des applications qui étendent les possibilités des navigateurs HTML¹⁹ (*HyperText Markup Language*) afin de pouvoir afficher un contenu 3D avec lequel

¹⁹ HTML est une application de SGML (*Standard Generalized Markup Language*) Standard International ISO 8879. La version actuelle de HTML est la 4.0.

l'utilisateur peut interagir. Ces ajouts aux navigateurs HTML peuvent être soit des *plugins* (ou co-navigateurs en français), soit des *applets* (appliquettes) Java ; les deux méthodes permettant de charger la description d'une scène 3D à partir d'une URL²⁰ (*Uniform Resource Location*) et de la présenter à l'utilisateur.

Les *applets* autorisent l'intégration dans une page HTML d'une application Java prenant en charge des opérations que le langage HTML ne permet pas. Il existe trois manières différentes de créer une *applet* pour gérer la 3D :

- Ecrire le moteur entièrement en Java, c'est-à-dire toutes les étapes du pipeline 3D à l'aide de fonctions Java. Cette méthode n'offre cependant pas des performances satisfaisantes puisque le langage Java est seulement interprété mais aussi parce que les calculs 3D ne sont alors pas optimisés par le matériel mais uniquement pris en charges par le processeurs de la machine cliente.
- Ecrire une *applet* Java utilisant des commandes d'une API 3D. Par exemple, il existe des interfaces permettant d'appeler des fonctions OpenGL depuis le langage Java. Cette méthode a les avantages, d'une part, de réduire le temps de développement de l'*applet* puisque l'on utilise le moteur OpenGL et, d'autre part, d'accélérer le rendu 3D si la carte graphique 3D reconnaît les commandes OpenGL.
- Ecrire une *applet* Java utilisant les classes Java3D. Cependant, cette méthode a le désavantage par rapport aux deux autres de ne pas être compatible avec toutes les plateformes. En effet, Java3D n'existe pour l'instant que pour Windows et Solaris et pour Linux en version bêta.

Quant à l'appel d'une application Java depuis un navigateur HTML, il peut se faire de deux manières différentes :

- L'*applet* est intégrée à la page HTML grâce à une balise <APPLET> ou <OBJECT>. Dans ce cas, le navigateur réserve, dans la page HTML, la place spécifiée et l'application Java utilise cet emplacement pour ses affichages. Le désavantage de cette solution est que l'*applet* Java doit prendre en charge tous les aspects de communication comme la récupération d'une URL.
- L'application Java est autonome et se comporte comme un *plugin*. Le kit de développement de *plugins* pour Netscape permet en effet de lancer aussi bien des applications exécutables que des classes Java. Dans ce cas, l'application Java se comporte exactement comme un *plugin*.

Ces applications Java ne sont pas la meilleure solution pour intégrer des possibilités 3D à un navigateur HTML. D'une part, le langage est seulement interprété et donc peu performant, et d'autre part, la gestion de la mémoire par le ramasse-miettes ne donne pas suffisamment de contrôle sur les ressources utilisées (voir les désavantages de Java3D dans la partie 2.4.3 page 62). Le langage Java est à utiliser, à notre avis, pour créer des applications de communications et d'interactions 2D (dont les fonctions sont bien intégrées) mais en aucun cas pour réaliser des calculs coûteux comme ceux nécessaires à la 3D.

La quasi-totalité des moteurs 3D s'ajoutant aux navigateurs HTML se trouvent donc sous la forme de *plugins*. Et parmi ceux-ci la plupart sont écrits dans un langage et compilés pour donner une bibliothèque dynamique que l'on peut charger et désallouer à souhait.

Le fonctionnement d'un *plugin* est relativement simple et nous verrons (dans le chapitre 4.3

²⁰ Berners-Lee, T., Masinter, L. and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, Décembre 1994.

page 111) que la transformation d'une application autonome en un *plugin* n'est pas très difficile. Ici, nous ne parlerons uniquement que du fonctionnement général d'un *plugin*.

Un *plugin* est une application associée à un navigateur HTML qui permet de prendre en charge un ou des types d'URL (*Uniform Resource Locator*) qu'en général le navigateur ne sait pas traiter. Le mécanisme de lancement d'un *plugin* en fonction du type de l'URL est simple. La chaîne des traitements menant à l'exécution d'un *plugin* est schématisée dans la Figure 37.

Lorsqu'un utilisateur désire consulter une nouvelle URL (en cliquant sur un lien dans une page HTML ou en demandant l'ouverture d'une nouvelle page en saisissant directement son adresse), le navigateur envoie une requête HTTP²¹ (*HyperText Transfert Protocol*) au serveur hébergeant cette URL pour lui demander de lui renvoyer son contenu. Quelle que soit la version du protocole HTTP utilisée, le serveur renvoie en premier lieu le résultat de la demande suivie du type MIME²² (*Multipurpose Internet Mail Extensions*) de l'URL. Ce type MIME, constitué d'un type et d'un sous-type de fichier, permet de savoir quel est le contenu de l'URL et donc son format. C'est à partir de cette information que le navigateur HTML (le client Web) détermine s'il peut prendre en charge lui-même le contenu de la réponse HTTP ou s'il doit rechercher et exécuter un *plugin* sachant le faire.

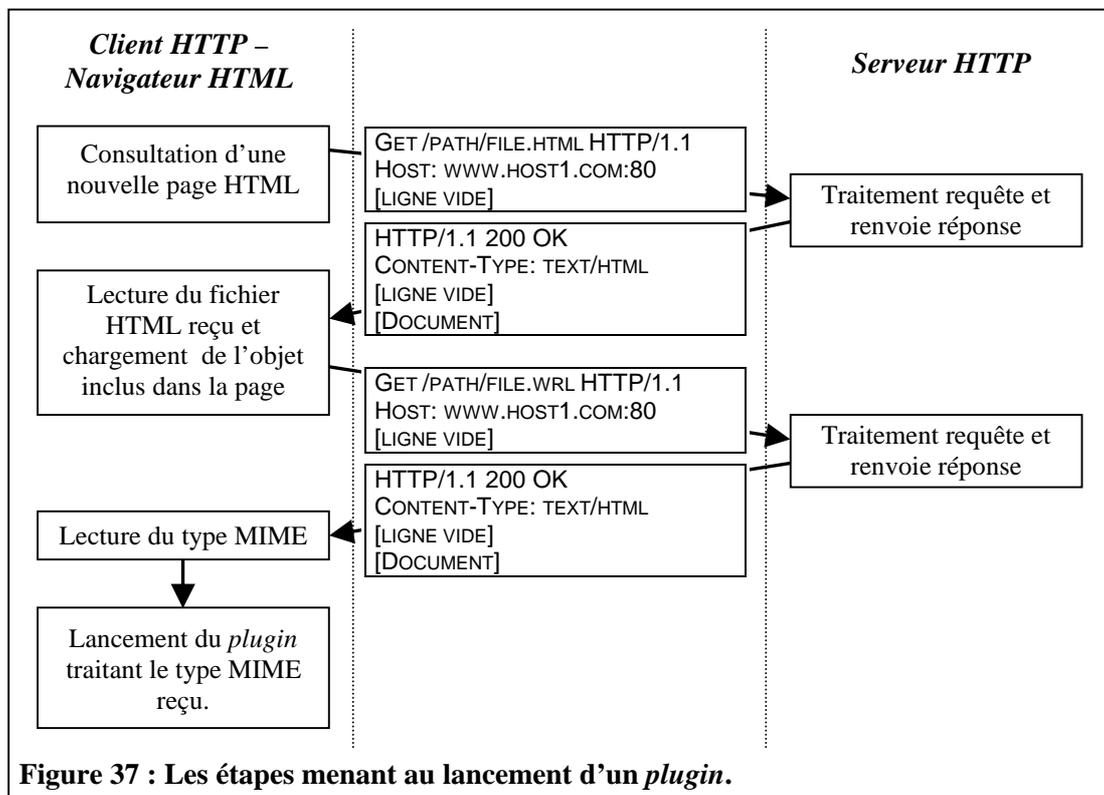


Figure 37 : Les étapes menant au lancement d'un *plugin*.

Tous les navigateurs auxquels on peut ajouter des *plugins* contiennent une liste associant les types MIME à l'application qui les prend en charge. Dans cette liste, sont également sauvegardées les

²¹ Le protocole de communication sous-jacent à la récupération de pages HTML. Les spécifications du protocole peuvent être trouvées sur le site du *Web Consortium* <http://www.w3c.org> dans la RFC 1945 pour la version 1.0 et dans la RFC 2068 pour la version 1.1.

²² Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, Novembre 1996.

Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, Novembre 1996.

Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, Novembre 1996.

différentes extensions qu'un fichier ayant ce type peut avoir. Ainsi, dans le cas où l'URL désignerait un fichier local, l'extension du fichier permet de définir le type MIME correspondant et donc de lancer l'application le prenant en charge. Comme nous le verrons dans la partie 4.3 page 111, il suffit d'ajouter la bibliothèque dynamique gérant un nouveau type MIME dans le répertoire contenant les différents *plugins* pour que les informations précédentes soient automatiquement ajoutées dans la liste des types MIME.

Le navigateur fournit un descripteur de fenêtre au *plugin* de telle manière, que si une URL nécessitant un *plugin* est incluse dans une page HTML (par la balise <INLINE> ou <OBJECT>), l'application gérant ce type MIME est lancée dans une fenêtre interne à la page HTML. Contrairement, si l'URL est l'URL de la page en cours alors toute la taille de la fenêtre du navigateur est allouée au *plugin*.

3.2.1 VRML

VRML (*Virtual Reality Modeling Language*) est un format de fichier qui, associé à un *plugin* (application augmentant les possibilités d'un navigateur HTML), forme un système de consultation d'un contenu 3D. Pour éviter un amalgame souvent fait par abus de langage, le véritable moteur 3D dans ce système est le *plugin* et non le "langage" VRML. En effet, VRML n'est pas un langage de programmation même s'il peut intégrer des scripts. Ce n'est qu'un format de fichier permettant de décrire des scènes 3D interactives avec possibilité d'y inclure des liens sur d'autres contenus consultables depuis un navigateur. Plus loin dans ce rapport, nous verrons plus en détails, d'une part, la structure des fichiers VRML et, d'autre part, la création d'un *plugin*. Ce qui nous intéresse précisément ici est le système VRML+*Plugin*²³ et ses caractéristiques et possibilités.

Historique du langage

L'initiative de fournir au Web un système 3D est à porter au crédit de Mark Pesce et de Toni Parisi. Probablement inspirés par des jeux comme Wolfenstein 3D et Doom, ils ont présenté, dès le printemps 1994 à la première conférence sur la réalité virtuelle associée au Web organisée par Tim Berners-Lee²⁴, leur application Labyrinth. A la suite de cette conférence sont nés le concept de Virtual Reality Markup²⁵ Language pour que les applications 3D sur le Web utilisent le même langage de description et la ferme intention de proposer rapidement une première version de ce langage. La *mailing-list* www-*vrml* fut créée pour recueillir les idées de chacun et, en automne 1994, Mark Pesce présenta la première spécification du langage.

Après maintes discussions, la décision fut prise d'utiliser une syntaxe proche du format de fichier en mode ASCII de Open Inventor de Silicon Graphics Inc. Ce format de fichier permettait déjà de décrire complètement une scène 3D puisqu'il intégrait la description des objets polygonaux, les lumières, les matériaux, les propriétés ambiantes, ... Ainsi la première version de VRML consistait en un sous-ensemble du format de fichier d'Open Inventor auquel furent adjointes des possibilités propres aux communications sur réseaux.

A la suite de la sortie de VRML 1.0 fut créé le groupe d'architecture de VRML (VAG : VRML Architecture Group) qui prépare dès 1996 la version 2.0 du langage. En août de la même année, le projet *Moving Worlds* de Silicon Graphics (avec la participation de Sony Research et Mitra) est retenu par la communauté VRML comme étant la nouvelle mouture de la version 2.0. Cette nouvelle version est fortement remaniée, endossant plus clairement une structure arborescente (on parle maintenant de nœuds plutôt que de balises), et en ajoutant les capacités d'interaction grâce à l'adjonction de scripts que l'on peut déclencher en fonction de certains événements.

²³ Une liste non exhaustive de plugins VRML est donnée à la page 76.

²⁴ Inventeur du langage HTML.

²⁵ Le mot 'Markup' fut changé en cours de spécification pour 'Modeling'.

L'évolution de VRML entre les deux normes au niveau des fonctionnalités touche principalement les points suivants :

- Sondeurs de proximité, de visibilité, ...,
- Animations automatiques,
- des scripts peuvent être intégrés dans le fichier VRML,
- textures fixes ou animées,
- son 3D,
- collisions.

Ces nouvelles fonctionnalités de VRML 2.0 lui permettent d'atteindre les qualités suivantes qui en font désormais sa force :

- Un monde réaliste :
 - On ne traverse pas les murs (collisions)
 - Image de fond (ciel)
 - On suit le relief du terrain grâce à la gravité
 - Le temps passe (gestion des horloges)
- Un monde animé :
 - Des éléments peuvent bouger dans la scène (des oiseaux volent, des voitures roulent)
 - Ils peuvent changer de couleur ou de forme
- Un monde interactif :
 - Les objets peuvent être déplacés
 - On peut les faire réagir en les touchant avec la souris ou en passant à proximité (un chien aboie lorsque l'on s'en approche, ouvrir une porte, déclencher un son)
- Un monde connecté :
 - On peut rencontrer un avatar et dialoguer avec lui
 - On peut créer des liens permettant d'aller sur un autre site en cliquant sur un objet dans la scène
 - On peut créer un monde composé d'objets venant d'autres sites

VRML 2.0 est toujours la dernière version du langage même si son nom est devenu VRML97 depuis qu'elle a été reconnue comme standard international par l'organisme de normalisation ISO en décembre 1997 sous le numéro ISO/IEC 14772-1:1997. La norme ne définit pas de dispositifs physiques ou de concepts dépendants du système comme la résolution d'un écran. Ainsi, le format de fichier VRML reste totalement indépendant des éléments d'un système et par conséquent est multi plates-formes.

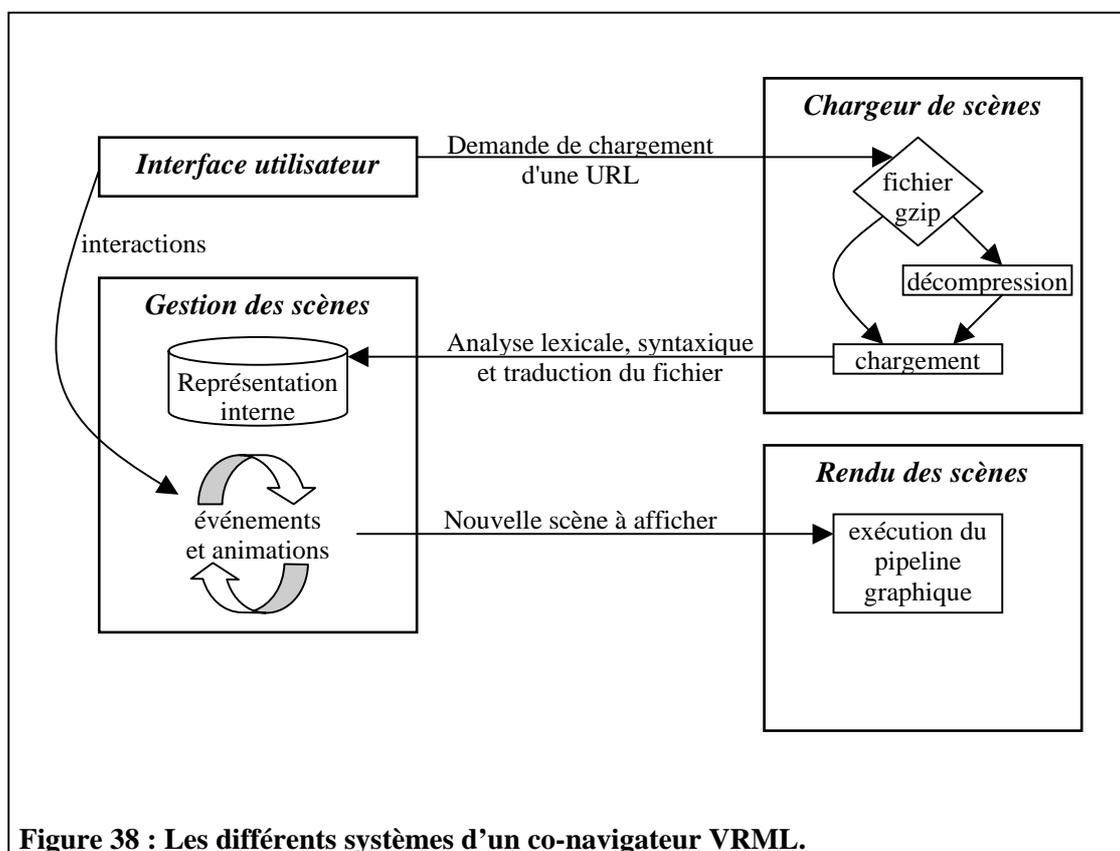
Les fonctions d'un navigateur VRML

Que le "moteur VRML" soit associé au navigateur HTML ou indépendant (application autonome), qu'il permette une consultation des URL ou simplement des fichiers sauvegardés en local (même si ce n'est pas la philosophie de départ puisque VRML intègre des ancres hypertextuelles), son fonctionnement est le même.

Les fonctions d'un navigateur VRML sont de charger une scène décrite à l'aide du langage VRML, de la dessiner à l'écran et de s'occuper des animations et de l'interaction avec l'utilisateur. Pour effectuer ces différentes tâches, un navigateur VRML comporte plusieurs sous-systèmes comme ceux donnés dans la Figure 38 (ces sous-systèmes sont seulement représentatifs et l'implémentation peut différer d'un navigateur à l'autre).

Le sous-système *interface utilisateur* gère l'emplacement occupé par la scène VRML dans une page Web. Ce sous-système positionne également les boutons, les menus et l'espace qu'occupera la scène 3D. Il répond également aux événements de l'utilisateur (frappe au clavier, sélection d'un élément d'un menu, déplacement ou clique de la souris, etc.) ; c'est-à-dire que lorsque l'utilisateur interagit avec la scène VRML, c'est ce sous-système qui répond aux événements et qui en informe les autres sous-systèmes.

Le sous-système *chargeur de scènes* gère, comme son nom l'indique, tout ce qui touche à la récupération et à l'interprétation des fichiers. Il gère, plus précisément, le chargement de la scène elle-même mais également celui des textures, des sons, des prototypes externes ou des scripts qui y sont utilisés. C'est ce système qui gère les URL et émet les erreurs lorsque quelque chose ne va pas. Il doit pouvoir aussi bien charger les fichiers au format UTF8²⁶ que les fichiers compressés au format gzip. Il fournit au sous-système de *gestion des scènes* une représentation interne de la scène. Le *chargeur de scènes* est le seul élément du moteur dans lequel les opérations ne sont pas critiques puisqu'elles n'interviennent qu'une seule fois avant tout rendu de la scène.



Le sous-système *gestion des scènes* gère la représentation interne au moteur 3D des scènes VRML. Cette représentation inclut des informations qui permettent au moteur de dessiner les scènes plus vite. Ce sous-système garde également une trace du point de vue de l'utilisateur, des caractéristiques de navigation et gère les événements lorsque l'utilisateur entre en collision avec un objet solide. De plus, le sous-système *gestion des scènes* garde un état des différents capteurs, scripts, sons et animations qui sont déclarés. Avec tout ce travail à effectuer, ce sous-système est considéré comme le centre nerveux d'un navigateur VRML. Les calculs à effectuer par ce sous-système sont souvent effectués lorsque celui de rendu des scènes ne travaille pas (*idle* -- inoccupé en français).

²⁶ Jeu de caractères des langues occidentales (norme ISO/IEC 10646-1:1993).

Le sous-système *rendu de scènes* fonctionne avec le sous-système *gestion des scènes* pour convertir la représentation interne de la scène VRML en une image à l'écran. Pour ce faire, ce sous-système récupère périodiquement la description de la scène et la redessine à chaque fois que l'utilisateur bouge le point de vue ou qu'une animation de la scène modifie un élément. Ce sous-système contient tout le pipeline graphique et effectue, par conséquent, la grande majorité des calculs. C'est donc le principal coupable lorsqu'on remarque qu'une scène n'est pas rafraîchie assez vite.

Liste d'outils VRML

Les navigateurs VRML disponibles sont nombreux. Le tableau suivant donne les plus connus d'entre eux. Nous ne nous intéressons qu'aux navigateurs ou *plugins* interprétant les fichiers VRML97. Pour chacun d'eux, nous donnons la société ou les particuliers le développant ou l'ayant développé, les machines sur lesquelles il est disponible, les bibliothèques 3D utilisées et quelques critères de comparaison. Les tirets ou les cases vides indiquent une information manquante dans ce tableau. Dans la colonne des plates-formes, Win32 indique que les bibliothèques standards utilisées sont 32 bits (donc fonctionnant aussi bien sous Windows 95, 98, 2000 que sous NT). Seule la colonne donnant la (les) bibliothèque(s) 3D utilisée(s) permet de savoir si le navigateur fonctionne sous NT. En effet, Direct3D n'est disponible, comme nous l'avons dit précédemment, que sous Windows 9x et 2000.

nom du navigateur	créateur (s)	plates-formes	API 3D	langages de script	taille texture	fichiers audio	fichiers texture	fichiers vidéo
description					URL			
Blaxxun Contact	Blaxxun interactive, Inc	Win32	OpenGL Direct3D	JavaScript	x,y = 2 ⁿ HL max	MIDI, WAV	BMP, GIF, JPEG, PPM, PNG, RGB, TGA	GIF89a
L'environnement virtuel le plus avancé parmi les produits de Blaxxun. Il permet d'utiliser différentes API 3D					http://www.blaxxun.com/products/index.html			
CASUS Presenter	Fraunhofer Institute for Computer Graphics	SGI, Solaris, Win32	OpenGL par binding Java	Java	x,y = 2 ⁿ HL max	SGI/Sol, MIDI, WAV	GIF, JPEG, XBM	
Navigateur VRML97 développé pour faire des animations. Contient un sous langage de VRML97 prévu à cet effet.					http://www.igd.fhg.de/CP/			
Community Place	SONY	Win32	Direct3D	Java JavaScript	HL max	MIDI, WAV	BMP, GIF, JPEG, RAS	?
Navigateur VRML97 supportant le partage d'un environnement entre plusieurs utilisateurs. Il englobe un outil de dialogue en mode texte et des comportements partagés écrits en Java.					http://www.community-place.com/			
Cortona	ParallelGraphics	Win32	OpenGL Direct3D Soft Rendering	Java JavaScript	Limite mémoire	MIDI, WAV	GIF, JPEG, PNG	MPEG, Flash
Navigateur VRML97 pour Netscape et Internet Explorer. Il utilise différents moteurs pour le rendu dont deux non accélérés offrant de meilleures images. Enfin, il propose de nouveaux nœuds VRML pour la gestion des NURBS ou du <i>bump mapping</i> , ...					http://www.parallelgraphics.com/cortona/			

CosmoPlayer	Platinum.	Win32, SGI, Mac	OpenGL	Java JavaScript	x,y = 2 ⁿ HL max	MIDI, WAV, AIFF/AI FC	GIF, JPEG, PNG, RGB	MPEG, Gif89a, Quick- Time
Le navigateur faisant office de référence. C'est le moteur VRML qui est reconnu comme respectant au plus près les spécifications de VRML97.					http://www.cai.com/cosmo/home.htm			
dpIV	Fighting Bull	Win32	Direct3D	Java	x,y = 2 ⁿ HL max	WAV	GIF, JPEG	MPEG
Ce navigateur permet de visualiser des scènes avec un nombres de trames constant et dans la plupart des cas élevé.					http://www.dpiv.com/swf/dpiv_m.htm			
FreeWrl	Lukka, Stewart et al	SGI, Solaris	OpenGL	Java, JavaScript	x,y = 2 ⁿ HL max	-	JPEG	-
Un navigateur <i>open source</i> . Actuellement, une interface EAI ²⁷ est en phase de développement afin de créer des environnements multi-utilisateur.					http://www.crc.ca/FreeWRL/			
Live 3d	Netscape	Mac, Win32	OpenGL	Java, JavaScript	x,y = 2 ⁿ	MIDI, WAV	GIF, JPEG	MPEG
Le navigateur développé par Netscape.					http://www.cai.com/cosmo/home.htm			
OpenWorlds	DRaW Computing	SGI, Win32, Solaris	OpenGL	Java, C++, C, lisp	Limite mémoire	MIDI, WAV	GIF, JPEG, PNG, PPM, RGB, TIFF, RAW	QuickTi me, MPEG, AVI
Un autre navigateur multi-utilisateur.					http://www.openworlds.com/			
Shout3D	EyeMatic	Toutes	Aucune	Java		-	BMP, GIF, JPEG	-
Un navigateur fonctionnant sans avoir besoin d'installer un <i>plugin</i> puisqu'il n'utilise que Java.					http://www.shout3d.com/			
SolidView	Solid Concepts Inc.	Win32	OpenGL	-		-	BMP, GIF, JPEG	-
Un navigateur dont l'interface intuitive permet de prototyper rapidement des scènes 3D.					http://www.solidview.com/			
Terraform	Brilliance Labs, Inc	Win32	Direct3D	Java	-	WAV	BMP, GIF, JPEG	AVI
Un navigateur offrant un niveau supplémentaire dans l'interaction avec la scène en autorisant la manipulation individuelle des objets.					http://www.brllabs.com/			

²⁷ *External Authoring Interface* permettant de piloter le *plugin* VRML depuis une *applet* Java.

Viscape	SuperScape	Win32	Direct3D	JavaScript	-	WAV, MIDI	BMP, GIF, JPEG	AVI, GIF89a
Viscape supporte son propre format comme nous le verons dans ce rapport en présentant les autres moteurs 3D du Web. Mais il fait également office de navigateur VRML97						http://www.superscape.com		
VRware	HyperWave	SGI, Solaris, Alpha, Linux	OpenGL	-	Limite mémoire	-	GIF, JPEG, XBM	-
Navigateur VRML97 public écrit en Java.						http://www.iicm.edu/vrwave		
VRMLview	SIM	Win32, SGI, BeOS, Linux	OpenGL	-	Limite mémoire	-	BMP, GIF, JPEG, PNG, PNM, XBM	-
Un navigateur VRML tout ce qu'il y a de plus normal excepté qu'il existe une version pour BeOS.						http://www.sim.no/vrmlview.html		
WorldProbe	Uppercut Software	Win32	OpenGL	-	-	WAV	BMP, GIF, JPEG	-
Le premier navigateur VRML97 pour Windows 9x et NT entièrement 32 bits. Il implémente un rendu stéréoscopique et l'édition de lumière pour que les utilisateurs se construisent un environnement sur mesure.						http://www.beyond-3d.com/probe/		
WorldView	Platinum	Win32, Mac	QuickDra w3D, Direct3D	JavaScript , Java (win)	?	WAV	BMP, GIF, JPEG, PNG, PPM, RAS	AVI (win)
Worldview est la deuxième référence avec Cosmo Player. Les textures peuvent être dégradées pour gagner en rapidité. Ce <i>plugin</i> semble changer de société très souvent. L'URL donnée ne sera fort probablement pas à jour très longtemps						http://www.platinum.com/worldview.html		

Et quelques traducteurs VRML freewares :

Nom du traducteur	Format d'entrée	Format de sortie
Dx2vrml	DX	VRML
IvToVRML	inventor	VRML
MacDXF2VRML	DXF	VRML (1.0 uniquement)
Obj2wrl	OBJ	VRML
Wld2vrml	VLD, PLG, FIG	VRML
Wrlgrid	Transforme les courbes mathématiques en 3D	
Vrmltranslator	DXF, 3DS, Inventor, OBJ	VRML
wc2pov v2.6	3DS, OBJ, POV	VRML, POV

3.2.2 Superscape – Viscape

La société Superscape est spécialisée dans les moteurs 3D et la création de contenu 3D depuis plus d'une dizaine d'années. Elle propose, depuis peu, un produit permettant d'évoluer dans une scène 3D chargée depuis une URL. Ce produit, Superscape Viscape, actuellement à la version 5.60, fonctionne conjointement à un navigateur HTML sur les plates-formes Win32. Il est disponible gratuitement sous la forme d'un contrôle ActiveX pour Internet Explorer ou sous la forme d'un *plugin* pour Netscape. Pour chacun de ces navigateurs, Superscape Viscape ne fonctionnera qu'avec une version 4.0 ou supérieure.

Superscape Viscape permet d'interagir avec des scènes 3D aux formats VRML97 ou SVR (créer par les modeleurs de Superscape : VRT la solution pour les entreprises, 3Dwebmaster la solution pour les professionnels ou Do3D pour les utilisateurs néophytes). Le *plugin* se décline sous trois formes différentes : Viscape SVR pour visualiser les fichiers SVR uniquement, Viscape VRML pour les fichiers VRML97 uniquement et Viscape Universe pour visualiser les deux.

Les moteurs 3D de toutes les applications de Superscape utilisent la librairie Direct3D et donc bénéficient d'une accélération matérielle. Sur le site Web de Superscape [SUPV] on peut d'ailleurs trouver un lien vers un benchmark 3D pouvant évaluer à distance, grâce à Viscape, les performances d'un ordinateur disposant des librairies Direct3D.

Le principal atout de Superscape, outre l'utilisation de Direct3D, provient de son intégration facile dans une page HTML. Tout est mis en œuvre pour faciliter les échanges d'informations entre la scène 3D et la page HTML. Les utilisateurs expérimentés pourront en effet utiliser des scripts Visual Basic (VBScript) ou ECMA (anciennement JavaScript). Viscape bénéficie également d'une interface externe utilisable avec Java et permettant à une application de changer des propriétés d'une scène au format SVR. Cette interface externe n'est néanmoins pas exploitable sur des scènes décrites avec le langage VRML. Quant à l'EAI (*External Authoring Interface*) VRML, l'interface externe entre Java et les scènes VRML, elle n'est pas supportée dans Viscape. Ce n'est pas un tort car cette interface ne fait pas encore partie des spécifications de VRML.

Bien que Viscape soit un *plugin* gratuit et les modeleurs disponibles pour une utilisation d'une durée de 30 jours, ce sont néanmoins des produits d'une société qui se garde de fournir des informations sur son format de fichier SVR. La plupart des informations sont en effet protégées par un mot de passe qui n'est donné qu'après l'acquisition d'un des modeleurs de la société. Cependant, à l'URL <http://www.superscape.com/support/viscapeinfo/index.htm>, il est possible de consulter une documentation expliquant comment créer une scène à l'intérieur d'une page HTML, comment créer des scripts ou comment utiliser l'interface externe.

3.2.3 Scol et Blaxxun

Scol est un langage de communication, une sur-couche de TCP/IP, développé par la société Cryo pour le projet "le deuxième monde" commun avec Canal+, qui établit une connexion directe et permanente entre deux points. Non seulement des données sont échangées par ce protocole mais il permet également d'exécuter des fonctions à distance dès lors que les deux machines l'autorisent. C'est à ce point qu'entre en jeu le moteur 3D puisque parmi les fonctions intégrées à Scol apparaissent des routines 3D.

En regardant de plus près les différentes possibilités de Scol²⁸, on s'aperçoit que ce langage est un véritable langage de programmation multimédia intégrant des fonctions permettant de gérer des aspects variés :

²⁸ Voir [SCOL] et plus particulièrement le lien fonctions dans la rubrique support.

- l'environnement, les fichiers,
- les graphiques 2D et 3D,
- le son et la vidéo,
- la liaison avec les navigateurs HTML,
- les communications réseaux, les protocoles SMTP²⁹ et HTML,
- les bases de données.

Scol est disponible en *plugin* pour les navigateurs Netscape et Internet Explorer versions 3.0 et supérieures. L'avantage de ce moteur par rapport à Viscap, outre qu'il intègre de nombreuses fonctions, est que de nombreux outils et un kit pour les développeurs sont disponibles gratuitement avec leurs documentations.

Aux dires de Philippe Ulrich, fondateur de Cryo, lors d'une conférence de présentation du "deuxième monde" au Conservatoire National des Arts et Métiers dans le cadre des séminaires du D.E.A. médias et multimédia, le langage Scol était destiné à être le moteur du microcosme virtuel. Cependant, en allant sur le site officiel du "deuxième monde"³⁰, nous nous rendons compte que le moteur actuel est blaxxun et non Scol. Et d'ailleurs, sur le site de Cryo, aucune phrase, aucun lien ne mentionne l'existence de ce projet commun avec Canal+, mais plutôt une communauté virtuelle appelée Cryopolis.

Aucun article, aucun commentaire, bref aucune explication concernant ce changement de moteur n'a été trouvée sur Internet. Cependant, puisque cela n'est pas vraiment le sujet de ce rapport, nous n'avons pas cherché trop longtemps à satisfaire notre curiosité, préférant à cela présenter les produits proposés par Blaxxun.

Blaxxun propose une série de produits, tous dans le domaine de la communication multimédia sur Internet. L'ensemble de ces produits offre à peu près les mêmes possibilités que le langage Scol :

- Blaxxun Contact 4.2 est le programme client gratuit permettant de visiter les sites hébergeant des environnements virtuels. Comme Scol, ce client intègre de nombreuses fonctions multimédias dont la possibilité d'afficher des scènes 3D au format VRML ou Viscap. Ce client existe sur les plates-formes Win32 sous la forme d'un *plugin*.
- Blaxxun Community Server 4.0 est le programme serveur payant mais disponible en version d'évaluation sur le site de Blaxxun³¹. Il permet de mettre en ligne des applications multimédias mêlant les aspects de communauté, de commerce et de collaboration.
- Blaxxun Community Platform SDK 4.0 est le kit de développement servant à créer une application "sur mesure" utilisant les produits Blaxxun. Il permet de modifier le côté serveur et le côté client.

3.2.4 MPEG4

MPEG4 est l'évolution naturelle, vu les possibilités multimédias actuelles, des précédentes normes MPEG (*Moving Pictures Expert Group*). Elle répond aux besoins d'intégration des différents

²⁹ *Simple Mail Transfert Protocol* : protocole servant à l'envoi des mails.

³⁰ <http://www.2nd-world.fr>.

³¹ Voir [BLAX] rubrique *Products*.

objets multimédias et d'interactions sur ceux-ci dans un contexte multi-utilisateur pour le travail coopératif.

MPEG1 permet de coder des images animées avec le son associé en vue d'une utilisation informatique. Son objectif est de rendre possible la lecture de séquences vidéos sonores à partir d'un support informatique ayant un débit de 1,5Mbits/s comme les premiers CD-ROM. Pour autoriser l'affichage de 25 images par seconde, la taille de la vidéo est fixée à 352*240 pixels, correspondant à la résolution des cassettes VHS.

MPEG2 est une évolution de MPEG1 supportant les pertes de données dues aux erreurs de transmissions. Les applications visées par cette norme sont principalement celles de diffusions télévisuelles. Les CD-ROM et DVD-ROM les plus récents ayant un débit suffisant peuvent cependant, associés à un processeur puissant ou une carte de décompression MPEG2, être un support pour les films. Dépendamment du support de diffusion, satellite, câble, réseau hertzien ou Compact Disc, les débits peuvent aller de 4 à 20Mbits/s. Outre la possibilité de coder des images entrelacées pour la télévision, MPEG2 grâce à un débit supérieur permet de transmettre 6 à 10 canaux d'images numériques pouvant aller jusqu'à la résolution PAL (704*576 pixels) dans une bande passante requise pour un seul canal analogique.

MPEG4 répond aux besoins de diffusion et d'accès aux informations multimédias sur différents types de réseaux homogènes ou hétérogènes, avec ou sans pertes de données. Cette future norme qui veut se placer comme un standard pour le codage d'informations multimédias, intègre également la possibilité d'interagir sur les objets diffusés. Cette norme permettra la transmission d'images et de sons naturels ou synthétiques 2D et 3D que l'on nomme AVOs (*Audio Visual Objects*).

MPEG4, comme toutes les normes MPEG, n'est pas qu'une définition d'un format de fichier mais une architecture logicielle complète reposant sur des spécifications détaillées pour le traitement des fichiers. Cette spécification inclut notamment les différents points suivants :

- les méthodes de codage des AVOs,
- le codage d'une scène complète composée d'AVOs agencés sous forme de graphe³²,
- le multiplexage et la synchronisation des différents flux de données contenant les AVOs,
- l'interface de communication entre les clients et le serveur MPEG4.

Pour gérer ces différents aspects se plaçant à des niveaux différents de la chaîne de restitution de l'information véhiculée sous le format MPEG4, l'architecture logicielle peut être présentée comme un ensemble de trois couches (que l'on peut voir schématisées dans la Figure 39) :

- La couche de distribution assure le transport des différents flux d'information jusqu'à l'application finale en faisant abstraction du type de réseau utilisé. Elle comprend les fonctions de multiplexage et de démultiplexage des différents flux élémentaires et la sous-couche DMIF (*Delivery Multimedia Integration Framework*). Cette sous-couche DMIF définit les interfaces de liaison avec le réseau, DNI (*DMIF Network Interface*), et l'application, DAI (*DMIF Application Interface*).
- La couche système fournit les mécanismes d'identification, de description et d'association des différents flux de données et permet la synchronisation entre les différents objets composant une scène.

³² Les données manipulées étant des objets (vidéos ou sonores, aussi bien naturels que synthétiques) composant un graphe, la compression MPEG4 est une compression de graphe au lieu d'une compression d'images rectangulaires comme dans les deux normes précédentes.

- La couche compression quant à elle effectue la compression/décompression de la scène et des différents objets la composant. La scène est codée sous la forme d'un arbre de nœuds comme en VRML auquel ont été ajoutées des fonctionnalités pour la gestion des flux audio et vidéo 2D. Ce graphe d'objets est ensuite compressé en vue d'une transmission plus rapide sous un format binaire appelé BIFS (*Binary Format For Scenes*). Chacun des objets de la scène est également compressé par son propre format de compression (les vidéos sont par exemple en MPEG2). Les objets peuvent même utiliser une compression non reconnue par la norme MPEG4 puisque le décodeur peut être transmis dans le flux de données décrivant l'objet.

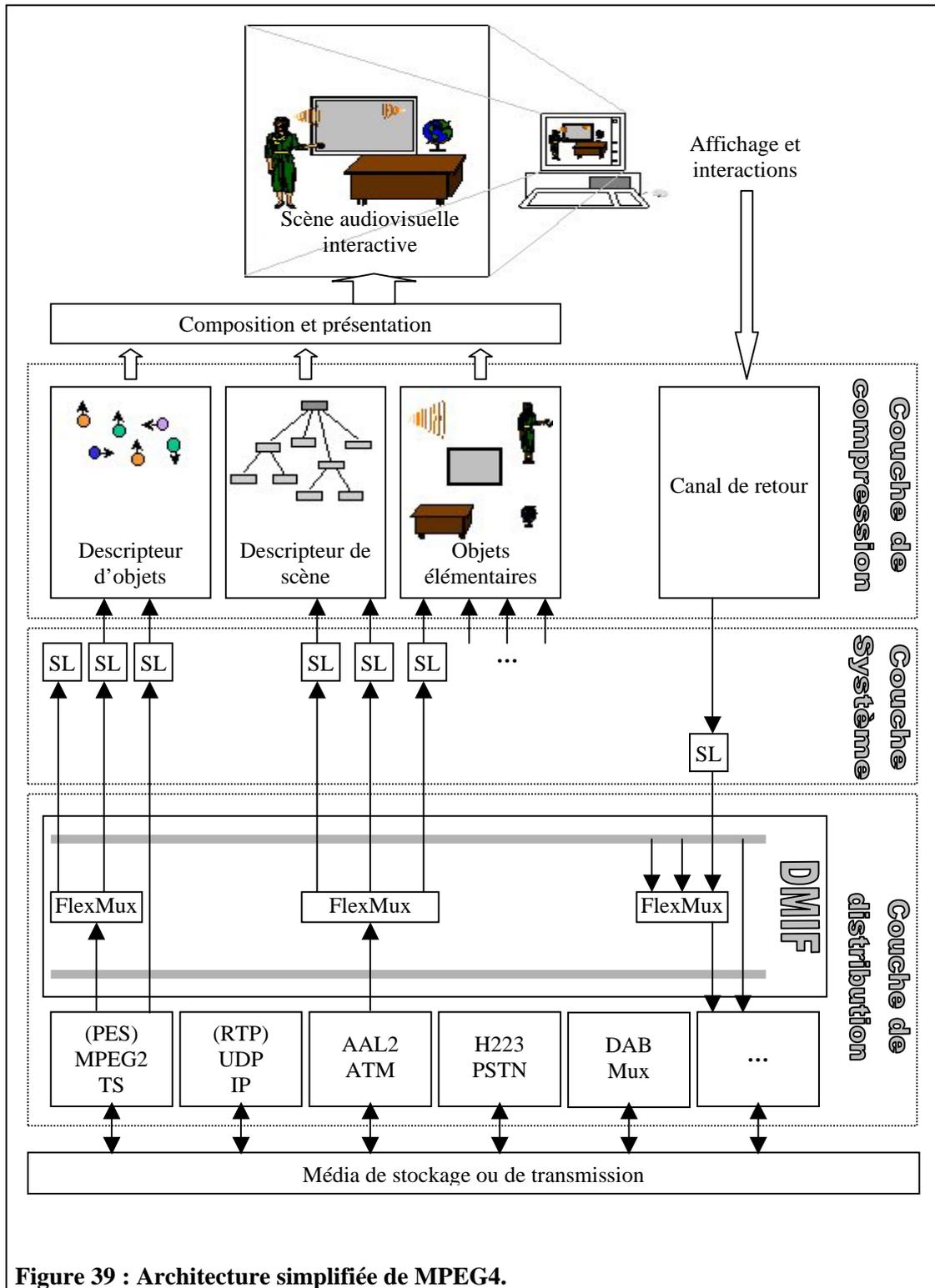


Figure 39 : Architecture simplifiée de MPEG4.

La norme MPEG4 est encore relativement floue. Jusqu'à ce qu'elle soit définitivement acceptée comme une norme ISO, la plupart des informations et des applications seront uniquement accessibles par les membres du groupe d'experts. Il nous est, par conséquent, impossible d'évaluer les possibilités 3D de MPEG4. Cependant d'après les informations présentées dans les documents [MPG1] et [MPG2] nous pouvons d'ores et déjà prévoir les avantages de cette future norme pour la 3D :

- la possibilité de réserver une qualité de service pour la transmission des flux de données,
- la compression par un facteur 10 de la scène devrait faire gagner un temps précieux lors de la récupération d'un document,
- la forte intégration des différents composants multimédias,
- la possibilité de créer des scènes multi-utilisateur avec un avatar pour chacun des membres³³,

Pour supporter le mode multi-utilisateur, les différents participants devront se connecter à un serveur. Ce serveur aura pour rôle de recevoir les actions des différents clients et, en fonction de celles-ci, de mettre à jour les scènes des autres participants.

3.2.5 Metastream

MetaCreation et Intel ont conjointement annoncé un nouveau format de fichiers non-propritaire permettant la création, le transport et la visualisation de données 3D. Les avantages principaux de ce format de fichier et de son *plugin*³⁴ associé sont les suivants :

- La taille moyenne des fichiers Metastream est plus petite (plus compressés) que n'importe quel autre format de fichiers 3D. Il en résulte un transport plus rapide des données sur le Web.
- Le *plugin* ajuste automatiquement les attributs des objets afin d'obtenir un rapport qualité/vitesse optimal.
- L'affichage des objets commence dès le début du chargement des premières faces (d'où *streaming* : le contenu s'affine avec l'arrivée de nouvelles informations dans le flux de données). La Figure 40 illustre les différentes étapes depuis le début du chargement jusqu'à l'obtention de l'apparence finale de l'objet.

L'inclusion dans une page HTML des URL vers un fichier Metastream se fait par la balise <OBJECT> ou la balise <EMBED>.

Ce format de fichier représente un travail intéressant puisqu'il apporte une solution au problème des fichiers non compressés comme VRML. D'autre part, il offre, pour la 3D, une solution à un problème fondamental d'Internet : la non-connaissance du matériel du client. Les créateurs de sites doivent en effet toujours considérer le public qu'ils visent en fonction des technologies utilisées. Dans le cas de la 3D, mais cela peut être généralisable à tous les médias gourmands en ressources, l'impossibilité de connaître les possibilités graphiques d'un client rend la création 3D difficile puisqu'il faut trouver un compromis entre la qualité de la scène et son temps de rendu chez les clients. La première solution consiste à réduire la qualité d'une scène (moins de polygones, moins de textures,

³³ La norme intégrera les objets FDP (*Facial Definition Parameters*) et FAP (*Facial Animation Parameters*) permettant d'afficher et d'animer un faciès humain en fonction des phonèmes qu'il doit prononcer ou des sentiments qu'il doit montrer.

³⁴ Voir sur le site de MetaCreations à l'URL <http://www.metastream.com>.

moins d'animations, ...) afin de créer un contenu que la majorité des clients pourront lire. La deuxième est de créer un contenu pour une certaine classe de client, ceux disposant de la plus grande puissance de calculs. Dans les deux cas, le créateur est frustré puisque, soit la scène qu'il diffuse n'est pas de la qualité souhaitée, soit elle ne s'adresse qu'à un petit nombre d'utilisateurs. Metastream est un remède à ce problème puisqu'une scène peut être déposée sur Internet avec une qualité optimale et en fonction de paramètres matériels tels que le processeur de la machine, les possibilités graphiques ou la bande passante de la liaison Internet, il ajuste automatiquement la taille de la scène et les options de rendu.



Figure 40 : Les différentes étapes du chargement d'une scène Metastream.

La dernière version du *plugin* Metastream, MTS3, offre les mêmes avantages que ses prédécesseurs : prise en compte du matériel du client pour le choix des options de rendu et *streaming* du flux de données 3D. Un effort particulier a été apporté à l'amélioration du rendu des scènes et donc à leur réalisme. De ce fait, les options suivantes ont été ajoutées par rapport à d'autres *plugins* :

- *Antialiasing* de la scène en temps-réel : l'image résultante (la trame courante) est adoucie par l'utilisation de filtres.
- *Antialiasing* progressif : l'image est raffinée en plusieurs passes (de 2 à 32) pour obtenir une image de qualité photographique.
- Projection des ombres en temps-réel : l'ajout d'ombre fournit une information supplémentaire pour apprécier la position d'un objet et pour appréhender d'une façon plus efficace sa taille.
- Coloriage Phong en temps-réel : la réflexion des lumières de ce modèle de coloriage rend les scènes plus réalistes.
- Intégration dans un page HTML : le fond de la scène peut être transparent pour que le contenu d'une page HTML se trouvant en arrière plan soit toujours visible.

3.3 Description de la navigation 3D dans ces outils

Même si les interfaces diffèrent d'un navigateur à l'autre, les fonctionnalités sont toujours très proches. La navigation dans une scène se fait généralement à l'aide de la souris. Pour la plupart des *plugins*, l'utilisateur doit choisir son mode de navigation en cliquant sur un bouton correspondant. Dans certains autres, comme le *plugin* de Metastream, l'utilisateur doit appuyer une touche tout en déplaçant la souris pour obtenir un mouvement particulier. De cette manière, les mouvements de la souris sont interprétés différemment et donnent donc un résultat différent. Nous donnons, ci-dessous, les différents mouvements possibles dans ces navigateurs. Les icônes sont tirées de Cortona de ParallelGraphics. Ces icônes ne varient que très légèrement d'un navigateur à l'autre ; elles sont en effet considérées maintenant comme étant des pictogrammes standards.

Nous donnons tout d'abord les trois métaphores pour la navigation dans une scène. Elles sont généralement présentes dans tous les *plugins*. Chacune d'elles englobe différents mouvements possibles. Elles sont donc en quelque sorte des « méta-métaphores » qui permettent de sélectionner une famille différente de mouvements. Les mouvements dont nous parlons sont tous des mouvements de caméra. Seule la métaphore pour examiner la scène (*Study*) peut être vue comme une rotation de tous les objets. Mais elle correspond également à un mouvement en orbite de la caméra autour de la scène entière.

	Permet de sélectionner des mouvements correspondants à une personne qui marche. Une gravité est appliquée pour que la caméra se trouve toujours à une certaine distance (celle que sépare les yeux des pieds) de l'objet directement en dessous.
	Ce mode simule des mouvements non assujettis à la pesanteur. Ils correspondent au vol d'un oiseau. Les mouvements de ce mode sont donc non contraints et plus libres.
	Ce mode fournit des mouvements pour examiner une scène.

Pour ces trois modes précédents, il existe donc différents déplacements. Chacun de ceux que nous donnons dans le tableau suivant n'est pas nécessairement possible dans tous les modes. Par

exemple, le mouvement *roll* correspondant à une vrille n'est pas possible dans le mode *walk*.

	Simule la marche par des translations en Z lorsque le bouton droit de la souris est appuyé et que l'utilisateur déplace le pointeur verticalement. Pour autoriser les changements de direction durant la marche, les mouvements horizontaux de la souris correspondent à des rotations selon l'axe Z.
	Effectue des translations sur l'axe des X par des déplacements horizontaux de la souris et des translations sur l'axe des Y par des déplacements verticaux.
	Rotation de la caméra selon l'axe des Y pour les mouvements horizontaux de la souris et rotation selon l'axe des X pour des mouvements horizontaux.
	Rotation de la caméra selon l'axe des Z quel que soit le mouvement de la souris.

Des outils supplémentaires sont très souvent disponibles pour des opérations particulièrement utiles :

	Zoom sur l'objet cliqué.
	Réajustement de l'assiette. L'orientation de la caméra est modifiée pour s'aligner sur le plan (X, Y).
	Les deux flèches permettent de passer d'une vue à la suivante définie dans le fichier VRML par des nœuds <i>Viewpoint</i> . En cliquant sur <i>view</i> les noms des vues apparaissent et l'on peut les sélectionner directement.
	Les paramètres de caméra sont remis aux valeurs initiales.
	La caméra est déplacée sur l'axe des Z pour contenir toute la scène visualisée dans la fenêtre.

3.4 Etude plus approfondie de VRML

VRML est un langage basé sur une organisation hiérarchique d'objets (appelés nœuds) ayant un nom et contenant des attributs (appelés champs) dont certains peuvent être modifiés par d'autres objets ou par l'utilisateur. Cette notion de graphe de scène identique à celle vue par exemple pour Java3D permet de regrouper différents nœuds devant subir les mêmes transformations. Comme Java3D donc, il existe deux types de nœuds : ceux qui servent à regrouper et opérer la même opération sur tous ses nœuds fils et ceux qui contiennent des données (formes 3D, sons, images, scripts, URL, ...). Un troisième type de nœuds existe avec VRML : ce sont des nœuds qui ne peuvent pas faire partie d'un regroupement de nœuds mais qui sont cependant attachés à un autre nœud par l'intermédiaire de l'un de ses champs.

Les champs d'un nœud sont de trois types différents :

- Les champs simples (*field*) sont les attributs du nœud et ne sont modifiables que par les

scripts,

- Les champs *eventIn* et *eventOut* reçoivent et émettent respectivement des événements ; ils ne portent pas de valeurs servant d'attributs pour le nœud,
- Les champs exposés (*exposedField*) sont des hybrides (entre les deux précédents) ; ils embarquent une valeur jouant sur l'apparence ou le comportement de l'objet et ils génèrent (lorsque leur valeur est modifiée) et reçoivent (pour modifier automatiquement leur valeur) des événements.

3.4.1 Les nœuds de regroupement

Les nœuds de regroupement sont au nombre de dix. Tous sont considérés comme des nœuds de regroupement parce qu'ils ont un champ pouvant contenir plusieurs autres nœuds. Quel que soit le nœud de regroupement utilisé, ses enfants subissent les mêmes transformations de telle manière que le système de coordonnées des enfants est relatif à celui du nœud de regroupement. On peut alors parler de nœuds parents puisqu'ils transmettent leurs propriétés à leurs enfants. Tous ces nœuds parents peuvent également être inclus dans un autre nœud parent, ce qui a pour effet la création de la hiérarchie sur plusieurs niveaux. Tous les nœuds de regroupement (excepté les nœuds *LOD*, *Switch* et les prototypes) contiennent les deux champs *bboxCenter* et *bboxSize* servant à définir une boîte englobant tous les nœuds de sa descendance. Cette boîte englobante est utilisée pour optimiser le rendu du moteur VRML et particulièrement pour éliminer rapidement les groupes de nœuds non visibles. Les 10 nœuds parents ainsi que le comportement qu'ils transmettent à leur descendance sont donnés dans le tableau suivant :

<i>Anchor</i>	Si l'utilisateur presse le bouton de la souris alors que le pointeur est sur un objet graphique 3D de la descendance de ce nœud de regroupement alors le navigateur HTML charge l'URL donnée dans le champ <i>url</i> du nœud <i>Anchor</i> .
<i>Billboard</i>	Les axes du système de coordonnées de ce nœud de regroupement sont modifiés de telle manière que l'axe Z pointe toujours vers la caméra (le point de vue). Toutes les formes 3D faisant partie de la descendance de ce nœud subissent par conséquent cette même transformation. Il en résulte que l'utilisateur perçoit toujours le groupe d'objets de la même façon quel que soit son angle de vue.
<i>Collision</i>	Ce nœud utilise la boîte englobante pour les calculs de détection de collision.
<i>Group</i>	Ce nœud sert uniquement à regrouper d'autres nœuds. Il ne transmet pas de comportement spécial à sa descendance mais permet de définir la boîte englobante tous les nœuds contenus pour effectuer rapidement le test de visibilité de ses descendants.
<i>Inline</i>	Ce nœud de regroupement recopie les données VRML présentes dans une URL. Il n'a pas de champ <i>children</i> contenant de multiples nœuds mais un champ <i>url</i> . Cependant tous les nœuds dans le fichier externe sont bien considérés comme étant des descendants.
<i>LOD</i>	Ce nœud permet de spécifier plusieurs niveaux de détails d'un même groupe d'objets en fonction de la distance à laquelle il se trouve du point de vue.

<i>Switch</i>	Ce nœud permet, en fonction de la valeur numérique de son champ <i>whichChoice</i> , d'afficher le groupe d'objets correspondant. Il est très utile pour afficher des objets dont on peut modifier l'état (interrupteur, ascenseur, ...).
<i>Transform</i>	Les descendants de ce nœud subissent les transformations spécifiées par les champs <i>rotation</i> , <i>translation</i> , <i>scale</i> et <i>scaleOrientation</i> .
<i>PROTO</i>	Ce type de nœuds (et le suivant) permet de définir des prototypes. Les prototypes servent à créer des nouveaux nœuds ayant des attributs propres à partir des nœuds existants dans la spécification de VRML. Il existe, pour cela, un système de passage de paramètres pour que les attributs du nouveau nœud correspondent à des attributs des nœuds qui le constitue.
<i>EXTERNPROTO</i>	Ce nœud donne l'URL d'un prototype externe au fichier dans lequel il se trouve.

3.4.2 Les nœuds enfants

Les nœuds enfants valides (que l'on peut ajouter dans la liste des descendants d'un nœud de regroupement) proviennent de 9 familles différentes. Pour chacune d'elles, nous verrons leur utilité, les nœuds la composant et leur description.

- Les nœuds parents vus précédemment.
- Les lumières : les trois nœuds servant à définir les trois types de lumières utilisables dans une scène VRML. Chacune d'elles comporte les champs *intensity* (donnant l'intensité, l'éclat), *color* (donnant la couleur de la lumière) et *ambientIntensity* (donnant l'intensité de la lumière ambiante diffusée par cette lumière). Elles diffèrent l'une de l'autre par le type d'éclairage :

<i>DirectionnalLight</i>	Illumine tous les descendants du nœud qui la contient.
<i>PointLight</i>	Illumine tous les objets de la scène qui se trouvent dans sa zone d'influence sphérique.
<i>SpotLight</i>	Illumine tous les objets de la scène qui se trouvent dans sa zone d'influence conique.

- Les sondeurs : les 7 nœuds suivants servent à déterminer le type d'événements qui doit être récupéré. A ces sept nœuds peut être rajouté le nœud *Anchor* puisqu'il définit également un événement. Mais puisqu'il englobe d'autres nœuds, nous préférons le classer parmi les nœuds de regroupement.

<i>CylinderSensor</i>	Événement activé lorsque le pointeur de la souris passe sur un descendant du nœud parent contenant le sondeur et que les coordonnées 3D de la souris, transformées en coordonnées cylindriques, pointent dans une zone cylindrique définie.
-----------------------	---

<i>PlaneSensor</i>	Événement activé lorsque le pointeur de la souris passe sur un descendant du nœud parent contenant le sondeur et que les coordonnées de la souris, projetées sur un plan, pointent dans une zone rectangulaire définie.
<i>ProximitySensor</i>	Événement activé lorsque le point de vue entre, sort ou est déplacé dans la zone définie par un pavé.
<i>SphereSensor</i>	Événement activé lorsque le pointeur de la souris passe sur un descendant du nœud parent contenant le sondeur et que les coordonnées du pointeur, en coordonnées sphériques, sont dans une zone conique définie.
<i>TimeSensor</i>	Événement activé soit à un moment précis soit à une fréquence précise.
<i>TouchSensor</i>	Événement activé lorsque le pointeur de la souris passe sur un descendant du nœud parent contenant le sondeur.
<i>VisibilitySensor</i>	Événement activé lorsqu'une zone définie par un pavé change d'état (devient visible ou invisible) en fonction des mouvements du point de vue.

- Les interpolateurs : ces 6 nœuds calculent des valeurs d'un type donné et déclenchent un événement indiquant un changement d'état permettant de mettre en place des animations. Le calcul des valeurs se fait par interpolation linéaire entre deux valeurs successives données.

<i>ColorInterpolator</i>	Interpolateur de couleurs.
<i>CoordinateInterpolator</i>	Interpolateur de coordonnées.
<i>NormalInterpolator</i>	Interpolateur de normales.
<i>OrientationInterpolator</i>	Interpolateur de rotations.
<i>PositionInterpolator</i>	Interpolateur de positions.
<i>ScalarInterpolator</i>	Interpolateur de flottants.

- Les nœuds liés : ces 4 nœuds sont gérés dans une pile afin qu'uniquement un de chaque sorte puisse être actif à un moment donné mais que l'on puisse les changer.

<i>Background</i>	Définit le type de fond : couleur, texture.
<i>Fog</i>	Utilisé pour simuler des effets atmosphériques.
<i>NavigationInfo</i>	Contient des informations sur la navigation comme le type (marcher, voler, ...), la vitesse, la limite de visibilité, la simulation d'une lumière portée par l'utilisateur, et la taille de l'utilisateur pour calculer les collisions.

<i>Viewpoint</i>	Définit différents points de vue sur la scène.
------------------	--

- Les objets graphiques et sonores : ces deux objets sont les seuls permettant d'ajouter dans la scène des objets perceptibles par l'utilisateur.

<i>Shape</i>	Permet d'ajouter un objet graphique 3D. Ce nœud est composé de deux champs attendant d'autres nœuds (dépendants). Le champ <i>geometry</i> doit référencer un nœud de géométrie et le champ <i>appearance</i> doit toujours référencer un nœud de type <i>Appearance</i> .
<i>Sound</i>	Permet d'ajouter un son à la scène. Son champ <i>source</i> référence soit un nœud <i>AudioClip</i> soit un nœud <i>MovieTexture</i> .

- Le nœud englobant les scripts :

<i>Script</i>	Ce nœud permet d'écrire des morceaux de codes effectuant des opérations sur des nœuds accessibles depuis des nœuds instanciés par le mot réservé <i>DEF</i> . Il sert principalement pour les animations mais également pour gérer la communication et l'affichage des différents avatars dans une scène.
---------------	---

- Le nœud d'information

<i>WorldInfo</i>	Ce nœud sert uniquement pour donner un titre et donner des informations sur la scène VRML mais n'est pas utilisé pour le rendu.
------------------	---

- Les instances de nœud : ce type de nœud ressemble aux prototypes. Cependant, là où les prototypes peuvent être appelés avec des paramètres (pour changer la couleur, la position, la taille ou quoi que ce soit d'autre), les appels d'instances sont une recopie intégrale d'un nœud défini précédemment. Pour utiliser une instance d'un nœud, il suffit de lui avoir préalablement donné un nom à l'aide de la commande *DEF*. Il est ainsi rendu utilisable grâce au mot-clé *USE* par tous les nœuds se trouvant plus loin dans le fichier VRML (et pas seulement ceux de sa descendance). Alors que les prototypes sont assimilables à des appels de procédures puisqu'il y a passage de paramètres, les instances sont plutôt des macros.

3.4.3 Les nœuds dépendants

Le troisième type de nœuds dont nous parlions précédemment sont les nœuds qui ne peuvent pas être des nœuds parents et qui ne peuvent pas être contenus dans un nœud parent. Ces nœuds peuvent, en effet, être utilisés qu'en attribut d'un autre nœud. Ce sont des nœuds que nous appellerons dépendants, donnant les formes géométriques, les propriétés des formes géométriques, les apparences et les propriétés des apparences. Nous les donnons dans le tableau suivant ainsi que les champs des nœuds auxquels ils peuvent s'attacher (nœud + champ entre parenthèses) et une description succincte de leur utilité.

<i>Appearance</i>	<i>Shape (appearance)</i>	Regroupe les options d'apparence de la forme 3D définie dans le champ <i>geometry</i> du nœud <i>Shape</i> .
<i>AudioClip</i>	<i>Sound (source)</i>	Donne l'URL du son à jouer par le nœud <i>Sound</i> .
<i>Box</i>	<i>Shape (geometry)</i>	Nœud créant une boîte.
<i>Color</i>	<i>ElevationGrid (color)</i> <i>IndexedFaceSet (color)</i> <i>IndexedLineSet (color)</i>	Permet de spécifier le triplet RGB d'une couleur. A ne pas confondre avec les champs de type <i>SFColor</i> .
<i>Cone</i>	<i>Shape (geometry)</i>	Nœud créant un cône
<i>Coordinate</i>	<i>IndexedFaceSet (coord)</i> <i>IndexedLineSet (coord)</i>	Permet de spécifier une liste de coordonnées de sommets.
<i>Cylinder</i>	<i>Shape (geometry)</i>	Nœud créant un cylindre.
<i>ElevationGrid</i>	<i>Shape (geometry)</i>	Nœud créant une grille d'altitudes permettant de décrire des reliefs.
<i>Extrusion</i>	<i>Shape (geometry)</i>	Nœud créant une forme 3D par extrusion.
<i>FontStyle</i>	<i>Text (fontStyle)</i>	Regroupe les options de tracé d'un texte.
<i>ImageTexture</i>	<i>Appearance (texture)</i>	Permet de spécifier une texture fixe pour l'apparence d'un objet.
<i>IndexedFaceSet</i>	<i>Shape (geometry)</i>	Nœud créant un ensemble de faces d'après leurs sommets.
<i>IndexedLineSet</i>	<i>Shape (geometry)</i>	Nœud créant un ensemble de lignes brisées.
<i>Material</i>	<i>Appearance (material)</i>	Permet de donner les propriétés du matériel d'un objet.
<i>MovieTexture</i>	<i>Appearance (texture)</i> <i>Sound (source)</i>	Permet de spécifier une texture animée comme apparence d'un objet ou d'utiliser la piste sonore pour un nœud <i>Sound</i> .
<i>Normal</i>	<i>ElevationGrid (normal)</i> <i>IndexedFaceSet (normal)</i> <i>IndexedLineSet (normal)</i>	Permet de spécifier une liste de coordonnées de normales.
<i>PointSet</i>	<i>Shape (geometry)</i>	Nœud créant un point.
<i>Sphere</i>	<i>Shape (geometry)</i>	Nœud créant une sphère.

<i>Text</i>	<i>Shape (geometry)</i>	Nœud créant un texte.
<i>TextureCoordinate</i>	<i>ElevationGrid (texCoord)</i> <i>IndexedFaceSet (texCoord)</i>	Permet de spécifier une liste de coordonnées de texture.
<i>TextureTransform</i>	<i>Appearance (textureTransform)</i>	Donne les transformations à appliquer à une texture avant de la plaquer sur un objet.

3.4.4 Le format de fichier

Les fichiers VRML portent l'extension *.wrl* (abréviation de *world*) et ont pour type MIME *model/vrml*. Cependant pour assurer une compatibilité avec les versions précédentes de VRML, un *plugin* devra également accepter le type *x-world/x-vrml*. Quelques co-navigateurs VRML associent aussi les fichiers ayant une extension *.gz* ou *.wgz* pour ouvrir les fichiers compressés par le programme *gzip*.

La structure générale d'un fichier VRML consiste en quatre parties successives : l'entête, les prototypes, le graphe de scène et le routage des événements, que nous décrivons une à une.

L'entête

Pour pouvoir identifier un fichier VRML, celui-ci devra toujours commencer par une ligne de la forme :

```
#VRML <version> <type d'encodage> [commentaires]
```

Les deux versions à l'heure actuelle sont *V1.0* pour VRML 1.0 et *V2.0* pour VRML 2.0 ou 97. Les deux types d'encodage de caractères supportés sont *ASCII* pour les fichiers VRML 1.0 et *utf8* pour les fichiers VRML 2.0/97. Ainsi les deux déclarations suivantes seront valables respectivement pour les fichiers au format VRML 1.0 et les fichiers au format VRML 2.0/97 :

```
#VRML V1.0 ascii    mon fichier VRML 1.0
#VRML V2.0 utf8    mon fichier VRML 97
```

Les prototypes

Les prototypes sont généralement placés après l'entête. Comme nous l'avons vu précédemment, ils permettent de créer un nouveau nœud à partir de nœuds VRML valides et/ou d'autres prototypes qu'il déclare lui-même. Dans le cas d'un prototype externe (*EXTERNPROTO*), le prototype est référencé par une URL. L'URL contenant le prototype devra alors être un fichier VRML valide ; c'est à dire commencer par l'entête et il devra également contenir la définition du prototype.

L'exemple suivant montre l'utilisation d'un prototype externe. Dans la colonne de gauche du tableau se trouve le fichier VRML que nous souhaitons charger. Il contient la déclaration d'un prototype externe qui se trouve dans un fichier nommé *TwoColorTable.wrl* que nous avons placé dans la colonne de droite du tableau.

<pre>#VRML V2.0 utf8 EXTERNPROTO TwoColorTable [field SFCOLOR legColor field SFCOLOR topColor] ["twoColorTable.wrl"]</pre>	<pre>#VRML V2.0 utf8 PROTO TwoColorTable [Field SFCOLOR legColor .8 .4 .7 Field SFCOLOR topColor .6 .6 .1] {</pre>
---	---

```

Transform {
  translation 0 0 -2
  children [
    TwoColorTable {
      topColor .7 0 0
    }
  ]
}

Transform {
  translation 0 0 2
  children [
    TwoColorTable {
      legColor 0 .7 0
    }
  ]
}

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1.0 1.0 1.0
    }
  }
  geometry Cylinder {}
}

```

```

Transform {
  Children [
    Transform { # plateau
      translation 0 0.6 0
      children [
        Shape {
          appearance Appearance {
            material Material {
              diffuseColor IS topColor
            }
          } # fin Appearance
          geometry Box {
            size 1.2 0.2 1.2
          } # fin Shape
        ] # fin children
      ] # fin Transform
    }

    Transform { # premier pied
      Translation -0.5 0 -0.5
      children [
        DEF Leg Shape {
          appearance Appearance {
            material Material {
              diffuseColor IS legColor
            }
          } # fin Appearance
          geometry Cylinder {
            height 1 radius .1
          } # fin Shape
        ] # fin children
      ] # fin Transform
    }

    Transform { # deuxième pied
      Translation .5 0 -0.5
      Children USE Leg
    }

    Transform { # troisième pied
      Translation -0.5 0 .5
      children USE Leg
    }

    Transform { # dernier pied
      Translation .5 0 .5
      children USE Leg
    }
  ] # fin du children principal
} # fin du Transform principal
} # fin du prototype

```

Ce prototype est très intéressant car il illustre deux possibilités offertes par VRML : le prototype et les instances de nœuds. Tout d'abord, il déclare le prototype *TwoColorTable* comportant deux attributs *legColor* et *topColor* correspondant, dans le modèle, respectivement à la couleur des pieds et du plateau de la table. Chacun de ces champs comporte une valeur par défaut (comme tous les champs VRML de tous les nœuds) dans le cas où elle serait omise lors de l'utilisation du nœud. Vient ensuite le corps du prototype, composé de nœuds VRML valides. Le plateau est tout d'abord défini par une boîte subissant une légère translation pour se placer correctement et ayant pour couleur diffuse *topColor* (affectée par le mot-clé *IS*). Puis un premier pied est défini comme étant un cylindre de

couleur *legColor* placé sous la table en un de ses coins. Cette forme est instanciée par le mot réservé *DEF* sous le nom *leg* afin d'être utilisée pour les trois derniers pieds grâce au mot réservé *USE*. Chacun des 3 derniers pieds de la table subit cependant une translation afin qu'il soit placé correctement.

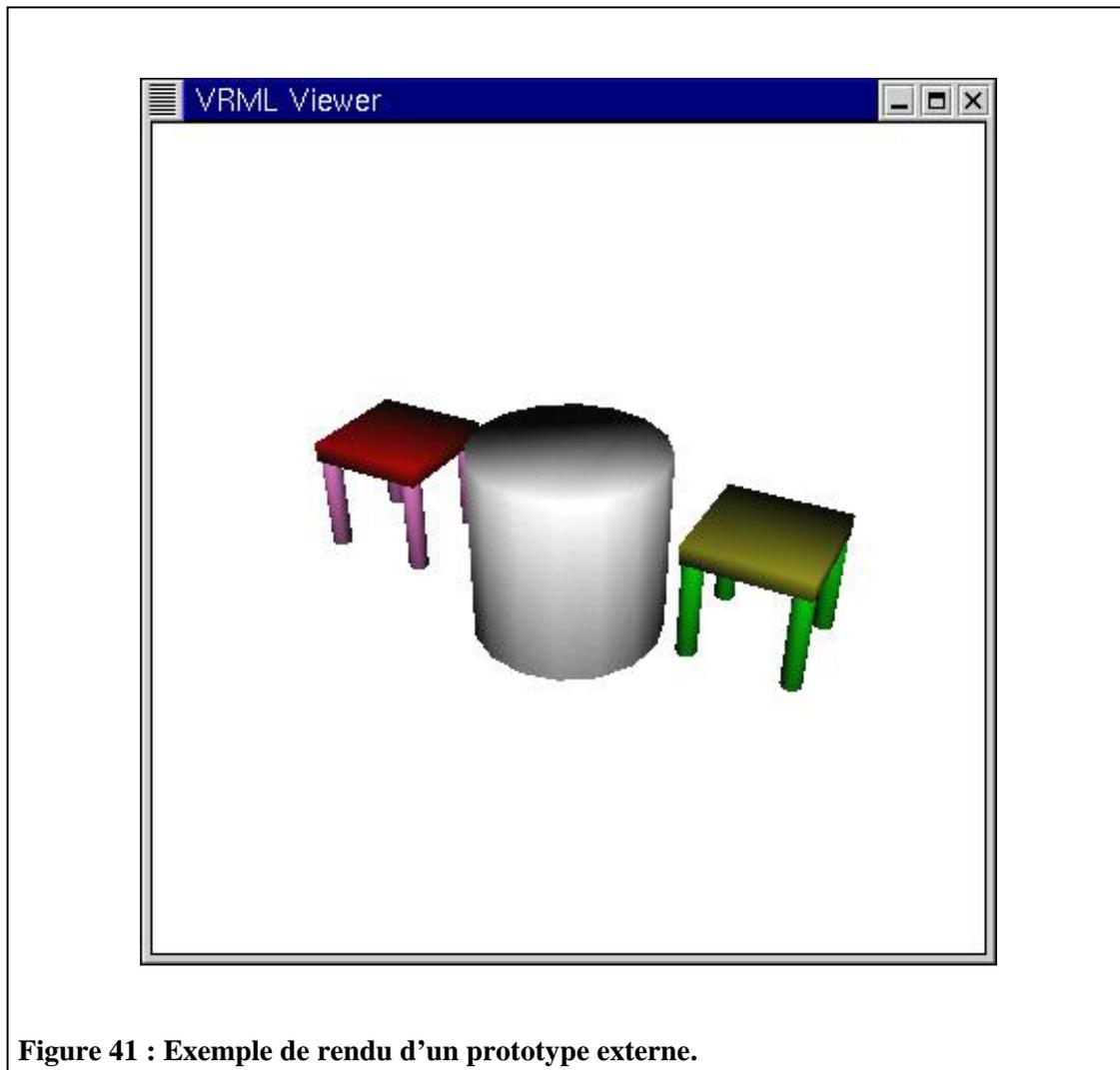


Figure 41 : Exemple de rendu d'un prototype externe.

Dans le fichier principal, l'utilisation de notre objet *TwoColorTable* se fait comme un nœud VRML valide maintenant qu'il a été déclaré. Le résultat obtenu est montré dans la Figure 41. Le rendu de la scène est effectué notre propre navigateur VRML que nous étudierons plus tard. Dans le fichier principal, nous utilisons deux fois le prototype :

- la première fois avec une couleur de plateau rouge et avec la couleur de pied par défaut (déclarée dans la définition du prototype (mauve)),
- la deuxième fois avec la couleur de plateau par défaut (kaki) et avec la couleur verte pour les pieds.

Le graphe de scène

Le graphe de scène présentant la hiérarchie des nœuds constitue la troisième partie d'un fichier VRML. Nous pouvons voir cette structure dans les deux fichiers précédents grâce à l'imbrication des différents éléments. Le schéma suivant correspond au graphe de scène du corps de notre prototype *TwoColorTable*.

4 UN NAVIGATEUR VRML

Résumé Cette partie décrit les différentes opérations nécessaires pour afficher et interagir avec le contenu d'un fichier décrivant une scène 3D. La scène, décrite en langage VRML, est tout d'abord vérifiée lexicalement et syntaxiquement. Durant cette étape, l'arbre de la scène est construit. Il est utilisé pour générer les fonctions OpenGL correspondantes. Ces différentes opérations sont illustrées concrètement par notre propre navigateur VRML écrit en langage C. Les options de navigation et de rendu que nous avons implémentées sont ensuite décrites en détails.

La première étape du processus menant à la visualisation d'une scène décrite dans un fichier au format VRML est la même que celles que soient les applications ou les *plugins* écrits. Il s'agit de lire le fichier source, de déterminer si tous les mots utilisés font partis de la syntaxe VRML et si l'ensemble de ceux-ci forme une expression valide. Ce travail est pris en charge par le sous-système *Chargeur de scènes* des *plugins* (voir la partie 3.2.1 page 74). Plus généralement, cette première étape est une constante lorsque l'on développe un traducteur ou un compilateur.

Connaissant les différentes unités lexicales et la grammaire des expressions bien formées d'un langage, il est relativement facile de mettre en œuvre un traducteur. Dans la plupart des cas (et dans le nôtre aussi), les expressions bien formées sont traduites en une structure arborescente interne (en mémoire vive, par opposition au fichier source qui est stocké sur un disque) exploitable et manipulable plus facilement.

Pour réaliser ce *parser* de fichiers VRML, nous avons utilisé la grammaire présente dans la spécification de VRML97 [WRL2] à laquelle nous avons adjoint quelques règles supplémentaires pour reconnaître les fichiers au format VRML1 [WRL1]. A partir de ces deux grammaires, nous avons créé une grammaire conforme à l'outil Yacc qui associé à Lex³⁵ permet de mettre en œuvre très facilement des traducteurs et compilateurs. Ces deux outils attendent en entrée leurs propres grammaires de reconnaissance et génèrent, en sortie, un code que nous pouvons ensuite compiler et lier dans notre programme qui doit utiliser ce *parser*. Dans notre cas, nous avons utilisé deux outils qui génèrent du code en langage C puisque nos programmes sont écrits dans ce langage.

Pour visualiser une scène VRML avec Java3D la problématique est différente puisque les outils Yacc et Lex n'existent pas pour l'environnement Java. Cependant, il existe quelques paquetages publics permettant de lire un fichier VRML et de le transformer en une structure de graphe de scène Java3D. L'écriture d'un navigateur VRML utilisant l'API Java3D est par conséquent grandement simplifiée et ne prend que quelques heures puisqu'il suffit de programmer les interactions (qui peuvent être basiques et il suffit alors d'utiliser les classes *Behavior* prédéfinies prévues à cet effet).

De nombreuses autres bibliothèques existent en Java permettant, d'une part, de faire les analyses lexicales et syntaxiques de fichiers VRML et, d'autre part, de faire un rendu 3D en passant par des fonctions OpenGL. Les premières utilisent des grammaires écrites à l'aide du compilateur de compilateurs JavaCC³⁶ (*Java Compiler Compiler*). JavaCC opère de la même façon que l'ensemble *Lex* et *Yacc* ; c'est à dire qu'il lit la spécification de la grammaire et la transforme en un programme Java qui peut reconnaître les expressions. De nombreuses grammaires permettant de « parser » différents types de fichiers sont également disponibles. Entre autres, les grammaires de VRML et des scripts ECMA permettent respectivement de parcourir la partie VRML et le contenu d'un nœud *Script*.

³⁵ De nombreux ouvrages sont consacrés à ces outils. En particulier, la "bible" en matière de compilation [AHO91] dit "le dragon" et les ouvrages qui nous ont particulièrement aidé [SIL95] [BEN91].

³⁶ Téléchargement et documentation à l'URL <http://www.metamata.com/JavaCC/>.

Les paquetages Java permettant d'effectuer le rendu sont les *OpenGL bindings*. Ceux-ci ont pour but de lier une application Java avec les fonctions OpenGL natives rassemblées sous forme d'une bibliothèque dynamique. Il existe de nombreuses implémentations de ces *OpenGL bindings*³⁷. La plupart sont disponibles pour différentes plates-formes.

L'ensemble grammaire de description en JavaCC et *OpenGL bindings* est une alternative à Java3D. Ses principaux avantages viennent des nombreux systèmes supportés, ne les limitant pas à ceux pour lesquels une implémentation de Java3D existe. D'autre part, il permet un plus grand contrôle sur la génération des commandes 3D. Une étude de cette solution est en cours dans le cadre du mémoire d'ingénieur CNAM de Jean-Marie Abisror à l'heure où est écrit ce rapport.

4.1 Les différentes étapes

Pour réaliser ce compilateur sous Linux, nous avons utilisé les utilitaires *flex* et *bison*, faisant partie des programmes GNU³⁸ (*GNU's Not Unix*) libres d'utilisation car faisant partie de la FSF³⁹ (*Free Software Foundation*) et compatibles avec les outils *Lex* et *Yacc* de Sun. Sous Windows, ces outils peuvent être récupérés dans l'environnement de développement GNU, fonctionnant dans une console MSDOS, appelé DJGPP⁴⁰ (*DJ Delorie's GNU Programming Platform*).

4.1.1 Analyse lexicale avec Lex

Lex effectue l'analyse lexicale des mots du langage traité. Il sert à définir les règles de reconnaissance des unités lexicales du langage (ou lexèmes), à éliminer les informations non utiles telles que les séparateurs et les commentaires et à lever les erreurs sémantiques de base (type de valeur attendue non présente, ...) allégeant ainsi le travail de l'analyseur syntaxique.

Son fonctionnement est celui de n'importe quel système de *pattern-matching* : il reçoit en entrée les caractères de l'expression à reconnaître et lorsqu'un ensemble de caractères lu dans le fichier source forme un lexème préalablement défini, il exécute les actions associées et il envoie ce lexème en sortie (généralement une valeur numérique définissant ce lexème) avec ses attributs (comme sa valeur par exemple).

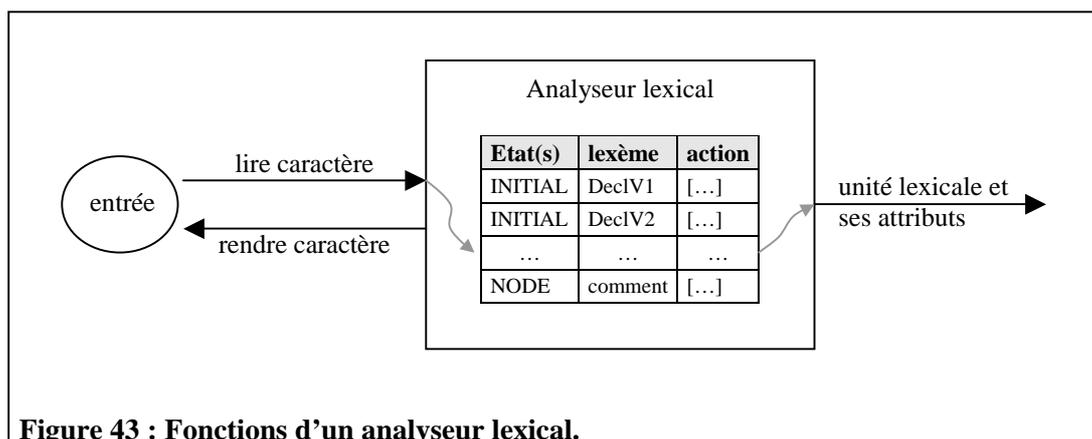


Figure 43 : Fonctions d'un analyseur lexical.

³⁷ La plupart sont disponibles à l'URL <http://www.opengl.org/Documentation/Implementations/Languages.html>.

³⁸ Projet démarré en 1984 visant à créer une suite gratuite de programmes comme ceux que l'on peut trouver sous les systèmes UNIX payants (voir <http://www.gnu.org>).

³⁹ La philosophie et l'intérêt de la distribution de logiciels libres de droits sont expliqués à l'URL suivante sur le site de la FSF : <http://www.fsf.org/home.fr.html>.

⁴⁰ Téléchargement et documentations sur le site <http://www.delorie.com/djgpp>.

De plus, Lex est construit sur le concept d'automate déterministe à états finis. Ainsi, des opérations distinctes pourront être exécutées pour un même lexème reconnu en fonction de l'état dans lequel l'automate se trouve au moment de la reconnaissance. Les différents lexèmes et les différents états fournis par le programmeur sont traduits par Lex en un automate déterministe implanté sous forme de tables. Exécuter l'analyse lexicale ne consiste alors qu'à parcourir le graphe déterministe sur les données fournies par le fichier source.

L'analyseur lexical met en œuvre, d'une part, des techniques spécialisées de gestion des tampons pour la mémorisation des caractères lus et, d'autre part, tout un automate de reconnaissance des symboles. Lex effectuant lui-même ces tâches à partir des règles qu'on lui fournit, il permet ainsi d'économiser un temps précieux puisque nous n'avons pas besoin de programmer nous même ces fonctions d'un analyseur lexical.

A partir des unités lexicales valides définies dans un fichier à l'aide d'une syntaxe simple, Lex génère l'automate en langage source et fournit la fonction *yylex* permettant de récupérer le lexème suivant et de changer, en fonction de celui-ci, l'état courant de l'automate. L'exemple suivant montre comment définir une règle reconnaissant la déclaration au début d'un fichier VRML, celle reconnaissant un commentaire et celle reconnaissant un identificateur et renvoyant sa valeur sous forme de chaîne de caractères.

```
// Les définitions des lexèmes de base
// Les séparateurs = espace ou tabulation ou retour chariot ou virgule
wsnml [ \t\r,]

// retour chariot sous Unix (Line Feed) et Dos (Line Feed+Carriage Return)
newline (\n)|(\n\r)

// La déclaration = texte de la déclaration jusqu'à la fin de ligne
declarationV2 ("#VRML V2.0 utf8")(.* )
declarationV1 ("#VRML V1.0 ascii")(.* )

// Les commentaires = tous les caractères après un '#'
comment (#.* )

// Les identificateurs
// Les caractères autorisés en début de chaîne
idStartChar ([^\x30-\x39\x00-\x20\x22\x23\x27\x2b-\x2e\x5b-\x5d\x7b\x7d])
// Les caractères autorisés après le premier caractère
idRestChar ([^\x00-\x20\x22\x23\x27\x2c\x2e\x5b-\x5d\x7b\x7d])

// Les règles
// Dans l'état INITIAL, si l'on rencontre une chaîne de déclaration, on
// passe dans l'état NODE
<INITIAL>{declarationV1}          { version=1; BEGIN NODE; }
<INITIAL>{declarationV2}          { version=2; BEGIN NODE; }

// Dans l'état NODE, si l'on rencontre un identificateur, on retourne le
// type IDENTIFIER et sa valeur, contenue dans yytext.
<NODE>{idStartChar}{idRestChar}*  { yylval.string = strdup(yytext);
                                     return IDENTIFIER; }

// Dans les états INITIAL et NODE, si l'on rencontre un commentaire, on ne
// fait rien (on le saute).
<INITIAL,NODE>{comment}           ;

// Dans les états INITIAL et NODE, si l'on rencontre un saut de ligne, on
// incrémente le numéro de la ligne courante et on initialise à 0 une
// variable gardant le nombre d'erreurs détecté dans la ligne courante.
<INITIAL,NODE>{newline}           { ++lineNum; lineNumber=0; }
```

4.1.2 Analyse syntaxique avec Yacc

Pour les langages nécessitant une analyse syntaxique (ceux dans lesquels une action élémentaire peut se composer de plusieurs lexèmes et donc où l'on doit vérifier que cet ensemble forme une expression valide), les lexèmes reconnus sont généralement récupérés par l'outil Yacc qui effectue cette deuxième étape. A l'aide des grammaires hors-contextes non ambiguës qu'il manipule, l'outil Yacc permet de spécifier des règles reconnaissant des expressions (ensemble de lexèmes) bien formées d'un langage. En d'autres termes, il vérifie que les données qu'on lui soumet respectent les règles définies du langage. Il permet également d'associer diverses opérations à effectuer lorsque certaines expressions sont reconnues. Dans notre cas, la plupart des actions que nous intégrerons, auront pour but la construction d'un arbre de l'expression.

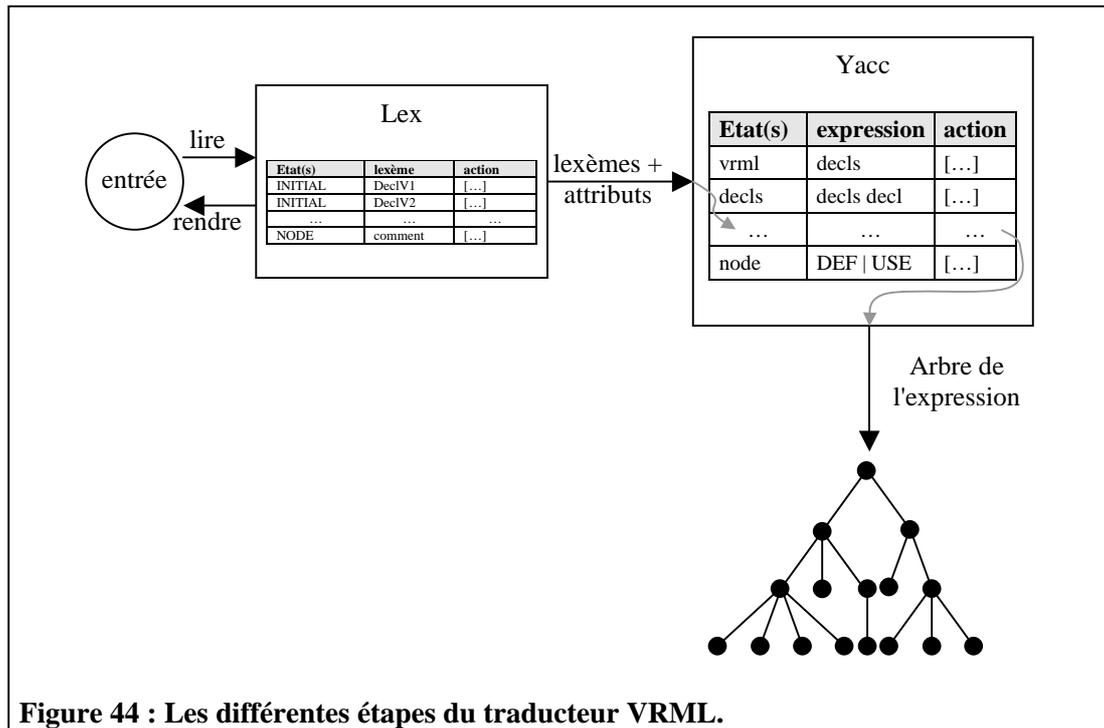


Figure 44 : Les différentes étapes du traducteur VRML.

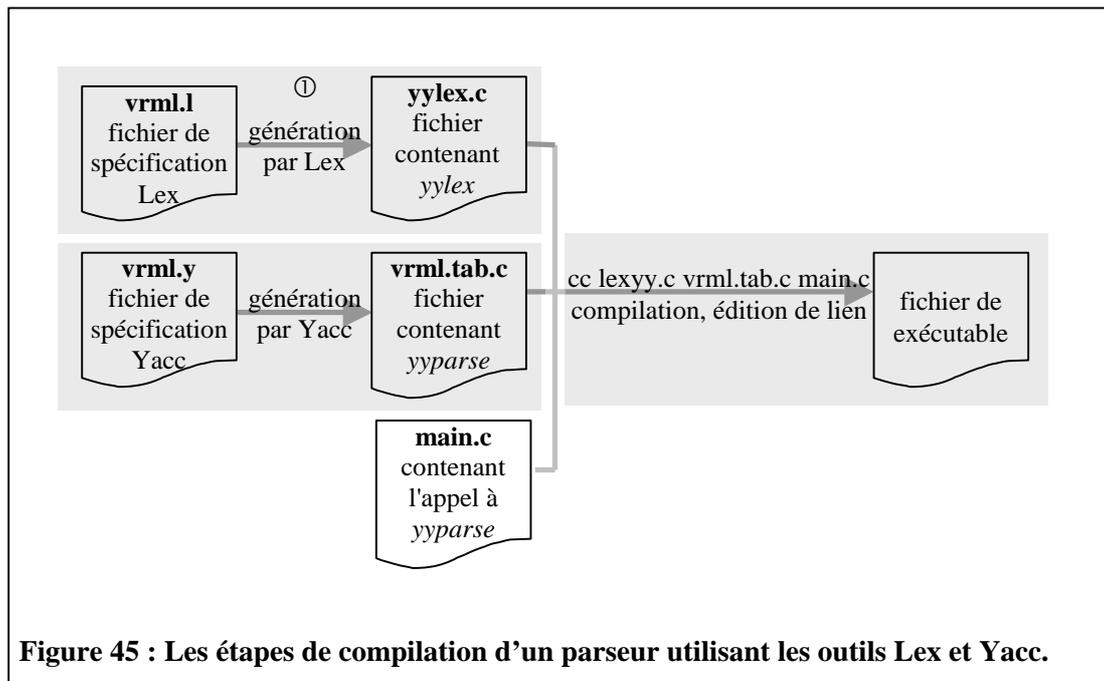
L'idée de base du fonctionnement de Yacc est très proche de celle de Lex (c'est-à-dire à base d'un automate d'états finis). Cependant, pour reconnaître les langages hors-contextes, le concept d'automate à états finis n'est pas suffisant. Il a donc fallu lui adjoindre une pile d'une capacité théoriquement illimitée, formant ainsi un automate à pile. Tout comme un automate déterministe, un automate à pile effectue des transitions d'états en fonction de la donnée en entrée, mais en plus, il tient comptes des symboles qui se trouvent au sommet de la pile et les modifie selon les besoins. L'analyse effectuée par Yacc est de type ascendante. Les expressions sont analysées de gauche à droite en utilisant une dérivation (réduction des expressions) à droite.

Le fichier d'entrée contenant la grammaire des expressions que l'on veut reconnaître est utilisé par Yacc tout comme Lex se sert du fichier décrivant les lexèmes : pour créer un nouveau fichier contenant du code source (en langage C dans notre cas). En particulier, la fonction *yyparse* générée permet de vérifier si la séquence d'unités lexicales à analyser respecte les règles grammaticales. L'expression à interpréter est entrée directement au clavier ou si le descripteur de fichier *yyin* a été initialisé, elle est récupérée dans le fichier auquel il se réfère. Si une règle est reconnue, l'action associée est exécutée.

Yacc présuppose la présence de la fonction *yylex* générée par Lex qui dissèque le fichier de données en lexèmes et les lui retourne sous forme d'identificateurs définis par la commande *%token* dans le fichier contenant la grammaire Yacc. A partir des différents lexèmes successifs et de la

grammaire des expressions valides, Yacc détermine si l'expression récupérée est autorisée pour l'état dans lequel il se trouve.

Pour obtenir un exécutable capable de traduire nos fichiers VRML, trois étapes seront nécessaires. Ces trois étapes sont schématisées dans la Figure 45. La première étape est la traduction des règles de Lex définies dans un fichier portant l'extension .l en un fichier en langage C contenant l'automate complet et la fonction *yylex*. La deuxième traduit le fichier des règles de Yacc en un fichier C gérant l'automate à pile et comprenant la fonction *yyparse*. Enfin, la troisième étape lie ces deux fichiers au reste du programme qui comprend bien sûr l'appel à la fonction *yyparse*.



Le programme suivant est un fragment, écrit pour Yacc, de notre analyseur syntaxique. En **italique gras** sont repérés les symboles non terminaux de l'analyseur et en **GRAS MAJUSCULE** sont représentés les symboles terminaux. La clause *%start* donne l'état initial de l'automate et la clause *%union* donne la structure permettant de stocker les différentes valeurs possibles que peut prendre un symbole. Au regard de la disparité des types de valeurs possibles pour un champ d'un nœud VRML (chaîne de caractères, liste de chaîne de caractères, liste d'entiers, liste de flottants, ...), nous avons choisi comme structure, un pointeur sur un type indéfini (*void **). Cela permet de donner n'importe quel type de valeur aux symboles et de les récupérer en connaissant le type de valeur attendu après un certain symbole. Dans le fragment de code ci-dessous, on sait qu'un symbole IDENTIFIER a une valeur de type chaîne de caractères (type renvoyé lors de l'analyse lexicale). Ou encore, on sait que la valeur du champ *size* du nœud *Box* est de type *SFVec3f* (triplet de flottants).

```

%{
// Inclusion des déclarations de routines utilisées dans les fragments
// de codes C
#include <stdio.h>
#include "stack.h"
#include "tree.h"
#include "values.h"

// Définition des variables externes
extern Node *tree;          // racine de l'arbre des expressions
[...]

// Définition des fonctions externes
  
```

```

extern void ErrorMessage(char *);
extern void WarningMessage(char *);
[...]

%}

// Déclarations YACC
%start vrmlscene // Etat de départ

%union { // Type de valeur manipulée
    void *value;
}

%type <value> IDENTIFIER // Type de valeur des symboles
[...]

%token IDENTIFIER DEF USE // Déclarations des symboles terminaux
[...]

vrmlscene: // Scène VRML = déclarations
{ // initialisation de la pile
    initStack(); }
declarations
{ // si tout s'est bien passé, on récupère la racine de l'arbre
    tree = popStack(); }
;

declarations: // déclarations =
// rien,
| declarations // ou déclarations + une déclaration
declaration
;

declaration: // Une déclaration =
nodeDeclaration // un noeud,
| protoDeclaration // ou prototype,
| routeDeclaration // ou un prototype externe.
;

nodeDeclaration: // Déclaration d'un noeud =
node // le noeud appelé tel quel,
| DEF IDENTIFIER // ou précédé du mot clé DEF,
{ // nouveau noeud de type DEF et de nom défini IDENTIFIER
// Positionnement de ce noeud en haut de la pile
Node *node = CreateNode(DEF, $2);
pushStack(node);
addDefName(node);
}
node
{ // Suppression du noeud DEF
popStack(); }
| USE IDENTIFIER // ou précédé du mot clé USE.
{ // nouveau noeud de type USE et de nom défini IDENTIFIER
// Positionnement de ce noeud en haut de la pile
pushStack(CreateNode(USE, $2));
useValue($2);
popStack(); }
;

```

L'ensemble Lex+Yacc nous sert à construire un arbre à partir d'une expression ou d'un fichier respectant la syntaxe VRML. C'est cet arbre, contenant tous les nœuds et les champs VRML, que le

programme principal parcourt ensuite afin de le traduire en des structures compatibles avec le moteur 3D utilisé par le programme.

Le fragment de code Yacc ci-dessus sera utilisé pour expliquer le principe de la construction de l'arbre. Comme nous l'avons vu plus haut, Yacc parcourt les expressions de gauche à droite. Non seulement les terminaux et les non terminaux sont lus dans cet ordre, mais aussi, les actions sont exécutées lorsqu'elles sont parcourues. Ainsi, dans la règle *vrmlscene*, la première action est exécutée avant de « descendre » dans la règle *declarations*.

Pour construire l'arbre nous manipulons principalement trois structures :

- la structure de nœud renferme toutes les informations utiles pour manipuler un nœud, comme son type (nous utilisons pour cela les entiers définis par Yacc à partir des tokens données), son nom s'il est défini (pour les nœuds de type DEF, USE, ...), un pointeur sur sa valeur, une liste de pointeurs sur ses nœuds fils... Pour notre programme, nous considérons comme nœud non pas uniquement les nœuds VRML mais également les champs des nœuds VRML. Dans la suite de ce rapport lorsque nous parlerons de nœud nous entendrons ce type de nœud de l'arbre. Pour parler des nœuds VRML, nous utiliserons le terme nœud VRML.
- la structure de pile garde une trace des nœuds précédemment parcourus que nous n'avons pas fini de parcourir parce qu'une dérivation à droite à eu lieu. Lorsqu'un nouveau nœud est lu, on le pousse sur la pile et lorsque nous l'avons entièrement parcourus (ainsi que sa descendance) on le retire de la pile. Cette façon d'opérer permet de savoir à quel nœud (celui qui est la tête de pile) il faut attacher le nœud que l'interpréteur commence à parcourir.
- les structures de valeur permettent, en fonction du type de valeur d'un nœud, de construire un pointeur sur la structure correspondante. C'est pourquoi dans Yacc nous avons défini un type de valeur « pointeur indéfini » pour que l'on puisse pointer sur toutes les différentes structures de valeurs correspondant à tous les types de valeurs possibles dans un fichier VRML. Les structures de valeurs sont construites par Lex à qui l'on fournit le type de valeur attendue pour qu'il sache dans quel état se mettre (pour lire le bon nombre d'arguments en fonction du type de valeur). Lorsque les paramètres de la valeur sont tous lus, le pointeur sur la structure construite est alors renvoyé à Yacc qui l'intègre dans le champ valeur de la structure du nœud pointé par la tête de pile.

Rappelons que c'est l'automate Yacc qui commande à l'automate Lex la lecture d'un nouveau lexème lorsqu'il en a besoin. C'est donc Yacc qui lit indirectement le fichier VRML et qui, par ailleurs, effectue une grande partie du travail. La Figure 46 montre comment l'arbre et les instructions OpenGL correspondants à l'exemple associé sont construits. Parce qu'il comporte beaucoup d'informations, quelques explications sont nécessaires.

Le schéma comporte cinq parties, chacune d'elles présentant un élément crucial de la construction :

- le fichier VRML duquel on souhaite construire l'arbre,
- les états dans le lequel se trouve l'automate Lex en fonction des lexèmes extraits du fichier VRML et les lexèmes qu'il renvoie à l'automate Yacc,
- la récupération des lexèmes et le parcours des règles en fonction de ceux-ci par Yacc, ainsi que les opérations sur la pile des nœuds,
- l'état de la pile des nœuds et les opérations effectuées pour la modifier et pour construire l'arbre,

- la hiérarchie des nœuds correspondant à l'arbre VRML et les appels aux fonctions OpenGL lors du parcours de chacun des nœuds.

La première opération consiste à construire un nœud racine fictif qui n'existe pas explicitement dans une structure VRML. Cela se fait dans la première action de la règle *vrmlscene* par l'appel à la fonction *InitStack*. Le fichier VRML est alors parcouru entièrement par dérivations successives en commençant par le non terminal *declarations* et lorsque ce parcours est terminé, et s'il n'y a pas eu d'erreurs fatales, on récupère le nœud racine dans la deuxième (et dernière) action de la règle *vrmlscene* par un appel à la fonction *popStack*.

Ensuite, tant que la fin du fichier VRML n'est pas atteinte et que Yacc se trouve dans un état qui requiert la lecture d'un lexème pour être résolu, il demande à Lex de lui renvoyer le type et la valeur du lexème suivant. Dans le schéma, nous avons matérialisé le type des lexèmes renvoyés plutôt que leur valeur parce que les transitions d'états dans Yacc se font en fonction des types. D'autre part, dans la partie concernant l'automate Yacc, les transitions entre états sont marquées à l'aide de flèches en pointillées parce que les transitions sont incomplètes : il existe d'autres états intermédiaires. Cependant, comme ils ne génèrent aucune instruction pour les étapes suivantes, nous les avons consciemment omis.

Les actions associées aux différents états de l'automate Yacc sont de trois types différents :

- la mise à jour de l'état de la pile des nœuds (pour garder constamment le dernier nœud lu en tête de pile),
- la détermination, en fonction du champ d'un nœud (état *nodeGut*), du type de valeur attendue par ce champ. Une seule méthode nous permet de récupérer cette information : écrire une routine qui renvoie le type de valeur attendue en fonction du type du nœud (que l'on connaît parce qu'il se trouve en tête de pile) et du nom du champ. Par exemple, si l'on demande à cette routine le type de valeur du champ *diffuseColor* du nœud *Material* elle renvoie *SFVec3F* qui correspond à un triplet de flottants. Dans le schéma, les deux flèches partant des états *nodeGut* de l'automate Yacc vers l'automate Lex matérialisent le changement d'état de Lex effectué par Yacc en fonction du type de valeur attendue. Ainsi, pour la lecture du lexème suivant, Lex se trouvera dans l'état correct,
- l'affectation des valeurs lues au nœud se trouvant en tête de pile (fonctions *AddValue*),

L'avant-dernier cadre concerne la pile de nœuds. Yacc empile un nouveau nœud (fonction *PushStack*) lorsqu'il commence l'analyse d'une nouvelle règle et que le nœud en question doit être sauvegardé pour apparaître dans l'arbre. Il dépile ce nœud (fonction *PopStack*) à la fin de la lecture de cette règle. La fonction *PushStack* ajoute non seulement un nouveau nœud en tête de pile mais référence également ce nouveau nœud comme fils de l'ancienne tête de pile. Par conséquent, lorsqu'un nœud sera dépiler, il conservera néanmoins toute la hiérarchie des nœuds dont il est l'ancêtre. Toutes les opérations "empiler" et "dépiler" sont schématisées dans la partie droite du cadre afin de comprendre l'évolution de la pile .

4.1.3 Traduction des nœuds VRML en appels de fonctions 3D

La dernière étape (dans le dernier cadre de la Figure 46) concerne le parcours de l'arbre obtenu et la génération des instructions d'une API 3D pour visualiser la scène. Les flèches allant des éléments de la pile aux éléments de l'arbre schématisent la création d'un nœud dans la structure d'arbre et donc des liens avec leur nœud père. L'arbre obtenu par empilages et dépilages successifs est parcouru ensuite en profondeur par les nœuds gauches d'abord de manière récursive. Lorsque qu'un nœud doit être pris en charge, une fonction de traitement des arguments de ce nœud est alors appelée.

Pour l'affichage de la scène par une API 3D, nous devons utiliser une structure pour sauvegarder les commandes à exécuter car le parcours de l'arbre des nœuds VRML pour l'affichage de chacune des trames est beaucoup trop coûteux. Nous avons choisi d'utiliser le mode retenu de chacune des API. Nous verrons cependant qu'une autre solution est envisageable et même préférable. Cette méthode permet, après un seul parcours de l'arbre nécessaire à la création de la structure, de libérer l'arbre ainsi que toutes les structures qui ont servies à la construire.

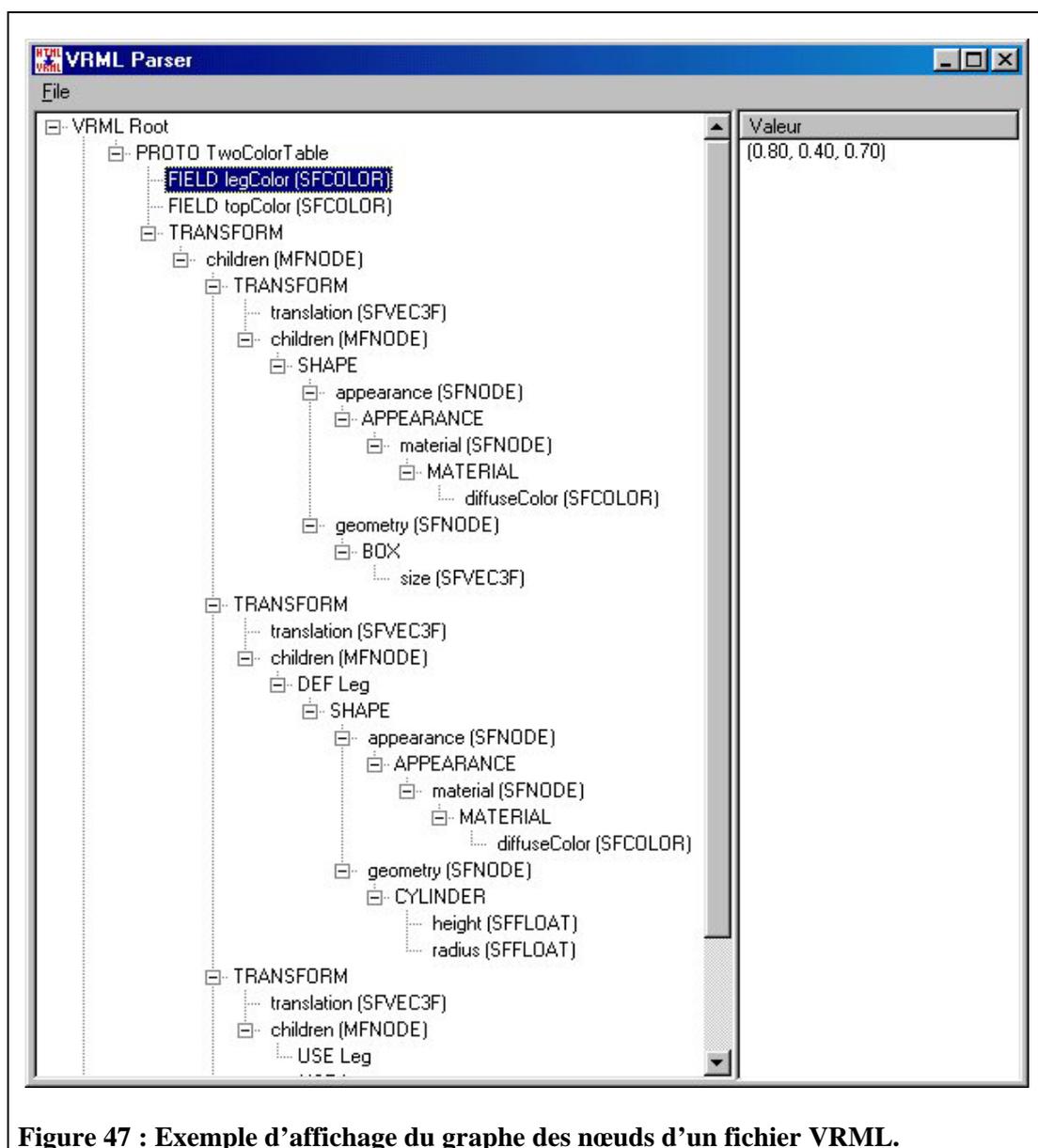


Figure 47 : Exemple d'affichage du graphe des nœuds d'un fichier VRML.

On peut se dire qu'il aurait été possible de générer directement cette structure depuis l'analyseur syntaxique sans passer par la création d'un arbre de nœuds. Cependant, deux raisons font

que cette façon de procéder est préférable et même obligatoire :

- Parce qu'aucun ordre dans les attributs des nœuds n'est à respecter. Par exemple, dans un nœud *Transform*, le champ *children* donnant les nœuds descendants peut être placé avant les transformations qu'ils doivent subir. En créant la structure de commandes à la volée, les transformations ne s'appliqueraient donc pas aux descendants. Or, le nœud *Transform* existe pour cela. Le passage par un arbre permet alors, lors de la création de la structure de commandes du mode retenu d'une API, de traiter les attributs dans un ordre qui convient.
- Parce que nous voulions présenter la hiérarchie des nœuds VRML dans une fenêtre (sous Windows), comme dans la Figure 47 montrant le résultat obtenu sur notre exemple de prototype. On y voit que l'application présente la hiérarchie complète des nœuds et l'on remarque que l'arbre du prototype externe y est intégré et donc qu'il sera géré lors de la traduction comme un prototype normal (non externe : *PROTO*). Les types de nœuds sont présentés et également leur nom pour les prototypes et les instances. Pour les champs des différents nœuds, sont affichés les types de valeurs (*SFVec3F*, *SFString*, ...). Dans la partie droite de la fenêtre est présentée la valeur de l'élément sélectionné dans l'arbre (ici la valeur par défaut de *legColor*).

Pour OpenGL, avant et après le parcours de l'arbre, on exécute respectivement les fonctions *glNewList* et *glEndList*. Ces routines permettent de créer une liste pouvant mémoriser toutes les commandes OpenGL comprises entre les deux appels. Ainsi, il suffira de faire appel à la fonction *glCallList* pour que les commandes OpenGL de la liste soient exécutées. Lors de la construction de la liste de commandes, lorsqu'un nœud parcouru peut être traduit en une opération OpenGL, une instruction est générée (c'est le cas, par exemple, du nœud *diffuseColor* qui génère l'instruction *glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, dc)* où *dc* est la valeur du nœud, attachée par la fonction *AddValue*). Les transformations spécifiées par le nœud de regroupement *Transform* sont prises en compte en exécutant les commandes *glTranslatef*, *glRotatef* ou *glScalef*. Afin que seulement les nœuds descendants du nœud *Transform* soient affectés par les transformations, on utilise les commandes *PushMatrix* et *PopMatrix* pour sauvegarder et restaurer la matrice de transformation respectivement avant les translation, rotation et homothétie et après que le nœud *Transform* soit entièrement parcouru.

Pour Direct3D, la hiérarchie des nœuds de regroupement est gardée grâce aux *frames* que nous avons vues lors de la présentation du mode retenu de Direct3D. Les lumières et les formes géométriques sont alors attachées à la *frame* active. La hiérarchie du graphe de scène pour Direct3D correspond donc exactement à l'arbre des nœuds alors que pour OpenGL les commandes sont toutes au même niveau. La similitude entre le modèle de données sous VRML et celui sous Direct3D ne s'arrête pas là, puisque, avec Direct3D, les transformations sont attachées à une *frame* et opérées par rapport à la *frame* parent comme sous VRML avec le nœud *Transform*. Grâce à ces ressemblances de conception, la traduction de l'arbre VRML en une structure du mode retenu de Direct3D est donc beaucoup moins compliquée qu'une traduction en commandes OpenGL.

Pour la traduction des appels de prototypes, le mécanisme est relativement simple. Lorsqu'un nœud inconnu est rencontré dans l'arbre, on effectue une recherche d'un prototype portant ce nom depuis le début de l'arbre des nœuds. Si le prototype est trouvé, il suffit alors de traduire entièrement son sous-arbre avec les valeurs correctes de paramètres (ceux transmis lors de l'appel ou le cas échéant ceux par défaut). Une fois que ce parcours est terminé, le parcours de l'arbre est repris après l'appel du nœud prototype.

Pour les nœuds *USE*, le mécanisme est le même sans la complication introduite par la gestion des paramètres. Lorsque l'on rencontre un nœud *USE*, on recherche sa définition (nœud *DEF*) dans tous les nœuds déjà parcourus (puisque la clause *USE* ne peut apparaître qu'après la clause *DEF*). Lorsqu'on l'a trouvée, il suffit, comme pour les nœuds *PROTO* de parcourir son sous-arbre intégralement et de le traduire en fonctions de l'API avant de reprendre le parcours normal de l'arbre après le nœud *USE*.

Une table aurait pu être utilisée pour trouver plus rapidement la déclarations des prototypes et des définitions d'instances. Elle associerait chacun des symboles à un pointeur sur le nœud de l'arbre où commence sa spécification. Cette table agirait donc comme une table des symboles.

4.2 Les différentes approches pour la lecture des fichiers VRML

Une scène VRML peut se composer de plusieurs fichiers ou URL. En effet, outre les textures et les sons, des données peuvent être externes au fichier principal si celui-ci comporte des prototypes externes ou des inclusions de fichiers par des nœuds *Inline*. L'interpréteur VRML doit donc être capable d'inclure ces données. Deux solutions sont envisageables pour gérer les inclusions lors du parcours d'un fichier VRML. La première méthode, celle mise en œuvre pour le visualisateur présenté, utilise une approche récursive alors que la deuxième est la méthode classique pour les compilateurs et traducteurs utilisant une table des symboles.

4.2.1 L'approche récursive

Le principe d'un compilateur récursif est simple à comprendre. Lors du parcours d'un fichier, si une inclusion d'un autre fichier est nécessaire, il suffit de sauvegarder dans une pile l'état du compilateur au moment de l'inclusion et de commencer le parcours du fichier inclus. Lorsque le parcours du fichier inclus est terminé, il faut réassigner au compilateur les valeurs sauvegardées avant l'inclusion. Ainsi, il continuera le parcours du premier fichier à l'endroit auquel il l'avait cessé.

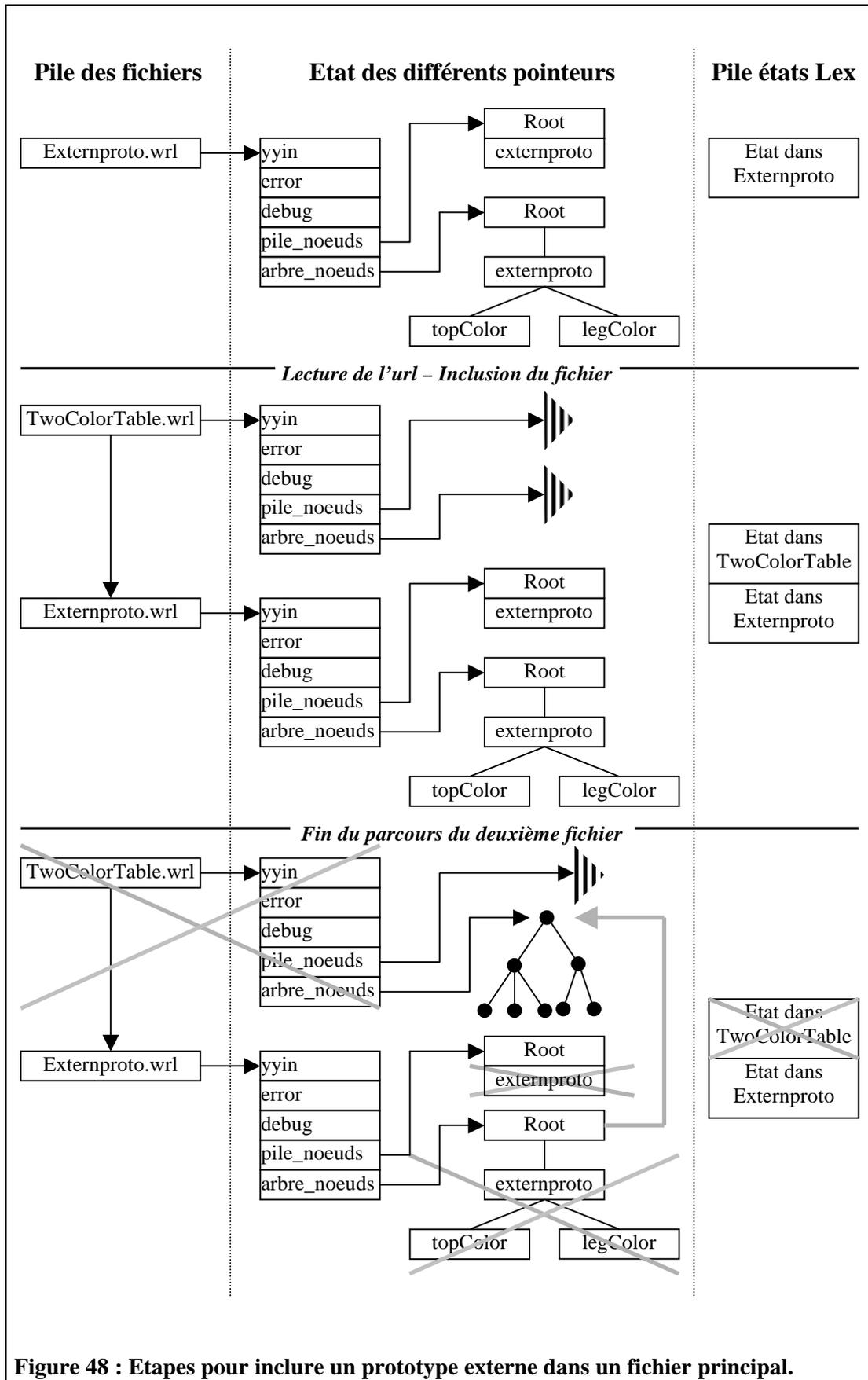
Avec les outils Lex et Yacc, nous devons sauvegarder l'état de deux automates. Pour l'automate Yacc, il suffit de lui dire que l'on souhaite un automate réentrant par l'ajout de l'option *%pure_parser* dans la partie déclaration du fichier Yacc. De cette façon, le code généré ne comportera pas de variables globales pour la gestion de l'état de l'automate. Toutes ces variables étant locales (dans la fonction *yyparse*), nous pouvons alors effectuer des appels récursifs à cette fonction sans perdre les valeurs de ces variables puisqu'elles seront sauvegardées comme tout le contexte d'exécution de la procédure avant appel d'une autre.

Pour l'automate Lex, l'opération est un peu plus compliquée. En effet, Lex génère la fonction *yylex* utilisée par *yyparse* pour récupérer le lexème suivant dans le fichier. Or, *yylex* utilise un descripteur de fichier *yyin* pour savoir dans quel fichier et à quel emplacement dans ce fichier il doit rechercher les lexèmes. Il faut donc que nous créions une structure de pile pour sauvegarder l'état d'un fichier. Lorsque le compilateur ouvre un fichier (que ce soit le fichier principal ou un à inclure), il sauvegarde auparavant les informations relatives à ce fichier et ce sont ces informations qui sont utilisées par le compilateur. Ainsi cette pile nous sert, d'une part, à fournir à l'automate d'analyse lexicale les informations utiles pour la lecture (celles se trouvant en tête de pile), et d'autre part, à sauvegarder les différents fichiers ouverts et surtout l'ordre d'ouverture. Les informations sauvegardées dans la pile avant ouverture d'un nouveau fichier sont les suivantes :

- le descripteur *yyin* de l'ancien fichier (et donc la position à laquelle s'est arrêtée sa lecture),
- le descripteur du fichiers d'erreurs (portant le nom du fichier VRML auquel nous avons remplacé l'extension *.wrl* par *.err*),
- le descripteur du fichier de débogage (même principe que pour celui des erreurs mais avec l'extention *.dat*),
- la pile des nœuds parcourus,
- l'arbre construit.

Lorsque le parcours d'un fichier est terminé, il suffit de recopier l'arbre de ses nœuds construit par le parseur dans l'arbre du fichier appelant (s'il existe en deuxième position de la pile). La recopie de l'arbre correspond uniquement à une affectation de pointeur : le nœud en tête de pile du fichier appelant est remplacé par la racine de l'arbre du fichier inclus. Lorsqu'il sera dépilé, il sera alors automatiquement ajouté au nœud père du nœud dans le fichier appelant. La Figure 48 illustre ces dires et aide surtout à les comprendre. Nous utilisons comme exemple le prototype externe de la table (de la page 92). Nous pouvons voir dans la Figure 47 que l'arbre du prototype a bien été intégré au fichier principal à l'emplacement où l'utilisation du prototype avait été déclarée. Dans cette même copie d'écran, nous n'avons pas pu montrer la suite de l'arbre des nœuds VRML à cause d'un problème évident de place, mais nous garantissons que figurent les deux appels au nœud prototype *TwoColorTable*.

Une deuxième pile doit également être utilisée pour sauvegarder l'état de l'automate généré par *Lex*. En effet, à l'inclusion d'un nouveau fichier, l'automate doit être mis à l'état initial. Sans sauvegarde d'état, après le parcours du fichier inclus, nous n'aurions plus l'état dans lequel s'est arrêté l'automate pour le fichier précédent (ce qui empêcherait de terminer son parcours correctement). Nous utilisons pour cela une pile prévue à cet effet, fournie dans le code généré par *Lex*. Les opérations d'empilage et de dépilage pour cette pile s'effectuent cependant au même moment que pour notre propre pile de fichiers. Nous avons donc écrit une fonction *ParseFile* qui effectue toutes ces opérations. Elle initialise puis empile les informations sur le nouveau fichier donné en paramètre puis lance le parseur qui utilisera cette nouvelle tête de pile. Après que le nouveau fichier a été parcouru, la procédure dépile ses informations sauvegardées afin que, lorsqu'elle se termine, l'interpréteur puisse continuer avec l'ancienne tête de pile.



4.2.2 Solution avec une table des fichiers

La manière plus commune pour compiler des fichiers inclus est de tenir à jour une table des fichiers à ouvrir. Cette table se comporte comme une table des symboles. Durant le parcours d'un fichier, si une inclusion ou des inclusions de fichier sont rencontrées, il faut sauvegarder les noms de ces fichiers dans la table. Une fois le parcours d'un fichier terminé, il suffit d'ouvrir un par un, dans l'ordre, les fichiers présents dans la table.

Cette méthode a deux avantages :

- un seul descripteur de fichier est utilisé puisqu'un seul fichier est ouvert à la fois et donc pas de risque de traiter un niveau d'imbrication dépassant le nombre de descripteurs de fichier autorisés,
- la structure de table permet de vérifier l'existence de cycles dans les inclusions de fichiers. Un exemple de cycle en VRML serait un fichier utilisant un prototype externe qui lui-même inclut le premier fichier dans un nœud *Inline*.

4.3 Contrôles sur le rendu et aides à la navigation

Bien que délicates et d'extrême importance, les étapes de lecture du (des) fichier(s) VRML composant une scène et leur traduction en fonctions 3D d'une API ne sont qu'une partie du travail. La seconde concerne la programmation d'outils pour permettre à un utilisateur d'évoluer dans la scène et d'en contrôler le rendu. Comme la plupart des clients VRML que l'on peut trouver sur Internet⁴¹, notre navigateur intègre en effet des options pour déplacer et orienter le point de vue sur la scène ainsi que pour changer le mode de tracé des objets.

4.3.1 Options de visualisation

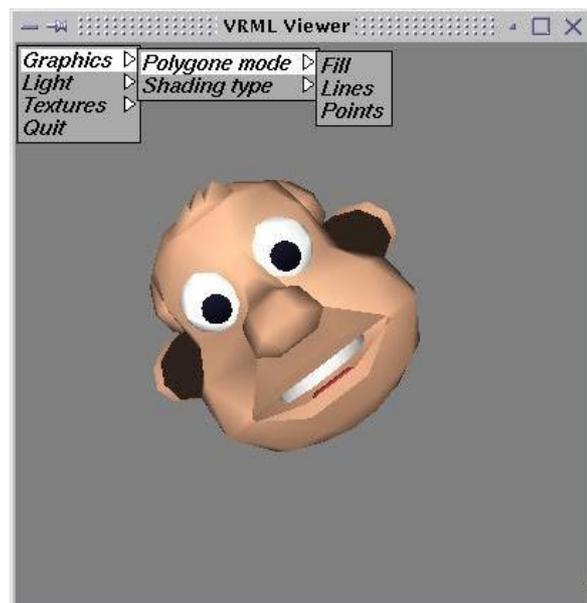
Les options d'affichage sont accessibles par un menu contextuel apparaissant lorsque l'on clique sur le bouton droit de la souris dans la fenêtre de visualisation. Elles permettent de changer différents paramètres influant sur le rendu de la scène.

- Le type de rendu offre le choix entre les modes plein, filaire ou « points ».

Le mode plein est le type de rendu standard. Les objets sont dessinés en tenant compte de leur profondeur (par le *ZBuffer*). Ils sont remplis soit par une couleur (par un coloriage à plat ou Gouraud) soit par une texture.

Le mode « points », ne montrant que les sommets des polygones, est très rarement utilisé.

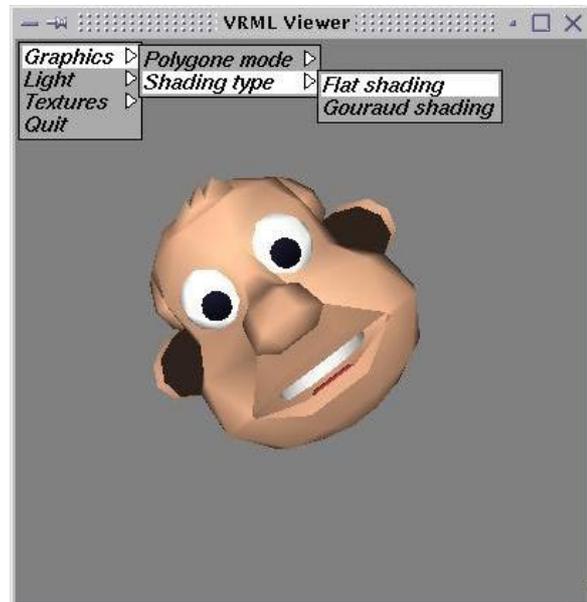
Le mode filaire par contre se révèle très pratique lorsque l'on souhaite visualiser une scène sans que les objets du premier plan cachent ceux qui se trouvent derrière eux. Il



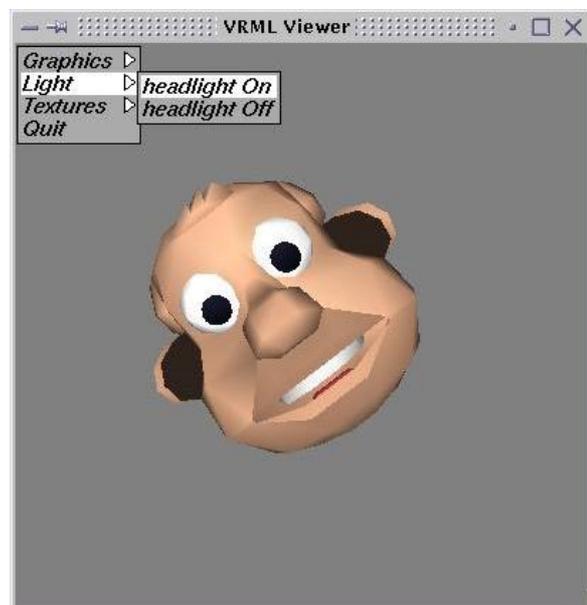
⁴¹ Voir la liste page 76.

permet également d'appréhender la structuration d'une scène, de voir comment se composent les différents éléments et très souvent suffit pour déterminer si la scène VRML décrite répond ou non aux attentes de son concepteur.

- Le type de coloriage est également modifiable pour obtenir une qualité de rendu plus ou moins grande. Les deux modes de coloriage sont classiques. Le moins coûteux (offrant le rendu le plus rapide) est le mode *flat* dans lequel les polygones sont remplis avec une couleur unie. Un coloriage plus réaliste est celui inventé par Gouraud. Il nécessite des opérations supplémentaires pour déterminer la couleur de chaque pixel visible du polygone par interpolations des couleurs des sommets. Pour chaque coloriage, la couleur d'un point peut également être dépendante de la lumière réfléchi. Pour voir les différences entre les modes de coloriage, on se reportera à la Figure 6 page 14.



- L'ajout d'une lumière de type *HeadLight* (lumière « casque de spéléologue ») permet d'éclairer la scène depuis le point de vue et donc de mieux discerner les objets face à l'utilisateur. Lorsque la scène est en contre-jour, cette lumière permet d'éclairer (et donc de voir) les objets du premier plan. Dans la phase de conception d'une scène VRML, cette lumière est également importante pour valider les normales des polygones. En effet, pour chacune des faces, la couleur la plus claire doit être obtenue lorsque les vecteurs normaux appliqués aux sommets sont opposés à l'orientation de la caméra (c'est-à-dire lorsqu'ils sont face à l'utilisateur).



- Un sélecteur permet également de supprimer les textures. Il est utile lorsque la scène est ralentie à cause du grand nombre de textures. Dans ce cas, les faces sont tracées avec la couleur de l'objet si elle est définie ou avec la couleur blanche par défaut.

4.3.2 Manipulation de la caméra

Parce que c'est le type de périphérique de pointage le plus répandu, la souris s'est imposée *de facto* pour interagir avec une scène 3D. Bien que celle-ci ne compte que deux degrés de liberté (elle permet uniquement de faire deux translations dans le plan horizontal), elle compte néanmoins des avantages. Les utilisateurs la manie en effet avec maîtrise et il est ainsi possible de profiter de cette connaissance. Toutes les opérations de sélection et d'interaction avec les objets 3D pourront (et devrons ?) en tirer partie. Des études ont, de ce fait, été menées sur la manipulation directe d'objets 3D à partir d'une souris 2D [CHE88][EMM90]. A l'heure actuelle, pour les applications 3D grand public, il existe deux modes d'interactions différents. Le premier demande l'utilisation conjointe de la souris et du clavier. Certaines touches effectuent les déplacements sur les différents axes ou des actions. Les rotations sont calculées à partir des mouvements de la souris avec un bouton appuyé. Les autres boutons de la souris restent alors disponibles pour l'implémentation de la sélection ou d'actions. La deuxième méthode utilise la souris et des *widgets*. Dans leur version la plus évoluée, les *widgets* peuvent venir en surimpression ou à côté de l'objet sélectionné et proposent des actions sur les objets. Plus classiquement, dans les navigateurs VRML, les *widgets* sont des images « clicables » représentant le type de déplacement que l'on peut effectuer avec la caméra. Par un clic de souris sur un de ces composants, un mode de navigation est sélectionné. A la suite de ce clic, les déplacements de la souris dans la scène sont interprétés pour effectuer la navigation sélectionnée.

Les options modifiant le rendu de la scène présentées précédemment sont classiques et apparaissent dans les autres navigateurs. Pour les opérations de navigation nous avons opté pour des méthodes plus originales tirées de travaux récents. Nous les avons préférées au paradigme de navigation tiré des *widgets* de visualisation utilisé dans les autres clients VRML. La navigation s'effectue à la souris, bien que les translations soient possibles à partir de certaines touches du clavier. La souris étant un périphérique offrant deux degrés de liberté, nous avons implémenté la technique présentée par MM. Zeleznik et Forsberg [ZEL99] permettant de contrôler une caméra virtuelle 3D sur les 6 degrés de liberté à l'aide d'une souris 2D et d'un seul de ses boutons sans avoir besoin d'un clavier ou de *widgets* 2D. *UniCam*, puisque tel est le nom de cette technique, permet ainsi de garder les autres boutons de la souris libres pour les opérations spécifiques d'une application, contrairement à la plupart des autres systèmes de navigation dans un espace 3D. Prises individuellement, les huit techniques de navigation implémentées ne sont que de modestes adaptations de techniques bien connues. Cependant, lorsque l'on considère ces techniques prises collectivement, *UniCam* est une sérieuse avancée pour la manipulation de caméra avec un pointeur 2D grâce à la sélection et l'adaptation de techniques qui se complètent et surtout grâce à la possibilité de changer de technique de contrôle de caméra par des transitions gestuelles simples et fluides par l'intermédiaire de la souris.

UniCam intègre huit techniques de manipulation de caméra regroupées dans 3 types de navigations différentes. Les différentes techniques sont choisies uniquement par un clic sur un bouton de la souris et par le début du mouvement initié par l'utilisateur. Le laps de temps durant lequel on calcule le mouvement initial de la souris est arbitraire. Cependant, au-delà de 1/10ème de seconde ce temps semble déranger les utilisateurs. D'autre part, la souris doit parcourir assez de chemin pour que le système puisse calculer le type de mouvement de caméra (pour une définition de 1280*1024, ce sera 15 pixels). Les différents modes de mouvements en fonction du delta de souris parcouru ont été choisis empiriquement, d'après les réactions des utilisateurs.

◆ Contrôles de la caméra

La fenêtre de visualisation est séparée en deux régions, l'une rectangulaire au centre et l'autre étant une bordure externe de taille environ 1/10ème de la taille de la fenêtre servant au rendu. Ces régions ne sont pas visibles mais chacune a son propre interpréteur de mouvements et d'actions de souris.

• Translations de la caméra

Un clic de souris dans la région centrale instaure pour les déplacements en X et Y

de la souris l'ensemble de mouvements {translation horizontale, zoom vertical} si le mouvement de la souris initié est vertical, soit l'ensemble {translation horizontale, translation verticale} si la souris a un mouvement initial horizontal.

- **Mouvements en orbite de la caméra**

UniCam permet également de faire subir à la caméra un déplacement en orbite autour du centre de la scène ou d'un point quelconque sélectionné. L'implémentation est faite de telle manière qu'un déplacement de la souris de la gauche à la droite de l'écran fait tourner la caméra de 360 degrés autour de l'axe Z et qu'un déplacement du haut au bas de l'écran opère une rotation de 180 degrés autour de l'axe X. De cette façon, la caméra se déplace sur une sphère en regardant toujours le même point. Deux mouvements en orbite ont été définis :

- **Autour du centre de la vue courante** : ce mouvement est initié en cliquant dans la région bordure de la fenêtre de visualisation. Pour cette technique de mouvement en orbite, le point autour duquel la caméra tournera sera pris sur l'axe de la vue courante à une profondeur arbitraire. Cependant, les utilisateurs souhaitent tourner autour de l'objet se trouvant au milieu de la scène, la position du centre de rotation est mise à jour lors de toute modification de la position et de l'orientation de la caméra. Ainsi, si l'utilisateur opère une translation pour amener un objet au centre de la scène, il pourra ensuite tourner la caméra autour.
- **Autour d'un point quelconque** : ce mouvement de caméra est initié en cliquant et en relâchant immédiatement le bouton de la souris lorsque le pointeur se trouve sur un objet dans la scène. Un point matérialisé par une sphère sera le centre de la rotation. Si l'utilisateur clique ailleurs que sur ce point alors la rotation se fait.

- ◆ **Mouvements automatiques**

Une autre façon d'opérer des translations et rotations est de placer un point de référence. Ce point de référence donné, la caméra se déplace de sa position courante jusqu'à une nouvelle position en fonction du point de référence. Deux variantes existent pour cette technique, chacune dépendante du point de référence que l'on place comme pour le mouvement en orbite précédent.

La première variante consiste à cliquer puis à relâcher le bouton de la souris lorsque le pointeur se trouve sur le point de référence. Cela initie une rotation de la caméra pour avoir une vue oblique de la surface sur laquelle se trouve le point de référence.

La seconde variante est pour zoomer sur une région. L'utilisateur clique sur le point de référence sans relâcher le bouton de la souris et déplace la souris vers la droite ou le bas de l'écran. Il en résulte que le point de référence devient une sphère transparente qui s'agrandit en fonction de la distance qu'a parcouru la souris. Cette sphère définit la région de zoom. Lorsque la région convient à l'utilisateur, il relâche la souris et la caméra s'anime pour obtenir une vue englobant au plus près la région sélectionnée.

- ◆ **Sauvegarde et restauration de la position de la caméra**

Les deux dernières fonctions dans l'implémentation des mouvements de caméra de *UniCam* sont la sauvegarde du point de vue et la restauration de celui-ci. Pour la sauvegarde, l'utilisateur clique et relâche le bouton de la souris pour créer un point de référence. Puis en cliquant sur le point créé et en gardant le bouton de la souris

appuyé, s'il se déplace ainsi vers le haut à droite de l'écran alors l'utilisateur entrera automatiquement dans le mode sauvegarde. Avec *UniCam*, une icône 2D représentant le point de vue sauvegardé s'affichera sous le pointeur de la souris et la suivra tant que l'utilisateur ne relâche pas le bouton de la souris. L'utilisateur peut alors interagir de quatre façons différentes avec cette icône :

- en agissant sur les bords de l'icône, il peut la déplacer,
- en agissant sur les coins de l'icône, il peut l'agrandir ou la rétrécir,
- en cliquant à l'intérieur de l'icône, l'utilisateur rappelle la vue sauvegardée,
- en cliquant sur l'icône sans relâcher le bouton de la souris et en la déplaçant vers le haut à droite, l'icône est effacée et donc la vue sauvegardée aussi.

La Figure 52 illustre de manière simplifiée comment sont obtenus les différents déplacements à partir des actions de l'utilisateur.

Nous aurions pu mettre ici le code ou le pseudo-code d'une implémentation possible d'*UniCam* en fonction des événements de souris reçus. Mais il serait trop complexe et trop long car il existe de nombreux événements à prendre en compte pour initier une des 8 techniques de navigation :

- trajet de la souris,
- direction du mouvement de la souris,
- position de la souris,
- éléments sur lesquels sont opérés les mouvements de souris (l'interprétation diffère si les mouvements ont lieu dans la fenêtre principale ou sur une vue sauvegardée),
- coordonnées de la souris lorsque son bouton a été appuyé et relâché,
- l'état dans lequel se trouve *UniCam* (les déplacements sont interprétés différemment d'une technique à l'autre).

Sachant que s'ajoutent à ces cas précédents, ceux pour la manipulation des WIMs et des zooms que nous présentons dans la partie suivante, nous obtenons une gestion des événements assez complexe.

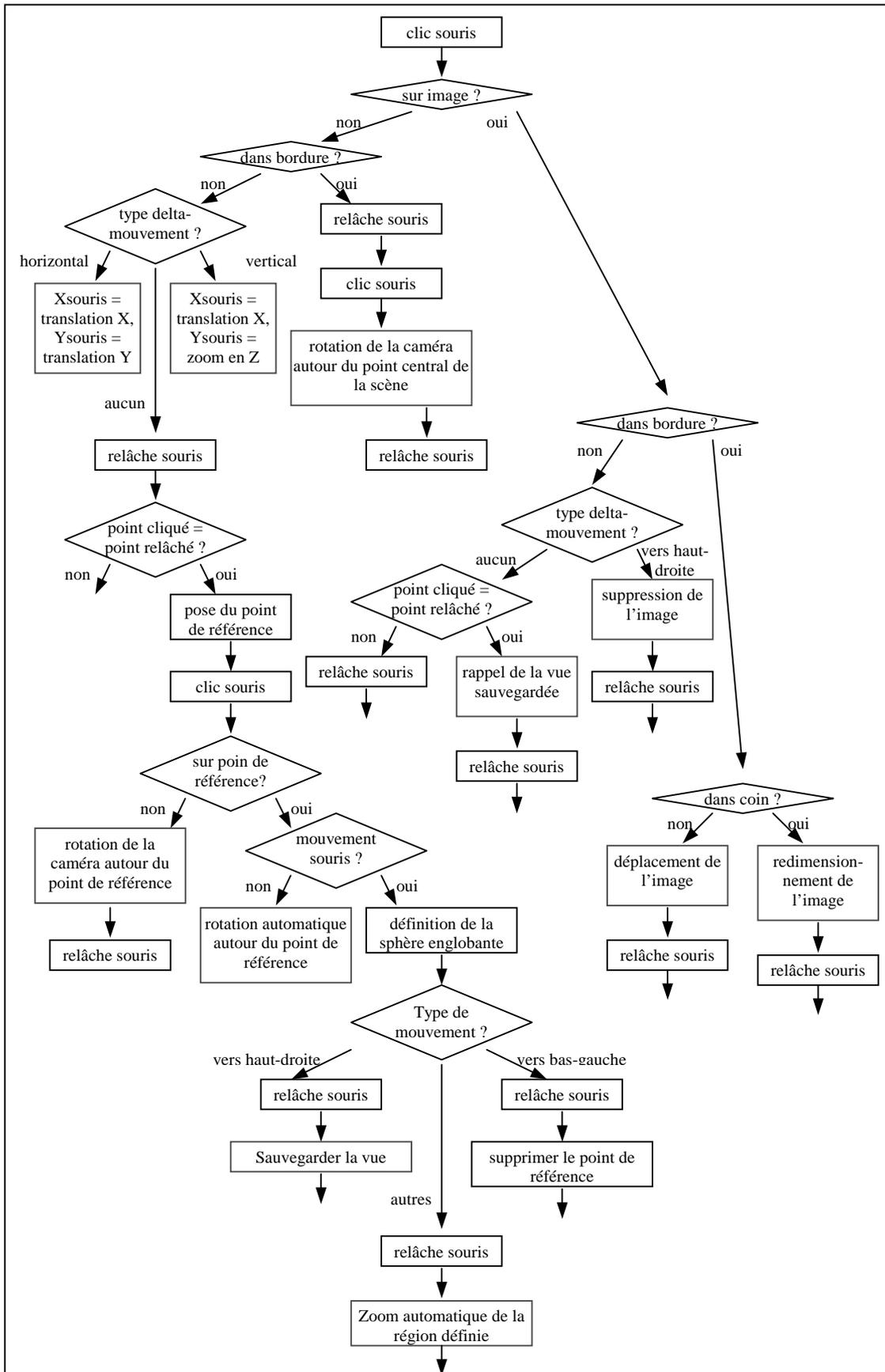


Figure 52 : Obtention des différents mouvements avec *UniCam*.

L'avenir appartient cependant aux nouveaux dispositifs de pointage ou capteurs dont les mouvements peuvent être plus naturellement associés à la navigation dans un espace 3D. Pour cette raison, nous avons également implémenté un contrôle de caméra à l'aide d'une SpaceMouse. Les nouvelles méthodes qui émergent ont été classées en deux catégories en fonction du type de gestes les contrôlant [BAE95] et pour chacune d'elles nous donnons quelques exemples :

- Non haptique⁴² (sans sensation de contact physique avec les objets virtuels) :
 - Caméra : utilisation d'une caméra vidéo dont les images capturées sont segmentées pour extraire les informations importantes. Cette méthode demande une grande puissance de calcul à cause des traitements en temps réel des images. Elle est dépendante des paramètres de l'environnement comme les variations de lumières.
 - Commandes vocales.
 - Sondage de champ électrique [SMI98] : mesure de la variation de charge accumulée entre deux groupes d'électrodes dépendante de la présence d'un objet entre les deux.
 - Champ électromagnétique [FUC99] : utilisation de deux groupes de trois bobines. Pour l'émetteur, trois bobines propagent un champ électromagnétique dans chacune des trois directions orthogonales (x, y, z). Les 3 bobines du récepteur, également orthogonales, recueillent ces flux en fonction de leur position et transmettent les coordonnées du récepteur. Cette méthode bien que très fiable et ayant une bonne fréquence de mesures reste coûteuse et souffre des interférences avec tout objet métallique.
 - Ultrason [BIB97], [FLE97] et [FUC99] : même principe que précédemment mais en utilisant des ultrasons. Les principaux problèmes de cette méthode est le délai de propagation des ondes sonores et les interférences à cause des réflexions.
 - Infrarouge [BIB97] : un émetteur envoie des rayons infrarouges à un récepteur qui mesure l'angle d'arrivée et la variation de l'intensité des rayons. Cette technique est inefficace lorsqu'un corps se trouve entre l'émetteur et le récepteur et lorsque l'angle des rayons n'excède pas 40°.
 - Capteurs à effets Hall : calcul de la tension aux bornes d'un semi-conducteur récepteur lorsque celui-ci est soumis à un champ électrique et magnétique dû à l'alimentation du capteur.
 - Inclinomètres : indication de la position angulaire par rapport à la verticale. Lorsque le capteur est en mouvement, cette technique indique l'accélération.
 - Compas : exploite l'effet Hall à partir du champ magnétique terrestre.
 - Gyroscopes et gyromètres : mesurent la vitesse angulaire d'un système en rotation autour d'un axe en utilisant le phénomène de la force de Coriolis.
 - Capteurs de pression mécanique : dispositifs dont la résistance électrique diminue lorsqu'on leur applique une pression.
 - Capteurs de flexion à encre conductrice [BUR93] : la flexion exercée augmente la distance entre les particules de carbone des deux couches d'encres conductrices, augmentant ainsi la résistivité du capteur.

⁴² Du grec *haptain* signifiant toucher.

- Mesure de signaux biologiques [PIC97] : utilisation de la tension musculaire, du rythme cardiaque, de la position des yeux, des ondes cérébrales, de la pression sanguine, ... pour contrôler un système.
- Haptique (les interfaces avec retour tactile et retour de force ou d'effort) : joystick, souris, stylet, gant, ...

Les principaux périphériques d'entrée 3D que l'on trouve dans le commerce se présentent sous la forme de *trackball* 3D ou de capteurs que l'on fixe sur la main ou la tête. Il gère tous au moins 3 degrés de liberté (pour les trois translations) et certains d'eux mesurent les rotations (donc 3 degrés de plus). Les capteurs les plus perfectionnés peuvent en compter plus. Le gant par exemple gère la position de la main et la flexion de chacun des doigts. Ces périphériques 3D ont été classés par C. Dumas [DUM99] et présentés dans [SIG2000].

Les périphériques isométriques comme les souris 3D ou les *trackballs* 2D possèdent une résistance infinie (c'est-à-dire qu'ils sont immobiles). Ils mesurent les mouvements en fonction des forces appliquées. Leur défaut principal est la décorrélation entre le mouvement de la main (quasi nul) et celui du pointeur. Le manque de retour de force gêne également pour les tâches de sélection par exemple.

Les périphériques isotoniques possèdent, eux, une résistance nulle (ils se déplacent avec la main). En 2D, c'est le cas des souris et en 3D celui des gants ou des systèmes à ultrasons. Leur principal défaut vient de la fatigue de la main après une utilisation prolongée.

Les périphériques élastiques sont à mi-chemin entre les deux types précédents. Ils retournent automatiquement à une position stable dès lors que l'on cesse d'y appliquer une force. Ils sont réputés pour leur utilisation plus intuitive grâce aux retours de force.

Le débat n'est toujours pas clos sur le type de périphériques 3D d'entrée le plus adapté à la navigation ou à la sélection d'objets dans l'espace. Celui qui fournit les meilleures performances pour chacun des cas n'a toujours pas été clairement déterminé [ZHA94]. Néanmoins, il s'avère que les périphériques isométriques sont préférés pour la navigation et les périphériques isotoniques pour la sélection. Les expérimentations mises en œuvre par C. Dumas, P. Plénacoste et C. Chaillou [IHM99] prouvent un léger avantage pour un périphérique isotonique lors d'une tâche de sélection. Cette étude a comparé le temps et le nombre de clics émis pour sélectionner un objet dans l'espace en fonction de différents paramètres comme le niveau d'expertise de l'utilisateur, le type du périphérique (isotonique ou isométrique) et de la présence d'ombres sous les objets. La combinaison optimale est obtenue avec les experts utilisant le périphérique isotonique pour sélectionner des objets ombrés. Cette étude permet donc également de valider l'utilisation des ombres pour fournir une information de profondeur. Quoi qu'il en soit, les utilisateurs novices sont désorientés pour sélectionner un objet, principalement à cause de la profondeur de l'objet.

Ces pointeurs isotoniques utilisent, pour les interactions dans un environnement virtuel, les déplacements d'une partie du corps humain (le plus souvent la main) dans l'espace physique. De la même manière, une souris 2D utilise un plan physique pour ses déplacements. Or, avec une souris classique, il est toujours possible de la soulever et de la ramener à un endroit du plan qui lui est alloué pour que l'interaction continue. Avec un périphérique 3D isotonique, une fois une limite du volume physique atteinte, il est impossible de se repositionner pour continuer un déplacement dans le même sens. Ce déplacement serait interprété comme une interaction puisque le pointeur serait toujours dans le volume d'interaction.

Les pointeurs 3D isométriques ou élastiques n'ont pas ce problème puisque leur résistance n'est pas nulle. Cependant, ce sont des capteurs détachés du corps donc moins intuitifs. On ne peut en effet pas imaginer un capteur corporel revenant à une position centrale automatiquement ! Or ce sont ceux offrant les interactions les plus riches et les plus réalistes. Le gant par exemple permet d'attraper

des objets virtuels et mieux encore d'en avoir la sensation.

4.3.3 Aides à la navigation

Comme nous l'avons vu, dans *UniCam* un certain mouvement permet de sauvegarder la vue courante sous forme d'une image et un autre mouvement sur cette image permet de la rappeler. Nous avons préféré aux images statiques utilisées dans *UniCam*, des WIMs (*Windows In Miniature*) présentées dans [STO95]. L'utilisateur interagit dans les WIMs avec les mêmes mouvements de souris que dans la vue principale. Elles offrent ainsi de multiples vues sur la scène ; ce qui apparaît très pratique lorsque la scène est conséquente. Dans la Figure 53, deux WIMs ont été ouvertes, l'une en haut à droite (dont le cadre est apparent car la souris se trouvait à l'intérieur au moment de la capture) et l'autre en bas à droite. Chacune d'elles montre une vue différente de la vue principale.

Comme on peut le voir dans la figure ci-dessous, les WIMs ont des tailles différentes. Elles peuvent être redimensionnées, déplacées et supprimées. Ces interactions s'opèrent toujours à l'aide de la souris. Elles se rajoutent donc à celles permettant de manipuler la caméra avec les concepts d'*UniCam*. La différence entre ces deux types vient des calculs nécessaires. Pour les mouvements de caméra, les déplacements 2D de la souris sont transformés en translations ou en rotations 3D du point de vue. Pour la manipulation des WIMs, tous les calculs restent en 2D puisqu'elles ne sont pas inclinables. La gestion des différents états de la souris est donc un point critique et, particulièrement, la détermination des positions où ont lieu les clics.

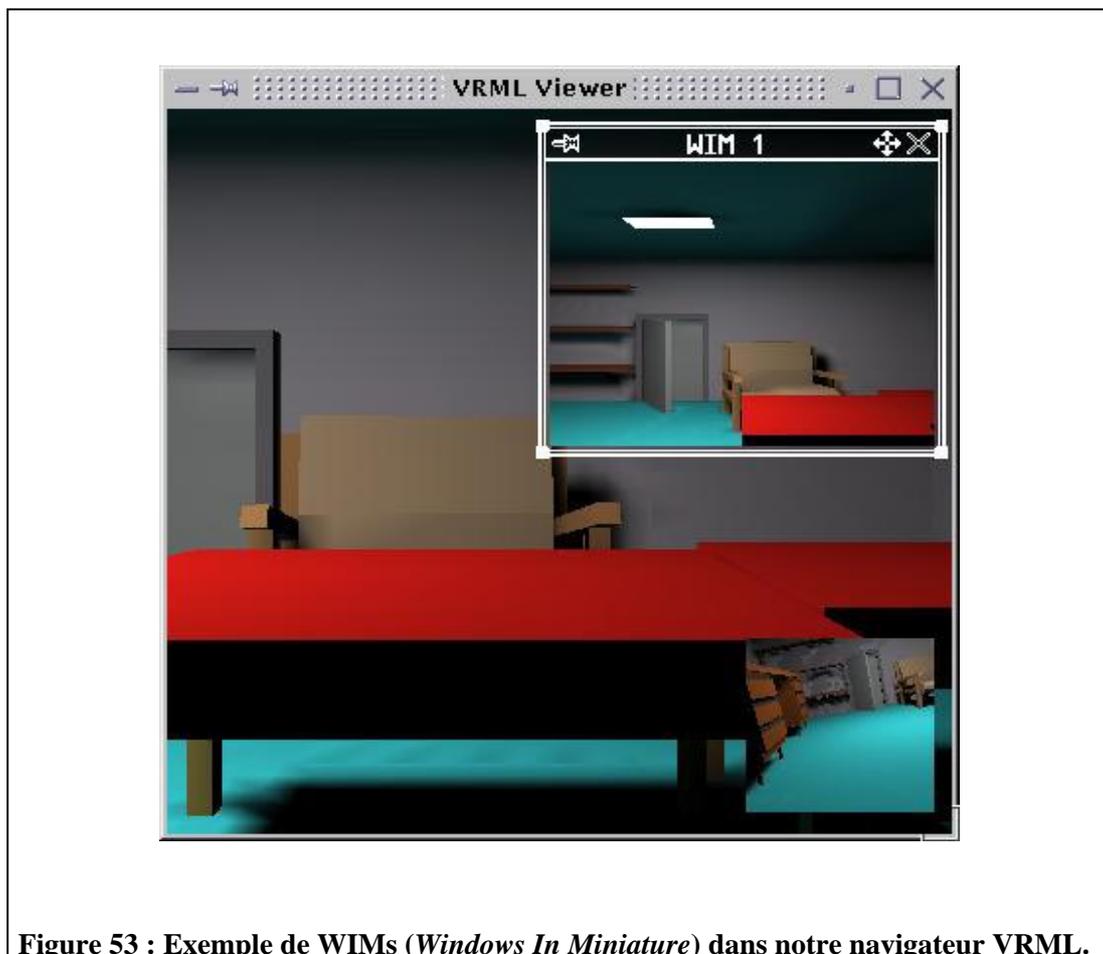


Figure 53 : Exemple de WIMs (*Windows In Miniature*) dans notre navigateur VRML.

La scène est recalculée pour chacune des WIMs à l'aide de ses propres paramètres de caméra. Les recouvrements sont gérés grâce à une pile de type FIFO (*First In First Out*) dans laquelle les

WIMs sont sauvegardées. Pour intégrer la rotation des WIMs, il aurait fallu générer une texture et la plaquer sur une face rectangulaire. Les interactions s'en seraient trouvées beaucoup plus difficiles à gérer. Le temps de rendu total (scène principale + WIMs) n'aurait pas souffert pour autant par rapport à la version implémentée puisqu'OpenGL permet d'effectuer un rendu dans une zone mémoire pouvant servir de texture. Le passage par un fichier de texture n'est par conséquent pas nécessaire.

La seconde aide à la navigation est la mise à disposition de zooms. Comme nous pouvons le voir dans la Figure 54, ces fenêtres de zoom grossissent la partie de la scène sur laquelle elles se trouvent. Cette loupe s'avère très pratique pour visualiser en détail une portion de la scène.

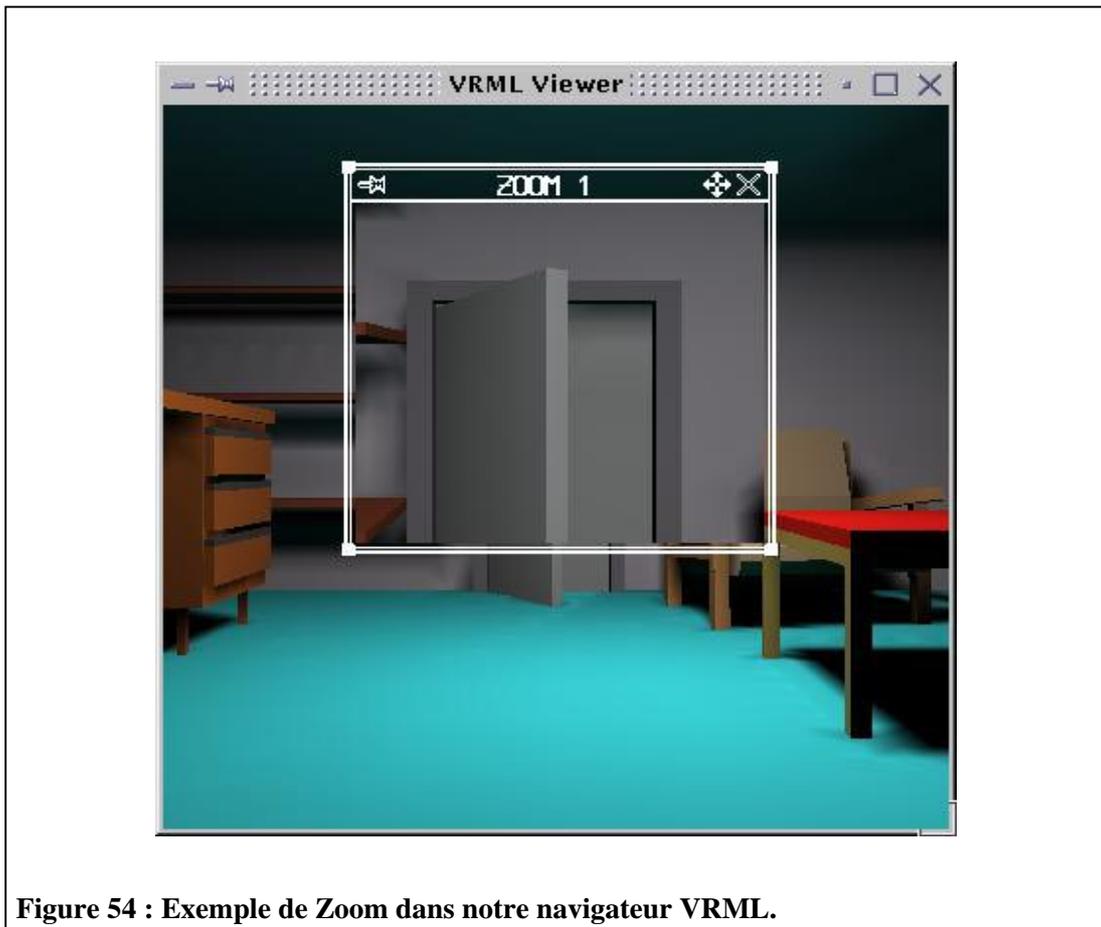


Figure 54 : Exemple de Zoom dans notre navigateur VRML.

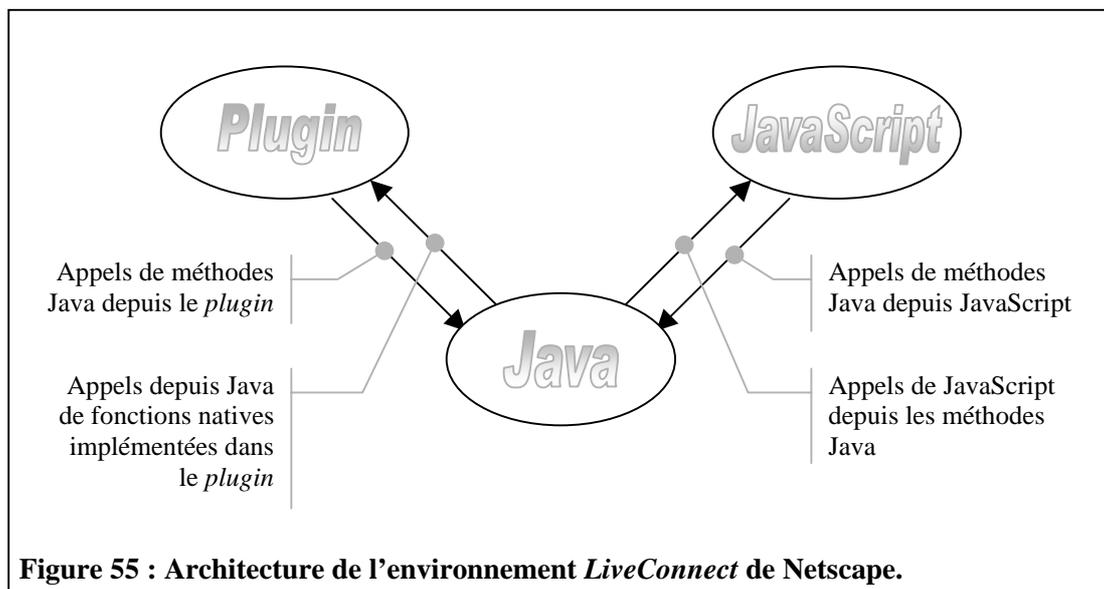
4.4 Transformation de l'application en un plugin VRML

Jusqu'à la version 2.0 (1996), Netscape permettait uniquement de lancer une application externe pour gérer une URL d'un type MIME non pris en charge par le navigateur lui-même. Ce mode de fonctionnement était connu sous le nom d'*application helper*. Les *plugins* permettent désormais d'intégrer le contenu de ce type d'URL dans le navigateur. Le mécanisme décrit page 72 menant de l'ouverture d'une l'URL au lancement de l'application la prenant en charge est le même pour les deux solutions. La différence entre ces deux modes vient du type d'application et de la gestion de la fenêtre dans laquelle elle est affichée. Dans le premier cas, l'application lancée est une application normale (exécutable depuis le système d'exploitation) gérant sa propre fenêtre. Dans le cas d'un *plugin*, l'application est une bibliothèque dynamique dont la fenêtre principale est gérée par le navigateur.

Pour construire un *plugin*, il nous faut donc un moyen pour récupérer le descripteur de la fenêtre créée par le navigateur. Ce descripteur sera utilisé pour définir dans le *plugin* quelle fenêtre reçoit les sorties écran. Pour cela, il existe, pour Netscape, un kit de développement permettant de

créer une instance du *plugin*, de la détruire, de gérer les URL reçues... Ce kit existe pour différentes plates-formes mais uniquement pour le langage C⁴³.

Le kit de développement fournit également un environnement appelé *LiveConnect*. Il permet aux programmeurs de tirer partie des langages Java et JavaScript pour gérer les interactions avec le *plugin* par l'intermédiaire d'un panneau de contrôle par exemple. La Figure 55 schématise l'architecture offerte par *LiveConnect*. Alors qu'un *plugin* et Java communiquent directement tout comme Java et JavaScript, JavaScript agit sur un *plugin* par l'intermédiaire de Java. Pour appeler des méthodes Java (ou JavaScript) depuis un *plugin*, il suffit d'utiliser le JRI⁴⁴ (*Java Runtime Interface*) pour générer un fichier entête en C. Il est bien sûr également possible de faire l'inverse : appeler les fonctions du *plugin* depuis Java ou JavaScript. Cette manière de faire ressemble fort à ce qui doit être fait avec le JNI (*Java Native Interface*) dont nous parlerons plus loin. Une documentation détaillée⁴⁵ existant sur cet environnement *LiveConnect* nous ne rentrons pas plus dans les détails, d'autant plus que nous ne l'utilisons pas.



Pour écrire le *plugin* VRML, nous avons utilisé un squelette de programme fournit avec le kit de développement. Celui utilisé, sans prise en charge des fonctions propres à l'environnement *LiveConnect*, donne les fonctions à implémenter. Il en existe deux sortes, celles qui ont pour but la gestion globale du *plugin* et celles qui gèrent ses différentes instances.

Toutes les fonctions suivantes sont appelées par le navigateur lors de son initialisation ou lorsqu'il crée une nouvelle instance du *plugin*. Ces fonctions remplacent en quelque sorte la fonction *main* et les fonctions de gestion de la fenêtre principale dans notre navigateur autonome.

4.4.1 Les fonctions de gestion globale

Les fonctions liées à l'initialisation du *plugin* sont appelées une seule fois lors de l'ouverture de Netscape. Sur chacune des plates-formes, la fonction *NPP_Initialize* permet d'effectuer des opérations globales devant avoir lieu lorsque Netscape charge le *plugin*, avant toute création d'une

⁴³ Les différentes versions du kit de développement et une documentation se trouvent à l'URL suivante : http://home.netscape.com/comprod/development_partners/plugin_api/index.html.

⁴⁴ L'URL de la documentation pour le JRI : <http://home.netscape.com/eng/jri/>.

⁴⁵ La documentation complète sur l'environnement *LiveConnect* est disponible à l'adresse suivante : <http://home.netscape.com/eng/mozilla/3.0/handbook/plugins/pjava.htm>.

instance.

C'est à ce moment que Netscape doit récupérer les informations sur le *plugin* comme son nom et sa description et le(s) type(s) MIME qu'il traite. Ce sont ces informations qui seront affichées lorsque l'on demande la liste des *plugins* installés dans le menu « Aide » rubrique « Information sur les plugins ». Sous Windows et Linux, la récupération de ces informations se fait de manière différente :

- sous Linux, Netscape appelle deux fonctions qui doivent faire partie de la librairie dynamique et qui doivent donc être écrites dans le programme du *plugin* : `NPP_GetMIMEDescription` qui retourne le(s) type(s) MIME traité(s) dans une chaîne de caractères et `NPP_GetValue` qui, en fonction de la valeur passée en argument, doit renvoyer le nom ou la description du *plugin* également dans une chaîne de caractères.
- sous Windows, Netscape récupère ces informations qui sont directement écrites dans la librairie dynamique grâce à un fichier ressource lié.

Un *plugin* a la possibilité de faire des appels à des fonctions présentes dans une classe Java par l'intermédiaire de l'interface *LiveConnect*. Netscape, lors de l'initialisation du *plugin*, demande le nom de cette classe par un appel à la fonction `NPP_GetJavaClass`. Puisque dans notre cas nous n'utilisons pas de classe Java, nous renvoyons simplement `NULL`.

Lorsque le navigateur est fermé, Netscape appelle la fonction `NPP_Shutdown` de tous les *plugins* chargés. C'est dans cette fonction que doit être désallouée la mémoire réservée dans la fonction `NPP_Initialize` pour les structures globales. Dans cette fonction doit être également libérée la mémoire utilisée par le *plugin* lui-même grâce à un appel à la fonction `NPN_MemFree`.

4.4.2 Les fonctions de gestion des instances

Nous voyons ici les fonctions gérant les différentes instances d'un *plugin*. Par commodité, nous donnons sous forme de liste les fonctions que nous avons effectivement utilisées.

- `NPP_New` est appelée pour initialiser une nouvelle instance. Les opérations à effectuer dans cette fonction sont l'allocation de mémoire et l'initialisation d'une structure gardant les informations sur cette instance. Les informations à initialiser sont :
 - le mode de rendu : dans une fenêtre incluse dans une page HTML (par la balise `<EMBED>`), dans tout l'espace disponible à l'intérieur de la fenêtre de Netscape ou dans le fond de cette fenêtre.
 - le descripteur de fenêtre et la procédure de gestion des événements de type *Callback* sont mis à `NULL`.
 - les autres informations que le programmeur ajoute dans la structure d'une instance.
- `NPP_Destroy` libère la mémoire utilisée par une instance que Netscape détruit.
- `NPP_SetWindow` est appelée lorsqu'une fenêtre d'une instance est créée ou modifiée. On sauvegarde la taille et la position de cette fenêtre dans la structure d'instance. Si nous sommes dans le cas où la fenêtre est créée, on sauvegarde aussi le descripteur et la fonction gérant les événements. Pour notre *plugin* VRML, nous initialisons également la fenêtre pour qu'elle puisse recevoir des commandes OpenGL. Sous UNIX, nous utilisons les API GLX et Mesa et sous Windows les API `glAUX` et OpenGL. Nous modifions

également les paramètres du point de vue à chaque fois que la fenêtre est modifiée puisque ceux-ci sont dépendants de la taille de la fenêtre.

- `NPP_NewStream` est la fonction appelée lorsqu'une nouvelle URL est chargée. Le programmeur doit y spécifier ce qu'il souhaite faire de ce flux de données en fonction de son type MIME et son type d'accès possible (séquentiel ou aléatoire). Puisque notre *parser* VRML parcourt des fichiers, nous avons choisi de sauvegarder le flux sous forme de fichier dans le cache local.
- `NPP_StreamAsFile` est appelée lorsque l'URL a été complètement chargée et a été sauvegardée dans un fichier du cache local. Dans notre cas, c'est le moment où nous devons parcourir ce fichier.

4.5 Travail restant à faire sur le parser

Enormément de travail reste à faire pour implémenter des fonctionnalités qui individuellement ne demandent pas beaucoup de temps de développement mais qui ensemble représentent un bon nombre de lignes de code à écrire.

Le *parser* en lui-même peut être considéré comme étant terminé puisqu'il reconnaît la plupart (plus précisément toutes jusqu'ici) des expressions VRML versions 1.0 et 2.0. La seule chose à y ajouter serait la possibilité de parser un code en *JavaScript* inclut dans le champ *url* du nœud *Script*. Néanmoins, d'une part, il existe un *parser* de code en *JavaScript* que l'on ne devrait pas avoir trop de mal à intégrer à notre projet, et, d'autre part, le parcours des expressions *JavaScript* se ferait à l'extérieur de l'interpréteur VRML : pendant le parcours de l'arbre, lors du traitement du champ *url* dans le nœud *Script*.

Ce qui est loin d'être terminé est la traduction de l'arbre des nœuds VRML obtenu après parcours du fichier source en des commandes d'API 3D. En effet, beaucoup de nœuds ne sont pas pris en charge et principalement ceux qui servent à gérer les aspects d'animation et d'interactivité.

Les animations ne sont pas et ne peuvent pas être traitées efficacement avec la liste de commandes que nous avons utilisée pour stocker la totalité de la scène. En effet, comme nous l'avons dit en présentant OpenGL, une liste de commandes ne peut pas être mise à jour, sauf si l'on en crée une nouvelle pour remplacer la précédente. La solution pour prendre en charge les animations, est donc d'utiliser une nouvelle structure intermédiaire gérée par nos soins permettant de stocker les commandes sous une forme séquentielle (et non dans un arbre) et que l'on peut modifier ponctuellement (contrairement à la liste de commande). Le rendu de la scène sera alors effectué en parcourant, pour chaque trame, cette structure (tableau ou liste chaînée) et en traduisant les nœuds VRML en commandes des API 3D.

Lorsque le moteur 3D le peut (lorsqu'il est inoccupé : dans la fonction `Idle` pour OpenGL), il se charge de voir les états des différentes animations, c'est-à-dire des senseurs et interpolateurs et de mettre à jour la scène en fonction de ceux-ci comme l'illustre la Figure 56.

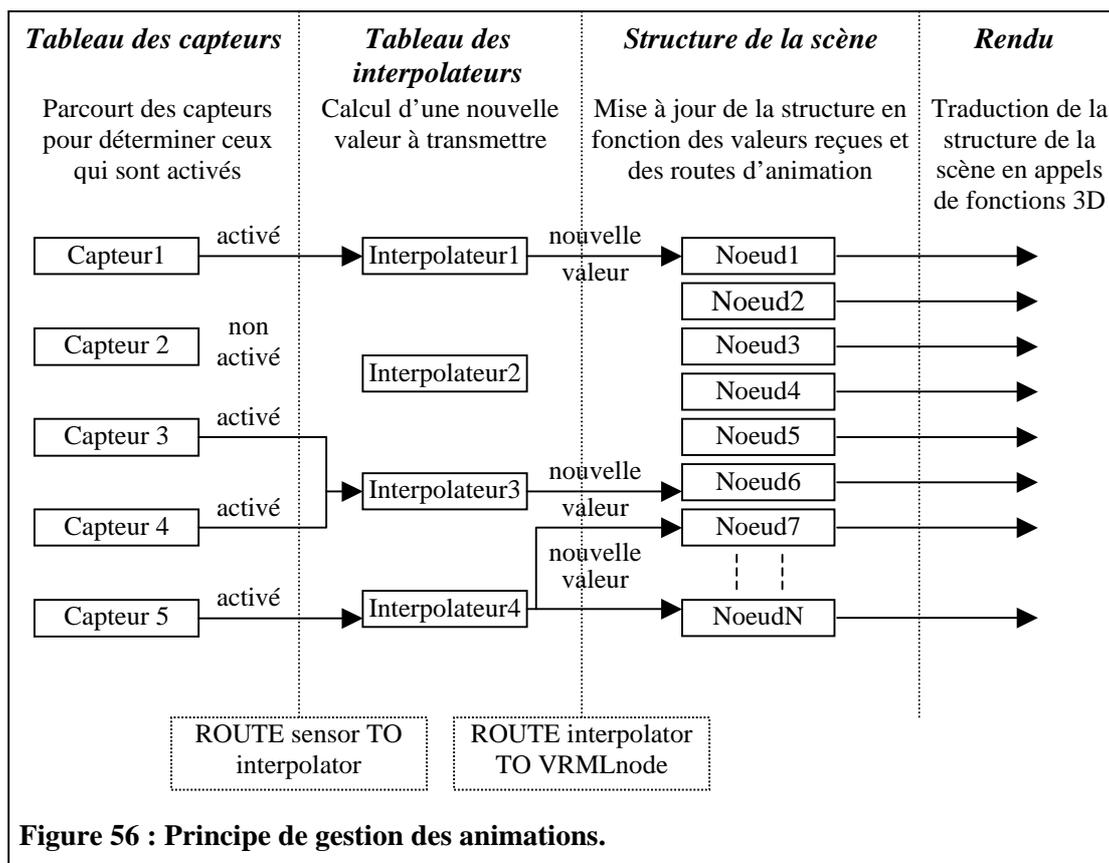
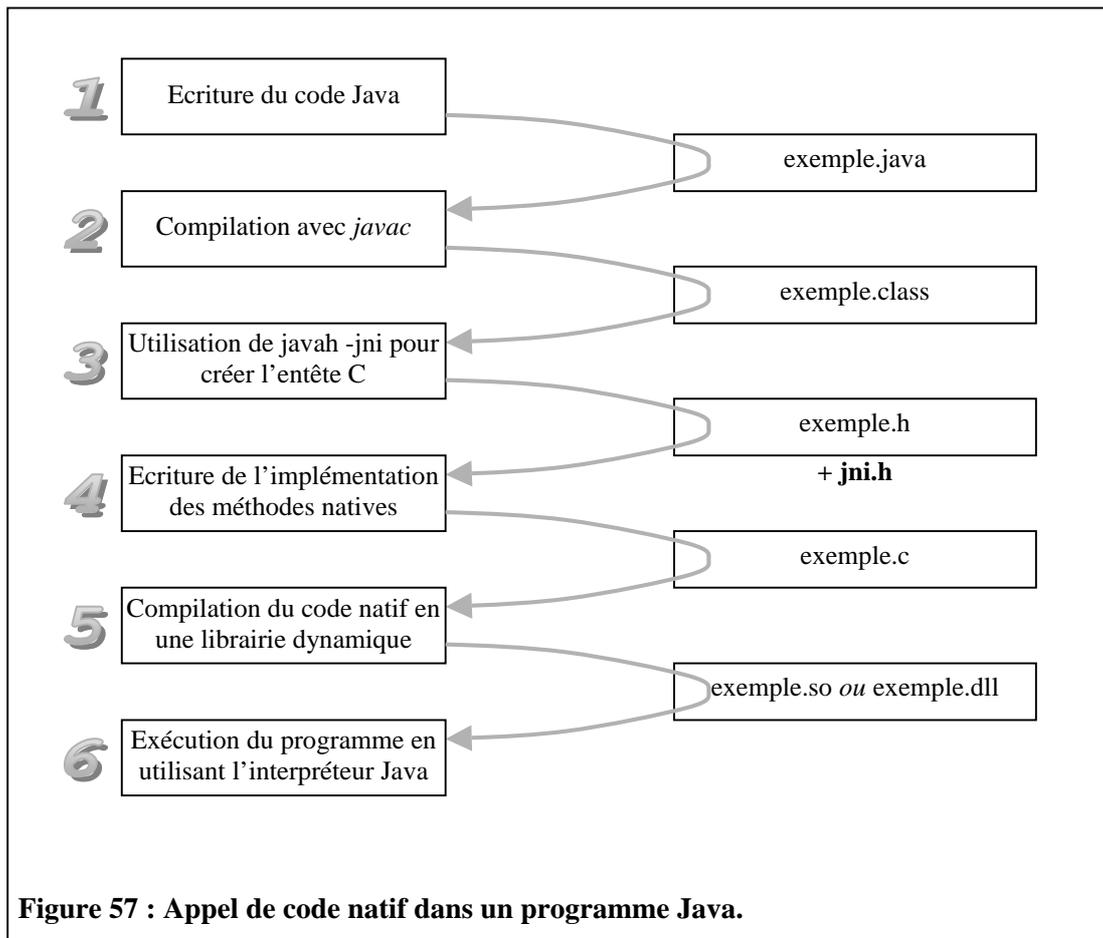


Figure 56 : Principe de gestion des animations.

Les listes de commandes peuvent toutefois être utilisées pour stocker les objets VRML considérés comme élémentaires. Ces objets sont les nœuds définis par un mot clé DEF ou toute une hiérarchie de nœuds qui ne comporte aucun mot clé DEF. Puisque ce sont les nœuds définis qui sont utilisables dans des animations, nous pourrions y accéder indépendamment. Du côté de l'implémentation, il suffira d'utiliser une liste chaînée de listes de commandes au lieu d'utiliser notre propre liste de commandes globale. Cependant, cette solution ne permettra pas d'utiliser les langages de script de la même manière qu'ils interviennent sur un autre moteur. En effet, comme nous le verrons dans la partie traitant de nos travaux parallèles, un script Java peut accéder et modifier tout nœud défini ou non. Pour autoriser un tel comportement il faut conserver l'arbre et le parcourir pour effectuer le rendu de la scène ; sans utiliser de listes chaînées.

L'aspect sur lequel la réflexion est moins avancée concerne l'exécution des scripts qui nécessite bien évidemment que l'on utilise un moteur 3D en mode immédiat comme pour les animations. La manière d'affecter le comportement du navigateur VRML depuis des appels à des méthodes de classes Java se fait, comme pour toute liaison entre un programme natif et une application Java, par un *binding*. Il s'agit d'implémenter des classes Java qui, elles-mêmes, référencent des fonctions du navigateur accessibles par une librairie dynamique. Cette dernière est l'interface entre l'application compilée et les classes pouvant la piloter. Le JNI (*Java Native Interface*) de *Sun Microsystems* est justement conçu pour permettre aux programmeurs d'applications Java d'inclure un code natif. La Figure 57 illustre les différentes étapes nécessaires pour intégrer et exécuter du code natif depuis une classe Java grâce à ce JNI.



Nous explicitons ici les opérations à effectuer pour chacune des étapes :

1. Ecriture du code Java

Une classe exemple utilisant une fonction native est fournie ci-dessous. Dans cette classe, la fonction native est déclarée (ligne 2) et la librairie dynamique est chargée (lignes 3 à 5) en vue d'utiliser la fonction (ligne 7) dans une méthode Java.

```

1.     class exemple {
2.         public native void maFonction();
3.         static {
4.             System.loadLibrary("exemple");
5.         }
6.         public static void main(String[] args) {
7.             new exemple().maFonction ();
8.         }
9.     }

```

2. Compilation du code Java

```
javac exemple.java
```

3. Création de l'entête C

```
javah -jni HelloWorld
```

Les nom des fonctions déclarées dans l'entête ont toujours la même syntaxe :
``Java_'+{nom de la classe}+'_'+{nom de la méthode}`

Ainsi, notre fonction `maFonction` de la classe `exemple` sera appelée en C :
`Java_exemple_maFonction.`

Les fonctions auront toutes le type `JNIEXPORT void JNICALL` et attendront deux arguments du type `JNIEnv *` et `jobject`.

4. Ecriture des méthodes natives

Les fonctions exportées par le JNI doivent être implémentées dans le fichier C. Elles doivent également respecter la syntaxe précédente. Dans notre cas, nous devons donc écrire le corps de la fonction :

```
JNIEXPORT void JNICALL
Java_exemple_maFonction(JNIEnv *env, jobject obj)
{
}
```

5. Compilation du code natif (sous UNIX)

```
cc -G -o libexemple.so -I/usr/local/java/include
-I/usr/local/java/include/solaris exemple.c
```

6. Exécution du programme

```
java exemple
```

Si une exception `UnsatisfiedLinkError` est levée alors la librairie contenant les fonctions natives n'a pas été trouvée.

Pour le *plugin*, nous pouvons également utiliser le JRI de Netscape dont nous avons parlé précédemment. Les différentes étapes sont quasiment les mêmes et elles sont détaillées dans la documentation de l'environnement *LiveConnect* (cf. note 45 page 121).

5 APPLICATIONS, TRAVAUX FUTURS

Résumé L'étude approfondie des moyens disponibles pour créer (voire diffuser) une scène 3D à un utilisateur final, ainsi que différentes techniques d'interaction, nous ont permis de réaliser quelques projets en marge de ce mémoire. De ces travaux, sont nés quatre articles dont les contenus sont détaillés.

5.1 Applications

Nous présentons, chronologiquement, les différents projets et articles qui sont issus de l'étude de VRML. Les deux premiers, le simulateur urbain et l'interface 3D de sélection d'ouvrages dans une bibliothèque numérique ont été réalisés avant ce mémoire d'ingénieur. Néanmoins, ils représentent la genèse de cette étude plus approfondie.

5.1.1 Simulateur Urbain

L'idée principale de l'éditeur d'espace urbain est de permettre la modélisation 3D d'un modèle urbain avec les outils d'étude habituels utilisés par un architecte, à savoir feutres et calques. Il autorisera ensuite de simuler une promenade dans un projet d'un point de vue sonore et visuel (d'où simulateur urbain). L'objectif était donc de proposer un outil ergonomique permettant de prototyper rapidement une maquette interactive 3D d'un projet.

En 1996, l'année de ce projet, est apparue la deuxième version de VRML. Afin de permettre aux architectes de communiquer leurs projets sur Internet et aux utilisateurs de se promener en temps-réel dans le nouvel espace urbain (à l'aide d'un *plugin*), ce format de fichier s'est imposé de suite comme le format de sauvegarde des scènes générées. Notre simulateur urbain peut ainsi être vu comme un modéleur VRML spécialisé.

L'objectif d'un urbaniste est d'étudier et de créer un espace urbain dans un site existant par ajout, suppression ou modification du tissu urbain, bâti, réseau, équipements, espaces verts. La modélisation d'un quartier est donc basée sur l'existant : le plan des courbes de niveaux du terrain, du bâti et, par exemple, d'emplacements d'espaces verts. Les paramètres tels que la hauteur de bâti, le type d'espace vert etc., seront associés en cliquant dans la zone du plan représentant l'objet à éditer. Les aspects d'animation de la rue (utilisant des vidéos en boucle) ainsi que les ambiances sonores, auraient également dû être automatiquement édités mais n'ont pas été pris en compte par manque de temps.

L'édition d'un espace urbain regroupe différentes étapes que nous avons considérées et pour lesquelles nous avons développé des outils. Chacune de ces étapes (schématisées dans la Figure 58), doit fournir à la fin du processus une scène complète. Cependant, le module permettant d'associer les différents éléments (c'est-à-dire le module intégrateur) n'a jamais été terminé. L'intégrateur fédère les autres modules. Puisqu'il n'a pas été complètement développé, tous les modules ne peuvent fonctionner que de manière autonome. Ce n'est cependant pas tout à fait vrai puisque le module bâti utilise le module multimédia pour la sélection des textures à appliquer aux différents immeubles. Intégrer tous les modules dans une application fédératrice les utilisant comme différents *plugins* n'est pas la plus grande partie du travail. La partie la plus délicate concerne plutôt la récupération des deux scènes (du terrain et du bâti) pour former la scène finale.

D'autre part, certains modules, pour la gestion des espaces verts et du mobilier urbain, pour ajouter des animations automatiques et pour l'intégration du son n'ont pas été développés. Ces

modules supplémentaires permettraient d'offrir un plus grand réalisme à la scène mais ne présentaient pas l'intérêt premier dans ce projet.

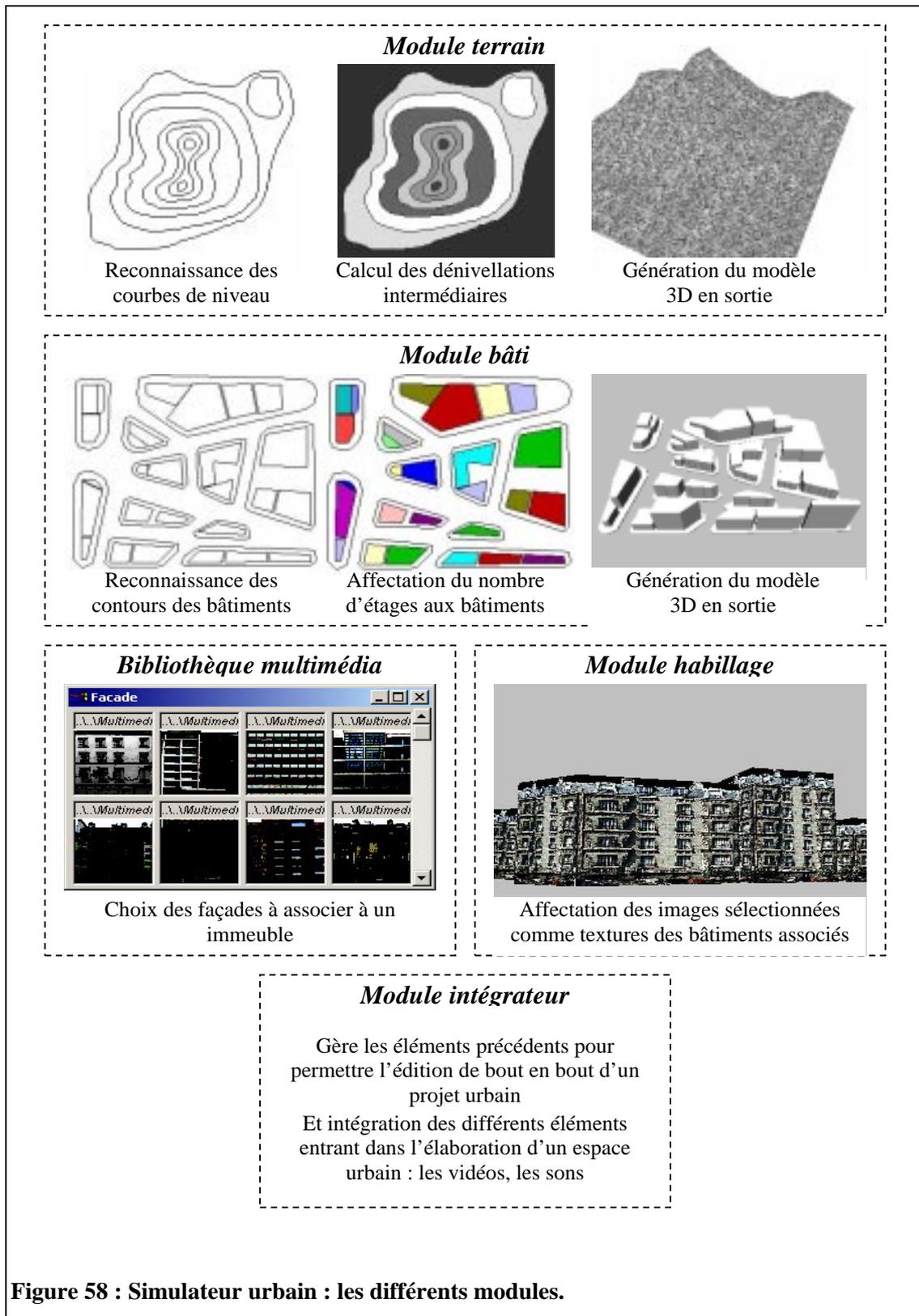


Figure 58 : Simulateur urbain : les différents modules.

La saisie du terrain

La plupart des mairies possèdent un fichier au format DXF (AutoCad) du terrain de leur commune. Si ce n'est pas le cas, il faut saisir le terrain concerné. A partir d'une carte comportant les courbes de niveau, notre logiciel génère automatiquement le terrain. La couleur de ces courbes étant toujours facilement différenciable de celles des autres informations, il est aisé, en connaissant cette couleur, de récupérer les informations souhaitées. La Figure 59 donne un exemple du fichier VRML généré en fonction des courbes données en entrée. Les élévations entre deux courbes successives sont calculées par une double interpolation linéaire horizontale et verticale.

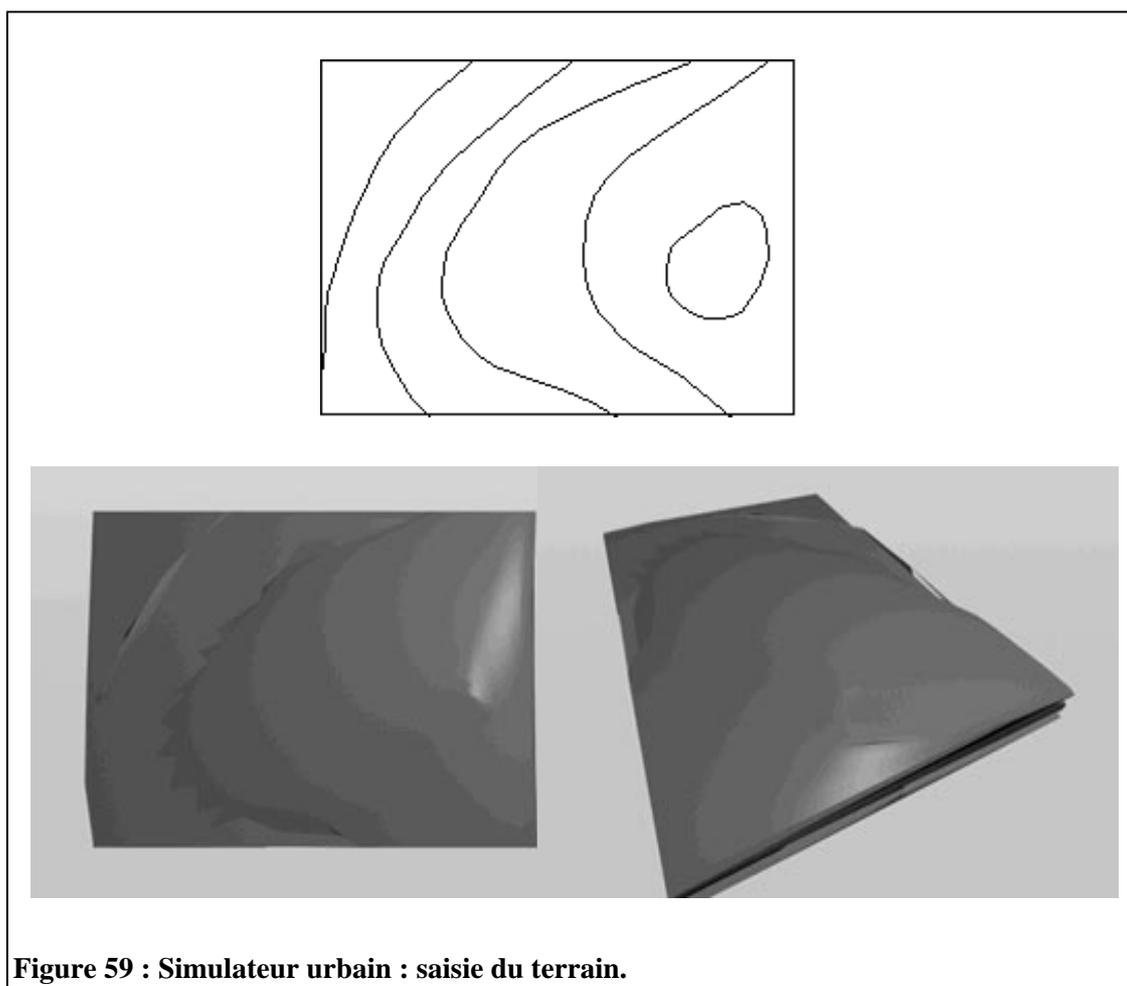


Figure 59 : Simulateur urbain : saisie du terrain.

La saisie du bâti existant

De même, on trace sur un plan de cadastre une zone fermée discriminant la limite du bâti à reconnaître. A chaque bloc reconnu, on alloue un nombre d'étages ainsi qu'un type de façade choisi dans une bibliothèque de données (une photo de façade existante prise par l'utilisateur peut être aussi scannée). La Figure 60 est une copie d'écran du module permettant d'affecter le nombre d'étages et les textures des façades aux immeubles. Les trois fenêtres dans cette figure représentent :

- la fenêtre « Terrain » contient le plan des différents bâtiments à habiller,
- la fenêtre « Options » permet de composer une texture en allouant une texture pour le rez-de-chaussée, une pour les étages et une pour le toit,
- la fenêtre « Preview » donne un aperçu de la façade de l'immeuble.

Il suffit de cliquer sur un bâtiment dans la première fenêtre pour que ses options apparaissent. Dans cette deuxième fenêtre, il est possible d'avoir un aperçu de la façade obtenue. Dans le fichier

VRML généré en sortie, une façade immeuble se composera d'autant de faces qu'il y a d'étages. Pour chacune de ces faces, la texture donnée est appliquée pour lui donner la bonne apparence. Une autre solution aurait pu être de générer une seule face par façade d'immeuble et de lui appliquer une texture fabriquée avec les sous-textures des étages choisies. Cependant, cela nécessiterait de créer autant de textures qu'il n'y a de façades différentes dans la scène.

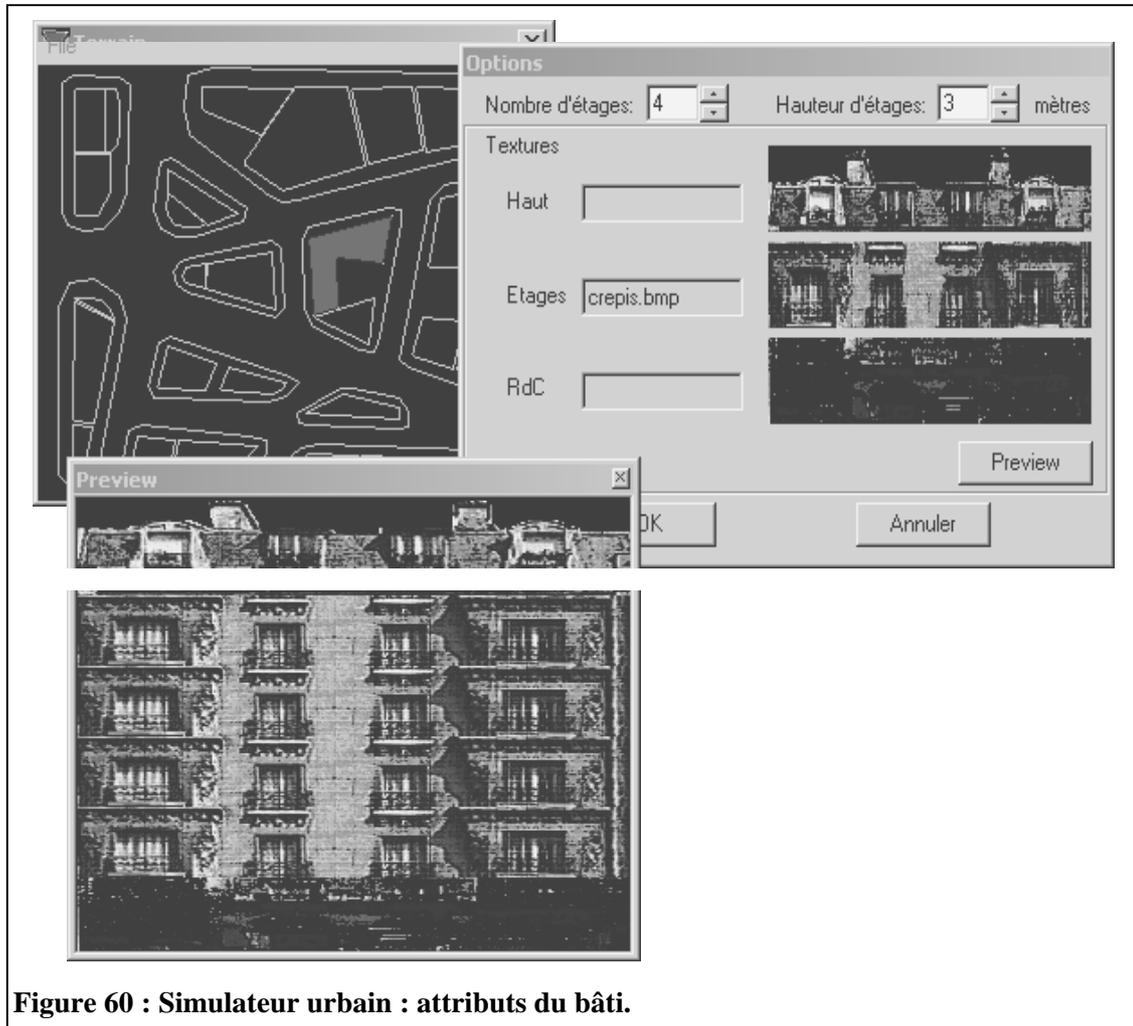


Figure 60 : Simulateur urbain : attributs du bâti.

5.1.2 Interfaces 3D pour les bibliothèques numériques

Maintenant que l'on sait sauvegarder tout livre, toute musique, tout film, ..., sous un format numérique et qu'il est possible de les consulter via Internet, le vieux rêve de H. G. Wells⁴⁶ datant de 1938 d'avoir une bibliothèque comportant toutes les cultures présentes et passées du monde semble presque une réalité. Une telle bibliothèque serait labyrinthique et ressemblerait fort à la bibliothèque de Babel imaginée par J. L. Borges⁴⁷. Cette bibliothèque qui porte le même nom que la tour à l'origine, selon la légende, de la séparation des hommes par des langues différentes, serait le rassemblement de toutes les cultures, rompant ainsi la punition divine infligée à l'époque de la tour. Le numérique donne la possibilité de recréer la bibliothèque d'Alexandrie au XX^{ème} siècle tout en faisant s'écrouler les murs de cette bibliothèque de Babel selon notre propre volonté en proposant des éléments multimédias sauvegardés ça et là sur Internet.

⁴⁶ Auteur britannique de science-fiction du XIX^{ème} siècle.

⁴⁷ BORGES J. L., *Fictions*, Collection Folio, Gallimard, 1983.

Jusqu'à une certaine taille de contenu, construire de telles bibliothèques n'est pas un problème. Le livre de Mickael Lesk [LES97] explique d'ailleurs comment les mettre en œuvre. L'avantage majeur des bibliothèques numériques est la possibilité de consulter des ouvrages depuis son ordinateur. Ceci paraît plus que nécessaire lorsque la bibliothèque contenant les livres que l'on veut consulter se trouve à l'autre bout du monde. Cette possibilité de consulter les ouvrages à distance nous intéresse particulièrement dans les cas des fonds anciens. En effet, il est plutôt difficile, voire impossible certaines fois, d'obtenir un droit pour consulter ces fonds. Cette protection est légitime puisqu'il peut s'agir de livres uniques sur lesquels les dégâts subis seraient irréparables. Une bibliothèque numérique autorise tout le monde, et non plus uniquement une élite, à consulter ces ouvrages à distance sans le moindre risque de les abîmer.

Une bibliothèque numérique a la possibilité d'être réorganisée facilement. Les classements et les recherches sont en effet très rapides. Il est, par exemple, possible de reclasser en quelques minutes tous les livres de la bibliothèque par date de parution au lieu du nom de l'auteur. Avec une bibliothèque traditionnelle, ce reclassement durerait des jours. On peut également dupliquer les livres pour les classer selon plusieurs critères différents. Cette duplication n'engendrant pas de coûts supplémentaires en numérique alors que, dans une bibliothèque réelle, il faut autant d'exemplaires du livre que de critères de classement.

Une bibliothèque peut ainsi être décrite par les quatre activités suivantes. Chacune d'elles est bien entendu automatisée si la bibliothèque est numérique.

- Collection : techniques pour comprendre à quelle information répondent les ressources.
- Organisation et représentation : classement et indexation des informations pour aider les utilisateurs.
- Accès : organisation physique de l'espace.
- Analyse, synthèse et propagation de l'information : réponse à des questions de référence, production de dossiers d'évaluation.

Depuis 1998, nous nous attelons donc à expérimenter l'utilisation de la 3D pour chacune des tâches précédentes. Une bibliothèque numérique sera composée de trois blocs distincts. Le premier bloc intégrant les trois premières activités précédentes sera une interface permettant de sélectionner les documents, c'est-à-dire de les rechercher et de prendre connaissance de leur fiche signalétique. Le deuxième bloc, appelé poste de lecture, sera une interface pour lire les documents sélectionnés. Le troisième bloc, quant à lui, autorisera l'utilisateur à annoter les ouvrages et mettre à disposition des autres lecteurs ses propres remarques. Ce dernier aspect pourra être intégré au poste de lecture.

Quelle que soit la bibliothèque numérique, il faut une interface faisant le lien avec l'utilisateur. L'interface la plus simple est celle qui présente les documents dans une liste textuelle de liens. Mais l'intérêt des bibliothèques numériques, à partir du moment où elles dépassent une certaine taille est de permettre à l'utilisateur d'interroger les catalogues à l'aide de requêtes qui peuvent être complexes. La mise en place de ce système de requêtes nécessite une interface pour la construction des requêtes et une interface de sélection des résultats. L'utilisateur peut alors sélectionner les livres qui l'intéresse afin de les consulter à l'aide de l'interface de lecture.

Les études que nous avons faites présentent deux des trois interfaces nécessaires pour travailler sur un livre numérique en 3D. Ces interfaces, prototypées à l'aide du langage VRML, sont en trois dimensions afin d'exploiter au mieux l'espace de travail. Notre préoccupation principale pour la construction de ces prototypes était de s'affranchir des aspects qui nous limitent dans le monde réel. Notre but n'était pas de recréer une bibliothèque à l'identique mais de profiter de tout l'espace 3D disponible sans contraintes. C'est pourquoi, les livres et l'utilisateur flottent au lieu de reposer sur des étagères ou sur le sol. Nous évitons également à l'utilisateur de reproduire des gestes qu'il effectuerait

dans une véritable bibliothèque : se déplacer jusqu'aux ouvrages d'intérêt, les emporter jusqu'à une table et enfin les consulter. Les déplacements dans une scène 3D est d'ailleurs assez peu intuitive pour les novices et nous évitons ainsi qu'ils soient désorientés.

Certaines bibliothèques « virtuellement réelle » comme [CEL99] et [KIP97], reproduisant à l'identique la bibliothèque réelle prise pour modèle, peuvent cependant offrir d'appréciables services. Les personnes habituées à une de ces bibliothèques retrouveront sans problème les différents ouvrages, puisqu'ils auront l'impression (et c'est vrai) de la connaître déjà. Pour nos expérimentations, nous avons utilisé les ouvrages numérisés pour le CNUM⁴⁸ (Conservatoire NUMérique des arts et Métiers). Ils sont issus du fonds ancien du CNAM qui n'est pas facilement accessible. Reproduire cet espace (qui n'est d'ailleurs pas prévu pour être visité ; les livres y sont stockés sur des étagères jusqu'au plafond) à l'identique n'a aucun intérêt. Nous préférons donc offrir des services autres, en espérant rendre les tâches de recherche et consultation des ouvrages plus faciles.

Interface de sélection

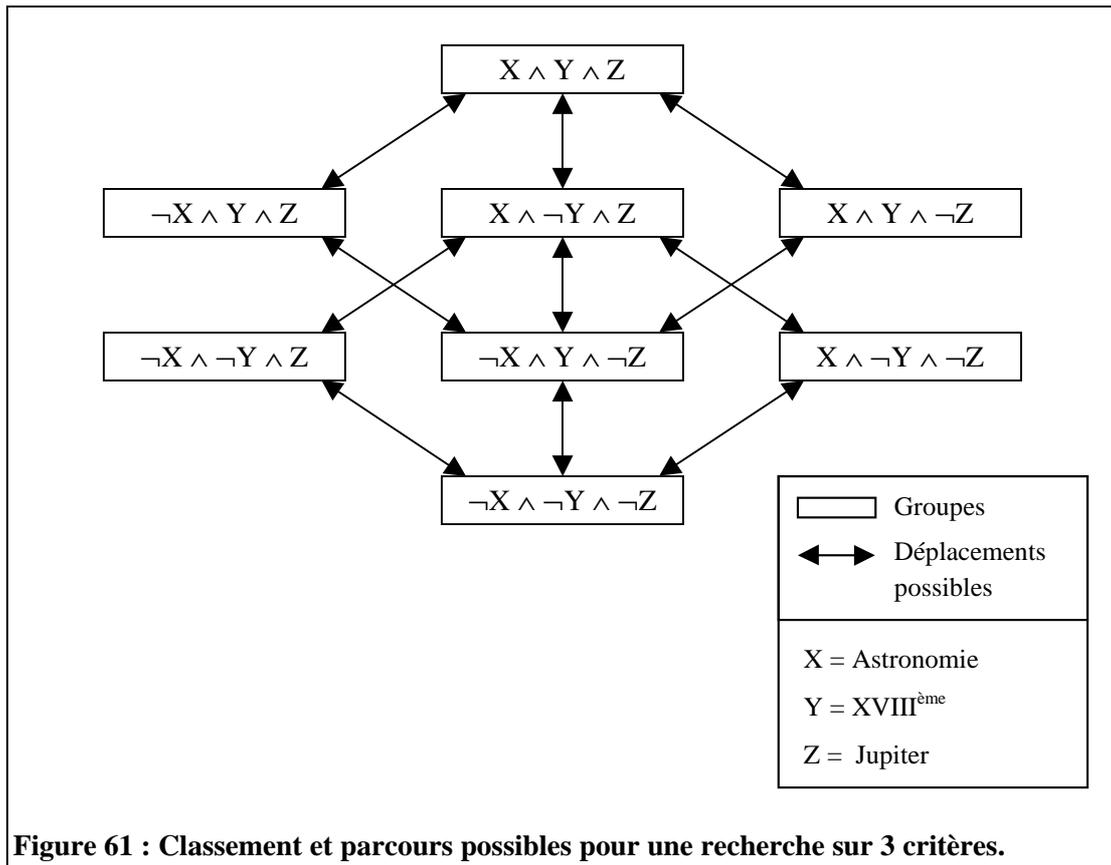
Pour la sélection des ouvrages, la 3D permet non seulement d'éviter la solution à base de longues listes interminables mais également d'offrir des informations visuelles supplémentaires grâce aux images des tranches des livres. Elle offre également la possibilité de lier la navigation dans la bibliothèque avec l'élargissement ou l'affinement de critères de recherche. Ce que nous voulons mettre en place, c'est la création "à la volée" d'une nouvelle organisation des livres dans la bibliothèque à chaque requête des clients. Les résultats seront alors séparés en groupes, ceux qui répondent à tous les critères, ceux qui répondent à tels critères et pas à tels autres, etc. Notre travail est alors d'offrir à l'utilisateur le moyen de passer d'un groupe d'ouvrages à un autre pour, soit affiner, soit élargir sa recherche.

Cette façon de construire et de soumettre les résultats à l'utilisateur favorise la découverte visuelle puisqu'il est possible de regrouper les ouvrages ne répondant pas à tous les critères à côté du groupe d'ouvrages pertinents. Or, cette manière d'aller « pêcher » un livre dans une bibliothèque est une heuristique très employée. On découvre souvent un livre intéressant par simple coup d'œil à côté de ceux que l'on recherchait vraiment. C'est pourquoi nous avons proposé cette manière de présenter les résultats aux requêtes. L'apparence d'un livre (sa taille, sa tranche) devient alors une information aussi importante que son titre ou son auteur et nous facilitons l'appréhension de cette information par l'utilisation des textures des livres. De cette manière, nous répondons à l'extrait suivant de A. Mengel : “ Les livres s'affirment grâce à leurs titres, leurs auteurs, leurs places dans un catalogue ou une bibliothèque, leurs illustrations de couvertures. [...] Je juge un livre à sa couverture ; je juge un livre à sa forme. ” [MEN98]. La manière de retranscrire cette citation en une interface 3D a été présentée à la conférence ACM Digital Library 1998 [CUB98].

Du côté pratique, pour créer une telle interface de recherche, nous utilisons un script CGI écrit en langage C. Il génère et renvoie dans une réponse HTTP une scène VRML décrivant la bibliothèque organisée en fonctions des critères de recherche. Les critères de recherche sont saisis à l'aide d'un formulaire classique décrit en HTML dont le bouton de soumission appelle le script CGI sur le serveur.

Afin de clarifier l'idée de liaison entre la réponse à une requête sous forme de groupes et la navigation, considérons par exemple le résultat d'une recherche sur trois critères pour trouver les ouvrages concernant les traités d'astronomie écrits au XVIII^{ème} siècle et parlant de la planète Jupiter. La Figure 61 montre, d'une part, les différents groupes d'ouvrages symbolisés par les nœuds et caractérisés par des opérations booléennes "et" et "non" sur les critères (vrai si les livres dans le groupe répondent au critère, faux sinon), et, d'autre part, les possibilités de navigation d'un nœud à un autre symbolisées par les liens.

⁴⁸ Accessible à l'URL <http://cnum.cnam.fr>.



Chacun des groupes est unique et leur nombre équivaut au nombre de permutations de l'ensemble des trois valeurs booléennes associées aux critères, de telle manière qu'aucun livre ne pourra être dans plusieurs groupes à la fois. On comprend, par exemple, que dans le groupe $X \wedge \neg Y \wedge Z$ se trouveront les livres concernant l'Astrologie et la planète Jupiter mais non écrits au XVIII^{ème} siècle. Les déplacements possibles de l'utilisateur depuis cet exemple de groupe pourront se faire selon 3 directions (la négation de chacun des critères), de telle manière qu'il pourra atteindre les groupes suivants :

- $\neg X \wedge \neg Y \wedge Z$ par négation du critère X
- $X \wedge Y \wedge Z$ par négation du critère Y
- $X \wedge \neg Y \wedge \neg Z$ par négation du critère Z

De manière générale, il y aura pour n critères de recherches :

- 2^n nœuds ou groupes de livres,
- $\sum_{p=0}^n C_n^p \cdot p$ liens ou chemins de navigation.

De chacun des nœuds de ce graphe il y aura n chemins possibles, correspondants à la négation d'un des n critères.

Il peut exister plusieurs représentations à partir de ce modèle. La première solution envisagée fut de représenter les groupes comme un arbre à l'envers, la racine étant le groupe des livres totalement pertinents. L'utilisateur aurait alors eu la possibilité de descendre ou de remonter le long d'un chemin pour affiner ou élargir sa recherche. Cependant, cette solution ne bénéficie pas des apports de la 3^{ème} dimension et oblige l'utilisateur à remonter des chemins au lieu d'emprunter tous ceux de la Figure 61.

Nous avons alors pensé à un système presque équivalent mais dans une espèce d'atomium comme celui de la Figure 62. Chacune des boules correspond à un groupe de livres. L'utilisateur aurait été placé dans la boule de l'atomium rassemblant les livres qui répondent à tous les critères ; il aurait alors eu la possibilité de se déplacer directement d'un groupe à l'autre sans repasser comme dans l'arbre au groupe précédent. Bien que cette architecture puisse être intéressante, elle ressemble d'ailleurs fort à celle de la bibliothèque de Babel imaginée par J. L. Borges, son étude n'a pas été prolongée. En effet, nous avons pensé qu'il serait plus judicieux de placer tous les livres dans une seule et même géométrie. Ceci afin d'encourager le lecteur à regarder d'autres livres que ceux du groupe de livres pertinents.

En voyant une illustration de la roue à livres de Ramelli⁴⁹, nous avons alors imaginé une bibliothèque cylindrique en pensant, plus particulièrement, au positionnement des livres en fonction de leur pertinence à la requête faite par l'utilisateur (voir le schéma de la Figure 61). Nous nous sommes cependant heurtés à un problème pour placer des livres devant répondre à plus de deux critères. En effet, avec deux critères seulement, il est aisé de positionner les livres. Dans le schéma de droite de la Figure 62 donné en exemple, on place l'utilisateur dans le cylindre face aux livres pertinents et dos aux livres non pertinents. Lorsqu'il se déplace dans le cylindre vers le haut, il découvre des livres répondant uniquement au premier critère alors que s'il descend, il découvre les livres répondant au deuxième.

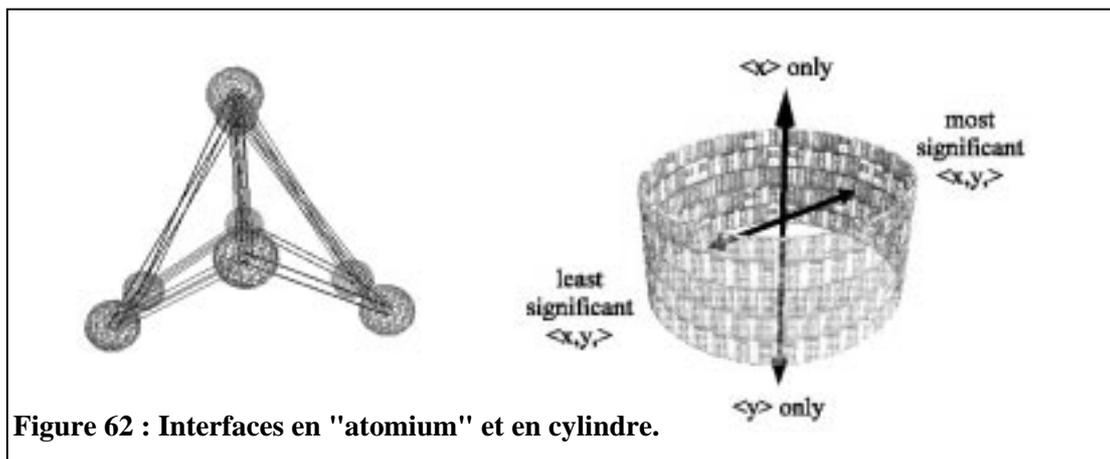


Figure 62 : Interfaces en "atomium" et en cylindre.

Enfin, pour permettre de placer plus de groupes que dans un cylindre, nous avons réparti les livres de telle manière qu'ils occupent l'intérieur d'une sphère. Considérons une requête comprenant n critères. La sphère dont le rayon est calculé en fonction du plus important des groupes est "découpée" en $n+1$ parties à l'aide de l'angle d'azimut (coordonnées sphériques) de la façon suivante :

- Le groupe des livres pertinents (répondant aux n critères) se trouve en azimut $-\pi/2$.
- Les groupes des livres répondant à $n-1$ critères se trouvent en azimut $-\pi/2 + \pi/n$.
- ...
- Les groupes des livres ne répondants qu'à $n-i$ des n critères se trouvent en azimut $-\pi/2 + \pi*i/n$.
- ...
- Le groupe des livres non pertinents (répondant à 0 critère) se trouve en azimut $\pi/2$.

Nous obtenons ainsi non pas des fuseaux horaires mais des fuseaux "nombre de critères pertinents". Puis, un second découpage est fait pour répartir uniformément ces groupes répondant à i des n critères de la requête en utilisant l'angle de la hauteur. Pour cela, nous savons que leur nombre

⁴⁹ Illustration présente dans l'histoire de la lecture de Mengel [MEN98].

est de $c=C_n^i$ (nombre de combinaisons des n critères pris i par i) pour la $i^{\text{ème}}$ des tranches précédentes. Nous obtenons donc un des groupes de la $i^{\text{ème}}$ tranche tous les $2\pi/c$. Il est alors facile de calculer le point P d'un groupe à l'aide des deux angles obtenus pour ce groupe et en fonction du rayon de la sphère.

Les livres appartenant à un groupe doivent alors être couchés sur un quadrilatère reposant sur un plan passant par le point P et perpendiculaire à la droite passant par P et le centre C de la sphère. Pour obtenir ce quadrilatère, nous calculons deux points supplémentaires. Le premier point (A) est obtenu en augmentant l'azimut du point P de la valeur $\pi/(2*(n+1))$. Le second point (B) est le résultat du produit vectoriel du vecteur PA avec le vecteur CP . Nous obtenons ainsi deux points sur la sphère qu'il faut réorienter pour que PA donne la verticale et PB l'horizontale, lorsque l'utilisateur se trouve en C et regarde vers P . Ceci est fait par une rotation des deux points de telle façon que PA , après rotation, se trouve sur l'axe Z . La méthode pour trouver la matrice de rotation est donnée dans le livre de Foley-van Dam [FOL90] à partir de la page 215. Le résultat obtenu est montré dans la Figure 63. Les livres peuvent alors être placés un par un en suivant les directions données par les deux vecteurs PA et PB de chaque groupe.

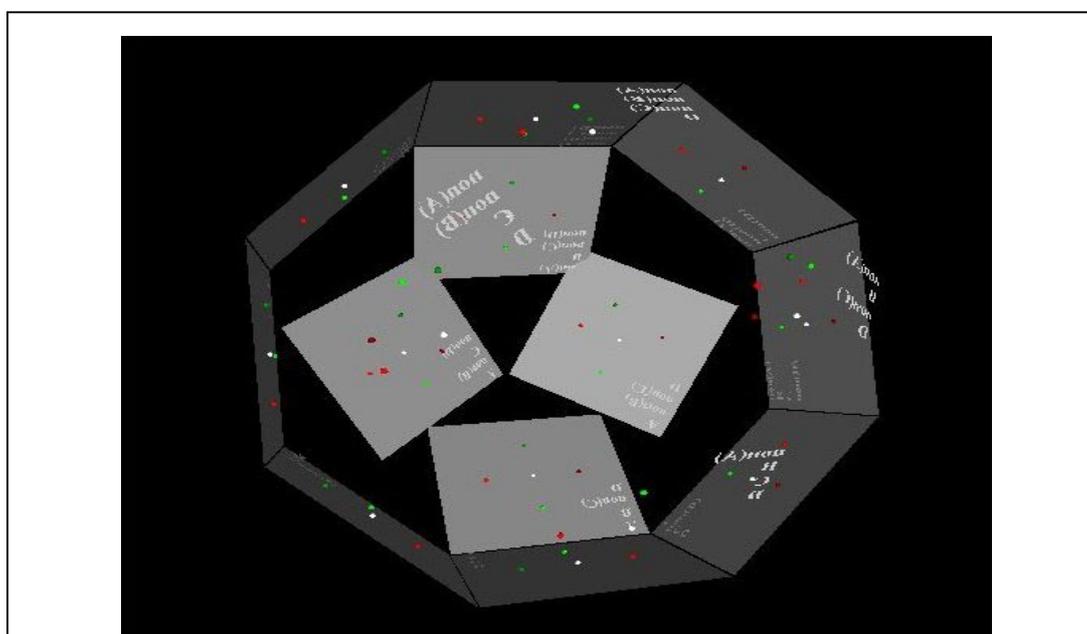


Figure 63 : Placement des groupes de livres dans une sphère.

Interface de lecture

De manière à être consistante avec l'interface de localisation présentée précédemment et afin de tirer partie également des possibilités d'organisation offerte par la 3D, un poste de lecture 3D semblait intéressant à étudier. Le poste de consultation doit permettre la lecture des ouvrages sélectionnés lors de la phase de recherche. D'une part, cette étude essaye de dégager les interactions 3D nécessaires sur l'objet livre mais, également, de mettre à jour certaines insuffisances du langage VRML pour spécifier les comportements 3D interactifs génériques (qui s'avère être un problème général aux langages de description des scènes 3D).

Le point important dans cette interface est de permettre aux utilisateurs d'organiser leur espace de travail, d'agencer les livres numérisés dans l'interface, de la même manière que les *web books* peuvent être organisés dans le *WebForager* [CAR96]. Pour faciliter ces interactions, nous avons présenté les éléments à manipuler dans des fenêtres 3D (c'est-à-dire des fenêtres 2D sur lesquels on interagit en 3D). Cela permet d'utiliser la connaissance qu'ont les utilisateurs des interactions sur les fenêtres 2D.

La scène initiale est constituée d'une seule fenêtre comportant les ouvrages sélectionnés. Pour l'instant, puisqu'il n'y a aucune liaison entre l'interface de localisation et le poste de lecture, ces livres sont tous les ouvrages consultables depuis le CNUM. Tous les livres sont représentés par leur tranche, offrant là encore des informations visuelles telles que la date de l'édition du livre grâce à son type de reliure, le nombre de page selon la largeur de la reliure ou encore le nombre total d'ouvrages.

Le moteur 3D utilisé pour cette expérimentation est celui géré par tout *plugin* VRML. En effet, nous utilisons le langage VRML pour décrire notre interface. Les interactions, quant à elles, sont spécifiées en Java, un des langages des langages de scripts permettant d'augmenter les possibilités de VRML. Notre scène peut, par conséquent, être dynamique (création, modification et suppression de contenu) grâce à ce couple VRML+Java. Elle est donc visualisable sous n'importe quelles applications prenant en charge ses deux langages : pratiquement tous les *plugins* VRML. L'utilisation de VRML comme langage support est une solution aussi valable que n'importe quelle autre. Ce qui peut paraître plus étonnant en revanche, c'est d'utiliser un *plugin* VRML comme moteur 3D de notre application à la place d'une API 3D. Cependant, cette approche offre quelques avantages :

- la bibliothèque est accessible depuis Internet,
- aucune installation n'est nécessaire pour lire les ouvrages de la bibliothèque, hormis celle d'un *plugin*,
- l'utilisateur emploie son *plugin* préféré dont les outils d'interactions qu'il connaît bien le satisfont.

De plus, l'utilisation d'un *plugin* VRML comme moteur 3D permet à l'utilisateur d'évoluer de deux manières différentes dans notre scène. La première d'entre elles se fait par l'intermédiaire des interacteurs disposés dans la scène. Elle s'adresse aux utilisateurs novices, ne connaissant pas bien les métaphores de navigation 3D. La deuxième, plutôt pour les experts qui la verront comme un complément à la première, s'appuie sur les outils de navigations standards d'un *plugin* : « walk », « pan », « study », etc.

Quant aux avantages d'une solution en 3D plutôt qu'une interface 2D, ils se résument en des considérations simples et courantes de la 3D :

- utilisation d'un volume théoriquement infini au lieu d'une surface fixe pour représenter les éléments de l'interface,
- utilisation de la profondeur pour détailler les informations (offrir un historique, afficher des informations dont la structure est véritablement 3D) ou les organiser,
- utilisation de la 3D pour son pouvoir de représentation.

D'autres avantages, non nécessairement propres à la 3D, sont néanmoins présent et fortement utilisés :

- la transparence qui permet d'apercevoir les éléments se trouvant derrière un autre,
- le point de vue de l'utilisateur sur les données peut changer (et en particulier un zoom est facilement implémenté),

Pour la consultation de documents, la 3D offre une solution au problème levé par l'utilisation de fenêtres « recouvrantes » (*overlapping windows*). Ces fenêtres, généralement utilisées dans les interfaces graphiques « modernes », sont gérées comme une pile. Dans la plupart des cas, uniquement la première est visible ; les autres étant soit recouvertes (partiellement ou entièrement) soit iconifiées. La Figure 64 montre l'interface de lecture 2D actuelle du CNUM juxtaposée au prototype d'interface 3D. Les documents sont ouverts et mis à l'échelle souhaitée et l'utilisateur sait toujours où ils se

trouvent grâce à la transparence. Dans une interface 2D, plusieurs fenêtres pourraient être ouvertes de la même manière mais la navigation de l'une à l'autre serait fort désagréable. Ici, les déplacements des fenêtres sont fluides, autorisant ainsi le lecteur à utiliser sa mémoire spatiale.

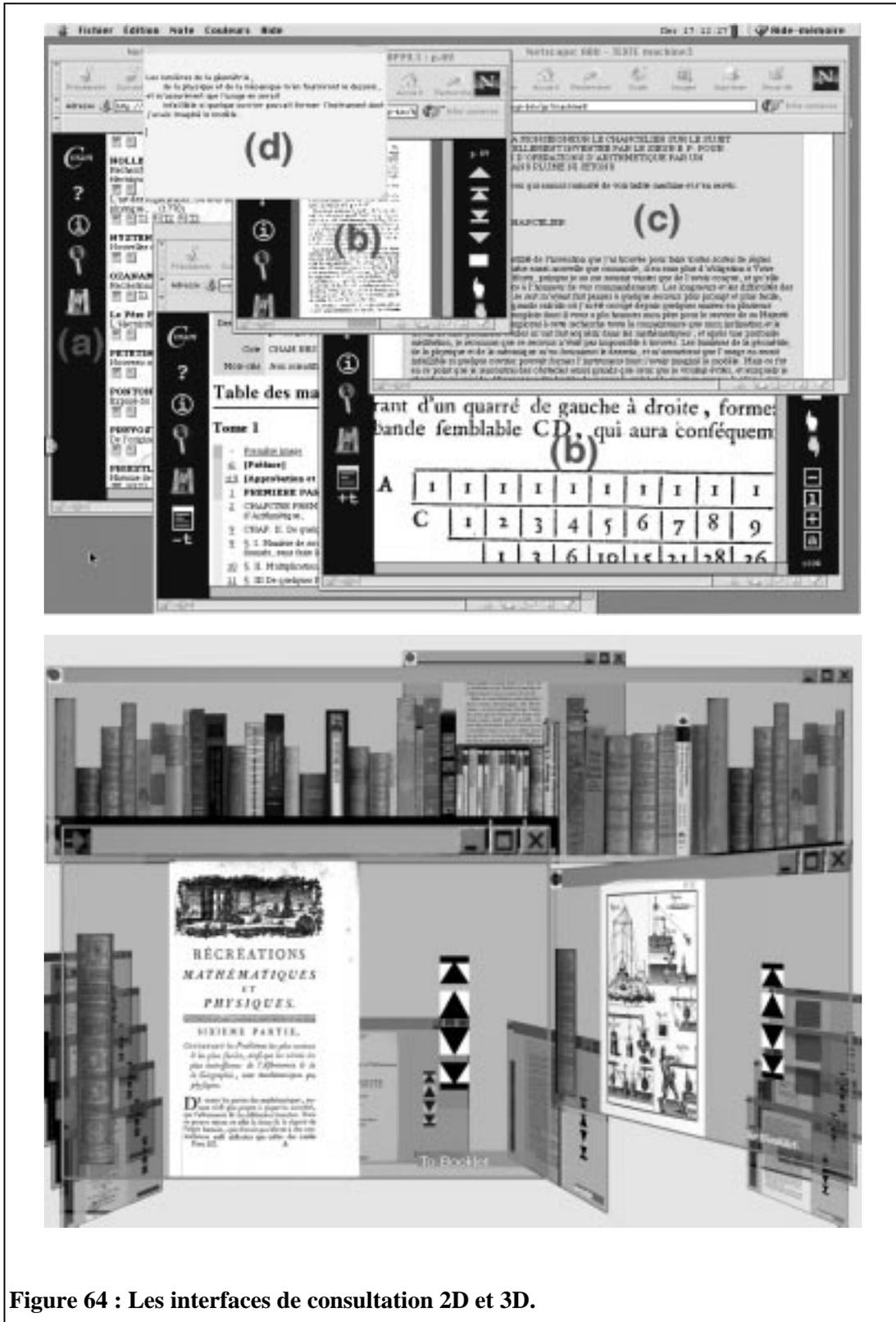


Figure 64 : Les interfaces de consultation 2D et 3D.

Tous les éléments dans l'interface sont en deux dimensions et peuvent donc être insérés dans

des fenêtres 3D transparentes. L'utilisateur peut agir directement sur les fenêtres pour les déplacer, les iconifier, les maximiser, les punaiser ou les fermer. Ses interactions issues des interfaces 2D sont bien connues et devraient satisfaire les utilisateurs novices qui n'ont pas ou peu d'expériences avec les manipulations d'objets en 3D. Ces utilisateurs peuvent même consulter les documents sans déplacer le point de vue sur la scène. Ils bénéficient néanmoins du rangement des fenêtres en perspective grâce au bouton d'iconification.

Des objets 3D auraient pu être une meilleure solution pour obtenir un meilleur effet visuel. Une métaphore de livre serait sans aucun doute plus pratique et compréhensible au premier abord. Les interactions avec cet objet pourraient être également beaucoup plus poussées. Par exemple, un outil pourrait être fourni pour feuilleter le livre afin d'appréhender la structure du livre ou retrouver un passage aidé par la mémorisation photographique que l'on a de lui. Cependant, ces interactions supplémentaires nécessiteraient d'accoler des *widgets* supplémentaires à côté du livre pour les activer. Soit l'apparence du livre en serait modifiée (et n'aurait plus rien de réel), soit les *widgets* devraient apparaître lorsque le livre serait activé. Nous ne nions pas le fait qu'une telle interface serait plus puissante. Cependant, dans le contexte de cette étude, le but recherché était d'offrir une interface intuitive. C'est pourquoi nous nous sommes appuyés sur des mécanismes d'interaction directement accessibles et facilement compréhensible car issus des interfaces 2D.

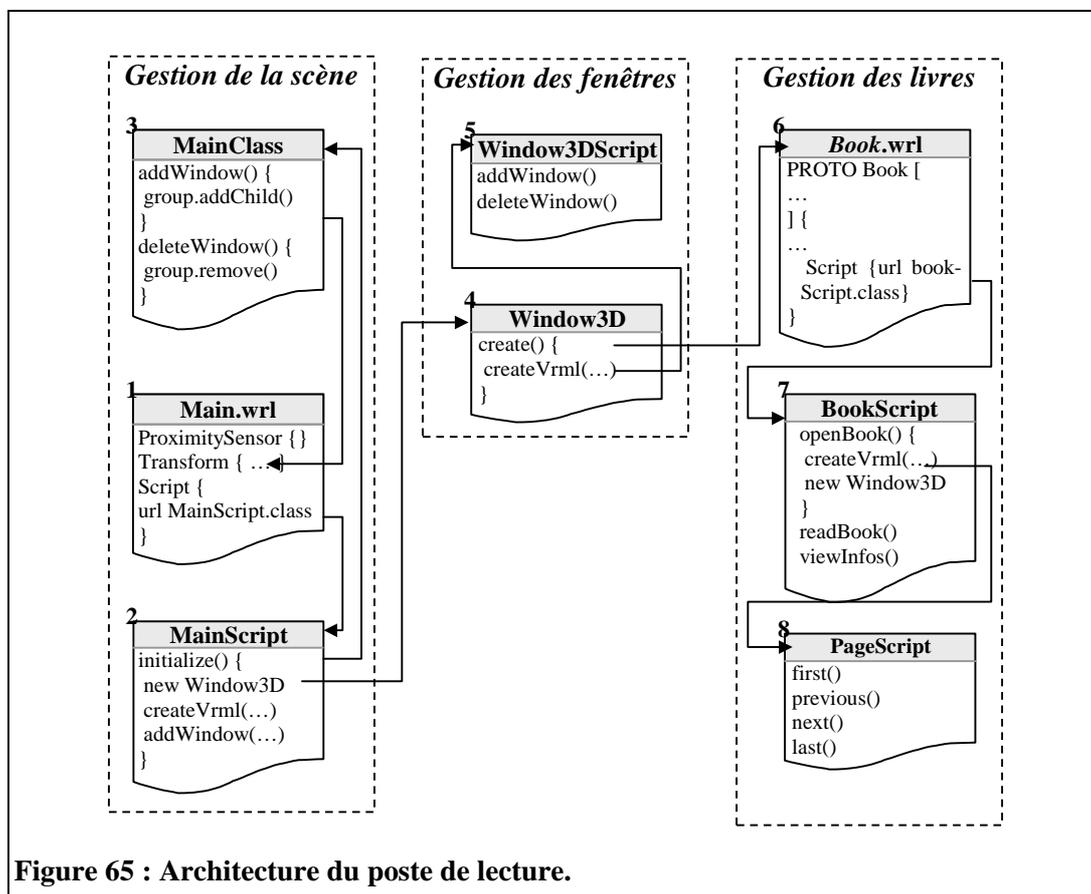


Figure 65 : Architecture du poste de lecture.

L'architecture du système et les détails d'implémentation ne seront pas présentés ici. Pour en prendre connaissance, on pourra se reporter aux articles dont nous donnons les références sur la page suivante. En résumant à l'extrême, le système se compose de trois blocs illustrés dans la Figure 65, chacun gérant un aspect particulier de la scène :

- le bloc principal (le gestionnaire de scène) lie les différents objets de la scène,
- le bloc de gestion des fenêtres ,

- le block de gestion des livres.

Chacun de ces blocs se compose de fichier(s) VRML et/ou de classes Java ; ces dernières prenant en charge les interactions sur les éléments du bloc dans lequel elles se trouvent. Quelque soit l'élément sur lequel interviennent les interactions, l'apparence de l'objet est modifiée la plupart du temps. Cela signifie que le graphe de scène VRML est modifié par les classes Java. Or, pour que les classes connaissent la sémantique des éléments du graphe, elles doivent le construire. De ce fait, les scripts représentent plus de 90% du code total de la scène. Ce couple VRML+Java peut être comparé à l'API Java3D puisque la syntaxe du langage VRML nous permet de décrire le graphe de scène et le langage Java est utilisé pour le modifier. L'apport de VRML en tant que langage support est dans ce cas négligeable et pour une simple et unique raison : l'absence d'interacteurs au niveau de la description des scènes.

C'est uniquement une voie que nous proposons pour résoudre ce problème et que nous envisageons d'étudier plus tard. Quelques nœuds de comportements simples permettraient de réduire la part des scripts et donc également l'effort de programmation nécessaire pour implémenter une telle scène. D'autre part, ces nœuds autoriseraient des non informaticiens à produire des scènes interactives. Dans le même ordre d'idée, les modeleurs spécialisés pourraient alors exporter des scènes interactives facilement. Les problèmes de confidentialité du code Java, qui représente tout le savoir dans ce genre de scène et qui est facilement consultable grâce à un traducteur de classe en code source Java, seraient aussi réglés. Ces considérations sur les problèmes posés par une telle solution ont été présentés à la conférence ERGO-IHM'2000 [CUB2000]. Quant à l'architecture de notre prototype, il a été présenté à la conférence Web3D'2001 [CUB2001].

5.1.3 Un bureau 3D

L'essoufflement certain des techniques graphiques 2D pour représenter et visualiser les nombreuses données qu'un système multimédia peut stocker est un constat reconnu. La tendance avec les applications actuelles est de réduire encore davantage la surface de travail par l'ajout de barres d'outils et de barres d'état. De nombreux projets tentent, pour régler ce problème, d'élargir la surface ou le volume du bureau, l'endroit où sont placés et manipulés les documents numériques.

Parmi ces techniques, les interfaces à changement d'échelle ont étudié l'utilisation de translations horizontales et de zooms verticaux sur les données. Ces interfaces font partie des interfaces 2.5D, englobant plus largement toutes les interfaces capables soit de modifier l'échelle des données, soit de gérer les déplacements du point de vue. Elles ont toutes démontrées l'importance des animations fluides pour la compréhension des modifications opérées. Elles ont également permis de mettre en évidence l'intérêt des espaces de travail théoriquement infinis pour organiser les données à visualiser.

Cependant, ces interfaces n'apportent rien de plus qu'une interface 3D. Elles se placent plutôt comme une alternative temporaire aux interfaces 3D qui demandent une puissance de calculs assez importante. Or, la plupart des ordinateurs d'entrée de gamme vendus aujourd'hui disposent d'une carte graphique capable d'accélérer les calculs 3D. Bientôt, les calculs nécessaires à la 3D représenteront le même pourcentage de temps d'utilisation du microprocesseur que les calculs requis par les interfaces 2.5D aujourd'hui. Ceci, parce que les microprocesseurs et les cartes graphiques 3D sont de plus en plus performants. La 3D offre, en plus de la 2.5D, la possibilité d'utiliser les rotations. L'organisation des objets s'en trouve facilitée et les mouvements du point d'observation plus naturels. Pour élargir son champs visuel, on tourne la tête plutôt que d'effectuer une translation du corps entier.

Ce prototype de bureau 3D est issu d'expérimentations sur les textures animées en OpenGL. Pour apprendre à gérer ces textures, nous avons utilisé les images de fenêtres du système XWindow dont le contenu était animé (par exemple, un mesureur de performance du système). Rapidement, il nous est apparu qu'il était possible d'intégrer toutes les fenêtres XWindow dans une scène 3D. Chaque fenêtre dans la scène étant représentée par une face texturée avec l'image de la fenêtre XWindow.

Le remplaçant du bureau 2D actuel n'a pas encore été trouvé. Cependant, nous sommes persuadés qu'il devra intégrer à la fois les applications avec une interface 2D et celles avec une interface 3D, qui sont de plus en plus nombreuses. Il n'est pas envisageable d'abandonner les applications 2D et la plupart n'ont aucune raison d'être transformées en application 3D. De la même manière, les applications en mode texte ont été intégrées dans toutes les interfaces graphiques 2D grâce à l'utilisation de console texte. Un sur-ensemble 3D permettrait donc non seulement d'englober les applications 2D mais également les applications en mode texte, elles-mêmes englobées par les interfaces 2D.

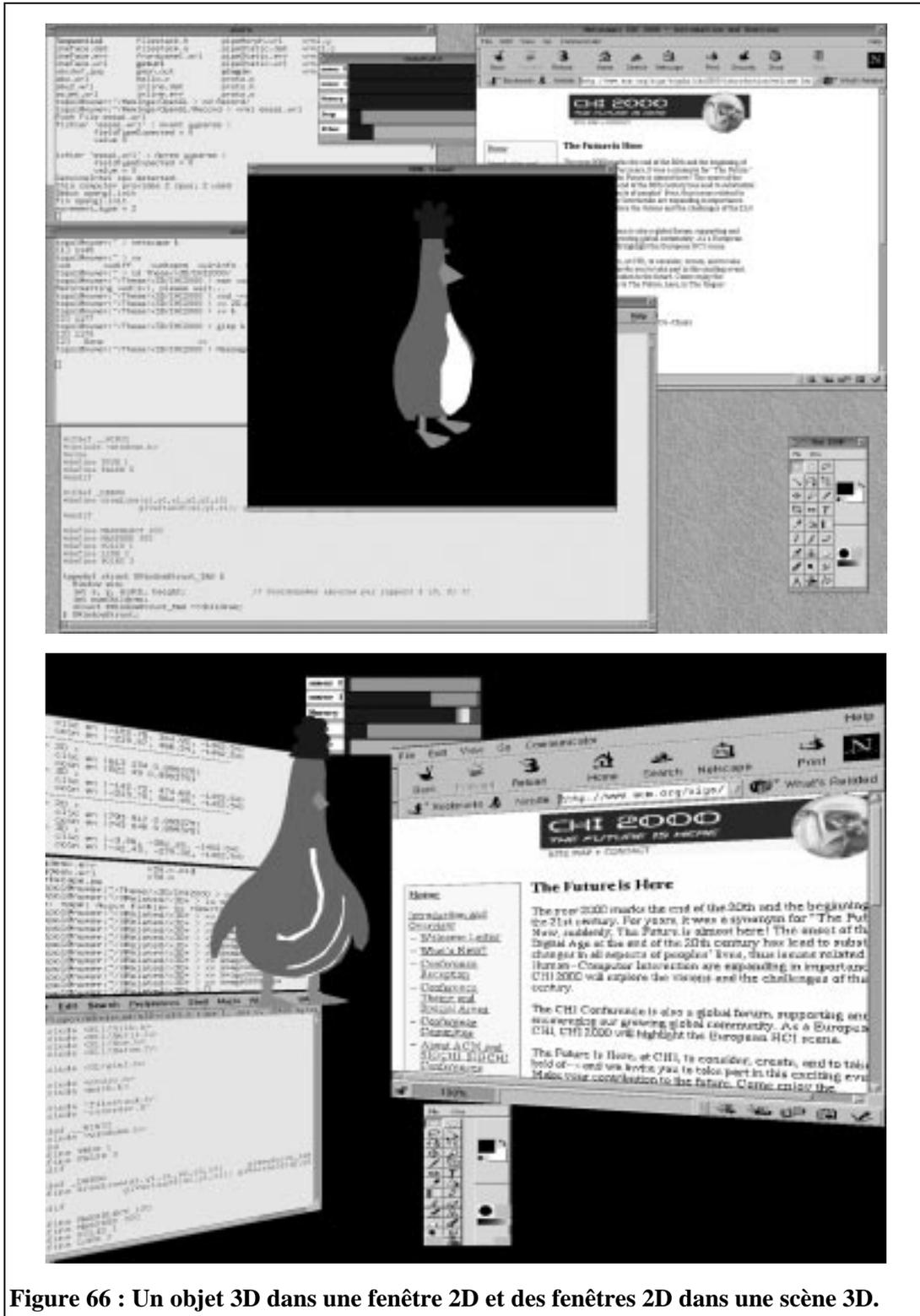


Figure 66 : Un objet 3D dans une fenêtre 2D et des fenêtres 2D dans une scène 3D.

D'autre part, la situation actuelle concernant la gestion des applications 3D est insatisfaisante. Les applications 3D sont en effet exécutées dans une fenêtre 2D et forcent l'utilisateur à changer constamment de métaphores de navigation lorsque qu'il passe d'une application avec une interface 2D à une application en 3D. Ce problème est constamment rencontré lorsque l'on visualise des scènes VRML. La Figure 66 illustre la différence entre une application 3D exécutée dans un système de fenêtrage 2D et des applications 2D accessibles depuis une scène 3D.

D'autres projets ont également eu pour tâche d'inclure des applications 2D dans un environnement 3D. Le gestionnaire de fenêtre MaW3D place les fenêtres dans un « tunnel » [LEA97]. Microsoft a également présenté un tel projet à la conférence CHI'2001, la *Task Gallery* [ROB2000]. La métaphore utilisée est celle de la galerie d'art. La ressemblance avec MaW3D est donc assez frappante puisque l'utilisateur est contraint à naviguer dans le couloir. Les fenêtres, dans ces deux tentatives, sont plaquée sur des murs. Nous pensons qu'une interface 3D ne doit pas être autant contrainte. Ces contraintes fortes n'autorisent pas l'inclusion d'objets 3D et rien (ni dans les articles, ni dans les copies d'écran) ne nous indique que c'est possible de le faire dans ces deux environnements. D'autre part, l'apport des rotations, aussi bien du point de vue sur la scène que des objets, est considérable pour l'organisation d'un espace 3D. Cependant, cela nécessite d'étudier les politiques et placements et les techniques de manipulations.

Le projet le plus proche du nôtre est sans conteste 3Dwm⁵⁰ (*3D window manager*) car il a été développé également sous Linux. Cependant, ce gestionnaire de fenêtre gère une seule et unique face texturée avec l'image complète du bureau 2D de XWindow. Contrairement à notre prototype, chaque fenêtre n'est donc pas orientable et/ou positionnable indépendamment. Elles peuvent être déplacée mais doivent rester dans le plan réduit de la face. Cette solution, malgré une esthétique très travaillée, n'apporte donc rien de plus que la 2D pour la gestion des fenêtres. Elle offre néanmoins la possibilité de manipuler dans un même environnement aussi bien des objets 2D que 3D.

Notre gestionnaire 3D a été développé sur XWindow car son architecture client/serveur permet de récupérer et de détourner l'usage des structures de données stockées par le serveur X. La première étape fut de récupérer les images des différentes fenêtres et de les inclure sur des faces OpenGL que nous avons ajoutées dans notre visualisateur de fichiers VRML. La récupération des images n'est pas un problème majeur. Nous avons implémenté exactement la même méthode que celle utilisée par les programme de capture d'écran sous XWindow (comme xwd). Cette méthode est également utilisée par le logiciel VNC (*Virtual Network Computing*) d'AT&T, l'émulateur d'écran inter-plates-formes.

La Figure 67 schématise les différentes opérations effectuées lors de l'initialisation de la scène et lors de l'interaction sur une fenêtre. Pour l'initialisation, l'arbre des fenêtres est récupéré par un appel à la fonction *XQueryTree*. Pour chacune des fenêtres filles de la fenêtre *root* (principale), nous récupérons ensuite sa taille. La fenêtre est ensuite redimensionnée de telle manière qu'elle ait une taille de $2^n * 2^m$, la taille de texture imposée sous OpenGL. L'image est ensuite récupérée et plaquée sur une face dont la taille est proportionnelle à l'image. Le sous-arbre des fenêtres de chacune des fenêtres de premier niveau est ensuite sauvegardé pour savoir, durant la phase d'interaction, sur laquelle l'utilisateur a agit. Il faut en effet savoir que sous XWindow, tous les éléments graphiques (*widgets*) sont des fenêtres.

Lorsque l'utilisateur effectue un clic de souris, on détermine en d'abord sur quel objet il a eu lieu. Si c'est une fenêtre, les coordonnées 3D du point cliqué sont calculées pour pouvoir déterminer les coordonnées 2D de l'endroit cliqué dans la fenêtre par rapport à son coin supérieur gauche. Le *widget* sur lequel a eu lieu ce clic de souris est ensuite extrait grâce à la sauvegarde de l'arbre des sous-fenêtres. Les événements « bouton appuyée » et « bouton relâchée » sont envoyés à ce *widget*. Les nouveaux *widgets* que cette action a pu créer sont stockés dans l'arbre des sous-fenêtres. Les

⁵⁰ A voir sur <http://www.3dwm.org>.

modifications d'apparence de la fenêtre suite à cette interaction sont reflétées automatiquement par le moteur OpenGL lorsqu'il est inoccupé. Toutes les fenêtres sont donc périodiquement remises à jour par le moteur. Ainsi, les animations automatiques sont également visibles.

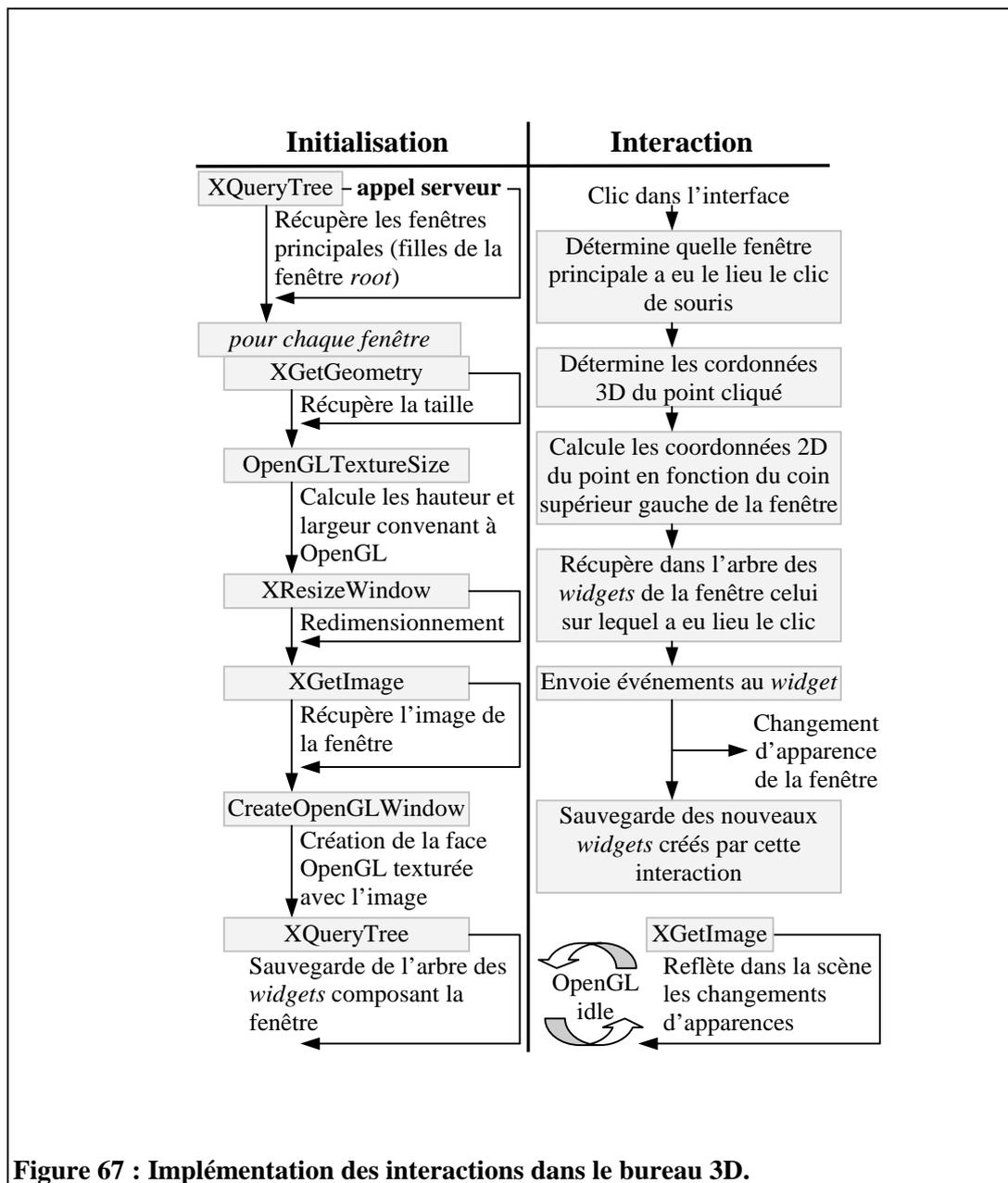


Figure 67 : Implémentation des interactions dans le bureau 3D.

Ce prototype répond à la même démarche que notre poste de lecture présenté précédemment, l'intégration des données 2D et 3D dans un même environnement où les interactions sont similaires sur les deux types de données. Cette solution peut être utilisée pour créer un poste de lecture. Il suffirait d'autoriser le rendu de scène VRML dans notre gestionnaire de fenêtre 3D. Dans ce cas, tout éditeur de texte pourra servir d'outil d'annotation à l'intérieur même de l'environnement 3D.

Cette application a été développée rapidement et un intérêt particulier devrait être apporté aux interactions sur les fenêtres elles-mêmes et non seulement avec leur contenu. Cela suppose d'étudier quelles interactions sont nécessaires. En ce qui concerne l'évolution de ce prototype, elle peut prendre deux directions différentes :

- vers un gestionnaire de fenêtre 3D utilisant le protocole X tel qu'il existe à l'heure actuelle (cela suppose que toutes les coordonnées 2D reçues du serveur soient converties en coordonnées 3D),
- vers un serveur X 3D où les coordonnées des fenêtres sont directement gérées en 3D.

Quelque soit la méthode choisie pour faire évoluer ce prototype, il sera nécessaire d'apporter une attention particulière au rendu des textes. Des techniques d'*antialiasing* devront être utilisées pour adoucir les contours des textes et des images. La prise en charge des autres événements devra être également étudiée. En particulier, les événements clavier qui permettent à l'utilisateur de saisir des données et donc ne pas être limité à la simple manipulation du pointeur.

5.2 Travaux futurs

Pour nos projets récents (le poste de lecture et le bureau 3D), le dénominateur commun est l'intégration d'éléments 2D dans une interface 3D. Nous sommes convaincus que la 3D s'imposera si les nombreuses applications 2D peuvent suivre cette évolution. Il nous semble, par ailleurs, que la 3D offre de nouvelles perspectives pour l'organisation d'un espace de travail et l'on se doit de l'appliquer également aux objets 2D.

Pouvoir profiter de l'organisation 3D des données 2D n'est cependant pas (et ne doit pas être) l'unique but. Bien évidemment, de nouvelles applications devraient bénéficier du pouvoir de représentation des métaphores 3D. Cela passe d'abord par l'étude des *widgets* 3D nécessaires. Comme pour les *widgets* disponibles via *Motif* ou *Athena*, les widgets 3D doivent être définis par leur apparence, leur comportement et les mécanismes permettant de les intégrer parmi d'autres éléments dont la structure n'est pas identique.

Les travaux présentés ne sont qu'à l'état de prototypes. Ces prototypes, ainsi que les études théoriques des interfaces 3D, nous ont néanmoins permis de révéler quelques faiblesses. La plupart de ces faiblesses sont issues du langage de description utilisé, dans notre cas VRML. Malheureusement, les mêmes lacunes existent pour la très grande majorité de ces langages. Nous ne remettons pas en cause l'aspect description graphique des scènes mais uniquement les possibilités d'interaction offertes. Par ailleurs, nous étudions également l'intégration d'un nœud sonore étendant les possibilités du nœud *Sound* proposé par VRML.

5.2.1 Comportements 3D

La seule manière de spécifier des comportements 3D évolués avec le langage VRML passe par l'ajout de scripts Java ou Javascript. A ce jour, il est bien difficile de trouver sur Internet des scènes VRML scriptées utilisées à des fins non anecdotiques, ou de démonstration. Le succès à large échelle de VRML repose sur une description des animations et des comportements aussi compréhensible que celle des objets. Cette simplification implique de réduire la place réservée aux scripts gérant les interactions. Nous proposons, pour cela, des nœuds de comportements applicables à tout objet. Nous optons également pour le remplacement des routages par des mécanismes simples de contrôle des liens entre nœuds. Nous différons sur ce point avec J.-F. Balaguer [BAL99] qui analyse fort bien les manques de VRML mais qui souhaite le faire évoluer vers un langage encore plus basé sur la programmation. C'est aussi le cas de l'architecture EMMA mêlant des objets 2D et 3D mais au prix d'une programmation complexe [MAR]. Ces propositions font évoluer VRML d'un langage descriptif vers un langage de programmation proche des concepts de Java3D. De notre côté, pour pallier l'inexistence d'un modèle haut niveau pour décrire les comportements nous l'augmentons sans pour autant dénaturer sa structure.

Pour l'interface de lecture, nous aurions souhaité des nœuds de comportement tels que les suivants :

- un nœud *Clickable* gérant les clics de souris et fournissant les trois apparences classiques : appuyée, relâchée et *roll-over*. Il permettrait de gérer l’affichage des étiquettes informatives et l’interaction menant à l’ouverture d’un livre.
- un nœud *Movable* signifiant que les interactions de l’utilisateur affectent ses nœuds descendants. Ce nœud permettrait de positionner et d’orienter les objets grâce à des interactions et non plus par navigation comme précédemment. En effet, pour déplacer une seule fenêtre dans le poste de lecture en VRML+Java, il faut punaiser toutes les fenêtres sauf celles que l’on souhaite déplacer.
- un mot clé *AFFECTS* pour éliminer la partie routage du fichier VRML. Lorsque le nœud de comportement ne modifie que ses nœuds descendants, ce mot clé n’est pas nécessaire. Il l’est pour intégrer directement dans le graphe de scène les routages d’événements vers un nœud non descendant (comme dans le code suivant).

Pour avoir une idée plus précise du code VRML obtenu, le pseudo-code ci-dessous décrit une fenêtre 3D avec les nœuds précédents. Cette nouvelle syntaxe remplace 450 lignes de Java servant à gérer les fenêtres et le fichier VRML est réduit à ≈125 lignes.

```

DEF Window3D Transform {
  children [                                     # Les nœuds constituant la fenêtre
    Shape { [...] }
    [...]
    Shape { [...] }
    DEF TitleBar Movable {
      # Les interactions autorisées
      canTranslate TRUE
      canRotate TRUE
      # Les événements émis par ce nœud
      translation AFFECTS Window3D.translation
      rotation AFFECTS Window3D.rotation
      children [...]                               # Les nœuds constituant la barre
    }
  ]
}

```

Nous pourrions utiliser ce langage VRML étendu pour prototyper des applications 3D, dans l’esprit des nombreux outils disponibles pour les environnements « plats ». Toutes les applications 3D devront également avoir accès à des primitives permettant :

- de spécifier le type d’intégration d’objets dans un objet conteneur,
- et de contrôler le placements des conteneurs et de leurs objets contenus.

Ces aspects ne sont pas propres à la 3D. En effet, parmi les classes AWT du langage Java, certaines gèrent le type de placement des éléments (les classes de type *Layout*) dans un conteneur et l’ajout d’objets dans ce conteneur se fait en fonction de la politique de placement défini. Définir ces différents paradigmes d’inclusion d’objets dans un conteneur n’est cependant pas une traduction directe de la 2D vers la 3D. L’ajout d’une troisième dimension complexifie la tâche mais le principe reste le même : les objets se réorganisent dans le conteneur lorsque l’on en ajoute un nouveau.

Ces politiques de placements, quelque soit le contexte dans lequel on se place (langage descriptif ou langage de programmation) permettront à un concepteur de réfléchir en terme de positionnements géographiques, relatifs à leur conteneur, plutôt qu’en terme de transformations. L’élimination, ou tout du moins l’abstraction, de ces aspects mathématique comme cela est fait dans Alice [CON94] rendra la description de scènes plus aisée.

5.2.2 Intégration du son spatialisé

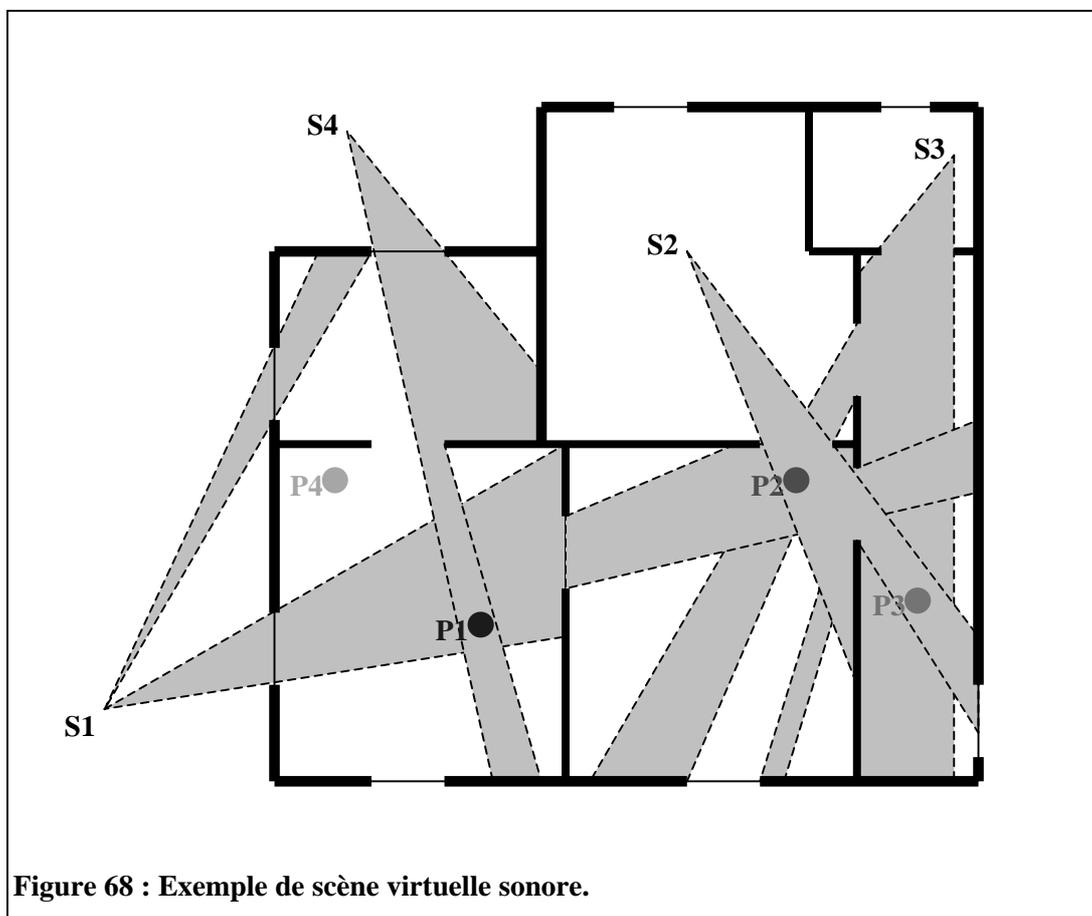
Le langage VRML souffre également de carences pour la gestion sonore. Alors qu'un fichier VRML pourrait servir pour la description d'environnement sonore 3D, cela est rendu impossible par l'absence d'attributs sonores des objets et par la pauvreté de la description des sources sonores.

Le rendu sonore d'un environnement 3D peut être fait de deux manières différentes : en respectant les propriétés du volume dans lequel l'utilisateur évolue ou en permettant un rendu qui n'a rien à voir avec la scène. Ce dernier est possible grâce à quelques variables globales qui définissent les propriétés sonores de la scène entière. Cette méthode permet donc de simuler l'acoustique d'une cathédrale alors que la scène décrit une minuscule pièce.

Le type de rendu sonore qui respecte le volume de la scène considérée peut être traité de deux façons différentes. La différence entre ces deux méthodes suivantes est très similaire avec la différence qui existe entre les méthodes du lancer de rayons et de la 3D temps-réel pour le rendu visuel d'une scène 3D :

- par le respect des lois physiques de réflexion et d'absorption des sons sur chacun des objets,
- par la description d'un « matériel sonore » attaché à chaque objet donnant son comportement simplifié face aux sons qui le percutent.

Cette deuxième méthode permettrait de spécifier le pourcentage de porosité d'un objet au lieu de donner ses attributs physiques. Il ne sera pas nécessaire comme dans la première de ces deux méthodes de gérer les réflexions sur les objets voisins, ni de connaître les angles d'arrivée des sons sur les faces. Ainsi, les temps nécessaires aux calculs de ce que l'utilisateur perçoit à une position donnée pourront être fait en temps-réel.



Dans la Figure 68, un exemple de ce que peut être une scène sonore est schématisé. Cette scène, vue du dessus, représente un appartement composé de différentes pièces. Les murs sont constitués d'un matériel sonore étanche ; c'est-à-dire qui ne laisse pas passer les sons. Les portes et les fenêtres par contre, laissent filtrer les bruits. Dans cette environnement 3D, sont ajoutées quatre sources sonores ponctuelles. Pour chacune d'elles, en fonction de l'étanchéité ou non des surfaces qu'elle frappe, nous avons schématisé par les surfaces grises les endroits dans l'appartement d'où elle peut être entendue. Le tableau suivant retranscrit, en fonction d'une des positions possibles de l'utilisateur schématisée dans la figure, les sources sonores qu'il entendra :

Position	Sources sonores audibles
P1	S1, S4
P2	S1, S2, S3
P3	S2, S3
P4	–

Dans cet exemple, on aurait pu considérer que les murs étaient poreux aux sons. Cependant, cela aurait complexifié la figure. De la même manière, on aurait pu considérer que seulement certains murs (les murs porteurs par exemple) étaient étanches. Enfin, les portes et les fenêtres pourraient filtrer les sons lorsqu'elles sont fermées et les laisser passer entièrement lorsqu'elles sont ouvertes. Cette souplesse, ajoutée au filtrage d'un son en fonction de la distance séparant un point de la source sonore, devrait permettre de spécifier des scènes sonores relativement réalistes.

On peut très bien imaginer d'autres comportements sonores pour les faces. En effet, le choix binaire (arrête/laisse passer les sons) n'est pas suffisant. L'unique possibilité de donner le pourcentage du volume du son qui peut traverser une face n'est pas non plus acceptable. Il faudrait des comportements plus souples, tels que la transformation simple des sons par des filtres prédéfinis ou que l'on décrirait.

MPEG4 permet, dans sa deuxième version, de gérer le rendu 3D sonore. Cependant, aucune documentation n'est claire sur ses véritables possibilités. Il est même impossible de savoir s'il existe une application qui gère cet aspect. Nous proposons donc d'utiliser VRML comme langage support car la description des scènes est claire. D'autre part, l'ajout de nœuds et d'attributs est facile et puisque nous disposons d'un moteur de rendu visuel, nous pouvons expérimenter ces nouveaux éléments. Et puisque MPEG4 reprend les nœuds VRML pour la description des scènes, il sera relativement aisé de l'utiliser lorsque la norme sera complètement disponible et compréhensible.

En VRML, nous pourrions prendre en charge toutes les méthodes de rendu sonore précédentes. Mais le but de VRML est de permettre une interaction temps-réel. Pour ne pas l'éloigner de ce rôle, nous proposons donc de gérer les « matériaux sonores ». Les sources sonores seront décrites de la même manière (nœud *Sound*) mais les « mappings sonores » des objets permettront de mixer et filtrer les différents sons de manière à obtenir une scène plus riche et acceptable du point de vue sonore.

CONCLUSION

Notre étude des interfaces 3D regroupe différentes facettes qui bien que très différentes sont toutefois indissociables. En premier lieu, une connaissance relativement complète des étapes menant d'une description de scène à son affichage est nécessaire. Pour l'aspect description de la scène (si celle-ci n'est pas créée entièrement par un programme mais plutôt lue dans un fichier support), cela suppose la maîtrise du langage utilisé (lorsque l'on n'utilise pas par un outil spécialisé). Dans notre cas, nous avons choisi VRML97 et nous avons poussé son étude jusqu'à l'écriture de l'interpréteur qui traduit les expressions reconnues en appels de fonctions d'une API 3D. L'aspect mathématique des différentes étapes du pipeline est également à prendre en compte sérieusement. En particulier, les transformations des objets composant une scène sont données par le concepteur ou le programmeur ; aussi bien avec un langage de programmation s'appuyant sur une API 3D qu'avec un langage de description comme VRML. Que ce soit pour la traduction des expressions VRML ou pour développer un programme générant lui-même les commandes 3D, la connaissance d'une bibliothèque de fonctions 3D est nécessaire pour accéder à une accélération matérielle. Nous avons, par conséquent, présenté les principales API 3D (OpenGL, Direct3D et Java3D) en donnant leurs avantages et inconvénients ainsi qu'un squelette de code pour les utiliser.

Les aspects précédents gravitent exclusivement autour de la conception des interfaces. Or, pour toute application exploitant une interface 3D, les interactions sont également d'une extrême importance. Alors qu'une interface 3D fournit à l'utilisateur une nouvelle manière d'organiser et de voir les informations, les interactions lui offrent les outils pour la manipuler, y naviguer et par conséquent de mieux appréhender les informations qu'elle renferme. Trouver des méthodes de manipulation et de navigation fluides et intuitives est aussi important que de trouver les métaphores visuelles adéquates pour révéler les informations. Les deux (interaction-navigation et représentation) sont indissociables. Elles aident à part égale à stimuler le processeur sensoriel humain que nous utilisons pour percevoir, sentir les choses. Or, il est plus rapide pour assimiler les informations que le processeur cognitif qui nous sert à analyser les informations lorsqu'elles ne sont pas immédiatement reconnues. Une interface 3D nous offre cette propriété quasi exclusive de pouvoir utiliser nos expériences réelles sur des objets virtuels sous réserve que des interactions particulièrement naturelles soient également implémentées.

De cette synthèse, deux constats peuvent d'ores et déjà être tirés. Premièrement, les interfaces 3D ont bénéficié de recherches poussées depuis une dizaine d'années alors que les interactions ne sont étudiées que depuis récemment. En d'autres mots, de nombreuses équipes de recherche ont démontré l'utilité de la visualisation 3D sans s'intéresser (ou si peu) à son « utilisabilité ». Il en résulte un terrible décalage entre ce qu'il est possible de représenter grâce à une interface 3D et ce que l'utilisateur novice peut appréhender par une manipulation hasardeuse à travers des dispositifs inadéquats. Les outils de création, de manipulation, de visualisation ainsi que les langages de description ou de programmation facilitent le développement et le déploiement des interfaces 3D. Cependant, les périphériques d'entrée/sortie idéaux (tout du moins les plus à même de faciliter la manipulation) pour la 3D ne sont pas, d'après nous, encore clairement définis.

Le deuxième constat vient de l'expérience acquise lors du développement de notre *parser* VRML. La traduction des expressions VRML en un ensemble de fonctions 3D est d'une relative facilité comparée à la difficulté pour proposer et programmer les outils d'interaction. Mettre en œuvre un visualisateur de scènes nécessite un temps certain mais ne pose pas véritablement de problèmes. Transformer ce visualisateur en un navigateur, c'est-à-dire lui adjoindre ces outils de manipulation de la caméra et des objets, représente l'aspect le plus intéressant mais également celui où l'on piétine le plus. Pour chaque dispositif d'interaction considéré, nous ne programmons qu'à la suite d'intuitions sur les différentes manières de les utiliser. Et celles-ci peuvent s'avérer être fausses lors des premiers essais de manipulation. Pour exemple, interpréter les mouvements d'une souris 3D de type

SpaceMouse pour que l'utilisateur dirige son point de vue sur la scène est beaucoup moins pratique et naturel que pour le déplacement et l'orientation d'un objet ou d'une scène.

Bien évidemment, quelques sujets relatifs à la 3D temps-réel ne sont pas ou pas complètement couverts dans ce rapport. Ce seront principalement les trois lacunes suivantes que nous essaierons de couvrir dans nos travaux futurs.

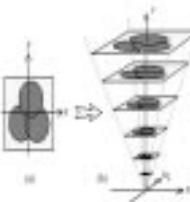
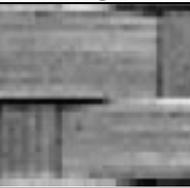
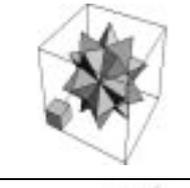
- L'étude plus en profondeur des interactions avec une scène 3D. Jusqu'alors, nous nous sommes contentés de voir les méthodes présentées sans essayer de proposer les nôtres. Les interactions étant dépendantes des informations présentées dans l'interface 3D, nous les étudierons plus particulièrement dans les deux contextes présentés dans le chapitre 5.2 page 143 : pour accéder à une bibliothèque numérique et pour un bureau 3D.

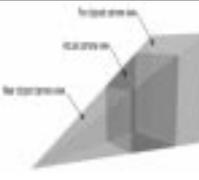
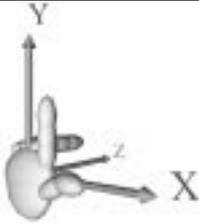
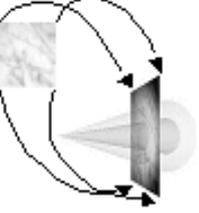
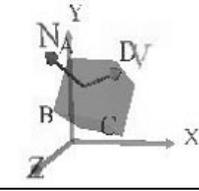
Dans le premier cas, il s'agira de déterminer les actions possibles sur un livre et/ou une collection de livres. Comment faciliter, aider la lecture croisée de plusieurs ouvrages ? Comment organiser son espace de travail comportant une collection d'ouvrages ? Quels outils offrir aux utilisateurs pour consulter les livres, les annoter, les classer, les ranger ?

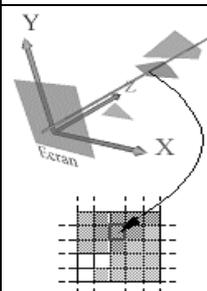
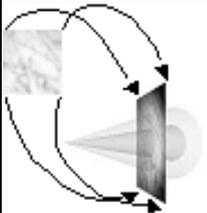
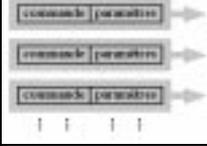
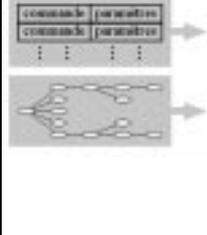
Pour le bureau 3D, les mêmes problèmes d'organisation de l'environnement de travail se posent. L'utilisation de l'espace et les outils de récupération des informations dans celui-ci semble en effet une constante lors de la conception d'une interface 3D ; quel que soit son rôle. Ici, ce sont plus particulièrement des problèmes d'agencement des fenêtres qui se posent. Comment les ranger ? les mettre de côté ? les retrouver ?

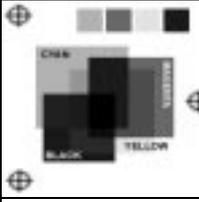
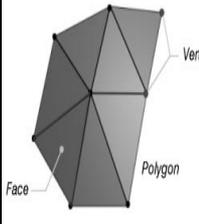
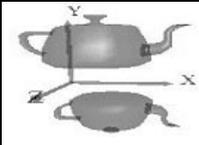
- L'étude des interactions et des animations dans VRML qui sont les maillons faibles du langage. Le système d'événements et les scripts nous paraissent en effet compliqué et peu fiable (car non-déterministe). Nous souhaiterions proposer une extension au langage VRML pour décrire les comportements des éléments d'une scène d'une manière aussi simple et puissante que sont décrits les objets graphiques. Cela va dans le sens de ce que nous disons précédemment : les comportements bénéficient de moins d'études et leur intégration s'en trouve retardée.
- Nous souhaitons également poursuivre nos travaux par la prise de mesures de performance. L'idée de calculer *a priori* la complexité d'une scène 3D est particulièrement attirante. Elle permettrait de connaître, en fonction des possibilités de la machine utilisée, quelles options de rendu devront être supprimées pour qu'une scène puisse être visualisée. Nous voyons dans cette étude une manière de rendre toute scène exploitable sur toute machine sans utiliser d'algorithmes adaptatifs qui nécessite une part non négligeable des calculs.

GLOSSAIRE

2.5D	La "deux D et demie" est une astuce pour optimisée la 3D temps-réel en faisant croire à l'observateur qu'il voit véritablement en trois dimensions. La scène entière (ou tous les objets de la scène) est construite à différentes échelles à partir de primitives graphiques 2D. Pour l'affichage, l'échelle est choisie en fonction du niveau de zoom désiré par l'utilisateur. Elle peut également être affichée en perspective pour que les objets donnent l'impression d'être des polygones.	
Aliasing	Lorsque les contours des objets paraissent dentelés. Le phénomène est surtout net pour les lignes diagonales. L'aliasing est visible lorsqu'un moteur effectue le rendu d'un objet sur un écran avec une mauvaise résolution, ne pouvant pas afficher les détails correctement.	
Alpha Channel	Une quatrième composante à rajouter aux trois couleurs primaires servant à définir une propriété supplémentaire pour la couleur d'un objet. Il peut s'agir par exemple d'un coefficient de transparence à appliquer au matériel.	
Ancêtre	Tout nœud dans un graphe qui se trouve au-dessus du nœud considéré.	
Anti-aliasing	La couleur des points proches des contours est calculée en fonction de la couleur des points voisins. Les contours sont alors plus doux et donc apparaissent moins dentelés.	
Bit depth	La profondeur de couleur est le nombre de bits nécessaires pour coder une couleur. Les bits incluent les trois composantes RGB et également le canal Alpha. Le nombre de couleur pour une profondeur de n bits est de 2^n .	
Bitmap	Littéralement traduit par « plan de bits », un <i>bitmap</i> considéré habituellement comme un mode de codage des images par la couleur de chacun de ses points (par opposition au mode vectoriel), ce terme est également couramment utilisé en 3D comme synonyme de « texture ».	
Boite englobante	Une approximation du volume d'un objet. Les boites englobantes sont utilisées par le moteur pour déterminer si l'objet est visible et s'il doit donc parcourir sa liste de polygones pour les afficher. Elles sont également utiles pour déterminer calculer les collisions.	
BSP	<i>Binary Space Partition</i> . Un arbre BSP divise l'espace 3D en plusieurs plans 2D pour accélérer les tris en profondeur. Il est utilisé par certains moteurs pour effectuer des optimisations dans le temps du rendu (<i>culling</i>) et pour implémenter les collisions.	
Caméra	La caméra est l'œil de l'observateur, l'endroit depuis laquelle la scène est vue.	

	scène est vue.	
Clipping	L'étape du pipeline 3D qui détermine, en fonction de la position et de l'orientation de la caméra, si un polygone est visible. Cette étape accélère de façon non négligeable le rendu d'une scène puisque certains polygones ne seront pas traités par le reste du pipeline.	
Collision	Détermination des intersections entre les trajectoires de deux objets ou entre la trajectoire d'un objet et de celle de la camera.	
Coordonnées	Un triplet de valeurs donnant au moteur la position des éléments dans l'espace. Dans le repère cartésien utilisé par un moteur 3D, les trois coordonnées sont communément appelées X, Y et Z.	
Coordonnées de texture	Ce sont des valeurs qui servent à positionner une texture sur un polygone. Un moteur utilise trois coordonnées de texture : U pour sa largeur, V pour sa hauteur et W pour sa profondeur dans le cas où l'on utilise une texture 3D (procédurale). Les valeurs de ces coordonnées sont généralement comprises entre 0 et 1. Pour chacun des sommets du polygone une coordonnée de texture donne le point de la texture lui correspondant.	
Concave	Si deux points à l'intérieur d'un polygone peuvent être reliés par un segment passant à l'extérieur de la face alors le polygone est concave. Ces types de polygone sont à proscrire pour les moteurs 3D car leur coloriage est un problème.	
Coplanaire	Quand plusieurs choses sont dans le même plan 2D. Les polygones traités par un moteur 3D doivent généralement avoir des sommets coplanaires.	
Convexe	Le type de polygone qui s'oppose à concave. Tout segment reliant deux points à l'intérieur du polygone reste entièrement dans la surface définie par le polygone.	
Culling	Le culling est un ensemble de techniques pour accélérer le rendu du pipeline 3D. La plus connue d'entre elles est le <i>Backface Culling</i> permettant de déterminer quels polygones sont vus par leur face arrière. Généralement, seule la face avant des polygones, déterminée par leur normale, doit être affichée.	
Double Buffer	Alors qu'une première zone mémoire est utilisée pour stocker l'image à afficher à l'écran, une seconde sert à calculer l'image suivante. Les deux <i>buffers</i> échangent leur rôle une fois l'image calculée.	
Fill Rate	Le nombre de textures ou de points qu'un moteur peut tracer en un temps donné. L'unité de mesure est le million de pixels par seconde (MPS).	
Fils (nœud)	Un nœud dans un arbre qui est rattaché à un autre nœud (son père).	

FPS	<i>Frames Per Second</i> . Cette unité mesure le nombre d'images affichées en une seconde. L'œil humain n'assimile que 24 images par secondes. Si le moteur passe en deçà de cette limite l'utilisateur le percevra.	
Frame Buffer	C'est un tableau utilisé pour stocker une image avant de l'afficher. Lorsque le rendu est terminé, l'image est affichée et l'image suivante est calculée. Généralement, une carte graphique gère deux <i>Frame Buffers</i> : un pour calculer la prochaine image (on le nomme le <i>back buffer</i>) et un pour afficher l'image courante sur le moniteur (le <i>front buffer</i>). Cette technique est connue sous le nom de <i>double buffering</i> . Les cartes les plus perfectionnées peuvent gérer jusqu'à quatre <i>buffers</i> .	
LOD	<i>Level Of Detail</i> . Plusieurs apparences, avec plus ou moins de détails, sont données par polygone. L'apparences d'un objet est choisie en fonction de sa distance à la caméra. Cette technique de <i>culling</i> est utilisée pour réduire le temps de calcul d'une scène.	
Mapping	Ce terme peut signifier plusieurs choses : <ul style="list-style-type: none"> - le processus de création d'une texture, - l'application d'une texture sur un polygone grâce à ses coordonnées de texture, - l'assignation des touches du clavier à des fonctions de l'application. 	
Matériel	Un ensemble de paramètres qui fournit les propriétés d'apparence d'un objet 3D : couleur, réflexion, texture...	
MIP mapping	Une méthode pour le filtrage de texture. Ce terme vient du latin <i>Multum In Pano</i> qui signifie « beaucoup de choses dans un petit espace ». Le <i>MIP mapping</i> permet de créer plusieurs textures à différentes échelles. En fonction de la distance à l'objet, la texture ayant la taille adéquate sera choisie par le moteur.	
Mode Immédiat	Il caractérise la transmission immédiate des commandes 3D entre une application et une API 3D. Ce mode est particulièrement utilisé pour les scènes interactives et/ou animées nécessitant une mise à jour périodique de leur contenu.	
Mode Retenu	Les fonctions 3D sont stockées (retenues) dans une structure au lieu d'être envoyées directement (une à une) à l'API 3D pour être tracées. L'application transmet cette structure pour calculer et afficher la scène au lieu de transmettre individuellement chacune des commandes sauvegardées. La structure peut aussi bien être un tableau (comme pour OpenGL) qu'un graphe de scène (pour Java3D, Direct3D ou Fahrenheit).	
Modèle de couleur additif	Les couleurs rouge, vert et bleu (RVB, RGB - <i>Red Green Blue</i>) sont les couleurs primaires. La couleur blanche est obtenue en mélangeant les trois couleurs comme pour la lumière. C'est par addition qu'on obtient donc le blanc. Ce modèle est utilisé en informatique pour représenter les couleurs.	

Modèle de couleur soustractif	Les couleurs primaires sont magenta, jaune, cyan et noire (CMYK – <i>Cyan Magenta Yellow black</i>).	
Morphing	Technique permettant de transformer par animation une texture ou une forme géométrique en une autre.	
Multi texturing	Ou <i>texture blending</i> . Permet de combiner plusieurs textures pour en former une nouvelle.	
Pipeline 3D	Ensemble des opérations menant de la description dans l'espace d'une scène et de son point d'observation jusqu'à sa représentation 2D sous forme d'image dans une fenêtre de visualisation.	
Pixel	Abréviation de <i>picture element</i> . Il y a deux sens donné à ce terme : les points constituant une texture et les points qui sont calculés par être affichés à l'écran par le moteur. Un pixel est composé d'une position (X,Y) et d'une couleur RGB.	
Polygone	Un ensemble de sommets (<i>vertices</i>) qui définissent une forme géométrique. La plupart des moteurs 3D supportent des polygones comptant de multiples sommets. Ces sommets doivent être donnés en respectant l'ordre trigonométrique. Cette convention est nécessaire pour calculer la normale qui sert à déterminer la face visible du polygone. Les différents polygones sont convertis en triangles par le pipeline 3D afin d'éliminer les problèmes liés aux calculs des polygones convexes ou dont les sommets ne sont pas coplanaires.	
Rendering	La transformation par le moteur 3D temps-réel des données décrivant une scène 3D en une image 2D affichable sur l'écran. Tout ce processus pris en charge par une série de calculs regroupés dans le pipeline 3D.	
Texel	Les texels (<i>textures elements</i>) sont les transformés des pixels d'une texture avant leur application sur un polygone. La distinction entre les deux termes <i>pixel</i> et <i>texel</i> est importante pour comprendre l'évolution de la texture. Les pixels de la texture sont transformés en texels avant d'être appliqués à un polygone. Ces texels sont ensuite eux-mêmes transformés en pixels sur l'écran.	
Transformation	Une opération effectuée par le moteur pour positionner, orienter ou changer l'échelle d'un objet 3D. Les transformations associées sont la translation, la rotation ou l'homothétie.	
Vertex	Un point 3D est défini par sa position par rapport à l'origine du repère cartésien. Il n'a pas de réelle utilité s'il n'est pas relié à une primitive graphique. La liste des <i>vertices</i> dans un polygone (un <i>vertex</i> - des <i>vertices</i>) sont les sommets du polygone.	

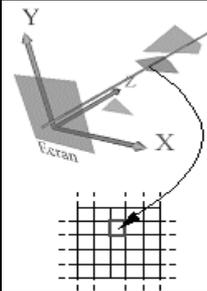
Z-Buffering	Un algorithme permettant de savoir quelles parties d'un objet sont visibles ou cachées par un autre. Les informations de profondeur de chacun des points transformés d'un polygone sont stockées dans le Z-Buffer. Pour un point de l'écran donné, ce sera la couleur du point de l'objet avec la plus petite coordonnées Z qui sera affichée.	
--------------------	--	---

TABLE DES FIGURES

FIGURE 1 : EXEMPLE D'UTILISATION DE LA 3D – LA MOLECULE DE CHLOROPHYLLE A.	10
FIGURE 2 : EXEMPLE D'UTILISATION DE LA 3D – JUPITER ET SES SATELLITES.....	11
FIGURE 3 : EXEMPLE D'UTILISATION DE LA 3D – SCENE VRML DU PROJET DE J.-M. LE GALLIC.....	11
FIGURE 4 : COMPARAISON DES IMAGES OBTENUES PAR LANCER DE RAYONS ET PAR FACETTAGE.	12
FIGURE 5 : IMAGES OBTENUES PAR LES METHODES DU LANCER DE RAYONS ET DE RADIOSITE.....	13
FIGURE 6 : SPHERES DESSINEES AVEC LES MODES DE COLORIAGE A PLAT (<i>FLAT</i>) ET GOURAUD.	14
FIGURE 7 : PRINCIPE DU LANCER DE RAYONS.	15
FIGURE 8 : LES DIFFERENTES ETAPES DU PIPELINE 3D.	17
FIGURE 9 : LES DEUX REPERES UTILISES EN 3D.	18
FIGURE 10 : LES TRANSFORMATIONS 3D.....	19
FIGURE 11 : DETERMINATIONS DU VECTEUR NORMAL A UNE FACE ET TEST DE VISIBILITE.....	20
FIGURE 12 : CALCUL DE LA PROJECTION.	21
FIGURE 13 : MODELE DE COLORIAGE GOURAUD ET PLACAGE DE TEXTURE.	21
FIGURE 14 : PRINCIPES DU <i>Z-BUFFER</i> ET DU <i>FRAME-BUFFER</i>	22
FIGURE 15 : DONNEES UTILISEES A CHAQUE ETAPE DU PIPELINE 3D ET <i>CULLING</i>	23
FIGURE 16 : PRINCIPE DES SPHERES ENGLOBANTES.	25
FIGURE 17 : RESULTATS OBTENUS PAR REDUCTION DU NOMBRE DE POLYGONES.....	26
FIGURE 18 : LES DIFFERENTES COUCHES D'UNE ARCHITECTURE 3D.	27
FIGURE 19 : FONCTIONS ACCELEREES PAR LES TROIS GENERATIONS DE CARTE GRAPHIQUE 3D.....	29
FIGURE 20 : PROCESSUS DE CREATION ET D'EXECUTION D'UNE APPLICATIONS 3D.	31
FIGURE 21 : L'ARCHITECTURE DRI DE XFREE86 4.0.	33
FIGURE 22 : PERFORMANCES 3D COMPAREES AVEC LE PROCESSEUR GRAPHIQUE GeFORCE.....	34
FIGURE 23 : POSITION DES API DANS LES COUCHES SYSTEMES WINDOWS.	37
FIGURE 24 : ARCHITECTURE LOGICIELLE ET PIPELINE GRAPHIQUE DE QUICKDRAW 3D.	39
FIGURE 25 : ARCHITECTURE LOGICIELLE ET PIPELINE GRAPHIQUE D'OPENGL.....	41
FIGURE 26 : POSITION DES DIFFERENTES LIBRAIRIES OPENGL.	42
FIGURE 27 : ARCHITECTURE LOGICIELLE ET PIPELINE GRAPHIQUE DE DIRECT3D.	50
FIGURE 28 : EXEMPLE D'UTILISATION DES <i>FRAMES</i> SOUS DIRECT3D.	52
FIGURE 29 : STRUCTURES NECESSAIRES A LA CONSTRUCTION D'UNE FACE SOUS DIRECT3D.....	53
FIGURE 30 : STRUCTURE DU GRAPHE DE SCENE JAVA3D.	59
FIGURE 31 : OBJETS ENTRANTS DANS LA CONSTRUCTION D'UNE SCENE AVEC JAVA3D.	61
FIGURE 32 : EXEMPLE SIMPLE DE GRAPHE JAVA3D.	62
FIGURE 33 : UTILISATION MEMOIRE POUR LES TEXTURES EN JAVA 2D ET JAVA 3D.	63
FIGURE 34 : L'ARCHITECTURE DE FAHRENHEIT.	64
FIGURE 35 : CHAINES DE PRODUCTION ET DE VISUALISATION D'UNE SCENE 3D.....	67
FIGURE 36 : DEUX TYPES D'OUTILS (SOFTIMAGE3D EN HAUT ET ISB EN BAS).	68
FIGURE 37 : LES ETAPES MENANT AU LANCEMENT D'UN <i>PLUGIN</i>	72
FIGURE 38 : LES DIFFERENTS SYSTEMES D'UN CO-NAVIGATEUR VRML.	75
FIGURE 39 : ARCHITECTURE SIMPLIFIEE DE MPEG4.....	82
FIGURE 40 : LES DIFFERENTES ETAPES DU CHARGEMENT D'UNE SCENE METASTREAM.....	84
FIGURE 41 : EXEMPLE DE RENDU D'UN PROTOTYPE EXTERNE.....	94
FIGURE 42 : EXEMPLE DE GRAPHE DE SCENE VRML.....	95
FIGURE 43 : FONCTIONS D'UN ANALYSEUR LEXICAL.	98
FIGURE 44 : LES DIFFERENTES ETAPES DU TRADUCTEUR VRML.	100
FIGURE 45 : LES ETAPES DE COMPILATION D'UN PARSEUR UTILISANT LES OUTILS LEX ET YACC.....	101
FIGURE 46 : LA CHAINE DE VISUALISATION D'UN FICHIER VRML.	105
FIGURE 47 : EXEMPLE D'AFFICHAGE DU GRAPHE DES NŒUDS D'UN FICHIER VRML.....	106
FIGURE 48 : ETAPES POUR INCLURE UN PROTOTYPE EXTERNE DANS UN FICHIER PRINCIPAL.	110
FIGURE 49 : CONTROLE DU RENDU.	111

FIGURE 50 : CONTROLE DU COLORIAGE.	112
FIGURE 51 : CONTROLE DE LA LUMIERE.	112
FIGURE 52 : OBTENTION DES DIFFERENTS MOUVEMENTS AVEC <i>UNICAM</i>	116
FIGURE 53 : EXEMPLE DE WIMS (<i>WINDOWS IN MINIATURE</i>) DANS NOTRE NAVIGATEUR VRML.	119
FIGURE 54 : EXEMPLE DE ZOOM DANS NOTRE NAVIGATEUR VRML.	120
FIGURE 55 : ARCHITECTURE DE L'ENVIRONNEMENT <i>LIVECONNECT</i> DE NETSCAPE.	121
FIGURE 56 : PRINCIPE DE GESTION DES ANIMATIONS.	124
FIGURE 57 : APPEL DE CODE NATIF DANS UN PROGRAMME JAVA.	125
FIGURE 58 : SIMULATEUR URBAIN : LES DIFFERENTS MODULES.	128
FIGURE 59 : SIMULATEUR URBAIN : SAISIE DU TERRAIN.	129
FIGURE 60 : SIMULATEUR URBAIN : ATTRIBUTS DU BATI.	130
FIGURE 61 : CLASSEMENT ET PARCOURS POSSIBLES POUR UNE RECHERCHE SUR 3 CRITERES.	133
FIGURE 62 : INTERFACES EN "ATOMIUM" ET EN CYLINDRE.	134
FIGURE 63 : PLACEMENT DES GROUPES DE LIVRES DANS UNE SPHERE.	135
FIGURE 64 : LES INTERFACES DE CONSULTATION 2D ET 3D.	137
FIGURE 65 : ARCHITECTURE DU POSTE DE LECTURE.	138
FIGURE 66 : UN OBJET 3D DANS UNE FENETRE 2D ET DES FENETRES 2D DANS UNE SCENE 3D.	140
FIGURE 67 : IMPLEMENTATION DES INTERACTIONS DANS LE BUREAU 3D.	142
FIGURE 68 : EXEMPLE DE SCENE VIRTUELLE SONORE.	145

REFERENCES

Ouvrages généraux consultés :

[BRE88] M. Bret. Images de synthèses – méthodes et algorithmes pour la réalisation d'images numériques. Dunod, 1988.

[COU95] J.-P. Couwenbergh. La Synthèse d'images. Marabout, 1995.

[FOL90] J. D. Foley, A. Van Dam, S. K. Feiner, J. F. Hughes. *Computer Graphics – principes and practice*. Addison-Wesley, New York, 1990.

[WAT93] A. Watt. *3D Computer Graphics*. Addison-Wesley, New York, 1993.

Thèses - Rapports :

[DOO2000] H. Doornbos. Le MMX d'Intel et le 3Dnow! D'AMD : comparaison des instructions et des parties spécifiques des processeurs des deux fabricants. Oral probatoire C.N.A.M. soutenu en Janvier 2000.

[DUM99] C. Dumas. Un modèle d'interaction 3D : Interactions homme-machine et homme-machine-homme dans les interfaces 3D pour le TCAO synchrone. Thèse soutenue le 1/10/99 à l'Université des Sciences et Technologies de Lille.

[LEG99] J.-M. Le Gallic. La visualisation des données géographiques – L'approche VRML - Java. Mémoire d'ingénieur C.N.A.M., Juin 1999.

[TOP98] A. Topol. Interfaces 3D pour les bibliothèques numériques. Mémoire de D.E.A. Médias et multimédia du C.N.A.M., Septembre 1998.

Articles cités :

[BAE95] R. Baecker, J. Grudin, W. Buxton, S. Greenberg. *Readings in Human-Computer Interaction : Toward the Year 2000*. Morgan-Kaufmann, 2nd edition, 1995.

[BAL99] J.-F. Balaguer. Less Is More : The Power of Simplicity. In *proc. of VRML'99*, February 23-26, 1999.

[BED94] B. B. Bederson. Pad++: A Zoomable Graphical Interface. In *Proceedings of CHI'94*.

[BED96] B. B. Bederson et al. A zooming web browser demonstration. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'96)*, Seattle, Washington, November 6-8, 1996, Demo Program, p. 1-2.

[BIB97] M. Bibes, E. Fléty. *Classification des Capteurs*. Pédagogie. IRCAM, 1997.

- [BIE93] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, T. D. DeRose. Toolglass and Magic Lenses : The See-Through Interface. In *Proceedings of Siggraph'93*, Anaheim, CA, August, Computer Graphics Annual Conference Series, ACM, New York, 1993, pp. 73-80.
- [BUR93] G. Burdea, Ph. Coiffet. *Réalité virtuelle*. Hermès, 1993.
- [BUX86] W. Buxton, B. A. Myers. A Study in Two-Handed Input. In *Proceedings of CHI '86*, Boston, MA, April 13-17), ACM, New York, (1986), pp. 321-326.
- [CAR91] S. K. Card, G. G. Robertson, J. D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of ACM Human Factors in Computing Systems Conference (CHI'91)*, pp. 181-188.
- [CAR96] S. Card, G. Robertson, W. York. The WebBook and the Web Forager : An Information Workspace for the World-Wide-Web. In *Proceedings of ACM SIGCHI CHI'96*. Vancouver, April 13-18, 1996.
- [CEL99] A. Celentano. Structuring and presenting WWW documents in a VR metaphore. *5ème conférence internationale Hypertextes, Hypermédias & Internet : Réalisations, Outils & Méthodes*. Univ. Paris VIII, Septembre 1999.
- [CHE88] M. Chen. A Study in Interactive 3D Rotation Using 2D Control Devices. *ACM Computer Graphics*, 22(4), pp. 121-127, août 1988.
- [CON94] Conway M., Pausch R., Gossweiler R., Burnette T. Alice : A Rapid Prototyping System for Building Virtual Environments. In *Proceedings of ACM CHI '94*, Boston, Massachusetts, April 1994, pp. 295-296.
- [CUB98] P. Cubaud, C. Thiria, A. Topol. Experimenting a 3D Interface for the access to a Digital Library. In *Proceedings of the third conference on Digital Libraries (DL'98)*. Pittsburg, June 23-26, 1998.
- [CUB2000] P. Cubaud, A. Topol, D. Vodislav. Les limites de VRML pour les comportements interactifs : étude de cas. In *Proceedings ERGO-IHM'2000*, Biarritz, France, Octobre, 2000.
- [CUB2001] P. Cubaud, A. Topol. A VRML-based user interface for an online digitalized antiquarian collection. In *Proceedings ACM Web3D'2001*, Paderborn, Germany, February, 2001.
- [EMM90] M. Emmerik. A Direct Manipulation Technique for Specifying 3D Object Transformations with a 2D Input Device. *Journal of the Yugoslav Committee ETAN*, 31(1-2), pp. 95-99, 1990.
- [FLE97] E. Fléty. *Sonars à Ultrason*. Rapport de stage, IRCAM, 1997.
- [FUC99] P. Fuchs. Les interfaces de la réalité virtuelle. *Les journées de Montpellier*, 1999.
- [FUR86] G. W. Furnas. Generalized fisheye views. In *ACM CHI '86*, pages 16--34, 1986.
- [GER95] N. Gershon, S. G. Eick. Visualization's New Tack : Making Sense of Information. *IEEE Spectrum*, November 1995, pp. 38-56.
- [KIP97] N. Kipp et al. *A Virtual Digital Library Interface with a Spacial Metaphore*. Dept. of Computer Science, Virginia Tech. December 1997.
- [LAM95] J. Lamping, R. Rao, P. Pirolli. A Focus + Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In *CHI'95. Proc. ACM Conf. On Human Factors in Computing Systems*. ACM Press, New York, 1995, pp. 401-408.

- [LEA97] G. Leach, G. Al-Qaimari., M. Grieve, N. Jinks, C. McKay. Elements of the Graphical User Interface. *In Proceedings of INTERACT'97*, Sydney, Australia, July 1997.
- [LES97] M. A. LESK. *Practical Digital Libraries (Books, Bytes and Bucks)*, Morgan Kaufmann Publishers, San Francisco, 1997.
- [MAC91] J. D. Mackinlay, G. G. Robertson, S. K. Card. The perspective wall : detail and context smoothly integrated. *In Proceedings of ACM CHI'91. Conference on Human Factors in Computing Systems*, New Orleans, Louisiana, April 28-May 2. ACM Press, New York, 1991, pp. 401-408.
- [MAR] Marrin, C. *Beyond VRML*. <http://www.marrin.com/vrml/private/EmmaWhitePaper.html>
- [MEN98] A. Mengel. *Une histoire de la lecture*. Edition Actes Sud, 1998, p.155.
- [OBE96] U. Obeysekare, C. Williams, J. Durbin, L. Rosenblum, R. Rosenberg, F. Grinstein, R. Ramaturi, A. Landsberg, W. Sandberg. Virtual Workbench – A non-immersive Virtual Environment for Visualizing and Interacting with 3D Objects for Scientific Visualization. *In proceedings of Visualization*, San Francisco, USA, 1996.
- [PIC97] R. Picard. *Affective Computing*. MIT Press, 1997.
- [ROB91] G. G. Robertson, J. D. Mackinlay, S. K. Card. Cone trees : animated 3D visualizations of hierarchical information. *Communications of the ACM*, 36(4), April 1993, pp. 57-71.
- [ROB93a] G. G. Robertson, S. K. Card, J. D. Mackinlay. Information Visualization Using 3D Interactive Animation. *Communications of the ACM*, Vol. 36, No 4, April 1993, pp. 57-71.
- [ROB93b] G. G. Robertson, J. D. Mackinlay. The Document Lens. *ACM UIST'93*.
- [ROB2000] G. Robertson M. Dantzich, D. Robbins, M. Czerwinski, et al. The task gallery: A 3D window Manager, *In Proceedings of CHI 2000*, pp. 494-501, 1-6 April 2000.
- [SHN94] B. Shneidermann. Dynamic queries for visual information seeking. *IEEE Software* 11, 6 (1994), pp 70-77.
- [SIG2000] H. Barad (organizer), E. Haines, D. Macri, T. Möller, K. Pallister. *Aggressive Performance Optimizations for 3D Graphics*. Course #43, SIGGRAPH 2000, July 2000, New Orleans, Louisiana.
- [SMI98] J. Smith, T. White, C. Dodge, D. Allport, J. Paradiso, N. Gershenfeld. Electric Field Sensing for Graphical Interfaces. *IEEE Computer Graphics and Applications*, Vol. 18, No. 3, May-June 1998, pp. 54-60.
- [STO94] M. C. Stone, K. Fishkin, E. A. Bier. The Movable Filter as a User Interface Tool. *In Proceedings of CHI'94*, Boston, MA, April 24-28. ACM, New York, 1994, pp. 306-312.
- [STO95] R. Stoakley, M. J. Conway, R. Pausch. Virtual Reality on a WIM : Interactive Worlds in Miniature. *In Proceedings of CHI'95 Human Factors in Computing Systems*, Denver, CO, May 1995.
- [TOP2000] A. Topol. Immersion of Xwindow applications into a 3D workbench. *ACM CHI'2000*, The Hague, Netherland, April 2000
- [VD91] A. Van Dam. Escaping Flatland. *UIST'91*. November 1991.
- [YOU96] P. Young. *Three Dimensional Information Visualisation*. Internal Report, Centre for Software Maintenance, University of Durham, March 1996.

[ZEL97] R. C. Zeleznik, A. S. Forsberg, P. S. Strauss. Two Pointer Input for 3D Interaction. *In Proceedings of 1997 Symposium on Interactive 3D Graphics*, Providence, Rhode Island, April 27-30, 1997.

[ZEL99] R. C. Zeleznik, A. S. Forsberg. UniCam – 2D Gestural Camera Controls for 3D Environments. *Symposium on Interactive 3D Graphics Atlanta GAUSA*, 1999.

[ZHA94] S. Zhai, W. Buxton, P. Milgram. The Silk Cursor: Investigating Transparency for 3d Target Acquisition. *In Proceedings of CHI'94*, ACM Conf. on Human Factors in Computing Systems, Boston, MA, April, 1994.

[WAT95] J. A. Watlington, M. Lucente, C. J. Sparrell, V. M. Bove, I. Tamitani. A hardware architecture for rapid generation of electro-holographic fringe patterns. *In proceedings of SPIE*, 1995.

Informations complémentaires :

Java et Java 3D :

Flanagan David, *JAVA in a nutshell, A Desktop Quick Reference*, O'Reilly, Sebastopol, 2^{ème} édition, 1997.

Flanagan David, *JAVA examples in a nutshell, A Tutorial Companion to Java in a nutshell*, O'Reilly, Sebastopol, 1997.

Arnold Ken, Gosling James, *The Java Programming Language*, The Java Series, Addison-Wesley, Reading (Massachussets)

Sowizral Henry, Rushforth Kevin, Deering Mickael, *The Java 3D API Specification*, The Java Series, Addison-Wesley, Reading (Massachussets), December 1997.

Site Java de Sun :

Pour le Kit de Développement Java (documentations et téléchargement) :

<http://www.javasoft.com/product/jdk/1.2/>

Pour Java3D (documentations et téléchargement) :

<http://www.javasoft.com/products/java-media/3D/>

OpenGL :

[OpenGLa] Woo Mason, Neider Jacky, Davis Tom. *OpenGL Programming Guide, Second Edition, The Official Guide to Learning OpenGL, Version 1.1*, OpenGL Architecture Review Board, Addison-Wesley Developers Press, Janvier 1997.

Tutoriels, Documentations, Téléchargement :

<http://www.opengl.org>

<http://reality.sgi.com/opengl/>

Direct3D :

[D3Da] Guide de références en ligne et tutoriels : <http://www.microsoft.com/directx/overview/d3d>

Informations générales sur Direct3D: <http://www.directxfaq.com/direct3D.htm>

VRML :

Hartman J., Wernecke J., *The VRML 2.0 Handbook - Building Moving Worlds on the Web*, Silicon graphics Inc., Addison Wesley Developers Press.

Campbell B., Marrin C., *Teach yourself VRML2 in 21 days*, Sams Net.

[WRL1] Spécifications VRML1

[WRL2] Spécifications VRML97

Le Consortium VRML (anciennement <http://www.vrml.org>) : <http://www.web3d.org/vrml/>

Le journal des développeurs VRML : <http://www.VRMLDevelopersJournal.com>

Tutoriel VRML97, liens vers les différents *plugins* (en français) :
<http://apia.u-strasbg.fr/vrml/>

Le groupe VRML francophone : <http://webhome.infonie.fr/kdo/vrml/>

MPEG4 :

[MPG1] O. Avaro et al. The MPEG-4 system and description languages : A way ahead in audio visual information representation. *Signal Processing: Image Communication*, Special Issue on MPEG-4, Vol. 9, No. 4, May 1997, pp. 385-431.

[MPG2] A. Marques. *Conception et développement d'un logiciel Client Serveur MPEG-4 pour des applications de communication de groupe et consultation multimédia*. Mémoire d'ingénieur C.N.A.M., Juin 1998.

[MPG3] Site officiel de la norme MPEG-4 :
<http://www.mpeg.org/MPEG/starting-points.html#mpeg4>

Lex et Yacc :

[SIL95] Silvero Nino, *Réaliser un compilateur ; Les outils Lex et Yacc*, 2^{ème} édition, Eyrolles, 1995.

[AHO91] Aho Alfred, Sethi Ravi, Ullman Jeffrey, *Compilateurs ; Principes, techniques et outils*, Collection Informatique Intelligence Artificielle, InterEditions, 1991.

[BEN91] Benzaken Claude, *Systèmes formels : Introduction à la logique et à la théorie des langages*, Collection Logique Mathématiques Informatique, Masson, Paris, 1991.

Divers :

[ISA] Isakovic K., Liste des différentes API 3D recensées à ce jour.
<http://www.cg.cs.tu-berlin.de/~ki/engines.html>

[BLAX] Site Blaxxun : chargement, documentations et informations
<http://www.blaxxun.com/>

[SCOL] Site Scol : chargement, documentations, ...
<http://www.cryo-networks.com/>

[SUPV] Site Superscape de Viscap :
<http://www.viscape.com>