Contribution à la définition d'un logiciel graphique pour la visualisation et le dialogue interactifs structurés

Philippe Genoud

Thèse

Présentée par

Philippe GENOUD

pour obtenir le titre de **DOCTEUR**

de **L'Université Joseph Fourier - Grenoble I**
(Arrêté Ministériel du 5 Juillet 1984)

(Spécialité Informatique)

Contribution à la Définition d'un Logiciel Graphique pour la Visualisation et le Dialogue Interactifs Structurés

Thèse soutenue le 12 Janvier 1989 devant la commision d'examen :

P.C. SCHOLL	Président
B. DAVID	Rapporteur
B. PEROCHE	Rapporteur
F. MARTINEZ	
Ph. BOULLE	

Thèse préparée au sein du laboratoire **ARTEMIS**

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. PAYAN Jean Jacques

Année Universitaire 1987 - 1988

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ARNAUD Paul	Chimie Organique
ARVIEU ROBERT	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BONNIER Jane Marie	Chimie Générale
BOUCHEZ Robert	Physique Nucléaire ISN
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques Pures
GAGNAIRE Didier	Chimie Physique
GERMAIN Jean-Pierre	Mécanique,
GIRAUD Pierre	Géologie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques Pures
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées

LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
PIERRE Jean-Louis	Chimie Organique
RENARD Michel	Thermodynamique
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VALENTIN Jacques	Physique Nucléaire I.S.N.
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ADIBA Michel	Mathématiques Pures
ANTOINE Pierre	Géologie
ARMAND Gilbert	Géographie
BARET Paul	Chimie
BLANCHI J.Pierre	STAPS
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BRUANDET J.François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
COURT Jean	Chimie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
GUITTON Jacques	Chimie

HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
JARDON Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKHOVE Claude	Géologie
LONGEQUEUE Nicole	Sciences Nucléaires I.S.N.
LUCAS Robert	Physique
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
NEMOZ Alain	Thermodymamique CNRS - CRTBT
OUDET Bruno	Mathématiques Appliquées
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PERRIN Claude	Sciences Nucléaires I.S.N.
PFISTER Jean-Claude	Physique du Solide
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	EEA. IUT 1

PROFESSEURS de 2ème classe

BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHEHIKIAN Alain	EEA. IUT 1
CHENAVAS Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
PEFFEN René	Métallurgie IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
VINCENDON Marc	Chimie IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Therapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiéne et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie-Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital sud
COUDERC Pierre	Anatomie-Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci
HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.
MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie-Obstétrique	Hopital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

Je tiens à remercier,

Monsieur Pierre Claude SCHOLL, professeur à l'Université Joseph Fourier, qui m'a fait l'honneur de présider le jury

Messieurs Bernard PEROCHE, professeur à l'Ecole des Mines de Saint Etienne, et Bertrand DAVID, professeur à l'Ecole Centrale de Lyon, qui ont accepté avec bienveillance de jouer le rôle ingrat de rapporteurs, et ceci malgré l'aspect dissuasif de ce mémoire qui en aurait rebuté plus d'un.

Monsieur Francis MARTINEZ, professeur à l'Université Joseph Fourier, qui en m'accueillant dans l'équipe graphique du laboratoire ARTEMIS m'a permis d'entamer cette thèse. Ces conseils m'ont été des plus précieux, sa compétence un modèle. Je lui suis extrêmement reconnaissant d'avoir toujours répondu présent, malgré les difficultés matérielles, et de m'avoir ainsi permis de mener à bout mes travaux.

Monsieur Philippe BOULLE, Ingénieur à GETRIS-IMAGE, qui m'a fait l'amitié de participer à ce jury.

Jean Francois GRABOWIECKI, qui sait combien je lui suis redevable de tout le temps qu'il m'a consacré tout au long de mes travaux, de la patience dont il a toujours su faire preuve, et de la pertinence de ses remarques. Son aide m'a été des plus précieuses, et m'a permis de surmonter les moments difficiles...

Je remercie aussi les membres de l'équipe graphique, Gilles KUNTZ, Sylvère BRUNEAUX, leurs conseils techniques avisés m'ont souvent été indispensables. A eux, il faut ajouter mes anciens "collègues" Victor Hugo ZARATE et Karim CHIBANE, avec qui j'ai partagé nos seulement mes bureaux, mais aussi toutes les angoisses et les joies qui jalonnent la vie du thésard de base.

Je ne saurai oublier les membres du laboratoire ARTEMIS et, en particulier, le secrétariat animé par Claudine MEYRIEUX dont le dynamisme et l'efficacité ont plus d'une fois été à mon secours.

Danielle ZIEBELIN et Olivier BOISSIER m'ont apporté une aide précieuse et leur support moral (et il fallait pouvoir me supporter !) ne m'a jamais fait défaut.

A cette liste déjà longue, il me faut encore ajouter, les membres du Département d'Informatique de l'Ecole Normale Supérieure de Lyon

dirigé par Michel COSNARD. En m'accueillant, ils m'ont permis de vivre une expérience o combien enrichissante et m'ont donné les moyens d'achever mes travaux.

Je remercie enfin les membres du service reprographie, pour le soin qu'ils ont apporté à la réalisation matérielle de ce rapport.

Bien sur, la liste ci-dessus n'est pas exhaustive. Comment exprimer en quelques mots toute la reconnaissance et la gratitude que je dois à tous ceux que j'ai cités, et comment citer tous ceux à qui je voudrais adresser mes remerciements ?

SOMMAIRE

Sommaire

1ère PARTIE : Etat de l'art

2ème PARTIE : CLOVIS une première Expérimentation pour la Visualisation Interactive Structurée

3ème PARTIE : Vers un Logiciel Intégré pour la Visualisation et le Dialogue Structurés

III : Modélisation et structuration hiérarchique des données graphiques

2ème PARTIE : CLOVIS une première Expérimentation pour la Visualisation Interactive Structurée

IV : CLOVIS du point de vue utilisateur

3ème PARTIE : Vers un Logiciel Intégré pour la Visualisation et le Dialogue Structurés

VI : De CLOVIS à un système pour la visualisation et le dialogue interactifs structurés : présentation générale du système.

VII : Les Objets d'interface

INTRODUCTION

"L'ordinateur graphique a ouvert l'ère de la culture par la communication à travers l'image". Cette affirmation de F. Vanderbroucke lors de EUROGRAPHICS'87 [Van 87] résume bien l'expansion formidable que l'infographie a connue ces dernières années et le rôle de plus en plus important que jouent les images de synthèse dans notre société...

Que de chemin parcouru en trentes ans, depuis les premières utilisations d'ordinateurs pour construire des dessins au Massachusset Institute of Technology, depuis le projet SAGE de système de défense et de contrôle aérien du département de la défense Américaine! Encore reservée, il y a peu, à une étroite communauté de spécialistes, l'infographie a connue une véritable "explosion" au cours de la dernière décennie et est sortie des laboratoires pour faire irruption jusque dans notre quotidien au travers de la vidéo et de la télévision.

Mais si l'emploi d'images de synthèse dans les spots publicitaires ou au cinéma constitue l'un des aspects les plus spectaculaires de l'infographie, celle-ci dépasse largement le cadre de l'audiovisuel et trouve ses applications dans la quasi-totalité des domaines où l'informatique peut jouer un rôle : que ce soit de la Conception Assistée par Ordinateur, à la médecine en passant par la simultation... Partout, l'image est le support de communication le plus naturel, le plus puissant et le plus fiable, à tel point que le graphique fait désormais partie intégrante de l'informatique et joue un rôle essentiel dans l'interface des ordinateurs.

Cette expansion extraordinaire du marché de l'infographie, observée ces dernières années, est directement liée aux formidables progrès de la technologie, et en particulier de l'électronique.

Ainsi, alors qu'il n'y pas très longtemps, le matériel graphique était cher, encombrant, pas toujours très maniable et que les résultats n'étaient pas toujours très spectaculaires, la situation actuelle est radicalement différente. L'intégration de plus en plus grande des circuits électroniques, en augmentant les performances tout en diminuant les couts de production, a rendu accessible à des utilisateurs de plus en plus nombreux des matériels aux capacités graphiques toujours plus sophistiquées. La manifestation la plus spectaculaire de cette évolution concerne certainement le domaine de la micro-informatique, où les machine actuelles font une part de plus en plus importante au graphique.

Cependant, si les progrès technologiques ouvrent des perspectives formidables pour les prochaînes décennies, avec des ordinateurs graphiques encore plus rapides, mieux intégrés, plus intelligents, l'optimisme général dans une débauche de superlatifs est à relativiser.

En effet, si la plupart des problèmes techniques ont trouvé des réponses, il s'agit le plus souvent de solutions ponctuelles adaptées au cas particulier des données et images à traiter. Et les utilisateurs potentiels de l'infographie, s'ils ont tendance à devenir de plus en plus exigeants (peut être en partie à cause des très spectaculaires images de synthèse qu'ils peuvent voir en particulier à la télévision) mésestiment bien souvent les difficultés liées au développement d'applications graphiques.

En effet il s'agit d'une tâche complexe dans laquelle il faut prendre en compte des matériels aux possibilités extrémement variées, et dont le cycle de vie tend de plus à ce réduire. Comme dans de nombreux domaines de l'informatique, on assiste également en graphique à une certaine "crise" du logiciel qui n'est pas toujours à la hauteur des capacités du matériel.

Les efforts de standardisation de l'interface des logiciels graphiques, qui ont débuté à la fin des années 70, constituent dans ce domaine une étape importante qui a certainement contribuée à une meilleure compréhension et diffusion du graphique. Cependant, ces système reflètent dans une large mesure l'état de l'art d'une époque et sont difficilement adaptables aux matériels et aux besoins actuels.

L'objet de cette thèse, est justement la contribution que nous avons apportée au sein de l'équipe graphique du laboratoire ARTEMIS, à la conception de logiciels de base permettant de répondre de manière efficace aux besoins des applications.

*Dans la **première partie** de ce rapport nous dressons un état de l'art des aspects logiciels de l'infographie. Le chapitre I nous permet de présenter un certain nombre des concepts de base qui nous ont guidé tout au long de notre travail. L'objet du chapitre II est l'étude des différentes approches possibles pour le logiciel graphique. Nous nous interessons tout particulièrement aux systèmes graphiques généraux, indépendants de leur contexte d'utilisation (du matériel graphique mais également du calculateur hôte et des applications). A ce titre nous décrivons différents logiciels de visualisation, en particulier les "standards" GSPC et GKS (Graphic Kernel system). Le chapitre III nous permet de revenir sur l'aspect modélisation, et l'utilisation dans ce cadre des structures hiérarchiques. A ce titre, nous présentons PHIGS (Programmers Hierachical Interactive Graphic System) qui intègre au système graphique l'activité de modélisation géométrique.*

*La **deuxième partie** de ce document consiste en la présentation du projet CLOVIS (Complexe Logiciel pour la Visualisation Interactive Structurée), un logiciel graphique que nous avons conçu et réalisé au laboratoire ARTEMIS. Organisé autours d'une base de données hiérarchique pour la représentation des objets graphiques, il propose aux applications des primitives de haut niveau prenant entièrement en charge les processus de base de la synthèse d'images : attribution, consultation, description et visualisation.*

Le chapitre IV, expose l'ensemble des fonctionnalités proposées aux utilisateurs (programmeurs d'application) par CLOVIS. Le chapitre V nous permet de revenir sur certains des points ayant trait à la réalisation et à l'implémentation du logiciel. Nous présentons l'organisation modulaire de celui-ci, les interfaces entre les différents niveaux. Nous insistons tout particulièrement sur la gestion et la structuration hiérarchique des données graphiques.

*La **troisième partie** présente un second logiciel graphique qui a été conçu et réalisé suite à l'expérience CLOVIS. En effet, si le "prototype" CLOVIS nous a permis de valider l'essentiel des concepts de base ayant guidé sa réalisation (intérêt de la structure hiérarchique, de la prise en charge des processus de base par le système graphique...), son utilisation a néanmoins révélé un certain nombre de lacunes. Deux points furent ainsi particulièrement mis en évidence : d'une part, le manque de souplesse d'utilisation de CLOVIS du fait que toute visualisation passait par la construction préalable d'une structure mémorisée dans la base de données graphiques hiérarchique, d'autre part l'insuffisance des fonctionnalités proposées pour la gestion du dialogue interactif et la définition d'interfaces graphiques homme-machine évoluées. Ce sont ces deux points qui ont donc fait l'objet de la suite de nos travaux et qui ont motivé la réalisation du second système que nous présentons ici.*

Le chapitre VI effectue une présentation générale du système. Nous mettons tout particulièrement en avant les éléments ayant trait à la prise en compte des problèmes liés à l'interface homme-machine. En, effet ce sont eux qui ont entrainé le plus de boulversements par rapport à ce que nous avions fait pour CLOVIS.

Dans le chapître VII nous revennons plus en détail sur les fonctionnalités qui ont été développées dans ce cadre et qui consistent en la prise en charge par le système graphique d'objets évolués spécifiquement destinés à la gestion du dialogue interactif (objets d'interface).

Le chapître VIII expose quand à lui les fonctionnalités ayant trait à la manipulation d'objets plus spécialement destinés à la visualisation (objets graphiques). Il s'agit essentiellement d'améliorations apportées à ce qui avait été développé pour CLOVIS : en particulier l'applications dispose de plusieurs modes qui permettent de combiner aisément la définition des objets graphiques avec leur visualisation et/ou leur mémorisation et/ou leur structuration hiérarchique.

Cette thèse relate donc le long chemin qui nous a conduit vers un logiciel graphique de haut niveau qui nous le pensons est de plus en plus complet tout en conservant une grande facilité et une souplesse d'utilisation. Cependant, il ne s'agit que d'une étape vers une meilleure définition et compréhension des sytèmes graphiques, qui ouvre des perspectives de recherche que nous présenterons en conclusion de cet ouvrage.

1ère PARTIE

ETAT DE L'ART

Chapître I : Concepts de base de la synthèse d'images

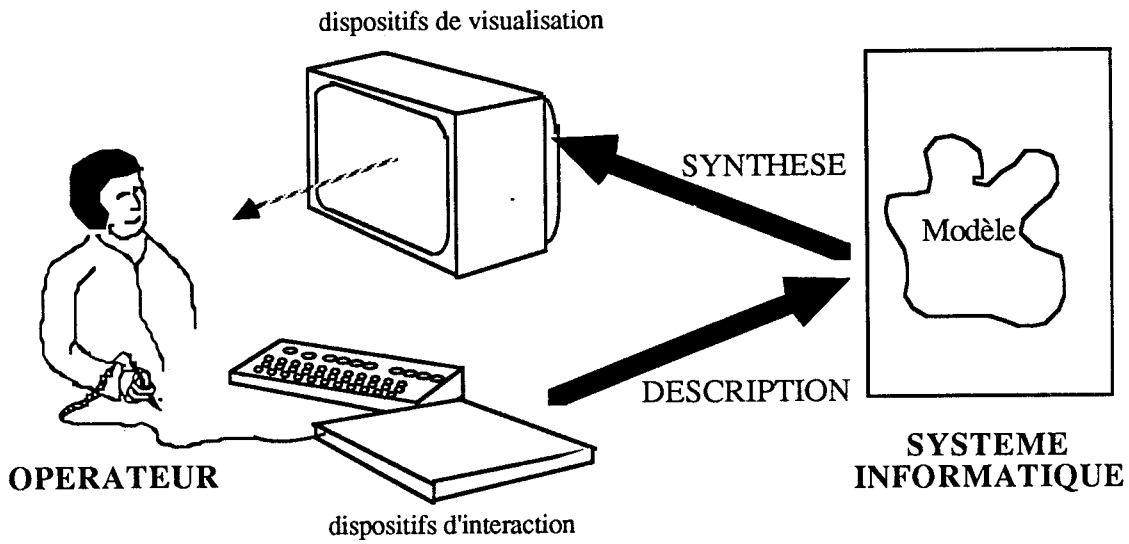
Chapitre I : CONCEPTS DE BASE DE LA SYNTHESE D'IMAGES.

1 LES INTERLOCUTEURS D'UN SYSTEME DE SYNTHESE D'IMAGES

L'image de part sa puissance évocatrice et la quantité d'informations qu'elle véhicule est un support privilégié pour la communication d'informations. Aussi est-il naturel que, très tôt, on se soit intéressé à son utilisation dans les systèmes informatiques.

L'**infographie interactive** est la production d'images à l'aide d'ordinateurs. Ces images ne servent pas uniquement à une présentation des résultats finaux mais **interviennent directement comme support du dialogue homme-machine.**

La figure ci-dessous schématise la "boucle d'interaction" entre les deux interlocuteurs que sont l'opérateur et le système informatique dont nous allons préciser le rôle dans les paragraphes qui suivent.



- figure I.1 : les interlocuteurs d'un système de synthèse d'images -

1.1 L'opérateur humain

Le rôle premier de l'opérateur humain est bien évidemment d'analyser les images qui lui sont présentées afin d'en extraire les informations synthétisées.

Dans un système interactif, l'opérateur peut agir **dynamiquement** sur l'image, en modifer le contenu, le format, la couleur... Pour ce faire les informations qu'il doit fournir en retour au système informatique sont le plus souvent définies sous "forme graphique" et introduites à l'aide des dispositifs d'interaction disponibles sur le poste de travail. Par exemple, la désignation d'un élément de l'image affichée à l'aide d'une souris, d'un clavier ou d'une tablette à digitaliser.

1.2 Le système informatique

Le système informatique a pour rôle la création, le stockage et la manipulation des "modèles" des objets. Il doit en particulier :

- produire des images à partir des informations contenues dans les modèles des objets,
- récupérer et traiter les informations introduites par l'opérateur à l'aide de divers dispositifs d'entrée. La plupart du temps ces informations agissent directement sur le modèle des objets et les modifications qu'elles y apportent peuvent être repercutées soit **immédiatement** sur la représentation visuelles des objets, soit **différées.**

Dans un système informatique faisant appel aux techniques de la synthèse d'images, on peut distinguer deux composants :

- **le système de synthèse d'images** proprement dit, regroupant l'ensemble des logiciels plus particulièrement destinés à la création et la manipulation d'images ;

- **le programme d'application** pour lequel la production d'images n'est pas obligatoirement la finalité, mais uniquement un support privilégié pour la communication d'informations avec l'utilisateur. Le domaine d'application du programme peut être tout autre comme on peut le constater par exemple pour de nombreux systèmes de C.A.O tels que le calcul de résistance des matériaux, les calculs aérodynamiques... Le programme d'application utilise les services du système de synthèse d'images pour dialoguer avec l'opérateur.

Le système de synthèse d'images joue donc, le rôle d'**intermédiaire entre le programme d'application et l'opérateur humain.** Il assure les échanges de l'application vers l'opérateur en effectuant la synthèse des informations qu'elles lui fournit afin d'aboutir à l'image finale qui sera visualisée. En ce qui concerne les échanges de l'opérateur vers l'application, le système de synthèse d'images se charge de transformer les données décrites par l'opérateur à l'aide des dispositifs de saisie en une forme acceptable par le programme d'application.

La séparation entre l'application et le système de synthèse d'images n'est pas toujours très nette. La frontière peut se situer à de nombreux niveaux :

- le système de synthèse peut se réduire à un simple ensemble matériel, l'application devant entièrement prendre en charge la gestion de celui-ci,

- le système de synthèse peut proposer une interface de plus haut niveau, et un ensemble d'outils logiciels spécifiques à la synthèse d'images.

La "puissance" d'un système de synthèse d'images est définie par ce niveau d'interface, car la prise en charge du processus de synthèse facilite l'écriture du programme d'application. Elle permet au concepteur de l'application se concentrer exclusivement sur les problèmes liés à cette dernière sans "s'embarrasser" avec les problèmes spécifiques à la visualisation graphique.

2 LES ETAPES DU PROCESSUS DE SYNTHESE D'IMAGES

Afin de mieux identifier le rôle d'un système de synthèse d'images, nous allons rapidement caractériser les différentes étapes qui interviennent dans le processus de synthèse et énoncer les différents types d'informations qu'elles utilisent [Mart 82].

2.1 Constitution d'un modèle d'objet

Le programme d'application doit contenir, sous forme de structures de données ou sous forme procédurale, un modèle du (des) objet(s) qu'il manipule et qui devront être présentés sous une forme visuelle.

Le modèle va donc regrouper un ensemble de données spécifiques de l'application (exemple : la résistance des matériaux, leur masse, etc...) et un ensemble de données plus particulièrement destinées à la présentation visuelle de l'objet. Nous nommerons désormais ces dernières informations : **attributs graphiques**.

Les attributs graphiques du modèle sont de natures différentes :

- des attributs définissant la forme intrinsèque des objets (par exemple des suites de coordonnées cartésiennes), nous les désignerons sous le terme de **MORPHOLOGIE (M)**,

- des attributs permettant le positionnement des objets les uns par rapport aux autres que nous désignerons sous le terme **GEOMETRIE (G)**,

- des attributs définissant l'apparence de chaque objet ; par exemple sa couleur, la texture des matériaux dont il est constitué, sa brillance, son mode de réflexion de la lumière... L'ensemble de ces données est regroupé sous le terme d'**ASPECT (A)**.

A ces informations, purement graphiques, peuvent être ajoutées des informations "structurelles" nécessaires à la gestion de ces données. Il s'agit d'informations d'**IDENTITE (I)** qui permettent la nomination et la désignation d'objets ou d'ensemble d'objets, et d'informations de **STRUCTURE (S)** qui permettent d'exprimer les relations liant les objets entre eux. Ces relations peuvent être de nature logique (appartenance, inclusion...), topographiques (contact, proximité...) ou fonctionnelles.

L'élaboration du modèle peut se dérouler en deux phases :

- la **description**, qui permet de définir les éléments constitutifs du modèle et leurs liens relationnels. Pour cela il est nécessaire de définir séparement chacune des informations graphiques élémentaires qui le composent. Cette description peut s'effectuer interactivement, les outils alors proposés devant exploiter au maximum les caractéristiques des objets pour en faciliter la description.

- la **construction,** qui est la réalisation effective du modèle de l'objet à partir des informations fournies lors de la description.

Exemple : la description d'un objet de révolution peut s'effectuer par la simple donnée d'un contour plan et d'un axe de révolution. La construction du modèle approximant sa surface à l'aide de faces polygonales se faisant par rotation (balayage) du contour autour de l'axe.

2.2 La visualisation

La visualisation des objets définis pour l'application peut être assimilée au processus photographique ("the camera paradigm") où deux opérations principales sont à considérer :

- la prise de vue elle-même,
- l'affichage de la vue obtenue.

2.2.1 La prise de vue

Cette étape nécessite la définition des paramètres régissant les conditions dans lesquelles le modèle de la scène est observé afin d'en produire une image.

Un premier groupe de paramètres permet de spécifier la transformation à appliquer aux coordonnées définissant les éléments à l'intérieur du modèle de la scène afin d'aboutir à une représentation de ceux-ci dans le repère de la vue. Dans le cas d'une image "tridimensionnelle" l'analogie avec la prise de vue à l'aide d'un appareil photographique est totale. L'opérateur doit indiquer quelle portion de la scène il désire visualiser (définition d'un point de visée), sous quel angle celle-ci sera observée (définition d'un point de vue), le type de projection utilisée (les caractéristiques de l'appareil de prise de vue c'est à dire la distance focale, l'ouverture, le format de prise de vue...).

Tous ces paramètres s'apparentent à des informations d'ordre géométrique (transformations de coordonnées) ; pour les distinguer des données géométriques définies précédemment lors de la description des objets, nous les identifierons par le terme de **GEOMETRIE DE PRISE DE VUE (G_V)**.

Le second groupe de paramètres s'applique à l'apparence des objets. Il s'agit de paramètres indiquant les conditions de visibilité de la scène (couleur et direction d'une ou plusieurs sources lumineuses, effet de brouillard... etc) que nous regrouperons sous la rubrique **ECLAIRAGE (E)**.

Il est à noter que les paramètres de géométrie de prise de vue et d'éclairage s'appliquent de façon globale et identique à tous les objets de la scène.

2.2.2 L'affichage de la vue obtenue

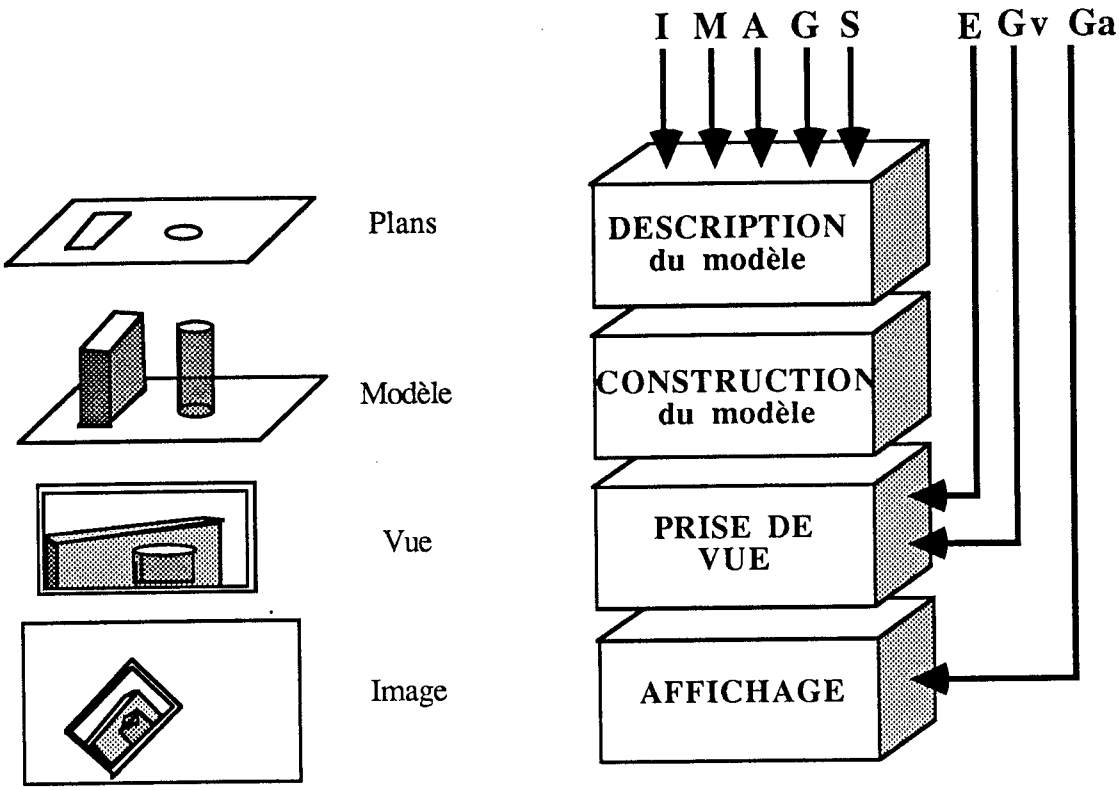
La deuxième étape est simplement la présentation de la vue obtenue précedemment sur la surface de visualisation (par exemple un écran de moniteur T.V.).

On peut, par exemple, situer une vue par rapport à d'autres vues qui seront affichées simultanément. Ici encore, il s'agit d'attributs de nature géométrique, qui permettent d'exprimer une transformation du repère de la vue au repère de la surface de visualisation, aussi nous les désignerons de façon spécifique par **GEOMETRIE D'AFFICHAGE (G_a)**.

2.3 Récapitulatif

Dans ce qui précède nous avons mis en évidence les différentes classes d'informations qui interviennent dans le processus de synthèse d'images. Celui-ci consiste à combiner (synthétiser) les attributs de ces différentes classes pour aboutir à l'information finale qui est l'aspect de chaque point sur la surface de visualisation.

La figure ci-dessous résume les principales étapes de la synthèse d'images et les niveaux où sont pris en compte les différents attributs élémentaires.



- figure I.2 : les différentes étapes de la synthèse d'images -

Il est clair qu'à l'intérieur de chacune de ces classes (**I**dentité, **M**orphologie, **A**spect, **G**éométrie, $\mathbf{G_v}$ géométrie de prise de vue, $\mathbf{G_a}$ géométrie d'affichage, **E**clairage, **S**tructure), nous pouvons distinguer un certain nombre de types d'attributs. Cette distinction peut s'effectuer en fonction de critères :

- sémantiques (par exemple les types CERCLE, CARACTERE à l'intérieur de la classe Morphologie),
- syntaxiques (modélisation des coordonnées sous forme cartésienne, ou sous forme polaire...),
- ou fonctionnels (unité de mouvement, unité d'aspect pour un ensemble d'éléments...).

Un système de synthèse d'images sera caractérisé en partie par les différents types d'attributs qu'il accepte. En particulier nous désignerons sous le terme d'**éléments** les différentes représentations qu'il supporte pour les objets.

Les types d'éléments les plus couramment rencontrés sont par exemple :

• les éléments fils de fer où seules les arêtes des objets sont définies, l'attribut Morphologie est alors représenté simplement par les coordonnées des points extrémités des arêtes,

• les éléments de types faces polygônales planes où l'attribut Morphologie est représenté par les coordonnées de la facette et du vecteur normal...

Les types d'éléments sont fonction des types d'attributs des classes de Morphologie, d'Aspect, de Géométrie, d'Identité et de Structure qui les composent et peuvent être de nature très variée.

Par ailleurs, la représentation supportée par le système de synthèse d'images peut également imposer que certains attributs lui soient communiqués sous une forme "présynthétisée". Par exemple, supposons que le système de synthèse ne prenne pas en charge la gestion des attributs géométriques, le programme d'application doit alors, préalablement, appliquer lui même les transformations géométriques aux coordonnées définissant les objets avant leur transmission au système de synthèse.

Il est donc clair que moins le système accepte une **définition séparée** par attributs et plus la responsabilité de l'application dans la gestion des informations graphiques et des processus de la synthèse d'images est importante.

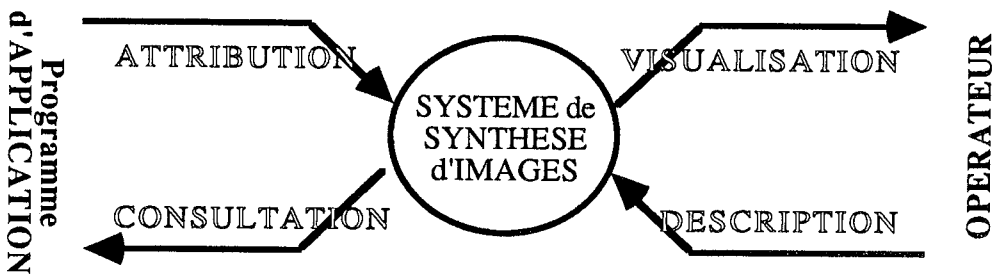
3 LES PROCESSUS DE BASE D'UN SYSTEME DE SYNTHESE D'IMAGES

Comme nous l'avons vu le système de synthèse d'images joue le rôle d'intermédiaire entre le programme d'application et l'opérateur. Aussi après avoir caractérisées les informations qu'il est ammené à manipuler, nous allons maintenant tenter de spécifier les différentes fonctions qu'il assume vis à vis de ces deux interlocuteurs.

Le système de synthèse d'images doit être capable d'assurer les échanges dans **les deux** sens entre l'application et l'opérateur.

On peut ainsi identifier quatre processus prenant en charge ces différents échanges (fig. I.3) :

- l'**attribution** de l'application vers le système de synthèse,
- la **consultation** du système de synthèse vers l'application,
- la **visualisation** du système de synthèse vers l'opérateur,
- la **description** de l'opérateur vers le système de synthèse.



- figure I.3 : les 4 processus de base de la synthèse d'images (d'après [Mart 82]) -

Bien entendu ces quatre processus sont sous le contrôle de l'application qui se charge de les invoquer par le biais des primitives proposées par le système de synthèse d'images.

3.1 Processus d'attribution

Le processus d'attribution consiste en une affectation dans les structures de données du système de synthèse d'**attributs calculés** par le programme d'application.

Le déroulement de processus d'attribution s'effectue en deux temps :

- la recherche dans les structures de données du système de synthèse des éléments concernés,
- l'affectation proprement dite.

3.2 Processus de description

Le processus de description consiste en l'affectation d'attributs graphiques à des éléments du système de synthèse d'images, ces attributs étant définis **interactivement** par l'opérateur à l'aide des dispositifs de dialogue (contrairement à l'attribution où les attributs sont calculés par le programme d'application).

Les processus de description peuvent être extrèmement complexes. En effet l'information rendue par les dispositifs de dialogue (le plus souvent un couple de coordonnées cartésiennes) doit être transformée à l'aide d'opérateurs de construction pour obtenir les attributs de base considérés par le système de synthèse d'images.

3.3 Processus de consultation

De la même manière qu'il peut affecter des attributs à des éléments du système de synthèse, le programme d'application doit pouvoir les récupérer. Par exemple, le programme d'application doit pouvoir consulter les attributs définis par l'opérateur lors d'une description interactive.

3.4 Processus de visualisation

Le processus de visualisation est de loin le processus le plus complexe du système de synthèse d'images et est aussi celui qui influe le plus son sur architecture.

Rappelons qu'un processus de visualisation doit assurer la synthèse des attributs descriptifs des éléments pris en compte par le système pour aboutir à l'information finale acceptée par le dispositif d'affichage (par exemple signaux vidéo pour les couleurs primaires Rouge, Vert et Bleu dans le cas d'un moniteur T.V.).

De façon générale un processus de visualisation met en jeu quatre types d'opérateurs élémentaires :

* **des opérateurs de synthèse**, ils se chargent de la composition de deux attributs en un nouvel attribut. Cela peut être, par exemple, l'application des informations géométriques aux informations morphologiques, ou la modulation de l'aspect en fonction des conditions d'éclairage.

* **des opérateurs de construction**, ils transforment une information d'un type donné en un autre. Par exemple à partir d'un cercle défini par son centre et son rayon, les opérateurs de

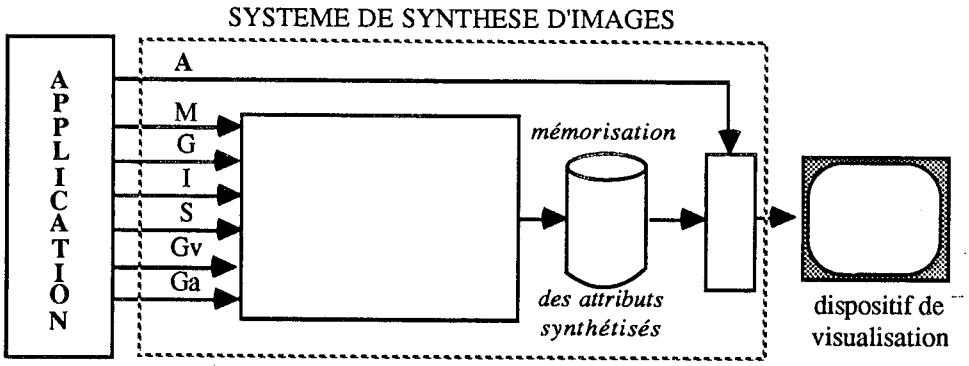
construction génèrent les points constituant son contour.

* **des opérateurs de composition**, ils concernent les attributs de plusieurs éléments simultanément. De tels opérateurs sont par exemple employés dans les algorithmes d'élimination des parties cachées sous la forme de tri comme l'ordonnancement des faces planes selon leur profondeur (algorithme de Newell Newell et Sancha [NNSa 72], [SSSc 74]).

* **des opérateurs de mémorisation**, ils permettent de stocker des informations sous une forme intermédiaire à l'intérieur du processus de visualisation. Il peut s'agir d'opérateurs de mémorisation "synchrone", qui, pour chaque entrée fournissent un résultat en sortie, par exemple une table de correspondance qui permet d'obtenir la normale à une face polygônale à partir de son identification ; ou bien d'opérateurs de mémorisation "asynchrone" jouant le rôle de mémoire tampon entre deux processeurs différents, un exemple typique est l'utilisation de mémoire d'images ou mémoire de trame entre le processeur graphique et le processeur d'affichage [Zara 85].

Un processus de visualisation peut ainsi être caractérisé par la **nature** (logicielle, microprogrammée, logique cablée) et l'**ordonnancement** de ces opérateurs.

En particulier, les opérateurs de mémorisation et la pénétration des attributs (c'est à dire le moment où leur prise en compte dans le processus de synthèse intervient) jouent un très grand rôle dans la puissance et l'interactivité du système. Après une mémorisation intermédiaire d'attributs partiellement synthétisés, l'affichage et le réa-affichage se réduisent à l'application des opérateurs qui suivent l'opérateur de mémorisation. Si de plus, des attributs sont pris en compte à ce niveau là (après l'opérateur de mémorisation), il est alors possible de les modifier et de revisualiser l'image avec une diminution considérable des temps de réponse. En effet il n'est alors pas nécessaire de refaire tout le traitement antérieur à l'opérateur de mémorisation.



- figure I.4 : La pénétration des attributs -

Par exemple, si l'attribut définissant la couleur des objets est pris en compte en dernier lieu, après que les autres attributs aient été synthétisés et mémorisés (fig. I.4), ses modifications peuvent être répercutées immédiatement et à peu de frais sur l'image affichée. Une telle organisation du processus de visualisation peut correspondre à un système à balayage de trame, utilisant des tables de couleur ("look-up tables"). La mémoire d'image contient l'information synthétisée qui pour chaque pixel est une adresse dans la table des couleurs. La couleur est ainsi prise en compte au moment de l'affichage (à chaque cycle de raffraichissement), et toute modification dans la table est immédiatement répercutée sur l'image.

4 CARACTERISATION DES SYSTEMES DE SYNTHESE D'IMAGES

La portée d'un système de synthèse d'images est, bien entendu, fonction du nombre d'éléments différents qu'il est capable d'accepter. Cependant pour le caractériser il faut également considerer l'ensemble des processus de base proposés pour la manipulation des attributs de ces éléments.

Ainsi nous dirons qu'un système de synthèse d'images est :

- **cohérent** si tous les processus qu'il propose sont définis pour tous les éléments traités et applicables au même(s) niveau(x) de structure;

- **symétrique** si tous les attributs de tous les éléments considérés peuvent être entièrement décrits de façon interactive;

- **ouvert** si tous les attributs décrits interactivement peuvent être consultés par le programme d'application.

Un système "maximum" réunira ces trois propriétés. En particulier il devra fournir un ensemble complet de processus pour l'attribution, la consultation et la description de tous les attributs de chacun des types d'éléments supportés.

Par ailleurs le système devra proposer un ensemble de processus de visualisation concernant tous les éléments permettant de choisir différents types de représentation selon les possibilités du matériel et les performances requises.

5 CONCLUSION

Dans ce chapître nous avons tenté de dégager les principales fonctions que doit assurer un système de synthèse d'images, ainsi que les caractéristiques des informations qu'il est appelé à manipuler. Tout au long de notre étude, nous nous servirons de ces concepts de base afin de tenter d'avoir une approche "unifiée" pour les systèmes de visualisation graphique interactifs.

Le rôle **essentiel** du système de synthèse d'images est de permettre aux applications de mettre en oeuvre **facilement** et de manière **efficace** les différents processus de base que sont l'ATTRIBUTION, la CONSULTATION, la VISUALISATION et la DESCRIPTION. Cependant, bien que nous n'ayons que peu insisté sur ces points, il est important de bien mesurer la compléxité de ces processus (en particulier du processus de visualisation) ainsi que la quantité d'informations qu'ils manipulent. C'est pourquoi, il nous parait souhaitable que le système de synthèse d'images assure au maximum la gestion de ces différents processus afin d'en faciliter l'utilisation et la compréhension par les programmes d'application. C'est un tel système que nous avons tenté de concevoir au cours de nos travaux, et que nous présenterons dans les seconde et troisième parties de ce rapport.

Chapître II : Différentes Organisations des Systèmes Graphiques - Exemples de Systèmes de Visualisation Généraux

1 INTRODUCTION

Durant la première période de dévelopement du graphique (1950-1960) des dispositifs de visualisation entièrement passifs étaient utilisés pour générer des images sans aucune interactivité [Ever 52]. Les interfaces logicielles alors développées se limitaient à de simples controleurs ("drivers") conçus pour piloter directement ces dispositifs et offrant les fonctions primitives du terminal.

Mais au début des années soixante l'apparition de terminaux de plus en plus **intelligents** et de dispositifs **interactifs** a nécessité la création de logiciels beaucoup plus élaborés pour permettre aux programmes d'application d'accéder à ces nouvelles possibilités. A la place de simples logiciels de pilotage du matériel de visualisation sont apparus des **systèmes graphiques** incorporant de nombreuses fonctions aditionnelles plus complexes, telles par exemple les transformations géométriques, la structuration de l'image et l'entrée graphique... On citera à ce titre le système SKETCHPAD développé par Sutherland en 1963 [Suth 63] et qui est souvent considéré comme le pionnier en ce domaine.

La réalisation de telles interfaces est une tâche difficile. Il s'agit en effet de systèmes complexes devant réaliser un équilibre difficile entre les ressources matérielles et logicielles disponibles.

La très grande diversité du matériel graphique (en particulier la prolifération de terminaux de technologies et de fonctionnalités très différentes) et l'évolution constante de celui-ci rend la définition d'une architecture logicielle et materielle pour les systèmes graphiques très difficile et a entrainé une approche cahotique et souvent innacordée du matériel et du logiciel.

Dans les paragraphes suivants, nous nous interesserons à l'architecture (organisation) de ces systèmes en mettant plus particulièrement l'accent sur les problèmes logiciels rencontrés. Ainsi nous présenterons les implications des différentes stratégies logicielles tant du point de vue des performances du système final que de son efficacité, de ses coûts de développement et de sa longévité. Nous nous attacherons ensuite aux problèmes plus particulièrement liés à la conception de systèmes graphiques généraux qui fournissent aux applications une interface uniforme indépendante du matériel et de l'implémentation. Nous terminerons notre exposé, par l'étude de quelques logiciels caractéristiques de cette approche, en essayant de dégager leurs fonctionnalités relativement aux concepts de base que nous avons développés au cours du premier chapître.

2 LES DIFFERENTES APPROCHES POUR LE DEVELOPPEMENT D'UN LOGICIEL GRAPHIQUE

2.1 Systèmes graphiques spécialisés

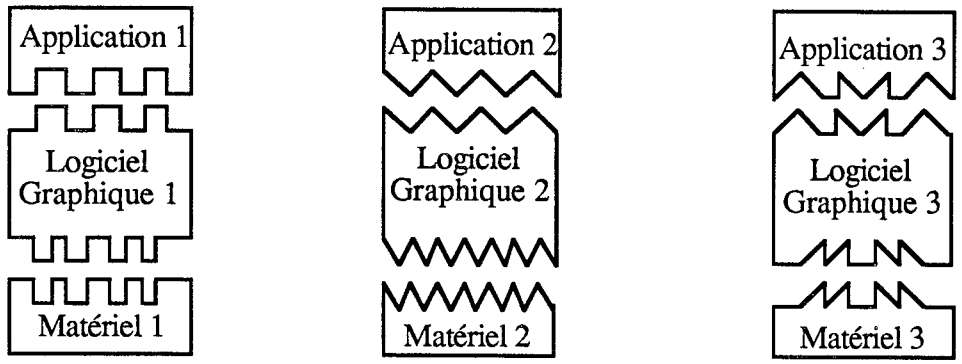
2.1.1 Présentation générale

Devant la très grande diversité du matériel graphique (en entrée (clavier, tablette, souris, boule roulante....) et en sortie (tubes à rayon cathodique à balayage cavalier, à balayage de trame, tables traçantes...), avec des possibilités au premier abord incompatibles, l'approche logicielle la plus immédiate est de concevoir un logiciel graphique spécifiquement adapté à une architecture matérielle donnée.

Ces logiciels que nous désignerons sous le terme de **systèmes graphiques spécialisés**, outre leur dépendance par rapport au matériel graphique sont aussi en général

dépendants du type d'application considéré, et même de leur environnement informatique (calculateur hôte, système d'exploitation).

Les particularités des équipements (matériels) utilisés ainsi que des applications concernées apparaissent au niveau de la structure interne et de l'interface fonctionnelle de ces systèmes (fig. II.1). Ceux-ci sont donc très différents les uns des autres, chacun possédant une interface qui lui est propre.



- figure II.1 : Systèmes graphiques spécialisés (d'après [Newm 78]) -

2.1.2 Avantages

Les systèmes graphiques spécialisés, parcequ'ils permettent de prendre en compte au mieux les possibilités du matériel et les besoins de l'application, offrent un niveau de performances remarquable (rapidité d'exécution, rapidité de l'affichage, haute interactivité).

En fait lors de la conception d'un tel système la situation idéale consiste à choisir la configuration matérielle la mieux adaptée aux besoins spécifiques de l'application avant tout développement du logiciel.

Cependant la possiblité de hautes performances présente en contre-partie de nombreux inconvénients dont nous exposons les principaux dans le paragraphe qui suit.

2.1.3 Inconvénients

Les systèmes graphiques spécialisés nécessitent un **développement logiciel** important qui peut s'avérer **long** et **excessivement couteux**. En effet, chaque système ayant ses propres caractéristiques, il est très difficile de tirer directement profit de systèmes précedemment développés pour la prise en compte d'une nouvelle application ou d'un nouveau matériel graphique. Souvent, la réécriture d'une partie, voire de tout le logiciel est indispensable. Cela nécessite du personnel hautement qualifié possédant de solides compétences en graphique et qui devra à chaque fois réapprendre et/ou s'adapter au nouveau matériel. D'autre part, la difficulté à trouver ou former des programmeurs ayant ces compétences, et cette impossibilité à pouvoir réutiliser complétement les travaux antérieurs et l'expérience acquise peuvent également nuire à la **qualité** des logiciels.

A cela il faut ajouter les problèmes liés à **l'évolution** et la **longévité** de tels systèmes. Vu la difficulté de prendre en compte tout nouveau matériel, le système graphique risque de devenir obsolète avec l'apparition sur le marché de nouveaux dispositifs plus performants. De même toute **modification** de l'environnement informatique doit être considérée avec attention, au risque sinon de voir bon nombre d'applications graphiques devenir **inutilisables**.

Nous exposons brièvement dans ce qui suit, la répercussion de ces divers inconvénients auprès des différents membres de la "communauté graphique" (voire de la "communauté informatique" en général) concernés par la conception, le développement et l'utilisation de systèmes graphiques.

Pour les ***utilisateurs*** , outre leur coût très élevé, de tels systèmes spécialisés entraînent un certain nombre de risques et incertitudes dans le développement et la gestion de leurs projets (problèmes de la longévité et de l'évolution du système que nous évoqués précédemment). Par ailleurs, les utilisateurs peuvent se trouver liés à un constructeur, voire à une gamme précise de matériel (et ceci aussi bien pour les dispositifs graphiques que pour le calculateur hôte et son environnement), ce qui n'est pas toujours économiquement souhaitable.

Pour les ***développeurs*** de logiciel (si ce ne sont pas l'utilisateurs finaux), les contraintes précédentes (coûts très élévés, dépendance) limitent le marché potentiel pour leurs produits. Ils doivent donc, soit se concentrer sur une gamme particulière de matériels en "espérant" qu'elle restera populaire, soit essayer de s'adapter au mieux à la plus grande variété de matériels possibles afin de pouvoir faire évoluer leur système lorsque de nouveaux dispositifs plus attractifs pénètrent sur le marché.

Pour les ***constructeurs*** de matériel les conséquences sont similaires (que ce soit les constructeurs de matériel purement graphique ou dans une moindre mesure les constructeurs de calculateurs hôtes). Vendre une machine à un acheteur potentiel est beaucoup plus difficile si une partie de ses applications devient caduque. Bien souvent, les utilisateurs préfèreront conserver leurs applications qui tournent avec un matériel moins performant plutôt que de faire en plus de l'achat du matériel un investissement logiciel important et parfois hasardeux.

2.2 Systèmes graphiques généraux

Historiquement, les premières applications graphiques furent typiquement réalisées à partir de logiciels spécialisés. Mais, si celles-ci ont alors convaincu beaucoup de monde quand à l'utilité et l'efficacité de l'infographie, leur développement resta relativement isolé du fait des inconvénients liés au développement de systèmes graphiques spécialisés .

Cependant, au début des années 70, l'apparition de matériel à un coût moindre, en particulier les consoles à tube mémoire, a entraîné une certaine "démocratisation" du graphique [Prei 78]. Et ce qui était encore acceptable pour une communauté réduite, utilisant du matériel hautement sophistiqué et de prix élevé devenait de moins en moins supportable pour des utilisateurs de plus en plus nombreux. C'est à cette époque que l'on a commencé à développer des logiciels graphiques de base destinés aux programmeurs d'application et que l'on s'est tourné vers la conception de **systèmes graphiques généraux** [HaHe 82].

2.2.1 présentation générale

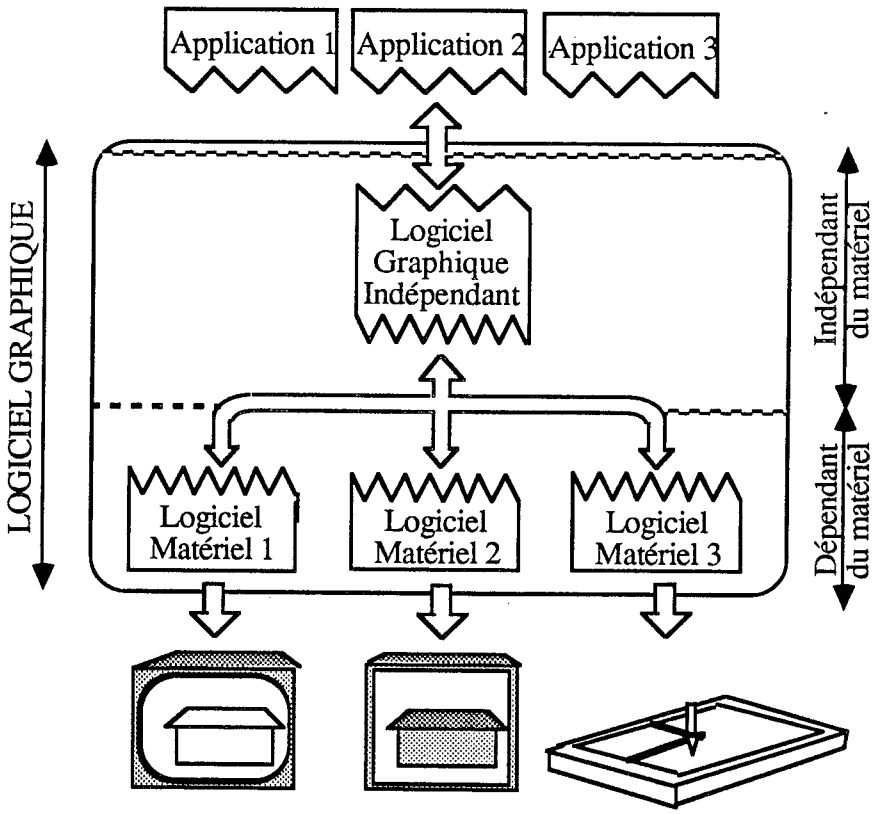
Un système graphique général est **indépendant** de son contexte d'utilisation qui peut être défini par l'ensemble des applications concernées, les matériels graphiques et l'environnement informatique considérés [LLMa 78].

L'***indépendance par rapport à l'application*** signifie que le système graphique propose une interface générale sans présupposition aucune sur le type d'application considéré. Le système graphique doit pouvoir ainsi permettre le développement de toute une **gamme** d'applications.

L'***indépendance par rapport à l'environnement informatique*** signifie que les programmes d'application doivent pouvoir être transportés d'un calculateur hôte à un autre, d'un système d'exploitation à un autre avec des changements minimum.

Complément indispensable de l'indépendance par rapport à l'environnement, ***l'indépendance par rapport au matériel*** signifie qu'un même programme d'application doit pouvoir utiliser des dispositifs de visualisation différents (d'une table traçante à une console CRT à balayage de trame) sans avoir à connaître leurs caractéristiques spécifiques. Il faut que le système graphique soit capable de générer des images les plus similaires possibles sur des dispositifs d'affichage différents sans provoquer de changements dans la structure du programme d'application [Newm 78]. De manière analogue l'indépendance par rapport aux dispositifs d'affichage est étendue aux dispositifs de dialogue (souris, tablette, réticule etc...) que l'application doit pouvoir utiliser indifféremment, sans se soucier de leurs particularités physiques.

L'indépendance par rapport au contexte d'utilisation peut être obtenue en réalisant un sytème graphique qui fourni un ensemble de fonctions de haut niveau accessibles au travers d'une interface identique sur chacune des installations graphiques. Les fonctionnalités qui sont proposées doivent, bien entendu, pouvoir être utilisées dans les divers programmmes d'application indépendemment des caractéristiques précises des différents matériels graphiques considérés.



- figure II.2 : Organisation d'un Système Graphique Général -

Pour assurer cette indépendance par rapport au matériel, on distingue à l'intérieur du système graphique un second niveau d'interface entre la partie générale du logiciel

indépendante du matériel et la partie spécifique qui permet de piloter les dispositifs particuliers. La prise en compte d'un nouveau matériel nécessite alors uniquement l'écriture du module le pilotant en respectant la définition de cette interface.

La figure II.2 ci-contre, schématise cette organisation sur laquelle nous reviendrons plus en détail par la suite.

2.2.2 Avantages

La supériorité des systèmes graphiques généraux par rapport aux systèmes graphiques spécialisés pourrait se résumer par un seul mot : **la portabilité** [Ende 85].

2.2.2.1 portabilité des programmes d'application

L'indépendance du système graphique par rapport à son environnement informatique permet la portabilité des programmes d'application d'un calculateur hôte à un autre, d'un système d'exploitation à un autre. Ceci diminue la dépendance éventuelle de l'*utilisateur* par rapport à un constructeur d'ordinateurs et ralentit l'obsolensce des logiciels graphiques qui peuvent ainsi s'éxecuter sur des générations successives de machines. Elle accroit la diffusion des systèmes graphiques qui peuvent toucher un public potentiel beaucoup plus vaste, au bénéfice des *développeurs de logiciels* et indirectement des *constructeurs de calculateurs* (en effet le succès commercial d'une machine est de plus en plus influencé par l'étendue des logiciels disponibles sur celle-ci). D'autre part, il est alors possible d'envisager l'exécution d'applications au travers d'un réseau local composé de différentes machines de marques différentes [WTNe 84].

Un point secondaire mais qui n'est tout de même pas négligeable, est également la possibilité pour les developpeurs d'applications graphiques d'utiliser des outils de développement logiciel largement diffusés. Cela peut entrainer une diminution des coûts de développement en employant des programmeurs expérimentés connaissant ces outils et capable de les employer efficacement.

2.2.2.2 portabilité de l'affichage

L'indépendance du système graphique par rapport aux dispositifs de visualisation autorise la **portabilité de l'affichage** qui complète la portabilité des applications et offre des avantages analogues. Ainsi elle libère dans une certaine mesure les *utilisateurs* de la dépendance par rapport aux *constructeurs de matériel graphique*.. Ces derniers voient s'accroitre le nombre d'applications tournant sur leurs produits et inversement les *developpeurs de logiciels* peuvent proposer leurs applications sur de nombreux matériels.

Par ailleurs, l'emploi de dispositifs graphiques différents peut influer sur la qualité des applications. Par exemple, il est envisageable de méler l'utilisation d'un terminal à tube mémoire pour une visualisation rapide du projet en cours d'élaboration avec l'utilisation d'une table traçante pour une sortie soignée du produit final.

De maniére complémentaire, la portabilité de l'affichage peut permettre aux options matérielles de rester plus longtemps ouvertes durant le développement d'un produit. Un système peut ainsi être "prototypé" sur plusieurs dispositifs et ensuite être optimisé pour le ou les matériels finalement choisis pour sa commercialisation.

2.2.2.3 portabilité de l'information graphique

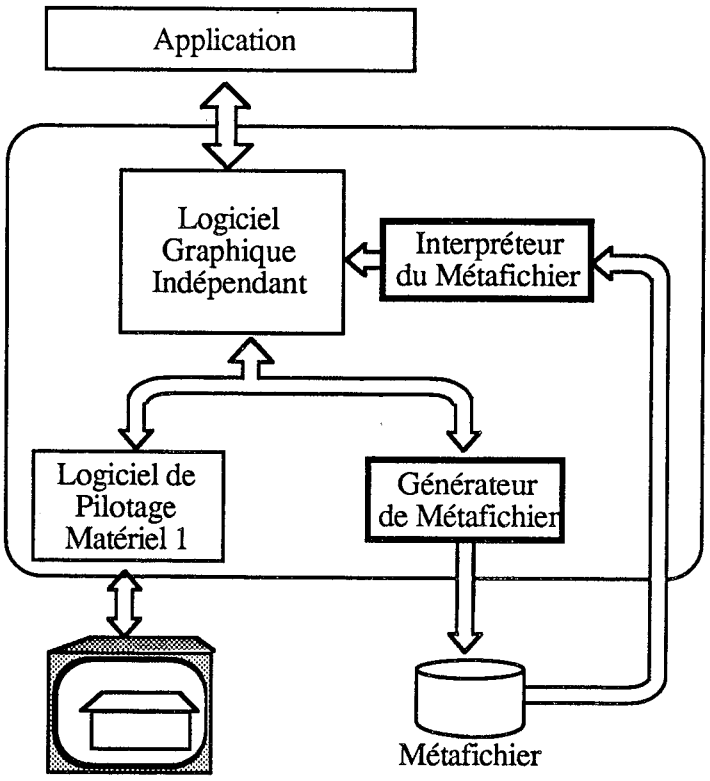
Une conséquence directe de la portabilité de l'affichage est la portabilité de l'information graphique. En effet, dans un système graphique général, il est possible de récupérer une représentation des images indépendante du matériel, au niveau de l'interface entre la partie logicielle générale et la partie logicielle spécifique pilotant les dispositifs de visualisation particuliers (voir fig. II.3). Une telle réprésentation, qu'il est alors possible de conserver sur des mémoires de masse, est en général désignée sous le terme de métafichier graphique ("metafile").

Les avantages offerts par cette possibilité sont les suivants :

- elle permet le stockage sur des mémoires de masse d'images qui peuvent ainsi être réutilisées par la suite (par exemple, une image peut être aisément conservée entre deux séances de travail).

- elle autorise l'échange d'images entre plusieurs sites, soit par une transmission au travers de réseaux téléinformatiques, soit plus simplement par un transport sur un médium de stockage (bandes ou disques magnétiques par exemple). Cette possibilité de stocker et d'échanger de l'information graphique entre plusieurs sites peut ainsi permettre le partage et l'utilisation efficace d'équipements onéreux.

- elle peut éventuellement servir dé lien entre des sytèmes graphiques différents (si l'interface au niveau de la partie dépendante du matériel est identique), les images produites par ces systèmes pouvant alors être regroupées en une représentation uniforme



- figure II.3 : Métafichier pour l'archivage d'images -

2.2.2.4 portabilité des programmeurs ou portabilité de l'éducation

Conséquence de la portabilité de l'affichage et de l'indépendance du sytème graphique vis à vis des applications, la "portabilité de l'éducation" réduit le problème de "l'apprentissage" ou de "l'entrainement" des programmeurs qui ne sont plus directement concernés par les caractéristiques particulières des dispositifs graphiques.

Ainsi l'utilisation judicieuse d'un logiciel graphique "près à être exécuté" (ready to run), testé et validé lors de réalisations précédentes permet de fournir plus rapidement un produit sur le marché, avec un coût de développement moindre et du personnel moins hautement qualifié que pour des logiciels spécifiques.

2.2.3 Inconvénients

Il faut cependant nuancer le tableau "idyllique" dressé ci-dessus. La portabilité est certes la solution à bien des problèmes rencontrés par les concepteurs et les utilisateurs de systèmes graphiques et présente de nombreux avantages surtout du point de vue économique, mais elle se paye en contrepartie par un **degré de performance** bien évidemment **moindre** que pour les systèmes graphiques spécialisés.

En effet un système graphique général doit s'adresser au plus d'applications, de machines et de dispositifs graphiques possibles. Il ne saurait donc être complétement approprié et optimum pour aucun d'entre eux. En particulier l'indépendance par rapport au matériel graphique est réalisée au détriment des performances du système. En effet, étant donnée l'extrême diversité du matériel, il est clair que la prise en compte simultanée de plusieurs dispositifs ne permet pas toujours de tirer parti des spécificités de chacun.

Il s'agit donc lors de la conception d'un système graphique général de réaliser un **équilibre** difficile entre la **généralité** et les **performances**. Ce bon équilibre étant comme nous le verrons par la suite, conditionné par une définition rigoureuse des divers niveaux d'interface présentés par ces systèmes, en particulier entre le partie du logiciel graphique indépendante et la partie dépendante du matériel.

Un second problème lié au point précédent et celui de l'**évolution** possible de tels systèmes avec l'apparition de nouveaux matériels de plus en plus performants. L'architecture du système risque d'être remise en cause par ceux-ci, à moins de ne pas prendre en compte toutes leurs possibilités. Ce problème n'est pas négligeable quand on constate l'intégration de plus en plus forte de l'électronique, la très rapide augmentation des perfomances des matériels qui prennent de plus en plus en charge des fonctionnalités qui relevaient auparavant du logiciel (par exemple le Geometry Engine des stations IRIS de Silicon-Graphics qui effectue tous les calculs de géometrie 3D, transformations, coupage, projection [Myer 84],[Will 85]), le tout accompagné d'une forte diminution des prix.

2.3 Conclusion

L'apparition des sytèmes graphiques généraux a été une étape très importante dans l'évolution de l'infographie. En effet elle a permis le ressèrement des liens de la "communauté graphique" après une période initiale de développement quasi anarchique où chaque utilisateur construisait indépendamment son propre système. De plus, elle a également contribué, parallélement aux progrès et à la baisse des prix du matériel, à une meilleure "diffusion" du graphique grâce à la portabilité des logiciels.

Les travaux effectués sur les systèmes graphiques généraux ont été très importants. Très rapidement les différents partenaires se sont rendus compte de l'intérêt de ceux-ci, et de

nombreux efforts on été faits pour tenter d'élaborer des **standards**. En effet les avantages de ces systèmes ne peuvent être réellement perçus que si des interfaces communes existent.

Aussi les différents organismes de normalisation nationnaux (ANSI,AFNOR,DIN...)[1] et internationnaux (ISO)[2] ont-ils créés des groupes de travail pour la standardisation des différentes interfaces des systèmes graphiques (interface avec l'application, interface entre partie dépendante du matériel et partie indépendante). Ceci est un travail de longue haleine, à la fois de par les processus adoptés par les organismes de standardisation [Zimm 87], mais également à cause de la difficulté à s'accorder sur les niveaux de fonctionnalité de ces systèmes et également, dans une certaine mesure, des conflits d'intérêt qu'un tel enjeu peu suciter [Mead 84].

A l'heure actuelle un ensemble de standards semble se dégager, il s'agit de GKS ("Graphic Kernel System") et PHIGS ("Programmers' Hierarchical Interactive Graphic Sysytem") en ce qui concerne l'interface avec l'application et CGI (Computer Graphic Interface) et CGM (Computer Graphic Metafile) en ce qui concerne l'interface avec la partie dépendante du matériel. Nous reviendrons dans les paragraphes et le chapitre suivant sur ces systèmes.

Mais, malgré les nombreuses qualités des systèmes généraux, les systèmes graphiques spécialisés restent encore indispensables pour des applications qui necessitent de hautes performances et une grande interactivité (par exemple pour des simulations temps réel).

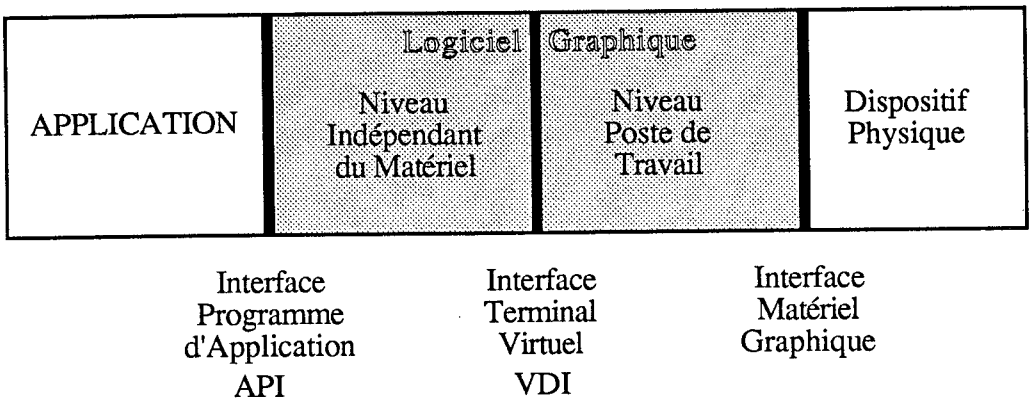
Cependant les "risques" liés au développement de systèmes spécialisés doivent inspirer une certaine prudence. Comme le remarque J. Warner [Warn 85] : *"Les concepteurs (de systèmes graphiques) doivent soigneusement étudier l'équilibre performances/indépendance, et doivent trouver si possible un moyen de conserver un minimum d'indépendance par rapport au matériel."* Une solution d'avenir est peut être la conception d'applications de haut niveau ("high end applications") à partir de systèmes graphiques généraux standards, quitte ensuite à leur rajouter des extensions permettant d'utiliser de façon optimale les spécificités des dispositifs sélectionnés.

1 - ANSI American Standards Institute,
- AFNOR Association Française pour la NORmalisation,
- DIN Deutches Institut für Normung.

2 - ISO International Standard Office

3 ORGANISATION DES SYSTEMES GRAPHIQUES GENERAUX

Dans cette partie nous allons tenter de mettre en évidence les problèmes techniques que soulève la conception de sytèmes graphiques généraux. Comme nous l'avons déjà évoqué, la principale difficulté réside dans le partage des tâches entre les différents composants du système (programme d'application, logiciel graphique indépendant du matériel, logiciel graphique dépendant et le matériel (voir fig. II.4)).



- figure II.4 : Les différents niveaux et interfaces d'un système graphique général -

Il s'agit donc de définir clairement les fonctionnalités de chacun des niveaux d'interface reliant les différentes couches de logiciel, en particulier, l'interface entre l'application et le système graphique que l'on désigne parfois par API (de l'anglais "Application Programmer Interface") et l'interface terminal virtuel ou VDI ("Virtual Device Interface") entre la partie du logiciel graphique indépendante et celle dépendante du matériel.

3.1 Interface avec le programme d'application

Le rôle de l'interface avec le programme d'application est de masquer au programmeur les "idiosyncrasies" du matériel graphique particulier qu'il utilise tout lui en permettant de développer la plus grande diversité d'applications possibles.

La difficulté consiste donc à décider de ce qui doit être inclus ou non dans le système graphique, ceci en tenant compte des besoins des diverses applications et des possibilités des matériels considérés.

3.1.1 Modélisation et Visualisation

Comme nous l'avons evoqué au chapitre I, deux grandes activités se mélent à l'intérieur d'une application graphique (au sens le plus large du terme c'est à dire le programme d'application proprement dit et l'ensemble des logiciels qu'il utilise (en particulier le système graphique)) :

- la **modélisation** qui consiste à décrire et construire une représentation (modèle) des objets que l'application manipule, et qui peuvent ensuite être affichés. La modélisation peut prendre des formes multiples selon les besoins propres de chaque application. Cependant dans l'optique d'un système graphique général il est essentiel que la description du modèle puisse s'effectuer indépendamment des caractéristiques des dispositifs graphiques. En particulier les coordonnées définissant la morphologie des objets et les éventuelles transformations géométriques permettant leur positionnement doivent pouvoir être exprimées dans un référentiel (système de coordonnées) approprié aux besoins propres de l'application sans qu'elle ait à se préoccuper des particularités d'adressage des dispositifs de visualisation utilisés.

Ce système de coordonnées dont l'unité est fixée au gré de l'application (m, cm, µ...) est communément désigné par le terme d'**espace utilisateur** ou **espace universel** ("World Coordinate Space") par opposition à l'**espace écran** ("Device Coordinate Space") qui lui, identifie le système de coordonnées propre à chaque dispositif d'affichage .

- la **visualisation** qui transforme les données graphiques contenues dans le modèle en une représentation acceptable par le ou les dispositifs d'affichage afin de produire une image. En particulier, elle doit appliquer ce que l'on appelle la **transformation de visualisation** ("Viewing Transformation") aux coordonnées universelles définissant les objets à l'intérieur du modèle afin d'aboutir aux coordonnées écran.

La transformation de visualisation est de nature purement géométrique et est définie par les paramètres de géométrie de prise de vue (Gv : qui indiquent quelle partie de l'espace utilisateur doit être affichée et la manière dont elle est observée) et les paramètres de géométrie d'affichage (Ga : qui indiquent où doit être affichée l'image obtenue). Elle est appliquée à chacun des éléments acceptés par le processus de visualisation et est réalisée en plusieurs étapes successives formant un véritable "pipe line" de visualisation.

Pour la visualisation de scènes bidimensionnelles (dans le plan 2D) la transformation de visualisation est définie de manière typique par la donnée d'une fenêtre ("Window" portion rectangulaire de l'espace utilisateur) et d'une cloture ("Viewport" portion rectangulaire de l'espace écran) et comprend les opérations suivantes :

- le coupage ("clipping") qui permet de rejeter les éléments ou les parties d'éléments qui ne se trouvent pas dans la fenêtre et ne doivent donc pas être visualisées,

- la transformation ("mapping") fenêtre/cloture qui permet de passer de l'espace utilisateur à l'espace écran (homthétie suivie d'une translation).

Pour l'affichage d'objets tridimensionnels (objets 3D) la transformation de visualisation est plus complexe car elle implique une projection de l'espace utilisateur 3D vers l'espace 2D du dispositif de visualisation. Pour une visualisation réaliste sous forme perspective, les paramètres de prise de vue définissant cette transformation sont définis de manière classique par la donnée d'un point de vue (centre de projection), d'une direction de visée, d'un plan de projection et d'une fenêtre sur ce plan de projection (qui défini l'ouverture du cône de vision) (fig. II.5.a) [CaPi 78]. Tous ces paramètres sont bien entendu exprimés dans l'espace utilisateur.

Les différentes étapes de la transformation de visualisation sont alors les suivantes (fig. II.5.b et II.5.c) :

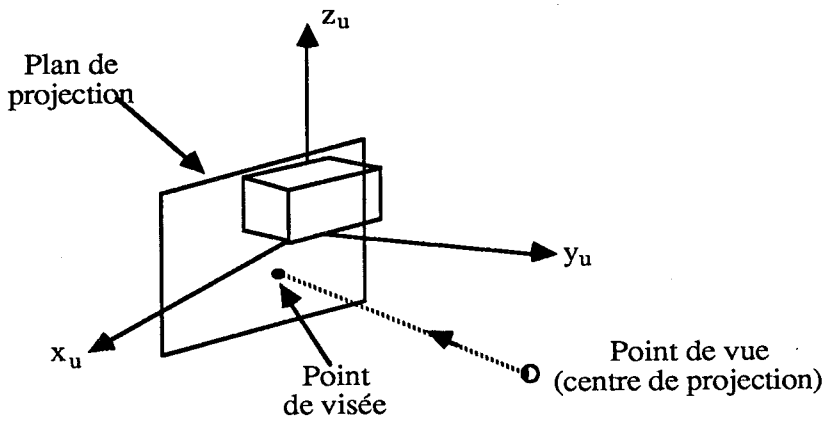
- transformation des coordonnées universelles en coordonnées définies dans le repère de visualisation (repère défini par le centre de projection et la direction de visée)

- coupage des éléments par rapport au cône de vision défini par les plans passant par le centre de projection et s'appuyant sur la fenêtre dans le plan de projection. Il est possible éventuellement de leur adjoindre un plan de coupage avant et un plan de coupage arrière parallèles au plan de projection.

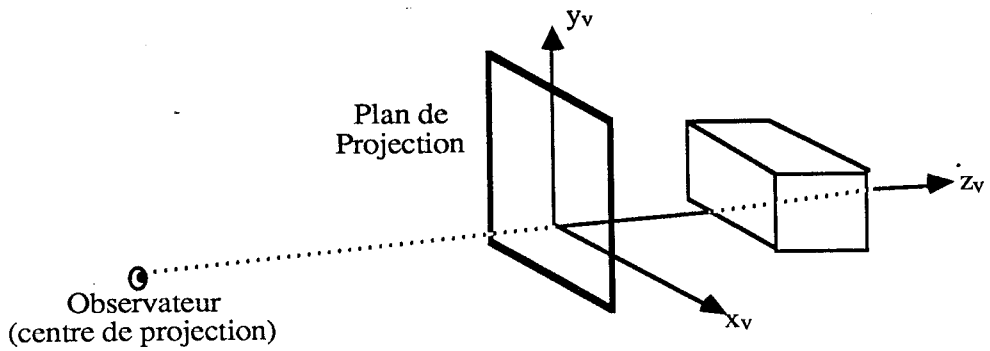
- projection proprement dite qui consite à passer de coordonnées 3D en coordonnées 2D dans le plan de projection en tenant compte de la déformation à apporter aux éléments en fonction de leur éloignement du point de vue,

- et finalement transformation de ces coordonnées projetées en coordonnées écran (analogue à la transformation fenêtre cloture du 2D).

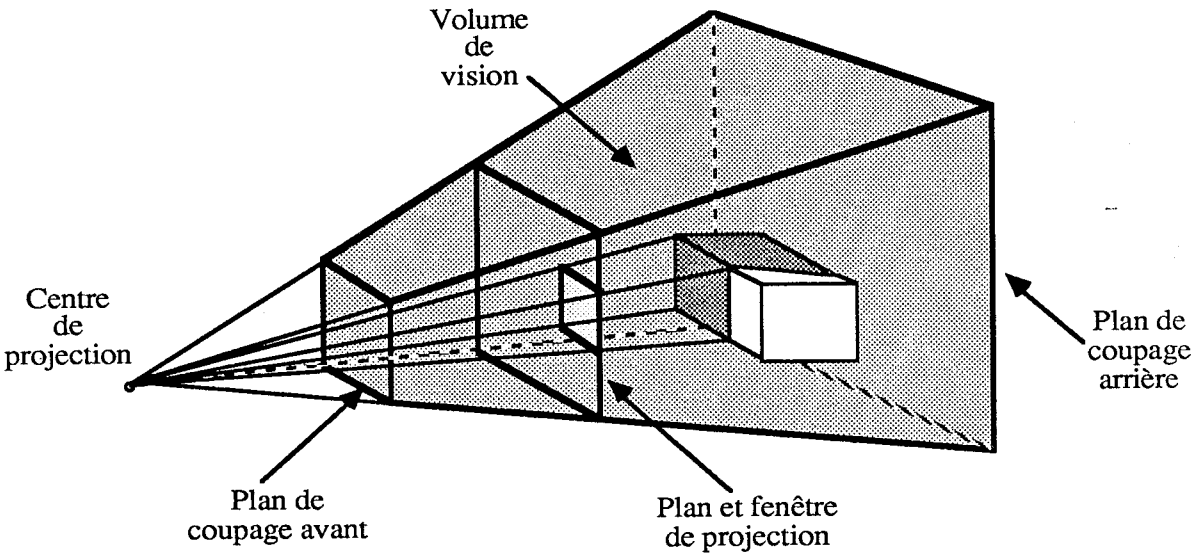
A ces opérations de transformation géométrique de coordonnées, il faut éventuellement ajouter l'application d'un algorithme d'élimination des parties cachées qui peut s'effectuer selon les cas dans l'espace écran ou dans l'espace utilisateur [SSSc 74].



a) définition de la prise de vue dans le repère utilisateur (O_u ,x_u ,y_u ,z_u)



b) passage dans le repère de visualisation (de projection) (O_v ,x_v ,y_v ,z_v)



c) coupage par rapport au volume de vision et projection

- figure II.5 : Les différentes étapes de la prise de vue 3D -

3.1.2 Séparation Modélisation Visualisation

Une date clé, à laquelle on fait souvent référence en infographie, est Mai 1976 où s'est tenu en France, à SEILLAC, une réunion d'un groupe de travail de l'IFIP[1] (IFIP WG5.2) s'interessant à la définition d'une méthodologie pour l'élaboration de systèmes et standards graphiques [GuTu 76]. L'une des décisions fondamentales prise au cours de ce séminaire, et qui orienta dans une large mesure les travaux ultérieurs effectués sur les systèmes graphiques généraux fût de **considérer comme deux systèmes séparés la visualisation et la modélisation.** Le système graphique devant prendre en charge uniquement la visualisation, la modélisation étant sous la responsabilité de l'application.

Les partisans de ce choix l'argumentèrent par le fait que de nombreux types différents de modélisation étaient nécessaires pour répondre aux différents besoins des applications et qu'il ne pouvait donc être envisageable d'intégrer cette tâche au système graphique général. Par ailleurs à l'époque de ce débat, il semblait également préférable de concentrer les efforts sur les problémes liés à la visualisation qui n'étaient pas encore tous pleinement perçus et ne faisaient pas l'objet d'un concensus général.

Avec un tel modèle, le rôle du système graphique se réduit essentiellement à permettre aux applications de définir la transformation de visualisation appropriée et de générer l'image en utilisant la description du (des) objet(s) en coordonnées universelles (ceci étant typiquement effectué en appelant des fonctions élémentaires de tracé avec des arguments en coordonnées universelles).

Les principales fonctions du système graphique sont alors :

- des fonctions de définition de la transformation de visualisation, (géométrie de prise de vue et géométrie d'affichage),

- des primitives graphiques qui permettent de définir le tracé d'objet élémentaires (points, segments, textes..),

- des primitives permettant de définir l'aspect de ces objets élémentaires,

- éventuellement des fonctions de segmentation de la liste d'affichage, qui permettent de mémoriser une description "structurée" de l'image, autorisant ainsi certaines modifications de l'image sans que l'application ait à redemander la visualisation de tous les éléments composant les objets à afficher,

- des fonctions d'entrée, qui permettent à l'application d'utiliser les différents dispositifs de dialogue,

- des fonctions de contrôle qui permettent de consulter l'état du système graphique.

3.1.3 Implémentation de l'interface avec le programme d'application - API

Pour fournir à un programmeur d'application les primitives graphiques lui permettant de contrôler de manière indépendante (du matériel graphique et de l'environnement informatique) la génération d'images et les fonctions d'interaction **trois stratégies** de base sont possibles :

1 IFIP : International Federation of Information Processing

- la conception d'un "**langage graphique**",

- l'**extension** d'un langage de haut niveau déjà existant,

- l'élaboration d'une **bibliothèque** de sous programmes ("subroutine package") pouvant être appelés à partir de langages de programmation existant.

3.1.3.1 Langage graphique

Cette solution permet de proposer des langages spécialisés adaptés à des utilisateurs dont les connaissances et l'intérêt ne sont pas l'informatique (par exemple des artistes). En effet le développeur n'est pas contraint de travailler avec la structure et la philosophie de langages déjà existant. (A ce titre, le graphisme "tortue" du langage LOGO de A. Peperts, est un bon exemple d'outil simple utilisable par des personnes non expérimentées pour des opérations de visualisation élémentaires...)

Cependant, créer un nouveau langage s'avère être une tâche très ardue. Aux difficultés de trouver un accord sur les objectifs et fonctionnalités qu'il doit présenter, s'ajoute le problème de crédibilité qu'affronte tout nouveau langage. Aussi, peu de ces langages ont dépassé le stade expérimental. Parmi ceux-ci, on peut néanmoins citer POSTSCRIPT [Post 84]. Orienté vers la mise en page de documents alliant du texte et du graphique, il tend à devenir un "standard" dans le domaine de l'impression (C'est, par exemple, le langage utilisé par les imprimantes à laser d'Apple et de Cannon).

Par ailleurs, dans la plupart des cas la puissance d'un langage de programmation complet est indispensable. Aussi, a-t-on bien souvent préféré se déporter vers une solution intermédiaire, moins onéreuse et risquée, qui consiste à adjoindre des extensions graphiques à des langages de programmation déjà existant.

3.1.3.2 Extension à des langages de programmation existants

Cette seconde solution qui consiste à augmenter un langage de programmation de haut niveau en lui ajoutant des instructions graphiques, peut être réalisée de deux manières différentes :

- **en modifiant le compilateur** afin d'accepter directement les extensions (par exemple GPL 1 extension graphique pour le langage PL1 [SMIT 71]). Mais il s'agit tout de même d'une solution coûteuse et difficile à mettre en oeuvre, sans parler des problèmes rencontrés pour maintenir de tels compilateurs. (Aussi cette formule n'a t'elle été adoptée initialement que par de puissants groupes de recherche et développement).

- la seconde solution qui présente moins de problèmes d'implémentation est la définition et construction de **préprocesseurs** qui reconnaissent les constructions graphiques particulières et les convertissent en instructions du langage hôte (par exemple extensions au langage FORTRAN chez BOEING [Hurw 67] dans les années 70).

Nous pouvons également citer MIRA-2D et MIRA-3D, extensions graphiques au langage PASCAL basée sur l'utilisation de types graphiques abstraits et développées audépartemnt d'informatique de l'université de Montréal [MaTa 82].

Cette solution offre une interface plus satisfaisante que l'appel à des sous programmes. A la plus grande lisibilité du code il faut ajouter la possibilité de détecter un certain nombre

d'erreurs à la compilation (ou lors du passage dans le préprocesseur). De plus l'inclusion du graphique offre à plus long terme une meilleure garantie de transportabilité si le langage étendu est largement diffusé.

Si initialement la solution langage a été peu souvent employée et se limitait à des groupes de recherche ou développement de part sa relative difficulté de mise en oeuvre, on assiste actuellement à une évolution due aux fulgurants progrès de la micro informatique et à une certaine démocratisation du graphique. Et l'on voit de nouvelles versions de langage intégrant des instructions graphiques (BASIC, TURBO-PASCAL) qui s'imposent de plus en plus comme des standards de facto. Toutefois, il s'agit encore d'un graphique aux performances faibles et s'adressant à un matériel de bas niveau, ce qui ne saurait répondre entièrement aux besoins d'applications complexes.

3.1.3.3 Bibliothèques de Sous Programmes ("Packages" Graphiques)

Il s'agit de l'approche qui vient le plus rapidement à l'esprit lorsque l'on parle de logiciel graphique. Elle consiste en l'élaboration d'une bibliothèque de sous programmes accessibles à partir d'un langage de programmation.

La raison principale du succès de cette approche consiste en sa facilité de mise en oeuvre, puisqu'elle ne nécessite aucune modification du compilateur ou du langage et affecte uniquement l'environnement d'exécution ("run time environment"). Cela permet de développer et modifier éventuellement le système graphique beaucoup plus facilement que pour la solution langage vue précédemment.

Parmi les reproches que l'on peut faire à cette méthode, on peut regretter une syntaxe utilisant uniquement des listes de paramètres qui n'est pas toujours très attractive. Par ailleurs lorsque de telles biblitohèques deviennent importantes, la gestion des différentes versions et leur eventuelle mises à jour soulève des problèmes qui relèvent du génie-logiciel.

De plus l'environnement d'exécution important nécessaire pour l'utilisateur de telles bibliothèques de sous programmes peut entraîner une certaine pénalisation au niveau de l'espace mémoire et des performances.

Cependant, les avantages contrebalançant largement les inconvénients, le choix d'implémentation de systèmes graphiques à partir de librairies de sous programmes a été et reste toujours la solution la plus largement répendue.

3.2 Interface avec le Matériel (VDI)

3.2.1 Rôle de l'interface terminal virtuel

Le lien de communication entre les routines indépendantes et les routines dépendantes du matériel à l'intérieur d'un logiciel graphique général est communément appelé **interface terminal virtuel.**

Cette interface constitue une description idéalisée d'une image correspondant aux commandes d'un terminal virtuel, abstraction des dispositifs physiques disponibles. Pour l'affichage sur un terminal particulier, ces commandes virtuelles sont transmises à un **gestionnaire de dispositif** ("device driver "ou "device manager") qui assure leur correspondance avec les commandes spécifiques du matériel.

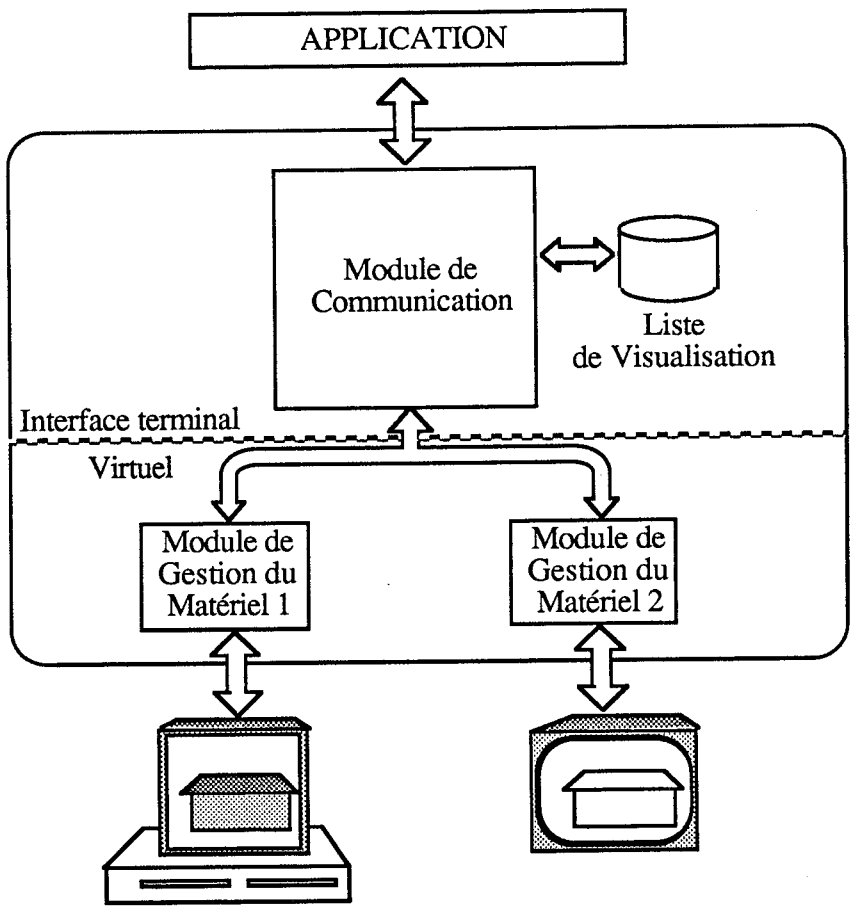
De cette manière tous les gestionnaires de dispositifs apparaissent de façon identique aux routines indépendantes, ce qui permet une prise en compte relativement aisée de tout nouveau matériel : il suffit d'écrire le gestionnaire associé.

3.2.2 Définition de l'interface terminal virtuel.

L'interface terminal virtuel joue un rôle **critique** dans les systèmes graphiques indépendants du matériel car elle doit être un point de convergence pour des dispositifs physiques très différents. Aussi la bonne définition de celle-ci constitue un problème central lors de la conception du logiciel graphique. De la situation de cette interface va dépendre l'étendue de son vocabulaire et la compléxité des gestionnaires de dispositif ("Device Drivers"). De par la très grande diversité des possiblités proposées par les différents matériels, il est clair que la définition de cette interface résulte d'un compromis entre les deux objectifs contradictoires que sont sa simplicité et ses performances.

3.2.2.1 Intersection des possibilités du matériel - terminal virtuel

Une première solution consiste à placer l'interface terminal virtuel à un niveau très bas représenté par **l'intersection** des possibilités des différents matériels. (Cela revient à l'extrême limite à ne considérer pour la visualisation que l'affichage de points et de segments de droite et l'effacement de tout l'écran).



- figure II.6 : Interface terminal virtuel -

Avec une telle définition de l'interface terminal virtuel, la plupart des fonctions sont effectuées par le logiciel d'une manière indépendante du matériel. En particulier le module de communication a la charge de simuler (en amont des mémoires réélles des terminaux) une mémoire d'entretien contenant une description structurée de l'image afin de permettre l'effacement sélectif et l'identification interactive de parties de l'image.

Par contre les gestionnaires des différents dispositifs physiques sont de taille relativement réduite et se limitent à de simples traducteurs (interpréteurs) des commandes virtuelles vers les commandes du matériel (fig II.6).

Cette méthode, qui bien souvent est la première venant à l'esprit a été employée pour de nombreux systèmes graphiques généraux. Elle présente pour principal avantage l'extrême simplicité des gestionnaires de dispositifs physiques, ce qui permet une prise en compte aisée et très rapide de tout nouveau matériel.

Cependant la prise en charge de nombreuses fonctionnalités par les routines indépendantes entraine un niveau de performances médiocre, exclu la possibilité de décharger le calculateur hôte au profit de processeurs satellites ou de terminaux intelligents et abouti à une sous utilisation dramatique de ceux-ci.

3.2.2.2 Logiciel adaptatif

Une alternative, à l'opposé de la solution précedente, consiste à définir l'interface terminal virtuel non plus comme l'intersection mais comme l'**union** des possibilités du matériel. Cela revient à positionner l'interface beaucoup plus "haut" en la rapprochant du programme d'application et à augmenter le vocabulaire de commande accepté par les gestionnaires de dispositif.

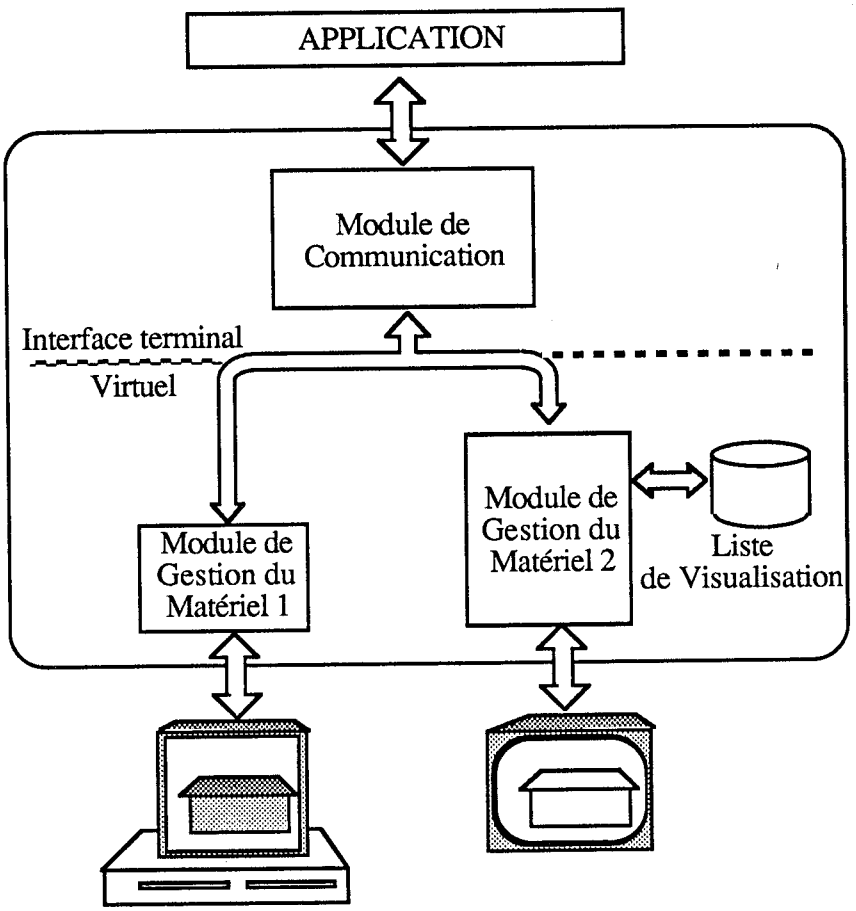
Ce placement est optimal pour ce qui est de tirer parti des spécificités des différents matériels. Le gestionnaire peut ainsi utiliser les possibilités du dispositif d'affichage chaque fois que possible, ce qui permet par exemple d'employer les listes de visualisation structurées de certains terminaux intelligents (fig. II.7.a) et certaines possibilités temps réel qu'ils autorisent.

Par contre pour des matériels moins évolués, le gestionnaire de dispositif devra éventuellement simuler par logiciel les fonctions non disponibles au niveau du "hardware" ou du "firmware" (par exemple prendre en charge une mémoire d'entretien structurée fig. II.7.b) et ignorer les fonctions non supportées par le matériel (par exemple la couleur sur un écran monochrome).

Dans une telle architecture logicielle, la partie indépendante du matériel se limite à un rôle de communication entre le programme d'application et les gestionnaires de dispositifs.

Une telle organisation permet une prise en compte et une utilisation beaucoup plus rationnelle des différentes possibilités des matériels et offre donc en conséquence un niveau de performances bien meilleur.

De plus la partie indépendante du matériel étant "amaigrie", il est éventuellement possible de libérer de nombreuses ressources sur le calculateur hôte en décentralisant les gestionnaires de dispositif vers des calculateurs satellites ou des stations de travail.



a) Terminal intelligent b) Terminal bas de gamme

- figure II.7 : Interface logiciel adaptatif -

Par contre, cette efficacité quand à l'utilisation des matériels se "paye" par une compléxité accrue des gestionnaires de dispositif qui assument la majeure partie du processus de visualisation. En particulier pour les matériels bas de gamme, une importante simulation par logiciel devra être mise en oeuvre au niveau des gestionnaires. Ceux-ci seront de taille beaucoup plus importante que pour la solution précédente, avec éventuellement une forte duplication de code pour des terminaux de même famille.

Cependant il faut relativiser ce problème car malgré les différences entre chaque gestionnaire, il nous semble qu'une partie du code, ou tout au moins la structure de ceux déjà construits, peut être réutilisée lors de la réalisation de nouveaux gestionnaires.

3.2.3 Standardisation de l'interface terminal virtuel

Comme pour l'interface avec les programmes d'application, des tentatives d'élaboration de standard internationnal sont développées pour l'interface terminal virtuel. Ces travaux sont effectués sous l'égide de l'ANSI et de l'ISO et sont désignés sous le terme de CGI/VDI ("Computer Graphics Interface" ou "Virtual Device Interface")

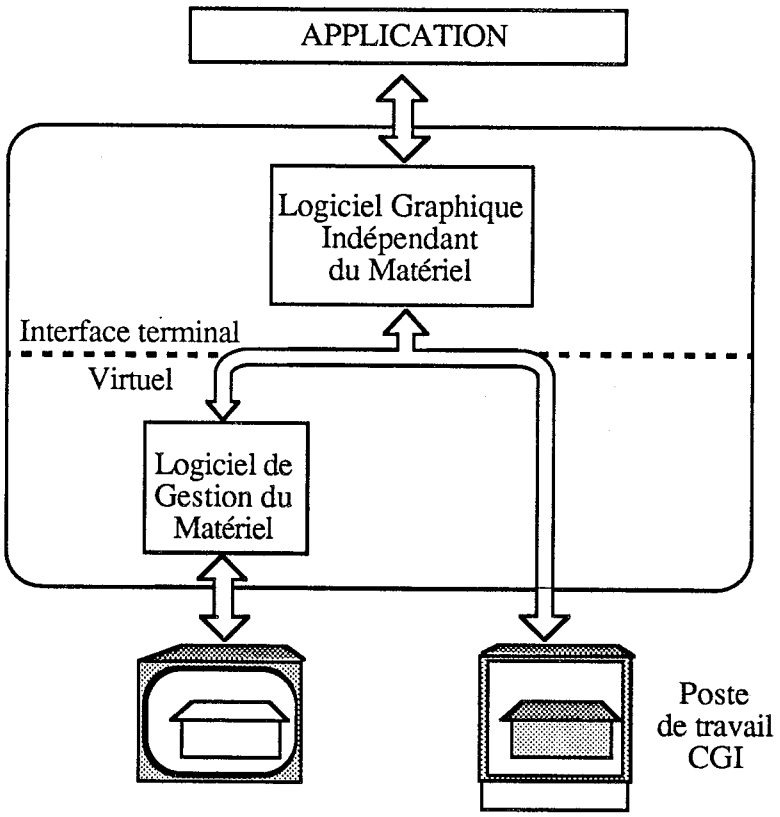
Un tel standard concerne les développeurs de systèmes graphiques, les vendeurs indépendants de logiciel, aussi bien que les constructeurs de matériel graphique (que ce soit de

périphériques, de cartes graphiques ou même de chips). Il permettrait de tirer pleinement parti des bénéfices offerts par une standardisation au niveau de l'interface avec les programmes d'application.

La réalisation de gestionnaires de dispositifs ("Device Drivers") qui reste actuellement une lourde charge pour les développeurs de logiciel, serait grandement facilité et pourait être effectuée à partir de squelettes de drivers et d'outils d'émulation.

Pour des applications, en particulier sur micro-ordinateurs, pour lesquelles un système graphique standard tel GKS est trop général pour correspondre à la taille et aux performances nécessaires, l'interface VDI pourrait fournir aux développeurs les outils dont ils ont besoin sans sacrifier l'indépendance par rapport au matériel...

Par ailleurs les constructeurs en adhérant à ce standard pourraient proposer un matériel s'interfaçant directement avec les systèmes graphiques (fig. II.8). A terme l'intégration du standard CGI/VDI dans le silicium étant envisageable [LeCo 85],[Grim 86],[Jete 86].



- figure II.8 : Intégration de l'interface CGI dans le matériel -

Cependant malgré tous les bénéfices potentiels qui en découleraient, l'élaboration d'une proposition pour VDI/CGI se heurte à de nombreux retards et difficultés.

Là aussi, et peut être encore plus que pour les standards au niveau de l'interface avec les programmes d'application (API), ces problèmes sont liés à la prolifération et à la rapide évolution du matériel. Faire un compromis entre toutes ces fonctionnalités (les divers matériels ayant des niveaux de performance extremement variés) tout en conciliant les différents intérêts des parties en présence s'avère être une tâche ardue.

Toutefois un modèle et une philosophie ont déjà été adoptés [Reid 85] et certains constructeurs tels IBM ont d'ores et déjà choisi une implémentation commerciale de l'ANSI-VDI comme base sur laquelle les développeurs indépendants de logiciels et les utilisateurs terminaux peuvent écrire leurs applications graphiques [Clar 85].

Il faut citer également dans l'effort de standardisation actuellement mené, les travaux concernant un standard international pour le stockage d'information graphique sous la forme de métafichiers (voir § 2.2.2.3). Ces travaux, répertoriés sous le terme de CGM (Computer Graphic Metafile) ou VDM (Virtual Device Metafile), sont directement liés à ceux du VDI/CGI car la définition de tels fichiers descriptifs d'images indépendants du matériel se trouve au même niveau d'interface. Le standard VDM/CGM peut être considéré en partie comme un sous ensemble du standard VDI/CGI [ANSI 83].

4 EXEMPLES DE SYSTEMES GRAPHIQUES GENERAUX : GRI-GRI, CORE-GSPC, GKS

4.1 Introduction

Dans cette partie nous présentons une étude de trois systèmes graphiques généraux : GRIGRI, le CORE-SYSTEM, et GKS (Graphic Kernel System). Tous trois sont issus des travaux que nous avons exposés dans la troisème partie de ce chapître. Ce sont des systèmes de visualisation et ils et se présentent sous la forme de bibliothèques de sous programmes appelables à partir d'un programme d'application.

En premier lieu nous étudierons GRIGRI (conçu au laboratoire IMAG (Institut de Mathématiques Appliquées de Grenoble)), qui, si sa contribution a été beaucoup moins importante que le CORE-SYSTEM et GKS, n'en demeure pas moins un système original, à l'origine des travaux que nous présentons cette thèse.

Ensuite, nous présenterons le CORE-SYSTEM car il a constitué un étape très importante dans l'élaboration des systèmes graphiques généraux et en particulier dans l'élaboration de standards internationnaux. De plus c'est le seul système qui intègre réellement des fonctionnalités pour le dessin en trois dimensions.

Finalement, nous examinerons GKS qu'il était également important d'étudier, car à partir des enseignements du CORE-SYSTEM il a pu "s'imposer" comme système standard au niveau internationnal.

Pour la présentation de ces trois systèmes nous avons adopté un schéma fondé sur les concepts généraux présentés dans le chapître I. Ainsi, nous regrouperons les primitives proposées par ces systèmes selon les processus d'ATTRIBUTION, de CONSULTATION, de DESCRIPTION, et de VISUALISATION. De la même manière, nous essayerons de présenter les différents attributs élémentaires manipulés en les rassemblant suivant les classes Morphologie, Aspect, Géométrie, Géométrie de prise de vue, Géométrie d'affichage, Eclairage.

4.2 GRIGRI

4.2.1 Historique et présentation générale

Le logiciel GRIGRI a été développé à la fin des années 1970 à l'Institut de Mathématiques Appliquées de Grenoble afin de piloter les différents matériels graphiques du CICG (Centre Interuniversitaire de Calcul de Grenoble) [LeDu 77], [Luca 77]. Ce logiciel fût une des premières expériences pour les systèmes graphiques généraux. Produit universitaire, il s'agit d'un système très simple pour la visualisation 2D, il comporte beaucoup moins de

fonctionnalités que les logiciels CORE-GSPC et GKS présentés par la suite (§ 4.3 et 4.6).

Néanmoins GRIGRI a apporté une contribution originale au développement de système graphique généraux en ayant une approche que l'on retrouve pas ailleurs [LLMa 78]. Par exemple, au contraire des autres logiciels de l'époque, la construction des objets graphiques est dans GRIGRI complétement disjointe de la visualisation. C'est pour cette originalité et pour son aspect historique que nous présentons GRI-GRI dans les pages qui suivent.

(remarque : GRIGRI a été diffusé en France et des extensions lui ont été ajoutées à cet effet, ainsi une version 3D fut développée [Gard 82]. Cependant, nous ne nous interesserons pas à celles-ci, nous contentant de mettre en évidence les principes qui furent à la base de ce logiciel.)

4.2.2 Attribution

4.2.2.1 Identité, Structure, Morphologie

Dans GRIGRI, les attributs d'Identité, de Structure et de Morphologie sont étroitement liés. En effet, ce sont les mêmes primitives qui permettent de les définir tous les trois simultanément. Il nous était donc difficile de les présenter de façon séparée.

Pour la définition des images GRIGRI considère deux types de données élémentaires : les points (ici des couples de coordonnées, GRIGRI ne prenant en compte que le dessin dans le plan) et les caractères alphanumériques.

GRIGRI permet une **structuration** de ces données élémentaires **à deux niveaux** :

- premièrement, un ensemble de points ou de caractères peuvent être regroupés en une **section** qui constitue l'élément de base manipulé par le logiciel

- deuxièmement, les sections peuvent elles-mêmes être regroupées dans une structure plus générale appelée **figure**. (Remarque : les sections qui appartiennent à une même figure seront alors définies par rapport au même système de coordonnées utilisateur).

Les deux primitives **POINTS** et **TEXTES** permettent de définir les morphologies qui constituent les sections d'une figure.

POINTS (nfig, tablong, tabx, taby, tab_marqueur)

transmet au logiciel graphique une suite de coordonnées qui serviront de base au tracé des différentes sections d'une figure.

-*nfig* est le numéro de la figure considérée,

-*tablong* est un tableau qui indique la longueur (nombre de coordonnées) de chacune des sections de la figure,

-*tabx* et *taby* sont les tableaux des coordonnées des points de la figure,

-*tab_marqueur* est un tableau dont chaque élément est un groupe de caractères qui permet de marquer ou étiqueter au moment de l'affichage les différentes sections de la figure.

Pour définir de façon complète la morphologie des différentes sections d'une figure (la primitive ***POINTS*** déclarant uniquement les points qui les constituent), il est nécessaire de leur associer un mode d'interprétation des coordonnées. Par exemple, celles-ci peuvent être considérées comme les sommets d'une ligne brisée, d'un contour polygonal fermé, ou d'une tâche polygonale. Le mode d'interprétation des coordonnées (ou mode de tracé) est spécifié, indépendamment de la déclaration des points des sections, à l'aide de la primitive ***MODE*** sur

laquelle nous revennons plus en détail au § 4.2.2.2..

TEXTES (nfig, tablong, tabtextes, tabx, taby)

permet de définir de manière analogue des sections de texte, dont les lettres sont contenues dans *tabtextes* et dont la longueur est spécifiée par *tablong*. A chaque section peut être associé un couple de coordonnées (*tabx* et *taby*) permettant de spécifier le point de départ du texte. Si ce couple est omis, le texte sera lié au texte de la section précédente.

Il faut noter que les primitives POINTS et TEXTES constituent des déclarations de données et non des affectations. Elles permettent de prévenir le logiciel graphique des types de données et des structures de rangement qu'il sera appelé à manipuler. Mais les différents tableaux passés en paramètres ne sont pas nécessairement remplis au moment de l'appel des primitives et doivent être conservés par le programme d'application.

Comme pour la structuration, il existe dans GRIGRI deux niveaux de dénomination qui correspondent :

- globalement, aux données repérées par un même numéro de figure (défini dans les primitives POINTS et TEXTES),

- au niveau au dessous, aux différentes sections d'une figure qui peuvent être regroupées à l'aide d'un numéro de corrélation (ou d'identification). Plusieurs sections à l'intérieur d'une même figure pourront avoir le même numéro de corrélation attribué au travers de la primitive MODE présentée au paragraphe suivant. Ainsi, comme nous le verrons par la suite, il sera possible de les distinguer au moment de l'affichage ou pour un effacement sélectif (voir § 4.2.4).

4.2.2.2 Aspect

L'aspect des sections est défini par l'intermédiaire de la primitive MODE, que nous avons déjà évoquée au paragraphe précédent. Cette primitive dont la forme d'appel est

MODE (nofig, nosect, nocorrel, descript)

permet donc en fait de fixer, à tout moment l'ensemble des caractéristiques d'une section (*nosect*) d'une figure (*nofig*) : son numéro de corrélation, son aspect et pour les sections de points le mode d'interprétation des coordonnées.

Les différents attributs élémentaires qui définissent l'aspect sont regroupés dans un descripteur (*descript*) et peuvent être pour les sections de points : la texture, l'épaisseur du trait, la couleur, etc... et pour les sections de texte la couleur, la police de caractères, etc...

4.2.2.3 Géométrie

Dans GRIGRI : l'utilisateur ne peut pas réellement définir d'attribut géométrique, si ce n'est la taille du texte exprimée là encore par la primitive MODE (dans les descripteurs). Sinon l'application ou le logiciel de mise en forme doivent effectuer les calculs éventuels de géométrie afin que les paramètres coordonnées *tabx* et *taby* utilisés à la déclaration des morphologies (primitives POINTS et TEXTES) intègrent déjà cette information. Il faut noter ici une incohérence de GRIGRI qui regroupe dans une même primitive les informations de MORPHOLOGIE et de GEOMETRIE.

4.2.2.4 Géométrie de prise de vue (Gv) et Géométrie d'affichage (Ga)

La géométrie de prise de vue est exprimée à l'aide de la primitive

FENETRE (nofen, igx, igy, sdx, sdy)

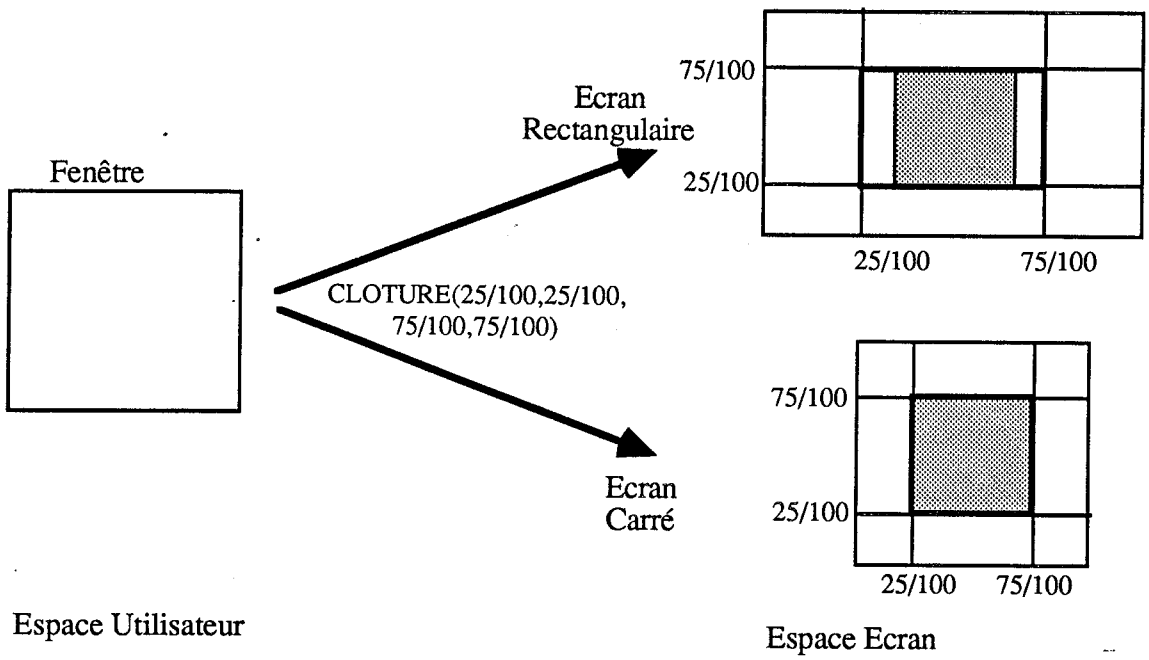
qui permet de définir un espace de départ (une fenêtre rectangulaire) dans un système de coordonnées laissé au libre choix de l'utilisateur (espace utilisateur). Le numéro de fenêtre *nofen* sert d'identificateur, ainsi la même fenêtre pourra être utilisée pour des ensembles de données différents (voir VISUALISATION et DESCRIPTION).

De manière analogue, la primitive

CLOTURE (noclot, igx, igy, sdx, sdy)

permet de définir un espace d'arrivée sur le dispositif d'affichage (Géométrie d'affichage). Les coins de la cloture sont définis dans un système de coordonnées arbitraire (centième d'écran par exemple), ainsi l'utilisateur peut définir la portion d'écran choisie pour l'affichage indépendamment des systèmes de coordonnées des dispositifs physiques réels

Le logiciel graphique assure automatiquement la transformation qui affiche le contenu de la fenêtre de la meilleure façon possible dans la cloture. Cette transformation est conçue de manière à utiliser au maximum la cloture, sans déformation de la fenêtre, et en cadrant au mieux celle-ci à l'intérieur de la cloture (voir fig.II.9). Le logiciel graphique effectue si nécessaire le coupage (clipping) du dessin par rapport aux bords de la fenêtre.



- figure II.9 : la transformation fenêtre cloture dans GRIGRI -

4.2.3 Consultation

Dans GRIGRI aucune consultation n'est possible, toutesles informations doivent être conservées par le programme d'application.

4.2.4 Visualisation

Dans GRIGRI le processus de visualisation est figé et ne dépend pas du terminal utilisé. L'utilisateur (le programmeur) demande explicitement une visualisation par l'invocation de la primitive

AFFICHER (nofig, nocorrel, nofen, noclot)

Les sytèmes de coordonnées utilisés pour la visualisation sont définis par référence à la fenêtre et la cloture désignées par leur numéro *nofen* et *noclot.*

nofig et *nocorrel* permettent de désigner les éléments (figures et sections de figures) à visualiser. En combinant judicieusement ces deux paramètres (table II.1), on peut afficher toutes les figures, une figure, toute sauf une ...etc.

	nfig	ncorrel
Nul	Toutes les figures	Toutes les sections
> 0	La figure nfig	Les sections corellées ncorrel
< 0	Toutes les figures sauf nfig	Toutes les sections sauf celles correllées ncorrel

- table II.1 : Les paramètres pour la visualisation dans GRIGRI -

Il est à souligner que la visualisation se fait indépendamment de la définition du dessin, ce qui signifie que l'on a la possibilité de construire toute une figure avant de l'afficher. Ce fait est très intéressant du point de vue du programmeur d'application, car il permet de conserver la logique de construction des objets à représenter sans se soucier des problèmes de tracé proprement dit. C'est pourquoi quand le mode d'une section est modifiée (ce qui est possible à tout moment) cela n'a aucun effet sur l'image affichée. Ce nouveau mode ne sera pris en compte que si l'utilisateur demande un nouvel affichage.

La primitive

EFFACER(nofig, nocorrel)

permet quand à elle d'obtenir des effacements sélectifs au niveau de chaque section. Les éléments à effacer sont désignés par leur numéro de figure,*nofig*, et de corrélation, *nocorrel*, en respectant les mêmes règles que pour l'affichage (table II.1 ci-dessus).

4.2.5 Description

De même que pour la visualisation, la description dans GRIGRI est un processus figé, c'est à dire prédéfini lors de la conception du logiciel.

Les primitives dialogue sont construites à partir d'actions de base qui permettent de les définir en termes fonctionnels sans se préoccuper de leur réalisation effective (dispositifs logiques). Les quatre types d'action élémentaire sont :

- introduction d'un couple de coordonnées,
- introduction d'une valeur alphanumérique,
- identification d'un élément de dessin,
- sélection d'une action parmi *n* proposées (menu).

Ces actions élémentaires peuvent être utilisées de manière composée à l'aide d'un paramètre nombre de répétitions. (La table II.2 page 58 donne les différentes significations de ce paramètre).

nb facteur de répétitions	signification
nb > 0	nb = nombre maximum d'entrées arrêt : - indication de fin de données de la part de l'opérateur, - maximum est atteint
nb = 0	nombre d'entrées inconnu a priori arrêt : - indication de fin de données de la part de l'opérateur.
nb < 0	sans signification

* le nombre de données effectivement relevé est transmis en retour.

- table II.2 : facteur de répétition pour la saisie dans GRIGRI -

Ainsi la primitive

POSITION (nb, nfig, nfan, nclot)

permet la description d'une section de points au travers d'une collecte de coordonnées. Les *nb* coordonnées spécifiés sont relevés par tout dispositif de communication utilisable et rangées dans les tableaux *tabx* et *taby* indiqués lors de la déclaration des données correspondant à *nfig*. La précision d'un numéro de cloture (*noclot*) et d'un numéro de fenêtre (*nfen*) permet de définir la transformation appropriée pour passer de l'espace écran à l'espace utilisateur.

Les concepteurs de GRIGRI ont essayé d'offrir une certaine cohérence en fournissant une primitive d'entrée symétrique avec la primitive "d'attribution" POINTS. Il est donc d'autant plus regrettable qu'il n'aient pas proposé la même chose pour les textes.

L'identification se fait par l'intermédiaire de la primitive

IDENTIFIER (nb, nfig, tfig, tcorrel)

qui permet de désigner une suite d'éléments d'un dessin. L'identification (numéro de figure, numéro de corrélation) est rangée dans les tableaux *tfig* et *tcorrel*. La donnée d'un numéro de figure *nfig* permet quand à elle de spécifier quelle figure ou quel ensemble de figures peut être identifié (la convention utilisée est la même que pour la visualisation, voir table II.1).

En plus de ces deux processus de description, GRIGRI propose des primitives de dialogue de plus bas niveau qui permettent l'introduction de données numériques , alphanumériques, et la définition de menus. Nous ne rentrerons pas dans le détail pour ces primitives, mais comme pour la collecte de coordonnées et l'identification, elles admettent un facteur de répétition et sont indépendantes de tout dispositif physique.

4.3 Le CORE-SYSTEM

4.3.1 Historique et présentation générale

Le CORE-SYSTEM est une description fonctionnelle pour un système de visualisation 3D, élaborée à la fin des années 1970 par le Graphic Standard Planning Comitee (GSPC) de l'ACM-SIGGRAPH.

Les objectifs de ce "groupe de travail", fondé en 1974 à Gaithersbury (Maryland USA) étaient initialement de préparer les bases pour de futurs standards graphiques. Cependant suite à la conférence de l'IFIPS à Seillac (France) en 1976, ses travaux s'orientèrent, résolument vers l'élaboration d'un système graphique général standard : le CORE-SYSTEM (ou CORE GSPC).

Une première description des fonctionnalités de ce système fut proposée en 1977 [GSPC 77], fortement influencée par le package graphique GPGS [BCVD 77] développé aux Pays-Bas. Les efforts du GSPC culminèrent en 1979 avec la publication d'un second document [GSPC 79], enrichissant la première proposition pour aboutir à un système graphique 3D complet.

En 1979 les activités de standardisation pour les systèmes graphiques furent reprises par l'ANSI (Américain Nationnal Standards Institute), où une rude bataille s'engagea pour l'homologation de CORE-SYSTEM opposé à GKS (Graphic Kernel System voir § 4) soutenu par l'ISO.

Simultanément le CORE-SYSTEM fut à l'origine du développement de très nombreux packages graphiques reprenant ses fonctionnalités. A titre d'exemple nous pouvons citer GW-CORE de l'université Georges Washington [FoWe 81], PLOT-10 IGL de Tektronix [Hvis 79], Template de Megatek, PGL de Hewlett Packard, Sun-Core de SUN-Microsystem [Sun 86]...pour les constructeurs de matériel graphique, DI-3000 de Precision Visual Inc., GSS-CORE de Graphic Software Systems Inc...pour les développeurs indépendants de logiciels. Tous ces systèmes ont eu une large diffusion (surtout aux Etats Unis), imposant en quelques sorte le CORE-SYSTEM comme un standard de facto.

Mais, l'ANSI décida finalement de porter son choix sur GKS et donc d'abandonner le CORE-SYSTEM. Par ailleurs, le GSPC n'a jamais défini de spécification formelle pour l'implémentation du CORE-SYSTEM à l'aide de langages de programmation de haut niveau (FORTRAN, PASCAL) [SpGa 86]. Aussi les différents packages "compatible-CORE" n'étaient pas réellement compatibles entre eux, chacun adoptant sa propre syntaxe (nom de fonctions, séquence de paramètres).

Ces différents problèmes, plus une certaine inadaptation du CORE à l'évolution fulgurante du matériel dans les années 80, ont entraîné la dissolution du GSPC en 1981 et l'abandon des travaux sur le CORE-SYSTEM. Néanmoins de nombreuses implémentations du CORE sont toujours largement utilisées (surtout pour le dessin 3D) et les concepts mis en évidence par le GSPC ont grandement influencé les travaux ultérieurs pour les standards graphiques, aussi avons nous jugé nécessaire la présentation de ce système. Pour ce faire nous nous sommes essentiellement basés sur le deuxième document du GSPC, publié en 1979 [GSPC 79].

4.3.2 Attribution

4.3.2.1 Morphologie - Primitives de sortie

Pour la visualisation d'objets à l'aide du CORE-SYSTEM, le programme d'application décrit leur morphologie en invoquant des primitives de sortie ("output primitives") qui correspondent au dessin d'objets élémentaires (segments de droite, lignes brisées , texte,

TYPE	DESCRIPTION	PRIMITIVES	POS. COURANTE
LINE	segment de droite débutant à la PC (Position Courante)	LINE_ABS_2(x,y) LINE_REL_2(dx,dy) LINE_ABS_3(x,y,z) LINE_REL_3(dx,dy,dz)	point spécifié par la primitive LINE
POLYLINE	ligne brisée débutant la PC	POLYLINE_ABS_2(tabx,taby,n) POLYLINE_REL_2(tabdx,tabdy,n) POLYLINE_ABS_3(tabx,taby,tabz,n) POLYLINE_REL_3(tabdx,tabdy,tabdz,n)	dernier point spécifié par la primitive POLYLINE
TEXT	texte débutant à la PC	TEXT(chaîne)	inchangée
MARKER	"marque" un point à l'aide d'un symbole	MARKER_ABS_2(x,y) MARKER_REL_3(dx,dy) MARKER_ABS_3(x,y,z) MARKER_REL_3(dx,dy,dz)	point spécifié par la primitive MARKER
POLYMARKER	"marque" une suite de points à l'aide d'un symbole	POLYMARKER_ABS_2(tabx,taby,n) POLYMARKER_REL_2(tabdx,tabdy,n) POLYMARKER_ABS_3(tabx,taby,tabz,n) POLYMARKER_REL_3(tabdx,tabdy,tabdz,n)	dernier point spécifié par la primitive POLYMARKER
MOVE	modification sans tracé de la PC	MOVE_ABS_2(x,y) MOVE_REL_2(dx,dy) MOVE_ABS_3(x,y,z) MOVE_REL_3(dx,dy,dz)	point spécifié par la primitive MOVE

- table II.3 : primitives de sortie dans le CORE-SYSTEM et effet sur la position courante -

symboles marqueurs). Il est à souligner que l'appel de ces primitives provoque l'affichage des morphologies correspondantes (nous détaillerons les opérations effectuées aux paragraphes 4.3.2.4 et 4.3.3), alors qu'avec GRIGRI l'affichage était séparé de la définition des objets et devait être demandé explicitement.

Les primitives de sortie sont définies à partir de coordonnées 2D ou 3D, exprimées dans un référentiel propre à l'application : l'espace universel ou l'espace utilisateur. Le CORE-SYSTEM maintient une position de dessin courante (en coordonnées universelles) utilisée comme point de départ implicite des primitives de tracé de segment, ligne brisée ou de texte. Les coordonnées fournies en paramètre des primitives de sortie, peuvent être exprimées soit en absolu (c'est à dire dans le repère universel), soit relativement à la position courante (dans ce cas elles définissent un déplacement par rapport à la position courante).

La table II.3 donne la liste des différentes primitives de sortie du CORE-SYSTEM et leur effet sur la position courante. Aux primitives de dessin LINE, POLYLINE, TEXT, MARKER, POLYMARKER, nous avons ajouté la primitive MOVE qui permet au programme d'application la modification de la position courante.

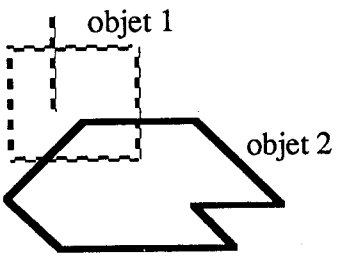
Remarque : cette notion de position courante est directement issue de la technologie de tube à balayage cavalier, et correspond à la position du faisceau d'electrons. Dans ce cas,il est parfois plus aisé de définir le dessin en terme de déplacements par rapport à cette position. La primitive MOVE, équivaut quand à elle à un déplacement du faisceau d'électrons en position éteinte.

4.3.2.2 Aspect - Attributs des primitives de sortie

A la différence de nombreux packages graphiques de l'époque, l'aspect des objets graphiques n'est pas spécifié par des paramètres passés dans les primitives de sortie. Dans le CORE-SYSTEM, existent un certain nombre d'attributs ("primitive attributes") qui permettent de contrôler séparemment l'apparence des primitives de sortie.

Ceux-ci sont spécifiés **modalement**, c'est à dire que toutes les primitives de sortie définies entre deux changements de la valeur courante d'un attribut verront leur aspect affecté de la même manière (figure II.10). A tout moment le programme d'application peut modifier la valeur courante d'un attribut en invoquant la fonction particulière associée.

```
Linestyle(tireté)    (* texture du trait *)
Linewidth(2)         (* ep  aisseur du trait *)
(* dessin de l'objet n°1 *)
LINE(...
...
POLYLINE(...

Linestyle(plein)(* modification de la
 texture mais pas de l'épaisseur du trait *)
(* dessin de l'objet n°1 *)
POLYLINE(...
....
```



- figure II.10 : spécification modale des attributs dans le CORE-SYSTEM -

Par ailleurs, dans le CORE-SYSTEM, tous les attributs des primitives de sortie sont définis de manière **statique**, cela signifie qu'aucun des attributs associés à une primitive de sortie lors de sa création ne pourra être modifié par la suite.

PRIMITIVE de SORTIE	ATTRIBUTS propres à la PRIMITIVE	ATTRIBUTS communs à toutes les PRIMITIVES
LINE POLYLINE	LINESTYLE : type de trait (plein, tireté...) LINEWIDTH : épaisseur du trait	COLOUR : index de couleur INTENSITY : brillance relative
TEXT	FONT : police de caractères CHARSIZE : taille des caractères CHARPLANE : orientation du plan d'écriture CHARUP : orientation des caractères dans le plan CHARPATH : direction d'écriture (gauche, droite, bas, haut) CHARSPACE : espacement CHARJUST : justification horizontale et verticale CHARPRECISION: précision de l'aspect du texte (string, char,stroke)	
MARKER POLYMARKER	MARKER-SYMBOL : no du symbole	

- table II.4 : les attributs des primitives de sortie dans le CORE-SYSTEM -

Parmi les attributs des primitives de sortie présentés dans la table II.4 ci-dessus, on remarquera la richesse des attributs géométriques pour le texte qui permettent de composer des textes tridimensionnels de très haute qualité. Cependant dans ce dernier cas cela peut nécessiter du matériel hautement sophistiqué ou d'importantes simulations par logiciel. Aussi l'attribut précision du texte (CHARPRECISION) permet de demander une prise en compte différente de ces attributs en fonction des possibilités du matériel. La précision "string" laisse au générateur de caractère du matériel le soin de choisir la présentation approximant au mieux ceux-ci. La précision "char" impose de positionner chaque caractère en respectant les attributs géométriques sans toutefois respecter leur orientation, le matériel faisant là aussi ce qu'il peut. La précision "stroke" impose quand à elle la prise en compte de tous les attributs du texte (auquel sera même appliquée la prise de vue perspective).

4.3.2.3 Structure et Identité - Segments

4.3.2.3.1 Définition de segments

Le CORE-SYSTEM permet au programme d'application de structurer ses données graphiques en regroupant un ensemble de primitives de sortie à l'intérieur d'un **segment**. Les primitives de sortie contenues dans un segment peuvent ensuite être manipulées simultanément. Une primitive de sortie appartient à un et un seul segment et les segments ne peuvent être imbriqués (un segment contient uniquement des primitives de sortie).

L'application construit un segment en demandant explicitement son ouverture (primitive OPEN-SEGMENT); un seul segment peut être ouvert à la fois. Toutes les fonctions primitives de sortie invoquées par la suite créent des morphologies élémentaires appartenant au segment. L'application termine un segment en demandant explicitement sa fermeture (primitive CLOSE-SEGMENT). On ne peut ajouter de primitives de sorties à un segment après l'avoir fermé.

Le CORE-SYSTEM propose deux types de segments selon qu'ils seront conservés ou non dans ses structures de données : les segments retenus ("retained segments") et les segments temporaires ("non retained segments").

4.3.2.3.2 Segments retenus

a) rôle des segments retenus

Les segments retenus constituent **l'unité d'identification et de modification** de l'image. Les primitives de sortie définies à l'intérieur de ceux-ci sont mémorisées et conservées par le CORE-SYSTEM jusqu'à ce que le segment soit explicitement détruit. Lorsqu'une nouvelle image est générée, les primitives maintenues dans les segments retenus, sont automatiquement réaffichées sans intervention de l'application.

Les segments retenus sont identifiés par un entier (appartenant à l'intervalle 1 ... 32767). A tout moment l'application peut les renommer, les détruire.

b) attributs dynamiques des segments retenus

Comme nous l'avons vu au paragraphe 4.2.2.2, le programme d'application ne peut modifier les attributs des primitives de sortie après leur définition. Par contre, à chaque segment est associé un ensemble d'attributs **dynamiques** qui contrôlent son apparence et qui peuvent être réinitialisés à tout instant par le programme d'application.

Les attributs dynamiques des segments retenus sont les suivants :

* **Visibilité ("Visibility")** : qui permet à l'application de rendre un segment visible ou invisible, c'est à dire afficher ou non toutes les primitives de sortie contenues dans le segment.

* **Surbrillance ("Highlightning")** : qui permet de distinguer un segment du reste de l'image en le rendant surbrillant.

* **Détectabilité ("Detectability")** : qui permet à l'application de rendre le segment "détectable" ou non lors d'une identification interactive avec un dispositif de dialogue. Si le segment est detectable, cet attribut permet de lui associer également une priorité de detection qui permettra de lever d'éventuelles ambiguités lors d'une désignation.

* **Transformation d'image ("Image Transformation")** : qui permet de modifier la position (par translation, rotation ou mise à l'echelle) de l'image d'un segment sur l'écran (géométrie d'affichage).

La modification des attributs dynamiques d'un segment spécifique s'effectue en indiquant son numéro lors de l'invocation des fonctions associées. Par ailleurs, afin d'éviter à l'application lors de la création de chaque nouveau segment retenu l'initialisation systématique de ses attributs dynamiques, ceux-ci pourront également être spécifiés modalement comme les attributs des primitives de sortie (cf § 4.2.2.2). (Le CORE-SYSTEM conserve une valeur courante pour chacun des attributs dynamiques qui s'applique par défaut à tous les nouveaux segments retenus, l'application pouvant quand elle le désire modifier cette valeur courante).

c) niveaux d'identification

Aux noms (numéros) des segments retenus spécifiés par le programme d'application lors de leur création, le CORE-SYSTEM permet d'ajouter un 2ème niveau d'identification des primitives de sortie lors d'une désignation interactive.

En d'autres termes, il est possible de nommer des primitives ou groupes de primitives de sortie à l'intérieur d'un segment retenu. Ce deuxième niveau d'identification est défini à l'aide de l'attribut des primitives de sortie PICK_IDENTIFIER. Cette possibilité peut s'avérer très

intéressante, en particulier pour la gestion de menus, en évitant la création de nombreux segments retenus.

4.3.2.3.3 Segments temporaires

Les segments temporaires sont destinés aux applications ou portions d'applications ne nécessitant pas de modification de l'image. Aussi les primitives de sortie définies à l'intérieur de segments temporaires ne sont pas mémorisées par le CORE-SYSTEM et ne sont pas réaffichées lors des regénérations de l'image. Les segments temporaires n'ont donc ni nom, ni attributs, il est impossible de les modifier. Les seules actions autorisées sont la création de nouveaux segments temporaires ou l'effacement de **tous** les segments temporaires.

Le traitement des segments temporaires dans le CORE-SYSTEM est "calqué" sur le comportement des consoles à tube mémoire. Les primitives définies à l'intérieur des segments temporaires sont affichées et restes visibles jusqu'à ce que l'image subisse une mise à jour nécessitant une régénération compléte (modification, suppression d'un segment retenu ou demande explicite de régénération (primitive NEWFRAME)). Dans le cas d'une telle modification et sur un tube mémoire, après effacement complet de l'écran, les segments retenus visibles sont réaffichés et les segments temporaires perdus.

4.3.2.4 Géométrie de prise de vue (Gv) et géométrie d'affichage (Ga) Transformation de Visualisation

La transformation de visualisation permet de sélectionner une partie de l'univers graphique de l'application (Espace Utilisateur WCS) et de spécifier la manière dont les objets seront projetés sur la surface de visualisation. Cette transformation géométrique est appliquée aux primitives de sortie définies en coordonnées utilisateur pour obtenir les coordonnées dans le repère du (des) dispositif(s) d'affichage.

Le CORE-SYSTEM permet de définir des transformations de visualisation 2D ou 3D, le 2D étant un sous-ensemble du 3D.

L'application peut définir une transformation de visualisation 2D de manière classique en spécifiant une "fenêtre" ("window") rectangulaire d'orientation quelconque dans l'espace utilisateur et une "clôture" ("viewport") sur la surface de visualisation. Afin d'éviter toute dépendance par rapport au matériel (chacun ayant son propore espace d'adressage), la clôture est définie dans un espace écran normalisé ("Normalized Device Coordinate System" ou "NDC") représentant un écran virtuel carré dont les bornes d'adressage sont comprises entre 0 et 1. Le CORE-SYSTEM prend en charge automatiquement la conversion de l'espace normalisé (NDC) vers l'espace écran ("Device Coordinate Space" ou "DC").

Pour le 3D le CORE-SYSTEM propose un mécanisme très général pour spécifier la transformation de visualisation qui plus compliquée que pour le 2D inclus une projection de l'espace tridimensionnel vers l'espace bidimensionnel. Il est ainsi possible de définir n'importe quel type de projection géométrique plane (perspective, parallèle orthographique ou oblique) [CaPi 78], en conservant une compatibilité ascendante avec le mécanisme de fenêtre-clôture du 2D.

Comme pour les attributs des primitives de sortie, vus précédemment, la transformation de visualisation est définie modalement : il existe une transformation courante appliquée à tous les objets définis jusqu'à ce que l'un des paramêtres la spécifiant soit modifié, la nouvelle transformation ainsi définie devenant la transformation courante. L'application peut à tout instant modifier la transformation de visualisation, la seule restriction étant que cela n'ait pas lieu alors qu'un segment est ouvert. En effet un segment correspond à **l'image** d'un ensemble de primitives de sortie (qui sont mémorisées après la transformation de visualisation en coordonnées normalisée ("NDC")), celles-ci doivent donc toutes avoir les mêmes paramètres de prise de vue.

4.3.2.5 Géométrie

Basé sur le principe de la séparation modélisation/visualisation, le CORE-SYSTEM n'offre **pas** réellement d'**attributs géométriques** que l'on puisse associer aux objets graphiques. Dans le cas des primitives de sortie, seul le texte possède des attributs de ce type ("Charplane", "Charupvector"... qui définissent son orientation), et en ce qui concerne les segments, l'attribut dynamique de transformation d'image fait partie de la classe Géométrie d'affichage (Ga). Cependant le CORE-SYSTEM offre au programme d'application, la possibilité de définir une transformation géométrique **courante** dans l'espace utilisateur ("World Coordinate Transformation"). Cette transformation est combinée automatiquement avec la transformation de visualisation courante, et la matrice résultante ("Composite Viewing transformation") est appliquée **directement** aux coordonnées universelles utilisées lors de l'invocation des primitives de sortie. Elle permet de positionner un objet dans l'espace utilisateur avant de lui appliquer la transformation de prise de vue.

(remarque : la composition des matrices en coordonnées homogènes [RoAd 76]) permet d'appliquer les transformations qu'elles définissent en une seule étape. Il est ainsi possible d'assurer une interface plus efficace entre le CORE-SYSTEM et un système de modélisation.)

4.3.3 Description - Fonction d'entrée ("Input Primitives")

4.3.3.1 Notion de dispositifs logiques

En ce qui concerne la description, le CORE-SYSTEM offre des fonctions d'assez bas niveau qui permettent à l'application l'utilisation des dispositifs de dialogue indépendamment des dispositifs physiques particuliers disponibles sur une configuration donnée. Ces fonctions d'entrée ("input primitives") sont basées sur le concept de **dispositifs logiques** d'entrée, abstraction de dispositifs physiques typiques.

CLASSE LOGIQUE	MODE de FONCTION-NEMENT	INFORMATION RENDUE
Entrée scalaire (VALUATOR)	echantillonné	valeur scalaire
Releveur de coordonnées (LOCATOR)	echantillonné	position en coordonnées normalisées (NDC) *
Releveur d'une suite de coordonnées (STROKE)	asynchrone (événement)	séquence de positions en coordonnées nomralisées (NDC) + le nombre de positions relevées *
Entrée chaîne de caractères (KEYBOARD)	asynchrone (événement)	chaîne de caractères ASCII et sa longueur
Désignation (PICK)	asynchrone (événement)	numéro du segment identifié + PICK_IDENTIFIER de la primitive désignée
Sélecteur (BUTTON)	asynchrone (événement)	entier désignant l'alternative selectionnée (numéro du bouton)

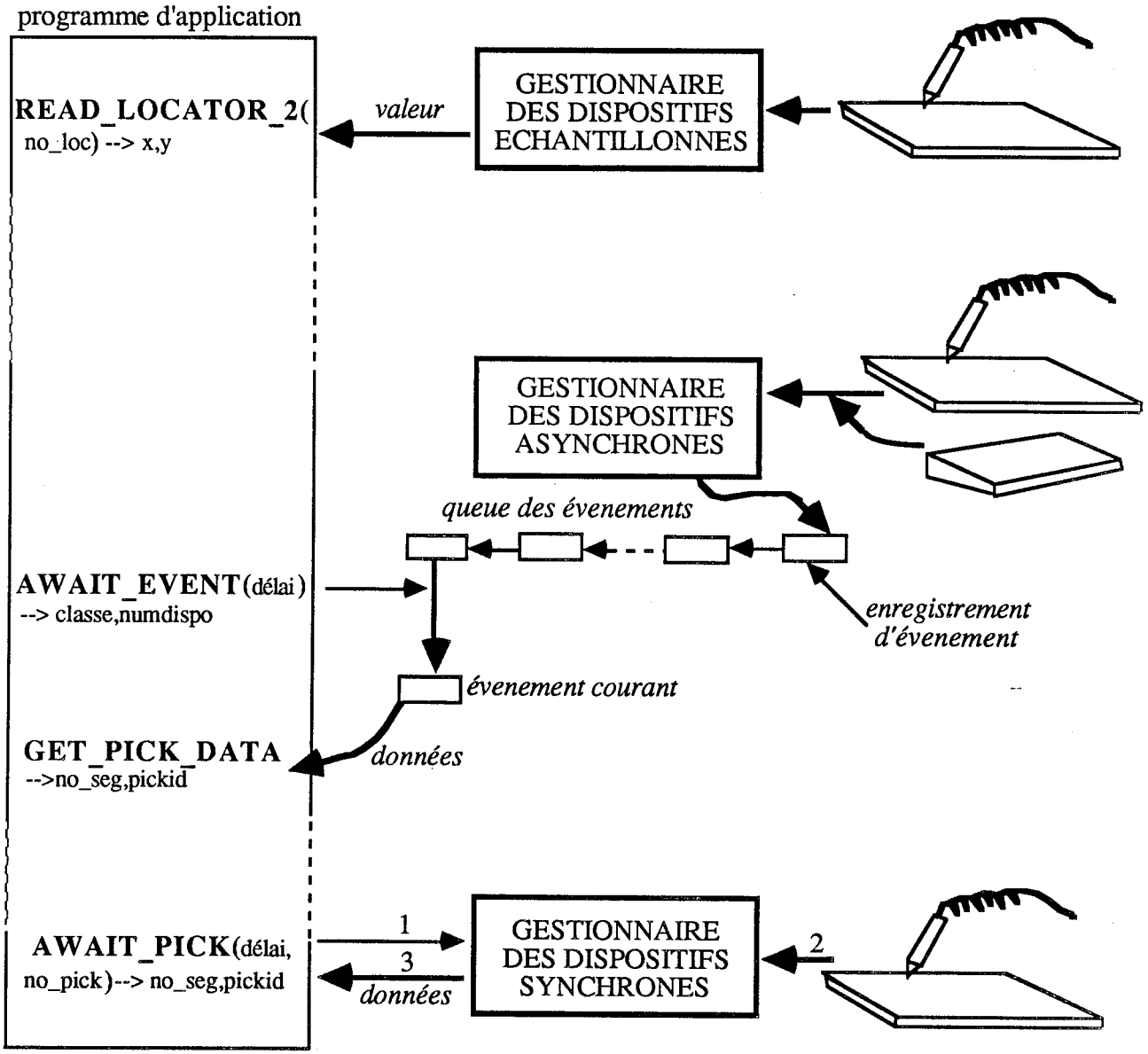
- table II.5 : les dispositifs logiques et leur mode de fonctionnement dans le CORE-SYSTEM -

Les dispositifs logiques sont définis par le **type** de données fournies et par la **manière** dont celles-ci sont acquises par le programme d'application. Le CORE_SYSTEM fait correspondre aux dispositifs logiques les périphériques d'entrée qui permettent d'implémenter le plus facilement les fonctionnalités désirées. Si sur le site considéré il n'existe pas de dispositifs physiques correspondant naturellement à ces fonctionalités, le CORE-SYSTEM les simule par logiciel. Le CORE-SYSTEM reconnait six classes de dispositifs logiques qui sont présentées dans la table II.5 ci-dessous, ceux-ci sont regroupés en différentes catégories selon leur mode de fonctionnement.

(* remarque : les dispositifs logiques LOCATOR et STROKE renvoient des coordonnées normalisées ("NDC"), cependant le CORE-SYSTEM propose des utilitaires pour transformer celles-ci en coordonnées utilisateur en leur appliquant l'inverse de la transformation de visualisation courante).

4.3.3.2 Classes et modes de fonctionnement des dispositifs logiques

La figure II.11 résume les différents modes de fonctionnement des dispositifs logiques dans le CORE-SYSTEM, que nous expliquons de façon détaillée dans les paragraphes suivants.



- figure II.11 : Modes de fonctionnement des dispositifs d'entrée dans le CORE-SYSTEM -

dispositifs échantillonés ("sampled devices")

Les dispositifs appartenant à cette catégorie (LOCATOR et VALUATOR) fournissent des valeurs qui peuvent être consultées (lues au vol) à tout instant par le programme d'application.

dispositifs asynchrones ("event-causing devices")

L'utilisation des dispositifs de cette seconde catégorie (STROKE, BUTTON, KEYBOARD et PICK) provoque un "événement" (event), et un enregistrement contenant l'information correspondante (event report) est placé dans une file d'attente (FIFO) maintenue par le CORE-SYSTEM. Le programme d'application pourra quand il le désire, retirer un "événement" de cette queue (primitive AWAIT_EVENT) et consulter l'information associée (primitive GET_*device*_DATA).

association de dispositifs echantillonés et asynchrones

Si les dispostifs asynchrones et les dispositifs échantillonés forment deux catégories distinctes, le CORE_SYSTEM offre néanmoins la possibilité de les associer . De cette manière la valeur courante d'un dispositif échantilloné peut être lue et conservée dans la file d'attente chaque fois qu'un évenement est provoqué par le dispositif asynchrone associé. Par exemple un curseur sur une tablette disposant de plusieurs boutons pourra combiner les dispositifs logiques LOCATOR (échantilloné) et BUTTON (asynchrone).

fonctionnement synchrone ("request mode")

La gestion asynchrone des dispositifs de dialogue vue précédemment n'est pas toujours nécessaire. Un grand nombre d'applications ne nécessitent pas l'utilisation simultannée de plusieurs dispostifs d'entrée. De plus l'implémentation d'une gestion asynchrone des dispositifs de dialogue peut s'avérer difficile voir impossible sous certains systèmes d'exploitation.

Aussi le CORE_SYSTEM fournit-il des fonctions d'entrée moins générales qui permettent une gestion synchrone des événements. A l'invocation de l'une de ces fonctions (primitives AWAIT_*device*) le CORE_SYSTEM attend que le(s) dispositif(s) désigné(s) soit(ent) actionné(s) ou qu'un délai passé en paramêtre soit dépassé. En retour l'information associée est renvoyée à l'application.

4.3.3.3 utilisation des dispositifs d'entrée par l'application

gestion des dispositifs

En principe le programme d'application peut accéder à un nombre quelconque de dispositifs logiques à l'intérieur de chaque classe. Ceux-ci sont identifiés de façon unique par leur classe et un numéro d'identification à l'intérieur de la classe.

Afin de pouvoir utiliser un dispositif logique, le programme d'application doit tout d'abord l'initialiser ce qui garanti l'accès à celui-ci (primitive INITIALISE_DEVICE(Classe,num)). Puis, avant toute interaction, le dispositif doit être activé (primitive ACTIVATE _DEVICE(Classe,Num)), sauf dans le cas d'une utilisation asynchrône (§ 4.2.3.2.2). Tous les événements provoqués ensuite par le dispositif sont placés dans la file d'attente, ou s'il s'agit d'un dispositif echantilloné, la valeur de celui-ci est disponible pour une éventulle lecture, et ceci jusqu'à ce que le dispositif soit explicitement désactivé par l'application.

Si l'application ne souhaite plus avoir accès à un dispositif elle peut alors le signifier au CORE-SYSTEM (primitive TERMINATE(Classe,Num)).

écho et caractéristiques des dispositifs

A chaque classe de dispositif de dialogue correspond une variété d'échos prédéfinis par le CORE-SYSTEM. L'application dispose ainsi d'un certain nombre de fonctions lui permettant de contrôler dans une certaine mesure les caractéristiques de l'écho des dispositifs logiques (ces fonctions ne peuvent être invoquées qu'après initialisation des dispositifs logiques).

De la même manière l'application peut contrôler certaines caractéristiques des dispositifs logiques, par exemple la précision pour les dispositifs d'identification (SET_PICK(no_pick,précision)) ou la plage de valeurs pour un dispositif de la classe VALUATOR.

4.3.4 Visualisation

4.3.4.1 le processus de visualisation

Dans le CORE_SYSTEM le processus de visualisation est **implicite**, il consiste simplement en l'application de la transformation de visualisation (éventuellement composée avec une transformation de modélisation) aux primitives de sortie au fur et à mesure de leur définition et à l'affichage de l'image résultante sur la ou les surfaces de visualisation selectées à cet instant. Le seul traitement effectué lors de la visualisation est éventuellement un coupage pour supprimer les points situés en dehors de l'espace de vision.

Par contre, mise à par la prise de vue perspective, aucun traitement pour la prise en compte du réalisme n'est réalisé. Ainsi il n'y a pas la possibilité de demander au logiciel graphique l'élimination des parties cachées, pas plus que la simulation de phénomènes d'éclairages qui sont totalement absents du système (du moins dans sa définition standard). Aussi, si l'utilisateur désire effectuer une visualisation réaliste, il se trouve dans l'obligation d'effectuer tout les traitements nécessaires au niveau de de l'application perdant du coup pratiquement toutes les facilités d'utilisations offertes par le logiciel graphique.

Le seul contrôle que le CORE-SYSTEM propose à l'utiliseur sur le processus de visualisation concerne son efficacité sur laquelle il est possible d'agir dans une certaine mesure, c'est ce que nous exposons brièvement au paragraphe suivant.

4.3.4.2 controle du processus de visualisation

Coupage

L'application peut demander au CORE-SYSTEM d'effectuer ou non un coupage par rapport aux limites du volume de vision (fenêtre en 2D, cône de vision en 3D . La désactivation du coupage permet d'éviter d'importants traitements au CORE_SYSTEM quand les objets décrits sont entièrement visibles et ne nécessitent pas de coupage. La réinitialisation de l'indicateur de coupage (activé ou désactivé) est possible à tout instant, la seule restriction étant qu'aucun segment ne soit alors ouvert.

L'affichage différé

Le CORE-SYSTEM propose un certain nombre de fonctions permettant à l'application de **contrôler** dans une certaine mesure l'**efficacité** des modifications apportées à l'image.

Les modifications de l'image visible peuvent être de nature différente et constituer :

- soit en l'**ajout** de nouveaux éléments (créations de nouvelles primitives de sortie, segments invisibles rendus visibles),

- soit en "l'altération" de l'image existante (destruction de segments, segments retenus rendus

invisibles, transformation de segments...). Ce deuxième type de modifications entraîne une "**régénération**" de l'image : l'image mise à jour doit refléter le nouvel état des structures de données graphiques du CORE_SYSTEM et ne sont réaffichés que les segments retenus visibles alors que les segments temporaires sont perdus. Ce terme de régénération est utilisé par analogie avec le processus mis en oeuvre pour des dispositifs d'affichage sans effacement sélectif tels les tubes mémoire (ou les tables traçantes) pour lesquels ce type de modification nécessitera un effacement complet de l'écran (ou la mise en place d'une nouvelle feuille de papier) et de redessiner tous les segments mémorisés.

Dans le CORE-SYSTEM il est possible de contrôler **séparemment** ces deux types de modifications en **différant** les mises à jour de l'image.

En supprimant la visibilité immédiate (IMMEDIATE_VISIBILITY(OFF)) l'application peut autoriser le système graphique à **retarder arbitrairement** la prise en compte des ajouts à l'image qui dans le cas contraire (IMMEDIATE_VISIBILITY(ON)) doit avoir lieu dès l'invocation des primitives les provoquant. Lorsque l'affichage est retardé, le programme d'application peut à tout moment demander la réalisation immédiate de toutes les modifications non encore effectuées (primitive MAKE_PICTURE_CURRENT).

En ce qui concerne les modifications entraînant une "régénération" de l'image, l'application a la possibilité de les regrouper et de signifier au CORE_SYSTEM que seule compte l'image finale. Ainsi quand l'application effectue une demande d'affichage différé (BEGIN_BATCH_OF_UPDATES) les effets visuels des fonctions provoquant des modifications de l'image invoquées par la suite, sont bloqués jusqu'à ce qu'une demande explicite d'affichage soit formulée (END_BATCH_ OF_UPDATES).

Le contrôle de la visibilité immédiate a pour objectif principal l'amélioration des communications entre le système graphique et le(s) dispositif(s) d'affichage. En effet dans certains cas et pour certains matériels, il peut être interessant de regrouper des commandes et données graphiques afin de les envoyer d'un seul coup (par exemple blocage des données pour l'optimisation des transferts vers une table traçante).

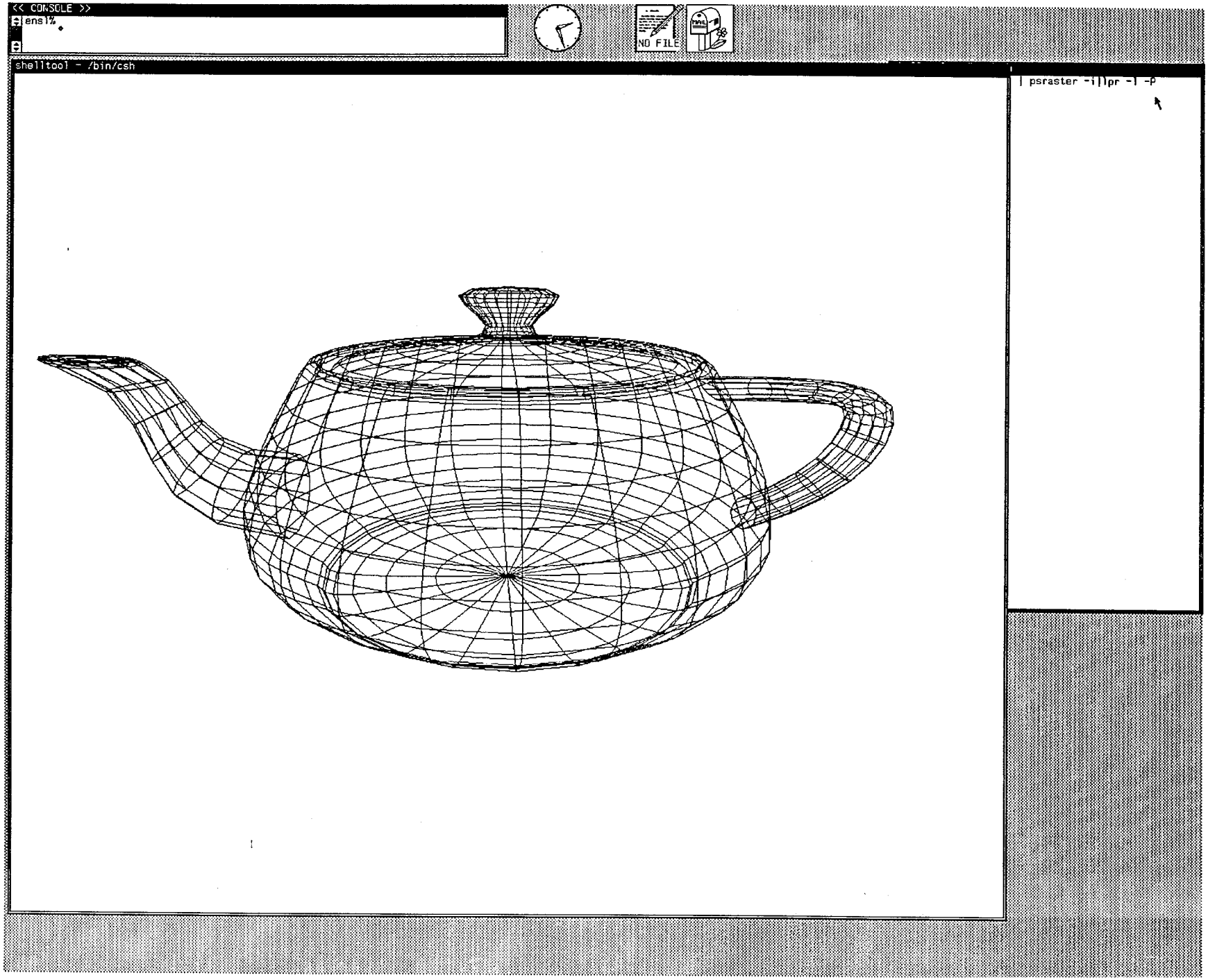
Le groupement de modifications quand à lui doit permettre la prise en compte et l'utilisation efficace des capacités fonctionnelles des dispositifs de visualisation lors de l'invocation des primitives provoquant une "régénération" de l'image. Un exemple typique est dans le cas d'une console à tube mémoire, la possibilité d'éviter des effacements multiples lors de destructions successives de segments retenus. Les fonctions de modification effectivement différées peuvent varier d'un dispositif d'affichage à un autre, ceci pouvant aller du tout (cas d'un table traçante) au rien (tube cathodique avec mémoire d'entretien).

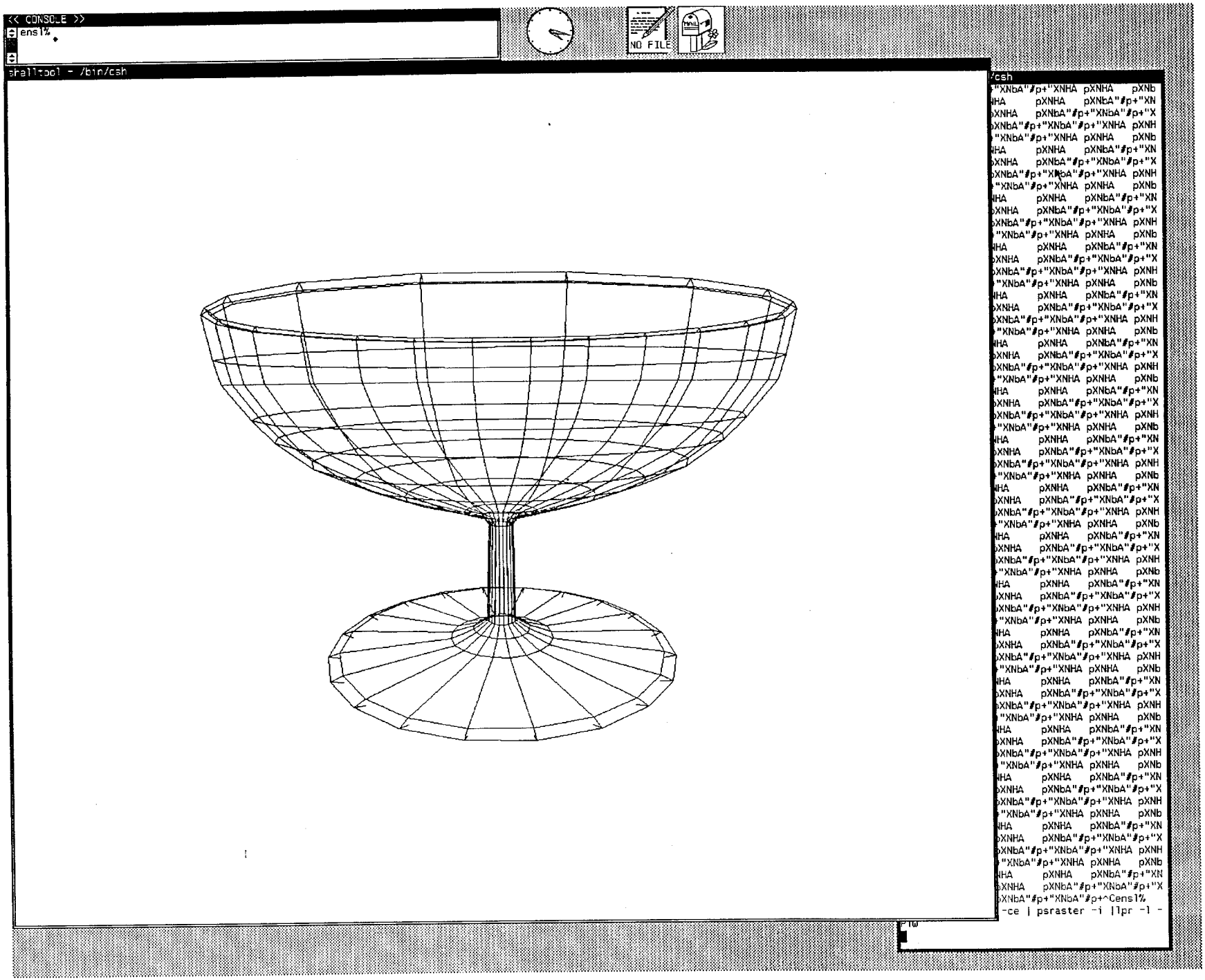
Il est à noter que pour l'affichage retardé, les modifications (ajouts) de l'image ont toujours lieu selon l'ordre dans lequel elles ont été demandées alors que pour les groupements de modifications (régénération) celles-ci n'interviennent pas obligatoirement dans l'ordre où elles ont été définies : seule compte l'image finale !

4.3.5 Consultation

De même que pour les modifications de l'image, l'unité de consultation est le segment retenu. Par l'intermédiaire de fonctions de consultation (INQUIRY functions) le programme d'application peut à tout moment retrouver l'information (valeur des attributs dynamiques) associée aux segments retenus.

Par contre il est impossible de recupérer l'information attachée aux primitives de sortie définies à l'intérieur des segments (par exemple la couleur des primitives de sortie). Le CORE_SYSTEM offre seulement un ensemble complet de fonctions de consultation de l'état courant du système, ainsi l'application peut connaître la valeur courante des attributs, pour les primitives de sortie, l'état des différents dispositifs de dialogue, etc ...


<< CONSOLE >>
ens1%
NO FILE
shelltool - /bin/csh
| psraster -i|lpr -l -P


CONSOLE
ens1%
NO FILE
shelltool - /bin/csh
/csh
ens1%
-ce | psraster -i |lpr -l -

4.3.6 Implémentation du CORE-SYSTEM

Le CORE_SYSTEM doit pouvoir être utilisé par une grande variété d'applications allant de la sortie de dessins statiques à des applications interactives. De plus sur certains matériels bas de gamme, l'implémentation de certaines fonctionnalités peut s'avérer difficile et nécessiter une simulation par logiciel compliquée et couteuse. Aussi n'est-il pas toujours souhaitable d'implémenter toutes les fonctionalités CORE_SYSTEM sur un site donné.

Pour éviter aux éventuels implémenteurs du CORE_SYSTEM d'effectuer un choix arbitraire parmi les différentes possibilités, ce qui nuirait fortement à la portabilité des applications, des niveaux avec une compatibilité ascendante ont été définis pour trois classes de fonctionalités : les sorties (output), les entrées (input), la dimension de l'espace utilisateur (2D ou 3D). On trouvera le detail de ceux-ci dans la table II.6 ci-dessous.

<table>
<tr><th>DIMENSION-
NALITE</th><th>SORTIES
(OUTPUT)</th><th>ENTREES
(INPUT)</th></tr>
<tr><td rowspan="2">1 Bi-Dimensionnel
(2D)

Uniquement les fonction-nalités 2D.</td><td>1 Sorties de Base
(Basic Output)

Toutes les primitives de sortie avec leurs attributs mais pas de segments mémorisés (segments retenus).</td><td>1 Pas d'Entrée
(No Input)</td></tr>
<tr><td rowspan="2">2 Sorties avec
Mémorisation
(Buffered Output)

Segments retenus avec attributs de visibilité et surbrillance (et en plus detectabilité selon le niveau d'entrée).</td><td rowspan="2">2 Entrees Synchrones
(Synchronous input)

Pas de séquences d'inter-action asynchrones.Toutes les classes de dispositifs logiques sont prises en compte ("Pick" ne l'est que si le niveau de sortie est ≥ 2)</td></tr>
<tr><td rowspan="2">2 Tri-Dimensionnel
(3D)

Fonctionnalités 2D et 3D.</td><td rowspan="2">3 Sorties
Dynamiques
(Dynamic Output)

Toutes les fonctionnalités de sortie sont présentes. En particulier, les attri-buts de transformation d'image.</td></tr>
<tr><td>3 Entrees Complètes
(Complete Input)

Toutes les fonctionnalités d'entrée sont implémentées.</td></tr>
</table>

- table II.6 : les différents niveaux d'implémentation du CORE-SYSTEM -

Les différentes implémentations du CORE-SYSTEM doivent respecter les régles suivantes :

- implémenter **toutes** les fonctionalités des niveaux choisis,
- n'implémenter **aucune** des fonctionalités des niveaux supérieurs.

4.4 GKS - Graphic Kernel System

4.4.1 Historique et présentation générale

GKS ("Graphic Kernel System") est le premier véritable standard international reconnu officiellement pour l'interface entre un système de visualisation graphique et un programme d'application. Fondé sur les mêmes principes que le CORE-SYSTEM, GKS avait initialement des objectifs plus restreints et etait destiné uniquement à la visualisation 2D (une extension 3D est en cours d'élaboration).

Le standard GKS est publié sous la forme d'un document donnant la description fonctionnelle de son interface. A celle-ci, est adjointe une couche supplémentaire spécifiant la syntaxe ("language binding") pour l'implémentation du système à l'aide de différents langages de programmation tels FORTRAN, ADA... Ce dernier point permet d'assurer une véritable compatibilité entre les différentes implémentations de GKS (compatibilité qui fit cruellement défaut au CORE-SYSTEM).

GKS, système initialement conçu par le DIN ("Deutsches Institute für Normung"), fut soumis en 1977 au groupe de travail de l'ISO chargé de la standardisation des systèmes graphiques (ISO WG2). L'ISO étudia alors simultanément les deux propositions CORE-SYSTEM du GSPC (soutenu par l'ANSI) et GKS. A l'époque certains experts pensaient que ces deux systèmes pouvaient faire l'objet d'une normalisation, GKS adressant simplement la visualisation 2D et le CORE-SYSTEM permettant le 3D. Il était nécessaire qu'une certaine cohérence existe entre ces deux systèmes, et les membres de l'ANSI participant aux réunions de l'ISO permirent à GKS de tirer profit des importants travaux menés sur le CORE-SYSTEM. Cependant, devant la nécessité d'adopter un standard international les discussions sur GKS et le CORE-SYSTEM tournèrent rapidement en un affrontement pour l'adoption de l'un ou de l'autre. Finalement en 1979, l'ISO opta pour le choix de GKS comme sujet de travail ("work item") et donc comme seul système succeptible d'une standardisation. Mais ce n'est qu'en 1985 que l'ISO publia officiellement le document établissant la norme GKS [ISO 85]. Peu de temps après ,l'ANSI ratifiait cette décision en adoptant GKS comme standard Américain [ANSI 85]. GKS est maintenant reconnu par de nombreux pays: la République Fédérale Allemande bien entendu, la Grande Bretagne, la France...

Mais si GKS définissait un standard 2D, le besoin d'un système 3D était réel (source de nombreux arguments pour les défenseurs du CORE-SYSTEM), l'ISO étudia dès 1983 l'éventuelle extension de GKS au 3D. La décision fut prise de conserver une compatibilité totale avec la norme 2D, quite à compliquer le système. C'est ainsi que ,courant 1987, l'ISO publia la définition de GKS-3D, extension de GKS pour la visualisation tridimensionnelle.

Dans les paragraphes suivants nous présentons une étude détaillée du standard GKS actuel (2D) qui est maintenant un système bien établi et dont la diffusion s'accroit d'année en année. Ensuite, nous évoquerons rapidement GKS-3D et les problèmes qu'il soulève.

4.4.2 Le concept de poste de travail dans GKS

4.4.2.1 Notion de poste de travail

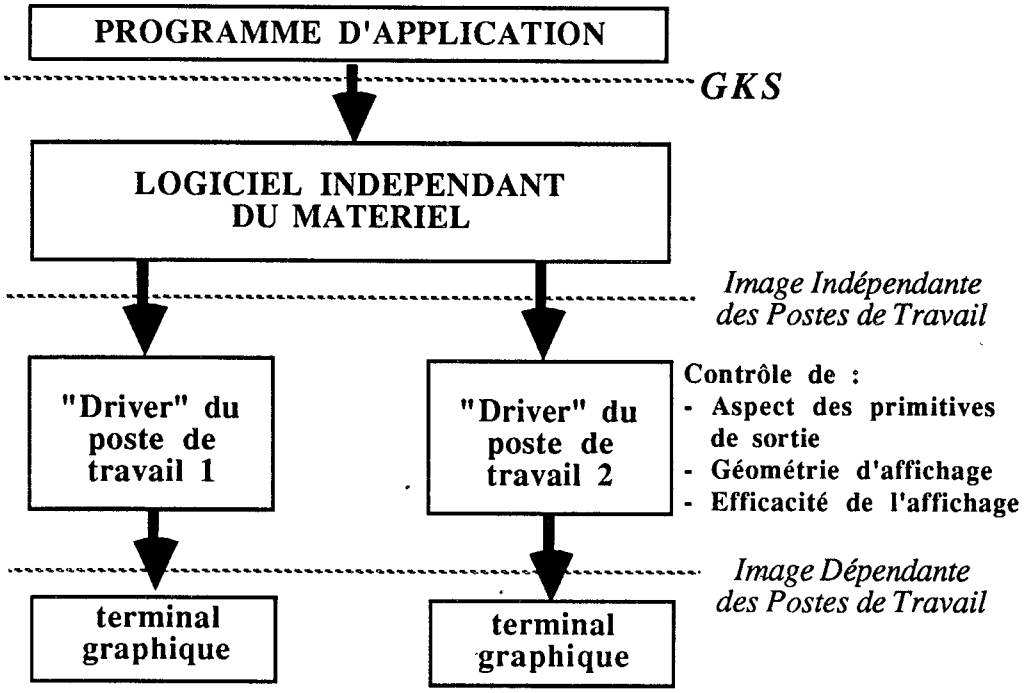
La notion de **poste de travail** ("workstation") est centrale dans GKS et constitue l'une des différences conceptuelles essentielle avec la proposition de CORE-SYSTEM du GSPC [EEKN 80].

Un poste de travail représente l'interface logique par laquelle le programme d'application contrôle les dispositifs graphiques. Il peut regrouper un ou plusieurs dispositifs physiques qui sont utilisés de manière **coordonnée** par l'opérateur. Cet ensemble (composé d'au plus une

surface de visualisation et d'un nombre quelconque de périphériques d'entrée) est traité dans son entier comme une unité logique et une seule par GKS.

GKS par l'intermédiaire de la notion de poste de travail divise le processus de visualisation en deux étapes (fig II.12) :

- dans un premier temps une image indépendante des postes de travail est construite,
- dans un second temps, pour chaque poste de travail actif, ces données indépendantes sont combinées avec des données dépendantes du poste de travail et transmises aux dispostifs d'affichage associés à chacune de ces stations.



- figure II.12 : Controle de l'affichage dans GKS à l'aide de la notion de poste de travail -

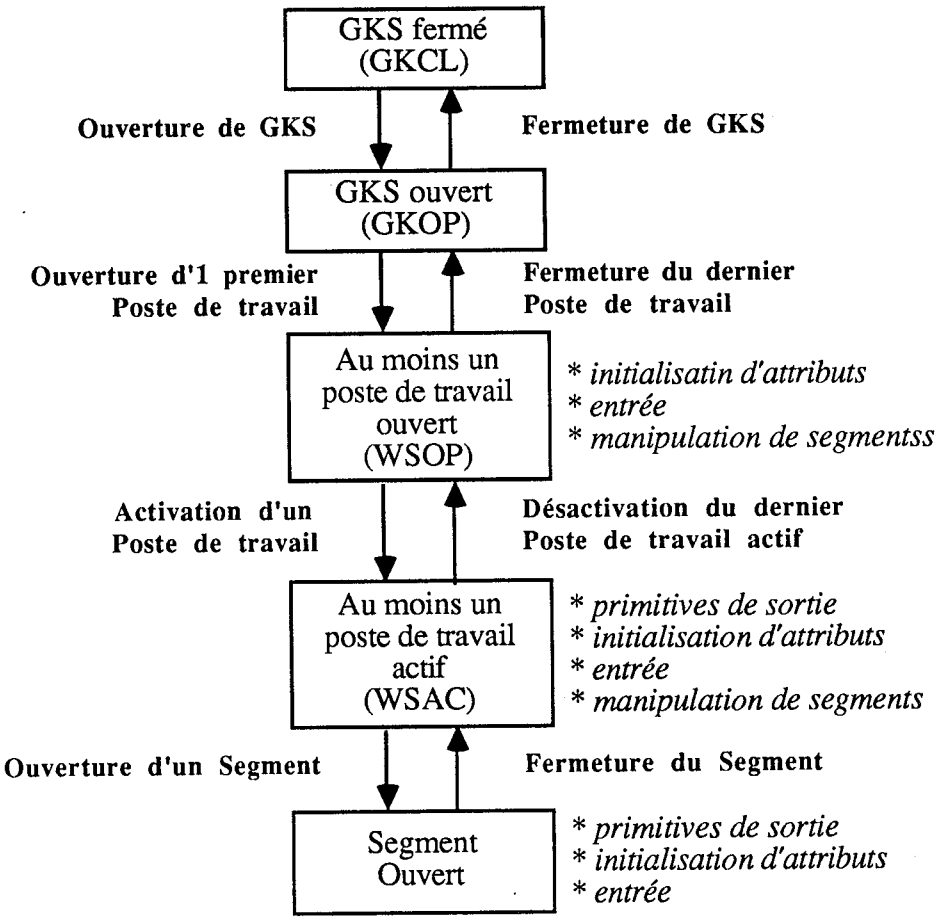
Grâce à ce concept, l'application peut contrôler de manière indépendante au niveau de chaque poste de travail :

- l'aspect des objets graphiques (table de groupage),
- la couleur des objets graphiques,
- la geométrie de l'affichage (transformation de poste de travail),
- l'efficacité de l'affichage (affichage différé).

Ce mécanisme offre, comme nous le verrons plus en détail par la suite, une plus grande souplesse que le CORE-SYSTEM et permet une meilleure prise en compte des différentes possibilités du matériel sans cependant affecter l'indépendance de l'application par rapport à celui-ci.

4.4.2.2 utilisation des postes de travail et transitions d'états dans GKS

Les postes de travail sont identifées par un numéro. Avant de pouvoir être utilisé, un poste de travail doit être explicitement ouvert. Cette action d'ouverture, associe le poste de travail virtuel au terminal graphique correspondant et donne accès à toutes ses possibilités, excepté l'affichage pour lequel le poste de travail doit être explicitement activé. La figure II.13 résume les différentes transitions d'état dans GKS et présente les actions autorisées dans chacun de ceux-ci.



- figure II.13 : les différents états de GKS et les actions autorisées -

Pour chacun des différents postes de travail présents sur une implémentation donnée, GKS conserve une table de description définissant ses possibilités et caractéristiques. Bien entendu ces tables sont figées et ne peuvent être modifiées. L'application peut toutefois consulter l'information qu'elles contiennent et adapter ainsi dans une certaine mesure son comportement en conséquence afin d'utiliser au mieux les possibilités du matériel.

4.4.3 Attribution

4.4.3.1 Morphologie - Primitives de sortie

Comme avec le CORE-SYSTEM, dans GKS les images sont construites à partir d'objets de base définis par l'invocation de fonctions primitives de sortie ("output primitives"). Mais à la différence du CORE-SYSTEM, GKS n'utilise pas la notion de position courante (primitives MOVE et DRAW). Ainsi, chaque primitive de sortie a ses coordonnées entièrement définies de façon interne et se suffit donc à elle même sans dépendre des autres.

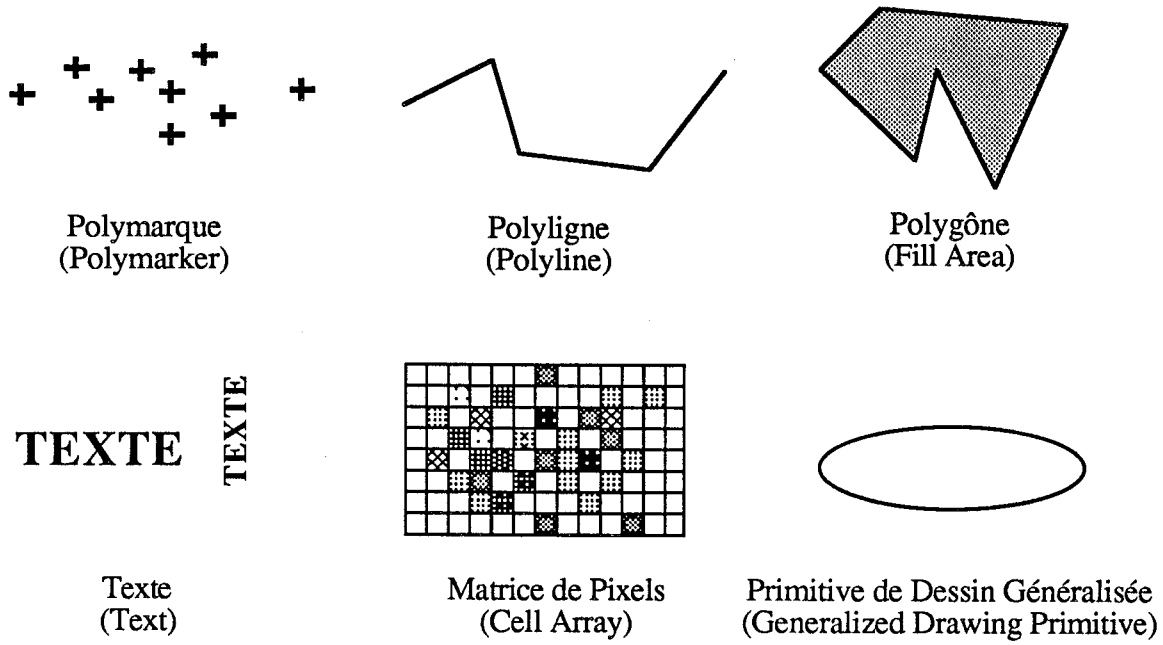
La figure II.14, page suivante, représente les six types de primitives de sortie de GKS qui sont :

* **Polymarque ("Polymarker")** : marque un séquence de points par un même symbôle.

* **Polyligne ("Polyline")** : defini une ligne brisée reliant une séquence de points.

* **Texte ("Text")** : dessine une chaîne de caractères à partir d'une position donnée.

* **Polygône ("Fill_Area")** : défini une tâche polygônale qui pourra être remplie avec une couleur uniforme, un motif ou des hachures.



- figure II.14 : les primitives de sortie dans GKS -

* **Matrice de Pixels ("Cell_Array")** : génération d'une matrice de pixels, chacun affecté d'une couleur particulière. (Cette primitive a été rajoutée afin de prendre en compte des consoles à balayage de trame et de considérer des applications du type analyse d'images ("image processing") [Stol 84]).

* **Primitive Graphique Généralisé ("Generalized Drawing Primitive")** : cette primitive de sortie est proposée par GKS afin de pouvoir utiliser efficacement certains dispositifs logiciels ou matériels de postes de travail particuliers (par exemple un générateur de cercles). Les objets sont alors caractérisés par un identificateur, un ensemble de points et de données complémentaires. GKS applique seulement les transformations géométriques à ces points, laissant leur interprétation au poste de travail. Les primitives graphiques généralisées sont très dépendantes de stations spécifiques et n'ont que très peu de signification en dehors d'une application et d'un environnement donné, leur portabilité n'étant pas garantie .

Les coordonnées passées en paramètres des primitives de sortie sont exprimées dans un repère propre à l'application (WCS). Les primitives de sortie sont transmises vers tous les postes de travail actifs au moment de leur invocation.

4.4.3.2 Aspect - Géométrie - Attributs des primitives de sortie

Un certain nombre d'attributs qui reprennent dans une large mesure ceux définis dans le CORE-SYSTEM, permettent de contrôler l'apparence des primitives de sortie sur les différents postes de travail où elles sont affichées (voir table II.7). L'originalité de GKS provient des différentes possibilités offertes pour leur spécification, en particulier du report d'une partie de leur contrôle au niveau de chaque poste de travail.

PRIMITIVE DE SORTIE	ATTRIBUTS
Polyligne (Polyline)	Type de Trait *(Linetype)* Index de Couleur *(Color Index)* Coeff. d'epaisseur du Trait *(Linewidth Scale Factor)* Index de Polyligne *(Polyline Index)*
Polymarque (Polymarker)	Type de marqueur *(Marker Type)* Coeff. de Taille de Marqueur *(Marker Size Scale Factor)* Index de Couleur *(Color Index)* Index de Marqueur *(Polymarker Index)*
Texte (Text)	Attributs non Géométriques : Couleur *(Color)* Espacement inter caractères *(Spacing)* Facteur d'agrandissement *(Expansion)* Police *(Font)* Precision du texte *(Precision)* (string, char, stroke) Index du texte *(Index)* Attributs Géométriques : Hauteur de caratères *(Character-Height)* Vecteur d'Orientation *(Character-up Vector)* Direction d'écriture *(Text Path)* Alignement du texte *(Text Alignment)* (horizontal : centré, cadré à gauche, cadré à droite vertical : haut,bas,mi-hauteur base,crète)
Polygone (Fill Area)	Attributs non Géométriques : Couleur *(Color)* Style de Remplissage *(Intérior Style)* (vide, plein, hachuré,motif) Index de Style *(Style Index)* Index des Polygônes *(Fill Area Index)* Attributs Géométriques : Point de Référence du Motif *(Pattern Reference Point)* Taille du Motif *(Pattern Size)*
Tableau de Pixels (Cell Array)	Aucun
Primitive de Dessin Généralisée (Generalized Drawing Primitive)	Dépend de l'installation

- table II.7 : Les attributs des primitives de sortie dans GKS -

On distingue deux types d'attributs de primitives de sortie dans GKS :

- **les attributs géométriques** : qui contrôlent la taille et la forme des primitives de sortie,

- **les attributs non géométriques** : qui contrôlent l'aspect des primitives de sortie.

les attributs géométriques

Les attributs géométriques concernent uniquement le texte et les motifs de remplissage des polygônes. Chaque attribut géométrique est contrôlé indépendamment des autres par une primitive d'attribution associée. Ceux-ci sont définis **modalement**. Les attributs géométriques des primitives de sortie sont exprimés en coordonnées universelles (WCS) et sont **indépendants** des postes de travail .

les attributs non géométriques

Les attributs non géométriques des primitives de sortie peuvent être contrôlés de deux manières différentes : **individuelle** ou **groupée**.

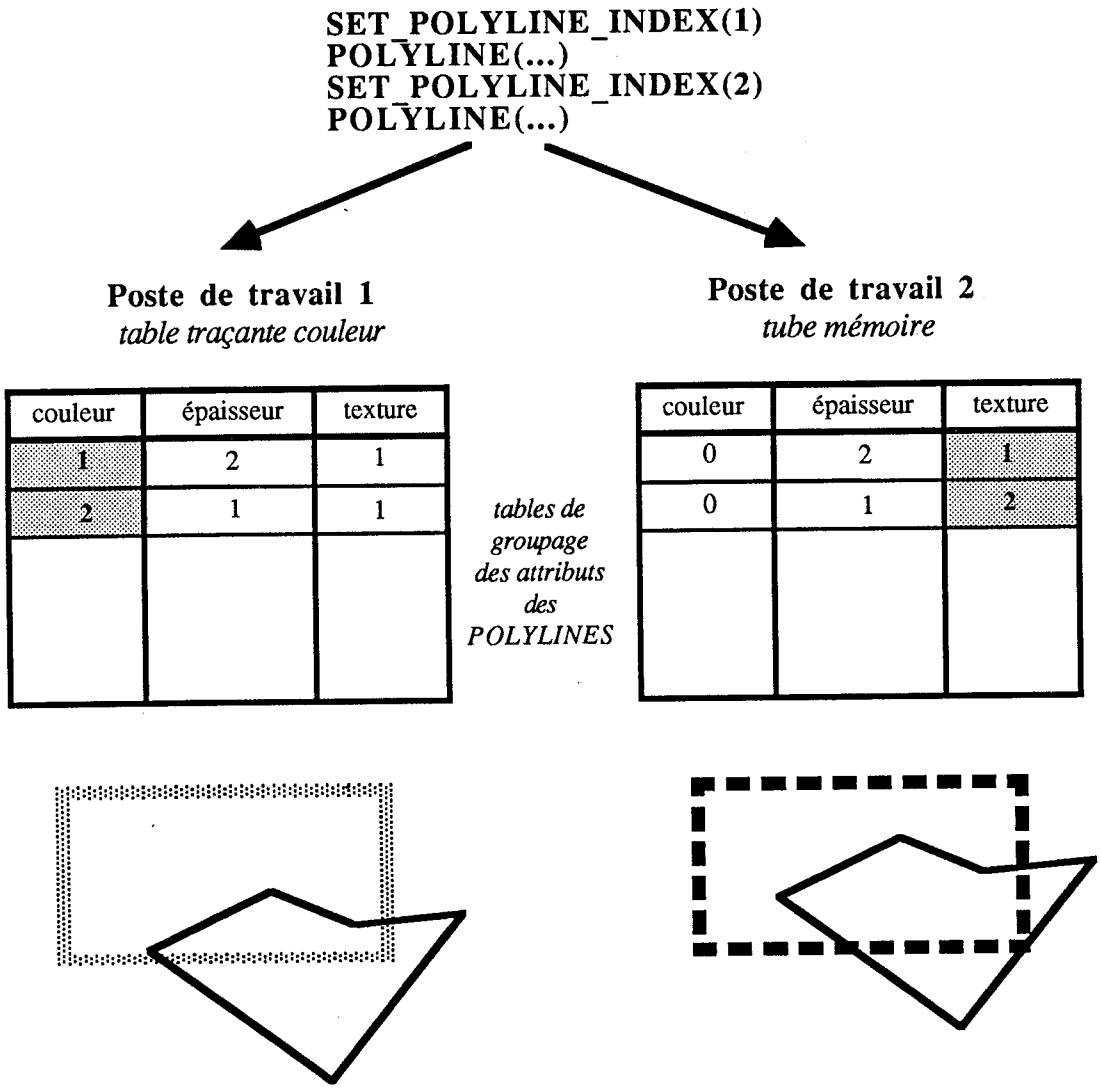
* **les attributs individuels** : chaque attribut non géométrique d'une primitive de sortie est défini séparément et modalement par une fonction d'attribution particulière. Comme pour les attributs géométriques, les attributs définis ainsi sont indépendants des postes de travail.

Cette approche conventionnelle (cf. le CORE-SYSTEM), d'un emploi simple et direct dans le cas d'un environnement avec un seul poste de travail, peut présenter certains inconvénients lors de l'utilisation simultanée de plusieurs dispositifs de visualisation. En effet la valeur d'un attribut est la même pour tous les postes de travail sur lesquels la primitive de sortie est affichée, mais les différents dispositifs physiques ne sont pas toujours capables de l'interpréter de manière équivalente (par exemple la couleur n'a pas de signification sur une console à tube mémoire).

Pour permettre à l'application une meilleure prise en compte des spécificités des différents dispositifs d'affichage, GKS propose grâce au concept de poste de travail un deuxième mode de définition des attributs [BEEH 82]:

* **les attributs groupés ("bundle attributes") ou attributs liés aux postes de travail** : dans ce cas **tous** les aspects non géométriques d'une primitive de sortie sont définis par un simple attribut qui est un index vers une "table de groupage" qui contient une représentation des attributs **dépendante** des postes de travail. Chaque poste de travail possède ses propres tables de groupage pour les attributs des différents types de primitives de sortie. Le programme d'application peut modifier les valeurs contenues dans ces tables, il est donc possible d'avoir des représentations différentes pour les attributs sur des postes de travail différents.

Par exemple (voir la figure II.15 qui suit), si l'opérateur utilise parallélement une console à tube mémoire pour une visualisation rapide et une table traçante couleur pour une sortie soignée il pourra adapter les attributs des primitives de sortie à chacun de ces postes de travail. Ainsi les représentations des attributs pour les index de Polylines différeront principalement de par leurs indices de couleur pour la table traçante, alors que pour le tube mémoire les représentations pour les mêmes indices se distinguerons par la texture du trait .



- figure II.15 : Tables de groupage des attributs dans GKS -

remarque : la couleur et les motifs de remplissage des polygônes sont également controlés à l'aide de tables propres à chaque poste de travail. Certaines des entrées de ces tables sont prédéfines, ainsi par exemple l'entrée 0 de la table des couleurs correspond à la couleur du fond et l'entrée 1 à la couleur de tracé par défaut. Les couleurs sont définies dans la table par la donnée de leurs trois composantes Rouge, Vert, Bleu (RVB).

GKS permet de combiner les deux manières de travailler (attributs individuels, attributs groupés) à l'aide d'indicateurs d'origine d'aspect ("Aspect Source Flags" ou "ASF") qui contrôlent si la valeur d'un attribut est individuelle ou non [GiDu 85].

Les attributs des primitives de sortie qu'ils soient géométriques ou non, sont définis de manière **statique** : ils ne peuvent plus être modifiés après la création de la primitive. Cependant l'utilisation d'attributs groupés permet un certain degré de modification. En effet si il n'est plus possible changer la valeur d'index pour une primitive de sortie, l'application peut néanmoins modifer la représentation des attributs dans les tables de groupage des postes de travail.

4.4.3.3 Géométrie de prise de Vue (Gv) Géométrie d'affichage (Ga) - Transformation de visualisation

Plutôt que de définir comme dans de nombreux packages graphiques, une simple transformation fenêtre clôture des coordonnées universelles propres à l'application vers les

coordonnées des divers dispositifs d'affichage, GKS considère **trois** systèmes de coordonnées et grâce au concept de poste de travail divise la transformation de visualisation en **deux étapes.**

Les trois systèmes de coordonnées sont :

- l'**espace universel (WCS)** : système de coordonnées utilisateur propre à l'application et dans lequel sont définies les primitives de sortie.

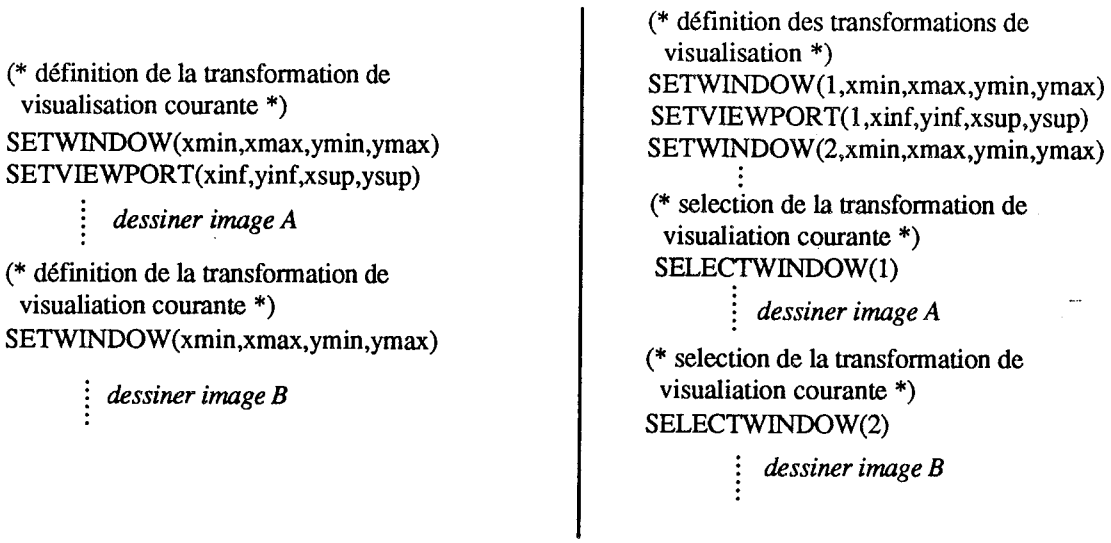
- l'**espace écran normalisé (Normalized Device Coordinate Space NDC)** : système de coordonnées normalisées qui permet une définition uniforme des systèmes de coordonnées pour tous les postes de travail (écran virtuel de bornes 0..1).

- l'**espace de coordonnées du (des) dispositif(s) d'affichage (Device Coordinate Space DC)** : coordonnées du dispositif d'affichage, propre à chaque poste de travail.

La transformation de visualisation se sépare donc en deux étapes autour de l'espace écran normalisé (NDC) (voir fig II.16 ci-contre) :

* la **transformation de normalisation** qui permet de passer des coordonnées universelles aux coordonnées normalisées .Cette transformation est spécifiée de manière classique par une fenêtre ("window") dans l'espace universel et une clôture ("viewport") dans l'espace normalisé. Elle est appliquée aux primitives de sortie dès leur définition. Cette transformation est indépendante du (des) poste(s) de travail et peut être réinitialisée à n'importe quel moment par le programme d'application.

Cependant au lieu d'associer la transformation de normalisation directement à une clôture et une fenêtre courante (la modification de la clôture ou de la fenêtre entrainant une modification implicite de la transformation courante) GKS offre la possibilité de définir et **conserver** des transformations **multiples** qui sont référencées par un index (fig II.17). La définition des systèmes de coordonnées est ainsi indépendante du choix de la transformation courante.

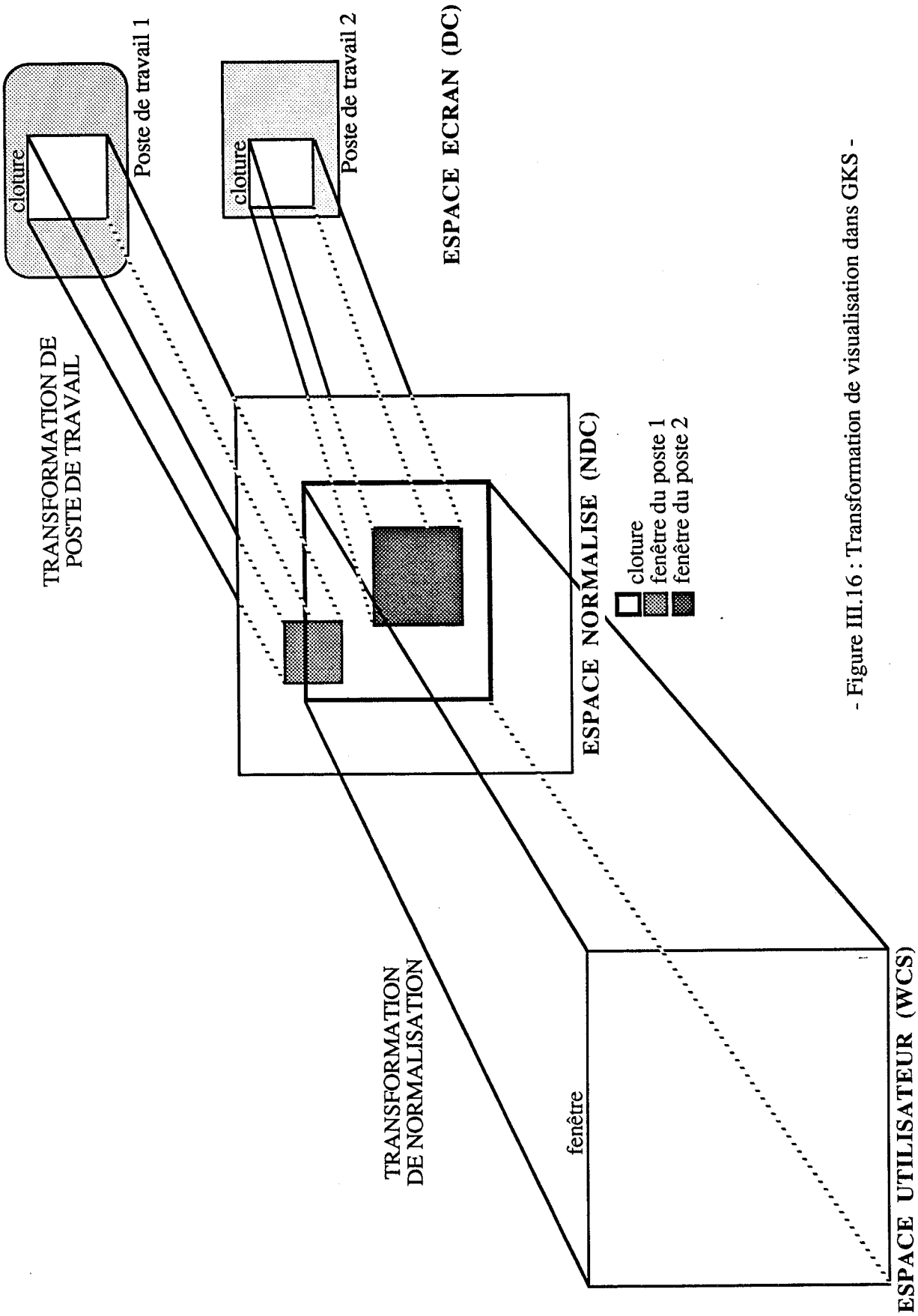


Définition Modale de la transformation de Visualisation (CORE-SYSTEM)

Définition de transformations de visualisation multilples (GKS)

- figure II.17 : spécification de la transformation de normalisation dans GKS -

* la **transformation de poste de travail (workstation transformation)** qui dans un deuxième temps effectue le passage des coordonnées normalisées (NDC) aux coordonnées des dispositifs d'affichage (DC) A chaque poste de travail est associé sa propre transformation,



- Figure III.16 : Transformation de visualisation dans GKS -

qui peut être réinitialisée indépendamment des autres. Celle-ci est définie par une "fenêtre de poste" ("workstation window") exprimée dans l'espace normalisé et une clôture ("workstation viewport") exprimée dans le système de coordonnées du poste de travail.

Cette séparation en deux étapes de la transformation de visualisation offre une plus grande flexibilité, par exemple il est ainsi possible de visualiser simultanément les mêmes primitives graphiques sur des postes d'affichage différents avec des échelles différentes. Mais, par contre au contraire de ce qui avait été fait pour GRI-GRI (cf. § 4.1.2.4), cette transformation peut introduire une déformation du dessin, le système faisant coincider systématiquement la fenêtre avec la cloture. Pour éviter cela, le programme d'application doit prendre soin de définir des transformations de visualisation avec des fenêtres et clotures dont le rapport de taille est identique.

4.4.3.4 Structure et Identité - Segments

Comme le CORE-SYSTEM, GKS utilise comme **seul** mécanisme de structuration de l'image, la notion de segment. Les primitives de sortie peuvent ainsi soit être envoyées directement vers les dispositifs d'affichage (cela signifie qu'après une régénération de l'image elles seront perdues), soit regroupées et mémorisées dans des segments. Les segments constituent dans GKS l'unité de **manipulation** et de **modification**.

Manipulation des segments

Lorsqu'un segment est défini (OPEN_SEGMENT), les primitives de sortie, **après** avoir été transformées à partir des coordonnées universelles (WCS) en coordonnées normalisées (NDC), sont "distribuées" vers **tous** les postes de travail actifs où elles sont stockées avec leurs attributs. Chaque poste de travail possède son propre mode de stockage des segments (WDSS "Workstation Dependend Segment Storage"). Après la fermeture d'un segment (CLOSE_SEGMENT) il n'est plus possible de lui ajouter ou supprimer des primitives de sortie graphique, pas plus que de changer leurs attributs.

Les segments sont identifiés par un entier unique (0..32387), l'application peut à tout moment renommer un segment (RENAME_SEGMENT), le détruire (sur un poste de travail particulier ou sur tous les postes de travail actifs au moment de sa création).

Attributs dynamiques des segments

Si l'application ne peut plus modifier le contenu d'un segment après sa fermeture, il lui est néanmoins possible d'agir sur son apparence globale par l'intermédiaire des attributs **dynamiques** qui lui sont associés. Les attributs définis à ce niveau reprennent dans une large mesure ceux du CORE-SYSTEM, on retrouve la **Visibilité** ("Visibility"), la **Surbrillance** ("Highlightning"), la **Détectabilité** ("Detectability") et la **Transformation d'Image** ("Segment Transformation").

Ce dernier attribut, permet de modifer la position d'un segment dans l'espace normalisé (par translation, rotation et mise à l'echelle). Cette transformation géométrique (de l'espace NDC vers lui-même) est conservée dans la liste d'état du segment et appliquée aux primitives contenues dans le segment avant tout affichage de celles-ci.

GKS introduit un nouvel attribut de segment : la **Priorité** ("Priority"). Il permet d'associer un numéro de priorité aux segments. Ainsi, la liste des segments est accédée de la priorité la plus faible vers la priorité la plus forte pour la visualisation et inversement pour l'identification. De cette manière, si des primitives de sortie à l'intérieur de deux segments se recouvrent (par exemple primitives FILL_AREA), celles avec la plus haute priorité sont visibles et sont celles reconnues lors d'une identification.

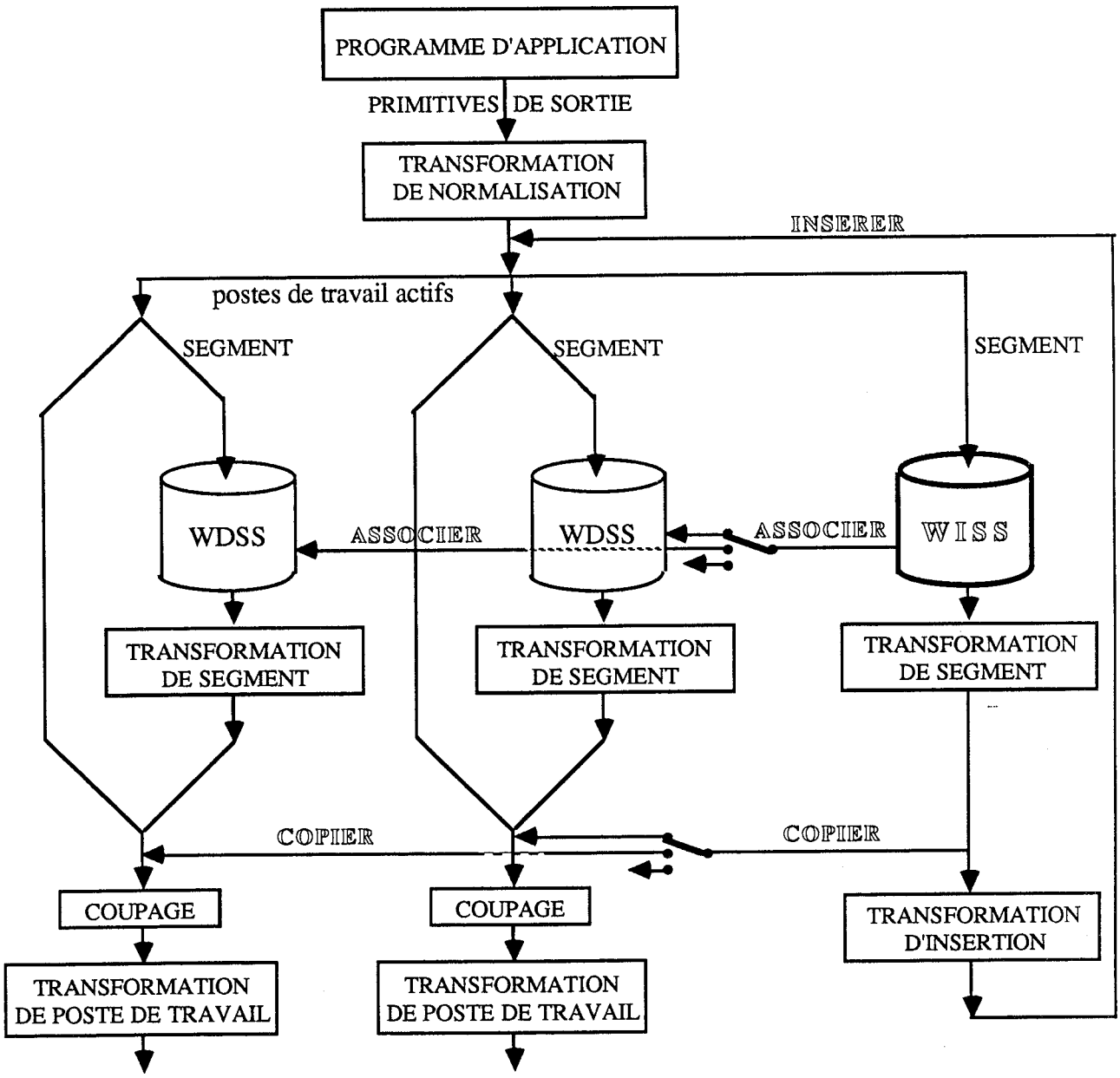
L'application peut à tout instant modifier les attributs des segments créés auparavant. Les attributs dynamiques des segments sont uniques et ne varient pas selon les postes de travail.

Identification

Le segment constitue l'unité d'identification dans GKS, les primitives en dehors d'un segment ne pourront être détectées. Mais comme dans le CORE_SYSTEM, il est possible d'ajouter un deuxième niveau de noms en associant un "pick-identifer" aux primitives de sortie à l'intérieur des segments. (le "pick-identifier" est un attribut des primitives de sortie, comme tel il ne peut plus être modifié après la création de celles-ci).

Stockage des segments indépendants des postes de travail

GKS considère un type de poste de travail particulier (un seul par implémentation) , le poste de stockage des segments indépendants (WISS Workstation Independent Segment Storage) qui permet de **conserver** des segments qui peuvent par la suite être **réutilisés** sur d'autres postes de travail. Il est ainsi possible de visualiser les primitives qu'ils contiennent sur des postes de travail qui n'étaient pas actifs au moment de leur création. Les actions possibles sont les suivantes (voir fig. II.18) :



- fig II.18 : Stockage des segments sur le poste de travail indépendant (WISS) -

- **copier** un segment vers un poste de travail : les primitives d'un segment du WISS sont sorties vers le poste de travail spécifié.

- **associer** un segment à un poste de travail : copie un segment du WISS vers le WDSS (Workstation Dependent Segment Storage) du poste de travail spécifié.

- **insérer** un segment sur le (les) poste(s) de travail actif(s) : les primitives mémorisées dans le segment du WISS sont replacées dans le flux de sortie de GKS. Elles, peuvent par exemple être réintroduites dans le segment courant (le segment actuellement ouvert).

4.4.4 Description - Fonctions d'Entrée ("Input Primitives")

Comme le CORE_SYSTEM, GKS offre des fonctions d'entrée d'assez bas niveau qui permettent à l'application de gérer les dispositifs de dialogue indépendamment de leurs caractéristiques physiques.

Le modèle proposé par GKS [RMPK 82] est un raffinement de celui élaboré pour le CORE_SYSTEM et traite les périphériques d'entrée en terme de dispositifs logiques utilisables selon les trois modes de fonctionnement synchrone (ou requête), asynchrone et échantillonné. Au contraire du CORE-SYSTEM, le mode de fonctionnement est indépendant de la classe logique du dispositif. Il peut être réinitialisé à tout moment par l'application (fonction SET_<input_class>_MODE).

La table II.8 , présente les six classes de dispositifs d'entrée considérées par GKS et la valeur logique qu'elles fournissent.

CLASSE	Valeur logique d'entrée
LOCATOR	une position en coordonnées universelles et un numéro de transformation normalisée
STROKE	suite de positions en coordonnées universelles et un numéro de transformation de normalisation
VALUATOR	valeur réelle
CHOICE	valeur entière correspondant à un choix parmi plusieurs alternatives
PICK	numéro de segment et un pick identifier
STRING	chaîne de caractères

- table II.8 : Les classes de dispositifs logiques dans GKS -

La seule différence notable par rapport aux dispositifs logiques du CORE-SYSTEM concerne les releveurs de coordonnées (classes LOCATOR et STROKE) qui dans GKS renvoient des coordonnées universelles (au lieu de coordonnées normalisées (NDC) dans le CORE). Ceci est rendu possible par le fait que GKS permet de mémoriser les différentes transformations de normalisation définies par l'application (cf. § 4.4.3). Ainsi dans le cas de l'affichage simultanné de plusieurs images définies chacune dans leur propre système de

coordonnées, il est possible de restituer à l'application des coordonnées universelles exprimées dans le repère approprié. Pour ce faire, l'application peut ordonner les transformations selon un ordre de priorité, la transformation des coordonnées écran en coordonnées universelles s'effectuant comme suit :

- transformation des coordonnées écrans (DC) en coordonnées normalisées (NDC) en appliquant l'inverse de la transformation de poste de travail,

- parcours de la liste des transformations de normalisation par ordre de priorité décroissant, jusqu'à ce que la position en coordonnées normalisées "tombe" dans la clôture associée,

- application aux coordonnées normalisées de la transformation de normalisation inverse.
Le numéro de transformation de normalisation retourné par ces dispositifs correspond à la transformation qui a été appliquée aux coordonnées normalisées pour obtenir les coordonnées universelles (cf. § 4.4.3.3).

4.4.5 Visualisation

Le processus de visualisation dans GKS est **implicite** et consiste simplement à l'application des transformations de normalisation et de poste de travail. Bien entendu, pour le système 2D que nous présentons ici, aucun réalisme n'est pris en compte dans ce processus. GKS est inadapté à ce type de visualisation, l'application devant alors prendre en charge les traitements correspondants. Néanmoins, comme nous l'avons déjà évoqué, une extension 3D a été définie pour GKS, celle-ci tachant d'aborder de manière la plus satisfaisante possible les problèmes liés à la visualisation tridimensionnelle. Nous reviendrons plus en détail sur ceci au paragraphe 4.4.8, nous contentant de présenter ici les mécanismes proposés à l'application pour contrôler l'efficacité de l'affichage.

4.4.4.1 Le coupage

Dans GKS, le coupage est un peu particulier du fait de la séparation de la transformation de visualisation en deux étapes (voir § 4.4.3.3). Il s'effectue lui même en deux temps qui sont :

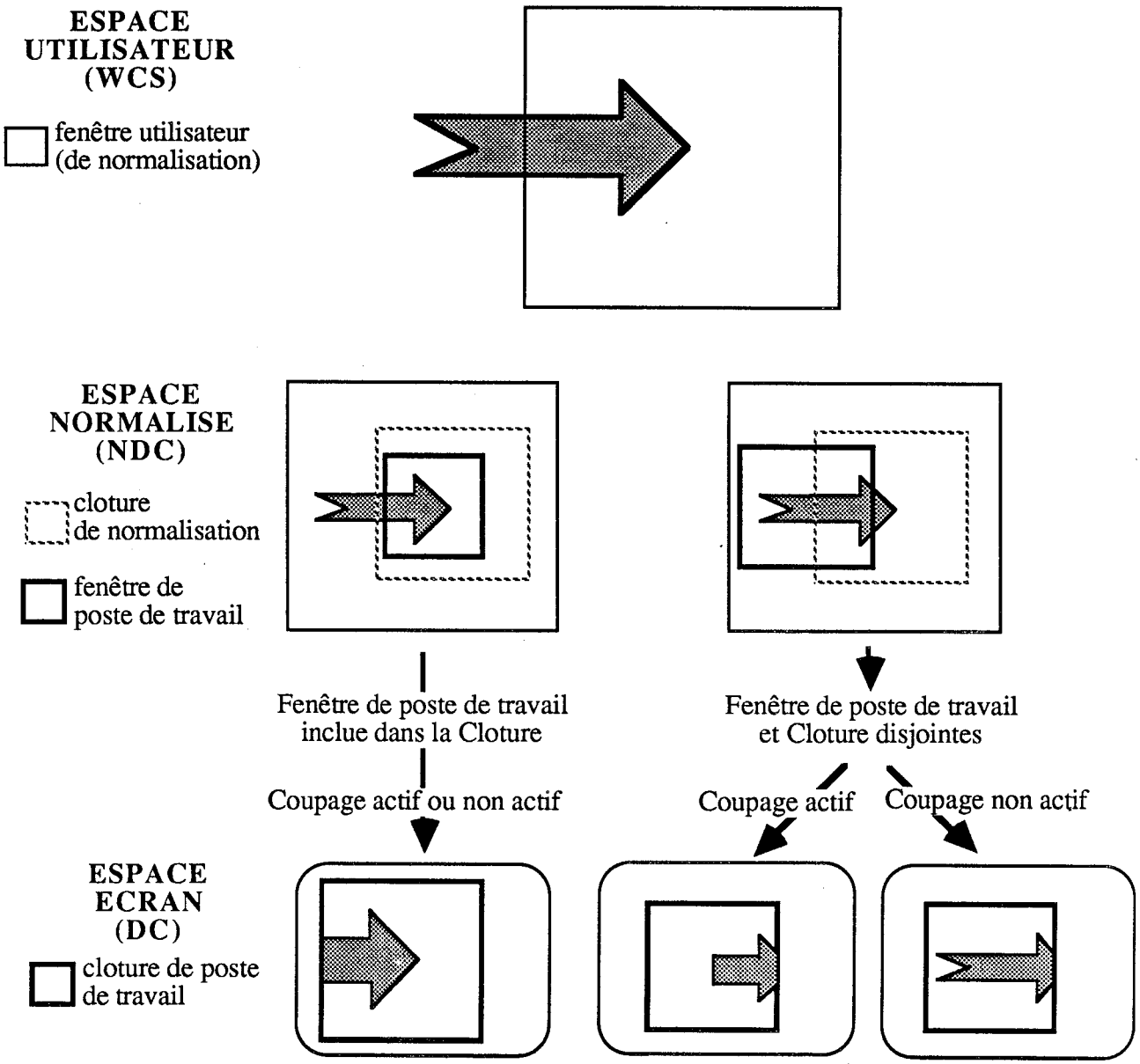
- un premier coupage par rapport à la clôture définissant la transformation de normalisation,
- un second coupage par rapport à la fenêtre définissant la transformation de poste de travail.

Le coupage selon la clôture "de normalisation" est activé ou désactivé lors de la définition des primitives de sortie. Cependant cette opération n'est pas réalisée pendant la transformation de normalisation (espace WCS ---> espace NDC), mais elle est retardée jusqu'à l'affichage effectif des primitives de sortie sur les différents postes de travail.

Ainsi les primitives de sortie mémorisées dans des segments sont stockées dans leur intégralité (en coordonnées NDC). Un rectangle de coupage leur est associé qui correspond à la clôture courante si le coupage est activé ou aux limites de l'espace normalisé sinon. Lors de l'affichage la transformation de segment est appliquée à ces primitives avant leur coupage. (remarque : le rectangle de coupage n'est pas affecté par cette transformation, aussi des objets se trouvant en dehors de celui-ci au moment de leur définition peuvent très bien devenir visibles par la transformation de segment !!!).

Le coupage par rapport à la fenêtre de poste de travail intervient après le coupage par rapport à la clôture "de normalisation". Celui-ci dépend de chaque poste de travail. Il est **toujours** effectué afin de garantir que l'affichage ne provoquera aucun débordement ("wraparound").

Ainsi pour être visible une primitive de sortie doit donc être à l'intérieur de la clôture de normalisation **et** à l'intérieur de la fenêtre de poste de travail. La figure II.19 résume les différents effets possibles du coupage selon qu'il est activé ou non et selon la position relative de la clôture de normalisation et de la fenêtre de poste de travail.



- figure II.19 : Les différents effets du coupage dans GKS -

4.4.4.2 L'affichage différé

Comme le CORE-SYSTEM du GSPC, GKS permet de contrôler l'efficacité de l'affichage en **différant** les modifications à apporter à l'image. Mais alors que le CORE contrôle le comportement du système graphique dans son entier, GKS reporte cette gestion au niveau des postes de travail, l'application pouvant ainsi s'adapter aux possiblités de **chacun** des dispositifs de visualisation qu'elle utilise.

Ce contrôle s'effectue au travers de la primitve SET_DEFERRAL_STATE (numposte, mode_différé,mode_régénération).Les paramêtres *mode_différé* et *mode_régénération* permettent de considérer séparemment les modifications consistant en de simples ajouts à l'image (primitives de sortie, copie, association, insertion de segments) de celles entraînant une "altération" de l'image existante (changement des valeurs des attributs dans les tables de groupage,modification de la transformation du poste de travail, des attributs dynamiques des segments, destruction de segments visibles...).

Pour ce qui concerne la prise en compte de l'effet visuel des fonctions de sortie, GKS permet un contrôle plus affiné que le CORE-SYSTEM au travers de valeurs possibles pour le paramètre *mode_différé* qui sont :

- dès que possible (ASAP "As Soon As Possible"),

- avant la prochaine interaction sur n'importe quel poste de travail (BNIG "Before Next Interaction Globally"),

- avant la prochaine interaction sur **le** poste de travail (BNIL "Before Next Interaction Locally"),

- à un instant dépendant de l'implémentation (ASTI "At Some TIme").

Le contrôle des altérations de l'image est effectué à l'aide du paramètre de mode de régénération implicite (*mode_régénération*) qui peut être soit :

- **autorisé** : la régénération de l'image à lieu pour chaque action la nécéssitant,

- **suprimé** : la régénartion implicite de l'image et interrompue jusqu'à ce qu'une demande explicite de mise à jour (UPDATE_WORKSTATION) ou un changement approprié de mode soit effectué (SET_DEFFERAL_STATE).

Certaines modifications peuvent être executées immédiatement sur certains postes de travail, alors que sur d'autres, leur prise en compte nécessite une régénération complète de l'image. Afin de permettre à l'application de s'adapter le mieux possible au matériel utilisé, dans la table de description de chaque poste de travail, existent des entrées qui indiquent quelles modifications :

- impliquent une régénération implicite (IRG "Implicit Regeneration"), (par exemple destruction d'un segment visible sur une console à tube mémoire)

- peuvent être effectués immédiatement (IMM) (par exemple modification de couleur sur console avec tables de couleurs).

4.4.6 Consultation

La consultation dans GKS consiste essentiellement à l'interrogation de l'état courant du système. Ainsi, n'est-il pas possible de consulter les attributs des différentes primitives de sortie stockées dans les segments. Seuls sont accessibles les attributs dynamiques de ces derniers et les réprésentations des attributs dans les tables de "groupage" des différents postes de travail.

4.4.7 Implémentation de GKS

Différents niveaux d'implémentation ont été définis pour GKS afin de prendre en compte le mieux possible les gammes les plus courantes de matériel et d'applications. Ces niveaux sont décrits selon deux modules orthogonaux : l'entrée et la sortie (table II.9).

NIVEAUX DE SORTIE*	NIVEAUX D'ENTREE• A	B	C
0	• pas d'entrées * Sortie minimale (pas de segments) Contrôle minimum	• mode requête uniquement	• entrée complètes : mode requête, echantillonage, événement
1	*Segmentation de base		
2	* Poste de stockage de segments indépendants (WISS)		

- table II.9 : Les différents niveaux d'implémentation de GKS -

Pour chacun de ces modules 3 niveaux ont été définis avec des possibilités croissantes (chaque niveau engloble les possibilités des niveaux inférieurs). Une mise en oeuvre de GKS devra réaliser exactement toutes les possibilités (et uniquement celles-ci) de l'un des 9 niveaux fonctionnels ainsi définis pour être valide et conforme à la norme [Scot 84].

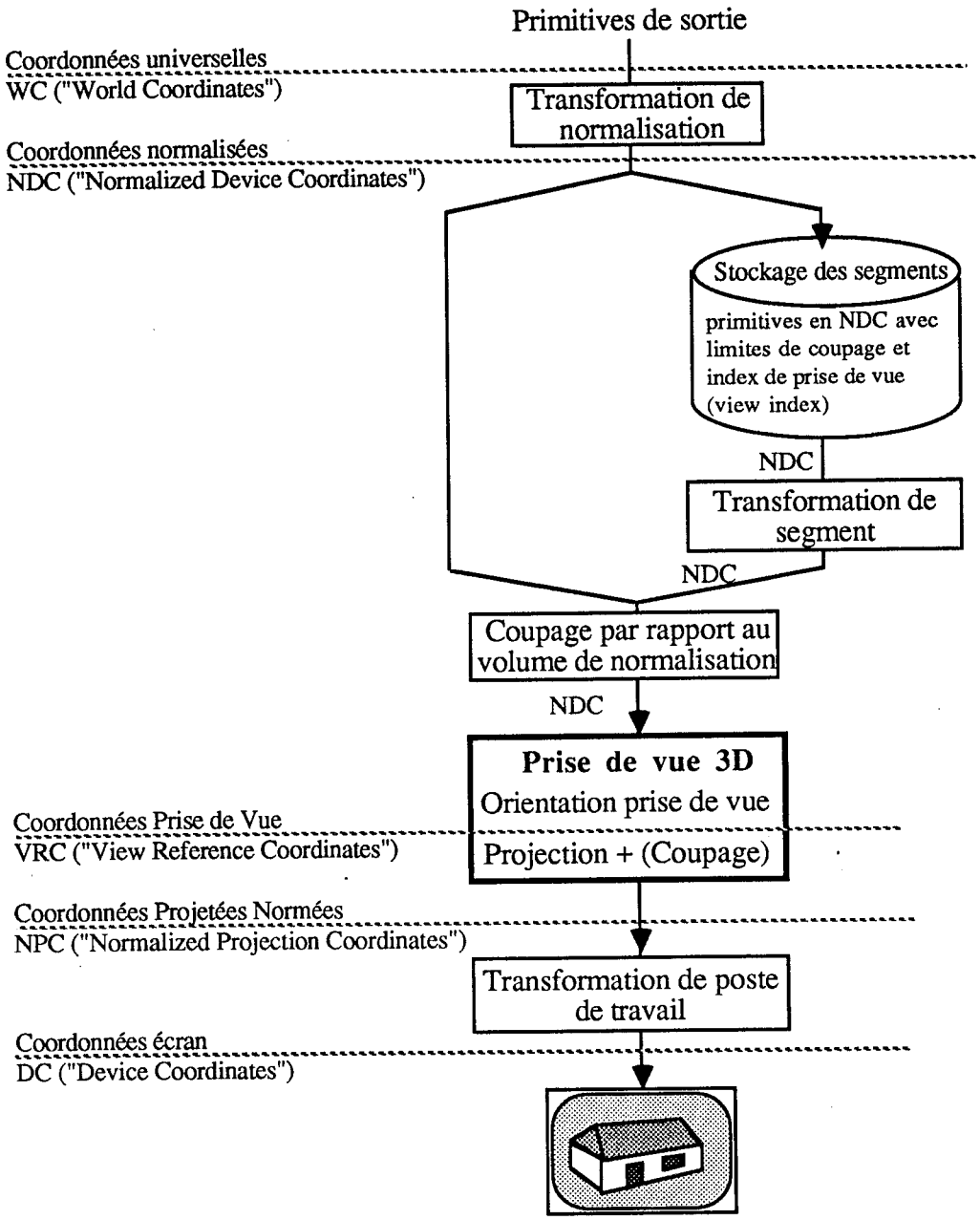
4.4.8 GKS-3D

GKS-3D [ISO 85-b] est la spécification d'un système offrant un ensemble de fonctions de base pour la visualisation 3D, tout en conservant une compatibilité totale avec la norme GKS (2D) (c'est à dire qu'il doit permettre l'execution correcte de tout programme destiné à un environnement GKS).

GKS-3D a donc été conçu comme une **extension** de GKS [Stra 85] avec pour contrainte majeure la non remise en cause des concepts du standard. Aussi, en plus de nouvelles fonctions spécifiques pour le 3D (primitives de sortie, prise de vue 3D...), il supporte toutes les fonctions de GKS. Cependant si GKS-3D permet de mêler des fonctions 2D à des fonctions 3D, il n'en demeure pas moins un véritable système 3D: lors de la visualisation, toutes les opérations géométriques s'effectuent dans l'espace tridimensionnel. Ainsi, les informations bidimensionnelles fournies à l'aide des fonctions GKS sont immédiatement converties en ajoutant une composante z=0 aux coordonnées et en insérant une ligne et une colonne identité aux matrices de transformation 3x3 afin d'obtenir des matrices 4x4. Par exemple, la primitive de sortie de type "Polyline" représente une suite de segments reliant une série de points 3D et peut être définie soit par la nouvelle fonction POLYLINE -3, soit à l'aide de la fonction GKS POLYLINE (dans ce cas, la primitive est dans le plan xOy (z=0)).

De manière analogue, GKS-3D reprend toutes les primitives de sortie de GKS ainsi que leurs attributs géométriques (tels l'orientation de texte qui s'effectue dans l'espace 3D). Les attributs d'aspect de ces primitives sont en tout points identiques à ceux de GKS. Par ailleurs, GKS-3D considère un nouveau type de primitive de sortie : FILL-AREA SET qui permet de définir des faces polygonales planes avec trous par la donnée d'un ensemble de contours.

La principale difficulté technique dans la définition de GKS-3D a été l'intégration de la prise de vue 3D au processus de visualisation tout en conservant l'homogénéité avec celui de GKS, c'est à dire, les deux étapes Transformation de Normalisation et Transformation de poste de travail. La figure II.20 présente les différentes étapes du processus de visualisation adopté pour GKS-3D.



- figure II.20 : Les étapes géométriques du processus de visualisation dans GKS-3D -

La transformation de normalisation assure, comme dans GKS, le passage de l'espace universel à un espace normé dans lequel les images peuvent être composées. Mais ici, ces deux espaces étant tridimensionnels, la transformation est spécifiée par la donnée de deux parallèlépipèdes rectangles définissant un volume de départ dans l'espace universel et un volume d'arrivée dans l'espace normalisé.

Cette transformation est appliquée aux primitives de sortie dès leur définition, qui sont donc transmises vers les différents postes de travail sous une forme normalisée.

De la même manière que pour GKS, la transformation de segment exprime une transformation géométrique de l'espace normalisé vers lui-même et est appliquée aux primitives conservées à l'intérieur de segments. Comme l'espace normalisé est tridimensionnel, cette transformation est définie par une matrice 4x4 en coordonnées homogènes.

Après l'application de la transformation de normalisation et de l'éventuelle transformation de segment, les primitives de sortie peuvent être coupées par rapport au volume définissant la "clôture normalisée 3D".

Les opérations de prise de vue 3D constituent l'étape suivante du processus de visualisation et s'appliquent donc à l'image (coupée) en coordonnées normalisées. Cette transformation est dépendante des postes de travail et est définie dans une table propre à chaque poste. Le choix d'une transformation de prise de vue s'effectue par un index (SET-VIEW INDEX) associé à chaque primitive de sortie. La première entrée de chaque table contient une transformation par défaut qui correspond à l'identité et qui, quand elle est utilisée, permet à GKS-3D de se comporter de manière identique à GKS.

La transformation de prise de vue est définie en deux parties à l'aide de deux matrices 4x4 qui spécifient l'orientation du repère de visualisation pour la première, la forme du volume de vision et le type de projection (parallèle ou perspective) pour la deuxième.

GKS-3D propose des primitives utilitaires pour la définition de ces matrices, basées sur le modèle de "caméra synthétique" analogue à CORE-SYSTEM. Néammoins, l'application peut construire et utiliser directement ses propres matrices.

La projection s'effectue du repère de la prise de vue vers le repère de projection normalisé dont les bornes sont comprises entre 0 et 1 pour les axes des x et des y, et 0 et -1 pour l'axe des z.

Le coupage par rapport au volume de vision est optionnel, et peut être activé ou désactivé par le programme d'application.

La dernière étape est finalement la transformation de poste de travail qui permet le passage de l'espace de projection normé vers l'espace écran du dispositif d'affichage. Sa spécification est analogue à celle de la transformation de visualisation mais est dépendante des postes de travail.

Le choix adopté pour GKS-3D de reporter l'opération de prise de vue 3D au niveau de chaque poste de travail après le stockage des segments, présente une nette amélioration par rapport au CORE-SYSTEM. Elle permet au niveau de l'implémentation de GKS-3D de tirer profit des éventuelles possibilités locales de stations graphiques haut de gamme en confiant directement au matériel tout ou partie de la prise de vue 3D [Gros 86].

Dans le même ordre d'idée, mais de façon beaucoup moins nette, GKS-3D prévoit l'accès optionnel aux possibilités d'élimination des lignes ou des parties cachées locales à un poste de travail. A cet effet un nouvel attribut est associé aux primitives de sortie HLHSR-IDENTIFIER ("Hidden Lines Hidden Surfaces Removal"), entier dont l'interprétation dépend des postes de travail. Au niveau de ceux-ci, un indicateur (HLHSR-MODE) peremt d'initialiser à l'aide d'un entier, un mode d'élimination des parties cachées parmi les différrentes techniques éventuellement disponibles. Le mode par défaut (HLHSR-MODE = 0) correspond à une visualisation sans aucune élimination des parties ou lignes cachées.

5 CONCLUSION

La volonté de développer des systèmes graphiques généraux a été une étape importante dans la compréhension et la diffusion de l'infographie. Elle a ainsi permis de dégager une méthodologie dans la conception des systèmes graphiques et d'unifier un certain nombre de notions de base. D'autre part l'indépendance de tels systèmes par rapport à leur environnement à rendu accessible le graphique à de nombreux nouveaux utilisateurs (que ce soit à travers l'utilisation d'applications que le développement de leur propres applications).

De cette volonté, a naturelement découlé le besoin d'adopter à un niveau internationnal un "standard" afin de tirer pleinement profit des avantages procurés par les systèmes graphiques

généraux. Ceci, a débouché sur la formidable "empoignade" qui a opposé les partisans de GKS à ceux du CORE-SYSTEM....[Mead 84][Sonde 84] etc.....

Cependant la comparaison entre ces deux systèmes n'est pas une tâche aisée. En effet, leur conception est basée sur des intentions initiales très différentes. Alors que le CORE-SYSTEM était destiné à supporter des systèmes graphiques aux hautes performances, interactif avec des possibilités de modélisation 3D, GKS adressait plus modestement la visualisation 2D, en insistant plus particulièrement sur la possibilité de sorties multiples (notion de poste de travail). Et si GKS, a ensuite evolué pour se rapprocher du CORE-SYTEM, tous deux ne se recouvrent pas entièrement. Par exemple, si une extension 3D a été definie pour GKS, elle est néanmoins limitée par la nécessité d'assurer une compatiblité avec la définition déjà existente de GKS. Le dessin tridimensionnel est possible avec GKS, mais le CORE-SYTEM s'avère par certains cotés mieux adapté, car il a intégré celui-ci dès sa conception.

Aussi, à l'instar de J. Warner [Warn 82] nous pensons que la notion de standard ne doit pas être vue comme un but absolu que l'on se fixe et sur lequel on se fige lorsqu'il est atteint, mais plutôt comme la traduction d'un certain degré de l'évolution de l'infographie. A ce titre, il est significatif que GKS tout comme le CORE-SYSTEM ont un certain nombre de fonctionnalités basées sur les posiblités des matériels des années soixante-dix, et qui ne correspondent plus vraimment à celles des nouvelles générations de matériel graphique. Ainsi la notion de segment, a-t-elle en grande partie été dictée par la prise en compte efficace des terminaux à tube mémoire, technologie dominante à l'époque.

Mais lors de la conception d'un système graphique général, anticiper et prendre en compte l'évolution future des matériels, comme le suggère B. Bruns de Megatek [Brun 82] nous parait quand à nous extrémement ardu, surtout pour la définition de ce qui doit devenir un standard". Nous pensons plutôt que la notion de standard n'est réellement envisageable qu'en la batissant à partir du bas, c'est à dire en commençant par définir une interface commune pour les matériels. C'est ce à quoi s'interressent les travaux concernant l'interface terminal virtuel (VDI-CGI), mais là encore ils restent le reflet d'un certain degré d'évolution de la technologie.

Cette constatation, ne remet pas en cause l'utilité de systèmes graphiques généraux, mais elle doit marquer les limites de ceux-ci, en particulier de ceux qui prétendent au titre de "standard".

Dans la définition d'interfaces graphiques de haut niveau pour les programmes d'application, GKS et le CORE-SYSTEM ont été une étape primordiale. Ils ont permis de dégager un certain nombre de concepts, désormais courramment reconnus; par exemple les notions de primitives de sortie, d'attributs des primitives de sortie.... Cependant, ils présentent à notre avis des opérateurs de niveau encore trop bas pour la visualisation et la description. La gestion de ces processus reste en partie à la charge des programmes d'application. GRI-GRI, à montré, lui, une voie interressante dans ce domaine en séparant clairement la construction des objets graphiques de leur visualisation. Au contraire, dans GKS et le CORE-SYSTEM, la notion de segment défini la structuration de l'image, et est donc étroitement liée à la visualisation.

Ceci nous permet de conclure, en remarquant que la prise en charge de manière cohérente par le système graphique des processus de base, attribution, consultation, description et visualisation passe par la nécessité d'offrir aux applications la possibilité de **définir** et **manipuler** les objets graphiques qu'elle souhaite visualiser. Ce second point a en grande partie été occulté dans les systèmes que nous avons présentés ici du fait de la séparation historique entre les tâches de modélisation et de visualisation. Sa prise en compte passe par une possibilité de structuration des objets graphiques, fonctionnalité absente de GKS, du CORE-SYSTEM ou de GRI-GRI qui ne présentent que deux niveaux de structuration. C'est cet aspect que nous étudions dans le chapitre qui suit.

Chapître III : Modélisation et structuration hiérarchique des données graphiques

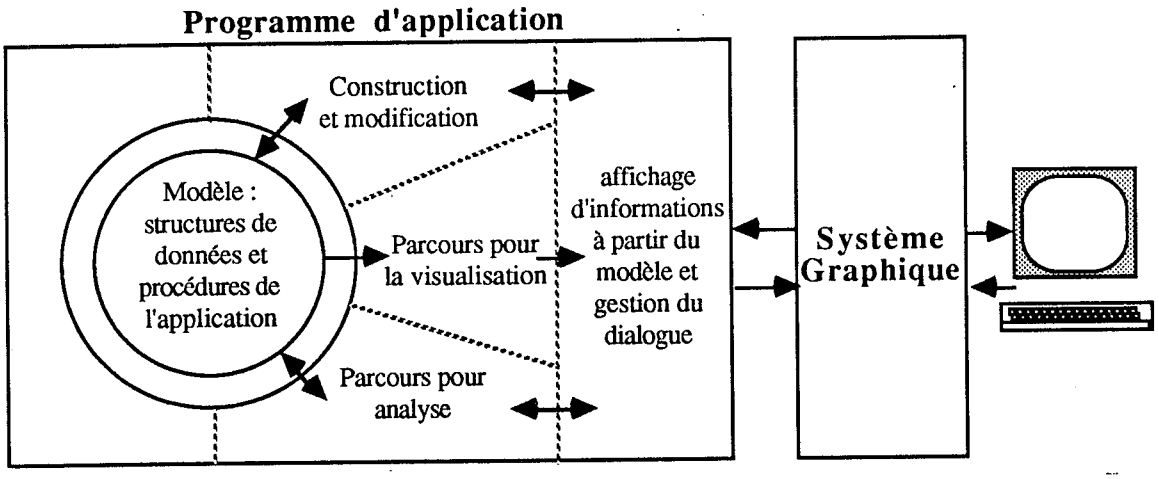
1 INTRODUCTION

Les systèmes graphiques que nous avons étudiés au chapitre précédent sont tous basés sur la séparation entre les activités de modélisation à la charge du programme d'application et les activités de visualisation assurées par le système graphique [GuTu 76].

Le rôle du système graphique se limite essentiellement à permettre aux applications de décrire des représentations visuelles (**images**), du modèle qu'elles manipulent. Pour cela, le programme d'application doit **extraire** du modèle les informations graphiques qu'il contient et les **mettre sous une forme** acceptable par le système graphique. Ceci est bien souvent une lourde charge pour le programme d'application qui doit avoir une très bonne connaissance du (des) processus de visualisation du système graphique. Par ailleurs, il y a une certaine redondance entre les données graphiques contenues dans le modèle et les informations conservées par le système graphique.

Un autre inconvenient dû à cette séparation, provient du fait que le système graphique travaille sur des **images** du modèle. Il permet, certes d'effectuer certaines manipulations interactives sur celles-ci, mais leurs répercussions éventuelles à l'intérieur du modèle est à nouveau entièrement sous la responsabilité de l'application. De manière symétrique, les transformations apportées au modèle par l'application ne peuvent pas toujours être transmises directement vers le système graphique: bien souvent il est nécessaire de reconvertir à nouveau le modèle afin d'en définir une nouvelle image.

La figure III.1 ci dessous, résume ces différents points. Tirée de [FoDa 82], elle présente les différentes activités (donc programmes) à l'intérieur d'une application pour l'exploitation d'un modèle et leurs rapports vis à vis du système graphique.



- figure III.1 : l'activité de modélisation dans une application graphique (d'après [FoDa 82]) -

Comme on peut le constater, ces restrictions limitent sérieusement les avantages offerts par le système de visualisation, et en particulier l'interactivité. Aussi, si la séparation modélisation / visualisation, a pu être justifiée du point de vue historique, elle n'a néanmoins jamais cessé d'être remise en cause de par les besoins de plus en plus exigeant d'applications manipulant des modèles de plus en plus complexes et de par l'évolution constante du materiel permettant d'effectuer des traitements de plus en plus importants.

Aussi, dans ce chapitre, nous présentons différentes solutions apportées en graphique à la modélisation d'objets complexes et la manière dont elles peuvent être prises en compte par les systèmes graphiques. A ce titre, nous étudierons de manière détaillée le système PHIGS, qui, comme les systèmes vus au chapitres précédents, offre aux applications une interface indépendante du materiel. Mais à la difference de ceux-ci, il assure une activité de modélisation qui permet d'envisager une plus grande interactivité, le système ne travaillant plus sur des images, mais sur des modèles d'objets.

2 STRUCTURATION HIERARCHIQUE DES DONNEES GRAPHIQUES

La modélisation d'objets complexes ("objets" au sens large) nécessite une importante structuration des données qui les définissent afin d'en permettre une exploitation efficace. Ainsi, emploie-t'on très souvent pour la représentation de ces modèles complexes des structures **hiérarchiques**. En effet, des hiérarchies multi-niveaux permettent d'exprimer simplement des notions de dépendances fonctionnelles ou toutes sortes de relations entre objets à inclure dans un modèle. De plus, elle offrent une efficacité inégalée en simplicité et rapidité pour tout ce qui concerne les modifications à apporter au modèle.

Aussi, est-il naturel que l'on retrouve de telles structures hiérarchiques pour la description d'objets destinés à la visualisation interactive. Dans ce qui suit, nous allons donc étudier l'apport qu'une telle structuration des données peut avoir dans le contexte graphique.

Deux approches prédominantes sont à distinguer [CCVa 85] :

- l'emploi de la structuration hiérarchique pour définir une **décomposition d'objets en sous objets**, bien adaptée à la représentation de relations géométriques,

- l'emploi de la structuration hiérarchique pour définir une **décomposition de l'espace**, qui est plus particulièrement destinée à l'amélioration des algorithmes de visualisation.

2.1 Décomposition hiérarchique d'objets en sous-objets

La décomposition hiérarchique d'objets en sous objets est une approche très naturelle qui correspond à un processus de construction ascendant ("bottom-up") des objets. Des composants ou objets élémentaires sont utilisés comme "blocs de construction" pour créer des entités de plus haut niveau, elles mêmes utilisées comme blocs de construction des niveaux supérieurs. Cette représentation permet la construction d'objets complexes de façon modulaire [MTLa 82].

2.1.1 Les transformations Géométriques dans une décomposition hiérarchique - Transformations de modélisation

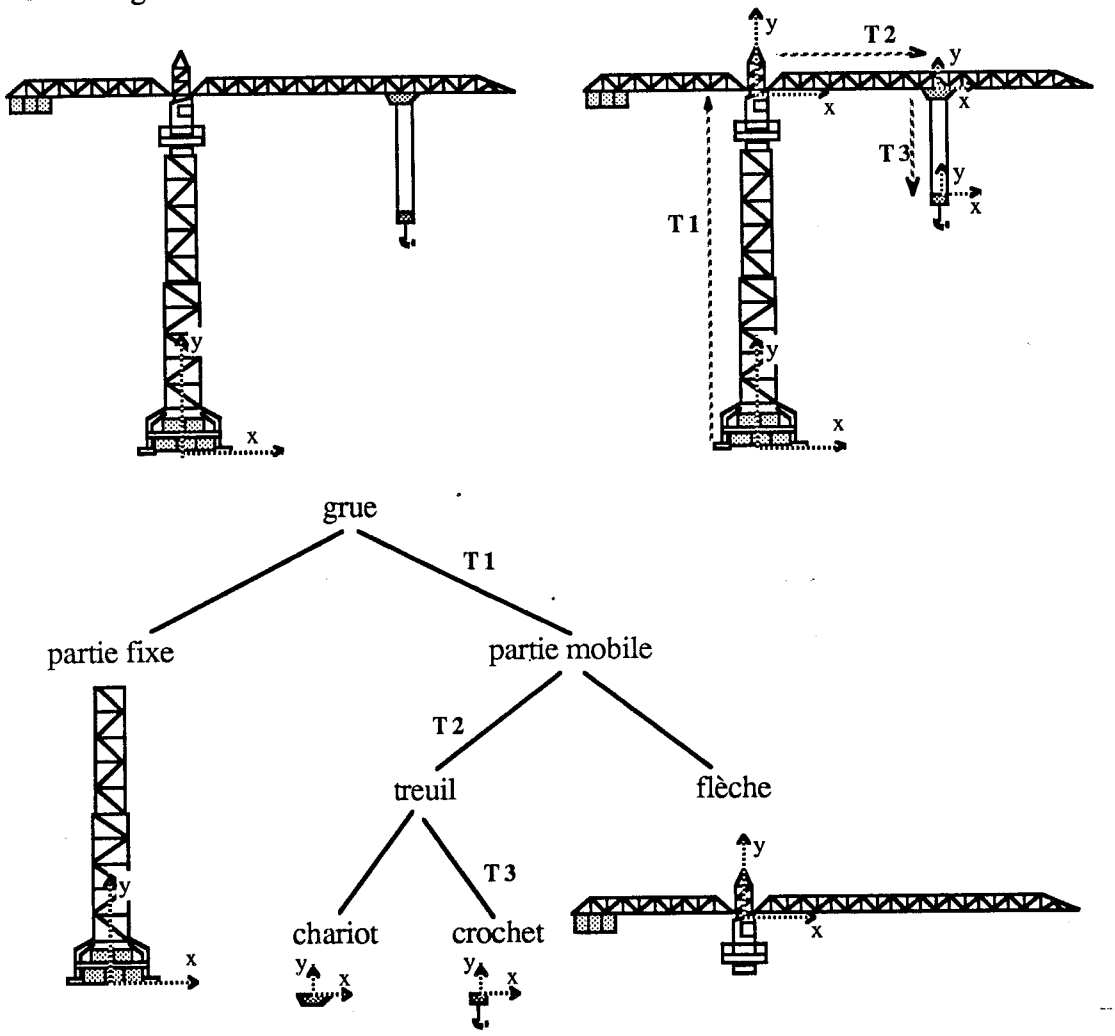
Les transformations géométriques jouent un rôle très important dans de telles représentations hiérarchiques : elles sont à la base de la construction du modèle. Elles servent à positionner les objets dans l'espace et à situer ainsi les différents composants d'un objet les uns par rapport aux autres.

Dans ce modèle, à chaque élément de la structure hiérarchique (objet élémentaire ou non) est associé un repère de **coordonnées locales** ("Master coordinates"). Les constituants d'un objet sont "assemblés" à l'aide de transformations géométriques, désignées sous le terme de **transformations de modélisation** ("Modelling transformation") qui expriment la position des sous-objets dans le repère local de l'objet de niveau supérieur. Le système de coordonnées associé à la racine de la structure hiérarchique correspond au repère de l'espace utilisateur ("World Coordinate Space").

Ainsi, les objets élémentaires (feuilles de l'arborescence) peuvent être décrits dans leur propre repère local ("Master Coordinate System"), qui peut être choisi de manière adéquate afin de rendre la description la plus aisée possible. (Par exemple, la description d'un objet symétrique est facilitée si l'on se situe dans un repère dont l'origine est précisement le centre de symétrie).

Les différentes transformations géométriques de modélisation qui placent les objets dans le repère de l'objet de niveau supérieur sont **composées** lors de la visualisation. Ainsi, un objet élémentaire est affiché en lui appliquant toutes les transformations géométriques définies dans le chemin qui relie le noeud le représentant au noeud racine de la structure hiérarchique.

La figure III.2 présente **une décomposition possible** pour une grue de travaux publics en ses différents constituants et indique les différentes transformations de modélisation qui permettent leur assemblage. La grue se décompose en une partie fixe qui correspond à son socle et une partie mobile qui regroupe la flèche et le treuil. Le treuil est lui même constitué de deux sous-objets : le chariot qui permet son déplacement le long de la flèche et le crochet qui sert à fixer des charges.



- figure III.2 : décomposition hiérarchique d'un objet et transformations de modélisation -

Les différentes transformations de modélisation qui permettent de situer les composants les uns par rapport aux autres sont :

- une translation T_3 qui positionne le crochet relativement au treuil (dont le repère coincide avec celui du chariot),

- une translation T_2 qui positionne le treuil relativement à la partie mobile (dont le repère coincide avec celui de la flèche),

- une translation T_1 qui positionne la partie mobile dans le repère de la grue (qui coincide avec celui de la partie fixe),

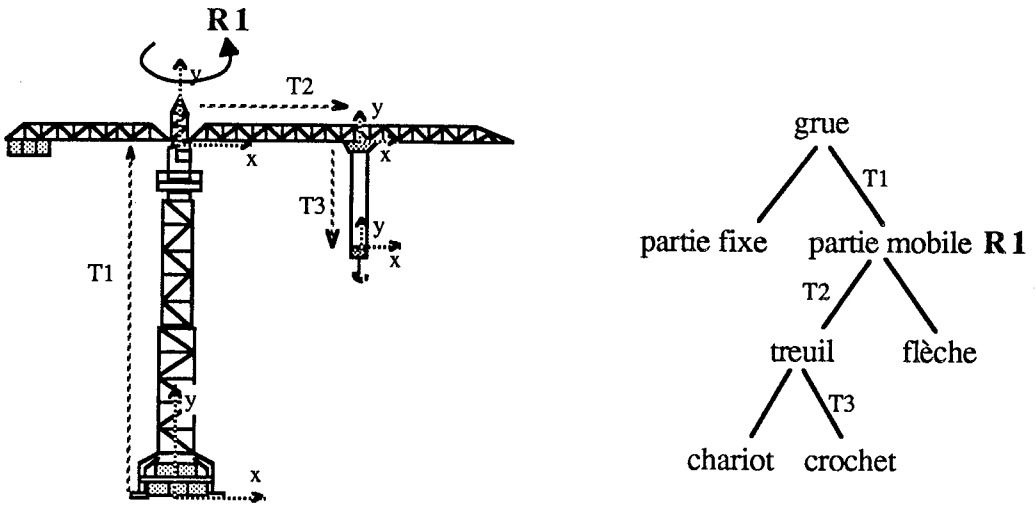
Lors de la visualisation, la transformation géométrique à appliquer aux coordonnées définissant le crochet dans son repère local ("Master Coordinates") est la composition des trois translations T_1, T_2 et T_3. De manière classique, les transformations géométriques sont représentées à l'aide de matrices en coordonnées homègenes et leur composition correspond au produit de ces matrices [RoAd 76]. Ainsi, par exemple, on obtient les coordonnées utilisateurs (x_u,y_u,z_u) du crochet en appliquant successivement aux coordonnées locales (x_l,y_l,z_l) les trois translations T_1, T_2 et T_3 c'est à dire la matrice produit des matrices correspondant respectivement à T_1, T_2 et T_3 :

$$(x_l,y_l,z_l,1) * T_3 * T_2 * T_1 = (x_l,y_l,z_l,1) * T = (x_u,y_u,z_u)$$

(cette transformation composée est en général désignée sous le terme de transformation globale)

La structuration hiérarchique exprime naturellement des relations fonctionnelles entre différents éléments. En particulier, la composition des transformations de modélisation entre les différents niveaux de structure permet d'assurer simplement l'unité de mouvement d'un ensemble de composants. La modification de la géometrie d'un objet s'applique uniformement à tous les sous-objets qu'il référence. Ainsi le déplacement du treuil s'effectue en remplaçant la transformation de modélisation T_2 par une nouvelle translation T'_2. (Ce déplacement se répercute de façon équivalente au chariot et au crochet qui constituent le treuil et dont la transformation globale devient respectivement $T'_2 * T_1$ et $T_3 * T'_2 * T_1$).

Les transformations géométriques, en plus du positionnement d'un objet dans le repère de niveau supérieur, peuvent également servir à définir le mouvement d'un objet dans son propre repère local. Par exemple, si nous reprenons notre exemple de la figure III.1, faire pivoter la grue autour de son axe vertical revient à appliquer à la partie mobile une rotation relativement à l'axe Oy de son repère (fig. III.3).



- figure III.3 : transformation géométrique locale -

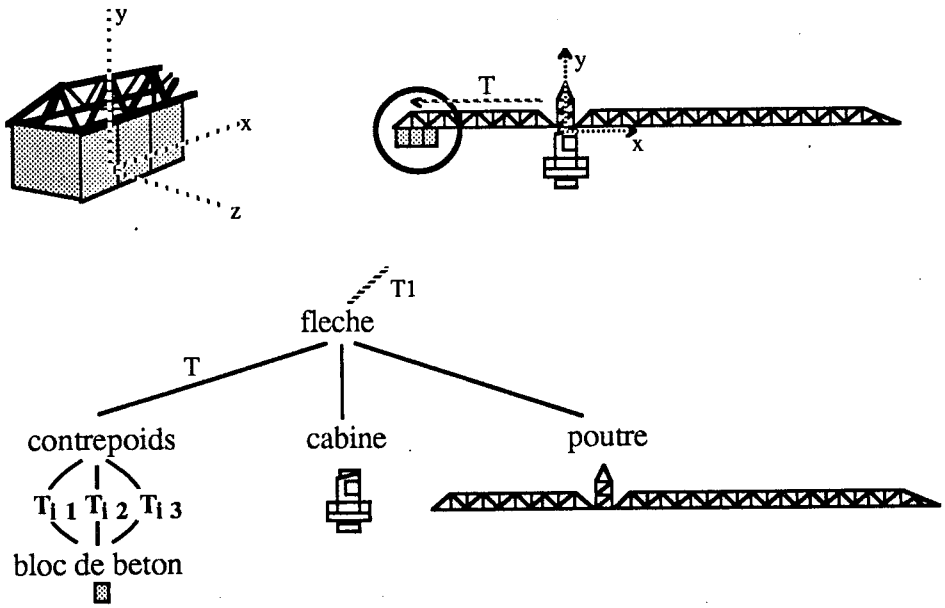
La prise en compte d'une transformation locale est effectuée en la postmultipliant avec la transformation de modélisation de l'objet. Ainsi, si on associe une rotation locale, R_1 , à la partie mobile de la grue les éléments qui la constituent ont alors pour géométrie globale :

- $R_1 * T_1$ pour la fèche,
- $T_2 * R_1 * T_1$ pour le chariot du treuil,
- $T_3 * T_2 * R_1 * T_1$ pour le crochet du treuil.

2.1.2 Partage de sous-objets - Instances

La décomposition hiérarchique d'objets en sous-objets permet également une **économie d'espace mémoire** considérable lorsque des éléments identiques apparaissent plusieurs fois dans une même scène. Dans ce cas les objets considérés n'ont besoin d'être définis qu'une seule fois, et peuvent être ensuite **instanciés** par l'application d'une transformation linéaire dans la hiérarchie. Il suffit ainsi de conserver uniquement les références vers les objets utilisés plusieurs fois, plutôt que de redéfinir ceux-ci à chaque fois. Les différentes occurences de l'objet sont particularisées chacune par une transformation de modélisation propre (parfois désignée sous le terme de transformation d'instance). Dans ce cas, la hiérarchie n'est pas véritablement un arbre mais un graphe sans circuit dont les noeuds sont les sous-objets et les arêtes les transformations de modélisation ou transformations d'instance.

Si l'on reprend notre exemple précédent, et que l'on décompose la flèche en sous-objets (fig. III.4), on remarque que le contrepoids est défini à partir de trois blocs de béton. Ces trois blocs sont en tous points identiques, seule leur géométrie diffère. Il est donc possible de ne définir qu'une fois l'objet *bloc de béton* et de le partager ensuite à l'aide de transformations d'instance, trois translations T_{i1}, T_{i2} et T_{i3} qui positionnent les trois occurences du bloc de béton dans le repère du contrepoids.



- figure III.4 : partage de sous-objets, transformations d'instance -

Les transformations d'instance se composent dans la structure hiérarchique comme les transformations de modélisation. Ainsi les transformations globales associées aux trois occurences du bloc de béton sont respectivement :

- $T_{i2} * T * T_1$
- $T_{i2} * T * T_1$
- $T_{i3} * T * T_1$

2.1.3 Autres apports de la structure hiérarchique en graphique

Les transformations géométriques appliquées dans une structure hiérarchique permettent comme nous l'avons vu d'exprimer très aisément une unité de mouvement entre plusieurs éléments. Cette unité fonctionnelle peut être étendue à d'autres attributs graphiques dont il est également possible d'envisager la composition dans la structure. Par exemple, on peut imaginer

que l'association d'une couleur à un niveau de la structure se "propage" à tous les niveaux de structure hiérarchiquement dépendants.

De façon symétrique à la composition ("propagation") des attributs dans la hiérarchie, avec une telle représentation des objets, la modification des entités peut s'effectuer sur la base des composants de structure. Si l'on modifie un composant, la répercussion (dans le modèle) sur tous les éléments de plus haut niveau qui font appel à lui est immédiate (et implicite).

D'autre part, la structure hiérarchique peut être exploitée pour les alogrithmes de synthèse d'images en tirant profit de la cohérence spatiale exprimée par cette représentation [Clar 76]. Par exemple, il est possible d'associer à chaque niveau d'objet dans la hiérarchie une boite englobante afin d'accélérer les opérations de coupage ("clipping") lors d'une visualisation. Ainsi lors de l'affichage d'une structure hiérarchique, le volume englobant est comparé au volume de coupage. Si il entièrement à l'intérieur ou entièrement à l'extérieur du volume de coupage, les sous-objets qu'il regroupe peuvent être trivialement acceptés ou rejétés sans nécessiter d'opération de coupage ultérieur. Ce simple test permet d'éviter le parcours de sous structures non visibles.

De façon analogue, dans le cas de calculs d'élimination des parties cachées, la vitesse du processus de tri peut être améliorée en ordonnant au préalable les volumes englobants et en effectuant ensuite les tests de visibilité uniquement sur les objets succeptibles d'être visibles. Comme, pour le coupage, le contenu d'une boite englobante peut ne pas être pris en compte s' il est complétement occulté par d'autres objets lors de la prise de vue...

2.1.4 Listes de visualisation Structurées

Dans une décomposition hiérarchique d'objets en sous-objets les feuilles correspondent naturellement à des objets élémentaires. De manière classique, ces objets élémentaires sont définis à partir de primitives de base analogues aux primitives de sortie des systèmes graphiques généraux (segment, cercles, faces polygônales planes, patches de surfaces....). En général elles définissent les limites de l'objets (on parle parfois de B-Rep ou "Boundary-Representation").

La visualisation d'une telle hiérarchie, s'effectue en parcourant la structure arborescente et en affichant chaque primitive rencontrée avec les attributs évalués dynamiquement.

Bien souvent, cette opération ne peut être effectuée par le matériel graphique. Il est alors nécessaire de convertir le modèle hiérarchique en un format acceptable par le processeur d'entretien et donc de découpler les opérations de mise à jour de celles de raffraichissement. Cette conversion peut être soit effectuée directement à partir du modèle de l'application, soit à partir d'une structure extraite de ce modèle et désignée parfois sous le terme de liste de visualisation structurées ("Structured Display File"). La liste de visualisation structurée contient uniquement l'information grahiques qui décrit le modèle.

Mais, de plus en plus, des possibilités de structuration hiérarchique des données sont intégrées dans le matériel graphique lui-même [ClDa 83], [Will 85]. Ainsi, on s'approche de la situation idéale qui consisterait à ce que la liste de visualisation strucutrée soit partagée à la fois par le programme d'application qui la construit et modifie et par le processeur d'affichage qui la lit et la parcours afin de l'afficher. Une telle possibilité permet de laisser à la charge du matériel la transformation du modèle pour sa visualisation et autorise des mises à jours instantanées de l'image. Elle évite également une duplication des données graphiques. Mais cette traduction "au vol" nécessite du matériel hautement performant et reste limitée de par la nécessité de rafraîchir périodiquement l'image.

2.1.5 Arbres de construction ou CSG (Constructive Solid Geometry)

Le développement de système de C.A.O. pour des objets tridimensionnels, à donné lieu à la définition de modèles **volumiques**, destinés à rendre les propriétés exactes de solides [Requ 80], [Requ 82]. Le modèle par arbre de construction est l'un des plus utilisés. Il utilise une grande partie des propriétés des structures hiérarchiques que nous avons exposées précédemment, c'est pourquoi nous le présentons ici.

2.1.5.1 Principe

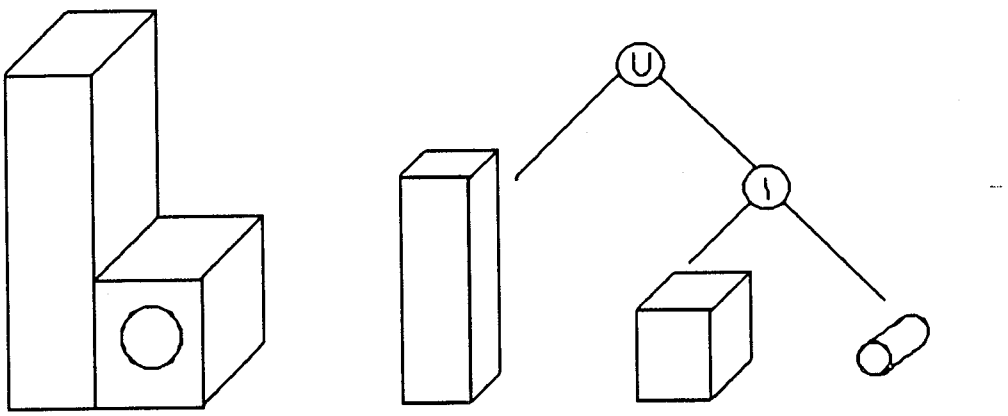
Le modèle par arbre CSG permet de décrire un volume à partir de volumes élémentaires combinés entre eux à l'aide d'opérateurs ensemblistes. Les volumes de base sont des volumes unitaires simples tels que :

- la sphère (centrée en (0,0,0) et de rayon 1)
- le cube (de cotés de longueur 1 et avec (0,0,0) comme coin inférieur gauche)
- le cylindre (centré en (0,0,0), de rayon et hauteur 1)
- le cône, le tore .....

Les opérateurs de composition (parfois appelés opérateurs booléens) sont :

- l'union (notée ∪ ou +),
- l'intersection (notée ∩ ou &),
- la différence (notée ^ ou -).

Ce principe de description des solides est récursif : de nouvelles opérations de composition pouvent être appliquées à des objets précédemment créés. Un objet complexe est ainsi naturellement représenté par un arbre dont les noeuds sont des opérateurs de composition et les feuilles des objets primitifs. Aux différents peuvent éventuellement être associées des transformations géométriques (homothéties pour déformer les objets unitaires de base, translations et rotations pour le positionnement relatif d'objets). Un tel arbre représente l'historique de la construction de l'objet à partir des volumes de base, la figure III.5 ci-dessous en donne un exemple.



- figure III.5 : Arbre de construction ou arbre CSG -

remarque : dans une forme plus générale de la représentation CSG, les sous-arbres peuvent être partagés, la structure hiérarchique n'est alors plus un arbre, mais un graphe sans circuits.

2.1.5.2 Avantages - Inconvénients

Ce modèle offre une grande facilité de description, la contruction d'objets complexes

consistant à batir simplement un arbre à partir de volumes élémentaires eux-même aisés à définir. (Cependant, il est clair que le pouvoir de description de ce modèle est directement lié au nombre et à la qualité des primitives admises. Il est bien adapté pour la définition de pièces mécaniques ou d'objets manufacturés mais se prète beaucoup moins à la définition d'objets non "réguliers" tels des plantes, paysages, visages...).

Directement liée à ce point, on peut également remarquer la facilité de stockage pour une telle représentation

En ce qui concerne la visualisation d'objets modélisés de cette manière, l'arbre de construction ne décrivant pas explicitement les contours de l'objet représenté, il est nécessaire soit d'utiliser un algorithme acceptant directement en entrée une telle représentation, soit de convertir l'objet vers une autre représentation pour laquelle des algorithmes sont disponibles [Kois 85].

Les algorithmes travaillant directement sur l'arbre de construction peuvent être divisés en deux familles :

- la première et la plus répendue, utilise les techniques de lancer de rayon [Roth 82]. Parfaitement adaptée à cette représentation, elles permettent le rendu avec un très grand réalisme des phénomènes d'éclairage et de transparence mais sont excessivement couteuses en temps de calcul (malgré differentes techniques d'accélération [Exco 88] et d'exploitation du parallèlisme [PrBo 87]).

- la deuxième utilise des méthodes de balayage ligne par ligne et s'applique à des arbres CSG dont les volumes ont une frontière définie par des polygônes [Athe 83].

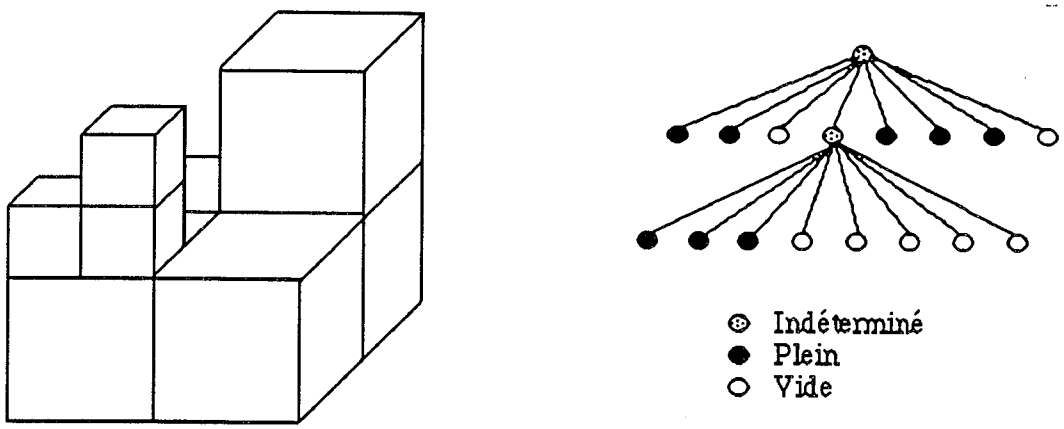
2.2 Décomposition hiérarchique de l'espace

2.2.1 Enumération Spatiale - Arbres Octaux

2.2.1.1 Principe

Cette représentation consiste à subdiviser (discrétiser) l'espace en cellules élémentaires ("volume elements" ou "voxels" par analogie aux "pixels" qui découpent en carrés élémentaires l'espace 2D de l'écran) et à fournir pour un objet la liste des cellules occupées.

Pour une telle représentation, la solution triviale qui consisterait en l'utilisation d'un tableau tridimensionnel [Ohas 85] dont chaque élément équivaudrait une à cellule élémentaire, n'est pas réellement envisageable dans l'état actuel de la technique pour des problèmes de capacité mémoire évidents.



- figure III.6 : Représentation par arbre octal (Octree) -

Aussi, afin d'aboutir à une représentation beaucoup plus compacte, on utilise une décomposition hiérarchique de l'espace en une succession de cubes de taille décroissante. Cette modélisation est la généralisation des "quadtrees" à l'espace tridimensionnel, et peut être représentée à l'aide d'un arbre octal ou "octree" [Meag 82]. Un objet (volume) est un arbre dont la racine est sa boite englobante. A chaque noeud de l'arbre est associé un type (plein, vide ou indéterminé) indiquant que la boite correspondant au noeud est située entiérement à l'intérieur du volume, entièrement à l'extérieur, ou en partie à l'intérieur et à l'extérieur. La boite d'un noeud de type indéterminé est divisée reccursivement en huit sous-boites de taille égale (qui correspondent aux fils de ce noeud). La décomposition s'achève lorsque l'on a atteint le niveau maximum de subdivision ou une cellule homogène (vide ou pleine) (fig. III.6).

2.2.1.2 Avantages - Inconvénients

La représentation d'objets sous forme d'Octrees, rend aisés le calcul (approximatif) de propriétés relatives à des solides (centre de gravité, volume, masse) et la réalisation d'opérateurs de composition ($\cup$, $\cap$, $\setminus$) qui se résument à de simples parcours d'arbres.

De façon analogue, la visualisation avec élimination des parties cachées d'un objet représenté par un "octree", est simple et rapide et consiste au parcours de l'arbre avec l'affichage des noeuds terminaux dans le bon ordre (d'arrière en avant) selon la position de l'observateur (8 cas sont possibles). Toutefois cette représentation discrète perd la continuité et la normale des surfaces, entrainant ainsi certaines difficultés dans le rendu du réalisme.

Par ailleurs, si les arbres octaux permettent de décrire et d'approximer avec la précision souhaitée toutes les formes imaginables (convexes, concaves, avec éventuellement des trous intérieurs), l'acquisition d'une telle représentation soulève de nombreux problèmes. Certes, des saisies automatiques à partir d'objets existants sont possibles comme par exemple les saisies tomographiques en médecine. Par contre, pour construire un objet imaginé de toutes pièces, la solution la plus simple consite à décrire l'objet à l'aide d'une autre représentation plus adéquate (par exemple par un arbre de construction) qui est ensuite convertie en arbre octal (tâche relativement aisée). Mais dans ce cas, on perd la puissance des "octrees" permettant de représenter des formes quelconques.

Une autre difficulté liée à l'utilisation des "octrees", concerne la prise en compte de transformations géométriques appliquées aux objets qui peuvent nécessiter une reconstruction de l'arbre.

2.2.2 Les Arbres BSP (Binary Space Partition)

2.2.2.1 Principe

Une autre décomposition hiérarchique de l'espace a été proposée par H.FUCHS et al. [FKNa 80], [FaGr 83] qui, elle aussi permet de visualiser rapidement des scènes tridimensionnelles en facilitant l'opération d'élimination des parties cachées. Cependant, elle est beaucoup moins générale que les arbres octaux, et ne s'applique qu'à des objets décrits à l'aide de faces planes.

L'approche proposée par FUCHS découle de travaux menés par SCHUMACKER [Schu 69], et consiste à tirer profit du fait que dans de nombreuses applications les images sont souvent générées à partir d'un même environnement avec pour seuls changements, la position et l'orientation du point de vue. Il est possible d'exploiter les propriétés géométriques d'un tel environnement statique, et au prix d'un prétraitement sur la base de données graphique, d'accélérer par la suite la détermination des polygones visibles en chaque pixel pour toutes les images possibles.

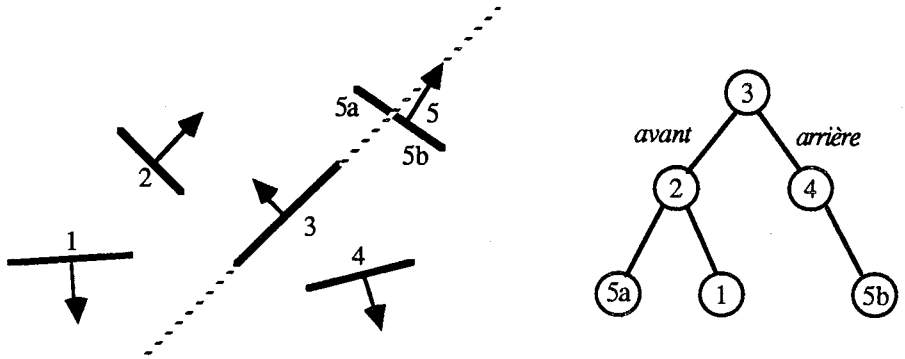
Ainsi, le plan de chaque polygône, divise l'espace en deux demi-espaces tels que si le point de vue est dans l'un de ces demi-espaces, aucun polygône situé dans l'autre demi-espace ne peut obstruer le polygône considéré, pas plus que tout autre polygône situé du même côté que le point de vue.

La solution présentée par FUCHS consiste à exploiter cette propriété en transformant la liste des polygones qui décrivent la scène en un arbre binaire qui défini alors une priorité de visibilité entre eux. Cet arbre, désigné sous le terme d'arbre BSP ("Binary Space Partition") est construit de telle sorte que:

- chaque noeud correspond à un polygône,

- les deux sous arbres d'un noeud sont tels que l'ensemble des polygônes qu'ils contiennent sont associés respectivement à chacun des deux demi-espaces définis par le polygône.

Si lors de la construction de l'arbre (fig. III.7), il existe des polygones qui intersectent le plan d'un noeud, ceux-ci sont coupés relativement à celui-ci [Hegr 85].



- figure III.7 : Exemple de représentation sous forme d'arbre BSP -

Suite à ce prétraitement, la visualisation s'effectue aisément à l'aide d'un simple parcours d'arbre qui en fonction des paramètres de prise de vue permet d'ordonner les polygones. Pour afficher les polygones du plus éloigné au plus proche, il suffit pour chaque noeud d'afficher recursivement tous les polygones situés dans le demi-espace opposé à celui dans lequel se trouve le point de vue, d'afficher ensuite le polygone associé au noeud, puis finalement, d'afficher recursivement les polygones situés du même coté que le point de vue.

2.2.2.2 Avantages / Inconvénients

Cette structuration hierarchique de l'environnement permet de résoudre facilement le problème d'élimination des parties cachées et d'engendrer rapidement les vues successives d'une même scène.

L'algorithme de visualisation très simple, peut facilement être implanté sur un processeur graphique programmable (FUCHS à réalisé une implantation sur un processeur en tranches AMD 2900) autorisant ainsi un usage interactif.

Cependant la représentation sous forme d'arbre BSP présente quelques limites. Elle ne permet le traitement que de scènes statiques décrites à l'aide de faces polygonales. De plus, si la construction de l'arbre BSP ne soulève aucune difficulté, elle peut suite à de multiples coupages de polygones aboutir à un arbre de taille significativement plus grande que la liste originale. Ainsi, en particulier si la visualisation est confiée à un processeur graphique, cela peut soulever des problèmes de mémoire locale pour conserver l'arbre BSP.

2.3 Conclusion

Dans les paragraphes qui précèdent nous avons présenté différents emplois de la structuration hiérarchique des données dans le cadre du graphique afin de bien montrer la diversité des utilisations qui peuvent en être faites.

Toutefois, il est clair que les décompositions hiérarchiques de l'espace, si elles sont bien adaptées à certains algorithmes de visualisation ne permettent pas de décrire de manière simple et rapide des objets complexes. Pour cela la décomposition hiérarchique d'objets en sous objets est beaucoup mieux appropriée. Aussi, est-ce elle qui est le plus généralement utilisée comme interface entre un système de modélisation et des applications interactives. Cependant la représentation d'objets à partir de primitives de sortie de base ou à partir de volumes élémentaires ont chacune leurs avantages et inconvénients. La première offre une très grande souplesse et liberté mais ne garantie pas la validité des solides ainsi représentés. La seconde, au contraire, permet de définir des solides de manière exacte [Requ 82]. Néanmoins, elle se limite à des objets dont les propriétés géométriques découlent de celles des volumes de base. Si elle permet un rendu avec un très haut degré de réalisme, en particulier à l'aide d'algorithmes de lancer de rayon, elle n'autorise pas à l'heure actuelle une véritable interactivité sur les images ainsi obtenues.

D'autres méthodes de description de solides existent (par exemple par balayage d'un contour sur une trajectoire [Boul 82]). Nous ne les avons pas présentées ici, mais elles peuvent néanmoins être utilisées pour décrire les objets de base. Par ailleurs, de plus en plus, des modèles spécialisés existent pour décrire de manière réaliste des objets spécifiques (ex: nuages, plantes....). Il est certain que la situation idéale, serait de faire cohabiter tous ces différents modèles et de pouvoir passer éventuellement automatiquement d'une représentation à une autre. On pourrait ainsi utiliser les décompositions hiérarchiques de l'espace pour les opérations de visualisation et la géométrie constructive pour l'élaboration du modèle. Cependant ceci reste un problème largement ouvert...

3 LE SYSTEME PHIGS

3.1 Présentation générale

PHIGS (Programmer's Hierarchical Interactive Graphics System) est une proposition pour un standard graphique, développée sous l'égide de l'ANSI (American Nationnal Standard Institute) et reprise comme sujet de travail par l'ISO [ANSI 85],[ISO 86],[SBMo 86].

Bien que, comme nous l'avons vu précédemment, GKS semble s'imposer comme un standard internationnal pour le dessin 2D et qu'une extension 3D soit en cours d'adoption, un certain nombre de personnes impliquées dans le processus de standardisation de l'ANSI ont considéré que celui-ci ne pouvait pleinement tirer profit des possibiltés de matériels évolués et donner entière satisfaction à des utilisateurs ayant besoin d'une interface puissante pour écrire des applications hautement interactives. C'est à partir de cette constatation qu'à été mis en oeuvre le projet PHIGS [Wrig 84],[Heck 86].

PHIGS réutilise de nombreux concepts élaborés au cours du développement de GKS, en particulier la notion de poste de travail virtuel ("abstract workstation"), de dispostifs logiques d'entrée, les primitives de sortie et leurs attributs de bases. Mais à ceux-ci ont été ajoutés de nouveaux concepts destinés à fournir un système 3D fortement interactif, prenant en charge l'activité de **modélisation géométrique**.

Il s'agit essentiellement de la **structuration des données graphiques** qui permet de regrouper les informations de base dans ce qui est désigné sous le terme de structure, ces structures pouvant être organisées **hiérarchiquement**. Cette organisation hiérarchique des données graphiques permet aux programmeurs d'application de décrire facilement et efficacement au système graphique, le modèle des objets qu'ils manipulent. En effet, ces modèles étant très souvent multi-niveaux, leur conversion vers le système graphique peut se faire sans difficultés, car celui-ci permet de conserver la même structuration. (Rappelons que GKS ne propose qu'un seul niveau de structuration : les segments).

Le deuxième point clé, concerne la possibilité laissée à l'application de **modifier à tout moment** les informations graphiques qu'elle a définies auparavant. Cette facilité permet à l'application de transmettre simplement et facilement vers le système graphique les modifications apportées au modèle qu'elle manipule : il suffit de changer (editer) le contenu de la ou des structures correspondantes. Ces modifications dynamiques de la base de données graphiques, peuvent être ensuite immédiatement répercutés sur l'image affichée, permettant ainsi la conception d'applications réellement interactives. A titre de comparaison, dans GKS le contenu des segments ne peut être modifié, la modification des attributs regroupés dans un segment nécessite sa destruction et son entière redéfinition avec les nouveaux attributs.

Dans la suite de cet exposé, nous ne reviendront pas ou que très brièvement sur les points de PHIGS directement issus de GKS [Krze 86] et nous nous attacherons à mettre en évidence les originalités de ce système.

3.2 Concepts de base de PHIGS

3.2.1 Postes de travail virtuels et Base de données graphique

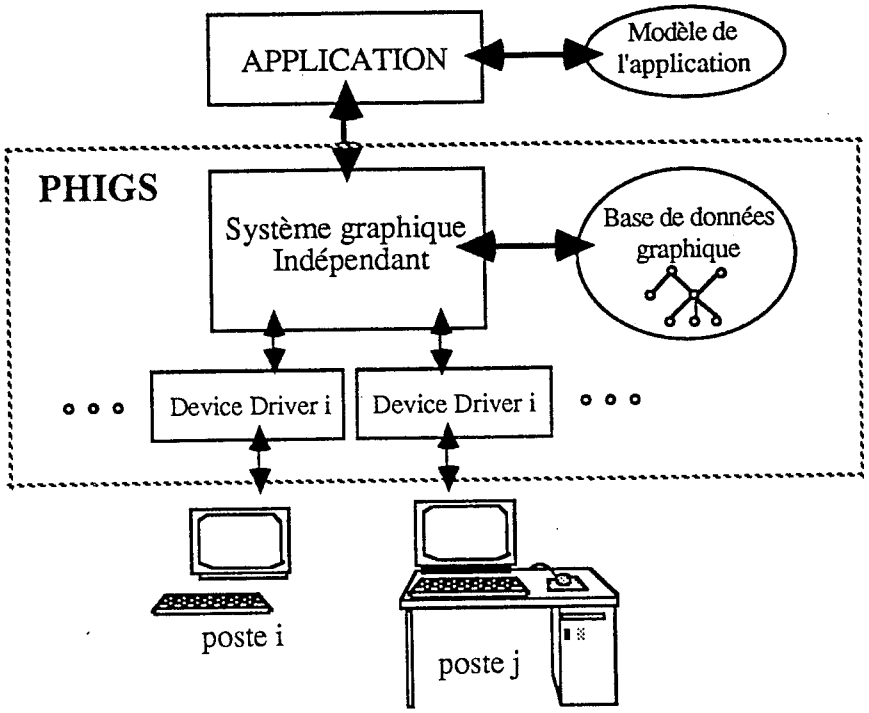
Comme GKS, PHIGS est basé sur le concept de poste de travail graphique virtuel ("abstract graphical workstation"), qui fourni une interface logique par laquelle le programme d'application contrôle les dispositifs physiques. Une implémentation de PHIGS peut proposer plusieurs postes de travail auxquels les applications peuvent accéder simultanément. Plusieurs types de postes de travail sont considérés : il s'agit de postes pouvant soit produire des images à partir des données graphiques, soit fournir des entrées graphiques, soit effectuer les deux à la fois. Ces postes de travail virtuels peuvent être implémentés par n'importe quelle combinaison de logiciel et de matériel.

Il est ainsi possible de contrôler de manière individuelle sur chacun des postes de travail un certain nombre de paramètres concernant la visualisation des objets graphiques (prise de vue; apparence des objets graphiques, efficacité de l'affichage) à l'aide de tables d'attributs dépendantes des postes. Cela permet, comme nous l'avons vupoyr GKS, de conserver l'indépendance par rapport au matériel tout en permettant de tirer profit des particularités spécifiques de tel ou tel dispositif physique.

Cependant, à la différence de GKS, où les données graphiques sont dès leur définition associées à un ou plusieurs postes de travail où elles sont stockées localement ("Workstation Dependant Segment Storage"), PHIGS sépare clairement la définition des données graphiques de leur visualisation. Ainsi, d'un point de vue conceptuel, lorsque l'application spécifie des données graphiques au système PHIGS, celles-ci sont introduites dans une base de données graphiques, ("Graphic Data Storage"), facilité unique, centralisée et indépendante de l'existence des possibilités ou du status de n'importe quel poste de travail particulier (fig. III.8).

Les informations introduites dans cette base de données sont organisées hiérarchiquement et peuvent être modifiées à tout moment par l'application. L'affichage d'un objet défini par une structure hiérarchique s'effectue sous le contrôle de l'application qui désigne explicitement la structure correspondante et le ou les postes de travail concernés (on parle du "postage" d'une structure). Nous reviendrons plus en détail sur ces points dans les paragraphes qui suivent.

La figure III.8 ci-contre, présente l'organisation générale de PHIGS. Il est cependant à souligner qu'il ne s'agit là que d'un modèle conceptuel, qui, s'il présente une vision centralisée à l'application, n'exclut en rien une implémentation qui stockerait les données graphiques de manière distribuée sur les différents postes de travail.



- figure III.8 : organisation et modèle conceptuel de PHIGS -

3.2.2 Structuration des données graphiques -Structure et Eléments de Structure

Les données graphiques de PHIGS sont organisées en unités appellées **structures** qui peuvent être reliées entre elles **hiérarchiquement**. A une structure correspond un identificateur unique spécifié par le programme d'application.

Les structures sont un mécanisme qui permet de regrouper les attributs de base ("structure elements") qui définissent les objets graphiques. Nous donnons ci-dessous la liste des éléments de structure en précisant à quelle classe d'attributs ils se rapportent. Ce sont :

- des primitives de sortie ("Output Primitives") (MORPHOLOGIE),
- des attributs de primitives de sortie ("Selection Attributes") (ASPECT),
- des transformations de modélisation ("Modelling Transformations") (GEOMETRIE),
- des références à d'autres structures ("Structure Execution Elements") (STRUCTURE),
- les selections de prise de vue ("Viewing Selection") (GEOMETRIE DE PRISE DE VUE),
- des étiquettes ("Labels"), (IDENTITE)
- des éléments d'identification ("Pick Identifer","Names-Sets") (IDENTITE),
- des données spécifiques à l'application

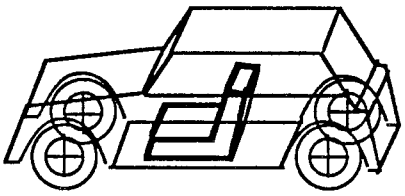
Il n'y a aucune restriction sur l'ordre et l'utilisation de ces éléments à l'intérieur d'une structure ce qui offre à l'application une totale liberté et une grande **fléxibilité** dans la représentation de ses données. Cependant, ces éléments ne peuvent être définis et exister en dehors d'une structure.

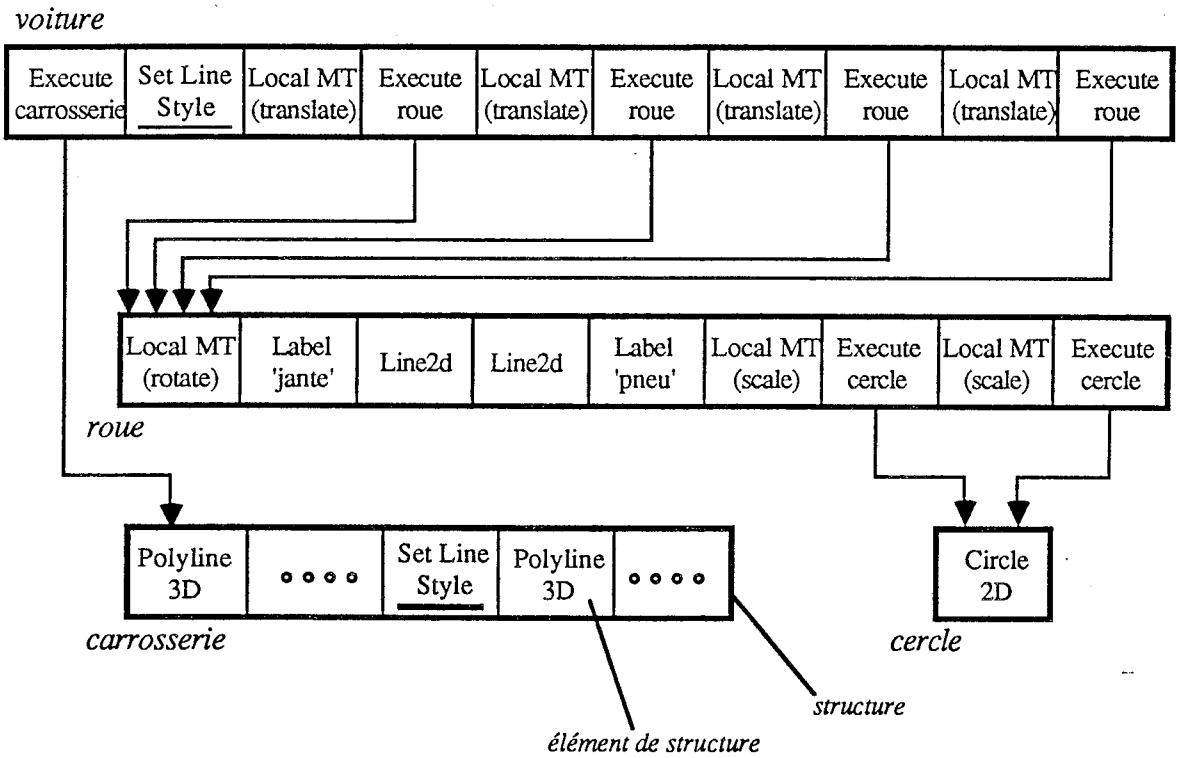
On remarquera que les éléments de structure reprennent dans une très large mesure les attributs graphiques définis pour les autres systèmes graphiques généraux, en particulier GKS. De nouveaux attributs auraient pu être définis, mais les concepteurs de PHIGS ont préféré conserver au maximum une "plate forme" commune avec GKS désormais bien établi.

Les éléments de référence à une autre structure ("structure execution") permettent de relier des structures les entre elles sous la forme un **réseau hiérarchique** de profondeur arbitraire. Ils

autorisent le **partage** de données parmi les différents niveaux. PHIGS n'impose pas de limites ni pour la profondeur d'imbrication de structures, ni pour le nombre de références pouvant être effectuées vers une même structure. La seule contrainte concerne les références aux structures qui ne doivent pas être récursirves, c'est à dire qu'une structure 1 ne peut faire référence à une structure 2 si celle ci référence la stucture 1 ou toute autre structure qui référence 1 directement ou indirectement : la hiérarchie des structures forme **un graphe acyclique.**

La figure III.9,ci-dessous, montre une représentation possible d'une voiture à l'aide de structures. Nous n'insitons pas sur la signification des éléments de structure sur lesquels nous reviendrons plus en détail par la suite (les éléments de type Local MT pour "local Modelling Transformation", sont des attributs géométriques qui permettent de positionner les différents composants de la voiture).





- figure III.9 : Structures et éléments de structure -

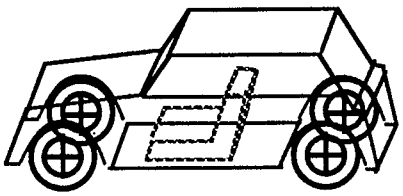
Le moment auquel les primitives de sortie définies à l'intérieur des structures sont effectivement affichées est **indépendant** du moment de leur création et introduction dans la base de données graphique. Ainsi, l'affichage d'objets s'effectue entièrement sous le contrôle de l'application qui désigne **explicitement** la ou les structures à visualiser et le ou les postes de travail concernés. Lorsqu'une structure est affichée (ou "postée") sur un poste de travail, elle est entièrement parcourue en accédant séquentiellement, à partir du premier, aux éléments qu'elle regroupe. Les attributs (attributs courants) sont évalués **dynamiquement** au fur et à mesure du parcours et s'appliquent aux primitives de sortie rencontrées qui sont alors affichées. Quand un élément d'éxécution de structure est rencontré, les actions suivantes ont lieu :

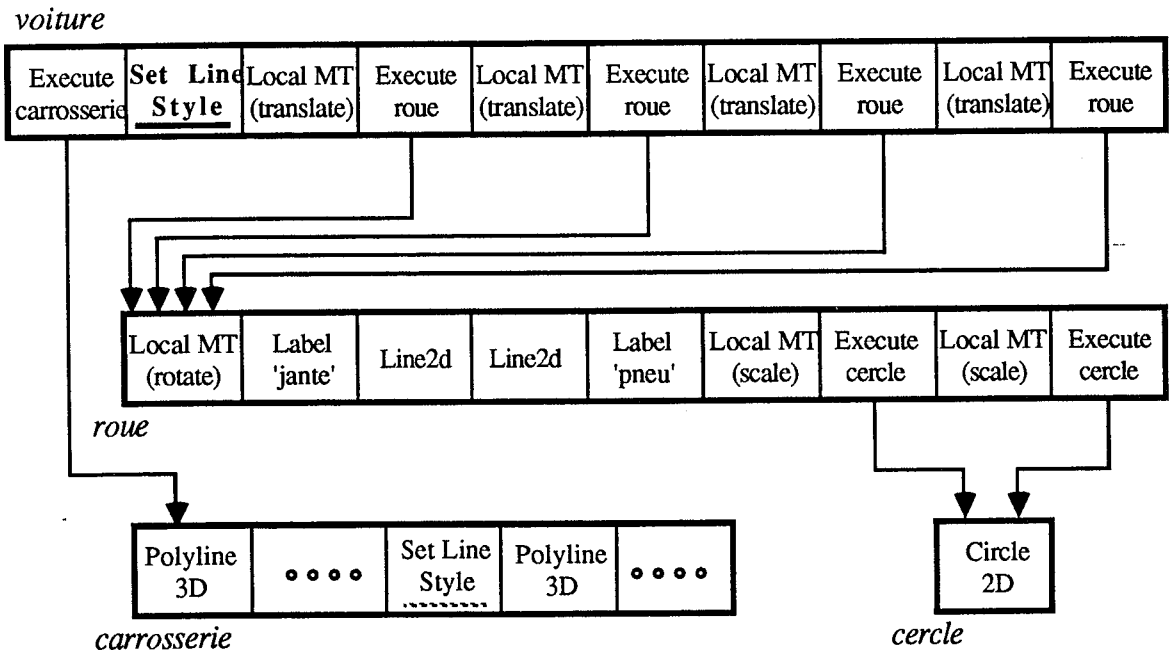
1) le parcours de la structure courante est interrompu,
2) les valeurs des attributs courants sont sauvegardées,
3) la structure exécutée (et toutes les structures qu'elle exécute) est entièrement parcourue,
4) les valeurs des attributs sont restaurées,
5) le parcours de la structure parente se poursuit.

De façon générale, l'évaluation des attributs graphiques au moment du parcours s'effectue en substituant l'attribut rencontré à l'attribut courant correspondant; sauf en ce qui concerne les attributs géométriques, pour lesquels une **composition** entre les différents niveaux de la hiérarchique peut être effectuée en lieu et place d'une simple substitution.

Un attribut affecte uniquement les primitives de sortie qui le succèdent dans la structure où il est défini ainsi que celles des éventuelles structures subordonnées qui suivent : les attributs graphiques associés aux primitives de sortie s'appliquent **hiérarchiquement** dans PHIGS. De cette manière, une primitive de sortie est affichée avec les derniers attributs définis au niveau de structure auquel elle appartient et en cas d'absence d'un ou plusieurs de ces attributs **hérite** des attributs évalués au niveau supérieur de structure. Au début du parcours d'une structure pour l'affichage, les attributs courants sont initialisés avec des valeurs par défaut qui sont ainsi appliquées en cas d'absence de définition de l'attribut correspondant dans la hiérarchie.

A titre d'exemple reprenons notre voiture. Si l'on veut particulariser les roues, par exemple en les dessinant avec un trait plus épais, il suffit de modifier, comme nous le montre la figure III.10, l'attribut d'aspect du trait ("Line style") dans la structure *voiture*. Lors du parcours de la structure pour la visualisation, il s'applique à tous les éléments qui le suivent, à savoir les primitives de sortie qui définissent la roue (les lignes pour la jante et les cercles pour le pneu).





- figure III.10 : évaluation dynamique et héritage des attributs -

Du fait de l'évaluation dynamique des attributs lors du parcours des structures pour la visualisation ("traversal time binding"), les données conservées dans la base de données graphique PHIGS peuvent être **modifiées** à tout instant par l'application. Ces modifications s'adressent soit aux structures (renomination, destruction), soit directement aux éléments de structure (insertion, suppression d'élements). Ces opérations désignées sous le terme d'édition de structures sont indépendantes du type des éléments et confèrent à PHIGS une très grande souplesse dans la modifications des données graphiques (la "granularité" de modification est la même que la "granularité" de création).

Lorsqu'une structure a été affichée, tous les changements qui lui sont apportés sont reflétés (soit immédiatement, soit de manière différée) sur le ou les postes de travail concernés. L'édition des structures évite ainsi de redéfinir complétement les données graphiques en vue de modifier l'image. Quand l'application désire ne plus visualiser une structure sur un poste de travail donné, elle la désigne explicitement comme devant être retirée de la liste des structures affichées sur ce poste. Cependant, le retrait d'une structure pour l'affichage n'affecte en rien son existence ou sa définition dans la base de données.

Le programme d'application peut également spécifier à PHIGS des structures qui ne seront pas introduites dans la base de données. Ce structures temporaires ou structures non retenues ("non retained structures") ne peuvent être modifiées.

3.3 ATTRIBUTION

Dans les paragraphes qui suivent, nous présentons les différentes primitives proposées au programme d'application en vue de la définition et de la structuration des données graphiques. Rappelons, à ce sujet que la définition des informations graphiques est disjointe de la visualisation. Aussi, les primitives que nous regroupons dans l'attribution concernent essentiellement la mise à jour de la base de données graphiques et n'ont d'effet sur l'image que si les structures auxquelles elles s'appliquent ont été explicitement affichées précédemment.

3.3.1 STRUCTURE

La possibilité d'*éditer* l'information graphique (c'est à dire de modifier à tout moment les structures et leur contenu) fait à tel point partie intégrante du système PHIGS que la création et la modification de structures s'effectuent de manière indentique. Aussi, n'existe t'il pas de primitive explicite de création de structure. Ce sont les primitives de manipulation des structures qui, si l'identificateur utilisé pour désigner la structure à laquelle elles s'appliquent n'est pas déjà associé à une structure existante (c.a.d. si l'identifiacteur est utilisé pour la première fois), créent implicitement une structure vide.

Les opérations suivantes peuvent donc provoquer la création d'une nouvelle structure:

- l'ouverture d'une structure pour édition (ajout, modification de ses éléments) d'une structure non existante,
- l'insertion à l'intérieur d'une structure d'une référence à une structure inexistante,
- l'affichage ("postage") d'une structure inexistante sur un poste de travail,
- une référence à une structure inexistante dans une primitive de modification d'identificateur de structure.

La fonction INQUIRE_STRUCTURE_EXISTENCE permet de savoir si la structure associée à un identificateur donné existe.

PHIGS permet à l'application de supprimer des structures de la base de donnée centralisée (primitives DELETE_STRUCTURE et DELETE_ALL_STRUCTURES). Dans ce cas les structures spécifiées sont détruites ainsi que toutes les références à celles-ci (éléments d'éxécution de structure) contenues dans d'autres structures. Si la ou les structures détruites ont été affichées sur un ou des postes de travail , elles sont rétirées ("depostées") de la liste des

structures visualisées sur ces postes puis supprimées de la base de données graphique. Le moment de la prise en compte de cette suppression au niveau de l'image affichée peut être contrôlé dans une certaine mesure par l'application comme nous le verrons au praraghe 3.6.3.2.

Il est également possible de "vider" complétement une structure donnée, c'est à dire de détruire tous les éléments qu'elle contient tout en maintenant son existence dans la base de données graphiques (primitive EMPTY_STRUCTURE).

3.3.2 Identificateurs de structures - IDENTITE

Les structures sont identifiées par des entiers, que l'application spécifie à son gré au moment de leur création implicite. A une structure est associé un seul identifcateur entier, et inversement à un entier est associée au plus une et une seule structure dans la base de données graphique.

Le programme d'application a la possibilité de changer les identificateurs de structure. Différentes fonctions lui sont proposées selon qu'il s'agit de modifier directement l'identificateur d'une structure, les références à un identificateur de structure (éléments d'éxécution d'une structure) ou les deux à la fois.

La primitive CHANGE_STRUCTURE_IDENTIFIER permet de remplacer l'identificateur d'une structure mais n'affecte pas les références à l'identificateur de structure original. S' il n'existe aucune référence à la structure originale, cette primitive ne pose aucun problème : l'ancien identificateur de la structure n'a plus d'existence et peut être réutilisé pour une nouvelle structure. Dans le cas contraire, l'ancien identificateur est conservé et une structure vide lui est associée. Si l'identificateur utilisé pour renommer une structure est déjà utilisé pour une autre, cette dernière est remplacée par la nouvelle structure.

La primitive CHANGE_STRUCTURE_REFERENCES remplace tous les éléments d'éxecution de structure référencant la structure originale par des invocations à la structure résultante. Si l'identificateur de structure résultant ne désigne pas une structure existante, une structure vide est crée et lui est associée.

Il est possible de combiner les deux fonctions ci-dessus en utilisant la primitive CHANGE STRUCTURE_IDENTIFIER_AND_REFERENCES

3.3.3 Eléments de structure

3.3.3.1 édition des éléments d'une structure

Comme nous l'avons évoqué précédemment, un point **primordial** dans PHIGS est la possibilité d'**éditer** les structures, c'est à dire de modifier **à tout moment** leur contenu. Ainsi, des éléments de structure peuvent être ajoutés, modifiés ou détruits, ceci quel que soit leur type (primitive de sortie, attribut, référence à une structure...) et leur position dans la structure. Grâce à cette facilité, PHIGS offre une plus grande fléxibilité et minimise les besoins de redéfinir les données en vue de les modifier (alors que dans GKS pour modifier le contenu d'un segment il faut le détruire puis le reconstruire entièrement) et permet une réelle interactivité.

Pour pouvoir être éditée, une structure doit tout d'abord être ouverte (primitive OPEN_STRUCTURE). Une seule structure peut être ouverte à la fois. Si la structure désignée n'existe pas, une nouvelle structure vide est créée et ouverte. L'édition terminée la structure doit être fermée (primitive CLOSE_STRUCTURE).

a) <u>accès aux éléments de structure - pointeur d'élément</u>

Une structure peut être considérée comme une séquence ordonnée d'éléments implicitement numérotés de 1 à n. L'accès aux éléments d'une structure se fait par l'intermédiaire d'un "pointeur d'élément" initialisé lors de l'ouverture de la structure avec le rang de son dernier élément. Les valeurs valides pour le pointeur d'élément sont dans l'intervalle 0 .. n, avec 0

référençant le début de la structure (c'est à dire la position avant le premier élément de la structure) et n référençant le dernier élément de la structure. Cette numérotation est **implicite** et est **toujours** maintenue consécutive (ainsi le numéro associé à un élément peut donc changer si des éléments sont ajoutés ou supprimés dans la structure).

Un certain nombre de fonctions de manipulation du pointeur d'élément sont proposées à l'application afin de pouvoir réaliser les différentes opérations d'édition.

Le programme d'application peut interroger PHIGS pour connaître la valeur courante du pointeur d'élément et initialiser ce dernier avec une valeur (en absolu ou relativement à la valeur courante).

Il est également possible de positionner directement le pointeur sur des éléments de structure particuliers appelés LABELS. Un label est un nom (alphanumérique) que l'on peut associer à un groupe d'éléments à l'intérieur d'une structure (primitive INSERT_LABEL) afin de les référencer pour une édition ultérieure, la primitive SET_ELEMENT_AT_LABEL positionnant le pointeur sur l'élément LABEL spécifié.

Par la suite, nous désignerons par **élément courant**, l'élément de structure indiqué par le pointeur d'élément. Des fonctions de consultation sont disponibles pour connaître son type et sa taille.

b) insertion et destruction d'éléments à l'intérieur d'une structure

Lorsque de nouveaux éléments de structure sont définis, ceux-ci sont ajoutés à la structure ouverte et insérés après l'élément courant. Après l'ajout d'un élément à une structure, le pointeur d'élément et mis à jour et désigne l'élément inséré.

Par ailleurs le contenu d'une structure dans son entier peut être dupliqué dans une autre structure (primitive COPY_STRUCTURE).

Pour la destruction d'éléments de structure, trois possibilités sont proposées :

- la destruction de l'élément courant (DELETE _ELEMENT),

- la destruction d'une séquence d'éléments compris entre deux positions incluses (DELETE ELEMENT_RANGE),

- la destruction d'une séquence d'éléments compris entre deux éléments labels non inclus (DELETE_ELEMENTS_BETWEEN_LABELS).

3.3.3.2 Exécution d'une structure - STRUCTURE

La structuration hiérarchique des données graphiques est réalisée à l'aide d'éléments d'exécution de structure, qui permet de référencer une autre structure à l'intérieur d'une structure. Les éléments de ce type sont définis à l'aide de la primitive EXECUTE_STRUCTURE. Si l'identificateur désignant la structure à référencer n'est pas encore associé à une structure de la base de données graphique, un nouvelle structure vide est créée et lui est associée.

3.3.3.3 Primitives de sortie - MORPHOLOGIE

Dans l'état actuel du projet PHIGS on peut penser qu'un concensus total existe avec GKS, en effet toutes les primitives de sortie de GKS (y compris les extensions 3D) sont supportées par PHIGS. Lors de l'invocation de l'une des fonctions primitive de sortie, celle-ci est insérée dans la structure courante actuellement ouverte. Si aucune structure n'est ouverte une erreur standard est provoquée.

3.3.3.4 Attributs des primitives de sortie - ASPECT

Les attributs qui contrôlent l'apparence des primitives de sortie reprennent, eux aussi, dans une large mesure ceux définis par GKS. Quelques attributs supplémentaires ont cependant été introduits, en particulier en ce qui concerne les tâches polygônales (FILL_AREA) pour lesquelles le contour peut être particularisé en lui associant une couleur et une épaisseur. Cependant, il est à regretter que dans un souci de compatibilité avec GKS les attributs pour la prise en compte d'un certain réalisme n'aient pas été introduits (textures, éclairage...). [Mead 86]

La définition des attributs s'effectue de manière analogue à GKS, ils peuvent être spécifiés soit :

- **individuellement** : chaque attribut est défini séparément,

- **de manière groupée** ("bundle attributes") : tous les aspects non géométriques d'une primitive de sortie sont définis par un index vers une table de groupage qui contient une représentation dépendante des postes de travail pour ces différents attributs. (Chaque poste de travail possède ses propres tables de groupage ce qui permet de spécifier l'aspect des primitives de sortie indépendamment du matériel tout en maintenant la possibilité d'adapter leur apparence visuelle aux dispositifs d'affichages particuliers).

Il est également possible de faire un compromis entre spécification individuelle et spécification groupée des attributs à l'aide de la primitive SET_ASF qui permet de selectionner les "Aspect Source Flag" indépendamment des postes de travail.

Les attributs définis dans les tables des postes de travail ne sont pas des éléments de structure, ils agissent comme des fonctions de contrôle. Par contre, les sélections d'attributs individuels, d'ASF, les index vers les tables de "groupage" sont des éléments de structure qui définissent la "réalisation" des primitives durant le parcours de la structure pour l'affichage.

3.3.3.5 Transformations de modélisation - GEOMETRIE

Comme nous l'avons vu au début de ce chapitre, les attributs géométriques (transformations de modélisation) jouent un rôle primordial dans la structuration hiérarchique des données graphiques. Ils permettent d'assembler les objets en positionnant leurs composants les uns par rapport aux autres et de partager éventuelement des composants à l'intérieur de un ou plusieurs objets (instances).

Dans PHIGS, les transformations de modélisation sont définies à l'aide d'éléments de structure qui comme pour les autres attributs s'appliquent à toutes les primitives de sortie qui les suivent (dans la structure même et dans les structures de niveaux inférieurs qui sont invoquées), et ceci jusqu'à la définition d'une nouvelle transformation de modélisation. Mais à la différence des attributs contrôlant l'apparence des primitives de sortie, les transformations géométriques peuvent être **composées** entre elles lors du parcours de la structure hiérarchique.

Ainsi, la transformation géométrique appliquée aux coordonnées décrivant les primitives de sortie (transformation de modélisation courante ou TMC) est **évaluée dynamiquement** au fur et à mesure du parcours de la structure hiérarchique. Elle est obtenue par la composition :

- de la Transformation de Modélisation Globale (ou TMG) qui correspond à la transformation de modélisation évaluée au niveau de la structure parente . (La TMG d'une structure racine est bien entendu l'identité)

- et de la Transformation de Modélisation Locale (ou TML), transformation géométrique évaluée à l'intérieur la structure courante.

$$\mathbf{TMC = TML \cdot TMG}$$

A l'entrée d'une structure, la transformation de modélisation locale courante (TML) est initialisée avec la transformation identité. Elle est ensuite modifiée chaque fois qu'un élément de structure attribut géométrique est rencontré. Ceux-ci (construits par la primitive SET_LOCAL_TRANSFORMATION_2/3) sont définis par une matrice en coordonnées homogènes (4x4) et un type de composition qui détermine la manière dont cette dernière est prise en compte dans le calcul de la transformation de modélisation locale. Les valeurs possibles pour le type de composition sont les suivantes :

- **postconcaténation** : la nouvelle transformation de modélisation locale est obtenue en postmultipliant la matrice spécifiée par la matrice de transformation de modélisation locale courante,
- **préconcaténation** : la nouvelle transformation de modélisation locale est obtenue en prémultipliant la matrice spécifiée par la matrice de transformation de modélisation locale courante,
- **remplacement** : la transformation de modélisation spécifiée remplace la transformation de modelisation locale courante.

Quand une sous-structure est invoquée, les transformations de modélisation globale (TMG) et locale (TML) courante sont sauvegardées. La transformation de modélisation courante (TMC = TML • TMC) devient la nouvelle TMG et la TML est initialisée avec la transformation identité. Au retour (le parcours de la sous-structure étant achevé), les valeurs sauvegardées de la TMG et TML sont restaurées.

Il est possible également d'agir directement sur la transformation de modélisation globale (TMG) par l'insertion d'éléments de structure (primitive SET_GLOBAL_TRANS FORMATION) qui, au moment du parcours de la structure hiérarchique, permettent de remplacer la matrice de la transformation de modélisation globale effective par une nouvelle matrice de transformation.

Pour faciliter à l'application la définition des attributs géométriques, PHIGS propose un certain nombre de primitives "utilitaires" de construction et composition de matrices de transformations géométriques (matrices de translation, rotation, mise à l'echelle, produit de matrices, transformation d'un point par une matrice...).

La figure III.11 ci-contre, reprend l'exemple de la grue que nous avions développé au début de ce chapître (voir § 2.1.1 et 2.1.2 et figures III.2,III.3 et III.4). Elle en présente une décomposition hiérarchique possible avec PHIGS, en ilustrant tout particulièrement les différentes utilisations des éléments de structure attributs géométriques.

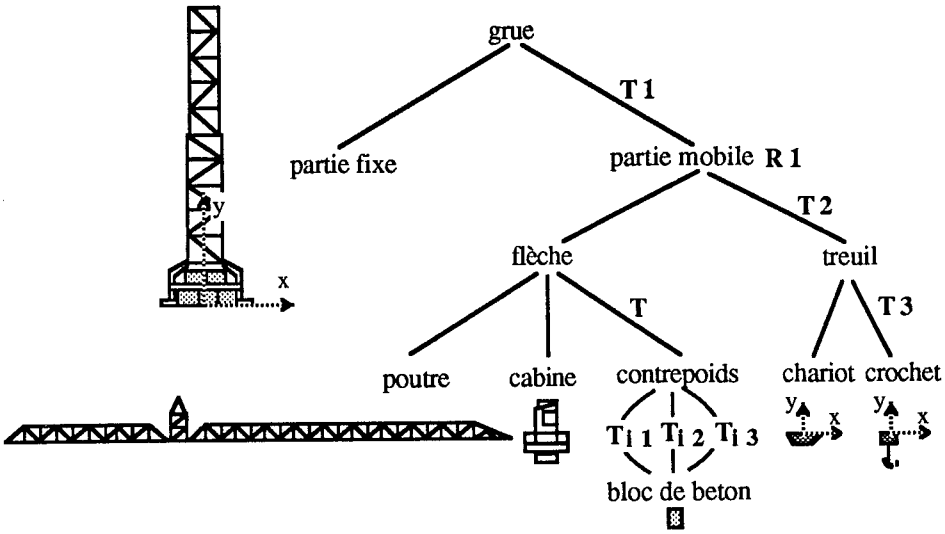
Les différentes transformations de modélisation utilisées dans la structure hiérarchique sont :

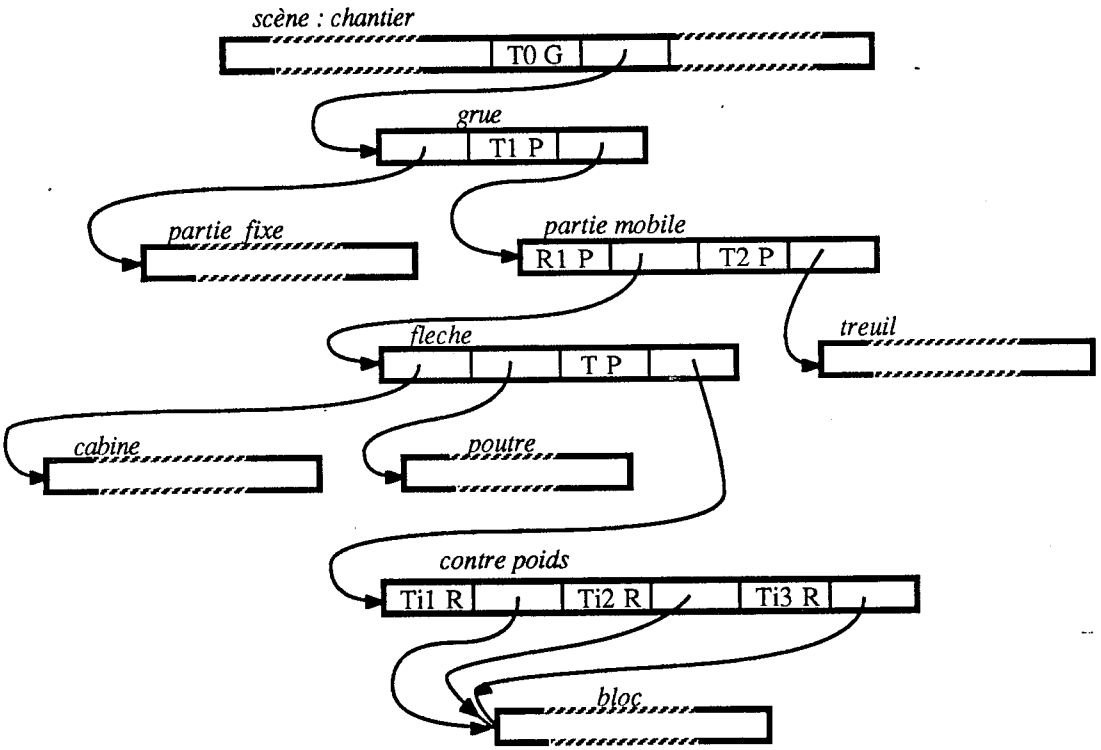
- T_0 qui positionne la grue de manière absolue (par substitution de la transformation globale courante) dans le repère de la scène visualisée,
- T_1 qui positionne la partie mobile de la grue dans le référentiel de la grue,
- R_1 qui exprime une rotation permettant de faire pivoter la partie mobile,
- T_2 qui positionne le treuil dans le référentiel de la partie mobile,
- T qui positionne le contrepoids dans le référentiel de la partie mobile,
- T_{i1},T_{i2} et T_{i3} qui placent dans le repère du contrepoids les différentes éléments qui le composent et qui sont en fait trois instances d'un même bloc.

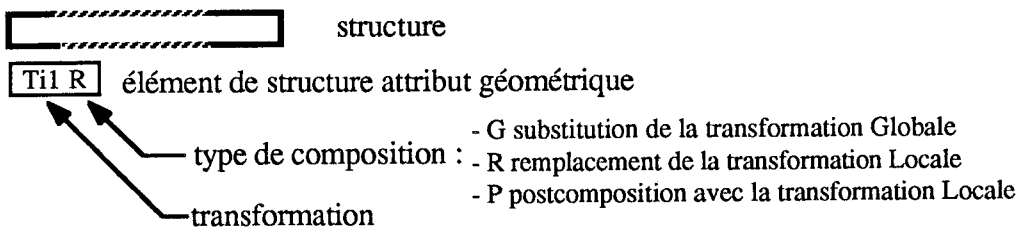
Les différentes primitives de sortie qui décrivent la morphologie de la grue sont affichées avec comme géométrie :

- T_0 pour la partie fixe,
- $R_1 \cdot T_1 \cdot T_0$ pour la cabine et la poutre,
- respectivement $T_{i1} \cdot T \cdot T_1 \cdot T_0$, $T_{i2} \cdot T \cdot T_1 \cdot T_0$ et $T_{i3} \cdot T \cdot T_1 \cdot T_0$ pour les trois instances du

bloc qui constituent le contrepoids.







- figure III.11 : Transformations géométriques dans PHIGS -

3.3.3.6 Transformation de visualisation - Géométrie de prise de Vue (Gv) - Géométrie d'Affichage (Ga)

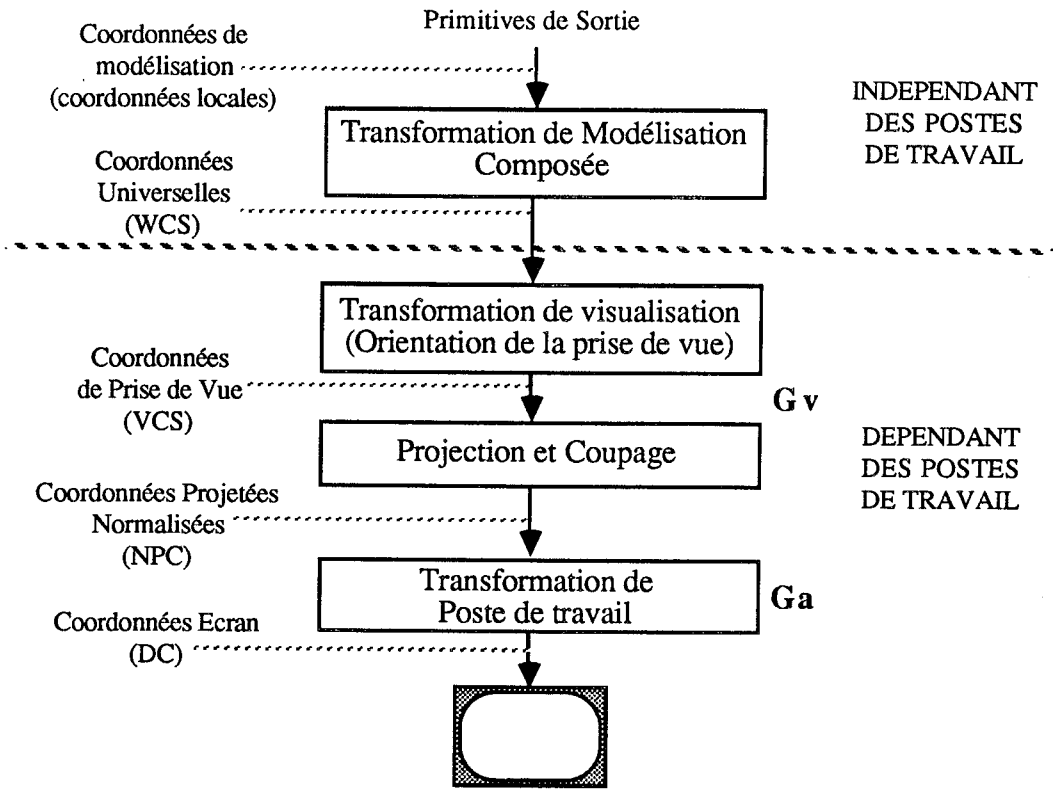
La géométrie de prise de vue est spécifiée par des éléments de structure définis à l'aide de la primitive SET_VIEW_INDEX. Ces attributs sont en fait un index qui désigne une transformation de visualisation dont la représentation est stockée dans une table sur le ou les postes de travail. Ainsi, comme pour les attributs groupés ("bundle attributes"), à une même valeur d'index peuvent correspondre des transformations de visualisation différentes selon les postes de travail.

Lors du parcours de l'arborescence pour son affichage sur un poste de travail la "prise de vue courante active" est initialisée avec la vue spécifiée dans la table du poste et s'applique ensuite aux primitives de sortie rencontrées jusqu'à ce qu'un nouvel élément SET_VIEW _INDEX soit rencontré ou jusqu'à la fin de la structure.

Comme le CORE SYSTEM ou GKS-3D, PHIGS permet de définir des transformations de prise de vue très générales qui englobent toutes les projections géométriques planes. Une prise de vue sur un poste de travail est définie en deux étapes par :

- une transformation de visualisation ("viewing transformation") qui défini un repère de visualisation ("Viewing Coordinate System" ou VCS) et le passage du repère universel (WCS) à celui-ci,

- une projection ("view mapping transformation") qui effectue le passage du repère VCS à un repère de projection normalisé ("Normalized Projection Coordinate Space" ou NPC). Cette transformation est définie par un type de projection (parallèle ou perspective), un point de vue ("View Reference Point"), la distance du plan de projection par rapport au point de vue, une fenêtre dans ce plan de projection, la distance des plans de coupage avant et arrière, des indicateurs pour le coupage et une cloture 3D ("3D viewport", parallélépipède rectangle du NPC).



- figure III.12 : les transformations géométriques dans le processus de visualisation PHIGS -

Comme pour les transformations de modélisation PHIGS propose un certain nombre de primitives utilitaires pour l'élaboration de la matrice de visualisation.

La géométrie d'affichage est définie par une transformation de poste de travail ("Workstation Transformation") qui transforme les coordonnées projetées normalisées (NPC) en coordonnées écran ("Device Coordinate" ou DC) du dispositf physique. Cette transformation est définie en sélectionant un volume de départ dans l'espace NPC ("Workstation Window") et en lui associant un volume d'arrivée dans l'espace écran ("Workstation Viewport"). Cette transformation permet le positionnement sélectif d'une image dans l'espace écran d'un poste de travail.

3.3.3.7 IDENTITE - "Names-Sets" - Filtres - Visibilité, Surbrillance, Détectabilité.

Très souvent, les applications ont besoin de modifier la visibilité, la "surbrillance" (c.a.d. la mise en évidence, par un procédé qui peut effectivement être la surbrillance mais également un changement de couleur ou un clignotement, cela dépend des implémentations) ou la déctabilité d'un ensemble de primitives. Si dans GKS, ces attributs sont naturellement directement associés à l'unité de structuration que constituent les segments, pour PHIGS le problème se pose de manière tout à fait différente du fait même de l'organisation hiérarchique des données graphiques. En effet, le regroupement des primitives visibiles, en surbrillance ou detectables est très dépendant des applications. Ainsi, ces primitives ne font pas toujours partie d'une même structure, pas plus qu'elles ne sont nécessairement reliées dans une même structure hiérarchique. D'autre part, les ensembles des primitives detectables, visibles et en surbrillance ne sont pas toujours disjoints.

Pour résoudre cette difficulté, PHIGS propose une méthode souple basée sur un attribut d'identité, l'**ensemble de noms** ("NAME-SET"), et des **filtres** associés de manière indépendante aux opérations de visibilité, surbrillance et détectabilité.

Le NAME-SET est un attribut dont la valeur est élaborée dynamiquement lors du parcours de la structure hiérarchique (pour une visualisation ou pour retrouver l'élément désigné lors d'une identification interactive). Celle-ci est définie à partir d'élément de structure particuliers qui permettent d'ajouter ou retirer des éléments (identificateurs) au NAME-SET (primitives ADD_NAMES_TO_SET et REMOVE_NAMES_TO_SET). Comme pour les autres attributs, la valeur du NAME-SET est sauvegardée à l'entrée d'une sous-structure et restaurée au retour.

Les filtres sont également des ensembles d'identificateurs qui en liaison avec le NAME-SET permettent de définir la visibilité, la surbrillance ou la détectabilité d'une primitive. Pour chacun de ces attributs (visibilité, surbrillance, détectabilité) sont définis deux filtres : l'un d'inclusion et l'autre d'exclusion.

Pour être visible, en surbrillance ou détectable une primitive de sortie doit avoir :

- **au moins** un des éléments de son NAME-SET dans dans le filtre d'inclusion correspondant,

- **aucun** des éléments de son NAME-SET dans le filtre d'exclusion correspondant.

Les filtres d'inclusion et d'exclusion pour la visibilité, la surbrillance et la détectabilité sont définis en dehors de toute structure. Leur valeur est établie de façon dépendante des postes de travail à l'aide de fonctions de contrôle (c.a.d. que lors de l'emploi simultané de plusieurs postes de travail, un même filtre peut avoir des valeurs différentes sur chacun de ceux-ci).

A titre d'exemple considérons le cas d'une structure représentant la conception d'un batiment pour lequel on désire distinguer les équipement des différents étages (plomberie, electricité, isolation, etc...). Afin de pouvoir identifier séparément et selectivement ces différents éléments la structure PHIGS sera construite comme suit :

Definition de la structure	*Valeur du "NAME-SET" lors du parcours de la structure pour l'identification*
.....	
ADD_NAMES_TO_SET(1er étage,plomberie)	{ 1er étage , plomberie }
.....	
éléments définissant la plomberie du premier étage	
.....	
REMOVE_FROM_NAME_SET(plomberie)	{ 1er étage }
ADD _NAMES_TO _SET(éléctricité)	{ 1er étage , électricité }
.....	
éléments définissant l'électricité du premier étage	
.....	
REMOVE_FROM_NAME_SET(1er étage)	{électricité }
ADD _NAMES_TO _SET(2ème étage)	{ 2ème étage , électricité }
.....	
éléments définissant l'électricité du deuxième étage	
.....	
REMOVE_FROM_NAME_SET(éléctricité)	{ 2ème étage }
ADD _NAMES_TO _SET(plomberie)	{ 2ème étage , plomberie }
.....	
éléments définissant la plomberie du deuxième étage	
.....	

Ainsi si l'on souhaite que l'opérateur puisse uniquement identifier l'électricité du 1er étage, on initialisera le "filtre de détection" du poste de travail concerné (fonction SET_DETECTABILITY_ FILTER) avec pour paramètres d'inclusion *{ 1er étage, électricité }* et pour paramètres d'exclusion *{ 2ème étage, plomberie, isolation}*.

3.4 CONSULTATION

En ce qui concerne la consultation PHIGS, permet au programme d'application de récupérer un certain nombre d'informations sur l'état du système et le contenu de la base de données graphique.

Pour l'état du système graphique, il s'agit de fonctions (INQUIRY_FONCTION) qui retournent les informations contenues dans les différentes listes d'état maintenues par PHIGS (par exemple indication de la structure ouverte, status des postes de travail....) et dans les tables de description propres à chaque poste de travail (possibilités du poste, valeurs des attributs dans les tables de groupage....).

Pour les informations stockées dans la base de données graphique, PHIGS permet la consultation des identificateurs de structure. Il est ainsi possible de savoir si un identificateur est déjà associé ou non à une structure existante, de retrouver la liste des identificateurs de toutes les structures existantes (sous forme d'un tableau), d'obtenir la liste des identificateurs des structures qui constituent une hiérarchie ou un réseau (INQUIRE_STRUCTURE_PATH) à partir d'une structure donnée.

D'autre part, il est possible d'accéder aux informations concernant les éléments de structure : comme nous l'avons vu lorsque nous avons présenté l'édition des structures, l'application peut récupérer la valeur du pointeur d'élément courant, le type et de la taille de l'élément courant, et éventuellement l'information contenu dans l'élément de structure lui-même. Pour cette dernière, il est néanmoins à remarquer que la primitive de consultation (INQUIRE_ELEMENT

_RECORD), fournit à l'application "l'enregistrement" complet associé à l'attribut. Cela impose à cette dernière de connaître la représentation des différents éléments de structure pour pouvoir en extraire l'information définissant les attributs graphiques.

3.5 DESCRIPTION

PHIGS contrôle les dispositifs de dialogue de manière identique à GKS, à l'aide du concept de dispositifs logiques d'entrée. On retrouve donc les six classes logiques d'entrée : releveur de coordonnées ("Locator"), releveur d'une suite de coordonnées ("Stroke"), entrée scalaire ("Valuator"), sélecteur ("Choice"), désignation ("Pick"), entrée de chaîne de caractères ("String"). Chaque dispositif d'entrée logique peut être exploité suivant les trois modes : requête ("Request"), échantillonage ("Sample") et événement asynchrône ("Event").

La seule variation notable entre PHIGS et GKS concerne l'identification et est directement liée à la différence de structuration des données graphiques autorisée par les deux systèmes. Dans GKS l'identification retourne simplement un numéro de segment et éventuellement un numéro d'identification ("Pick Identifier"). En ce qui concerne PHIGS, l'identification d'une primitive désignée interactivement ne peut se limiter à l'identificateur de la structure à laquelle elle appartient et au NAME-SET éventuellement associé (cf. §3.3.3.7). En effet, du fait de l'organisation hiérarchique des données graphiques, une même structure peut être instanciée (invoquée) plusieurs fois et en conséquence affichée plusieurs fois simultanément. Aussi, pour identifier **sans ambiguité** la primitive graphique désignée, est-il nécessaire de retourner un "chemin d'identification" ("PICK PATH") complet qui correspond à la séquence de parcours des éléments entre la structure racine et la primitive. Le "chemin d'identification" est présenté sous la forme d'un tableau dont chaque entrée désigne l'élément de structure d'un niveau et contient l'identificateur de la structure dans laquelle il se situe, son rang dans celle-ci ainsi que le NAME-SET qui lui est associé. L'élément terminal est la primitive désignée, tous les autres éléments correspondent à des références de structure (éléments d'éxécution de la structure de niveau inférieur).

Par ailleurs, PHIGS propose, grâce aux attributs NAME-SET et au mécanisme de filtre que nous avons vus au 3.3.3.7, de restreindre l'ensemble des éléments identifiables à un instant donné.

3.6 VISUALISATION

3.6.1 Affichage de structures ("structure display")

La visualisation des structures s'effectue de façon dépendante des postes de travail et sous l'entier contrôle de l'application. Ainsi, un structure n'est affichée que si elle a été explicitement associée à un ou des postes de travail à l'aide de la primitive POST. (Par la suite, nous désignerons un telle structure par le terme de structure "postée").

Lorsqu'une structure postée est affichée, le reseau hiérarchique qu'elle définit est parcouru et les attributs qu'il regroupe sont évalués dynamiquement. Au début de ce parcours, les attributs modaux sont initialisés à partir de valeurs par défaut contenues dans la table de description de PHIGS. Si la structure spécifiée dans l'appel de POST n'existe pas dans la base de données graphique, une nouvelle structure vide est automatiquement créée.

Plusieurs structures peuvent être postées simultanément sur un même poste de travail. Aussi, au moment du "postage" d'une structure, l'aplication peut lui associer une priorité d'affichage.

Lorsque des modifications (éditions) sont effectuées sur une structure postée, PHIGS assure automatiquement la mise à jour de l'image sur le (les) poste(s) de travail concerné(s). L'application conserve néanmoins un certain contrôle, sur une une base dépendante des postes de travail, afin que les modifications de l'image puissent être effectuées de manière efficace en fonction des possibilités du matériel (voir § 3.6.2).

Ainsi une structure postée reste affichée sur un poste de travail , avec chaque fois que nécessaire la répercursion sur l'image des modificatons qui lui sont apportées, tant que l'application ne la retire pas explicitement de la listes des structures visualisées sur ce dernier.(Pour chaque poste de travail, PHIGS conserve la liste des structures qui ont été postées). Ce retrait d'une structure de l'affichage sur un poste de travail s'effectue à l'aide de la primitive UNPOST. Il est possible de retirer toutes les structures à la fois (primitive UNPOST_ALL).

Le système PHIGS permet à une implémentation de tirer parti au mieux des possibilités de matériels évolués en reportant la réalisation du processus de visualisation dans la partie dépendante du matériel [AbBu 86]. Ainsi, les structures postées peuvent être éventuellement stockées et parcourues localement sur un poste de travail, minimisant les interactions entre le calculateur hôte et le poste.

Cependant, même si le processus de visualisation est pris en charge automatiquement par le système graphique, et si l'évaluation dynamique des attributs permet des modifications interactives de l'image (à condition que le parcours des structures soit suffisament rapide), PHIGS, comme GKS, est un standard qui dans une certaine mesure est surpassé par la technologie [Mead 86]. En effet, il n'offre pas réélement la possibilité de visualiser des solides 3D sous forme réaliste en n'intégrant pas dans sa définition standard l'ombrage ("shading"), les modèles d'éclairage, l'élimination des parties cachées. PHIGS n'indique pas comment ces fonctionnalités pourraient être introduites dans le système, ce qui si elles sont prises en compte le rend dépendant de l'implémentation.

3.6.2 Contrôle de l'affichage (Deferring picture change)

PHIGS permet à l'application de contrôler l'efficatité des opération d'affichage sur les différents postes de travail. Ce contrôle s'effectue de manière très similaire à celui effectué dans GKS en différant les mises à jour de l'image sur une base dépendante des postes de travail.

La primitive SET_DEFERRAL_STATE permet à l'application de choisir un affichage différé qui prend en compte le mieux les possibilités du matériel en fonction de ses besoins. Deux attributs sont définis pour cela :

DEFERAL_MODE : qui contrôle l'instant où sont pris en compte les effets visuels des requètes qui consistent simplement en un ajout de données, c'est à dire :

- l'insertion d'une primitive de sortie ou d'un élément d'execution d'une autre structure dans une structure postée,
- la copie du contenu d'une structure dans une structure postée,
- le "postage" d'une structure.

Les valeurs possibles pour le mode différé sont les suivantes :

- ASAP As Soon as Possible,
- BNIL Before Next Interaction Locally,
- BNIG Before Next Interaction Globally,
- ASTI At Some Time,
- WAIT l'effet visuel des fonctions est retardé jusqu'à ce que le mode diferré soit modifié par une autre valeur.

DISPLAY_CHANGE : qui contrôle l'instant où les changements dans la base de données graphique autres que de simples ajouts ainsi que les modifications dans les tables des postes de travail ont leur effet visuel pris en compte. Selon les possibilités physiques du/des postes de travail, certaines de ces modifications peuvent être immédiatement répercutées sur l'image affichée, tandis que pour d'autres leur prise en compte nécessite une régénération complète de l'image c'est à dire la revisualisation de toutes les structures postées. (Une régénération

complète peut être également explicitement demandée par l'invocation de la primitive REDRAW_ALL_STRUCTURES). Ce mode permet à l'application de s'adapter et de tirer profit des capacités du matériel graphique, en fixant un éventuel délai pour la prise en compte des modifications nécessitant une régénération implicite. Deux valeurs sont possibles :

- ALLOWED : la régénération implicite de l'image est autorisée chaque fois que nécessaire,

- SUPPRESSED : l'effet des modifications nécessitant une régénération de l'image est différé jusqu'a ce qu'une prise en compte de celles-ci soit explicitement demandée.

Les actions différées peuvent être rendues "visibles" à tout moment par un changement approprié du mode DEFERRAL_STATE ou bien par l'utilisation de la primitive UPDATE WORKSTATION.

Au niveau de chaque poste de travail, il existe sa la table de description un certain nombre d'entrées qui indiquent quelles modifications peuvent être apportées dynamiquement (immédiatement) à l'image et quelles opérations conduisent à une régénération implicite. En consultant ces entrées, le programme d'application peut ainsi adapter son comportement au matériel utilisé pour son exécution et ainsi tirer un meilleur profit de ses performances.

3.6.3 Visualisation temporaire sans mémorisation - Structure temporaire ("non retained structure")

L'application peut si elle le désire visualiser des informations de façon temporaire sans nécessairement passer par la définition de structures stockées dans la base de données graphique. Pour cela, elle utilise une structure particulière qui conceptuellment n'est pas mémorisée et qui est désignée sous le terme de structure temporaire ("non retained structure").

Les éléments définis après l'ouverture de la structure temporaire (primitive OPEN_NON_RETAINED_STRUCTURE) ne sont pas mémorisés et sont directement utilisés pour l'affichage. Les attributs graphiques sont évalués et les primitives de sortie visualisées au fur et à mesure de leur définition. Si une structure mémorisée est invoquée, elle est immédiatement parcourue en héritant des attributs courants évalués au niveau de la structure non retenue. (A l'ouvertue de la structure non retenue, ces attributs courants sont initialisés avec les valeurs par défaut définies dans les tables de description de PHIGS).

Les postes de travail sur lesquels les éléments définis dans la structure temporaire sont affichés sont explicitement désignés à l'aide de la primitive POST_NON_RETAINED_STRUCTURE. Si cette désignation intervient après l'ouverture de la structure non retenue, seuls les éléments définis par la suite sont pris en compte sur le poste indiqué.

La primitive UNPOST_NON_RETAINED_STRUCTURE permet de suspendre sur un poste de travail, l'affichage des éléments non mémorisés.

Quand la structure non retenue est fermée (primitive CLOSE_NON_RETAINED_STRUCTURE), les éléments qui ont été définis précédemment restent affichés sur chacun des postes de travail alors sélectés, ceci jusqu'à ce qu'une régénération de l'image intervienne.

Cette facilité, permet à l'application de définir des images temporaires, qui ne sont pas modifiées (ces structure ne sont pas "éditables") et ne nécessitent donc pas de mémorisation. Le mécanisme est le même que pour les structures conservés dans la base de données, et tous les types d'éléments de structure peuvent être utilisés offrant ainsi la même puissance de description à l'application. Cependant, si la visualisation sans mémorisation porte sur un objet structuré hiérarchiquement, les structures de niveaux inférieurs doivent être définies auparavant et donc mémorisées dans la base de données graphique, ceci même si elles ne sont plus utilisées par la suite.

3.8 Archivage des structures

PHIGS offre la possibilité de conserver sur support magnétique des structures individuelles et/ou des réseaux de structures (la structuration hiérarchique des données est conservée et peut être ainsi récupérée) dans ce qui est désigné sous le terme de **fichiers d'archive**. Si, ces derniers jouent un rôle analogue aux méta-fichiers de GKS, leur format est néanmoins totalement différent et incompatible.

Après avoir ouvert un fichier d'archive (primitive OPEN_ARCHIVE_FILE), l'application peut lui ajouter des structures (ARCHIVE_STRUCTURE), des réseaux de structures, c'est à dire une structure et tous ses descendants (ARCHIVE_STRUCTURE_NETWORK), ou toutes les structures actuellement définies dans la base de données graphique (ARCHIVE_ALL _STRUCTURES). L'archivage terminé le fichier d'archive doit être fermé à l'aide de la primtive CLOSE_ARCHIVE_FILE.

Des fonctions d'interrogation permettent de connaître le contenu d'un fichier d'archive. Il est ainsi possible de récupérer les identificateurs de toutes les structures archivées.

De façon symétrique, les structures stockées dans un fichier d'archive peuvent être récupérées et introduites dans la base de données graphique. Ceci s'effectue, après avoir ouvert le fichier d'archive concerné, à l'aide des primitives :

- RETRIEVE_STRUCTURE qui permet de récupérer uniquement une structure désignée dans le fichier. (Si à cette structure étaient subordonnées d'autres structures également archivées, celles-ci sont aussi récupérées afin de restituer la structure hiérarchique originale),

- RETRIEVE_ALL_STRUCTURE qui récupère toutes les structures conservées dans le fichier.

Lors des opérations d'archivage ou de récupération de structures archivées, des conflit d'identificateurs peuvent intervenir : l'identificateur de la structure à archiver est utilisé par une structure déjà présente dans le fichier d'archive ou bien l'identificateur de la structure à récupérer est déjà utilisé dans la base de données graphique.

La résolution de ce type de problèmes s'effectue en fonction de la valeur d'un indicateur (positionné par la primitive SET_CONFLICT_RESOLUTION) et qui peut être :

- soit MAINTAIN, dans ce cas la structure déjà présente est conservée et la nouvelle structure ignorée,

- soit REPLACE, dans ce cas la structure déjà présente est supprimée et remplacée par la nouvelle structure.

3.7 Conclusion

A travers l'organisation hierarchique des données graphiques et la possibilité d'éditer à tout moment les éléments de structure (ceci de manière totalement indépendante des postes de travail, l'évaluation des attributs graphiques s'effectuant au moment du parcours d'une structure pour sa visualisation), PHIGS permet de répondre aux exigences d'applications dont les besoins fonctionnels ne peuvent être entièrement satisfaits par des systèmes graphiques tels GKS ou le CORE-SYSTEM. Il rend possible la conception des applications dynamiques, hautement interactives qui sinon auraient du être construites au dessus du CORE-SYSTEM ou de GKS et n'auraient donc pu exploiter de manière vraimment efficace les nouveaux materiels graphiques.

En effet, comme le constate J. Bono [Bono 85], l'implémentation de l'ensemble des fonctionnalités proposées par PHIGS requière des materiels hautement performants, que soit pour le calculateur hôte ou pour les dispositifs d'affichage (par exemple des terminaux du niveau Tektronix 4115, Ramtek 9640..)

Aussi, plutôt qu'un "nouveau" standard, certains considèrent PHIGS comme un "**autre**" standard, qui, d'un niveau plus élevé, répond à des besoins differents que ceux adressés par GKS (ou le CORE-SYSTEM) tant du point de vue des applications que de la prise en compte du materiel graphique.

Ainsi, est-il raisonnable de penser que ces sytèmes constituent une famille de "standards" qui permettent de couvrir l'ensemble des besoins des applications :

- le CORE SYSTEM ou GKS pour les applications "bas de gamme" qui produisent des images mais ne nécessitent pas une grande interactivité (par exemple la visualisation de résultats en physique ou mathématiques..),

- PHIGS pour des applications plus sophistiquées nécessitant un graphique dynamique avec un haut degré d'interactivité (par exemple la CAO mécanique...).

Diverses implémentations de PHIGS ont été déjà réalisées, avant même que le système ne soit entièrement défini par les instances de normalisation. L'une des toutes premières a été FIGARO de Template, la division logiciel de Megatek [Mead 86], on peut également citer parmi d'autres GRAPHIGS chez I.B.M... Par ailleurs, PHIGS a influencé la conception de certains matériels haut de gamme qui proposent des fonctionnalités proches des siennes [Gros 86]. Ce sont ces materiels sophistiqués, "PHIGS-Like", que ce système doit rendre accessibles aux programmeurs d'application au travers d'une interface de haut niveau.

4 CONCLUSION

Ce chapitre a tenté de montrer l'interêt primordial que présente la structuration hiérarchique des données dans le contexte du graphique. Elle sert de base à la modélisation géométrique qui offre un grand pouvoir de description et manipulation pour des objets complexes.

A ce titre, PHIGS est l'exemple même d'un système tirant parti d'une telle structure pour la représentation des objets graphiques. En prenant en charge l'activité de modélisation et la structuration hiérarchique des objets il offre :

- une puissance de description nettement supérieure à celle que peuvent proposer GKS, le CORE-SYSTEM ou tout autre système graphique général n'adressant que les problèmes propres à la visualisation.

- une plus grande interactivité, de par la possibilité de modifier à tout instant tous les éléments quels que soient leur classe et leur niveau de structure et en répercutant directement ces modifications sur l'image affichée.

PHIGS met en évidence, la nécessité de ne pas dissocier l'activité de modélisation de la visualisation afin de définir un système graphique puissant et cohérent. En déchargeant les applications de la gestion du modèle des objets, le système peut en particulier répercuter immédiatement et efficacement sur l'image affichée toutes les modifications apportées à la description des objets.

C'est ce qu'offre PHIGS, en prenant en charge la modélisation, en dissociant la construction des structures de leur visualisation et en permettant de modifier tous leurs éléments de manière cohérente. C'est également ce que nous nous sommes proposé de réaliser avec le système CLOVIS (Complexe Logiciel pour la Visualisation Interactive Structurée) que nous présentons dans la deuxième partie de ce rapport.

2ème PARTIE

CLOVIS une première Expérimentation pour la Visualisation Interactive Structurée

Chapître IV : CLOVIS du point de vue utilisateur

1 INTRODUCTION

CLOVIS (Complexe Logiciel pour la Visualisation Interactive Structurée) est un système **graphique général** que nous avons conçu et réalisé au sein de l'équipe graphique du laboratoire ARTEMIS. Il est destiné à répondre à une vaste gamme d'applications allant de la Conception Assistée par Ordinateur à la synthèse d'images.

La réalisation effective du prototype CLOVIS, nous a permis de valider l'ensemble des idées que nous avons développées au cours de la première partie de ce rapport. Ainsi, l'objectif essentiel qui a guidé la définition de ce système était de proposer aux programmeurs d'application une interface de haut niveau les déchargeant complètement des problèmes purement graphiques, tout en leur laissant une grande liberté de choix dans la manipulation des objets graphiques [Mart 82].

Pour cela, les objets graphiques sont décrits à partir d'un ensemble d'attributs élémentaires (correspondant aux différentes classes que nous avons mises en évidence au cours du chapitre I) **structurés hiérarchiquement**. Ce choix, d'une structuration hiérarchiques des données graphiques, offre, comme nous l'avons vu au chapitre précédent, une grande puissance de description et une souplesse de manipulation inégalée. Afin d'assurer une interface de haut niveau avec les programmes d'applications, CLOVIS prend entièrement en charge la gestion de ces structures de données graphiques hiérarchiques. Il nous a ainsi été possible de définir un ensemble **restreint** et **cohérent** de primitives qui correspondent aux quatre processus de base de la synthèse d'image que nous avons présentés au chapitre I et qui sont l'attribution, la consultation, la description et la visualisation. Ces primitives assurent la structuration et l'exploitation des données graphiques et permettent aux programmes d'application de communiquer de manière **simple** et **efficace** avec le système graphique.

L'application des primitives associées aux quatres processus de base est possible à n'importe quel niveau de la structure hiérarchique décrivant les objets, garantissant ainsi la **cohérence** du système. La définition de processus de description interactive offre une **symétrie** que l'on ne retrouve pas dans les systèmes que nous avons étudiés dans la partie qui précède. Finalement le mise en oeuvre d'un processus de consultation complet et cohérent assure l'**ouverture** du système graphique. Ainsi CLOVIS et un prototype de système graphique qui répond d'une manière que nous jugeons satisfaisante aux critères de qualité que nous avions évoqués au chapitre I de cette étude.

Dans ce chapître nous présentons le système CLOVIS du point de vue fonctionnel. Ainsi,après avoir rapidement présenté le système de très générale, nous décrirons en détail ses diverses fonctionnalités au travers des différentes primitives qu'il propose à l'utilisateur. Le chapitre V qui suit s'attechera quand à lui aux aspects plus techniques de cette réalisation.

2 PRESENTATION GENERALE CLOVIS

2.1.La base de données hiérarchique

CLOVIS met à la disposition des concepteurs d'applications graphiques un ensemble de primitives lui permettant la définition et la structuration des objets graphiques de manière **hiérarchique**. Les informations graphiques sont regroupées au sein d'une "base de données" hierarchique unique, entièrement gérée par le système CLOVIS.

Le choix de la structuration des données graphiques sous forme hiérarchique découle de la puissance et de la souplesse qu'offre une telle représentation comme nous avons pu le voir dans le chapitre précédent. Elle permet ainsi au programmeur d'application de définir et d'exprimer les objets qu'il manipule beaucoup plus facilement et sans limitation au contraire des logiciels graphiques classiques qui proposent une structure rigide à deux ou trois niveaux (CORE-SYSTEM, GKS, GRIGRI).

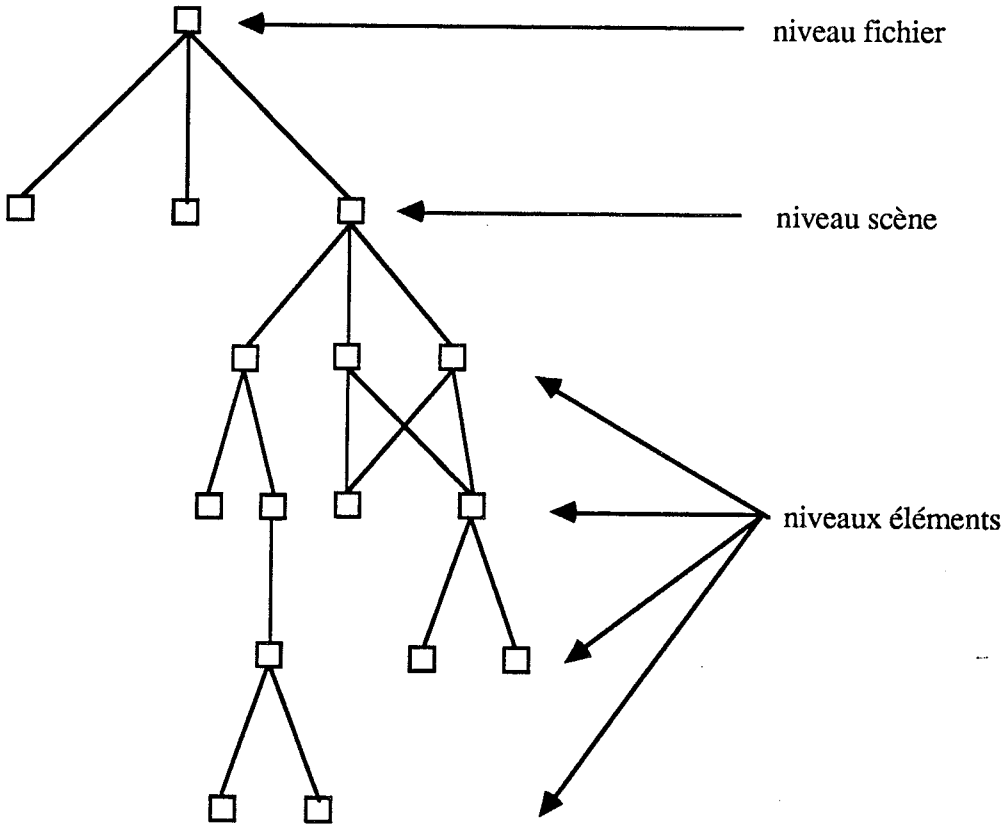
La "base de données" graphiques hiérarchique gérée par CLOVIS présente d'un point de vue logique trois niveaux de structuration (fig IV.1) :

- le niveau **fichier** créé à l'initialisation du système graphique, racine de l'arborescence totale et qui est bien évidemment toujours présent,

- le niveau **scène** qui correspond aux racines des différentes structures qui seront créées dans la base de données graphique,

- le niveau **élément** qui permet la description des scènes graphiques par une décomposition hiérarchique des objets en sous-objets. Il est à remarquer comme le montre la figure V.1 qu'il est possible de **partager** des structures au niveau élément. C'est à dire que plusieurs noeuds différents peuvent faire référence à des sous-structures identiques. Cela correspond à la notion d'**instance** qui permet de partager des objets identiques en leur affectant des attributs différents (en particulier des attributs géométriques). (La base de données graphique n'est donc pas une véritable arborescence, mais un graphe sans circuits...).

Par ailleurs, le nombre de niveaux éléments est illimité ce qui permet au programme d'application de choisir en toute liberté une structuration judicieuse de ses données graphiques parfaitement adaptée à ses besoins.



- figure IV.1 : Les différents niveaux de structuration dans CLOVIS -

Comme nous le verrons par la suite **toutes** les opérations graphiques peuvent être appliquées de manière **identique** et **cohérente** à tous les niveaux de structure.

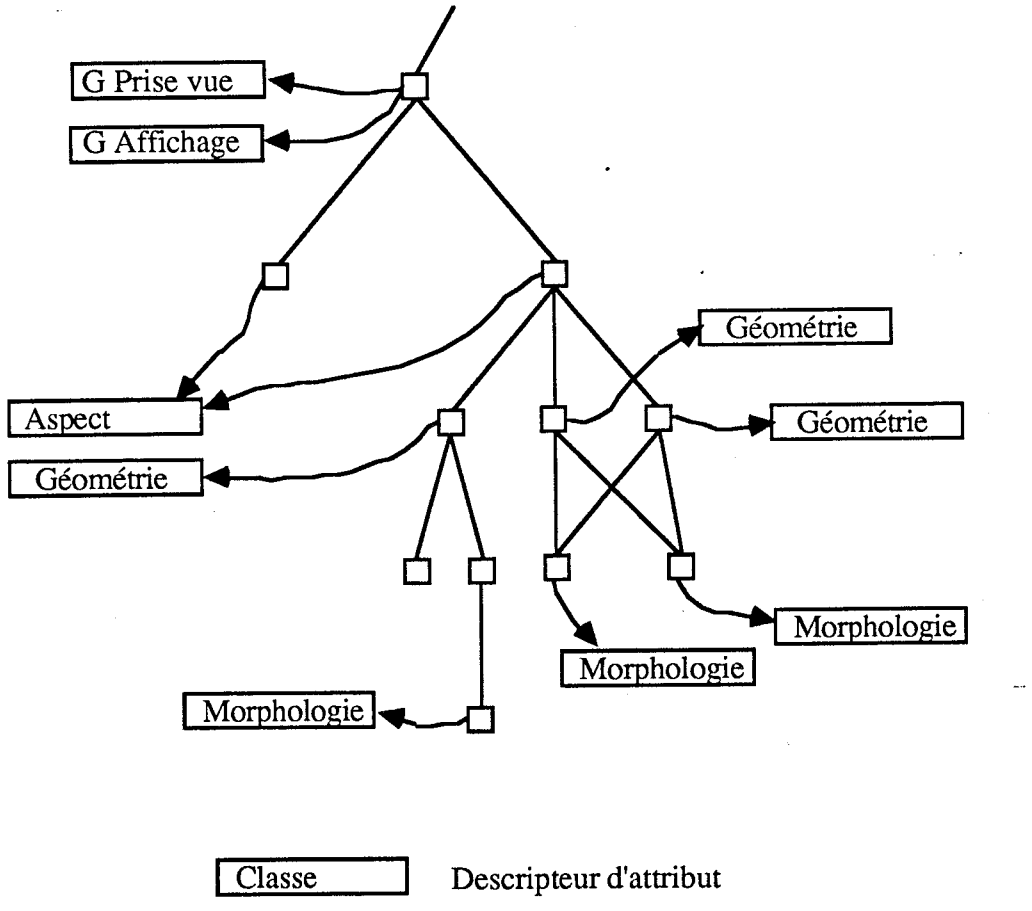
2.2 Les attributs élémentaires dans CLOVIS

Aux différents éléments de la base de données hiérarchique, il est possible d'associer un ensemble d'**attributs graphiques élémentaires** regroupés selon les différentes **classes** que

nous avons mises en évidence au chapitre I, à savoir l'Aspect, la Géométrie, la Morphologie, l'Eclairage, la Géométrie de prise de vue et la Géométrie d'affichage. Les attributs de Structure et d'Identité sont quand à eux inhérents à l'organisation hiérarchique de la base de données graphique que nous avons exposée au paragraphe précédent.

A l'intérieur de chacune de ces classes existent un certain nombre de types d'attributs (par exemple le type d'attribut Cercle2D à l'intérieur de la classe Morphologie). A chaque type d'attribut correspond une structure de données particulière qui permet sa modélisation et que nous désignerons sous le terme de **descripteur**. La création de ces descripteurs d'attributs et la gestion de l'espace mémoire qui leur est alloué peuvent être entièrement confiés au logiciel CLOVIS.

Les descripteurs d'attributs, quels que soient leur classe et leur type, peuvent être rattachés **dynamiquement** à n'importe quel élément de la base de données graphiques hiérarchique indépendamment de son niveau dans la structure, comme nous le montre la figure IV.2. Un même attribut peut être partagé par plusieurs éléments, mais par contre **un seul** attribut par classe peut être associé à un noeud de la structure hiérarchique. Les attributs sont **hérités** dans la structure hiérarchique : les attributs associés à un élément s'appliquent récursivement à tous les éléments de niveau hiérarchique inférieur (en se composant éventuellement avec les attributs présents).



- figure IV.2 : Association d'attributs élémentaires aux éléments de la structure -

2.3 Les processus de base

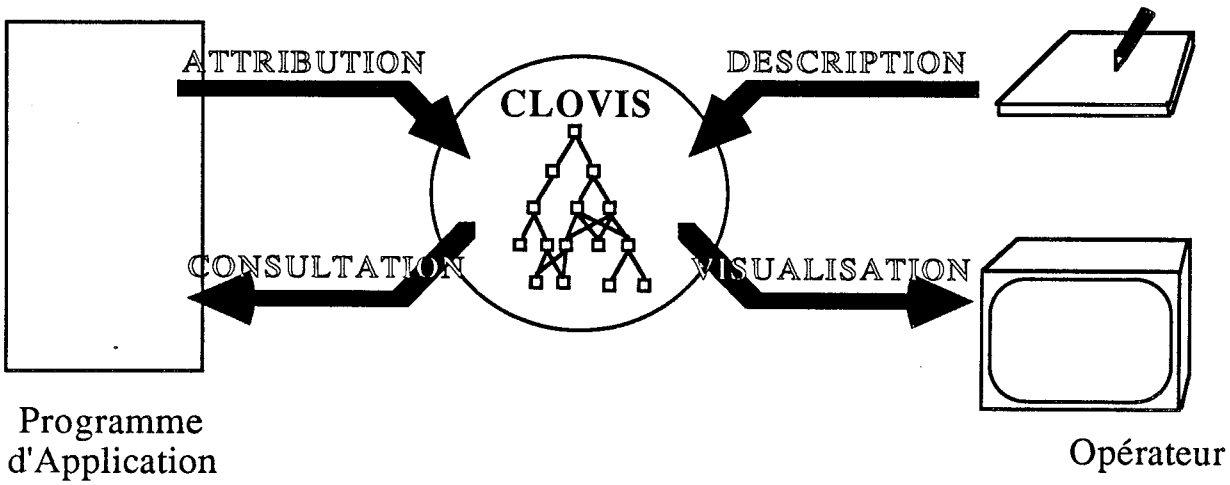
Finalement CLOVIS propose un ensemble de primitives de haut niveau prenant en charge l'exploitation de la base de données graphique au travers des quatre processus de base de la synthèse d'image (fig. IV.3) :

- l'**ATTRIBUTION** qui permet à partir du programme d'application de construire et d'associer des attributs graphiques élémentaires aux différents éléments (noeuds) de la structure hiérarchique,

- la **CONSULTATION** qui permet au programme d'application de récupérer la valeur des attributs attachés à un élément quelconque de la base de données hiérarchique,

- la **DESCRIPTION** qui pour chaque type d'attribut fournit des processus interactifs standards de construction utilisant les dispositifs de saisie tels qu'une tablette à numériser, un réticule ou un clavier,

- la **VISUALISATION** qui permet d'afficher sur l'écran selon un processus donné le ou les objets résultant de la synthèse des attributs graphiques associés à un ou plusieurs noeuds hiérarchiquement dépendants. L'affichage d'un élément s'effectue ainsi en parcourant sa structure hiérarchique et en combinant les attributs élémentaires rencontrés. Les règles de composition des attributs peuvent varier selon le processus de visualisation.



- figure IV.3 : Les processus de base dans CLOVIS -

Les processus d'ATTRIBUTION, CONSULTATION, DESCRIPTION et VISUALISATION sont entièrement pris en charge par CLOVIS, les programmeurs d'application peuvent les invoquer directement par l'appel d'une simple primitive. Ainsi les développeurs d'applications n'ont plus à construire eux mêmes les processus à partir d'opérateurs de bas niveau. Par ailleurs, il est à noter que ces quatre processus de base s'appliquent à n'importe quels éléments de la base de données hiérarchique, indépendamment de leur niveau, assurant ainsi une **cohérence** totale du système.

2.4 Les avantages de la structuration hiérarchique dans CLOVIS

Comme nous l'avons déjà vu au chapître précédent, les propriétés de la structuration hiérarchique des données offrent de nombreux avantages dans le contexte graphique et permettent d'alléger et simplifier certains processus. Parmi les bénéfices qu'apporte l'utilisation d'une base de données graphique hiérarchique dans CLOVIS, nous pouvons citer :

- la **puissance** qu'elle offre au programme d'application pour la description des objets qu'il manipule. Cette puissance est accrue par la non limitation du nombre de niveaux de structuration et par la possibilité de réutiliser et partager des structures déjà existantes (instances),

- la facilité d'association des attributs graphiques aux objets grâce à l'**héritage** et la **composition** des attributs dans la structure hiérarchique,

- la **cohérence** du système, toutes les opérations pouvant être appliquées indistinctement à n'importe quel niveau,

- la **puissance d'interaction**, la structure graphiques hiérarchique étant entièrement **dynamique**, et pouvant être modifiée à tout moment sous le contrôle du programme d'application.

- la possiblité de **limiter** la portée des opérations appliquées aux structures de données graphiques à des sous arborescences (contextes), protogeant ainsi aisément les données graphiques hors du contexte des effets de bord classiques,

- l'**optimisation** des processus de visualisation et de description (coupage, élimination des parties cachées...)

2.5 L'implémentation de CLOVIS

CLOVIS a été implémenté sous la forme d'une bibliothèque de sous programmes écrits en langage PASCAL et s'exécutant sur un VAX/11/780 sous le système d'exploitation VMS. Il a pu cependant être porté sans difficultés majeures sur une SM90 avec un système "UNIX-like" (SMIX).

Lors de la réalisation de cette première version du système, plusieurs terminaux de visualisation ont été pris en compte, allant de terminaux bas de gamme pour le dessin au trait (terminaux compatible Tektronix 4010) [GeGr 85] au terminal de synthèse de synthèse d'images HELIOS conçu au laboratoire ARTEMIS [Ferr 81],[Boul 85].

Dans les paragraphes qui suivent nous présentons l'ensemble des primitives proposées aux programmes d'application par CLOVIS. Celles-ci sont regroupées selon quatre catégories :

- les **primitives de structuration** destinées à la construction et à l'accès des éléments de la base de données hiérarchique (paragraphe **3**),

- les primitives **de gestion des attributs élémentaires** qui permettent au programme d'application de définir et de manipuler facilement les descripteurs d'attributs. Leur étude dans le paragraphe **4** permettra de présenter l'ensemble des attributs graphiques élémentaires qu'il est possible d'associer aux différents éléments de la structure hiérarchique. Nous insisterons tout particulièrement sur la manière dont ces attributs se composent dans l'arborescence lors de son parcours pour une visualisation,

- les primitives assurant la **gestion** des quatres grands **processus de base** que sont l'attribution, la consultation, la description et la visualisation (paragraphe **5**),

- les **primitives utilitaires** qui permettent en particulier l'archivage de structures hiérarchiques et la gestion des erreurs lors de l'utilisation de CLOVIS (paragraphe **6**).

3 LA STRUCTURATION HIERARCHIQUE DES DONNEES GRAPHIQUES DANS CLOVIS

Dans les paragraphes qui suivent nous présentons l'ensemble des primitives de haut niveau offertes par CLOVIS pour la gestion dynamique de la structure de données graphiques hiérarchique. Ces primitives permettent **à tout moment** l'accès (la référence), la création, la destruction d'une sous-arborescence **à un niveau quelconque** de la structure.

3.1 Mécanisme de dénomination des éléments de la structure hiérarchique

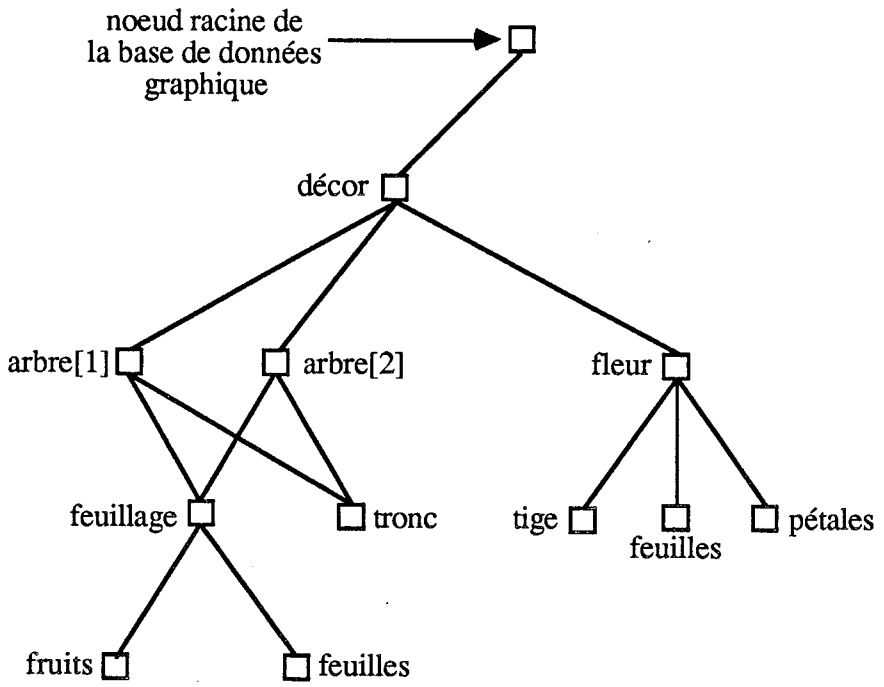
Lors de la conception du logiciel CLOVIS le problème de l'accès aux différents éléments de la structure de données graphiques s'est posé. En effet, afin de pouvoir être reconnus les différents noeuds de la structure arborescente doivent pouvoir être **identifiés** à l'aide de noms codés. A partir de ces noms, il est alors également possible d'exprimer la structure par l'emploi de règles syntaxiques permettant de combiner les identificateurs d'éléments de plusieurs niveaux.

3.1.1 Choix d'un codage pour l'identification des éléments

Il s'est avéré que l'emploi d'un **codage numérique** (comme dans les logiciels GRIGRI ou GKS avec les numéros de segments) n'offre pas la puissance de dénomination nécessaire pour la manipulation de la "base de données" hiérarchique de CLOVIS. En effet si cette solution est parfaitement adaptée aux systèmes à un ou deux niveaux de structuration et permet un décodage très rapide, elle manque de lisibilité et limite les facilités offertes par l'utilisation de règles syntaxiques pour des systèmes proposant plus de deux niveaux de structuration.

Aussi, avons nous opté pour un **codage alphanumérique** des identificateurs de noeuds. Cette seconde solution, bien qu'elle nécessite une place beaucoup plus importante et une analyse syntaxique plus coûteuse procure une meilleure lisibilité (les éléments peuvent ainsi avoir des noms significatifs par rapport aux objets qu'ils décrivent, qu'il est alors beaucoup plus facile de mémoriser). De plus elle offre une grande souplesse d'emploi sans limiter le nombre de niveaux autorisés.

Ainsi, chaque élément de la structure hiérarchique possède un identificateur qui est une chaîne alphanumérique de vingt caractères maximum. Un noeud doit avoir un nom unique à son niveau ce qui permet de le distinguer sans ambiguité parmi ses "frères". Par contre, des noeuds situés à des profondeurs ou dans des branches différentes peuvent partager le même nom. Dans le cas de noeuds qui permettent de partager des sous structures identiques (noeuds instances), ceux-ci partagent tous le même identificateur mais sont différenciés à l'aide d'**indices**.



- figure IV.4 : exemples d'identificateurs de noeuds -

3.1.2 Les expressions de création et d'accès

La désignation des éléments d'une structure s'effectue de manière classique à l'aide d'expressions parenthésées (définissant en fait un chemin dans l'arborescence) qui regroupent les noms des noeuds des différents niveaux parcourus. Le logiciel CLOVIS considère différents types d'expressions selon qu'il s'agit d'exprimer la **création** d'une structure ou la **référence** (l'accès) à une structure existante.

3.1.2.1 Syntaxe générale

Nous présentons dans ce qui suit le détail de la syntaxe régissant les différents types d'expression possibles (selon qu'il s'agit d'exprimer la création d'une structure arborescente ou la référence à une structure existante). Pour ce faire nous employons une forme grammaticale inspirée de la notation BNF et dont les règles sont les suivantes :

- tout symbole entre " et " est un symbole terminal.
- tout symbole entre < et > est un symbole non terminal.
- les expressions entre { et } sont optionnelles.
- le symbole | signifie une alternative.
- le symbole * indique 0 ou n occurences.

3.1.3.2 Expressions de création

Les expressions de ce type sont utilisées pour la création d'une structure hiérarchique, elle permettent de définir l'identité et la structuration des éléments qui la composent.

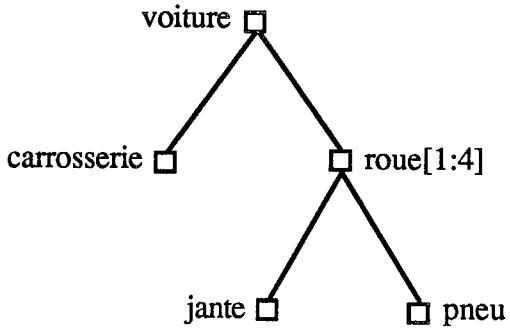
La syntaxe des expressions de création est la suivante :

< expression de création > ::= < identif_noeud > {"(" < sous-structures > ")"}
< sous-structures > ::= < structure > {"," < structure >}*
<structure> ::= < identif_noeud > { "(" < sous-structures > ")" }
< identif_noeud > ::= < identificateur > { "[" < indexes > "]" }
< indexes > ::= { < borne > } ":" { < borne > }
< borne > ::= { <signe> } < entier non signé >
< signe > ::= "+" | "-"
< entier non signé > ::= < chiffre > { <chiffre> }*
< identificateur > ::= < lettre > { < lettre > | < chiffre > }*

Les valeurs possibles pour les bornes d'indice sont comprises dans l'intervalle [BORNEMIN, BORNEMAX] qui sont des constantes prédéfinies correspondant aux bornes minimum et maximum d'indexation dictées par la limitation de la capacité machine pour les entiers. L'ommission d'un indice équivaut à considérer la borne minimum ou maximum selon les cas.

L'utilisation de noms indicés à l'intérieur d'une expression de création a pour fonction la définition du nombre potentiel d'instances qui pourront être obtenues à partir d'un noeud. A la création de la structure **un seul** exemplaire de chaque noeud est effectivement créé. Ce n'est que par la suite (voir paragraphe 3.3.2.4) que celui-ci pourra être éventuellement éclaté en différentes instances par l'utilisation de références indicées. Si aucun indice n'est précisé le noeud est potentiellement référençable entre BORNEMIN et BORNEMAX sinon les bornes déclarées définissent le domaine indiçable.

A titre d'exemple l'expression **"voiture(carrosserie,roue[1..4](jante,pneu))"** permet de construire la structure schématisée par la figure IV.5 qui suit.



- figure IV.5 : création d'une structure -

3.1.3.3 Expressions de référence

Les expressions de référence permettent de désigner des éléments à l'intérieur de la base de données hiérarchique. Le chemin d'accès aux éléments peut être exprimé **relativement** à un noeud quelconque utilisé comme racine implicite lors de l'évaluation des expressions de référence et désigné sous le terme de noeud contexte. Le contexte par défaut est le noeud racine du fichier graphique, nous verrons au paragraphe 3.4.2.1 les primitives qui permettent à l'application de définir ses propres contextes.

Le système CLOVIS distingue deux types d'expressions de référence :

- les références **SIMPLES** qui permettent d'accéder à **un** et **un seul** élément de la structure.
- les références **MULTIPLES** qui permettent d'accéder simultanément à **plusieurs** éléments de la structure par l'expression de plusieurs chemins.

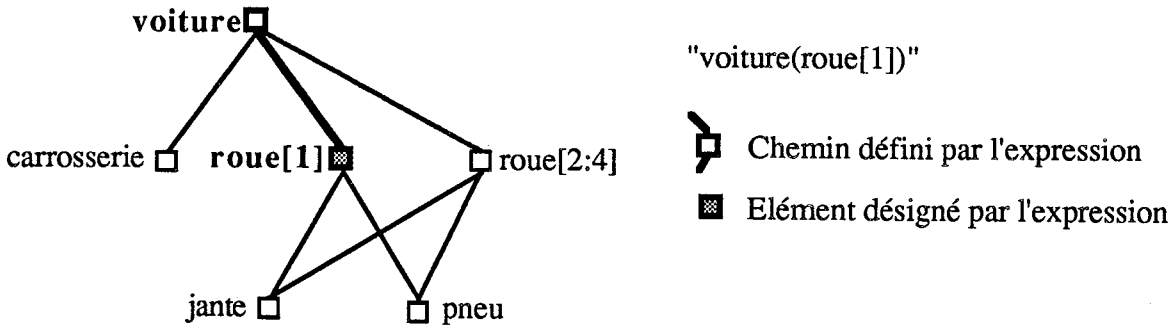
a) <u>références simples</u>

La syntaxe des expressions de référence simple est la suivante :

< référence simple > ::= < identificateur > {"[" < indexes > "]"} {"(" < référence simple > ")"}
< indexes > ::= { < borne > } ":" { < borne > }
< borne > ::= { <signe> } < entier non signé >
< signe > ::= "+" | "-"
< entier non signé > ::= < chiffre > { <chiffre> }*
< identificateur > ::= < lettre > { < lettre > | < chiffre > }*

L'absence d'une borne d'indiçage signifie que le considère l'élément existant avec le plus petit indice (respectivement le plus grand) pour la borne inférieure (respectivement supérieure).

La figure IV.6 donne un exemple de référence simple sur la structure précédemment créée. On remarquera en particulier l'utilisation de la référence indicée à la roue numéro 1 qui provoque "l'éclatement" en deux de la structure à ce niveau.



- figure IV.6 : Expression de référence simple -

b) références multiples

La syntaxe des expressions de référence multiple est la suivante :

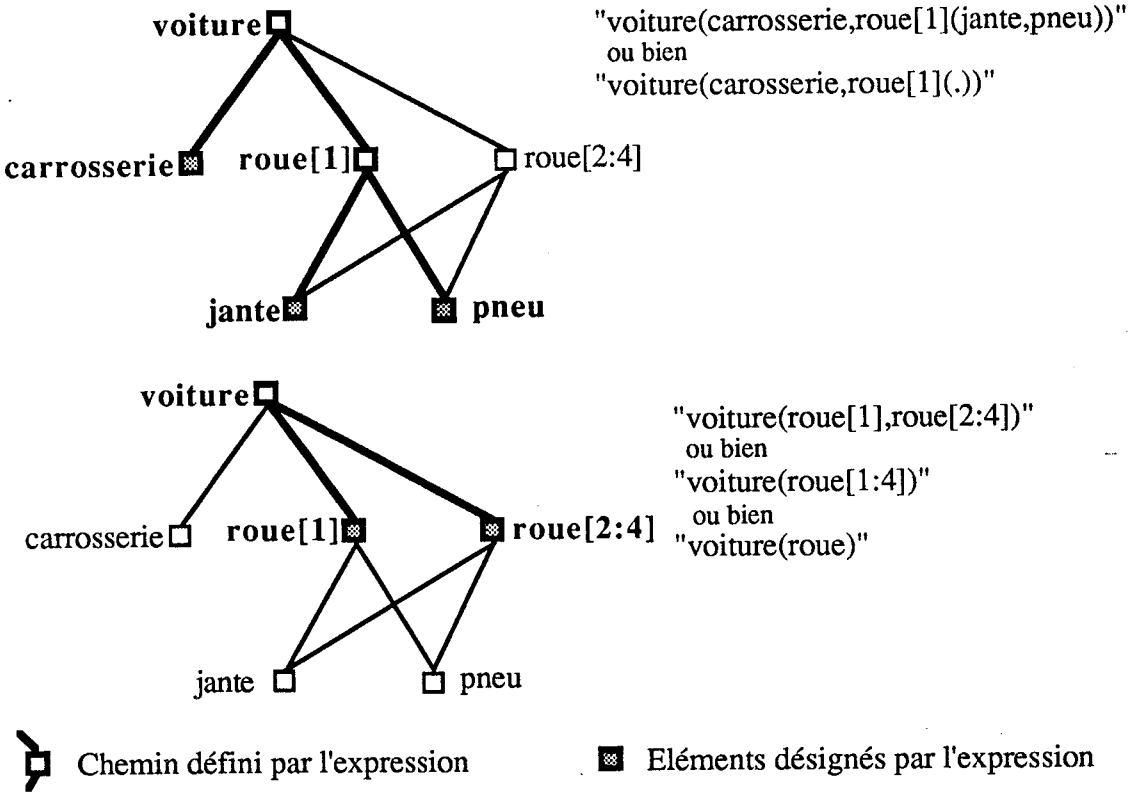
< expression de référence > ::= < référence simple> | < référence multiple >
< référence multiple > ::= '>' | '.' | < branche > { ',' < branche > }*
< branche > ::= < identif_noeud > { "(" < référence multiple > ")" }
< identif_noeud> ::= < identificateur > { "[" < indexes > "]" }
< indexes > ::= { < borne > } ":" { < borne > }
< borne > ::= { <signe> } < entier non signé >
< signe > ::= "+" | "-"
< entier non signé > ::= < chiffre > { <chiffre> }*
< identificateur > ::= < lettre > { < lettre > | < chiffre > }*

où :

> : représente tout le sous arbre issu du noeud de niveau supérieur et référence donc les feuilles de l'arborescence située sous ce noeud.

. : représente tous les noeuds d'un même niveau.

On trouvera dans la figure IV.7 ci dessous, des exemples d'expressions de référence multiple appliquées à la structure déjà utilisée dans les exemples qui précédent. Il est à noter les différentes utilisations possibles des indices pour l'accès à l'ensemble des roues.



- figure IV.7 : Expressions de référence multiple -

3.2 Primitives de construction de structures hiérarchiques

Nous présentons ici les primitives qui permettent de construire effectivement la structure hiérarchique des objets à l'intérieur de la "base de données" graphiques (création, copie, retrait, destruction de structures...).

3.2.1 Création d'une structure hiérarchique

A tout moment le programme d'application peut définir une structure hiérarchique par l'invocation de la fonction primitive

refrac <-- STRUCTURE(chaîne)

où :

- *chaîne* est une chaîne alphanumérique de type expression de création qui décrit la structure et l'identité des éléments de la structure hiérarchique créée,

- *refrac* est l'adresse du noeud racine de la structure créée.

remarque : *refrac* est un pointeur sur un élément de type noeud. Les différents types considérés par CLOVIS et nécessaires au programme d'application sont définis dans un fichier (TYPES_CLOVIS) que le programmeur peut inclure (directive de compilation "include") dans son (ses) programme(s). Ainsi par exemple, les variables destinées à recevoir l'adresse d'un noeud de la structure hiérarchique sont déclarées avec le type prédéfini *ptr_noeud* (pointeur sur un noeud). De la même manière, le fichier inclus CSTES_CLOVIS contient les différentes constantes utilisées par CLOVIS (par exemple les constantes pour désigner les classes et types d'attributs élémentaires).

La structure hiérarchique ainsi construite n'est rattachée à aucun élément de la "base de données" graphique. Nous désignerons de telles structures sous le terme d'**objet libre**. L'insertion de la structure dans la "base de données" graphique doit être effectuée explicitement à l'aide de l'une des primitives que nous présentons dans les paragraphes qui suivent et est donc sous la responsabilité du programme d'application. Cette séparation de la création de structures et de leur rattachement aux éléments déjà présent dans la "base de données" graphique, permet à l'application de manipuler ses propres structures et de les réutiliser selon ses besoins.

Lors de l'invocation de la primitive STRUCTURE, le logiciel CLOVIS effectue une analyse syntaxique de l'expression de création fournie en paramètre et vérifie sa validité. En cas d'erreur, *refrac* est affectée à NIL et le système signale l'erreur rencontrée. (voir traitement des erreurs).

3.2.2 Insertion d'une structure dans la base de données graphique

Une structure hiérarchique précédemment définie à l'aide de la primitive STRUCTURE peut être à tout moment insérée dans la "base de données" graphique à la demande du programme d'application. L'ajout d'une nouvelle structure à la "base de données" graphique consiste à son rattachement à un ou plusieurs éléments présents dans cette dernière. Le rattachement d'une structure s'effectue en insérant le noeud racine de celle-ci parmi les fils d'un ensemble déterminé de noeuds de la base de données hiérarchique. Cet ensemble de noeuds concernés par les opérations de maniupulation de structure est désigné sous le terme d'**entité de travail**. L'entité par défaut correspond au noeud racine du fichier graphique qui est toujours présent. Le programme d'application peut définir ses propres entités de travail à l'aide d'expressions de référence simples ou multiples (voir paragraphe 3.3.2).

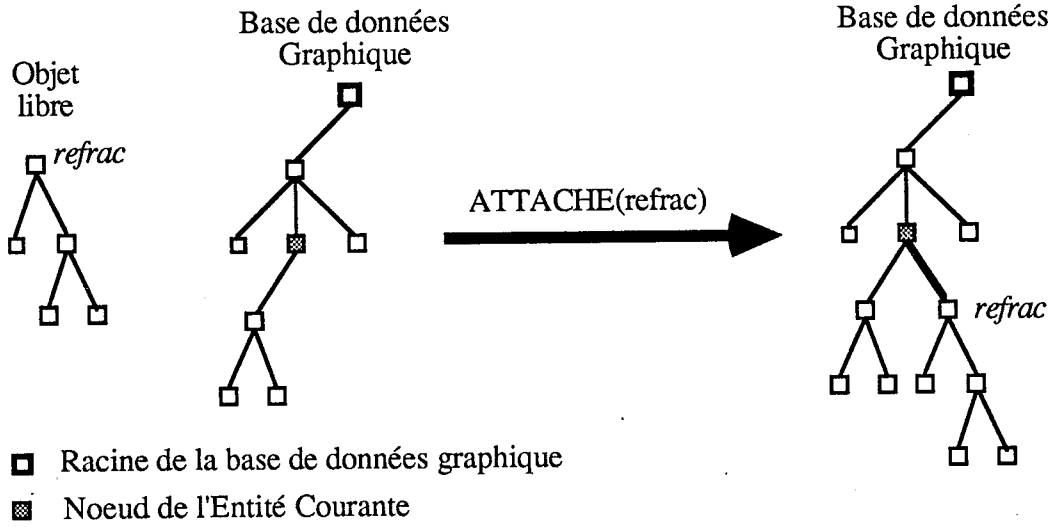
Plusieurs possibilités sont offertes pour l'ajout de structures au fichier graphique qui peut s'effectuer sous la forme d'un "attachement" proprement dit, d'instanciation ou de copie.

3.3.2.1 Rattachement d'un objet libre à l'entité courante

Une structure définie à l'aide de la fonction STRUCTURE, vue précédemment, peut être rattachée fichier graphique en utilisant la primitive

ATTACHE(refrac)

qui insère en fin de la liste des fils du noeud défini par l'entité courante le noeud d'adresse *refrac* racine de la structure à "accrocher" (fig IV.8).



- figure IV.8 : Attachement d'une structure au fichier graphique -

Avant de procéder à l'attachement effectif de la structure *refrac* au fichier graphique, un certain nombre de contrôles sont effectués :

- la structure ne peut être rattachée qu'à un seul élément à la fois, l'entité de travail courante doit donc être une référence simple et ne désigner qu'un seul noeud,

- *refrac* doit désigner un objet libre, c'est à dire non rattaché au fichier graphique,

- la structure *refrac* ne peut être insérée parmi les fils du noeud de l'entité de travail que si son identificateur (nom + bornes d'indices) n'entre pas en conflit avec les identificateurs des noeuds déjà existant à ce niveau. Ainsi, CLOVIS vérifie qu'il n'existe pas parmi les fils du noeud de l'entité de travail des éléments de même nom que *refrac* et dont les bornes d'indices recoupent celles du noeud *refrac* .

Si l'une de ces conditions n'est pas respectée, la primitive ATTACHE n'a pas d'effet et un message d'erreur le signale à l'opérateur.

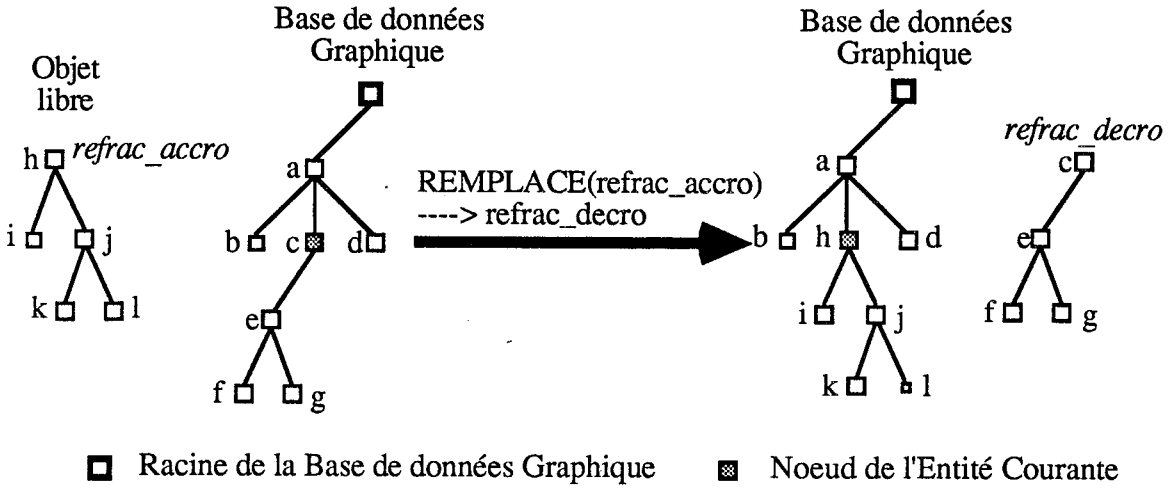
3.2.2.2 Remplacement d'une structure

La primitive

REMPLACE(refrac_accro) ----> refrac_decro

permet de remplacer une structure du fichier graphique par une nouvelle structure. La structure à remplacer est définie par l'entité courante (qui doit donc être une référence simple) et est décrochée du fichier graphique. L'adresse de sa racine, *refrac_decro* , est retournée par la

fonction REMPLACE. *refrac_accro* est l'adresse de la racine de la structure substituée à la structure décrochée.



- figure IV.9 : remplacement d'une structure -

Comme pour la primitive ATTACHE présentée dans le paragraphe précédant, des contrôle sont effectués afin de vérifier que le remplacement de la structure ne provoque pas de conflits d'identificateurs dans le fichier graphique.

3.3.2.3 Création d'une instance de structure

Comme nous l'avons déjà évoqué au § 2.1, il est possible dans CLOVIS de partager une même structure entre plusieurs éléments de la "base de données" hiérarchique, ce que nous désignons sous le terme d'instance.

La définition d'une instance d'une structure s'effectue par l'appel de la primitive

INSTANCE(refrac, bornesup, borneinf) ---> ref_inst

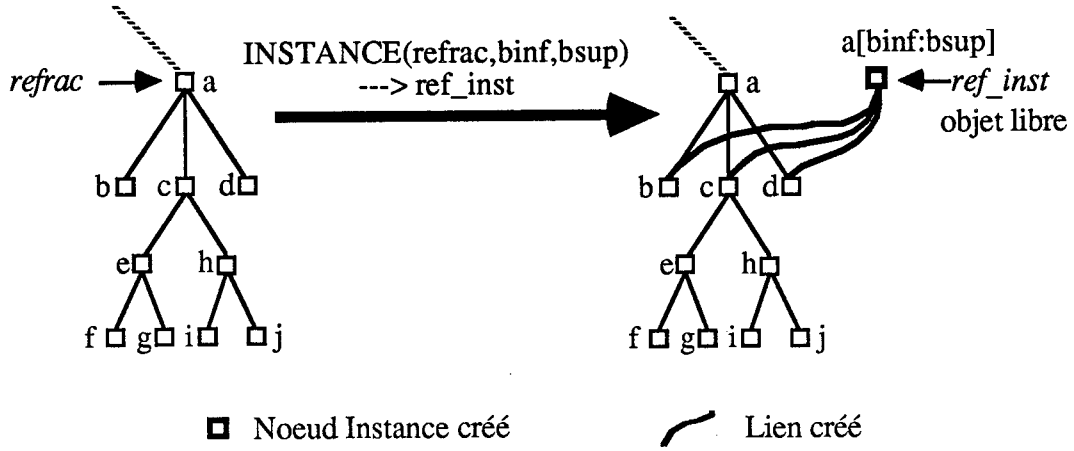
où :

- *refrac* est la référence de la racine de la structure à instancier,

- *bornemin* et *bornemax* sont des entiers qui définissent l'intervalle d'indexation autorisé lors de références indicées ultérieures à l'instance,

- *ref_inst* est l'adresse du noeud instance créé qui permet le partage de la sous structure de *refrac*.

Lors de la définition d'une instance (voir figure IV.10 p. 139), tout se passe pour l'utilisateur comme si l'on créait une copie du noeud racine original (référencé par *refrac*). Cette copie partage les mêmes sous-structures et conserve le même nom que le noeud original, mais permet par contre de particulariser l'instance à l'aide d'attributs qui lui sont propres. Le noeud instance créé est **un objet libre**, l'application doit donc l'affecter explicitement à l'un des éléments de la base de données graphiques à l'aide de l'une des primitives ATTACHE ou REMPLACE vues dans les paragraphes précédents.



- figure IV.10 : Création d'une instance de structure -

Remarque : Le noeud *refrac* passé en paramètre de la primitive INSTANCE peut être absolument quelconque. Il peut s'agir d'un objet libre (non rattaché au fichier graphique) ou non, d'un noeud instance ou non.

3.2.2.4 Création implicite d'instances - références indicées

Il existe un second mécanisme qui permet, par l'intermédiaire des noms indicés, la création aisée d'instances multiples d'un noeud à un même niveau de structure du fichier graphique.

Comme nous l'avons vu au paragraphe 3.1.3.3, une référence indicée à un noeud provoque implicitement "l'éclatement" de la structure en différentes instances si les bornes d'indices du ou des noeuds de même nom ne correspondent pas aux bornes de la référence. Nous donnons dans ce qui suit les règles détaillées pour la création d'instance à partir de références indicées.

Une référence indicée avec un intervalle d'indexation $[r_1,r_2]$ entraine "l'éclatement" d'un noeud de la structure existante si l'intervalle d'indexation $[b_1,b_2]$ de ce noeud est tel que :

$$[b_1,b_2] \cap [r_1,r_2] \neq \varnothing \quad \text{et} \quad [b_1,b_2] \not\subseteq [r_1,r_2]$$

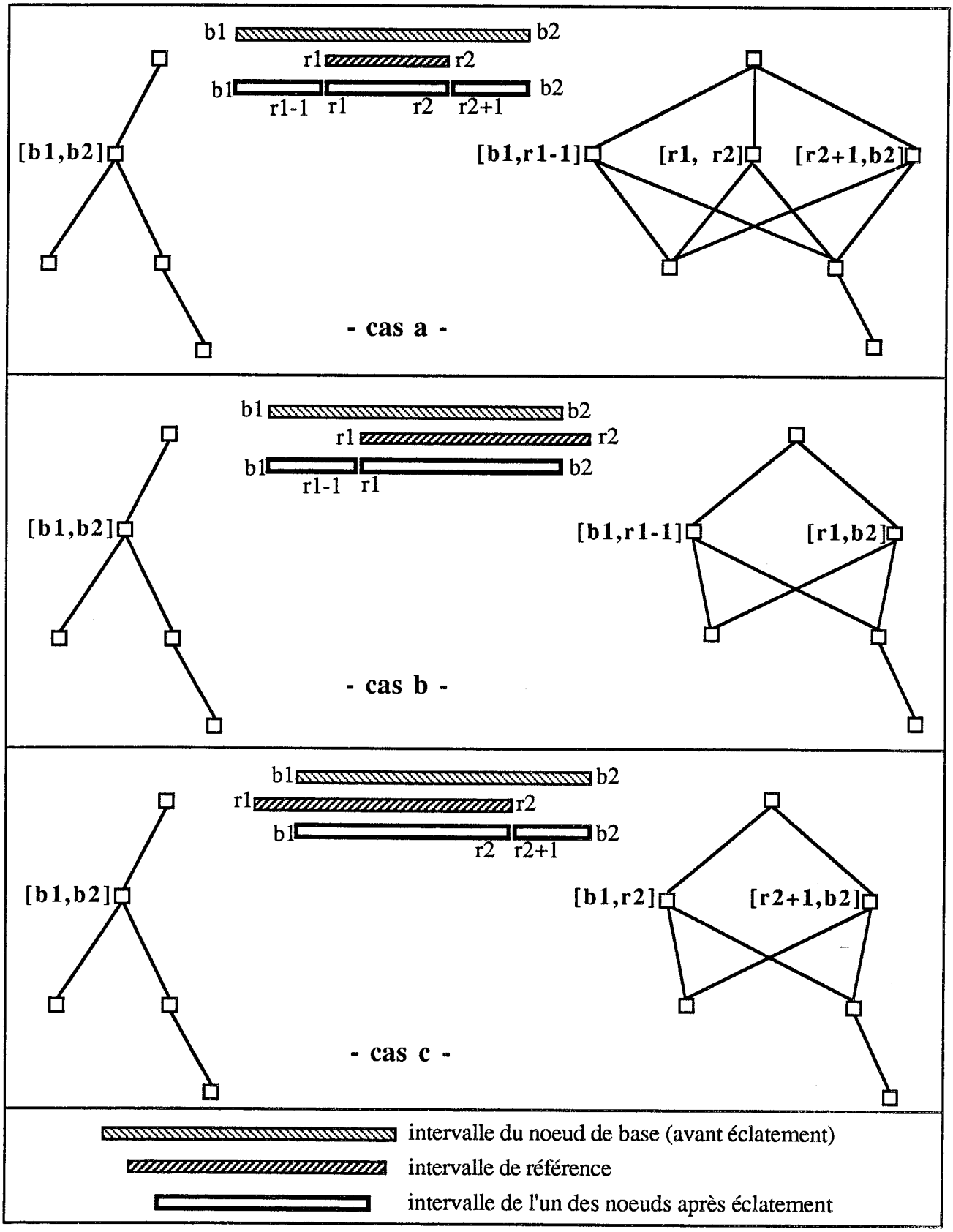
Trois cas sont alors possibles pour l'éclatement, schématisés par la figure IV.11, selon la valeur des bornes d'indices du (des) noeud(s) existant et de celles utilisées pour la référence :

a) si $b_1 < r_1$ et $b_2 > r_2$ alors le noeud $[\ b_1, b_2\ ]$ sera éclaté en trois noeuds d'intervalle d'indiçage $[\ b_1, r_1 - 1]$, $[\ r_1, r_2\ ]$ et $[\ r_2 + 1, b_2\ ]$

b) si $b_1 < r_1$ et $b_2 \leq r_2$ alors le noeud $[\ b_1, b_2\ ]$ sera éclaté en deux noeuds d'intervalle d'indiçage $[\ b_1, r_1 - 1]$ et $[\ r_1, b_2\ ]$

c) si $b_1 \geq r_1$ et $b_2 < r_2$ alors le noeud $[\ b_1, b_2\ ]$ sera éclaté en deux noeuds d'intervalle d'indiçage $[\ b_1, r_2\ ]$ et $[\ r_2 + 1, b_2\ ]$

Cette possibilité d'éclatement implicite de la structure en plusieurs instances, décharge et simplifie le programme d'application pour la définition d'instance multiples à un même niveau. Elle évite ainsi la création successive de noeuds instances par la primitive INSTANCE puis leur affectation répétitive à la base de donnée graphique (primitives ATTACHE ou REMPLACE).



- figure IV.11 : les différents cas d'éclatement de la structure CLOVIS -

3.2.2.5 Copie d'une structure

La fonction primitive

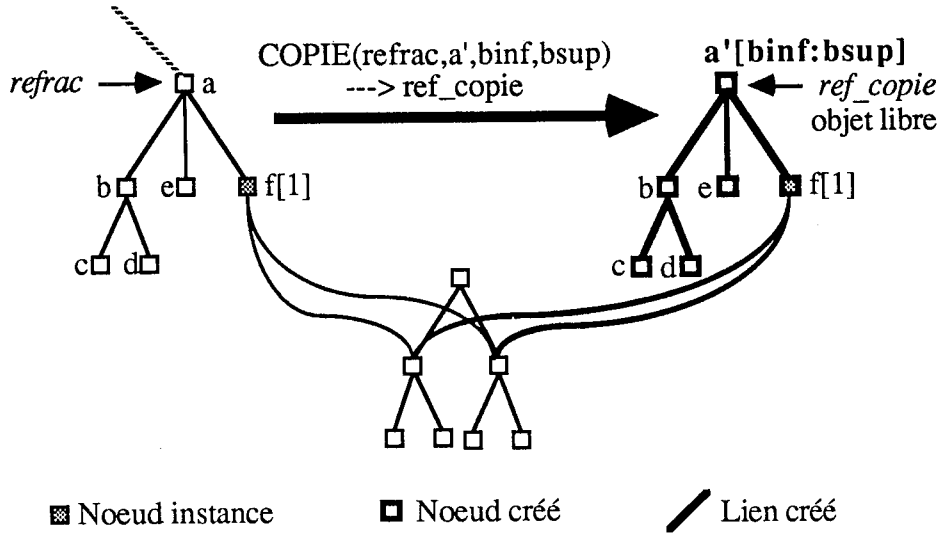
COPIE(refrac,identif,binf,bsup) ----> ref_copie

permet de créer la copie complète d'une structure. Dans ce cas, tous les noeuds de la structure dont la racine est référencée par *refrac* sont effectivement dupliqués.

ref_copie est l'adresse de la nouvelle structure ainsi créée, et pourra être réutilisé pour son rattachement ultérieur à la "base de données" graphique (*ref_copie* est un objet libre),

Identif permet de donner un nom à la structure dupliquée, et *binf* et *bsup* définissent ses nouvelles bornes d'indexation.

refrac peut être un objet libre ou non, mais ne doit pas être un noeud instance.



- figure IV.12 : Copie d'une structure -

Remarque : les noeuds de la sous structure de *refrac* dupliqués conservent le même identificateur. Si parmi ceux-ci se trouvent des noeuds instance, on ne duplique évidemment pas la sous-structure qu'ils partagent : par exemple le noeud f[1] sur la figure IV.12.

3.2.2.6 Récapitulatif - Comparaison instanciation / copie

Pour récapituler, il est important de bien souligner les différences qui séparent la création d'instances de la copie d'une structure.

La notion d'instance permet de partager des structures identiques, tout en les distinguant en affectant à leur racine des attributs distincts. Il faut cependant bien souligner que le partage de ces structures implique que la modification d'une sous structure de n'importe quelle instance (aussi bien au niveau de la structure que des attributs) se répercutent dans toutes les instances.

La copie permet elle, au programme d'application de définir facilement et rapidement des structures initialement identiques qui au contraire de l'instanciation peuvent évoluer indépendamment par la suite. Ainsi, la modification d'une structure dupliquée n'interagit en aucune manière sur les autres exemplaires de cette structure utilisés par ailleurs dans la base de données hiérarchique.

3.2.3 Suppression de Structure

La suppression de structure s'effectue de manière symétrique à la construction du fichier graphique. Ainsi, elle peut consister soit en un simple retrait d'une structure du fichier graphique, soit en la destruction complète d'une arborescence.

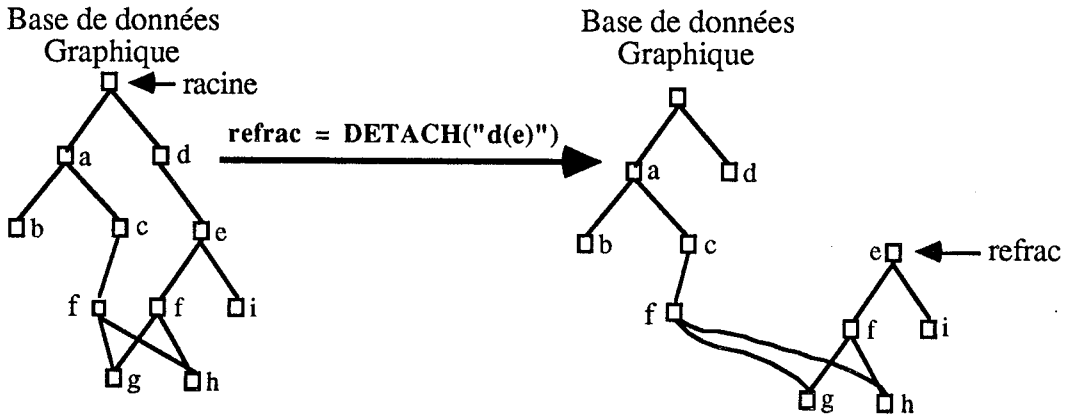
3.2.3.1 Retrait d'une structure de la base de données graphique

De la même façon qu'il est possible de rattacher une structure à la base de données graphique à l'aide de la primitive ATTACHE, on peut **retirer** (détacher) une structure de celle-ci à l'aide de la fonction

DETACH(<référence simple>) ---> refrac

Cette primitive "décroche" de la base de données graphique la structure dont la racine est désignée par la référence simple passée en paramétre et renvoie son adresse.

L'effet de cette primitive est simplement la suppression du fichier graphique du lien à la structure *refrac* comme nous le montre la figure IV.13, la structure détachée a oujours une existence et elle pourra être réutilisée par la suite (par exemple à l'aide de la primitive INSTANCE)



- figure IV.13 : retrait d'une structure de la base de données graphique -

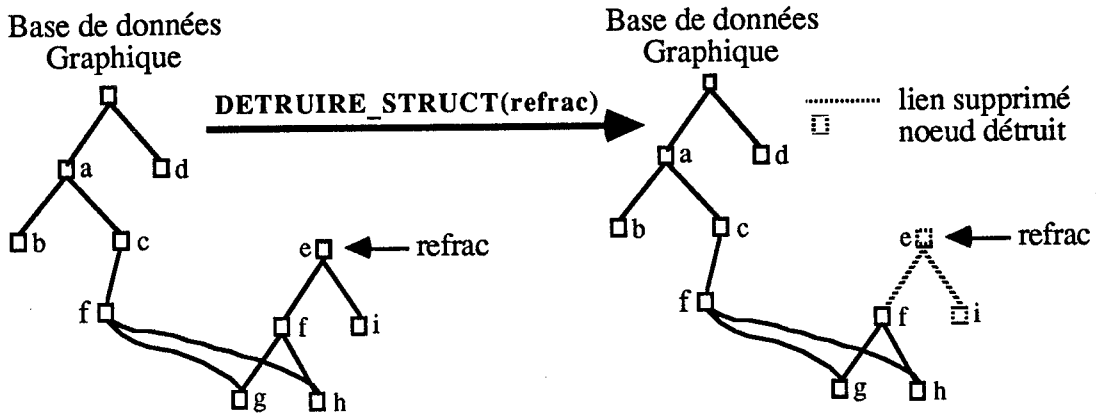
remarque : suite à la primitive DETACH, l'entité de travail courante est invalidée et repositionnée sur l'entité par défaut correspondant au noeud contexte.

3.2.3.2 Destruction d'une structure

La destruction d'une structure consiste à la suppression effective d'une structure hiérarchique, c'est à dire à la libération de l'espace mémoire qui lui est alloué. Ceci s'effectue par l'intermédiaire de la primitive

DETRUIRE_STRUCT(refrac)

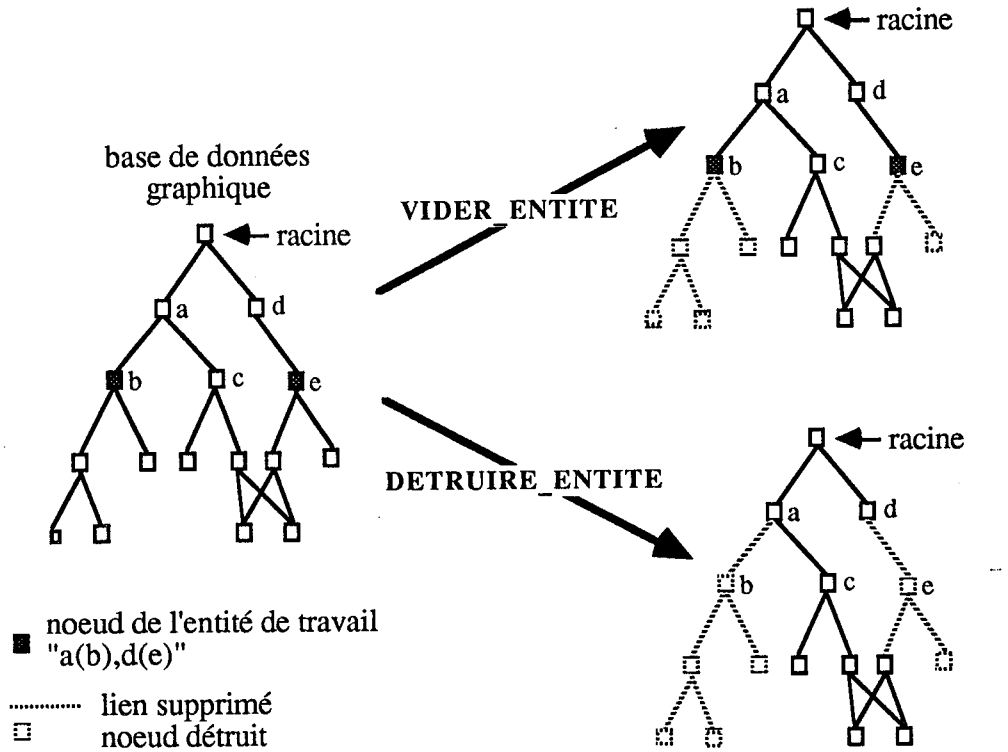
où *refrac* est la référence de la racine de la structure à supprimer. Celle-ci ne doit pas être rattachée au fichier graphique (directement sous forme de sous-objet ou indirectement sous forme d'instance). Pour les sous-structures issues de *refrac* , dans le cas d'instances seuls les liens sont supprimés, alors que dans le cas de sous-structures non partagées (sous-objets) les noeuds sont effectivement détruits. La figure ci dessous nous montre l'effet de cette primitive sur la structure décrochée dans l'exemple précédent.



- figure IV.14 : destruction d'une structure -

3.2.3.3 Primitives de suppression liées à l'entité de travail courante

A l'aide des primitives précédentes la destruction ne peut s'effectuer que sur une seule structure à la fois (passage par la racine de la structure). Cela peut s'avérer "pénalisant" lorsque l'on désire supprimer des éléments situés dans des branches différentes du fichier graphique. Pour cela nous avons introduit deux autres primitives (fig. IV.15) qui portent sur les structures désignées par l'entité de travail courante (qui peut être définie par une référence multiple).



- figure IV.15 : supression de l'entité de travail courante -

La primitive **VIDER_ENTITE** décroche et supprime toutes les sous structures situées sous les noeuds définis par l'entité courante.

La primitive **DETRUIRE_ENTITE** s'applique en plus aux noeuds de l'entité courante. Suite à l'application de cette primitive, l'entité, c'est à dire l'ensemble des noeuds sur laquelle elle a porté, n'a plus d'existence et de signification et est donc remplacée par l'entité de travail

par défaut (le noeud contexte). Aussi si l'entité de travail coincide avec le noeud contexte, il est impossible d'exécuter la primitive DETRUIRE_IDENTITE (le noeud contexte est inaltérable) et son invocation provoque un message d'erreur.

remarque : comme pour DETRUIRE_STRUCT les noeuds concernés par ces deux primitives ne sont effectivement détruits que s'ils ne sont pas partagés par d'autres structures, sinon seuls les liens sont supprimés.

3.3 Primitives d'accès à la structure

3.3.1 Séparation entre l'entité de travail et l'entité de visualisation

Les primitives d'accès à la structure que nous présentons ici, permettent de désigner à l'aide d'expressions de référence les éléments de la base de données hiérarchique qui sont manipulés par les processus d'ATTRIBUTION, CONSULTATION, DESCRIPTION et VISUALISATION.

Cependant à un instant donné, l'ensemble des noeuds (entité) de la structure arborescente concernés par les divers processus de base, varie selon la nature du processus. Ainsi les éléments considérés par les processus d'attribution ne coïncident pas en général avec ceux manipulés par le processus de visualisation (il s'agit bien souvent d'un sous ensemble de ceux-ci).

C'est pourquoi, afin d'éviter à l'utilisateur de redéfinir sans cesse une nouvelle entité avant l'invocation de tout nouveau processus nous avons jugé utile de distinguer deux types d'entites définies et manipulées séparément :

- **l'entité de "travail"** qui défini l'ensemble des éléments qui sont concernés par les processus d'ATTRIBUTION, CONSULTATION et DESCRIPTION, et comme nous l'avons déjà vu au paragraphe 3.2 par les primitives de construction de la base de données hiérarchique,

- **l'entité de "visualisation"** qui défini de **manière indépendante** l'ensemble des éléments qui seront pris en compte par le processus de visualisation.

3.3.2 Primitives pour la définition de l'entité de travail

3.3.2.1 Définition d'un contexte de travail

Le logiciel CLOVIS permet au programme d'application de désigner un noeud quelconque de la structure arborescente comme étant la racine implicite ou **contexte** pour les opérations (autres que la visualisation) effectuées ultérieurement sur la base de donnée graphique.

L'intérêt de cette notion de contexte est multiple, en effet elle permet :

- de réduire les chaînes de caractères identifiant les noeuds de l'arborescence, les expressions de référence ultérieures seront relatives au noeud contexte.

- d'assurer une certaine protection des données graphiques en limitant le champ d'application des actions à la partie de la structure située sous le noeud contexte pendant la durée de celui-ci.

Le contexte de travail par défaut correspond au noeud racine de l'arborescence. La définition d'un contexte de travail s'effectue par l'appel de la primitive

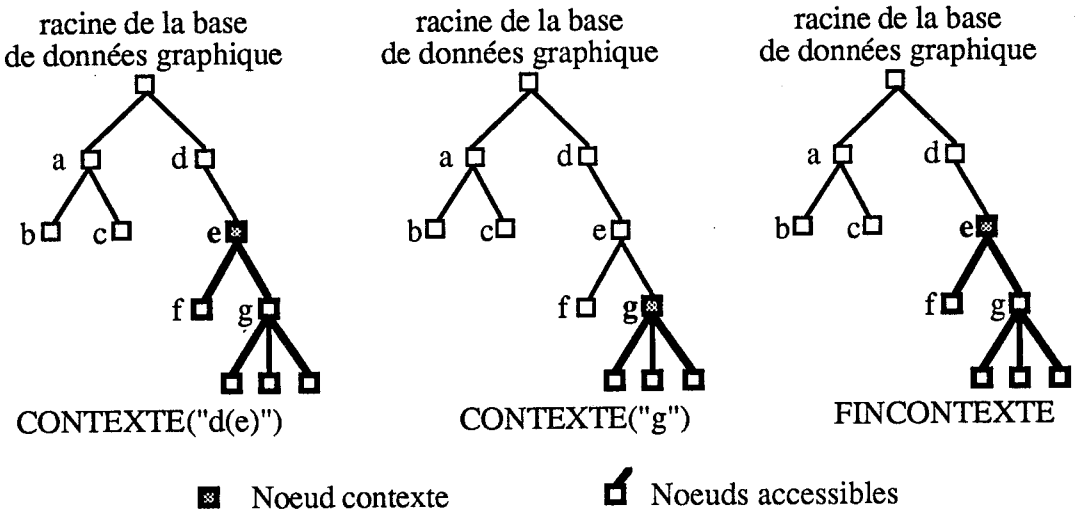
CONTEXTE_TRAV(référence)

où *référence* est une expression alphanumérique de type référence simple, elle désigne **le** noeud

qui devient la nouvelle racine de travail implicite.

Il est possible de définir de manière successive plusieurs contextes de travail qui sont "imbriqués". En effet la définition d'un nouveau contexte se fait relativement au contexte courant (l'expression de référence simple et relative). Les contextes successifs sont conservés (empilés), le retour au contexte de travail précédant (dépilement) s'effectuant par l'appel de la primitive

FINCONTEXTE_TRAV.



- figure IV.16 : définition et imbrication des contextes de travail -

L'imbrication des contextes est à rapprocher à la notion de bloc employée dans les langages de programmation structurés de style Pascal. Elle encourage une programmation structurée et modulaire en permettant de calquer la structure graphique sur celle du programme d'application.

Ainsi, un sous programme peut être astreint à un contexte donné. Cela permet en particulier, à l'application d'assurer un contrôle correct sur la portée des opérations qu'il effectue.

3.3.2.2 Définition de l'entité de travail

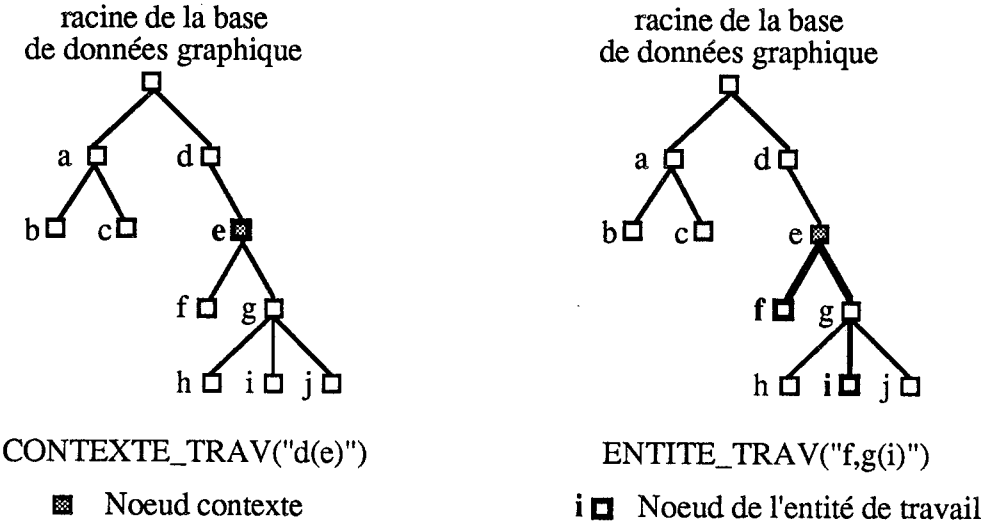
Comme nous l'avons déjà vu auparavant, l'entité de travail est l'ensemble des éléments de la base de données graphique qui sont concernés par les processus d'ATTRIBUTION (y compris de structure), de CONSULTATION et DESCRIPTION et par les primitives de construction de structure (attachement, création d'instances...).

Le programme d'application défini l'entité de travail à l'aide de la primitive

ENTITE_TRAV(chaîne)

où *chaîne* est une expression de référence simple qui désigne **relativement** au contexte courant l'ensemble des noeuds concernés. *chaîne* peut être indifféremment une expression de référence simple ou multiple.

remarque : il est à souligner que contextes et entité de travail sont des entités **séparées** qui existent **simultanément**. L'entité courante désigne explicitement les noeuds "effectivement actifs" parmi l'ensemble des noeuds "potentielement actifs" définis par le contexte.



- figure IV.17 : Définition d'une entité de travail -

3.3.2.3 Règles de définition des contextes et des entités de travail

- A l'initialisation de CLOVIS, l'entité de travail par défaut (de la même manière que le contexte par défaut) correspond au noeud racine de l'arborescence.

- A la définition d'un nouveau contexte, l'entité de travail prend pour valeur ce noeud contexte.

- Une seule entité de travail peut être définie à la fois. Aussi la définition d'une nouvelle entité remplace la précédente.

- La destruction de structure s'appliquant aux noeuds définis par l'entité courante, on se repositionne à la fin d'une telle opération sur l'entité par défaut (noeud contexte).

Remarque : L'impossibilité de définir et d'utiliser simultanément plusieurs entités de travail est due au fait que les opérations successives sur celles-ci peuvent altérer la structure du fichier graphique (c'est le cas, en particulier, de la destruction de structures). En effet cela soulève des problèmes de cohérence, rien ne garantissant alors que la structure correspondant à la définition d'une entité soit toujours la même durant toute la durée de vie de celle-ci.

3.3.3 Primitives pour la définition de l'entité la visualisation

3.3.3.1 Définition d'un contexte de visualisation

Pour respecter une certaine cohérence avec les primitives pour la définition de l'entité de "travail", il nous est apparu utile d'introduire la notion de contexte de visualisation. De manière analogue au contexte de travail, la spécification d'un contexte de visualisation permet de désigner un noeud quelconque de la structure hiérarchique comme étant la racine implicite pour les opérations ultérieures ayant trait à la visualisation.

La primitive

CONTEXTE_VISU(<référence simple>)

permet la définition d'un nouveau contexte de visualisation. Comme pour les contextes de travail, la définition d'un nouveau contexte de visualisation s'effectue relativement au contexte précédent. Les contextes de visualisation sont empilés et le retour au contexte précédent est réalisé par la primitive

FINCONTEXTE_VISU

Outre les avantages de simplification des expressions de référence pour la définition de l'entité de visualisation, et de "protection" en limitant à la sous arborescence située sous le noeud contexte les opérations de visualisation, l'utilisation de contextes de visualisation offre la possibilité d'évaluation des attributs. Ainsi lorsqu'un contexte de visualisation est défini, les attributs qui lui sont associés peuvent évalués (une fois pour toutes durant la "durée de vie" du contexte). Par la suite, lors de la visualisation d'une structure située sous le noeud contexte, ces attributs peuvent être directement utilisés, sans avoir à être réévalués (il n'est pas nécessaire de parcourir le chemin entre le noeud contexte et la racine de la structure hiérarchique).

3.3.3.2 Définition de l'entité de visualisation

La définition de l'entité de visualisation s'effectue par l'intermédiaire de la primitive

ENTITE_VISU (chaîne)

et respecte les mêmes règles que l'entité de travail :

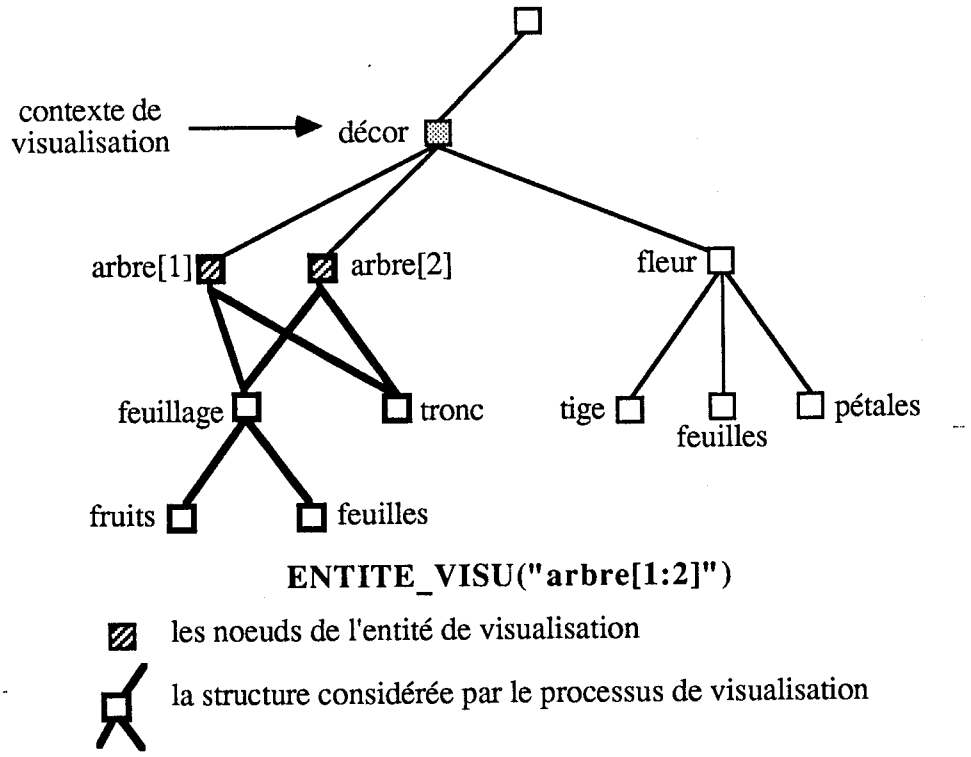
- *chaine* est une expression de référence (simple ou multiple) relative au contexte de visualisation courant,

- une seule entité peut définie à la fois, la définition d'une nouvelle entité remplaçant l'ancienne).

Les éléments concernés par le processus de visualisation sont (fig. IV. 18) :

- les éléments (noeuds) désigné par l'entité de visualisation courante,

- tous les éléments qui leur sont hiérarchiquement dépendants, c'est à dire les sous-structures rattachées aux noeuds de l'entité de visualisation courante.



- figure IV.18 : Définition d'une entité de visualisation -

remarque : les contextes est entités de travail et de visualisation existent simultanément et peuvent être définis de manière totalement indépendante.

3.3.3.3 Règles de définition des contextes et entités de visualisation

Les règles de définition de contexte et de l'entité de visualisation sont identiques à celles définies au paragraphe 3.3.2.3 pour les primitives de définition de l'entité de travail (le contexte par défaut est la racine de l'arborescence, l'entité par défaut est le noeud contexte, etc...).

Cependant les primitives de définition des entités portant directement sur la structure de la base de données graphique, il est nécessaire de maintenir leur cohérence lors de l'évolution de cette dernière. Ainsi, comme nous l'avons vu au §3.2.3 la destruction de structure invalide l'entité de travail courante qui est remplacée par l'entité par défaut.

Les différentes modifications de structure pouvant interférer avec les primitives de structuration logique sont les suivantes :

- ajout d'une structure,
- suppression d'une structure,
- éclatement implicite d'une structure par référence indicée à 1 noeud.

Dans ce qui suit nous allons étudier ces différents cas, et les problèmes éventuels qu'ils soulèvent.

a) Ajout d'une structure

L'ajout d'une structure s'effectue relativement à l'entité de travail courante, mais ne modifie en rien la structure déjà existante et n'affecte donc pas les références à celle-ci. Cela n'a donc aucune incidence directe sur le contexte ou l'entité de visualisation courants qui restent inchangés. Dans le cas où la structure ajoutée se trouve sous l'un des noeuds de l'entité de visualisation celle-ci sera bien entendu prise en compte par le processus de visualisation.

b) Destruction d'une structure

La destruction de structure s'applique aux éléments de l'entité de travail courante. Celle-ci est totalement indépendante du contexte et de l'entité de visualisation. On risque cependant de supprimer un (des) noeud(s) ayant été référencé(s) lors de la définition de contextes ou de l'entité de visualisation qui n'auraient alors plus de signification.

Aussi pour éviter ce type d'incohérences, la destruction de structure n'est autorisée que si aucun des noeuds de l'entité de travail n'a été référencé dans les expressions de définition des contextes et de l'entité de visualisation.

Dans le cas contraire, la suppression de structure n'est pas effectuée, un message d'erreur le signalant à l'utilisateur qui devra prendre les mesures nécessaires pour effectuer la destruction sans porter atteinte à la cohérence des contextes et/ou de l'entité de visualisation.

c) Eclatement d'une structure suite à une référence indicée

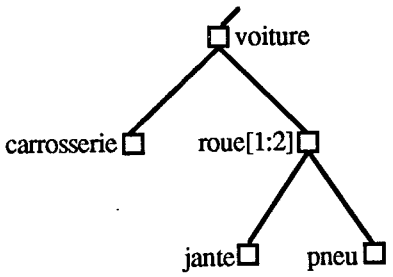
De la même manière, "l'éclatement" d'un noeud suite à l'emploi d'une référence indicée dans l'une des primitives de définition de contexte ou d'entité (de travail ou visualisation) peut modifier la signification d'expressions ayant servi à définir l'autre entité.

Par exemple l'expression ayant défini le contexte de visualisation n'est plus une référence simple si lors de la définition d'un contexte ou d'une entité de travail une référence indicée à un des noeuds qui la composent provoquent un éclatement de structure (fig IV.19)

Aussi pour éviter toute ambiguité suite à un éclatement de structure, CLOVIS impose de manière analogue à la destruction qu'il n'y ait pas de recouvrement entre les entités logiques. C'est à dire qu'un noeud qui subit un "éclatement" lors de l'appel d'une primitive de définition d'un contexte ou d'une entité (de travail ou visualisation) ne doit pas être référencé par un contexte ou l'entité de l'autre nature.

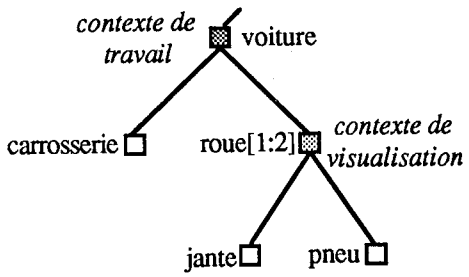
ATTACHE(STRUCTURE("voiture(
carrosserie,roue[1:2](jante,pneu))")

{la structure ci-contre est définie et rattachée au fichier graphique}



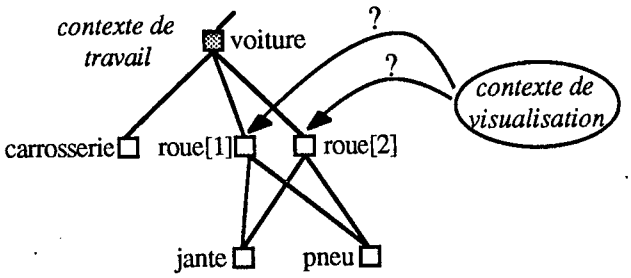
CONTEXTE_TRAV("voiture")
CONTEXTE_VISU("voiture(roue)")

{ Définition des contextes de travail et de visualisation}



ENTITE_TRAV("roue[1]")

{ la référence roue[1] n'est pas acceptée car elle provoquerait un éclatement de structure invalidant le contexte de visualisation }



- figure IV.19 : interférences entre entités logiques de travail et de visualisation -

Dans le cas contraire, la primitive n'a aucun effet et un message d'erreur le signale à l'utilisateur. C'est à lui de prendre ses précautions (redéfinir ses contextes et identités) afin d'autoriser si il le désire l'éclatement.

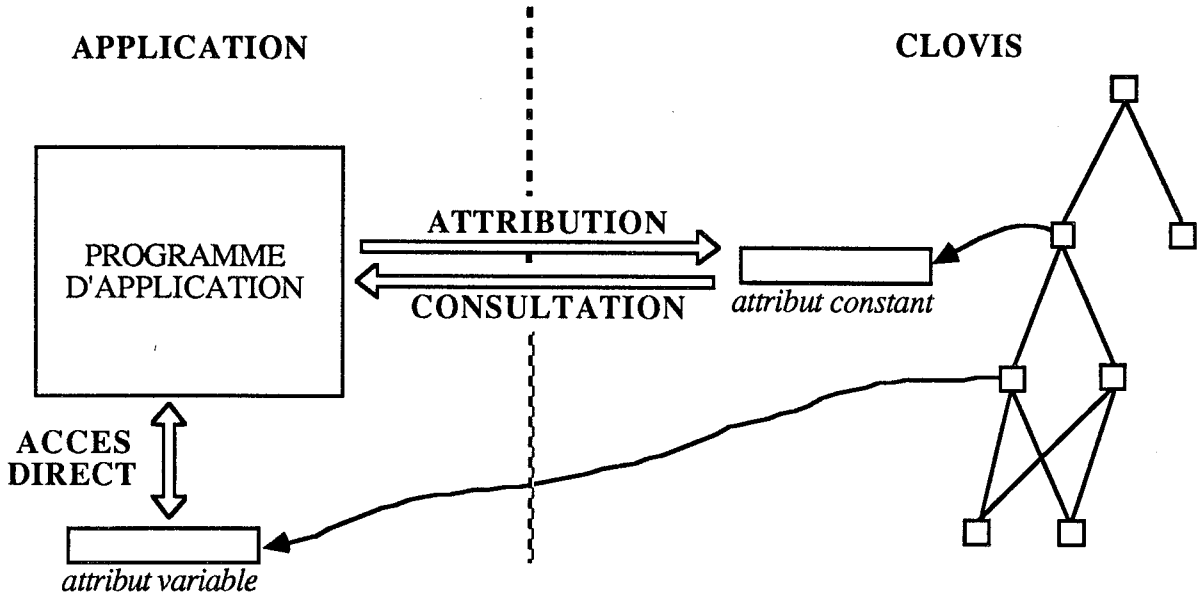
4 LES ATTRIBUTS ELEMENTAIRES DANS CLOVIS

Dans ce qui suit nous présentons les attributs élémentaires considérés par CLOVIS. Comme nous l'avons déjà indiqué au paragraphe 2.2, ces attributs sont regroupés en classes, chaque classe pouvant comporter différents types particuliers. Les attributs sont modélisés à l'aide de descripteurs qui sont associés dynamiquement aux éléments de la base de données graphique, ceci à n'importe quel niveau de structure. Un seul attribut par classe peut être affecté à un élément d'un niveau donné (sauf pour l'aspect qui comme nous le verrons pose un problème particulier).

Les attributs élémentaires que nous présentons sont les attributs de base standards proposés à **toute** application quel que soit le matériel utilisé. Cependant en fonction du degré de sophistication et de réalisme des objets ou des possibilités particulières de certains postes de travail, de nouveaux types d'attributs peuvent être introduits à l'intérieur de chacune des classes. De par son architecture modulaire, CLOVIS permet une prise en compte aisée de ces extensions, tant au niveau de l'ajout de nouveaux types d'attributs qu'au niveau du matériel (cf. chapitre V).

4.1 Attributs constants et attributs variables

Deux catégories d'attributs peuvent être définies selon la manière dont ils sont gérés par le programme d'application (figure IV.20) : les attributs **constants** et les attributs **variables.**



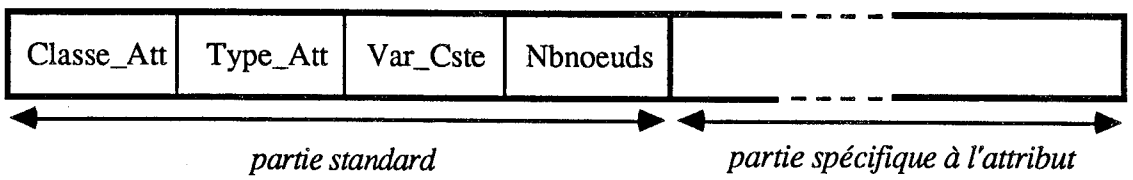
- figure IV.20 : Attributs variables et attributs constants dans CLOVIS -

Les **attributs constants** restent inchangés tout au long de leur existence. Leur gestion est entièrement assurée par CLOVIS, en déchargeant ainsi le programme d'application. Ce dernier n'a accès aux attributs constants qu'au travers des informations transmises au système par le processus d'attribution ou récupérés via le processus de consultation. Ainsi un attribut à caractère constant ne peut pas être directement modifié par l'application qui doit le remplacer explicitement par l'affectation d'un nouveau descripteur. De plus un attribut à caractère constant est "libéré" ("DISPOSE" en Pascal) automatiquement par CLOVIS lorsqu'il n'est plus référencé par aucun noeud de la structure arborescente.

Aussi, afin de minimiser les échanges entre l'application et le système graphique qui parfois peuvent être lourds CLOVIS propose un second type d'attributs : les **attributs variables**. Ceux-ci sont conservés par le programme d'application qui prend complètement en charge leur gestion. Dans ce cas, il accéde directement aux descripteurs d'attributs et peut ainsi les créer, modifier ou détruire sans intervention du système graphique. Par exemple le programme d'application peut modifier des attributs variables sans passer par le processus d'attribution. L'utilisation d'attributs variables impose que l'application connaisse la structure de leurs descripteurs que nous présentons dans le paragraphe suivant.

4.2 Format général des descripteurs d'attributs

Le format des descripteurs est fonction du type de l'attribut représenté. Cependant tous partagent une partie commune utilisée par le logiciel CLOVIS pour assurer leur gestion interne (fig IV.21).



- figure IV.21 : Structure générale des descripteurs d'Attributs -

La partie commune à tous les descripteurs se décompose comme suit :

- *Classe_Att* : définit la classe de l'attribut,

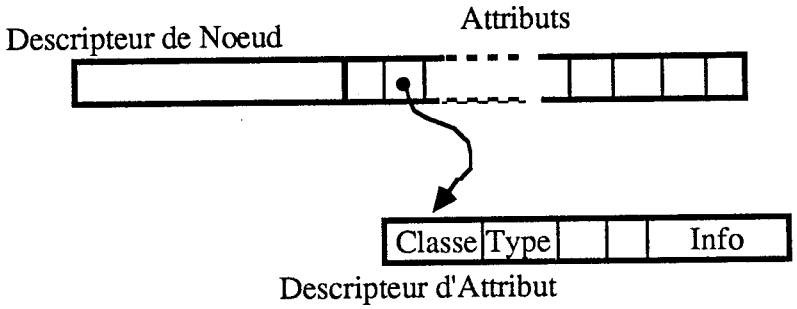
- *Type-_Att* : définit le type de l'attribut à l'intérieur de la classe,

-*Var_Cste* : définit la manière dont l'attribut est géré par CLOVIS, attribut constant (*Var_Cste* = CSTE) ou attribut variable (*Var_Cste* = VARI), (CTSE et VARI sont des constantes prédéfinies dans le fichier CSTES_CLOVIS que le développeur d'applications peut inclure dans ses programmes),

- *Nbnoeuds* : est un compteur qui détermine le nombre de noeuds de la structure arborescente référençant le descripteur d'attribut.

A chaque format de descripteur correspond un type structuré ("record" en PASCAL). Pour chaque classe d'attribut un fichier contient les déclarations des différents types de descripteurs considérés, que l'on peut ainsi inclure directement (option de compilation "include") dans les programmes d'application en fonction des attributs manipulés.

remarque : les descripteurs d'attribut sont attachés de façon dynamique aux différents éléments du fichier graphique. Pour ce faire et sans rentrer dans le détail de la réprésentation interne des noeuds que nous étudierons par la suite, il existe au niveau de chaque noeud du fichier graphique un ensemble d'emplacements destinés à recueillir les adresses des descripteurs des attributs associés à l'élément (voir figure ci dessous). Ces emplacements sont complètement banalisés et peuvent contenir l'adresse de n'importe quel type d'attribut.



- figure IV.22 : Attachement des attributs aux noeuds du fichier graphique -

4.3 Primitives de modélisation des attributs

Dans le but d'éviter au programme d'application la construction des descripteurs d'attributs pour les types de base, CLOVIS propose un ensemble de fonctions standard de modélisation.

Ces fonctions construisent dynamiquement les descripteurs d'attributs à partir d'informations élémentaires fournies en paramètre et renvoient leur adresse.

Par exemple, la fonction :

adr_desc <--- CERCLE_2D(x, y, r)

construit le descripteur d'un attribut pour le type CERCLE_2D de la classe MORPHOLOGIE à partir des coordonnées de son centre et de son rayon, et renvoie son adresse.

Grâce à ces primitives de modélisation, le programme d'application n'a pas l'obligation de connaitre la structure détaillée des descripteurs d'attibuts. Il peut ainsi en confier entièrement la gestion à CLOVIS et ne les manipuler qu'indirectement au travers des processus d'ATTRIBUTION, CONSULTATION et DESCRIPTION. En particulier ces fonctions peuvent

être utilisées pour la définition d'attributs à caractère constant qui sont ainsi manipulés uniquement par CLOVIS (voir § 5.1.3).

Pour chaque type d'attribut CLOVIS propose au moins une fonction de modélisation standard que nous présenterons parallèlement aux attributs de base considérés par CLOVIS dans les paragraphes suivants.

De la même manière qu'il est possible de construire directement les descripteurs d'attribut à l'aide de fonctions de modélisation, le programmeur d'application peut souhaiter accéder à l'information définissant les attributs sans pour autant manipuler la structure de leurs descripteurs. Aussi à chaque fonction de modélisation d'attribut, correspond une fonction de consultation ou "démodélisation" qui permet d'accéder aux informations ayant permis la construction du descripteur d'attribut. Ainsi, si l'on reprend l'exemple de type CERCLE_2D, la fonction de démodélisation associée est

CONS_CERCLE-2D(adr_desc,x,y,r)

où *adr_desc* est l'adresse du descripteur de l'attribut (de type CERCLE_2D) et *x*, *y* et *r* renvoient la valeur du centre et du rayon du cercle.

Toutes les fonctions de démodélisation sont conçues de façon symétriques aux fonctions de modélisation, en ajoutant le suffixe CONS_ au nom de la primitive de modélisation et l'adresse du descripteur à la liste des paramètres. Les autres paramètres sont en tout point identiques (mais ils sont, bien entendu, passés par adresse et non plus par valeur). Aussi, lors de la présentation dans les paragraphes qui suivent, des attributs élémentaires considérés par CLOVIS, nous nous contenterons de donner uniquement la syntaxe des fonctions de modélisation sans revenir sur les fonctions de démodélisation.

4.4 Les classes et types d'attributs dans CLOVIS

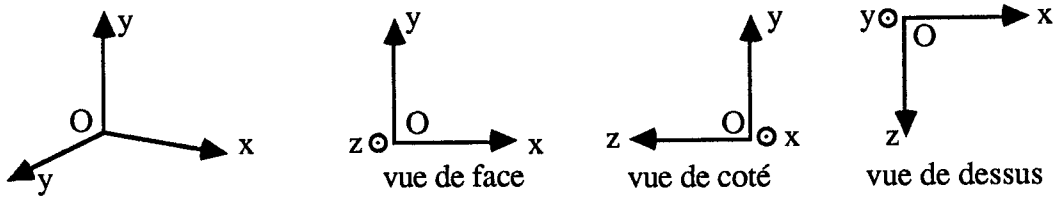
Dans ce qui suit nous présentons de façon détaillée les différentes classes et types d'attributs considérés dans cette première version de CLOVIS. Nous essayerons, en évitant de trop anticiper sur les processus de base (ATTRIBUTION, CONSULTATION, DESCRIPTION et VISUALISATION) exposés au § 5, d'indiquer les différentes règles qui régissent l'utilisation de ceux-ci et en particulier la manière dont ils se composent dans la structure arborescente lors de la visualisation.

4.4.1 Attributs Morphologiques

Les attributs de la classe morphologie permettent de définir la forme intrinsèque d'un élément. Comme les attributs des autres classes ceux ci-peuvent être attachés à n'importe quel niveau de structure.

Les coordonnées définissant les différentes morphologies sont exprimées dans le repère local des différents noeuds auxquels elles sont affectées, lors de la visualisation celles-ci sont transformées dans le référentiel absolu (repère de la scène) en leur appliquant hiérarchiquement les transformations géométriques associées aux différents niveaux de la structure (voir § 4.4.3 concernant les attributs géométriques).

Il est possible de définir des objets bidimensionnels aussi bien que des objets tridimensionnels. Dans ce dernier cas la convention adoptée pour les repères est la suivante : repère direct Oxyz, avec le plan oxy correspond à une vue de face, comme nous le montre la figure IV.23. Ce choix permet de traiter ainsi simultanément et de façon aisée des morphologies bidimensionnelles et tridimensionnelles, ainsi que de considérer la visualisation 2D comme un cas dégénéré du 3D (vue de face et projection parallèle dans le plan z=0) .



- fig. IV.23 : repère pour la définition des objets tridimensionnels -

a) les morphologies bidimensionnelles.

Les différents types d'attributs morphologiques 2D reprennent ceux que l'on retrouve de façon classique dans de très nombreux logiciels graphiques (figure IV.24). L'utilisateur a ainsi à sa disposition les types :

• **POINTS2D** qui défini un "nuage" de points dans le plan. L'application peut construire les descripteurs des attributs de ce type à l'aide de la fonction de modélisation

MPOINTS2D(nbpts,tabx,taby) ---> adr_descr

où

- *nbpts* indique le nombre de points du nuage,
- *tabx,taby* sont des tableaux qui contiennent les coordonnées (réelles) de ces points.

• **SEGMENTS2D** qui désigne une séquence de segments définis par la suite des coordonnées de leurs extrémités. La construction de descripteurs pour les attributs de ce type peut être confiée à la fonction

MSEGMENTS2D(nbseg,tabx,taby) --> adr_descr

où

- *nbseg* indique le nombre de segments qui composent l'attribut,
-*tabx,taby* contiennent les coordonnées des extrémités de ces segments (deux éléments successifs des tableaux tabx et taby définissent un segment, ces tableaux contiendront donc 2*nbseg coordonnées).

• **LIGNE2D** qui représente une ligne brisée définie par la suite des coordonnées de ses sommets. La fonction de modélisation proposée pour ce type d'attribut est :

MLIGNE2D(nbpts,tabx,taby) --> adr_descr

où

- *nbpts* indique le nombre de sommets de la ligne brisée et
- *tabx,taby* contiennent les coordonnées de ces sommets.

• **POLYG2D** qui désigne un polygone plan (ligne brisée fermée), défini par la suite des coordonnées de ses sommets. La construction de descripteurs d'attributs de ce type peut être confiée à la fonction :

MPOLYG2D(nbpts,tabx,taby) --> adr_descr

où

- *nbpts* indique le nombre de sommets du polygône,
- *tabx,taby* contiennent les coordonnées de ces sommets.

• **CERCLE2D** qui réprésente un cercle exprimé par la donnée de son centre et de son rayon. La fonction de modélisation pour les cercles est :

MCERCLE2D(x,y,r) --> adr_descr

avec*x,y* les coordonnées du centre du cercle et *r* son rayon.

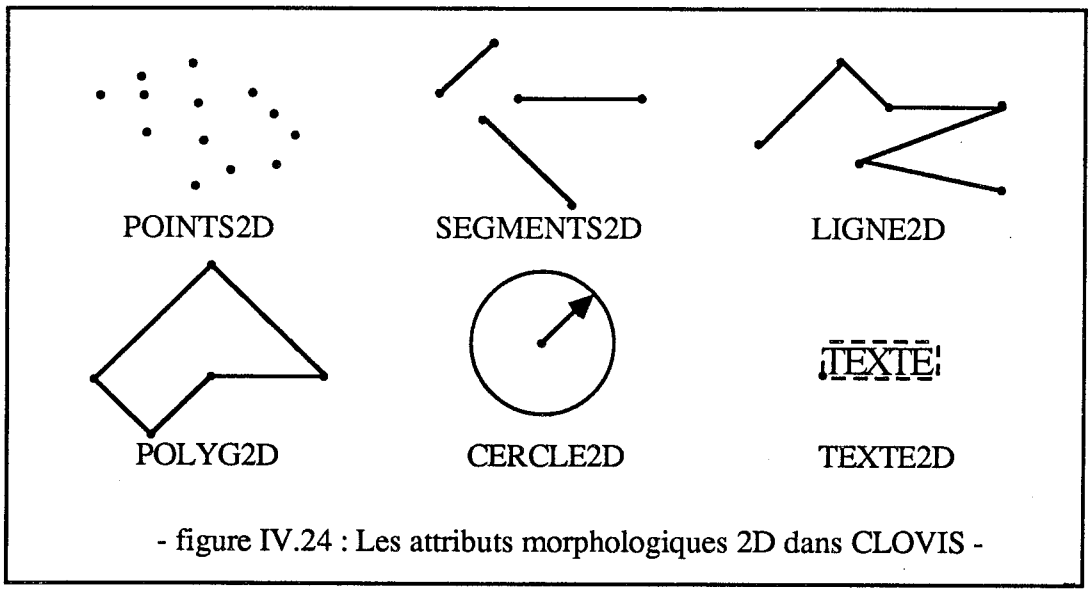
• **TEXTE2D** qui désigne un texte alphanumérique defini par la suite des caractères ASCII qui le composent et les coordonnées du coin inférieur gauche de la chaîne. La fonction de construction pour les descripteurs d'attributs de ce type est

MTEXTE2D(x,y,nbcar,chaîne) --> adr_descr

avec

- *x,y* les coordonnées du point (coin inférieur gauche) à partir duquel débute le texte,
- *nbcar* le nombre de caractères du texte,
- *chaîne* le tableau de caractères qui contient le texte à afficher.

remarque : les types POINTS2D, SEGMENTS2D et LIGNE2D définissent des morphologies de type "fil de fer" ("wireframe"), par contre les types POLYG2D et CERCLE2D définissent des morphologies qui pourront aussi bien être visualisées sous forme de dessin au trait (représentation du contour uniquement) que sous forme de surface pleine. C'est l'attribut d'aspect qui défini la manière d'interpréter ces derniers (voir § 4.4.2).



- figure IV.24 : Les attributs morphologiques 2D dans CLOVIS -

b) les morphologies tridimensionnelles.

Les différentes types de base pour les morphologies tridimensionnelles proposés par CLOVIS sont l'extension des types bidimensionnels vus précédemment (voir figure IV.25). On retrouve donc les types suivants :

• **POINTS3D** qui défini un "nuage" de points dans l'espace. L'application peut construire les descripteurs des attributs de ce type à l'aide de la fonction de modélisation

MPOINTS3D(nbpts,tabx,taby,tabz) ---> adr_descr

où

- *nbpts* indique le nombre de points du nuage,
- *tabx,taby ,tabz* sont des tableaux qui contiennent les coordonnées (réelles) de ces points.

• **SEGMENTS3D** qui désigne une séquence de segments définis par la suite des coordonnées de leurs extrémités. La construction de descripteurs pour les attributs de ce type peut être confiée à la fonction

MSEGMENTS3D(nbseg,tabx,taby,tabz) --> adr_descr

où

- *nbseg* indique le nombre de segment qui composent l'attribut,
- *tabx,taby,tabz* contiennent les coordonnées des extrémités de ces segments.

• **LIGNE3D** qui représente une ligne brisée définie par la suite des coordonnées de ses sommets. La fonction de modélisation proposée pour ce type d'attribut est :

MLIGNE3D(nbpts,tabx,taby,tabz) --> adr_descr

où

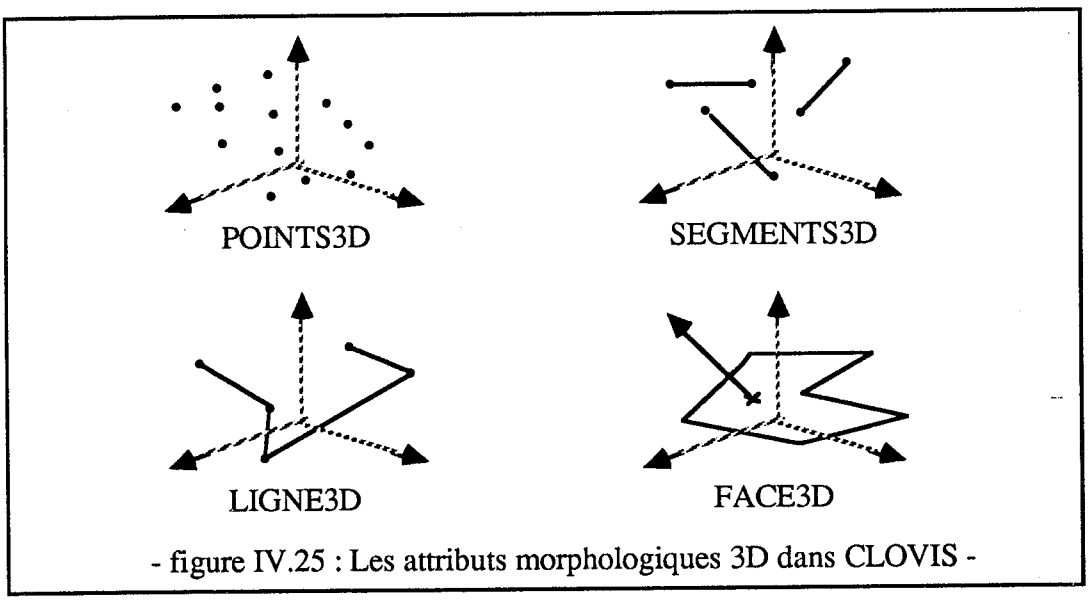
- *nbpts* indique le nombre de sommets de la ligne brisée et
- *tabx,taby ,tabz* contiennent les coordonnées de ces sommets.

• **FACE3D** qui désigne une face (supposée plane) dans l'espace, définie par la suite des coordonnées de ses sommets et sa normale orientée. La construction de descripteurs d'attributs de ce type peut être confiée à la fonction :

MFACEG3D(nbpts,nx,ny,nz,tabx,taby,tabz) --> adr_descr

où

- *nbpts* indique le nombre de sommets du polygône définisant la face,
- *nx,ny,nz* definissent le vecteur normal à la face,
- *tabx,taby,tabz* contiennent les coordonnées des sommets du polygône.



- figure IV.25 : Les attributs morphologiques 3D dans CLOVIS -

4.4.2 Attributs d'Aspect

Le problème de la définition et de la modélisation de l'aspect des objets graphiques constitue certainement l'un des points qui soulève le plus de difficultés. En effet, les différentes manières d'exprimer l'aspect sont extrêmement dépendantes du matériel (par exemple la couleur peut être définie par la valeur des trois composantes primaires Rouge, Vert Bleu ou bien par un simple code).

C'est pourquoi les paramètres d'aspects sont pour la plupart exprimés à l'aide de mots clés référençant les possibilités locales des postes de travail.

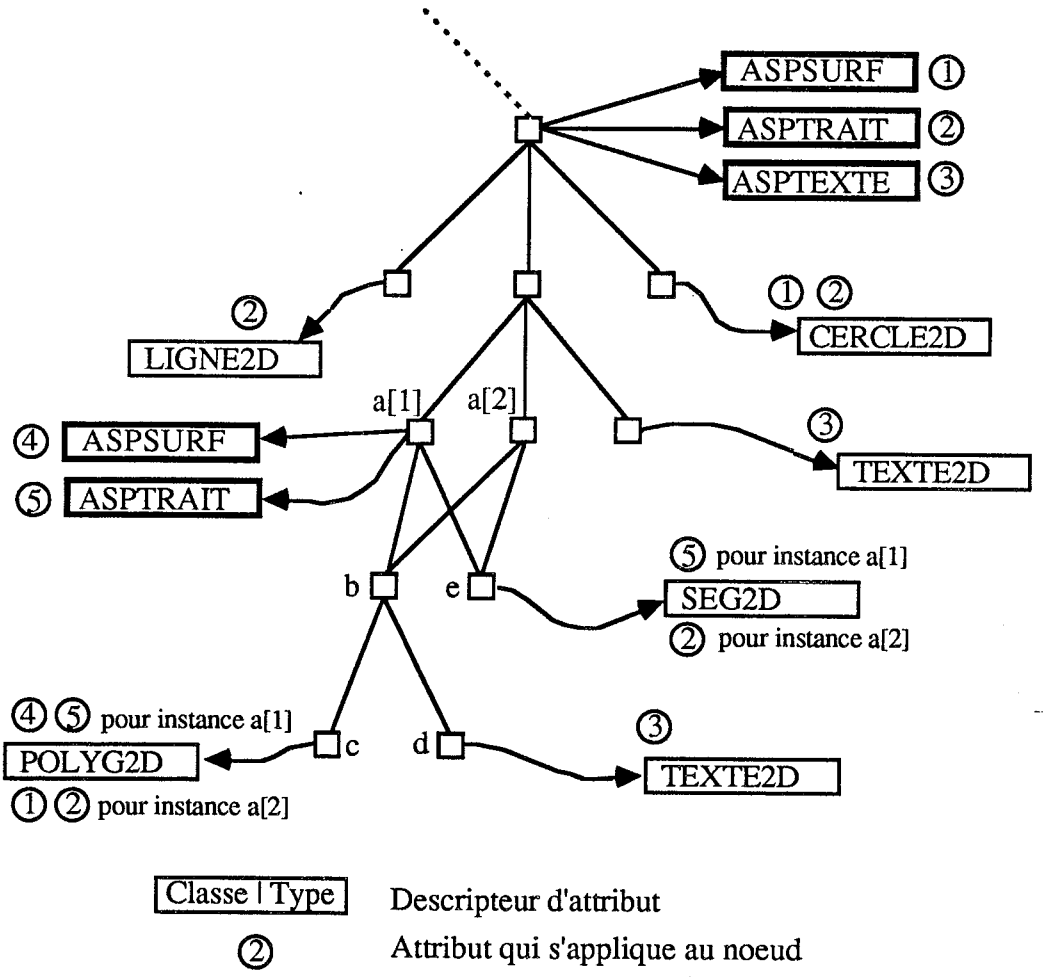
4.4.2.1 Les "sous-classes" d'aspect

L'aspect ne forme pas une classe monolithique dans CLOVIS, mais se divise en trois "sous-classes" qui concernent chacune des types morphologiques bien précis. Ainsi, on distingue :

- **l'aspect du trait** (classe ASPTRAIT) qui s'applique aux morphologies "fil de fer" (MPOINTS2D/3D, SEGMENTS2D/3D, LIGNE2D/3D) ainsi qu'aux contours des morphologies de type "surface" (c'est à dire celles pour lesquelles l'attribution d'un aspect à leur intérieur a une signification : POLYG2D,CERCLE2D,FACE3D).

- **l'aspect surface** (classe ASPSURF) qui permet de définir l'intérieur des morphologies de type "surface" (POLYG2D, CERCLE2D, FACE3D).

- **l'aspect du texte** (classe ASPSURF) qui lui concerne uniquement les éléments de type texte (TEXTE2D).



- figure IV.26 : Les différentes classes d'attributs d'aspect dans CLOVIS -

Il est donc possible d'affecter à un même élément de la structure hiérarchique un attribut pour chacune de ces trois classes. Les attributs sont **hérités** dans la structure hiérarchique : ils s'appliquent à tous les éléments de la sous-structure située sous le noeud auquel ils sont affectés. Dans le cas où plusieurs attributs d'aspect sont définis à des niveaux de structure différents, la visualistation d'un élément s'effectue en utilisant les attributs d'aspect

correspondant à son type morphologique, situés le plus bas dans la hiérarchie. La figure IV.26 illustre l'utilisation des attributs d'aspect dans CLOVIS; on remarquera les noeuds instances a[1] et a[2] qui permettent le partage d'une sous-structure affichée avec des attributs différents selon qu'il s'agit de l'instance a[1] ou a[2].

Chacun des attributs à l'intérieur de ces différentes classes regroupe un ensemble d'informations qui sont:

- pour l'aspect du trait (ASPTRAIT) :
  - la couleur,
  - la texture (trait plein, tireté court, tireté long, mixte),
  - l'épaisseur,
  - la visibilité,
  - le clignotement.

- pour l'aspect des surfaces (ASPSURF) :
  - la texture de remplissage (motif), (celle ci peut être unie),
  - la couleur,
  - la visibilité,
  - le clignotement,
  - le modèle de reflexion (pour les faces3d).

- pour l'aspect des textes (ASPTEXTE) :
  - la police de caractères utilisée,
  - la couleur,
  - la visibilité,
  - le clignotement,
  - la taille des caractères,
  - leur orientation.

4.4.2.2 Gestion et Représentation interne des attributs d'aspect

Plutôt que de regrouper dans un seul descripteur d'attribut toutes les informations liées à un aspect, nous avons préféré définir pour chacune d'elles un type d'attribut élémentaire (et donc un descripteur le modélisant). Ainsi, par exemple, la couleur du trait et la texture du trait constituent respectivement les types COULTRAIT et TEXT_TRAIT à l'intérieur de la classe aspect du trait ASPTRAIT.

Les raisons de ce choix sont multiples :

- il permet une définition séparée des différentes informations regroupées sous un même aspect. Ainsi seuls les attributs élémentaires caractérisant l'aspect d'une structure sont associés à un noeud (par exemple la couleur du trait). Cela offre à l'utilisateur une beaucoup plus grande souplesse dans la spécification de l'aspect des objets graphiques, beaucoup plus simple qu'une définition "en bloc" de toutes les information rattachées à un même aspect.

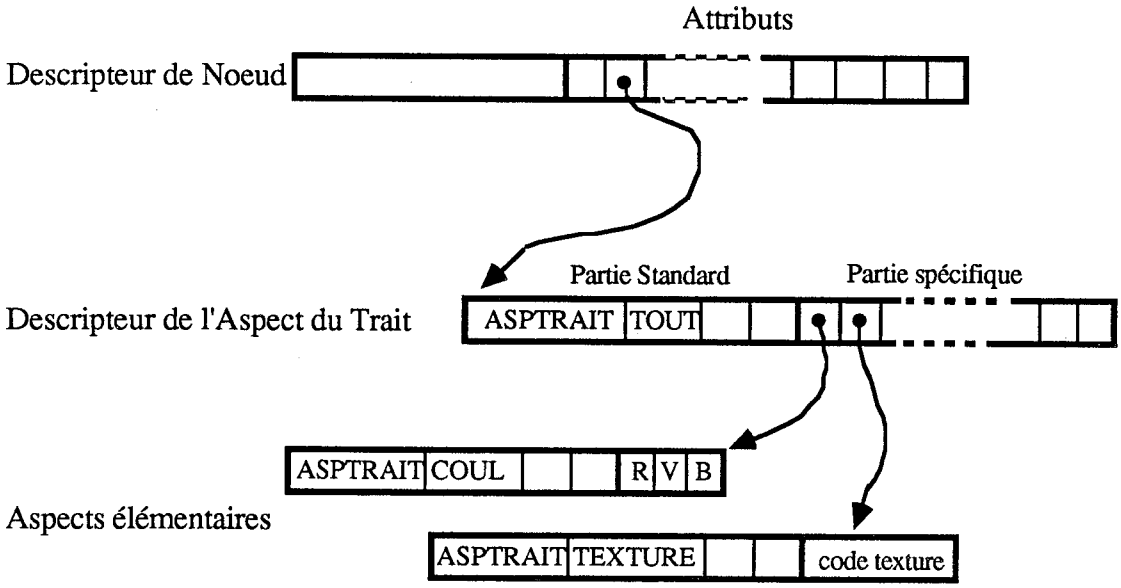
- il permet un gain de place car seules sont mémorisées les informations d'aspect particularisant une structure.

- il permet finalement le partage et la réutilisation des informations élémentaires d'aspect entre plusieurs structures (partage du descripteur d'attribut élémentaire par plusieurs noeuds de la structure hiérarchique).

Cependant, cette solution soulève, en contrepartie, certains problèmes car contrairement aux attributs des autres classes, plusieurs attributs d'une même classe d'aspect peuvent "cohabiter" simultanément sur un même noeud (par la exemple couleur du trait, texture du trait...dans la classe ASPTRAIT).

Bien entendu, on aurait pu multiplier les classes d'attributs, chaque aspect élémentaire constituant une classe en lui même. Mais cela aurait introduit un nombre très important de classes différentes et entrainé un perte de place importante au niveau de chaque noeud du fichier graphique. En effet les descripteurs de noeud doivent contenir au moins autant d'emplacement destinés à recevoir les adresses des descripteurs d'attributs qu'il existe de classes d'attributs.

Aussi avons nous adopté une solution intermédiaire basée sur une structure à deux niveaux (figure IV.26). Les descripteurs d'attributs d'aspect élémentaires ne sont pas directement raccrochés aux noeuds du fichier graphique mais rattachés à un descripteur intermédiaire ("descripteur d'aspect global") qui permet de regrouper tous les aspects d'une même classe (ASPTRAIT, ASPSURF ou ASPTEXT).



- figure IV.27 : Attachement des Attributs d'aspect au fichier graphique -

Cette gestion à deux niveaux est transparente pour l'utilisateur qui n'a pas à se soucier de la présence ou non du descripteur intermédiaire lors de la définition d'un aspect élémentaire.

Ainsi, si un aspect élémentatire est affecté à un noeud ne possédant pas d'aspect de la classe de l'attribut, le descripteur "global" est crée automatiquement et l'attribut raccroché à celui-ci. Sinon, l'affectation se fait directement au descripteur "global". Dans ce cas, si un attribut du même type existe déjà, l'adresse du nouvel attribut le remplace, sinon cette dernière est rangée dans le premier emplacement disponible.

Outre l'économie de place mémoire (la place ne sera réservée que lors de l'association d'un aspect à un noeud) cette solution présente plusieurs avantages :

- elle permet de traiter les informations d'aspect d'une même classe soit individuellement (en manipulant les descripteurs d'aspects élémentaires) soit en bloc (en manipulant le descripteur global) de manière uniforme (pour l'application ces attributs se comportent de manière similaire quel que soit leur niveau).

- elle permet la prise en compte aisée de nouvelles informations élémentaires à l'intérieur d'une classe d'aspect (il suffit de définir un nouveau type de descripteur, ceux-ci étant traités de façon banalisée au niveau de l'attribut global).

4.4.2.3 Les aspects par défaut

Dans le cas où le programme d'application ne définirait pas d'aspect dans une structure hiérarchique, un aspect par défaut doit être appliqué lors de la visualisation des différentes morphologies qu'elle contient. Ces aspects par défaut sont associés à la racine du fichier

graphique lors de l'initialisation de CLOVIS.

Ceux-ci peuvent cependant être redéfinis à tout moment par l'application à l'aide de primitives d'initialisation particulières :

ASPTRAIT_DEF(coultrait,texture_trait.....)
ASPSURF_DEF(coulsurf,texture.....)
ASPTEXT_DEF(coultext,police.....).

4.4.3 Attributs Géométriques

Les attributs géométriques définissent de manière classique dans une hiérarchie une transformation de modélisation qui positionne un élément relativement à l'élément de niveau supérieur (voir chapitre III).

A chaque noeud est associé un repère local, dans lequel sont exprimées les coordonnées définissant son attribut morphologique. L'attribut géométrique attaché à ce noeud spécifie donc les transformations géométriques qu'il faudra appliquer aux points exprimés dans le repère local à ce noeud pour obtenir leurs coordonnées dans le repère du noeud père.

L'absence d'attribut géométrique en un noeud signifie que son repère local et celui de son père coincident (la transformation liant ces deux repères est l'identité).

Ainsi, à partir de la géométrie d'un noeud par rapport à son père il est possible d'obtenir la géométrie de ce noeud par rapport à n'importe quel autre noeud hiérarchiquement supérieur. Ceci s'effectue en composant les géométries successives des noeuds entre l'élément considéré et l'élément dont le repère sert de référentiel de base.

Dans CLOVIS deux catégories d'attributs géométriques sont considérées selon la dimension du référentiel absolu. Il est possible de définir des transformations dans le plan pour les scènes bidimensionnelles (type **Géom2D**), et des transformations dans l'espace pour des scènes tridimensionnelles (type **Géom3d**).

Un attribut géométrique est modélisé de manière classique à l'aide d'une matrice en coordonnées homogènes [RoAd 76] .

Il peut être parfois utile lors de la définition d'un attribut géométrique de tenir compte de la géométrie déjà existante pour le noeud et de ne pas se contenter de remplacer purement et simplemement l'ancienne transformation par une nouvelle. C'est pourquoi pour chacun des deux types géométriques (Géom2D et Géom3D) CLOVIS propose trois "sous-types" qui permettent de tenir compte des différents cas de figure possibles lors de l'affectation d'un attribut géométrique. Ils permettent de composer automatiquement l'attribut géométrique avec celui déjà existant dans l'arborescence lors de son affectation. Cela décharge l'application qui peut ainsi modifier directement la géométrie d'un noeud sans avoir besoin de consulter l'attribut déjà existant, construire un nouveau descripteur en composant les deux transformations, affecter celui-ci à la structure.

L'utilisateur peut ainsi définir :

- une transformation géométrique **absolue** (type **Géom2D_ABS** ou **G2om3D_ABS**). Un attribut de ce type défini simplement la position d'un l'élément dans le repère local de l'élément de structure de niveau supérieur. Ainsi, lors de son affectation à un noeud l'attribut géométrique est substitué à l'attribut géométrique éventuellement existant .

- une transformation géométrique **relative** (type **Géom2D_REL** ou **Géom3D_REL**) qui exprime un déplacement du noeud dans le repère local de l'élément de niveau supérieur de structure **relativement** à sa position courante. Aussi lors de l'affectation à un noeud d'un attribut de ce type, il est composé avec la géométrie déjà existante. Le nouvel attribut

géométrique du noeud est obtenu en postmultipliant la matrice du descripteur existant par la matrice du descripteur de l'attribut relatif.

- une transformation géométrique **locale** (type **Géom2D_LOC** ou **Géom3D_LOC**) qui exprime un déplacement du noeud dans son propre repère local sans affecter sa position actuelle par rapport au repère de l'élément de niveau de structure supérieur. Lors de son affectation, un attribut de ce type sera donc composé avec la géométrie éventuellement existante en prémultipliant la matrice qui défini celle-ci par celle du descripteur de géométrie relative.

Il faut bien remarquer que ces différents types sont uniquement une facilité offerte par CLOVIS à l'utilisateur pour la définition de la géométrie des éléments de la structure. De manière interne CLOVIS ne considère que les deux types Géom2D et Géom3D.

Pour la construction des descripteurs d'attributs géométriques, CLOVIS propose diverses primitives de modélisation dont la forme la plus générale est pour les transformations 2D :

GEOM2D(typgeom,m11,m12,m21,m22,m31,m32) --> adr_descrp

et pour les transformations 3D

GEOM3D(typgeom,m11,m12,m13,m21,m22,m23,
m31,m32,m33,m41,m42,m43) --> adr_descrp

- *typgeom* indique le type de la transformation : absolue (**ABS**), relative (**REL**) ou locale (**LOC**),

- m_{ij} sont les coefficients (réels) définissant la matrice de transformation en coordonnées homogènes.

Les matrices ainsi définies s'appliquent aux coordonnées des objets de la façon suivante :

- pour les transformations 2D : $(x,y,1) * \begin{pmatrix} m11 & m22 \\ m21 & m22 \\ m31 & m32 \end{pmatrix} = (X,Y)$

- pour les transformations 3D : $(x,y,z,1) * \begin{pmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \\ m41 & m42 & m43 \end{pmatrix} = (X,Y,Z)$

En plus de ces deux fonctions de modélisation générales, CLOVIS offre par ailleurs d'autres primitives qui permettent la construction automatique des matrices pour les transformations géométriques élémentaires que sont les translations, rotations et homothéthies.

Pour la géométrie 2D :

TRANSLATION2D(typgéom,x,y) ---> adr_descr

permet de définir une translation absolue, relative ou locale de vecteur (x,y).

ROTATION2D(typgéom,x,y,a) ---> adr_descr

permet de définir une rotation absolue,relative ou locale de centre x , y et d'angle a..

HOMOTHETIE2D(typgéom,x,y,hx,hy) ---> adr_descr

permet de définir une homothétie absolue, relative ou locale de centre x , y avec les facteurs d'échelle hx selon l'axe des abscisses et hy selon l'axe des ordonnées.

Pour la géométrie 3D on retrouve des fonctions équivalentes :

TRANSLATION3D(typgéom,x,y,z) ---> adr_descr

permet de définir une translation absolue, relative ou locale de vecteur (x,y ,z).

ROTATION3D(typgéom,x1,y1,z1,x2,y2,z2,a) ---> adr_descr

permet de définir une rotation absolue, relative ou locale d'angle a par rapport à l'axe défini par les points ($x1$, $y1$, $z1$) et ($x2$, $y2$, $z2$).

HOMOTHETIE3D(typgéom,x,y,z,hx,hy,hz) ---> adr_descr

permet de définir une homothétie absolue,relative ou locale de centre x ,y ,z avec les facteurs d'échelle hx selon l'axe Ox, hy selon l'axe Oy et hz selon l'axe Oz.

Pour la description des attributs géométriques seuls existent des processus pour la définition des transformations élémentaires (rotations, translations, homothéties). Les transformations décrites sont exprimées dans le repère de la scène visualisée (type géométrie ABSOLUE)

4.4.4 Attributs de Géométrie de prise de Vue (Gv) et de Géométrie d'affichage (Ga)

Nous présentons simultanément les attributs de géométrie de prise de vue (Gv) et de géométrie d'affichage (Ga) car leur comportement est analogue dans CLOVIS.

4.4.4.1 Les types d'attributs Gv et Ga et leur composition dans la structure hiérarchique

A l'intérieur de la classe **Gv** trois types d'attributs sont pris en compte :

- **GV_2D** qui défini la prise de vue pour une scène bidimensionnelle sous la forme d'une fenêtre rectangulaire dans l'espace utilisateur,

- **GV_3D_PERSP** et **GV_3D_PARAL** qui définissent la prise de vue pour une scène tridimensionnelle, respectivement sous la forme d'une projection perspective et d'une projection parallèle.

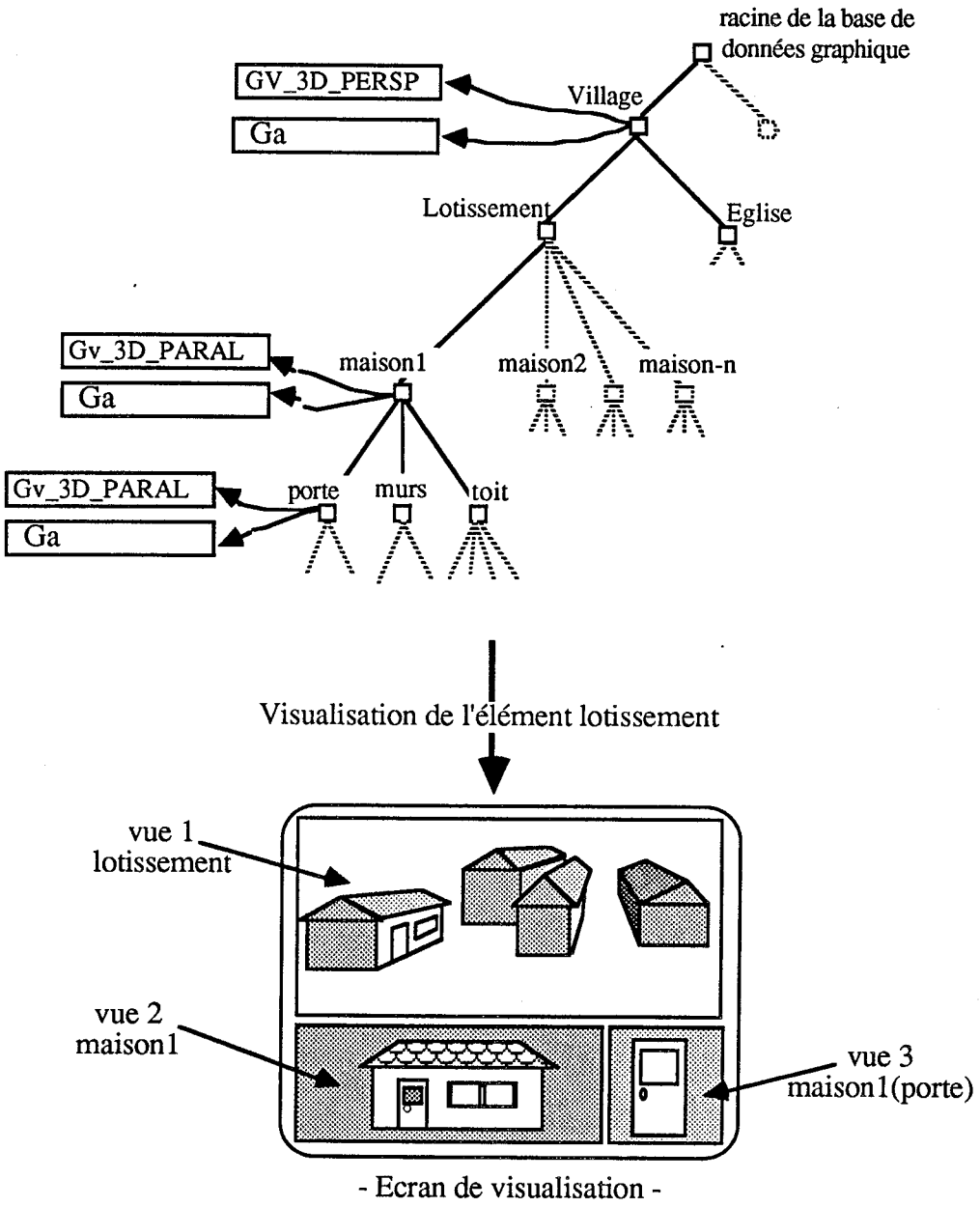
Un seul type d'attribut est considéré pour la classe **Ga**, il est défini par une clôture rectangulaire exprimée dans l'espace d'adressage du dispositif de visualisation.

Les attributs des classes Gv et Ga peuvent être associés à n'importe quel niveau de la structure, rendant possible la visualisation simultanée de sous-objets différents avec des prises de vue différentes et dans des clotures différentes. Pour cela, les attributs de géométrie d'affichage et de géométrie de prise de vue se composent dans la structure hiérarchique de la manière suivante :

- Lors de la visualisation d'un élément on lui applique les premiers attributs des classes Ga et Gv trouvés en explorant l'arbre vers le haut. Ainsi dans l'exemple donné par la figure IV.27, l'élément *"lotissement"* est affiché avec les attributs de prise de vue de la scène *"village"* à laquelle il est raccroché.

- Toutefois il est possible que dans la structure définissant l'élément visualisé se trouvent d'autres attributs de classe Gv ou Ga. Dans ce cas, le sous-objet auquel il sont associés est affiché deux fois : la première avec les attributs Gv et Ga de l'élément désigné pour la

visualisation, la seconde avec ses propres attributs Gv et Ga. Ainsi, dans notre exemple l'élément *"maison1"* du *"lotissement"* est affiché simultanément avec le lotissement dans son ensemble avec une prise de vue perspective (vue1) et individuellement avec une prise de vue parallèle (vue2). Cette composition des attributs Gv et Ga s'applique de manière récursive dans l'arborescence. De cette façon, la *"porte"* de la *"maison1"* peut elle même être affichée dans une vue particulière (vue3).



- figure IV.27 : Composition des attributs de Géométrie de Prise de vue et de Géométrie d'Affichage -

4.4.4.2 Construction des attributs de géométrie de prise de vue

CLOVIS propose plusieurs primitives pour la construction des descripteurs des attributs de prise de vue.

FENETRE(xg,yg,xd,yd,clip) --> adr_descr

permet de définir un attribut de type **GV_2D** où *xg,yg,xd,yd* sont les coordonnées des coins inférieurs gauche et supérieurs droit de la fenêtre dans l'espace utilisateur et *clip* est un booléen indiquant si un coupage par rapport aux bords de la fenêtre doit être effectué ou non.

Pour définir une prise de vue perspective (attribut de type **GV_3D_PERSP**), CLOVIS propose la primitive

GV3D_PERSP(xoeil,yoeil,zoeil,xvis,yvis,zvis,vxup,vyup,vzup,
xfg,yfg,xfd,yfd,clip,d_avant,d_arrière) --> adr_descr

La signification des différents paramètres (voir fig.IV.29) de cette fonction de modélisation est la suivante :

- *xoeil,yoeil,zoeil* : position de l'observateur (centre de projection) dans l'espace utilisateur.

- *xvis,yvis,zvis* : point de visée (PV). Le plan de projection est perpendiculaire au vecteur VNP = (xvis-xoeil,yvis-yoeil,zvis-zoeil) et contient le point de visée.

- *vxup,vyup,vzup* : indiquent l'orientation du repère (PV,u,v) dans le plan de projection. La projection orthogonale de ce vecteur (vxup,vyup,vzup) sur le plan de projection défini la direction du vecteur v, le vecteur u est tel (u,v, VNP) forme un repère indirect.

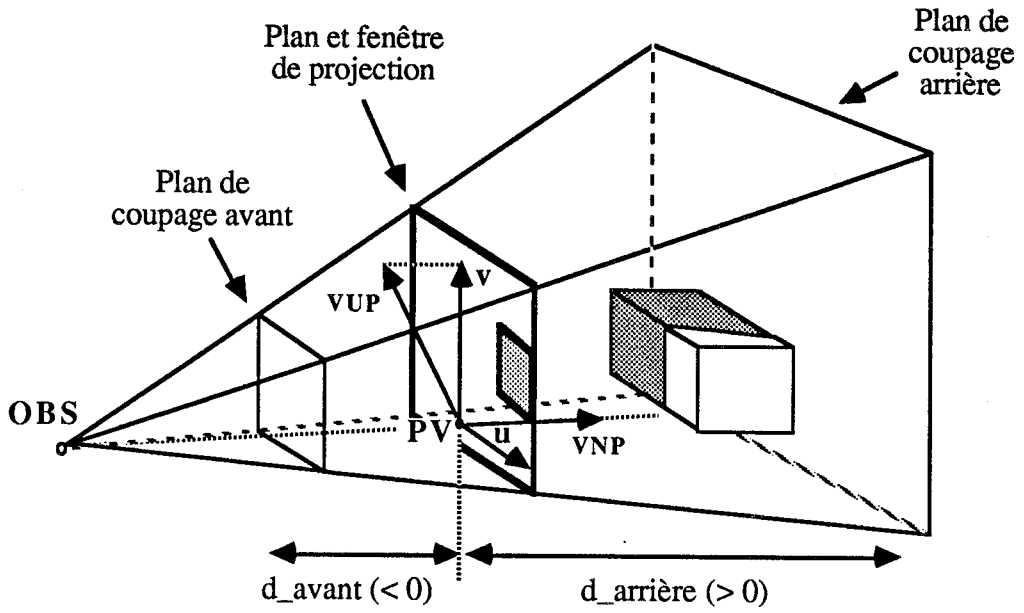
- *xfg,yfg,xfd,yfd* : limites d'une fenêtre dans le plan de projection, exprimées dans le repère (PV,u,v). Le volume de vision est défini par les projecteurs passant par les coins de cette fenêtre.

- *clip* : est un indicateur pour l'activation du coupage par rapport au volume de vision. Différentes valeurs sont possibles selon que le coupage doit être effectué uniquement par rapport à la fenêtre ou en considérant un plan de coupage avant ou arrière. Ces différentes valeurs sont obtenues en combinant les constantes :

  - **néant** : pas de coupage du tout,
  - **fen** : coupage par rapport au bords du volume de vision,
  - **avant** : coupage par rapport à un plan de coupage en avant du point de visée et parallèle au plan de projection,
  - **arrière** : coupage par rapport à un plan de coupage en arrière du point de visée et parallèle au plan de projection.

Par exemple la valeur **fen+avant+arrière** pour l'indicateur *clip* signifie que le coupage doît être effectué en prenant en compte les limites de la fenêtre plus les plans de coupage avant et arrière.

- *d_avant* et *d_arrière* : definissent les plans de coupage avant et arrière et sont les mesures algébriques exprimant la distances de ces plans relativement au plan de projection. Les valeurs positives définissent une distance selon la direction de visée (VNP). Si le coupage par rapport à ces plans n'est pas activé, ces paramètres n'ont aucune signification.



OBS: observateur (xoeil,yoeil,zoeil)
PV: point de visée (xvis,yvis,zvis)
VNP: vecteur normal au plan de projection (direction de visée)
VUP: vecteur d'orientation du repère (PV,u,v) dans le plan de projection

- figure IV.29 : paramètres de définition de prise de vue perspective -

Une seconde primitive permet de définir une géométrie de prise de vue tridimensionnelle, il s'agit de

GV3D_PARAL(dx,dy,dz,xvis,yvis,zvis,vxup,vyup,vzup,xfg,yfg,xfd,yfd, clip,d_avant,d_arrière) --> adr_descr

qui permet de spécifier une projection parallèle orthogonale (attribut de type **GV_3D_PARAL**). La signification des différents paramètres (voir fig.IV.30) de cette fonction de modélisation est la suivante :

- *dx,dy,dz* : direction de projection.

- *xvis,yvis,zvis* : point de visée (PV). Le plan de projection est perpendiculaire au vecteur VNP (dx,dy,dz) et contient le point de visée.

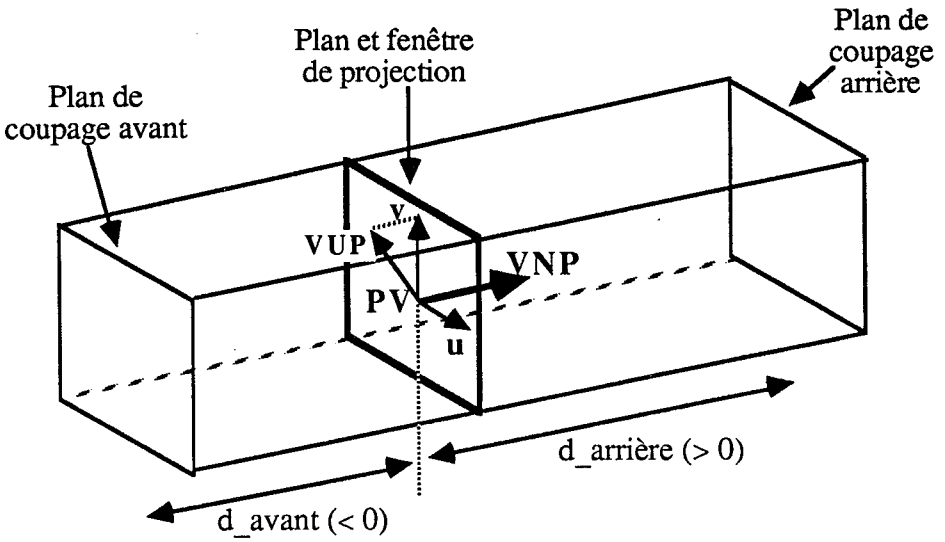
Les autres paramètres ont une signification identique à ceux utilisés pour la définition de projections perspectives :

- *vxup,vyup,vzup* : définissent l'orientation du repère (PV,u,v) dans le plan de projection.

- *xfg,yfg,xfd,yfd* : limites de la fenêtre dans le plan de projection.

- *clip* : indicateur pour l'activation du coupage par rapport au volume de vision.

- *d_avant* et *d_arrière* : definissent les plans de coupage avant et arrière et sont les distances de ces plans par rapport au plan de projection.



VNP : vecteur normal au plan de projection (direction de projection) dx,dy,dz
PV: point de visée (xvis,yvis,zvis)
VUP : vecteur d'orientation du repère (PV,u,v) dans le plan de projection

- figure IV.30 : paramètres pour la prise de vue parallèle -

4.4.4.3 Construction des attributs de géométrie d'affichage

Comme nous l'avons vu précédemment la géométrie d'affichage exprime la zône du dispositif de visualisation où doit être affichée une structure. Cependant à cette information purement géométrique il est nécessaire d'ajouter plusieurs paramètres qui définissent l'aspect de cette zône :

- la couleur éventuelle du fond cette zône,
- l'affichage ou non d'un cadre autour de cette zône.

Ces attributs sont étroitement liés à la définition de la zône, aussi les avons nous inclus dans les descripteurs de géométrie d'affichage. Pour le cadre nous avons dans cette première implémentation uniquement considéré sa mise en évidence par un simple trait.

La construction d'un descripteur d'attribut de géométrie d'affichage peut être réalisée par l'intermédiaire de la fonction

CLOTURE(xg,yg,xd,yd,viscadre,ref_coul_cadre, visfond,ref_coul_fond) --> adr_descr

- *xg,yg,xd,yd* sont les limites (coin inférieur gauche et coin supérieur droit) d'une clôture définie dans l'espace d'adressage du dispositif de visualisation utilisé. Afin d'effectuer une définition de la géométrie d'affichage indépendante de l'espace d'adressage du dispositif de visualisation ces coordonnées sont exprimées en centième d'écran (comme pour GRI-GRI). Le logiciel CLOVIS se charge automatiquement de leur conversion en coordonnées du dispostif physique de visualisation.

- *viscadre* indique si le cadre de la cloture doit être visualisé et *ref_coul* est l'adresse de l'attribut de couleur a utilisé pour le tracé du trait.

- *visfond* indique si la cloture doit être affichée avec un fond dont la couleur est donnée par *ref_coul* (l'adresse de l'attribut de couleur à utiliser).

4.4.4.4 Attributs Gv et Ga par défaut

Comme pour l'aspect, des attributs de géométrie de prise de vue et de géométrie d'affichage sont associés en permanence à la racine du fichier graphique .

Ces attributs par défaut sont :

- pour la prise de vue : une projection parallèle dans le paln z = 0, avec une fenêtre de coin inférieur gauche (0,0) et de coin supérieur droit (1,1).

- pour la géométire d'affichage : un clotûre correspondant à l'ensemble de la surface du dispositif de visualisation avec uniquement la visualisation de son contour.

4.4.5 Attributs d'éclairage

Dans la classe éclairage (**E**) un seul type d'attribut est considéré. Il défini la direction et la couleur d'une source lumineuse. C'est un attribut global, c'est à dire que pour la visualisation d'une structure un seul attribut d'éclairage peut être pris en compte. Dans un souci de cohérence avec la définition de la prise de vue où un attribut peut être attaché à n'importe quel niveau de structure, nous avons choisi de laisser la même liberté pour l'affectation de l'éclairage. La règle de composition des attributs d'éclairage est la suivante :

- une structure (et donc tous les éléments qui la compose) est visualisée en utilisant l'attribut d'éclairage trouvé le premier en explorant l'arborescence vers la racine.

- les attributs d'eclairage situés aux éléments des niveaux inférieurs de la structure visualisée sont ignorés.

La figure IV.31 qui reprend l'exemple du lotissement, illustre l'utilisation d'attributs d'éclairage en conjonction avec des attributs de prise de vue et de géométrie d'affiche.

Le "lotissement" dans son entier est visualisé avec les attributs de prise de vue et d'éclairage (E1) de la scène "village" auquel il est raccroché.

L'élément "maison 1" du lotissement est également affiché avec la même prise de vue, dans une clôture différente (Ga_2). Lors de la construction de cette vue séparée de la maison 1, l'attribut d'éclairage (E_2) qui lui est associé et utilisé. Par contre l'attribut d'éclairage E_3 situé à un niveau inférieur des racines des structures visualisées (respectivement "lotissement" et "maison1") n'est pas pris en compte lors de l'affichage.

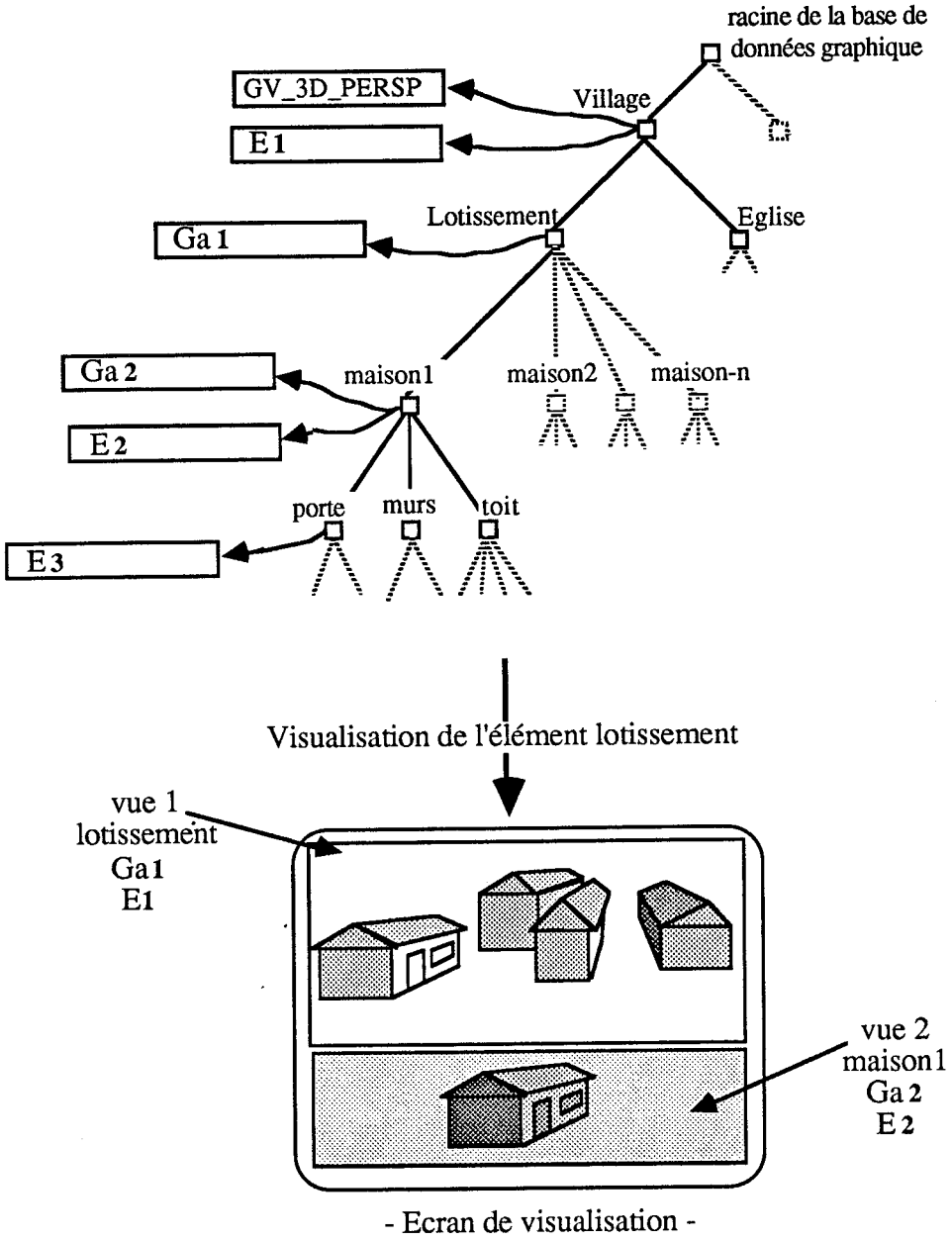
La construction d'un descripteur d'attribut d'éclairage se fait par l'intermédiaire de la primitive

ECLAIRAGE(dx,dy,dz,r,v,b) --> adr_descr

où

- *dx,dy,dz* sont les composantes du vecteur indiquant la direction de la source lumineuse,

- *r,v,b* sont les coefficients exprimant les composantes rouge, verte et bleue de la couleur de la source lumineuse.



- figure IV.31 : prise en compte des attributs d'éclairage dans la structure -

5 LES PRIMITIVES DES QUATRE PROCESSUS DE BASE

Dans cette partie nous présentons les primitives associées aux quatre processus de base de la synthèse d'images et que nous avons définis comme étant : l'ATTRIBUTION, la CONSULTATION, la DESCRIPTION et la VISUALISATION. Dans la conception de ces primitives nous avons été guidés par le souçi que le système de synthèse d'images prenne en charge au maximum la gestion de ces processus. C'est pourquoi, nous avons réduit à l'extrême le nombre de primitives proposées à l'utilisateur, avec quasiment une seule primitive pour chaque processus.

Comme nous l'avons vu au paragraphe 2.3.2, les processus d'attribution, de consultation et

de description concernent les noeuds définis par l'entité de travail courante, alors que le processus de visualisation se réfère aux noeuds de l'entité de visualisation.

Il est donc de la responsabilité du programme d'application de définir à tout instant les entités de travail et de visualisation à l'aide des primitives de structuration logique (définition de cntexte et d'entité) avant d'invoquer un processus quelconque.

5.1 Attribution

5.1.2 La primitive d'attribution

Le processus d'attribution permet au programme d'application d'affecter un attribut aux éléments de l'entité de travail courante.

Son invocation s'effectue par l'appel de la primitive :

ATTRIBUER(adr_att, var_cste)

où :

- *adr_att* : est l'adresse du descripteur de l'attribut à attacher aux éléments constituant l'entité courante,

- *var_cste* : est une constante indiquant si l'attribut est géré par l'application (attribut variable, *var_cste* est égale à la constante prédéfinie VARI) ou pris en charge par CLOVIS (attribut constant, *var_cste* a dans ce cas la valeur CSTE).

Le descripteur d'attribut d'adresse *adr_att* est affecté à chacun des noeuds de l'entité courante. Un noeud de la base de données graphique ne peut posséder à un instant donné qu'un **seul** attribut par classe. Aussi l'affectation à un noeud d'un attribut d'une classe déjà présente a pour effet (voir § 4.4) :

- de substituer l'ancien par le nouveau pour les classes MORPHOLOGIE, Ga, GV, ASPTRAIT , APSSURF et ASPTEXT (et leur sous classes d'aspect) et pour les types GEOM2D_ABS et GEOM3D_ABS des classes GEOM2D ou GEOM3D,

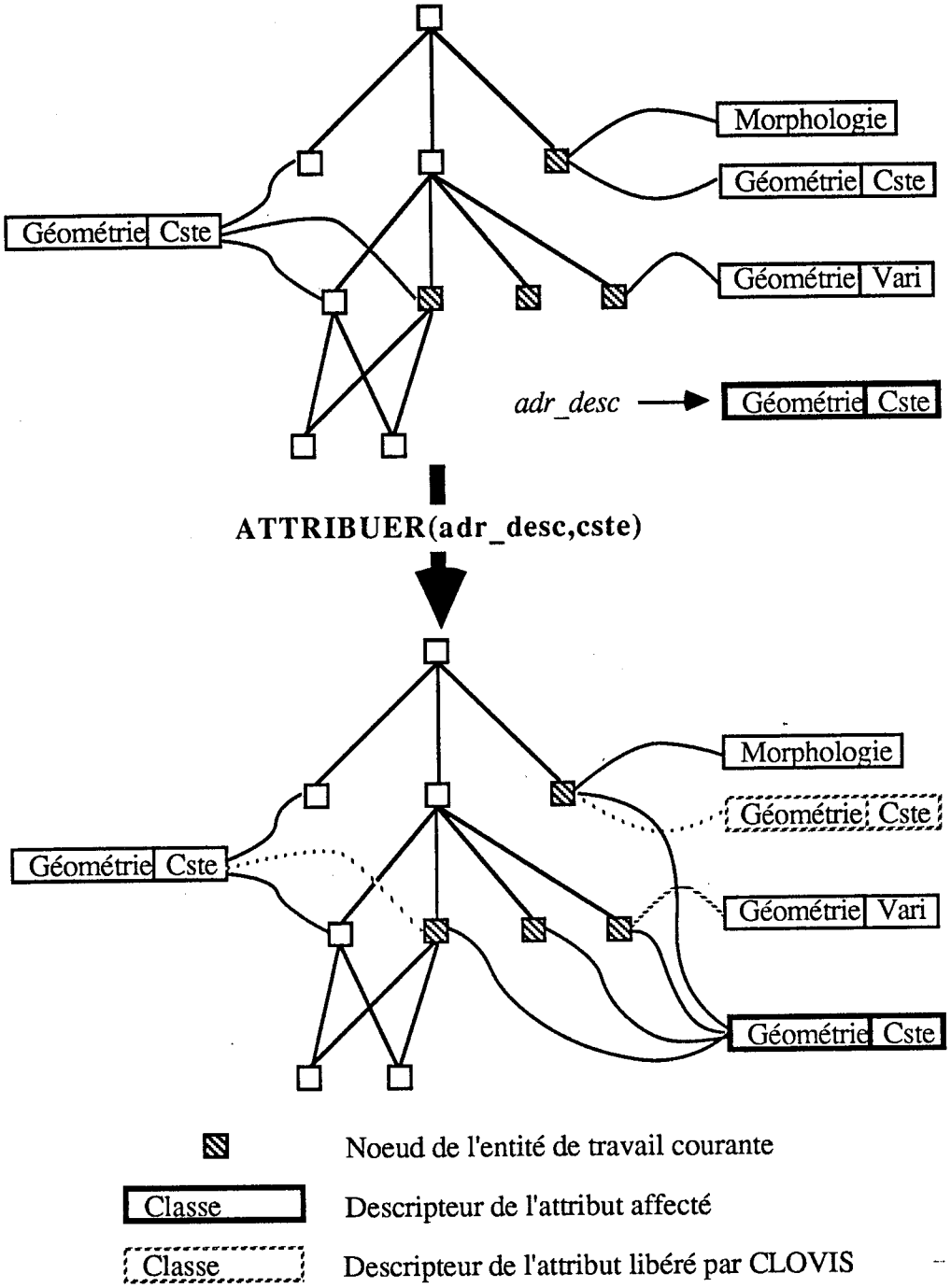
- de substituer l'ancien attribut par le nouveau composé avec l'ancien pour les type GEOM2D_REL, GEOM2D_LOC et GEOM3D_REL, GEOM3D_LOC dans les classes GEOM2D et GEOM3D.

Dans le cas où un attribut de type constant qui a été "décroché" pour être remplacé par un nouvel attribut, n'est plus référencé par aucun élément de la structure arborescente, CLOVIS libère automatiquement l'espace qui lui était alloué. La figure IV.32 page suivante, résume les différents effets de la primitive d'attribution.

Du fait qu'un noeud ne possède qu'un seul attribut par classe, il est possible de supprimer éventuellement un attribut à l'aide de la primitive

SUPPRIMER_ATT (classe_att)

où *classe_att* désigne la classe de l'attribut à supprimer. Si l'attribut supprimé est un attribut constant et n'est plus référencé par aucun élément de la base de données hiérarchique, l'espace qui lui était alloué est, comme pour l'attribut décroché lors d'une attribution, libéré automatiquement.



- figure IV.32 : Affection d'un attribut à l'entité de travail -

5.1.2 Règles d'attribution des attributs à caractère constant ou variable

Le caractère constant ou variable d'un attribut est défini par l'un des champs de son descripteur. Aussi en fonction de la valeur de celui-ci (CSTE ou VARI) l'affectation d'un attribut à la structure s'effectue sous la forme variable ou constant suivant les règles définies dans la table IV.1 qui suit.

champ varcste du descripteur	paramètre var_cste de la primitive ATTRIBUER	
	VARI	**CSTE**
VARI	- affectation du descripteur passé en paramètre d'ATTRIBUER.	- création d'une copie du descripteur passé en paramètre d'ATTRIBUER; et affectation de celle-ci avec le champ varcste à CSTE.
CSTE	----> erreur affectation d'un attribut constant en tant qu'attribut variable non autorisée.	- affectation du descripteur passé en paramètre d'ATTRIBUER.

table IV.1 : règles d'affectation des attributs variables et des attributs constants.

Ainsi, dans le cas ou le type de l'attribut (variable ou constant) correspond avec la valeur du paramètre *var_cste* utilisé pour la primitive ATTRIBUER, CLOVIS affecte directement son descripteur aux éléments concernés. Par contre, pour l'attribution sous forme constante d'un attribut variable, CLOVIS crée automatiquement une copie de l'attribut qui sera gérée sous forme constante. Finalement, dans le cas de l'attribution sous forme variable d'un attribut constant, l'affectation ne peut être effectuée et la primitive ATTRIBUER est sans effet.

5.1.3 Utilisation des fonctions de modélisation des attributs avec la primitive d'Attribution

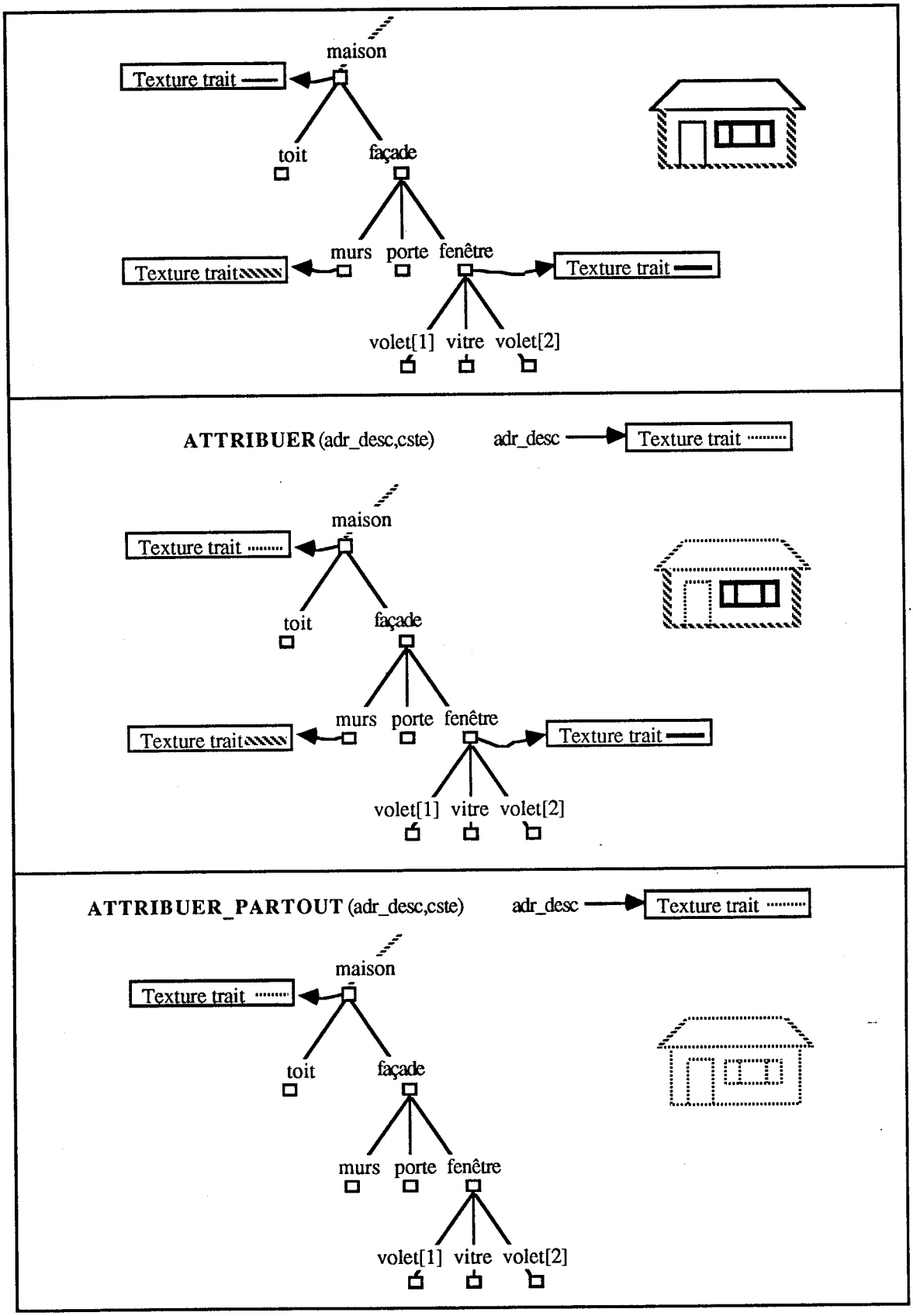
Les fonctions de modélisation des attributs (voir § 4.3) que peut invoquer le programme d'application, laissent au système CLOVIS la charge de la modélisation des descripteurs d'attributs. L'application n'a ainsi pas nécessairement besoin de connaître la structure détaillée de ceux-ci. Cependant, il lui reste la tâche d'appeler le processus d'Attribution avec l'adresse du descripteur afin d'effectuer l'affectation à la structure.

Dans le cas d'attributs constants, le programme d'application peut ignorer jusqu'à l'adresse des descripteurs en invoquant directement dans la primitive ATTRIBUER la fonction de modélisation (par exemple ATTRIBUER (Morphologie, CERCLE_2D(x,y,r), cste)).

C'est pourquoi les fonctions de modélisation des attributs initialisent le champ *varcste* du descripteur d'attribut à CSTE. Par conséquent c'est à l'application de modifier elle-même ce champ si l'attribut ainsi créé doit être affecté sous forme variable.

5.1.4 Attribution "Globale"

Si le processus d'attribution tel que nous l'avons défini dans les paragraphes qui précèdent permet d'associer n'importe quel type d'attribut à n'importe quel niveau de structure, il peut parfois imposer des manipulations lourdes et importantes au programme d'application. C'est le cas en particulier, lorsque l'on souhaite que l'attribut affecté à un élément s'applique à **tous** les éléments de niveaux inférieurs. En effet, de par les règles de composition des attributs que nous avons exposées au § 4.4, il est parfois nécessaire de supprimer tous les attributs de même classe existant aux niveaux inférieurs. Par exemple sur la figure IV.33, si on désire affecter au noeud *maison* une texture pointillée pour le trait qui se répercute dans toute la sous structure il faut



- figure IV.33 : les différents processus d'attribution -

retirer tous les attributs de texture de trait qu'elle contient. Cette tâche de suppression n'est pas toujours très aisée : elle impose de connaître avec précision où se trouvent les différents attributs d'aspect du trait et peut nécessiter la définition successive de plusieurs entités de travail. Aussi afin de simplifier cette opération dont l'utilité nous est apparue à l'usage, avons nous décider de la prendre en charge automatiquement au niveau de CLOVIS. Pour cela, CLOVIS propose un second processus d'attribution invoqué par la primitive

ATTRIBUER_PARTOUT(adr_desc,var/cste)

dont les paramètres ont une signification identique à ceux utilisés par la primitive ATTRIBUER. Les règles régissant l'affectation de l'attribut référencé par *adr_desc* aux noeuds définis par l'entité de travail sont identiques à celles du processus d'attribution vu précédemment. Mais en plus, tous les attributs existant au niveau des sous structures des éléments auxquels l'attribut *adr_desc* est associé et de classe identique à celui-ci sont supprimés.

Pour la suppression d'attributs, on avons également généralisé la primitive SUPPRIMER. Ainsi

SUPPRIMER_PARTOUT(classe_att)

retire tous les attributs de la classe *classe_att* situés dans les sous-structures des noeuds de l'identité de travail courante.

5.2 Consultation

5.2.1 Consultation des attributs élémentaires

Tous les attributs élémentaires associés aux éléments de la base de données graphique peuvent être consultés par l'application. La consultation s'effectue à l'aide de la primitive :

CONSULTER(classe_att) ---> adr_desc

Cette fonction n'a de signification que si l'entité de travail est une référence simple (référence à un seul noeud). Dans le cas contraire, la fonction est sans effet et un code erreur le signale à l'utilisateur.

La primitive CONSULTER renvoie :

- l'adresse du descripteur de l'attribut de classe *classe_att* si il existe pour le noeud référencé,
- NIL dans le cas contraire.

Si le programmeur d'application ne souhaite pas manipuler des descripteurs d'attribut il peut appliquer les fonctions de démodélisation (voir § 4.3) aux descripteurs renvoyés par la primitive CONSULTER. Dans ce but, les fonctions de démodélisation étant propres à chaque type d'attribut, CLOVIS la propose la primitive :

CONSULTER-TYPE-ATT(adr-desc) --> typatt

qui renvoie le type d'un attribut.

5.2.2 Consultation des attributs de structure et d'identité

La primitive CONSULTER ci-dessus permet uniquement la consultation des attributs des classes M,A,G,Gv,Ga,E. Pour les attributs d'identité (I) et de structure (S) inhérants à la structure du fichier graphique, des primitives particulières sont proposées.

Ces primitives sont réparties en deux groupes :

- les primitives qui permettent de consulter l'état des contextes et identité de travail ou de visualisation,

- les primitives qui permettent d'accéder à l'identité et la structure d'éléments quelquonques du fichier graphique.

5.2.2.1 Consultation des contextes et entités de travail et visualisaton

a) Consultation des contextes

La fonction primitive

NB_NIV_CTXT(ctxt_visu_trav) --> entier

permet de connaître le nombre de niveaux de contextes de visualisation (*ctxt_visu_trav* = VISU) ou de travail (*ctxt_visu_trav* = TRAV) empilés. Le contexte initial, contexte par défaut, qui correspond à la racine du fichier graphique est de niveau 0.

L'application a par ailleurs la possibilité de consulter la structure d'un contexte de niveau quelconque à l'aide de la primitive

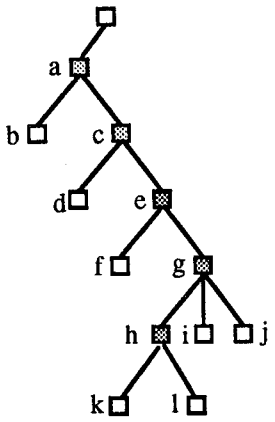
CONS_CTXT(ctxt_visu_trav, niv) --> chaine

où:

- *ctxt_visu_trav* défini la nature du contexte à consulter, travail (*ctxt_visu_trav* = TRAV) ou visualisation (*ctxt_visu_trav* = VISU),

- *niv* est le niveau de contexte consulté, (il doit appartenir à l'intervalle [0 .. NB_NIV_CTXT (ctxt_visu_trav)]),

-*chaîne* contient l'expression alphanumérique qui défini le contexte de niveau *niv* **relativement** au contexte de niveau inférieur.



CONTEXTE_TRAV("a(c)")

CONTEXTE_TRAV("e(g(h))")

NB_NIV_CTXT(trav) --> 2
CONS_CTXT(trav,2) --> "e(g(h))"
CONS_CTXT(trav,1) --> "a(c)"

- figure IV.34 : Consultation de contextes de travail -

b) Consultation de l'entité de travail ou de visualisation

De façon similaire à la consultation des contextes, le programme d'application peut récupérer les informations relatives aux entités de travail ou de visualisation courantes.

Pour ce faire, on utilisera la fonction primitive :

CONS_IDENTITE(idt_visu_trav) --> chaine

où :

- *idt_visu* est une constante (VISU ou TRAV) qui défini la nature de l'entité consultée (de visualisation ou de travail),

-*chaîne* contient l'expression alphanumérique définissant l'entité relativement au contexte courant.

5.2.2.2 Consultation de structure

a) accès à l'adresse d'une structure

La primitive CONS_STRUC permet d'accéder directement à la structure en récupérant l'adresse d'un noeud qui peut ensuite être réutilisée dans l'une des primitives de construction vues au § 3.3. Sa forme est la suivante :

CONS_STRUCT ("référence simple") --> adr_noeud

où *adr_noeud* est l'adresse de l'élément désigné par l'expression alphanumérique passée en paramètre de la primitive et qui doit bien entendu être une référence simple.

Remarque : La consultation de structure n'est possible que sous le contexte de travail. Aussi l'expression de référence utilisée lors de l'appel de la primitive est relative au contexte courant.

b) Consultation de la structure d'un élément

Afin que l'application puisse "accéder" à l'identité et la structure de n'importe quel élément situé sous le contexte de travail, CLOVIS propose, en complément de la primitive CONS_STRUCT, la fonction suivante :

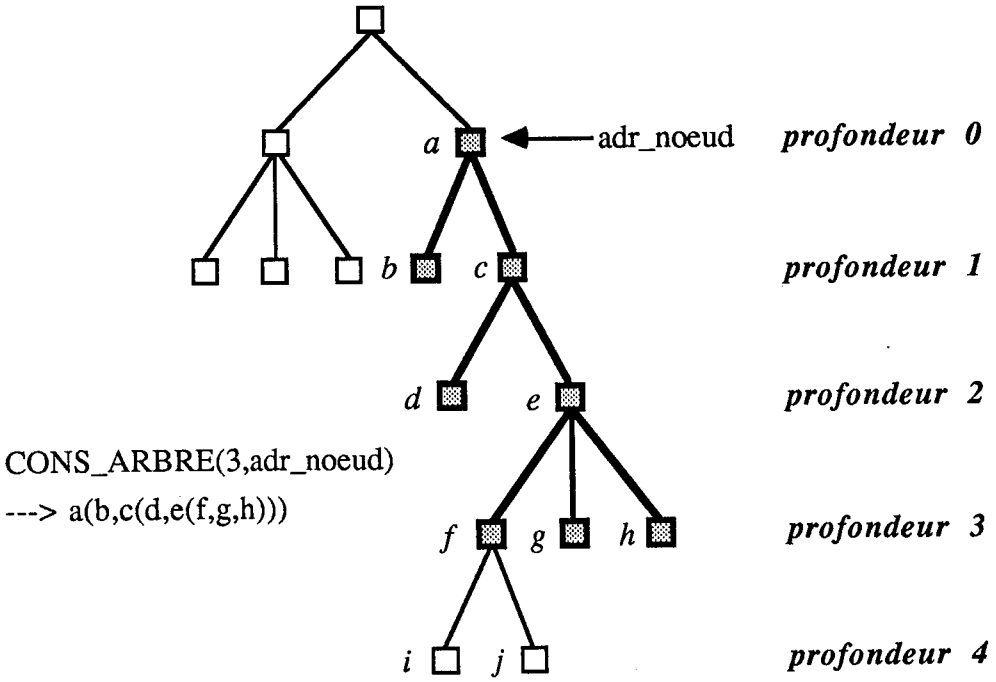
CONS_ARBRE(profond, ard_noeud) --> chaine

où :

- *adr_noeud* est l'adresse de l'élément racine de l'arbre dont on veut connaître la structure et l'identité des éléments qui le composent

- *profond* est un entier qui permet de contrôler la profondeur maximum jusqu'à laquelle on veut pousser la consultation. (La profondeur 0 correspond au niveau du noeud racine, une profondeur négative signifiant que la consultation porte sur tout le sous arbre issu de *adr_noeud*).

En retour, cette primitive renvoie une chaîne alphanumérique correspondant à la structure de racine *adr_noeud* (voir l'exemple de la figure IV.35).



- figure IV.35 : Consultation de la structure d'un sous-arbre -

5.3 Le processus de description

5.3.1 Le principe du processus de description

La description a pour rôle de proposer à l'application un ensemble de processus qui lui permettent de définir **graphiquement** et **interactivement** les différents attributs pouvant être associés aux éléments de la base de données graphiquee.

Initialement, dans les toutes premières versions de CLOVIS, le processus de description était conçu dans un souci de symétrie, comme le pendant intéractif du processus d'attribution. Ainsi les attributs définis lors de son invocation, étaient-ils automatiquement affectés aux noeuds composant l'identité de travail courante.

Mais nous avons opté finalement pour une autre méthode, dans laquelle le processus de description se contente de retourner au programme d'application l'adresse du descripteur de l'attribut défini intéractivement, et ceci sans affectation aux éléments de la base de données graphique. En effet, cette seconde solution nous a semblée plus judicieuse et générale, car elle permet à l'application soit d'affecter immédiatement l'attribut décrit en invoquant la primitive ATTRIBUER , soit de conserver le descripteur de l'attribut pour un usage ultérieur.

5.3.2 Description par construction - Description par référence

A l'intérieur de CLOVIS les processus de description des différents attributs se répartissent en deux grandes catégories :

- les processus de description par **construction** pour lesquels l'utilisateur définit entièrement l'attribut à l'aide des dispositifs de saisie, en suivant des méthodes de construction prédéfinies selon le type de l'attribut.

- les processus de description par **référence** pour lesquels l'attribut retourné est celui d'un élément désigné par l'opérateur sur l'écran de visualisation.

Dans ce qui suit nous présentons les primitives graphiques qui permettent d'invoquer ces divers processus.

5.3.2 Primitive de description par construction

Le programme d'application demande la description par construction d'un attribut à l'aide de la primitive :

DECRIRE_CONST(type_att, dispo, processus) --> adr_desc

où

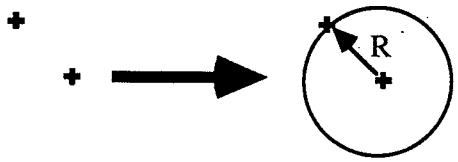
- *type_att* défini le type de l'attribut à décrire,

-*dispo* désigne le dispositif de saisie utilisé (tablette, réticule, clavier...),

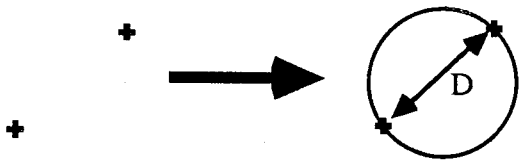
-*processus* défini la méthode de description proprement dite en fonction du type de l'attribut. Par exemple pour le type CERCLE2D, CLOVIS propose trois processus de description par construction standards qui comme nous le montre la figure IV.36 sont respectivement la définition d'un cercle par la donnée de son centre et rayon, la donnée de trois points de sa circonférence, ou la donnée de son diamètre.

-*adr_desc* est l'adresse du descripteur de l'attribut, construit à partir des informations données par l'utilisateur.

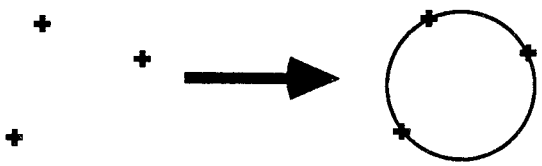
DECRIRE_CONST(CERCLE2D,dispo,CENTRE_RAYON)



DECRIRE_CONST(CERCLE2D,dispo,DIAMETRE)



DECRIRE_CONST(CERCLE2D,dispo,TROIS_POINTS)



- figure IV.36 : Processus de description par construction d'un Cercle 2D -

Remarque : comme pour les fonctions de modélisation des attributs, le champ var_cste du

descripteur de l'attribut est initialisé avec la valeur par défaut CSTE.

Pour chaque type élémentaire d'attribut CLOVIS, propose au moins un processus de description par construction garantissant ainsi la symétrie du système.

5.3.3 Primitive de description par référence

La description par référence permet de définir un attribut par simple désignation d'un élément quelconque affiché sur l'écran. Quelque soit le type de l'attribut le processus est le même : l'utilisateur sélectionne un élément de l'image et CLOVIS renvoie le descripteur de l'attribut avec lequel cet élément a été visualisé.

L'invocation d'une description par référence s'effectue par l'intermédiaire de la primitive :

DECRIRE_REF(classe_att, dispo) --> adr_descr

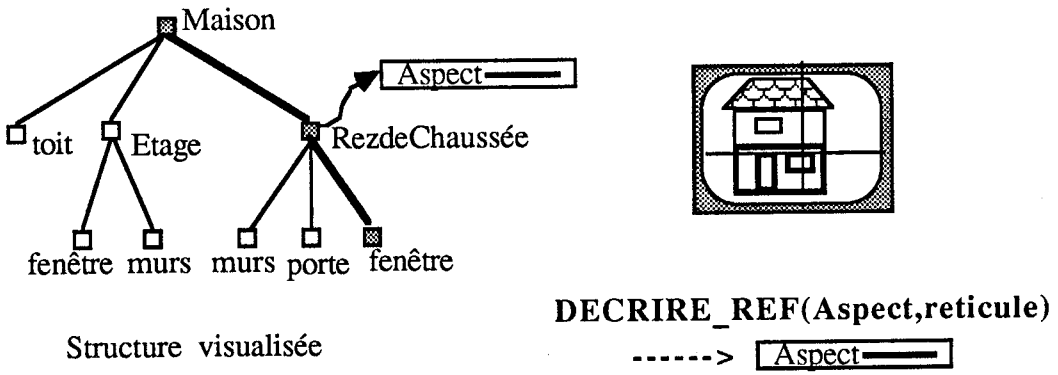
où

-*classe_att* : indique la classe d'attribut considérée,

-*dispo* : défini le dispositif de dialogue utilisé pour la désignation,

-*adr_desc* : est l'adresse du descripteur d'attribut créé par la primitive et retourné à l'application.

Remarque : Il faut bien noter que le descripteur d'attribut dont l'adresse est renvoyée par la primitive DECRIRE_REF, n'est en aucun cas celui de même classe éventuellement associé au noeud correspondant à l'élément désigné. Celui-ci est créé dynamiquement par CLOVIS et équivaut à l'attribut tel qu'il a été **évalué par le processus de visualisation** lors de l'affichage de l'élément. Ainsi il peut être obtenu par la composition d'attributs élémentaires affectés à des éléments de niveaux hiérarchiques supérieurs (fig IV.37). Comme pour la description par construction, le champ var_cste de l'attribut est initialisé avec la valeur par défaut CSTE.



- figure IV.37 : Description par référence -

5.3.4 L'identification

Un cas particulier de description par référence concerne la désignation d'un élément afin de retrouver les informations d'identité et de structure qui lui sont associées. Ce processus particulier est désigné sous le terme d'**identification**, et est invoqué à l'aide de la primitive :

IDENTIFIER(dispo, chaîne_idtf) --> booléen

Cette fonction renvoie la valeur VRAI si le point désigné appartient effectivement à l'un des éléments visualisés, dans ce cas *chaîne_idtf* est une chaîne alphanumérique qui correspond à l'expression désignant l'élément identifié relativement au contexte de visualisation courant.

Dans le cas contraire, IDENTIFIER renvoie la valeur FAUX et le paramètre *chaîne_idtf* n'a alors aucune signification.

dispo indique le dispositif de dialogue utilisé pour l'identification.

Remarque : si le point indiqué par l'utilisateur appartient simultanément à plusieurs objets, l'identification porte sur le premier élément rencontré satisfaisant au critère de recherche. D'autres solutions sont envisageables dans ce cas, comme par exemple indiquer succesivement les objets désignés jusqu'à ce que l'opérateur arrête lui même l'identification, ou bien donner la liste de tous les objets désignés; mais aucune d'elles n'a été effectivement réalisée.

5.4 Visualisation

Le processus de visualisation, au contraire de l'attribution et de la consultation concerne l'entité de visualisation et non l'entité de travail.

L'affichage d'éléments de la base de données graphique est réalisé en invoquant la primitive

VISUALISER(proc_visu)

qui parcourt les structures hiérarchiques définies par l'entité de visualisation et assure la synthèse des différents attributs élémentaires en fonction du processus de visualisation choisi et désigné par le paramètre *proc_visu* .

Le contrôle du processus de visualisation s'effectue en fait sur trois plans simultanément : la prise en compte de l'image éventuellement existante, l'interprétation des attributs géométriques et le degré de réalisme de la présentation. Ainsi, le paramètre *proc_visu* est obtenu en combinant par addition des constantes prédéfinies régissant chacun de ces points.

Le premier point qui concerne la manière dont la visualisation agit sur l'image déjà affichée. Dans cette première réalisation de CLOVIS, deux cas ont été prévus et correspondent aux constantes :

- **efface** indique que la visualisation provoque au préalable un effacement total de la surface de visualisation,
- **ajout** signifie que la visualisation est un simple ajout à l'image déjà affichée qui ne doit pas être modifiée.

Le second point à trait à la manière dont les attributs géométriques sont évalués lors du parcours de la structure pour l'affichage. Il est contrôlé par les deux constantes :

- **geom_abs** signifie que les attributs géométriques des éléments visualisés sont évalués dans le repère du noeud scène auquel ils sont rattachés. C'est à dire l'attribut géométrique utilisé pour la visualisation de la morphologie d'un élément est obtenu en composant tous les attributs géométriques de ces "ancêtres".
- **geom_loc** signifie au contraire que les attributs géométriques ne sont évalués que dans le repère local du noeud racine de la structure visualisée. Ce repère local est soit celui du noeud de l'entité de visualisation ancêtre de l'élément visualisé, soit celui d'un noeud définissant une nouvelle vue (c'est à dire un noeud auquel est associé un attribut Gv ou Ga). L'interêt de ce second type de processus est qu'il permet la visualisation aisée d'objets dans leur repère local qui bien souvent est un repère privilégié pour leur construction.

Le dernier point définissant le processus de visualisation concerne l'interprétation des autres attributs définissant les objets et en particulier leur aspect. Il s'attache au degré de réalisme que doit adopter le processus de visualisation lors de la synthèse de ces attributs. Le programmeur d'application a ainsi le choix entre plusieurs types de présentations définies par les constantes :

- **trait** qui signifie que la présentation doit avoir lieu sous forme de dessin au trait et sans élimination des lignes cachées pour le 3D (présentation "fil de fer" ou "wireframe"),
- **tache** qui indique que lors de l'affichage les attributs d'aspect des surfaces doivent être pris en compte et donc le remplissage des tâches effectué,
- **éclairage** qui signale qu'en plus les attributs d'éclairage doivent être considérés.

D'autre processus sont prévus, en particulier pour l'élimination des partie cachées avec le choix possible entre plusieurs techniques donnant des effets de réalisme varié (simple élimination des lignes cachées, élimination des parties cachées, élimination des parties cachées avec effet de transparence et d'ombres...).

Si la visualisation s'effectue deux fois de suite sur les mêmes éléments et avec le même processus, le second affichage doit pouvoir lorsque c'est possible prendre en compte uniquement les modifications sans réinterpréter complétement la structure. C'est le cas par exemple, si un changement de couleur est effectué et que celle-ci est gérée par des tables au niveau du matériel.

Dans cette optique, la constante **maj** peut être utilisée dans la définition du processus de visualisation. Elle permet de signaler que les modifications apportés ultérieurement aux éléments composant l'entité visualisée devront être prises en compte aux mieux (en évitant au maximum la réévalution des attributs et la reconstruction de l'image) lors de son réaffichage. L'emploi de **maj** provoque lors de l'affichage de la structure un marquage des éléments qui la composent, ce marquage est ensuite mis à jour automatiquement par CLOVIS lors des modifications apportées à cette structure (changement d'attributs, ajout d'éléments...).

Le réaffichage de la structure s'effectue par l'appel de la primitive **REEVAL**. Celle-ci provoque en fonction du marquage des noeuds le parcours uniquement des éléments de la structure nécessitant une réévalution. L'invocation de cette primitive n'a de signification que si le dernier appel du processus de visualisation (primitive VISUALISER) a été effectué avec la constante **maj** et si depuis l'identié de visualisation n'a pas été redéfinie.

6 PRIMITIVES ANNEXES DE CLOVIS

6.1 Initialisation de CLOVIS

Avant toute utilisation de CLOVIS, l'application doit initialiser le système à l'aide de la primitive **INIT_CLOVIS**.

Cette primitive initialise les structures de données de CLOVIS, ainsi que les dispositifs matériels (écran de visualisation, dispositifs de dialogue). Aussi son appel en cours d'utilisation de CLOVIS aura pour effet de remettre à zéro la base de données graphique et d'effacer l'écran.

Suite à l'initialisation de CLOVIS, la base de données graphique est constituée par le noeud racine auquel sont associés les atributs par défaut.Les contextes et entités de travail et de visualisation sont tous initialisés avec cet élément.

6.2 Archivage / récupération d'une structure

CLOVIS offre au programme d'application la possibilité d'archiver, c'est à dire de sauvegarder sous forme de fichier sur un périphérique de mémorisation, des éléments de la base

de données hiérarchique. Une telle sauvegarde conserve la structure complète de l'élément et tous les attributs associés aux noeuds qui le composent. Elle peut aussi être réutilisée directement par CLOVIS pour générer des éléments de la structure arborescente ce qui présente de multiples intérêts :

- possibilité de conserver des objets entre plusieurs utilisations d'une application. La construction d'un objet complexe pourra ainsi être effectuée au cours de séances de travail successives.
- d'échanger des objets structurés entre différentes applications
- de constituer des bibliothèques d'objets prédéfinis qui pourront par la suite être réutilisés par les différentes applications.

La demande d'archivage d'une structure s'effectue par l'intermédiaire de la primitive :

SAUVE_STRUC(adr_struct, nomfich)

où :

- *adr-struct* est la racine de la structure archivée
- *nomfich* est le nom du fichier sous lequel la structure est sauvegardée (si ce nom est déja utilisé CLOVIS le signale à l'opérateur et lui demande de confirmer si il veut "écraser" le fichier existant).

La récupération d'une structure archivée s'effectue par l'intermédiaire de la fonction :

RESTAURE_STRUCT(nom_fich) --> adr_struct

où *adr_struct* est l'adresse de la racine de la structure reconstituée (avec ses éventuels attributs) à partir du fichier d'archive d'identificateur *nom_fich.*

6.3 Gestion des erreurs

CLOVIS est capable de détecter un certain nombre d'erreurs suite à l'invocation des différentes primitives qu'il propose. La gestion de celles-ci est sous l'entier contrôle de l'application qui peut soit utiliser des procédures standard de CLOVIS, soit effectuer elle-même son propre traitement des erreurs. Pour cela elle dispose de la primitive

MODE_ERR(trace : booléen)

qui défini la manière dont CLOVIS va reporter les erreurs.

Dans le cas où le paramètre trace vaut VRAI, chaque fois que CLOVIS détecte une erreur, il la signale immédiatement en affichant un message standard sur l'écran de visualisation (accompagné d'un BEEP!!). Ce mode peut être utile pour la mise au point de programmes.

Dans le cas contraire (trace = FAUX), CLOVIS n'affiche plus aucun message. L'application doit alors effectuer elle-même le contrôle et la gestion des erreurs. Pour ce faire elle a accès à un code erreur positionné par toute procédure ou fonction CLOVIS grâce à la primitive

CONS_ERREUR ---> entier

Cette fonction, renvoie 0 si le dernier appel d'une primitive CLOVIS n'a pas déclenché d'erreur, et sinon le numéro de l'erreur éventuellement détectée. A l'aide de ce mécanisme l'application peut ainsi effectuer un traitement "personnalisé" des erreurs, approprié à ses besoins.

7 CONCLUSION

Dans CLOVIS, le système graphique est entièrement organisé autours de la base de données hiérarchique permettant à l'application de définir les objets à visualiser en structurant les attributs graphiques élémentaires sous une forme arborescente. L'ensemble des primitives du logiciel se répartissent de manière cohérente selon six groupes de fonctionnalités :

- la **structuration** pour construire et accéder à la structure des éléments de la base de données hiérarchique (attributs de Structure et d'Identité),

- la **définition** (ou modélisation) **d'attributs graphiques élémentaires**, permettant au programme d'application de construire les différents types d'attributs des classes M,G,A,Ga,Gv et E,

- la **description** qui permet de construire interactivement des attributs graphiques élémentaires,

- l'**attribution** qui permet d'affecter des attributs élémentaires aux différents éléments de la base de données graphique,

- la **consultation** qui permet au programme d'application de récupérer l'information associée aux différents éléments de la base de données hiérarchique,

- la **visualisation** qui permet l'affichage d'éléments de la base de données graphique en parcourant la structure hiérarchique qui les définit et en composant les différents attributs élémentaires qu'elle regroupe.

La réalisation de ce logiciel nous a permis de valider les concepts que nous avons présentés au chapitre I. En particulier, l'utilisation d'une structure hiérarchique pour la définition des objets graphiques, sa gestion par le logiciel graphique, et la manipulation de celle-ci par l'intermédiaire d'un jeu de primitives de haut niveau prenant en charge les processus de base que sont l'attribution, la description, la visualisation et la consultation, ont permis de définir un système puissant, cohérent, symétrique et ouvert.

Ce premier "prototype" de logiciel graphique pour la visualisation interactive structurée à été utilisé, dans le cadre du projet de C.A.O. de circuits CASCADE mené au laboratoire ARTEMIS, pour la réalisation d'EDICAS [Mart 84], un editeur graphique destiné à l'entrée et la modification de circuits décrits à différents niveaux d'abstraction.

Cependant, si CLOVIS présente un certain nombre de fonctionnalités absentes des logiciels graphiques de base disponibles alors (en particulier les "standards" GKS et CORE-SYSTEM), nous avons pu à l'usage de celui-ci nous rendre compte de certaines limites de notre approche. En effet, si la structuration hiérarchique des données graphiques et l'emploi de primitives de haut niveau pour les processus de base permettent la définition d'un système cohérent au jeu de primitives relativement restreint, l'utilisation du système présente parfois un trop grande rigidité. Celle-ci s'exprime au travers de deux points essentiels qui sont :

le passage obligé par la structuration hiérarchique des attributs graphiques pour la définition et la visualisation d'objets

Ainsi toute visualisation d'objet doit **nécessairement** être précédée par la construction de sa structure, la définition des attributs qui le caractérisent et leur attribution aux différents éléments de la structure. Cette utilisation de la structure hiérarchique, si elle est indispensable pour la définition de scènes complexes, s'avère beaucoup trop contraignante pour répondre aux besoins d'applications simples. Un exemple évident correspond au cas d'applications (ou portions

d'application) désirant uniquement afficher des objets élémentaires qui ne sont pas modifiés par la suite.

le passage obligé pour les entrées par les processus de description d'attributs graphiques

En effet, si la totale prise en charge des processus de description des attributs graphiques par des opérateurs de haut niveau, s'est avérée être un point très positif, elle n'en demeure pas moins, elle aussi, parfois trop contraignante. Les opérateurs de description des attributs graphiques permettent de gérer simplement et efficacement le dialogue entre l'utilisateur et le système graphique, mais ils sont beaucoup moins adaptés pour la définition du dialogue interactif avec l'application elle-même qu'il est pourtant souhaitable d'effectuer le plus souvent graphiquement. Ainsi, il serait parfois nécessaire à l'application d'accéder à des opérateurs d'un autre niveau lui permettant de définir sa propre interface graphique interactive.

Ce sont ces deux principales constatations qui ont motivé la conception d'un second système que nous présentons dans la troisième partie de cette thèse. Cependant avant l'étude de ce système, nous presentons dans le chapitre suivant un certain nombre d'aspects techniques liés à la réalisation et l'implémentation de CLOVIS.

Chapitre V : CLOVIS : Réalisation et Implémentation de CLOVIS

1 INTRODUCTION

L'objet de ce chapitre est de présenter les principaux points liés à l'implémentation de CLOVIS. Ainsi, nous interesserons-nous à l'organisation générale du logiciel, aux principales structures de données utilisées et à la réalisation d'un certain nombre des primitives que nous avons exposées au chapitre précédent.

Bien, sur il n'est pas question d'étudier en détail l'ensemble des logiciels qui constituent CLOVIS. Aussi, nous mettrons l'accent sur les problèmes ayant trait à la structuration hiérarchique des données graphiques et sur lesquels notre travail s'est porté plus précisement. Les autres points, en particulier la prise en compte de différents matériels de visualisation on été étudiés dans le cadre d'autres travaux [Boul 85].

Nous rappelons que le logiciel CLOVIS a été ecrit en langage PASCAL, sur un ordinateur VAX 11/780 fonctionnant sous système VMS. Le choix du langage s'est porté sur le PASCAL car ils nous a semblé le mieux adapté pour la réalisation d'un premier prototype en autorisant une programmation claire et structuréee. De plus, le PASCAL sous VMS intégre un certain nombre de fonctionnalités qui permettent une programmation modulaire (modules avec compilation séparée, inclusion de modules à la compilation (directives "include"), indispensable à la réalisation de logiciels importants.

Ainsi CLOVIS, représente approximativment 10000 lignes de PASCAL, le code objet de la bibliothèque occupant quand à lui environ 120 KOctets.

2 ORGANISATION GENERALE DU LOGICIEL CLOVIS

2.1 L'organisation hiérarchique du logiciel

L'ensemble des logiciels qui composent CLOVIS est organisé de manière hiérarchique en trois principaux modules (fig.V.1).

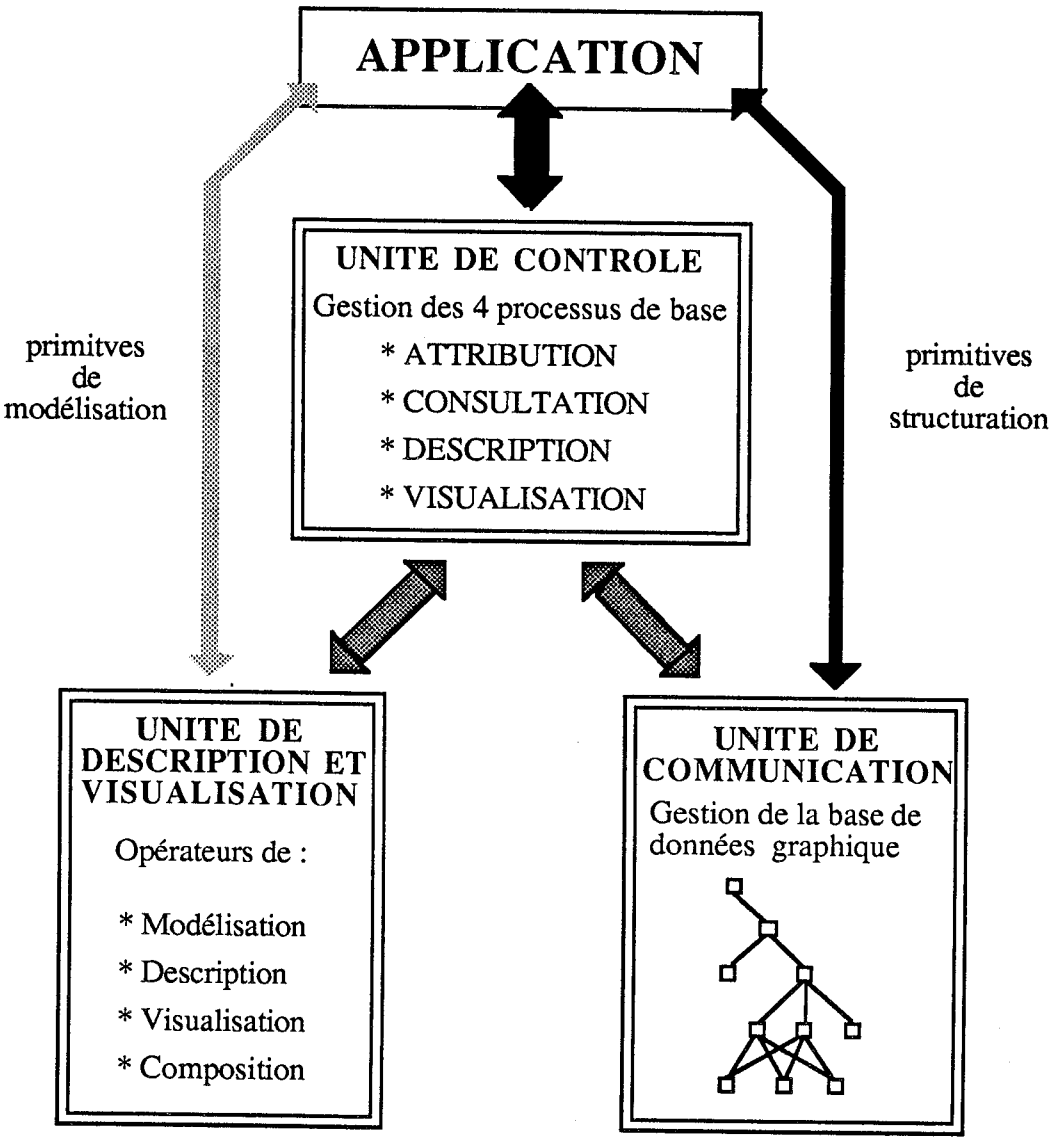
L'**unité de communication** est un module **universel indépendant** tant des applications que du matériel graphique, qui assure la gestion de la base de données graphique hiérarchique. Elle regroupe les primitives de manipulation de la structure hiérarchique (création, destruction, parcours ...) et les primitives de base d'affectation et consultation d'une information **quelconque** (attribut élémentaire) présente dans la base de données.

L'**unité de description et visualisation** est une banque d'opérateurs de base (composition, synthèse, décomposition, visualisation ...) pour les différents attributs graphiques élémentaires que l'application peut associer aux éléments de la base de données hiérarchique. Ces primitives ne sont pas directement accessibles à l'application. Alors que l'unité de communication est entièrement banalisée, cette unité est dependante des attributs considérés et pour certains opérateurs du matériel graphique (en particulier pour les opérateurs de visualisation).

C'est l'**unité de contrôle et d'ordonnancement** qui à partir des opérateurs des unités de niveau hiérarchique inférieur, construit les primitives des quatres processus graphiques de base (ATTRIBUTION, CONSULTATION, DESCRIPTION, VISUALISATION) qui sont invoquées par l'application. C'est elle qui réalise la quasi totalité de l'interface* avec les programmes d'application et assure l'idépendance de celle-ci vis à vis des processus et des opération élémentaires, en proposant un jeu standard de primitives de haut niveau.

* mis à part les primitives de structuration de l'unité de communication et les primitives de modélisaton des attributs de l'unité de description visualisationqui sont directement accessibles.

Comme l'unité de description et visualisation, l'unité de contrôle est dans une certaine mesure dépendante du matériel considéré. En effet si dans bien des cas, un changement de matériel peut simplement consister à la simple modification d'un opérateur terminal de l'unité de description visualisation, il peut parfois conduire à une refonte totale des processus afin d'utiliser au mieux ses possibilités.



- figure V.1 : organisation hiérarchique du logiciel CLOVIS -

2.2 Les conséquences de l'organisation hiérarchique

L'organisation hiérarchique en trois unités présente plusieurs intérêts :

- La **banalisation** des structures de données à l'intérieur de l'unité de communication permet la structuration, l'affectation et la recherche d'attributs graphiques d'un type quelconque. Ainsi, l'indépendance totale de ce module par rapport au matériel et aux attributs graphiques élémentaires considérés garanti l'extensibilité du système. En effet, la prise en compte éventuelle de nouveaux attributs et leur mémorisation dans la base de données graphique nécessite simplement l'écriture des opérateurs les concerant dans l'unité de description et visualisation et éventuellement la modification des processus au niveau de l'unité de contrôle.

- l'unité de contrôle permet d'assurer l'**indépendance** par rapport au matériel en proposant des primitives de haut prenant en charge les processus de base. La **séparation** entre les opérateurs élémentaires (unité de description visualisation) et les problèmes liés à leur application et leur **ordonnancement** (unité de contrôle) doit faciliter l'adaptabilité du logiciel aux spécificités du matériel graphique [Mart 82]. En effet, cette organisation hiérarchique évite le goulot d'étranglement rencontré pour les systèmes organisés en couches (voir chapitre II) et constitué par la fixation a priori de l'interface entre les différents niveaux de logiciel (interface terminal virtuel). En particulier les étapes des processus de visualisation dépendent du matériel utilisé, du type des attributs graphiques considérés, et des performances souhaitées, leur découpage a prori est donc très pénalisant. Au contraire, avec CLOVIS l'ordonnancement du processus de visualisation n'est pas fixé à l'avance.

Cependant, si l'organisation hiérarchique permet d'envisager une réalisation efficace des processus de base, la prise en compte d'un nouveau matériel est une tâche plus complexe que pour un logiciel décomposé en couche et pour lequel il "suffit" d'écrire un "driver".

En effet, la prise en compte d'un nouveau matériel, nécessite l'écriture des opérateurs le concernant au niveau de l'unité de description et visualisation mais peut également imposer la modification de l'unité de contrôle en nécessitant une refonte des processus de base. Toutefois, nous pensons que dans bien des cas le problème n'est pas aussi difficile. Ainsi, pour des matériels aux fonctionnalités similaires seuls les opérateurs élémentaires changent, au niveau de l'unité de contrôle, il n'y a pas de modification des processus. Le problème est alors équivalent à celui de l'écriture d'un "driver", avec la possibilité éventuelle de conserver tels quels certains des opérateurs de l'unité de description et visualisation indépendants du matériel (par exemple des opérateurs de modélisation ou de composition).

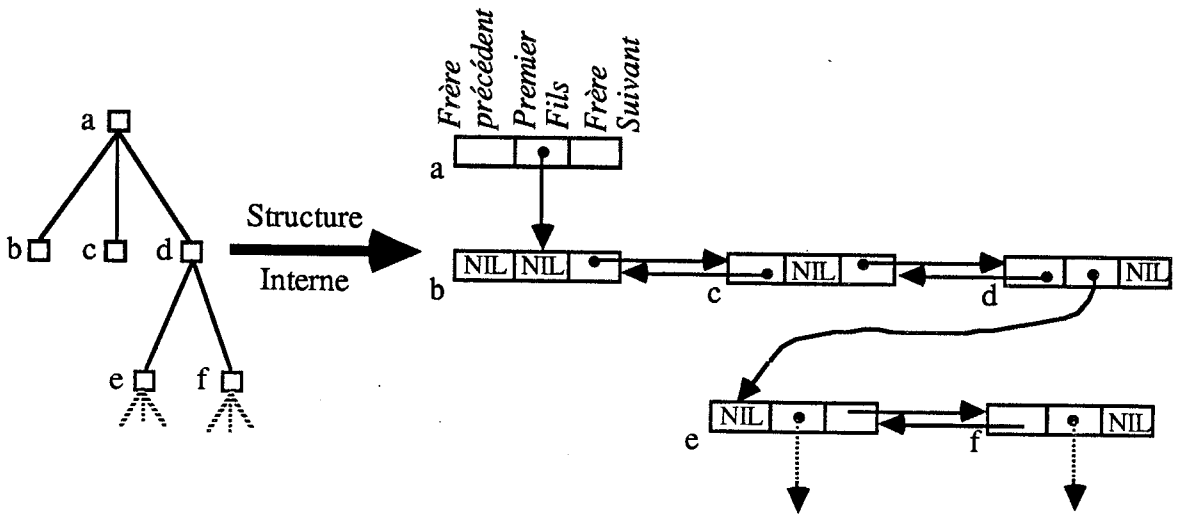
Dans les paragraphes qui suivent nous présentons plus en détail ces trois unités. Nous accordons une importance particulière à l'unité de communication qui a été l'objet plus spécifique de nos travaux. Elle est au coeur du système, puisque c'est autours de la structure de données hiérarchique qu'elle gère que sont organisées les primitives proposés par l'unité de contrôle. Les opérateurs qu'elle fourni sont à la base de la construction de celles-ci, l'invocation des opérateurs de l'unité de description visualisation s'effectuant toujours en relation avec l'exploitation de la base de données graphique.

3 L'UNITE DE COMMUNICATION

3.1 La représentation interne des éléments

3.1.1 Les différents types de noeuds

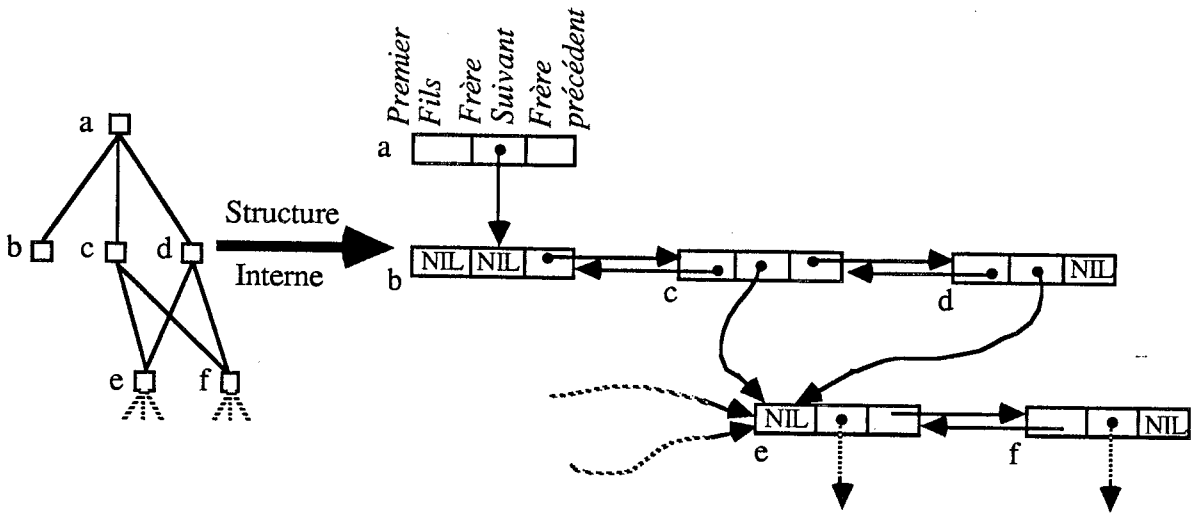
La base de données hiérarchique gérée par 'unité de communication est définie comme étant une structure arborescente dont la profondeur et le nombre d'éléments par niveau sont quelconques (au libre choix du programme d'application) et qui peut évoluer dynamiquement (possibilité de rajouter, supprimer des éléments à n'importe quel niveau). Aussi, afin de satisfaire ces conditions, la structure de données utilisée pour modéliser les noeuds de l'arborescence sera bien entendu à base de listes chainées, où comme nous le montre la figure V.2 pour chaque noeud élément, on mémorisera à l'aide de pointeurs l'adresse de ses deux frères ainsi que celle de son premier fils.



- figure V.2 : Représentation de la structure à l'aide de listes chaînées -

Mais si cette représentation est bien adaptée pour une véritable arborescence, elle se révèle incomplète et inadéquate dans le cas de structures partagées (instances). En effet la modification de celles-ci doit se répercuter vers tous les éléments qui les référencent. Une telle structure de données imposerait une mise à jour éventuelle de nombreux liens (comme nous le montre la figure V.3 ci-contre pour le cas de la suppression d'un noeud partagé) dont la gestion ne serait pas des plus aisée.

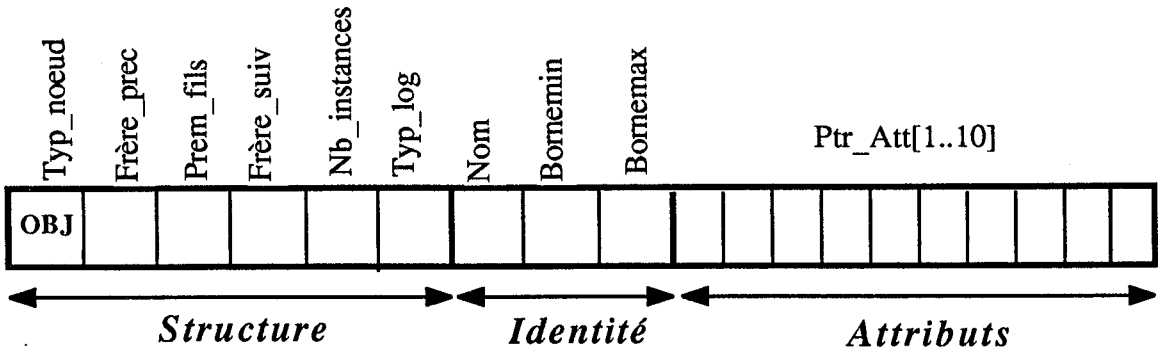
C'est pourquoi, afin d'adapter ce mécanisme au cas particulier de CLOVIS, a-t-il été nécessaire de distinguer deux types de noeuds : les **noeuds objets** qui définissent une arborescence classique et les **noeuds instances** qui servent à partager une sous-structure.



- Figure V.3 : Problème du partage de Sous-Structures -

3.1.2 Les noeuds objets

Les noeuds de la première catégorie permettent de définir la décomposition d'objets en sous objets de manière arborescente. C'est ce type de noeud qui est utilisé lors de la création d'une nouvelle structure. Le descripteur de ces noeuds se décompose en trois parties qui correspondent (figure V.4) aux informations de structure, d'identité et aux attributs élémentaires des objets.



- figure V.4 : Structure interne des noeuds de type objet -

La **structure** est définie par :

- *Typ_noeud* qui indique la nature du noeud. Deux valeurs sont possibles pour *Typ_noeud* selon que l'élément est raccroché ou non à un noeud, ce sont :

  - **OBJ_LIBRE** : qui signifie que le noeud est la racine d'un objet libre,
  - **SOUS_OBJ** : qui signifie que le noeud est attaché à une structure et constitue donc un sous-objet.

- *Frère_prec*, *Frère_suiv* et *Prem_Fils* sont des pointeurs référençant respectivement le frère précédent, le frère suivant et le premier fils de l'élément,

- *Nb_instances* correspond au nombre de noeuds de type instance (voir § suivant) utilisés pour partager la structure dont l'élément est la racine.

-*Typ_log* indique l'appartenance ou non du noeud à l'une des sous structures définies par les différentes entités logiques (contextes ou identités). Les valeurs possibles pour ce champ sont :

  - TRAV si le noeud fait partie de l'une des entités logiques de travail,

  - VISU si le noeud fait partie de l'une des entités logiques de visualisation,

  - NUL sinon.

remarque : une noeud peut appartenir simultanément à une entité logique de travail et à une entité logique de visualisation. Dans ce cas, son type logique est définie par la combinaison des deux constantes TRAV + VISU.

L'**identité** du noeud est donnée par :

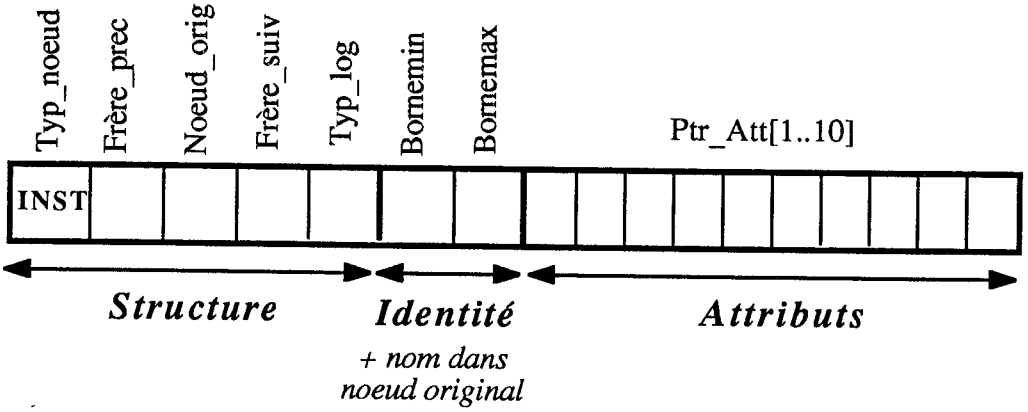
- *Nom* qui est la chaîne alphanumérique permettant de désigner l'élément,
-*Bornemin* et *Bornemax* qui sont les bornes d'indiçage inférieures et supérieures du noeud.

La dernière partie du noeud concerne la gestion des **attributs élémentaires** associés à l'élément. Elle est composée de dix emplacements (*ptr_att[1:10]*) destinés à accueillir des pointeurs vers les descripteurs des divers attributs élémentaires associés à l'élément.

On remarquera qu'au niveau des noeuds de la structure arborescente les descripteurs d'attributs élémentaires sont traités de façon complètement banalisée : un champ peut accueillir un pointeur vers un descripteur d'attribut absolument quelconque, quels que soient sa classe et son type. Cette possibilité assure l'indépendance de l'unité de communication vis à vis des attributs considérés et offre une grande souplesse pour l'adaptation de CLOVIS à des cas particuliers ou pour l'adjonction de nouveaux attributs.

3.1.3 Les noeuds instances

La deuxième catégorie de noeud est utilisée lors de la création d'instances (à l'aide de la primitive INSTANCE ou lors d'un éclatement de structure suite à une référence indicée) et permet le partage de sous structures. La figure V.5 en schématise l'organisation qui est similaire à celle des noeuds de type objets.



- figure V.5 : Structure interne des noeuds de type instance -

Les attributs de **structure** sont définis par :

- *Typ_noeud* qui indique le type de l'élément, dans ce cas il s'agit d'un noeud instance (Typ_noeud = INS)

- *Frère_prec* et *Frère_suiv* qui sont comme pour les noeuds objets des pointeurs vers les noeuds frères précédent et suivant.

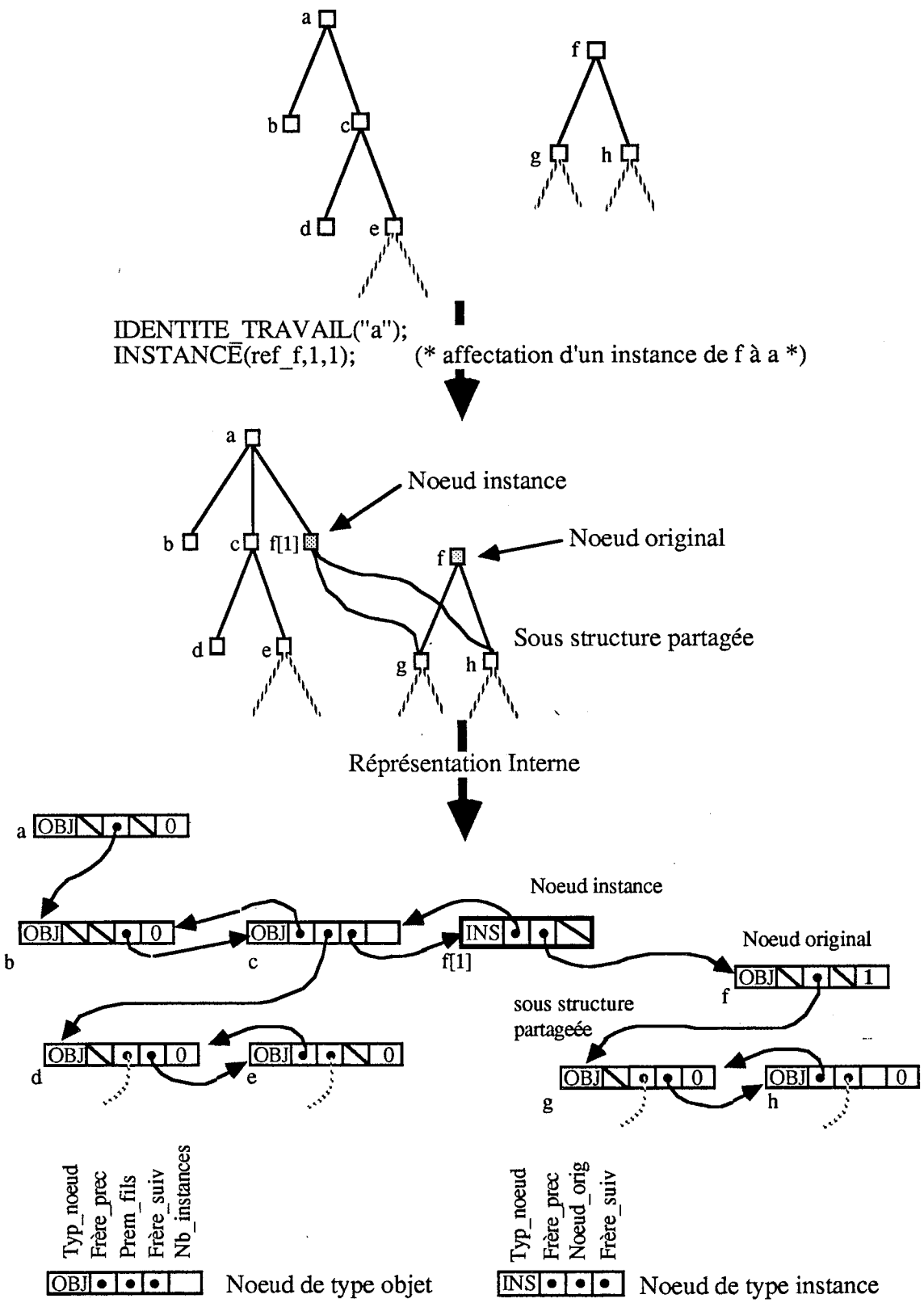
- *Noeud_orig* est un pointeur vers le noeud "original", racine de la structure que le noeud instance sert à partager (fig. V.6). Ainsi l'accès à la sous structure partagée s'effectue **indirectement** en passant par le noeud ayant servi à sa construction (c.a.d. le noeud qui a été passé en paramètre de la primitive INSTANCE ou qui a été éclaté suite à une référence indicée). Lorsque un noeud de type instance est créé et qu'un lien vers le noeud original est établi, le compteur du nombre d'instances de ce dernier (champ *nb_inst* du descripteur des noeuds de type objet, voir § 3.1.3) est incrémenté.

Ce mécanisme d'indirectrion pour l'accès à la sous structure du noeud instance est totalement transparent pour l'utilisateur. Il permet de répercuter vers **toutes** les instances toute modification apportée aux structures qu'elles partagent, de façon simple et immédiate.

Comme pour les noeuds de type objet *typ_log* indique l'appartenance ou non du noeud à une entité logique de travail ou de visualisation.

Les attributs d'**identité** des noeuds de type instance se limitent au bornes d'indiçage (*Bornemin* et *Bornemax*). Le nom du noeud est en effet identique à celui du noeud original où il est conservé.

En ce qui concerne les **attributs élémentaires** qui permettent de particulariser l'instance leur gestion est la même que pour ceux des noeuds de type objet.



- Figure V.6 : Partage de sous structures avec les noeuds de type Instance -

3.2 Gestion interne des noeuds

Dans les paragraphes qui suivent, nous présentons les actions de base (ajout, suppression d'un élément) de gestion de la structure hiérarchique. Utilisées de façon interne à l'unité de communication, elles interviennent dans les procedures det plus haut niveau implémentant les différentes primitives de structuration.

3.2.1 Le Noeud Courant

Pour la gestion interne du fichier graphique, l'unité de communication conserve l'adresse d'un noeud courant et de son père dans deux variables globales (respectivement *noeudcour* et *pèrecour*). Celles-ci sont positionnées par les primitives de parcours d'arborescence que nous presentons au paragraphe 3.4.

Le noeud courant sert de contexte aux primitives de bas niveau que nous présentons ci-dessous (elles s'appliquent implicitement à ce ce noeud) .

3.2.2 Ajout d'un noeud

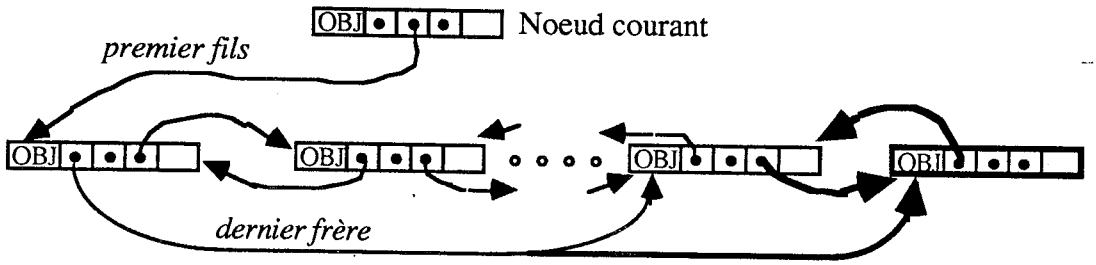
La primitive interne à l'unité de communication

INS_NOEUD(refnoeud)

réalise l'insertion sous le noeud courant de la sous-structure dont la racine est le noeud d'adresse *refnoeud*. Cette opération d'insertion intervient lors de la création, la copie, l'attachement et l'instanciation de structures.

L'ajout d'une sous-structures'effectue en inserant son noeud racine (*refnoeud*) queue de la liste des fils du noeud courant. Ce choix d'une insertion en fin plutôt qu'en tête de liste se justifie du fait de l'utilisation de cette opération d'insertion de noeud lors de la duplication de structure (primitive COPIE). En effet la structure originale est copiée de façon récursive à partir d'un parcours préfixé (RGD), aussi l'insertion des noeuds créés en queue de liste s'impose afin de produire une arborescence en tout points identique à l'arborescence originale. (Une insertion en tête de liste aurait produit une inversion de l'ordre des éléments de chaque niveau).

Afin de minimiser les accès lors de l'ajout d'un élément (parcours séquentiel de tous les fils du noeud courant), la liste des noeuds d'un niveau est en fait organisée sous forme d'anneau (fig. V.7). Le premier élément de la liste permet d'accéder directement au dernier grâce à son champ *frère-prec* qui contient l'adresse de celui-ci.



- figure V.7 : Insertion d'un noeud -

3.2.3 Supression du noeud courant

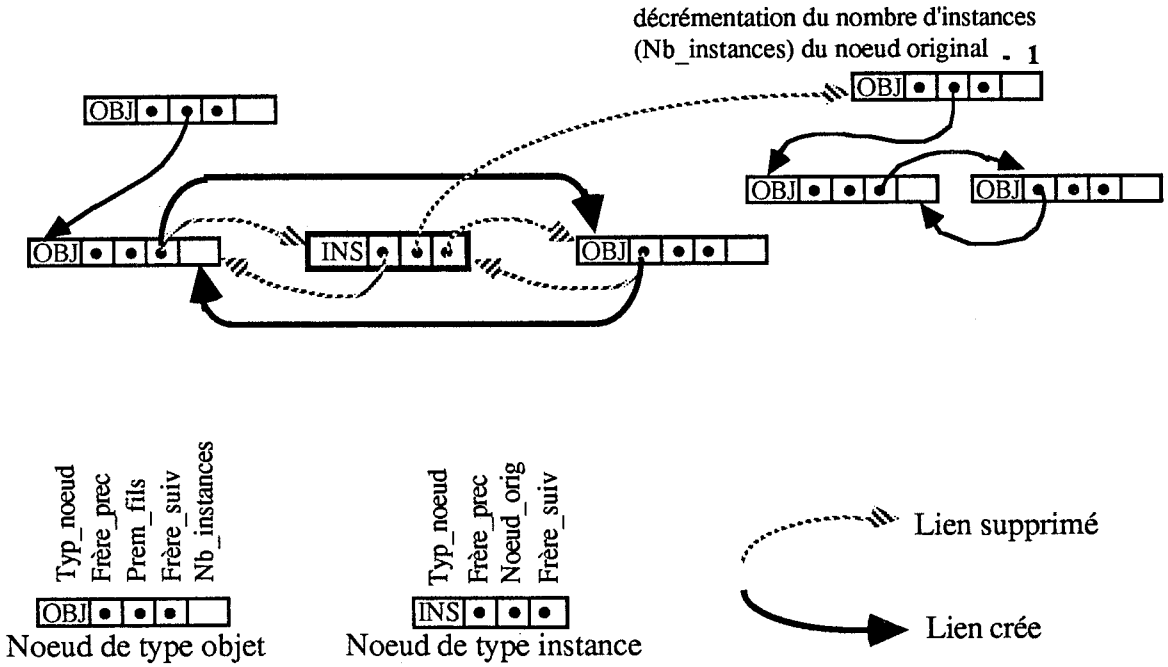
La suppression du noeud courant consiste au retrait de celui-ci de la liste de ses frères* et en la libération éventuelle de l'espace qui lui est alloué. Elle est effectuée à l'aide de la primitive

DEL_NOEUD

qui est utilisée dans l'unité de communication pour la réalisation des primitives de destruction de structure (DEL_STRUCTURE, VIDER_IDENTITE, DETRUIRE_IDENTITE).

* C'est pourquoi en plus du noeud courant il est nécessaire de conserver son père *pèrecour* afin de mettre à jours les liens dans le cas où le noeud supprimé est en tête de liste.

Deux cas sont à distinguer selon que le noeud à supprimer est de type *objet* (OBJ) ou de type *instance* (INS).



- figure V.8 : Destruction d'un noeud de type Instance -

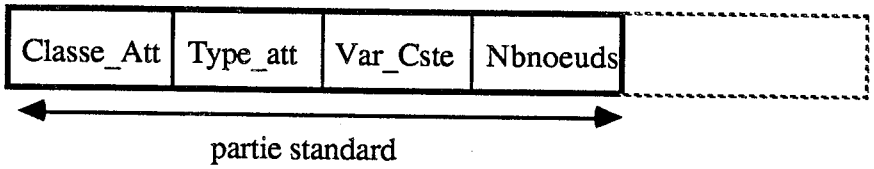
Si le noeud supprimé est de type *instance*, l'unité de communication le retire de la liste des éléments de son niveau, et libère l'espace mémoire qui lui est alloué. Cependant avant de procéder à la destruction du descripteur du noeud, le champ *Nb_instances* du noeud original référencé par l'instance est décrémenté (fig V.8).

Si le noeud courant est de type *objet*, il est bien entendu retiré de la liste des éléments de son niveau, mais après, il n'est réellement supprimé (avec libération de l'espace mémoire alloué à son descripteur) que si plus aucune instance ne le référence (c.a.d. si son champ *Nb_instances* est égal à 0). Dans le cas contraire, s'il partage sa sous structure avec un ou plusieurs noeuds instances, il est conservé par CLOVIS, mais est marqué comme objet libre n'étant plus directement rattaché au fichier graphique (champ *Typ_noeud* = OBJ_LIBRE).

3.3 Gestion des attributs au niveau de l'unité de communication

3.3.1 Les descripteurs d'attributs banalisés

Comme nous l'avons vu au début de ce chapitre, l'unité de communication de CLOVIS, considère les attributs graphiques de façon entièrement **banalisée**. C'est à dire qu'elle les traite de manière totalement identique quel que soit leur classe et type. Pour cela l'unité de communication ne prend en compte que la partie commune à tous les descripteurs d'attributs (fig V.9), et accède à ceux-ci par l'intermédiaire de pointeurs banalisées (définis comme un type pointeur sur des structures correspondant à la partie standard des attributs (type *ptr_desc*)).



- Figure V.9 : Descripteur d'Attribut Banalisé pour l'unité de communication -

Ainsi, seule l'information nécessaire à la gestion mémoire des descripteurs d'attributs est "visible" pour l'unité de communication qui n'a pas à ce soucier de l'information spécifique propre à leur modélisation. Il s'agit, rappelons le, de :

- *Classe_Att* et *Type_Att* qui définissent la classe et le type de l'attribut,

- *Var_Cste* qui définit si l'attribut est constant (entièrement géré par CLOVIS) ou variable (directement accessible par l'application),

- *Nbnoeuds* qui indique le nombre d'éléments (noeuds) de la base de données hiérarchique qui se partagent l'attribut.

3.3.2 Les primitives de gestion des attributs

Les primitives de gestion des attributs que propose l'unité de communication permettent d'affecter, de rechercher ou de supprimer un attribut quelconque du noeud courant positionné automatiquement lors du parcours de structures (voir § 3.4). Ces primitives sont destinées à l'unitité de contrôle pour la réalisation des processus de base, ellles évitent à cette dernière la manipulation directe des descripteurs de noeuds et d'attributs.

3.3.2.1 Affectation d'un attribut

La primitive

AFF_ATT (adr_desc)

permet d'affecter un attribut au noeud courant.*adr_desc* est un pointeur (banalisé) qui contient l'adresse du descripteur de l'attribut.

L'unité de communication recherche si le noeud courant possède déjà un attribut de la même classe que celui référencé par *adr_desc*. Dans ce cas, le descripteur de l'attribut existant est "décroché" et remplacé par le nouveau descripteur. Le nombre de références de l'ancien attribut (champ *Nbnoeuds*) est décrémenté. Si celui-ci est de type constant (champ *var_cste* = CSTE) et n'est plus référencé par aucun élément (*Nbnoeuds* = 0) l'espace mémoire alloué à son descripteur est automatiquement libéré par CLOVIS.

S'il n'existe pas d'attribut de la même classe que celui affecté, l'adresse du descripteur de ce dernier est rangée dans la première position libre. (Dans le cas où il n'y aurait plus aucune place disponible pour accueillir le pointeur vers le descripteur d'attribut, un code d'erreur le signale à l'utilisateur).

3.3.2.2 Suppression d'un attribut

La suppression de l'un des attributs du noeud courant s'effectue à l'aide de la primitive

DEL_ATT(classe_att)

où *classe_att* désigne la classe de l'attribut à détruire. La destruction éventuelle du descripteur de l'attribut supprimé s'effectue de la même manière qu'au paragraphe précédent : si l'attribut est de type constant est n'est plus référencé par aucun noeud de la structure hiérarchique.

3.3.2.4 Recherche d'un attribut

L'unité de contrôle peut également accèder aux attributs du noeud courant au travers de la fonction

ptr_desc <--- RCH_ATT(classe_att)

qui renvoie l'adresse de l'attribut de classe *classe_att* du noeud courant si il existe, NIL sinon.

3.4 Les primitives de parcours de structure

3.4.1 Parcours de sous arbre et parcours d'entité logique

Les primitives de parcours permettent de traverser une structure hiérarchique en appliquant pour chaque élément rencontré différentes actions selon la nature de celui-ci (noeud terminal ou non) et le sens du parcours (ascendant ou descendant). Ces primitives sont **au coeur** du système CLOVIS. Accessibles à l'unité de contrôle pour la construction des différents processus de base, elles sont également utilisées de façon interne par l'unité de communication pour la réalisation des primitives de structuration des données graphiques directement destinées à l'application.

Deux types de primitives existent selon qu'elles assurent le parcours d'une sous arborescence complète à partir d'un noeud donné, ou la traversée d'une sous structure extraite après analyse d'une expression utilisée par l'une des primitives de structuration (définition de contexte, d'identité...).

Nous présentons dans les paragraphes qui suivent ces différentes primitives, ainsi que les structures de données internes à l'unité de communication nécessaires à leur réalisation.

3.4.2 Parcours d'une sous arborescence complète

La primitive de l'unité de communication **PAR_ARBRE** permet le parcours préfixé (Racine, Gauche, Droite) de toute la sous arborescence dont le noeud courant est la racine.

Sa forme est la suivante :

PAR-ARBRE (traiterac, actiond,actionf, actiona)

actiond, actionf et *actiona* sont des paramètres de type procédure et désignent les actions qui seront à appliquer lors de la traversée de l'arborescence, respectivement :

- aux noeuds non terminaux parcourus en descendant dans la structure,

- aux noeuds feuilles,

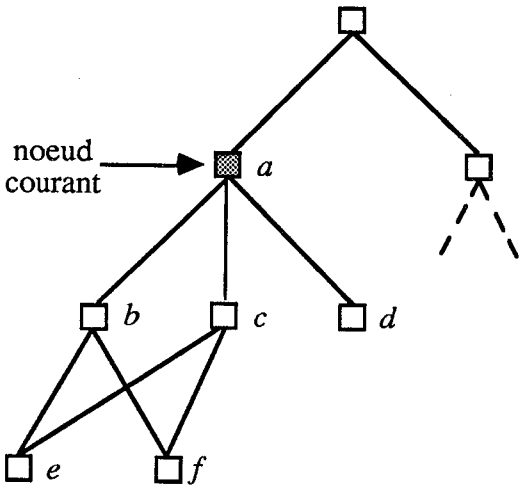
- aux noeuds non terminaux parcourus dans l'ordre ascendant.

Si le booléen *traiterac* a pour valeur VRAI, le noeud racine est pris en compte lors du parcours et est traité de façon analogue aux autres éléments par les procédures *actiond* et *actiona* ou *actionf*. Dans le cas contraire (*traiterac* = FAUX) le noeud racine n'est pas considéré pour les différentes actions invoquées en paramètre de PAR-ARBRE.

La figure V.10 page suivante montre les séquences d'appels engendrées lors de l'exécution de la primitive de parcours de structure.

PAR_ARBRE(Vrai, ACTION_D, ACTION_F, ACTION_A)

engendre les appels suivants :



ACTION_D (a)
ACTION_D (b)
ACTION_F (e)
ACTION_F (f)
ACTION_A (b)
ACTION_D (c)
ACTION_F (e)
ACTION_F (f)
ACTION_A (c)
ACTION_F (d)
ACTION_A (a)

- figure V.10 : Parcours d'une sous-arborescence complète -

La traversée de la structure arborescente est réalisée à l'aide d'un algorithme itératif qui utilise un mécanisme de pile pour explicitement sauvegarder ou restituer l'environnement lors du parcours descendant ou ascendant de l'arbre. A la descente d'un niveau, on empile le noeud courant, la remontée au niveau supérieur s'effectuant en dépilant le noeud courant.

On trouvera, ci-contre, la présentation sous forme algorithmique de la primitive PAR-ARBRE.

La fonction PREMIERFILS permet de traiter les noeuds de type *instance* (INS) de manière analogue aux noeuds de type *objet* (OBJ). En effet, lors du parcours de la structure l'élément correspondant au premier fils d'un noeud instance est en fait obtenu de manière indirecte comme étant le premier fils du noeud objet référencé par le champ *orig* de son descripteur. (voir § 3.1.3)

On remarquera qu'il est possible d'interrompre à tout instant le parcours descendant d'une branche de l'arbre à l'aide de la variable globale *Arret_branche*. L'abandon d'une branche peut ainsi être obtenu en initialisant à VRAI cette variable booléenne dans la procédure *ACTIOND*. Ce type d'interruption de parcours peut être utilisé lors de la visualisation d'une structure afin d'éviter l'évaluation de sous arborescences ne provoquant pas de modification de l'image.

De la même manière il est également possible d'interrompre de façon définitive le parcours d'arbre. En positionnant la variable globale *Arret_Parcours* à VRAI, les procédures *ACTIONA*, *ACTIONF* ou *ACTIOND* peuvent demander à tout instant l'arrêt de la traversée de la structure hiérarchique. Une telle interruption est souhaitable par exemple pour un processus d'identification, où le parcours d'arborescence cesse dès que l'élément désigné est rencontré.

Remarque :

Le parcours d'arborescence aurait pu être réalisé à l'aide d'un algorithme récursif plus simple et utilisant moins de code (en effet il n'aurait pas à réaliser la sauvegarde et la restauration explicite de l'environnement). Nous lui avons préféré la solution itérative qui permet une plus grande efficacité, le cout de la récursivité dependant trop du compilateur.

```
fonction PREMIERFILS(n : ptr_noeud) ----> ptr_noeud
debut
   si n->typ_noeud = OBJ alors     {* noeud de type objet, sous objet *}
      retour(n->premfils)
   sinon                {* noeud de type instance *}
      retour((n->orig)->premfils)
fin

procedure PAR_ARBRE(traiterrac : booléen;  ACTIOND,ACTIONF,ACTIONA : procedure);
var rac: ptr_noeud;
debut

   arrêt_branche = FAUX;
   stop_parcours = FAUX;
   rac = noeudcour;
   si PREMIERFILS(noeudcour) = NIL alors {* la racine est une feuille *}
      si traiterrac alors ACTIONF
   sinon                                    {* la racine n'est pas une feuille *}
      si traiterrac alors ACTIOND;
      si arrêt_parcours alors retour();
      EMPILE(noeudcour);
      noeudcour = PREMIERFILS(noeudcour);

      répéter
         si noeudcour = NIL alors   {* tous les noeuds fils du noeud sommet de pile
                                    ont été traités : noeud ASCENDANT *}
            noeudcour = DEPILE();

            noeudsuiv  = noeudcour->frèresuiv;
            ACTIONA;
            si arrêt_parcours alors retour();
            noeudcour = noeudsuiv;

         sinon si PREMIERFILS(noeudcour) ≠ NIL alors
                        {* noeud non terminal noeud DESCENDANT *}
            ACTIOND;
            si arrêt_parcours alors retour();
            si arrêt_branche alors  {* arrêt du parcours de la branche *}
               noeudcour = noeudcour->frèresuiv;
               arrêt_branche = FAUX
            sinon
               noeudcour = PREMIERFILS(noeudcour)
               EMPILE(noeudcour)
            finsi

         sinon {* PREMIERFILS(noeudcour) = NIL, noeud FEUILE *}
            noeudsuiv = noeudcour->frèresuiv;
            ACTIONF;
            si arrêt_parcours alors retour();
            noeudcour = noeudsuiv
         finsi
      jusqu'à ( SOMMET_PILE() = rac et  noeudcour = NIL);
                              {* tous les fils de la racine ont été traités *}
      noeudcour = DEPILE();
      si traiterrac alors ACTIONA;
   finsi
fin
```

- Algorithme de parcours d'une structure hiérarchique -

3.4.3 Parcours d'une sous arborescence extraite après analyse

Les primitives de parcours que nous présentons ici, sont celles qui permettent de traverser les sous-strucutres correspondant aux entités logiques, contextes et identités de travail et de visualisation.

Ces sous structures sont décrites à partir des expressions de référence utilisées comme paramètres des primitives de création de contextes et d'identités que nous avons étudiées au chapître IV. Aussi, afin que leur parcours puisse être effectué **efficacement**, sans rechercher à chaque fois l'ensemble des noeuds désignés par l'entité logique, les séquences d'accès correspondantes sont "**extraites**" et mémorisées dans les structures de données de l'unité de communication. Celles-ci sont générées à la définition d'une entité logique, lors de l'étape d'analyse qui vérifie si l'expression alphanumérique utilisée est correcte tant du point de vue syntaxique que sémantique. Elles peuvent ensuite être **réutilisées** à chaque parcours durant toute la "durée de vie" de l'entité logique correspondante.

Dans ce qui suit nous présentons les structures de données utilisées pour mémoriser ces séquences d'accès et les primitives de parcours qui les exploitent. La manière dont celles-ci sont générées au moment de l'analyse des expressions fournies est évoquée au paragraphe 3.5 qui suit cet exposé et étudiée plus en détail en annexe 1.

3.4.3.1 Les structures de données pour le parcours

Suite à l'analyse, le parcours préordre RGD correspondant aux sous-structures définies par la chaîne est conservé explicitement sous forme "linéaire" dans une table qui contient la suite des adresses des noeuds à traverser accompagnées d'un code indiquant le sens de parcours. La signification des différents codes est la suivante :

DSC : pour les noeuds parcourus en descendant,
FEU : pour les noeuds correspondants aux feuilles du sous arbre extrait,
ASC : pour les noeuds parcourus en remontant.

Il faut également ajouter deux codes **NIV** et **SSA** qui permettent respectivement de traiter les cas particuliers d'expressions de références utilisant le "." (qui désigne tous les éléments d'un niveau) et le symbole ">" (qui désigne une sous arborescence complète).

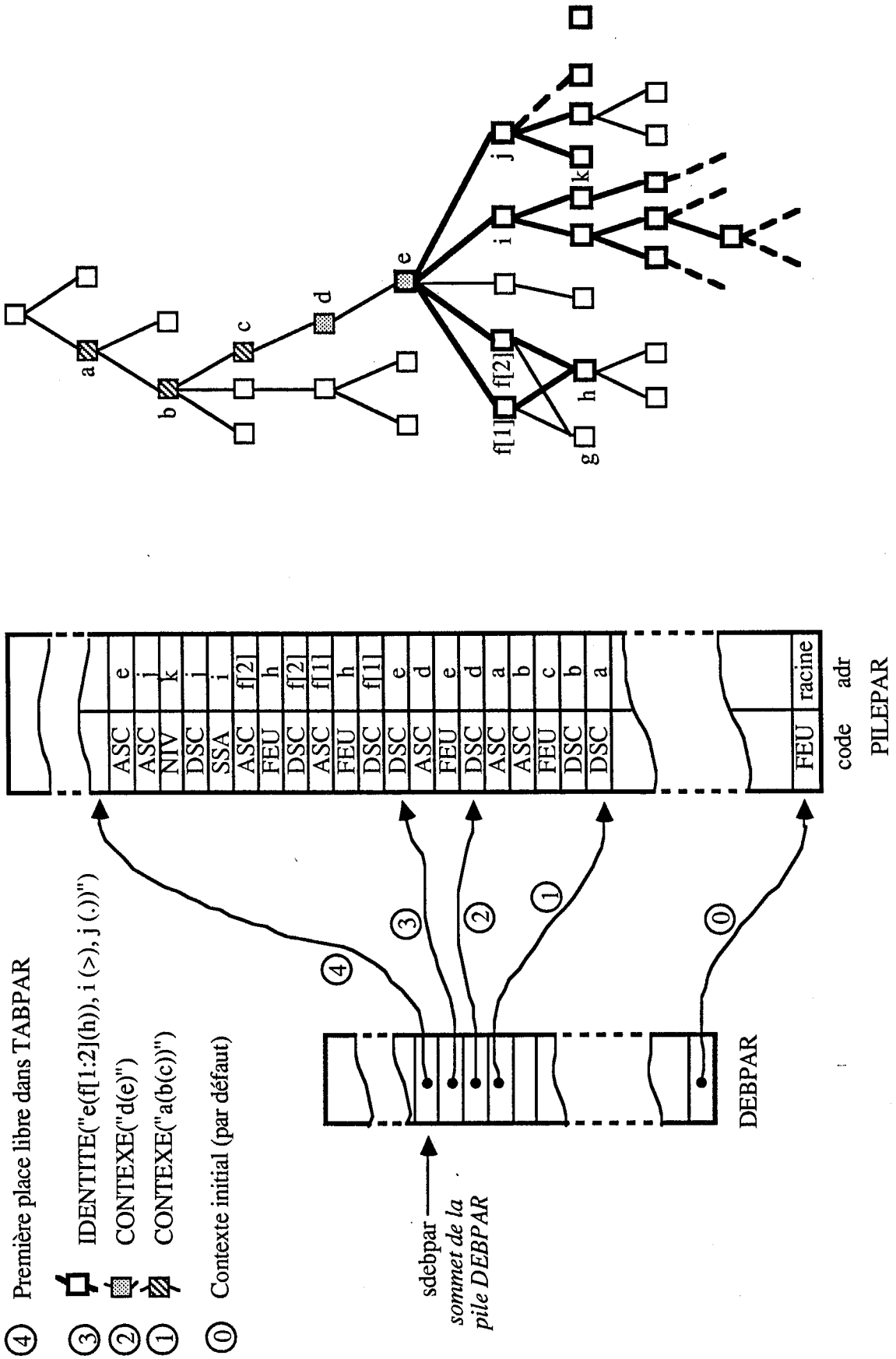
Pour le premier cas, seul le premier noeud du niveau est conservé dans la séquence de parcours avec le code **NIV**. Pour le second cas, seule la racine du sous arbre à parcourir est stockée avec le code **SSA**. Cela simplifie l'analyse, et permet de limiter le nombre d'éléments mémorisés dans la table, la primitive de parcours effectuant automatiquement les accès appropriés en fonction du code.

La table des séquences d'accès est gérée à l'aide d'un mécanisme de pile qui permet de conserver les parcours des différents contextes imbriqués et de l'identité courante (voir fig. V.11 page 201).

La pile DEBPAR contient les adresses du début des séquences de parcours des différents contextes et de l'identité courante qui sont elles conservées dans la table TABPAR. Le bas de DEBPAR (et de TABPAR) correspond à la définition du contexte par défaut (la racine de la base de données graphique) toujours présent. Sont empilées ensuite les définitions des différents contextes puis celle de l'identité courante. Le sommet de la pile DEBPAR contient l'adresse de la première position libre dans TABPAR.

La séquence de parcours de l'entité logique de niveau i est ainsi définie par les éléments de TABPAR compris entre la position DEBPAR[i] et DEBPAR [i + 1] - 1.

En fait, ces structures de données existent en double afin de permettre la définition simultanée et indépendante des entités logiques pour le travail et la visualisation (il s'agit respectivement de DEBPAR_TRAV, TABPAR_TRAV et de DEBPAR_VISU, TABPAR_VISU).



- figure V.11 : les structures de données pour le parcours des entités logiques-

3.4.3.2 Les primitives de parcours des entités logiques

L'unité de communication propose deux primitives pour le parcours des entités logiques (une pour le travail, l'autre pour la visualisation). Le principe est analogue à celui de la primitive de traversée d'une sous arborescence vue précédemment (§ 3.4.2), elles permettent d'appliquer des actions différentes aux noeuds rencontrés selon qu'ils sont parcourus dans le sens descendant, en remontant ou qu'il s'agit de feuilles. Ce sont :

pour les entités de travail

PARCOURS-TRAV(no_ident, ACTIOND, ACTIONF, ACTIONA)

et pour les entités de visualisation

PARCOURS-VISU (no_ident, ACTIOND, ACTIONF, ACTIONA)

- *no_ident* désigne le numéro d'imbrication de l'entité logique à parcourir (position dans DEBPAR_VISU ou DEBPAR_TRAV),

-*ACTIOND*, *ACTIONF*, *ACTIONA* sont les procédures à appliquer respectivement aux noeuds traversés en descendant, aux feuilles et aux noeuds traversés en remontant.

Nous donnons ci-dessous l'algorithme du sous programme de parcours pour les entités logiques de travail. Celui pour les entités associées à la visualisation est identique, seules les structures de données utilisées changent (TABPAR_VISU au lieu de TABPAR_TRAV et DEBPAR_VISU au lieu de DEBPAR_TRAV).

```
procedure PARCOURS-TRAV(no_ent : entier;  ACTIOND,ACTIONF,ACTIONA : procedure);
{* parcours RGD de la sous arborescence définie par l'entité logique de travail de niveau *}
{*no_ent . ACTIOND, ACTIONF  et ACTIONA  sont les procédures à appliquer*}
{* respectivement aux noeuds en descendant, aux feuilles et aux noeuds en remontant. *}
i : entier;
debut
   arrêt_branche = FAUX;
   stop_parcours = FAUX;
   pour i = DEBPAR_TRAV[no_ent] jusqu'à DEBPAR_TRAV[no_ent + 1] faire
      noeudcour = TABPAR_TRAV[i].adr;
      choix TABPAR _TRAV[i].code
         DSC :                    {* noeud en DESCENDANT *}
            ACTIOND;
            si arrêt_parcours alors retour();
         FEU :                          {* noeud FEUILLE *}
            ACTIONF;
            si arrêt_parcours alors retour();
         ASC :                    {* noeud en REMONTANT *}
            ACTIONA;
            si arrêt_parcours alors retour();
         SSA :          {* traitement de tout le sous arbre dont noeudcour est racine *}
            PAR_ARBRE(noeudcour, VRAI,ACTIOND,ACTIONF,ACTIONA);
         NIV :              {* parcours de tous les noeuds du niveau de noeudcour *}
            repéter
               ACTIONF;
               noeudcour = noeudcour->frèresuiv;
            jusqu'à  noeudcour = NIL;
      finchoix;
   finpour
fin
```

- Algorithme de parcours des entités logiques -

3.4.3.3 La gestion interne de structures de données pour le parcours

Dans ce qui suit nous présentons les différentes primitives internes à l'unité de communication utilisées pour la gestion des structures de données associées au parcours des sous-structures définies par les entités logiques.

a) Définition d'une nouvelle séquence de parcours.

La définition d'une nouvelle séquence d'accès doit être intialisée à l'aide de la primitive

DEBPROG(type_prog)

qui incrémente le pointeur sommet de pile DEBPAR_TRAV ou DEBPAR_VISU selon la valeur de *type_prog* (TRAV ou VISU). Le nouveau sommet de la pile d'adressage des séquences d'acccès est initialisé avec la valeur de la première position libre dans PILEPAR_TRAV ou PILEPAR_VISU (cette valeur est donnée par DEBPAR_TRAV/VISU[sommet - 1]).

b) Génération de la séquence d'accès.

La génération de la séquence d'accès associée à une entité logique s'effectue en empilant dans PILEPAR_TRAV (ou PILEPAR_VISU selon la nature de l'entité) la succesion des accès à chaque noeud definissant l'entité. Cela est réalisé par l'intermédiaire de la primitive

GENERE(type_prog,adr_noeud,mode)

où :

- *type_prog* désigne la nature de l'entité logique (VISU ou TRAV),

- *adr_noeud* adresse le noeud accédé,

- *mode* indique la nature de l'accès à ce noeud (ASC, DSC, FEU, SSA ou NIV).

Le couple (*adr_noeud,mode*) est ajouté au sommet de PILEPAR_TRAV (ou PILEPAR_VISU). Rappelons que celui-ci est donné par le contenu du sommet de la pile DEBPAR (_TRAV ou _VISU) qui sera donc incrémentée.

c) Supression d'un programme de parcours.

Pour supprimer la séquence d'accès correspondant à l'entité sommet de pile, on utilisera la primitive

FINPROG(type_prog)

qui décrémente simplement le pointeur vers le sommet de DEBPAR_TRAV ou DEBPAR_VISU selon la valeur de *typrog*.

3.5 L'analyse des expressions

L'analyse des chaines de caractères fournies en entrée des primitives de structuration s'effectue simultanément sur plusieurs plans :

D'une part elle vérifie si les expressions sont **syntaxiquement** correctes, c'est à dire si elles respectent la grammaire des différents types d'expressions (références simples, multiples, expressions de création) que nous avons présentées au chapître IV.

D'autre part l'analyse opère un contrôle sur la **sémantique** des expressions en fonction de leur contexte d'utilisation. Il s'agit ainsi de vérifier que les identificateurs (éventuellement

indicés) employés dans les primitives de structuration logique (contextes, identités) adressent bien des éléments existants dans la structure hiérarchique. Cela peut être, également, le test de l'unicité de la branche désignée par les expressions de référence simple, ou la recherche d'éventuels conflits entre identificateurs d'un même niveau (par exemple, recouvrement d'indices).

Ces différents contrôles syntaxiques et sémantiques réalisés lors de l'analyse peuvent provoquer différents cas d'erreur lors de l'appel des primitives de structuration qui seront alors indiqués à l'application (table V.1).

NATURE DE L'ERREUR	Type d'expression
Identificateur (nom) de noeud inconnu pour le niveau.	RS, RM
Indiçage incorrect IDEB > IFIN.	RS, RM, CR
Il n'existe pas d'élément dont les bornes d'indiçage correspondent à celles associées à l'identificateur courant.	RS, RM
Eclatement de structure impossible, il invalide une entité logique	RS, RM
Plusieures références à un même noeud à un même niveau.	RM
Création impossible, noeud déjà existant.	CR
Référence à plusieurs noeuds alors que référence simple requise.	RS
"," interdite dans ce type d'expression.	RS, CR
"." interdit dans ce type d'expression.	RS, CR
")" et "." interdits au même niveau.	RM
">" interdit dans ce type d'expression	RS, CR
Symbole non autorisé	RS, RM, CR

RS : référence simple, RM : référence multiple, CR : expression de création

- table V.1 : Les Erreurs détectées lors de l'Analyse -

Parallèlement à la vérification de la syntaxe et de la sémantique des expressions, l'analyse assure dans le cas d'expressions de référence la **génération** dans les structures de données appropriées des séquences d'accès aux noeuds correspondants à la chaîne étudiée. Celles-ci sont utilisées comme nous l'avons vu précédemment au paragraphe 3.4.3 par les primitives de parcours des entités logiques. Pour les expressions de création la structure définie est effectivement **construite** simultanément à l'analyse de la chaîne.

La primitive interne à l'unité de communication qui réalise l'analyse des expressions est

ANALYSE(chaîne, typ_expr) --> err

- *chaîne* contient l'expression alphanumérique à analyser,

- *typ_expr* spécifie la nature de l'expression. Les constantes prédéfines *CRE,SIMPL* ou *MULT* sont utilisées pour indiquer qu'il s'agit respectivement d'une expression de création, de

référence simple ou de référence multiple. Dans le cas d'une expression de référence, la constante *SIMPL* ou *MULT* est combinée avec l'une des constante *TRAV* ou *VISU* qui permet de définir s'il s'agit d'une entité de structureation logique pour la visualisaton ou le travail.

L'analyse renvoie un résultat qui est 0 si elle c'est déroulée correctement et sinon le numéro de l'erreur qui a été detecté.

Le détail de la réalisation de l'analyse, qui est basée sur un automate d'états finis, est donné en annexe 1, à laquelle le lecteur pourra éventuellement se reporter.

3.6 Réalisation des primitives de structuration

Les paragraphes qui suivent présentent la réalisation des primitives de structurations directement accessibles aux applications que nous avons présentées au § 3 du chapitre IV. Ces primitives utilisent l'analyse des expressions evoquées ci-dessus et les procedures de parcours de structure que nous avons étudiées précédemment.

3.6.1 Primitives de structuration logique

Les primitives de structuration logique sont basées essentiellement sur l'analyse des expressions qu'elles utilisent. Comme nous l'avons c'est cette dernière qui prend entièrement en charge la génération des séquences d'accès destinées au parcours des sous-arbroescences qu'elles définissent. Ainsi, seuls l'initialisation des structures de données associées (piles DEBPAR et PILEPAR) et le marquage des noeuds incombent réellement aux procédures de l'unité de communication réalisant les primitives de structuration logique.

3.6.1.1 Définition d'un nouveau contexte

La primitive **CONTEXTE_TRAV(chaîne)** (ou **CONTEXTE_VISU(chaîne)**) se décompose donc comme suit :

```
démarquer les noeuds de l'identité de travail (ou de visualisation) courante,

FINPROG(TRAV/VISU)  (* dépilement de la séquence d'accès correspondant à cette identité *)
DEBPROG(TRAV/VISU) (* définition du début d'une nouvelle séquence d'accès *)

erreur <-- ANALYSE(chaîne,SIMPL + TRAV/VISU) (* analyse de l'expression de référence
                    simple contenue dans chaîne qui défini le nouveau contexte *)

si erreur = 0 alors (* l'analyse s'est déroulée correctement *)
   marquer les noeuds du nouveau contexte
sinon
   définir identité de travail/visualisation par défaut identique au noeud contexte.
```

Le marquage ou démarquage des noeuds appartenant à une entité logique (contexte ou identité) s'effectue simplement en appliquant la primitive de parcours d'entité logique PARCOURS-TRAV (ou PARCOURS-VISU) à la séquence d'accès sommet de pile avec comme traitement pour les noeuds feuilles (FEU) et ascendant (ASC) une action qui positionne le champ *entité* du descripteur du noeud courant avec la valeur TRAV,VISU ou NUL selon la nature de l'opération à réaliser.

3.6.1.2 Définition d'une nouvelle identité

La primitive de définition d'une nouvelle identité de travail (de visualisation) **IDENTITE_TRAV(chaîne)** (**IDENTITE_VISU(chaîne)**) est analogue à la primitive de définition d'un nouveau contexte ci-dessus. L'identité courante est supprimée pour être remplacée par la nouvelle identité. Le traitement effectué est le suivant :

```
démarquer les noeuds de l'identité de travail (ou de visualisation) courante,

FINPROG(TRAV/VISU)  (* dépilement de la séquence d'accès correspondant à cette identité *)
DEBPROG(TRAV/VISU) (* définition du début d'une nouvelle séquence d'accès *)

erreur <-- ANALYSE(chaîne,MULT + TRAV/VISU) (* analyse de l'expresion de référence
            éventuellement multiple contenue dans chaîne qui défini la nouvelle identité *)

si erreur = 0 alors (* l'analyse s'est déroulée correctement *)
   marquer les noeuds de nouvelle identité
définir identité de travail/visualisation par défaut identique au noeud contexte.
```

3.6.1.3 Terminaison de contexte

Le dépilement d'un contexte de travail ou de visualisation (primitives **FIN_CONTEXTE**) ne fait bien sur pas appel au programme d'analyse. Il consiste simplement au démarquage des noeuds et à la mise à jour de la pile définissant les séquences d'accès associées aux entités logiques. Le traitement effectué est le suivant :

```
(* ---- tout d'abord on supprime l'identité courante définie dans le contexte à dépiler ----*)
démarquer les noeuds de l'identité de travail (ou de visualisation) courante,
FINPROG(TRAV/VISU)  (* dépilement de la séquence d'accès correspondant à cette identité *)

(* ---- suppression du contexte courant, et retour au contexte de niveau supérieur --- *)
démarquer les noeuds du contexte de travail (ou de visualisation) courant,
FINPROG(TRAV/VISU)  (* dépilement de la séquence d'accès correspondant à ce contexte *)

(* ---- il faut ensuite, redéfinir une identité de travail ou visualisation qui par défaut est le ---*)
(* ---- nouveau noeud contexte ----*)
DEBPROG(TRAV/VISU)
définir identité de travail/visualisation par défaut
```

3.6.2 Primitives de structuration du fichier graphique

3.6.2.1 Création d'une structure

La fonction **STRUCTURE(chaîne)** qui permet la création d'une structure hiérarchique se compose des traitements suivants :

```
erreur <-- ANALYSE (chaîne, CRE)
si erreur = 0 alors                    (* la création s'est déroulée sans erreur *)
   --> @ du noeud courant              (* ce noeud est la racine de la structure créée *)
sinon                                  (* la chaîne expression de création est incorrecte*)
                                       (* l'analyse a été interrompue *)
   détruire le sous-arbre qui a éventuellement été construit en partie
   --> NIL
finsi
```

La création de la structure proprement dite est réalisée comme nous l'avons vu par la procédure d'Analyse. En cas d'échec de celle-ci, une partie de la structure à néanmoins pu être construite (celle correspondant au début de l'expression correctement analysée). Il est donc nécessaire de supprimer ces éléments afin de libérer l'espace qui leur a été alloué. Cette récupération est réalisée en utilisant la procédure de parcours d'arbre (voir § 3.4.1) sur la structure hiérarchique dont la racine est le noeud courant et en appliquant l'action de destruction du noeud courant (DEL_NOEUD vue au § 3.2.3) aux éléments feuilles (FEU) et aux éléments traversés en remontant (ASC). Ceci est donc réalisé par l'appel :

PAR-ARBRE (VRAI, NOP, DEL_NOEUD, DEL_NOEUD)

(Remarque : c'est à cause de cette utilisation potentielle de la primitive de parcours d'arbre pour la destruction de structure que l'on effectue dans PAR_ARBRE le chaînage vers le frère suivant avant l'application de ACTIONA ou ACTIONF).

3.6.2.2 Copie d'une structure

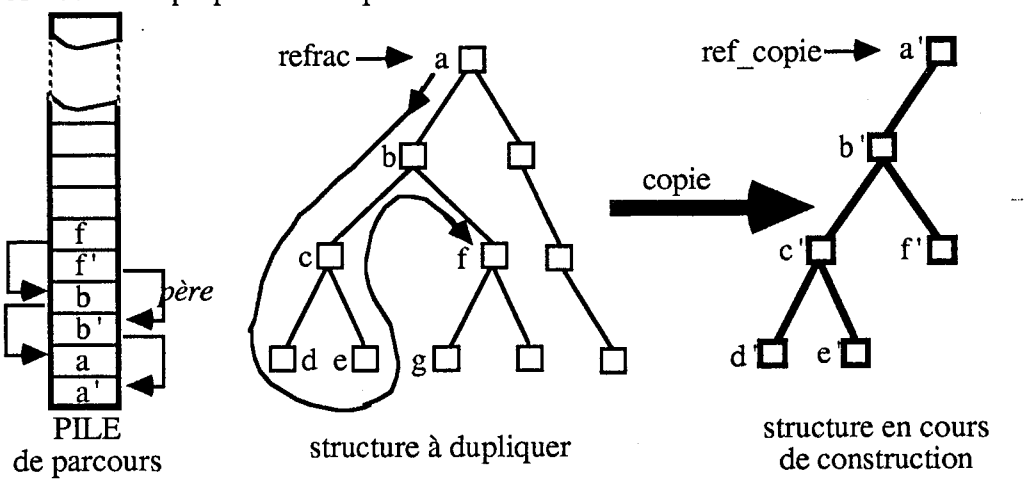
La deuxième possibilité pour la construction de la base de données graphique hiérarchique est la création d'une copie d'une structure déjà existante.

Ainsi, la primitive **COPIE(refrac,identif,borneinf,bornesup) --> refcopie** qui crée une structure identique à celle de racine *refrac*, et dont la racine *refcopie* aura pour identificateur *dentif[borneinf:bornesup]*.

La duplication de la structure de racine *refrac* est une action relativement complexe basée sur la primitive de parcours d'un sous-arbre (PAR_ARBRE) dont elle exploite au mieux les possibilités. C'est pourquoi nous détaillons ce traitement dans ce qui suit.

La structure originale à dupliquer est traversée à partir de sa racine *refrac* à l'aide de PAR_ARBRE; les noeuds rencontrés en descendant et les noeuds feuilles sont entièrement dupliqués (un nouveau noeud du même type (INS ou OBJ) est construit avec le même identificateur et les mêmes attributs). Dès que la copie d'un élément est ainsi créée, elle est insérée en bonne position dans la nouvelle structure, c'est à dire qu'elle est ajoutée à la liste des fils de son noeud père.

Afin de pouvoir réaliser les différents chaînages entre les noeuds dupliqués, il est indispensable de sauvegarder ceux-ci dans une pile de la même manière que sont conservés les noeuds originaux au moment du parcours. Pour cela on réutilise la pile destinée au parcours d'arbre (PILE). Lors du parcours de la structure originale, les noeuds rencontrés en descendant sont dupliqués et chaînés avec leur père et empilés. Pour cela, les noeuds dupliqués sont stockés dans la pile. Ainsi, lorsqu'un noeud est créé, son père se trouve sous le sommet de la pile (le noeud original a également été empilé par le procedure de parcours d'arbre). La figure V.12, schématise l'état de la pile lors du parcours pour duplication, le noeud f vient d'être traité en descendant, il a été dupliqué, sa copie f' a été chaînée avec son père b' puis empilée, finalement le noeud f a été lui-même empilé par PAR_ARBRE. Les noeuds feuilles subissent le même traitement, sauf qu'ils ne sont pas empilés. Lors de la traversée , en remontant de la structure originale, les noeuds dupliqués sont dépilés.



- figure V.12 : Parcours d'une structure pour la création d'une copie -

La seconde difficulté dans la copie de structure concerne les noeuds de type instance. En effet ceux-ci doivent être considérés comme des noeuds de type "feuille" car il ne faut pas dupliquer la sous structure qu'ils partagent. Aussi le parcours de la branche est-il interrompu quand le noeud dupliqué est une instance de structure.

```
procedure COPIE(refrac : ptr_noeud; identif : tabchar; borneinf,bornesup : entier)
(* duplique la structure de racine refrac avec l'identificateur identif[borneinf:bornesup] *)
var ref_copie : ptr_noeud;

   procedure COPIE
   (* action à appliquer aux "feuilles" de l'identité de travail *)

      procedure COPIE_DSC()
      (* action appliquée lors du parcours descendant du sous-arbre de racine refrac *)
      var ncopie : ptr_noeud;
      debut
         ncopie := DUPLIQUE(noeudcour)      (*création d'une copie du noeud courant *)
         INS_NOEUD(PILE[spile - 1],ncopie) (* insertion du noeud ncopie en queue de la*)
                                  (* liste des fils du noeud situé sous le sommet de PILE *)

         si ncopie->typ_noeud = OBJ alors
            EMPILE(ncopie)
         sinon                          (* noeud de type INStance *)
            arrêt_branche = VRAI;    (* le parcours de la branche est interrompu *)

      fin;

      procedure COPIE_FEU()
      (* action appliquée lors du parcours des feuilles du sous-arbre de racine refrac *)
      var ncopie : ptr_noeud;
      debut
         ncopie := DUPLIQUE(noeudcour)      (*création d'une copie du noeud courant *)
         INS_NOEUD(PILE[spile - 1],ncopie) (* insertion du noeud ncopie en queue de la*)
                                  (* liste des fils du noeud situé sous le sommet de PILE *)

      fin;

      procedure COPIE_ASC()
      (* action appliquée lors du parcours ascendant du sous-arbre de racine refrac *)
      debut

         DEPILE()  (* retire de PILE la copie du noeud qui y avait été placé lors du *)
                   (* parcours descendant *)

      fin;

   debut  (* COPIE *)

         ref_copie := DUPLIQUE(refrac);
         ref_copie->nom := identif;
         ref_copie->bornemin := bornemin;
         ref_copie->bornemax := bornemax;
         EMPILE(ref_copie);
         PAR_ARBRE(refrac,faux,COPIE_DSC,COPIE_FEU,COPIE_ASC);
         DEPILE()
         COPIE := ref_copie

   fin
```

- Algorithme détaillé de la copie de structure -

3.6.2.3 Rattachement d'une structure au fichier graphique

Le "rattachement" d'une structure à la base de données hiérarchique peut être réalisé à l'aide des primitives **ATTACHE(refrac)** et **REMPLACE(refrac) --> refdecro.**

Pour chaque noeud "feuille" de l'identité de travail courante on recherche parmi ses fils les éléments dont l'identificateur est identique à celui du noeud *refrac*, racine de la structure à rattacher, et dont les bornes d'indiçage interfèrent avec celles de *refrac*. Si cette liste est vide, le noeud *refrac* est inséré dans la liste des fils du noeud "feuille". Dans le cas contraire où l'identification du noeud *refrac* entrerait en conflit avec un noeud déjà existant au même niveau, l'attachement n'est pas effectué et un message d'erreur le signale à l'utilisateur.

Nous donnons ci-dessous, le schéma algorithmique du traitement réalisé pour la primitive ATTACHE, (celui de la primitive REMPLACE est similaire, on vérifie seulement que l'identité de travail est une référence simple, et on retourne l'adresse du noeud remplacé).

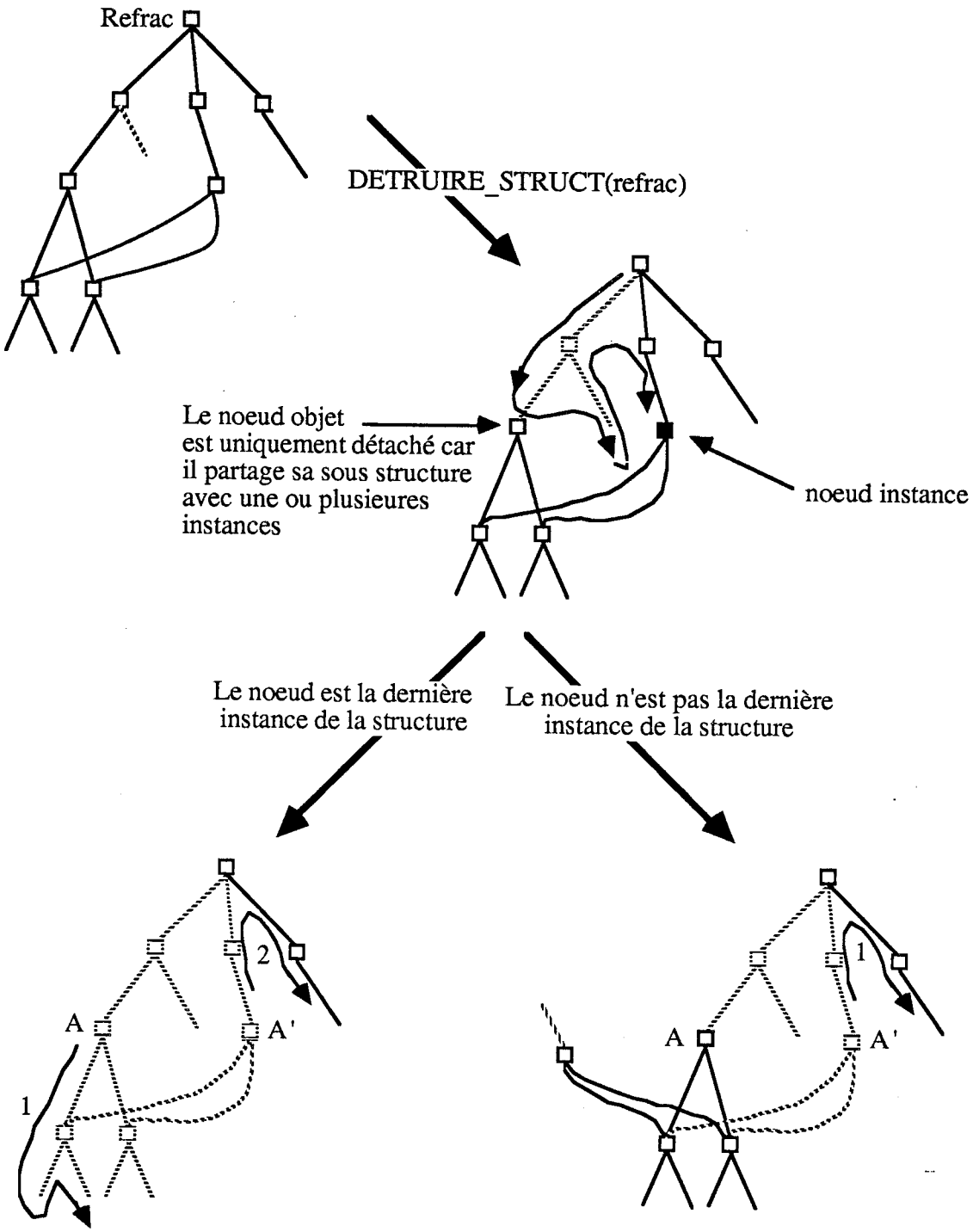
```
procedure ATTACHE(refrac : ptr_noeud);
(* attache la structure de racine refrac aux noeuds de l'identité de travail courante *)

   procedure AJOUT;
   debut
      recherche de conflit entre l'identificateur de refrac et les identificateurs des fils du noeud
      courant,
      si pas de conflit alors
         INS_NOEUD(refrac)
      sinon
         erreur(attachement impossible, conflit d'identificateurs)
   fin

debut

      PARCOURS-TRAV (sdebpar_trav-1, NOP, AJOUT, NOP);
      (*sdebpar_trav -1 désigne dans DEB-PAR-TRAV le début de l'identité de travail courante*)

fin
```

3.6.2.4 Destruction de structure

La destruction de structure (primitive **DETRUIRE_STRUC** directement accessible par l'application) présente beaucoup d'analogies avec la copie de structure étudiée précédemment : elle aussi nécessite un parcours d'arbre avec un abandon éventuel de branches si les noeuds rencontrés en descendant sont des noeuds de type instance ou des sous objets partagés.

En effet, une structure partagée ne peut être supprimée que si plus aucune instance ne la réfèrence. Si ce n'est pas le cas (champ *Nb_instances* du noeud $\neq 0$), il ne faut pas la détruire. Le noeud est donc simplement détaché de l'arborescence et le parcours de sa sous structure est abandonné (fig. VI.13 p 210). Cette structure est conservée par le système jusqu'à ce que toutes les instances qui la référencent soient effectivement détruites, dans ce cas elle est alors à son tour supprimée de la même manière.

De la même manière, la supression d'une instance de structure ne doit pas altérer la structure originale à laquelle elle se réfère. Il est donc nécessaire de la traiter comme une "feuille" afin de ne pas supprimer les sous-structures qu'elle partage, c'est à dire qu'il faut uniquement supprimer le noeud instance sans traverser sa sous structure (fig.VI.13).



Le noeud instance A' est supprimé
1 : la sous structure de A n'est plus utilisée par aucun noeud : destruction de A
2 : poursuite de la destruction de la structure

Le noeud instance A' est supprimé
la sous structure de A est encore partagée par d'autre noeuds instances, elle est conservéé
1 : poursuite de la destruction de la structure

- figure VI.13 : destruction de structure -

Ces deux points soulèvent un certain nombre de difficultés que l'on peut résoudre relativement simplement grâce à la généralité de la primitive de parcours d'arborescence PAR_ARBRE. Nous donnons dans ce qui suit, sous une forme algorithmique proche du PASCAL, la réalisation de la primitive de destruction de structure DETRUIRE_STRUC.

```
procedure DETRUIRE_STRUC(refrac : prt_noeud);
(* Destruction de la structure de racine refrac *)
debut
   si refrac->typ_noeud = OBJ_LIBRE ou refrac->typ_noeud = INS_LIBRE alors
      DEL_STRUCT(true,refrac);  (* destruction de la structure refrac, racine comprise*)
   sinon
      erreur : on ne peut detruire un objet non libre
   finsi
fin;

procedure DEL_STRUC(traiterac : booléen; refrac : ptr_noeud);
(* destruction de la structure de racine refrac , avec le noeud refrac compris si traiterac *)
(* sans le noeud refrac  sinon *)

   procedure DEL_INS;
   (*  opération de suppresion d'un noeud de type instance *)
   var noeudorig : ptr_noeud;
   debut
      noeudorig := noeudcour-->orig;
      DEL_NOEUD; (* destruction du noeud instance, avec décrémentation du compteur
                    de références du noeud original (voir § 3.2.3 *)
      si noeudorig-->Nb_instances = 0 alors
         (* il n'existe plus d'instances du noeud original noeudorig *)
         si noeudorig est dans la liste des noeuds détachés lors d'une destruction alors
            DEL_STRUC(true,noeudorig);
            retirer noeudorig de la liste des noeuds détachés lors d'une destruction
         finsi
      finsi
   fin;

   procedure DEL_DSC;
   (* operation de destruction à appliquer lors du parcours descendant de l'arborescence *)
   debut
      si noeudcour-> typ_noeud = INS alors (* noeud de type instance *)
         DEL_INS;
         arrêt_branche := vrai; (* on ne détruit pas la sous structure partagée par le noeud
                                   instance *)
      sinon
         si noeudcour->typ_noeud = OBJ et noeudcour-->Nb_instances ≠ 0 alors
            (* le noeud est la racine d'une sous structure partagée *)
            détacher noeudcour de la structure et le marquer comme objet libre
            ajouter noeudcour à la  liste des noeuds détachés lors d'une destruction
            arrêt_branche := vrai; (* on ne détruit pas la sous structure du noeud courant
                partagée par un ou plusieurs noeuds instance *)
         finsi
      finsi
   fin;

   procedure DEL_FEU;
   (* operation de destruction à appliquer lors du parcours des feuilles de l'arborescence *)
   debut
      si noeudcour-> typ_noeud = INS alors (* noeud de type instance *)
         DEL_INS;
      sinon
         si noeudcour->typ_noeud = OBJ et noeudcour-->Nb_instances ≠ 0 alors
            (* le noeud est la racine d'une sous structure partagée *)
            détacher le noeud de la structure et le marquer comme objet libre
            ajouter noeudcour à la  liste des noeuds détachés lors d'une destruction
         sinon
```

```
            (* le noeud courant est un sous objet non partagé : Nb_instances = 0 *)
            DEL_NOEUD; (* destruction du noeud *)
         finsi
      finsi
   fin;

   procedure DEL_ASC;
   (* operation de destruction à appliquer lors du parcours ascendant de l'arborescence *)
   debut
      (* le noeud courant est un sous objet non partagé : Nb_instances = 0 *)
      DEL_NOEUD; (* destruction du noeud *)
   fin;

debut
   PARCOURS_ARBRE(traiterac,refrac,DEL_DSC,DEL_FEU,DEL_ASC);
fin;
```

- Algorithme de destruction de structure -

Les autres primitives de destruction de structure proposées par CLOVIS, **VIDER_IDENTITE** et **DETRUIRE_IDENTITE** utilisent directement la primitive de destruction d'arbre présentée ci dessus. Celle-ci est appliquée à chacun des noeuds feuilles de l'identité de travail avec la prise en compte ou non de ces noeuds dans la destruction selon que l'identité est entièrement détruite ou simplement vidée.

```
procedure VIDER_IDENTITE;
(* destruction des sous structures issues des noeuds de l'identité de travail *)
debut
   PARCOURS-TRAV(sdebpar_trav,NOP,DEL_ARBRE(vrai,noeudcour),NOP);
   (* sdebpar_trav pointe vers la séquence d'accès au noeuds de l'identité de travail
      dans la pile DEBPAR_TRAV *)
fin;

procedure DETRUIRE_IDENTITE;
(* destruction des noeuds de l'identité de travail et de leurs sous structures *)
debut
   si l'identité courante ≠ racine du fichier graphique alors
      PARCOURS-TRAV(sdebpar_trav,NOP,DEL_ARBRE(faux,noeudcour),NOP);
      (* sdebpar_trav pointe vers la séquence d'accès au noeuds de l'identité de travail
         dans la pile DEBPAR_TRAV*)
      définir une nouvelle identité par defaut égale au noeud contexte courant
   sinon
      erreur : impossible de détruire la racine du fichier graphique
   finsi
fin;
```

3.7 Primitives de consultation d'entités logiques et de consultation de structures

La primitive de consultation du nombre de niveaux de contextes **NB_NIV_CTX(visu/trav)** renvoie le nombre de niveaux de contextes définis. Ce résultat est obtenu immédiatement à partir de la pile des adresses des débuts des séquences de parcours de ces entités (pile DEBPAR_TRAV ou DEBPAR_VISU que nous avons présentées au paragraphe 3.4.3.1). Le nombre de contextes est égal à valeur du sommet de cette pile moins deux (*spile-2*), qui désigne le dernier contexte défini.

Les primitives **CONS_CTXT(visu/trav,niv)** ou **CONS_IDENTITE(visu/trav)** renvoient une chaîne qui correspond à l'expression alphanumérique définissant une entité logique. Leur réalisation s'effectue à l'aide des primitives de parcours PARCOURS-VISU ou PARCOURS-TRAV, la chaîne étant reconstituée à partir des identificateurs des noeuds traversés.

Nous présentons dans ce qui suit, sous une forme algorithmique, la primitive **CONS_ENTITE_LOGIQUE** qui effectue cette reconstitution des expressions. *Niv* indique le niveau de l'entité logique à traiter (niveau du contexte ou si *niv* est égal à la valeur du sommet de la pile DEBPAR - 1, identité de travail ou de visualisation). La variable locale *dernoeud* quand à elle sert à conserver le sens de traversé du dernier noeud traité, il permet ainsi la prise en compte de plusieurs noeuds à un même niveau dont les identificateurs sont séparés par une virgule (','). La fonction **IDENTIF(adrnoeud)** que nous utilisons pour la construction de l'expression, renvoie l'identificateur du noeud accompagné de ses bornes d'indiçage. Ainsi lors de consultations de structures, l'expression retournée contiendra systèmatiquement les nom indicés de chaque noeud.

```
fonction CONS_ENTITE_LOGIQUE(visu/trav,niv) ---> chaîne;
(* consultation de l'expression correspondant à l'entité de structuration logique de travail
   ou de visualisation de niveau niv *)
var dernoeud : entier; (* indicateur du sens de traversé du dernier noeud traité *)

   procedure NOM_ASC;
   (* opérateur de reconstruction des expressions pour traversée des noeuds en remontant *)
   debut
      CONCATENER(chaîne,')');
      dernoeud = ASC
   fin;

   procedure NOM_FEU;
   (* opérateur de reconstruction des expressions pour traversée des noeuds feuilles *)
   debut
      si dernoeud =FEU ou dernoeud = ASC alors
         CONCATENER(chaîne,',');
      finsi
      CONCATENER(chaîne,IDENTIF(noeudcour))
      dernoeud = FEU
   fin;

   procedure NOM_DSC;
   (* opérateur de reconstruction des expressions pour traversée des noeuds en descendant *)
   debut
      si dernoeud =FEU ou dernoeud = ASC alors
         CONCATENER(chaîne,',');
      finsi
      CONCATENER(chaîne,IDENTIF(noeudcour))
      dernoeud = DSC
   fin;

debut
   dernoeud = DSC;
   chaîne = '';
   si trav alors
      PARCOURS-TRAV(niv,NOM_DSC,NOM_FEU,NOM_ASC);
   sinon
      PARCOURS-VISU(niv,NOM_DSC,NOM_FEU,NOM_ASC);
   retour(chaîne)
fin;
```

La primitive de consultation de la structure d'un élément **CONS_ARBRE(refrac)** qui produit la chaîne correspondant à toute la structure issue de *refrac* utilise les même opérateurs de construction, mais ils sont appelés dans la primitive de parcours PAR_ARBRE qui permet la traversée d'une structure hiérarchique complète à partir d'un noeud donné.

4 L'UNITE DE DESCRIPTION VISUALISATION

L'unité de description et visualisation est une banque d'opérateurs élémentaires pour les différents attributs considérés par le système graphique. Ces opérateurs sont mis à la disposition de l'unité de contrôle pour la réalisation des processus de base. Cette organisation permet, plutôt qu'un partionnement en couches, de reporter l'ordonnancement du procesus au niveau de l'unité de contrôle.

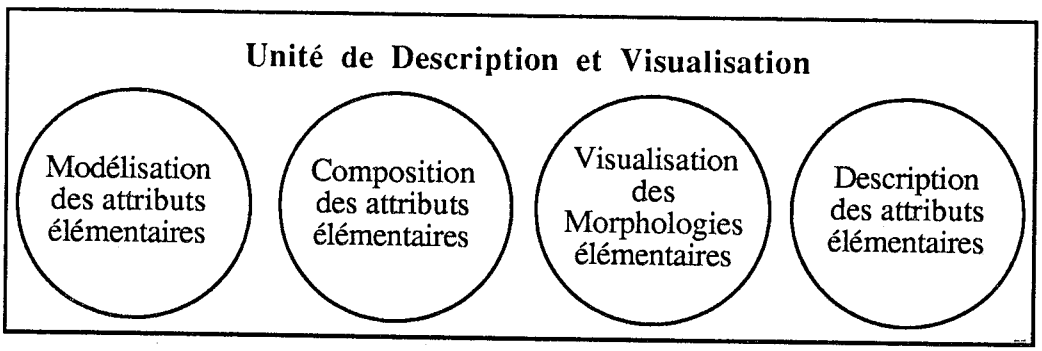
Quatre types d'opérateurs élémentaires peuvent être distingués au sein de l'unité de description et visualisation :

- les opérateurs de **modélisation** qui concernent directement les descripteurs des attributs élémentaires que nous avons présentés dans la troisème partie du chapitre IV. permettent la construction effective des descripteur, l'accès à l'information qu'il continnent.

- les opérateurs de **composition** ou **décomposition**, qui permettent la composition d'attributs. Il s'agit par exemple de la composition des attributs géométriques, de l'application d'une transformation géométrique à des coordonnées.....

- les opérateurs de **visualisation** qui permettent l'affichage des morphologies de base. Ces opérateurs effectuent la synthèse des attributs élémentaires.

- les opérateurs de **description** qui permettent d'effectuer les interactions avec l'utilisateur, (collecte de coordonnées, echos....)



- figure VI.14 : Les différents modules de l'unité de description et visualisation -

Il est clair que le module de modélisation est dans une large mesure indépendant du matériel. Les opérateurs qu'il comporte sont dépendants des types des attributs considérés à l'intérieur de chacune des classe. Leur modélisation peut être indépendante du matériel considéré, ce sont les opérateurs de composition, et surtout de visualisation et desciption qui effectuerons les conversions nécessaires pour passer à une représentation acceptée par les dispositifs physiques concernés.

Les autres modules quand à eux sont dépendants du matériel graphique utilisé. Selon l'intelligence de celui-ci, il pourra s'agir d'un simple appel d'opérateurs matériel évolué, ou d'un simulation par le logiciel. Par exemple la composition des transformations géométriques peut être effectuée par un le logiciel si le matériel ne les gère pas. Le changment de matériel, nécessite

donc la réécriture des ces modules. Cependant, les opérateurs étant extrèmement modulaires, et leur ordonnancement étant reporté au niveau de l'unité de contrôle, ceci ne s'avère pas une tache extrèmement difficile. Pour deux matériel de fonctionnalités équivalente beaucoup d'opérateurs peuvent être réutilisés ou facilement adaptés.

Nous avons personnellement participés à la réalisation des différents modules de l'unité de description et visualisation pour piloter un terminal TEKTRONIX 4114 [GeGr 85]. Une autre unité à été définie pour le pilotage du synthéseur d'images HELIOS, prototype réalisé au laboratore ARTEMIS [Fere 81],[Boul 86].

Du fait de la banalisation de l'unité de communicatin, qui gère la structuration hiérarchique des données graphiques, la prise en compte de nouveaux types d'attributs élémentaires ne pose pas de difficultés. Il suffit de rajouter dans chacun des modules les opérateurs correspondants. Nous avons pu tester cette modularité en intégrant un nouveau type de morphologie : les courbes de niveau.

5 UNITE DE CONTROLE

L'unité de contrôle assure l'ordonnancement des processus à partir des opérateurs élémentaires proposés par l'unité de description/visualisation et l'unité de communication. Les différents processus sont réalisés à partir des primitives de parcours de structure de l'unité de communication. Les traitements effectués sur les éléments rencontrés se font en invoquant les opérateurs de l'unité de description visualisation correspondant au processus réalisé.

Il est clair que l'ordonnancement des processus dépend bien entendu du matériel utilisé pour la visualisation. La prise en compte d'un nouveau matériel peut nécessiter une totale refonte de ceux-ci. Cependant, la généralité des primitives de parcours, et la nette séparation entre les processus (unité de controle) et les opérateurs élémentaires (unité de description et visualisation) permettent, nous le pensons, une réalisation relativement aisée dans de très nombreux cas de figure.

Pour notre part nous avons réalisé une unité de contrôle pour des terminaux bas de gamme sans éffacement sélectif (simple tubes mémoire) [GeGr 85]. Cette unité de contrôle est facilement adaptable à tout type de terminal bas de gamme. La majeure partie des traitements étant effectuée par des opérateurs logiciels, l'ordonnacement de processus reste le même, seuls vont différer les opérateurs de l'unité de description et visualisation.

5.1 Processus d'attribution et de consultation

De par la généralité des primitives de parcours de la structure hiérarchique les processus d'attribution et de consultation sont réalisé de manière extrèment simple. Il s'agit d'accéder aux noeuds correspondant à l'entité de travail courante et d'invoquer les primitives banalisés de gestion des attributs de l'unité de communication (recherche, affection d'attribut). Nous donnons ci-dessous le schéma de la primitive ATTRIBUER qui affecte un attribut aux élements de l'entité de travail courante.

```
procedure  ATTRIBUER(refatt,var/cste)
/* affecte l'attribut refattaux éléments de l'entité de travail courante */
var ref  : ref_attribut; /* pointeur banalisé sur  l'attribut à affecter */

   procedure Attribuer_élément;
   /* affection de l'attribut à  l'élément courant */
   var refgeom : ref_attribut;
   debut
      si classe_attribut(ref) = geom alors /* eventuelle composition avec  géométrie presente */
      debut
         refgeom := RCH_ATT(geom); /* attribut Géométrique de l'élément */
```

```
            AFF_ATT(COMPOSE_GEOM(ref,refgeom));
            /* Composition des dex attributs géoémtriques et affectation à l'élément courant de
            l'attribut composé. COMPOSE_GEOM est une primitive de l'unité description et
            visualisation qui effectue la composition des géométrie */
         fin
         sinon
            AFF_ATT(ref);  /* affectation sans  composition */
      fin

   debut

      /* pour les règles d'attribution des attributs variables et constants , voir p. 170 */
      si var_cste =VARI et type_att(refatt) = CSTE) alors
         erreur(attribution impossible)
         retour;
      finsi
      si var_cste =CSTE et type_att(refatt) = VARI) alors
         ref := COPIE(refatt)
      sinon
         ref := refatt;

      PARCOURS-TRAV (sdebpar_trav-1, NOP, Attribuer_élément, NOP);
      (*sdebpar_trav -1 désigne dans DEB-PAR-TRAV le début de l'entité de travail courante*)

   fin
```

Le processus de consultation est quand à lui réalisé de manière immédiate en appliquant l'opérateur RCH_ATT à l'élément de l'entité de travail courante accédé par PARCOURS_TRAV.

5.2 Processus de visualisation

Pour le processus de visualisation, nous avons réalisé un processus correspondant au dessin au trait sans élémination des partie cachées. Il s'agit alors d'un simple parcours de structure avec une évaluation dynamique des attributs. Cette évaluation est effectué par un mécanisme de pile. A chaque noouvel élément rencontré, un nouveau niveau de pile est créé avec les valeurs des attributs évalués précédemment. Les attributs de l'élément sont composé avec ces attributs sommet de pile. Une fois l'élément traité pour la visualisation (sa sous-structure parcourue), les attributs du niveau précédent sont restitués par un simple dépilement.

```
procedure  VISUALISER

   procedure TRT_ELMT;
   /*empile et compose les attributs de l'élément courant , si il a une morphologie  elle est affichée */
   debut
      Pour chaque classe d'attribut considérée faire
            ref := RCH_ATT(classe)
            /* composition de l'attribut avec celui du niveau précédent */
            /* empilement de l'attribut composé */
            Empiler(COMPOSE(classe,ref,sommetpile d'évalution pour la classe))
      fin pour
      ref := RCH_ATT(Morpho);
      si ref <> nil alors  /* afficher la morphologie */
         afficher la morphologie évaluée avec les attributs courants (attributs sommets
         de la pile d'évaluation). Pour cela, on invoque les opérateurs de visualisation
         de l'unité de descripton/visualisation correspondant au type de la morphologie
      fin si
   fin
```

```
    procedure DEPIL_ATT
    /* l'élément courant et sa sous-structure on été affichés, on revient aux attributs du
       niveau précédent */
    debut
       Dépiler la pile d'évaluation des attributs
    fin

    procedure Visu_élément;
    /* visualisation de l'élément courant */
    debut
          PAR_ARBRE(vrai,TRT_ELMT,TRT_ELMT,DEPIL_ATT)
    fin

debut

    /* visualisation de chacn des éléments de l'entité de visualisation courante */
    PARCOURS-TRAV (sdebpar_visu-1, NOP, Visu_élément, NOP);
    (*sdebpar_trav -1 désigne dans DEB-PAR-VISU le début de l'élément de travail courante*)

fin
```

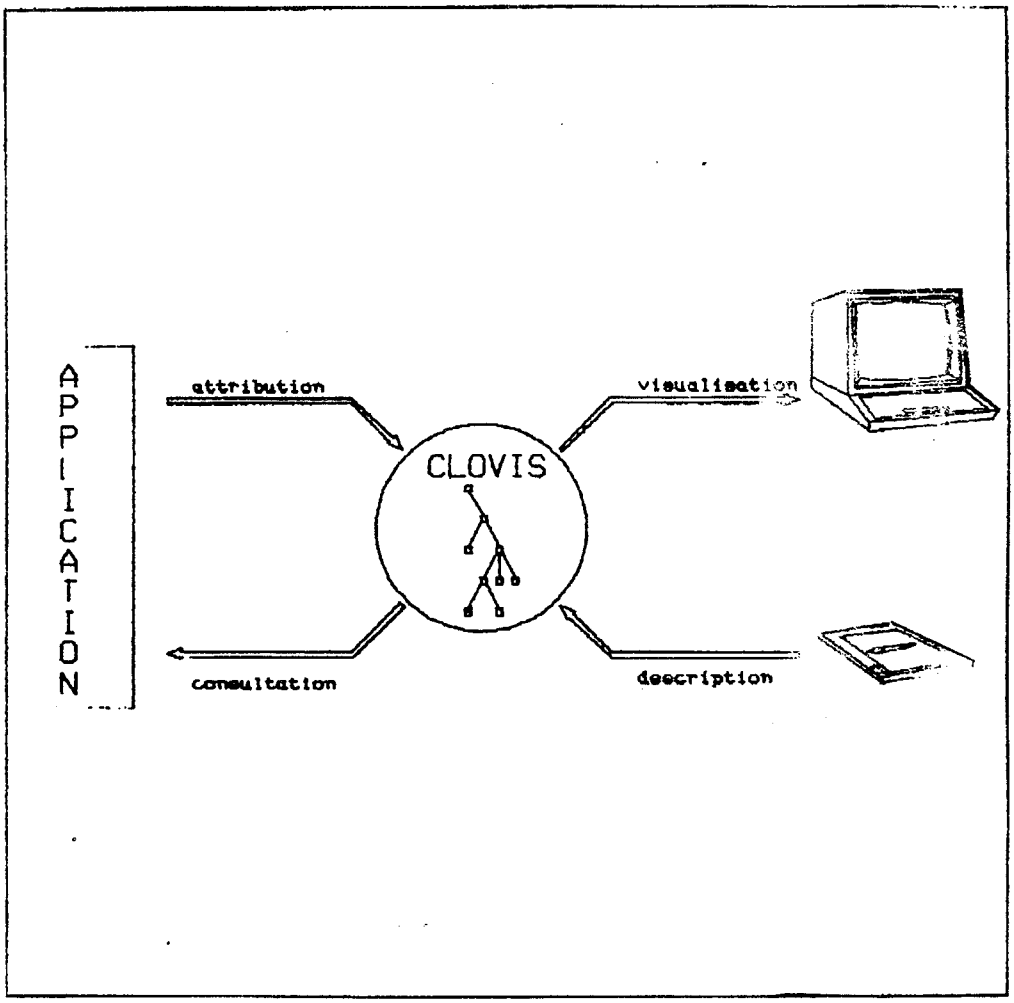
6 CONCLUSION

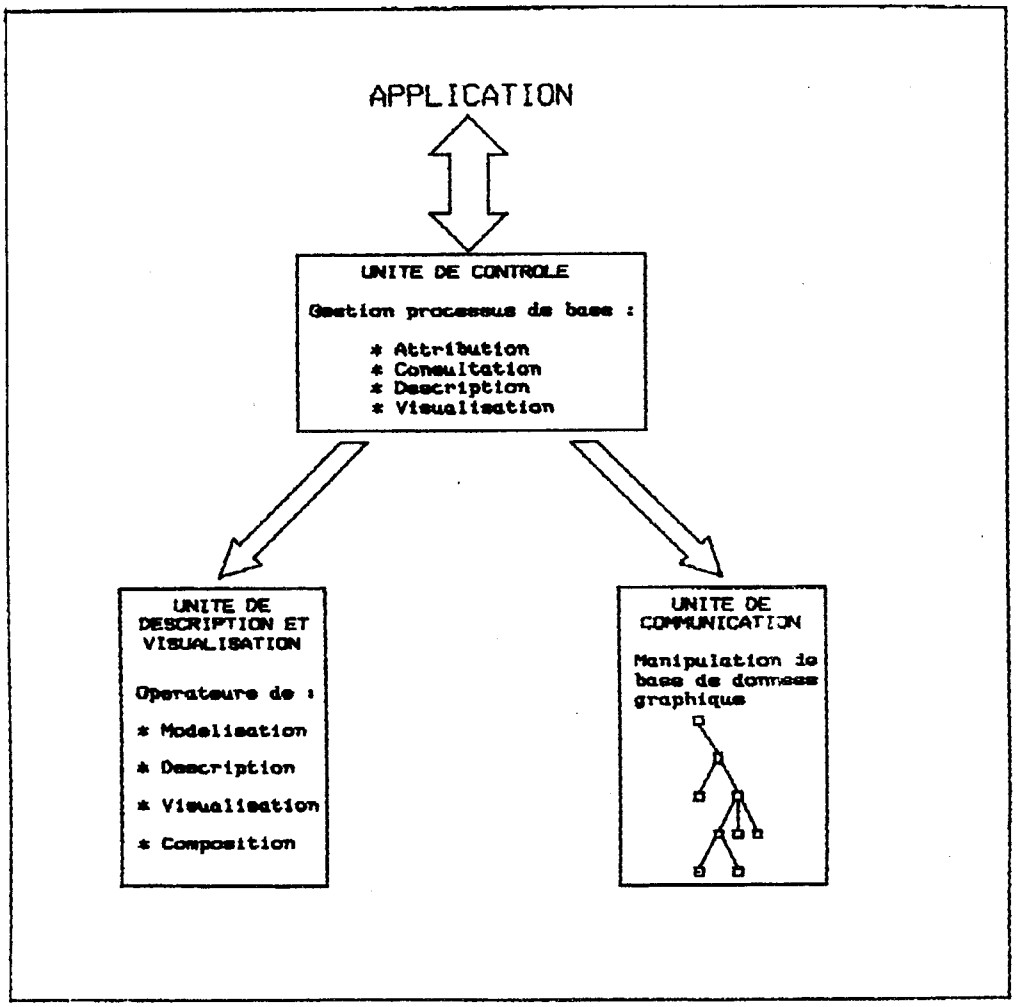
L'implémentation de CLOVIS nous a permi de valider dans une certaine mesure notre approche. Nous nous sommes tout particulièrment intéressés à la réalisation de l'unité de communication. Par la puissance des opérateurs qu'elle propose (parcours de structures hiérarchique) et par sa banalisation (ses opérateurs sont indépendants des types d'attributs élémentaires manipulés), elle permet d'écrire simplement les primitives de haut niveau correspondant aux processus de base (Attribution, Consultation, Visualisation et Description) réalisés au niveau de l'unité de contrôle. Cela a montré en particulier l'intérêt que présente la structuration hiérarchique des données dans le contexte du graphique et la prise en charge de sa gestion par le système graphique tant du point de vue de la puissance de description que la facilité de modification qui se répercute immédiatement sur l'interactivté du système.

La modularité du système assure son extensibilité. La prise en compte de nouveaux types d'objet et d'attributs s'effectue simplment en rajoutant les opérateurs correspondants au niveau de l'unité de description et visualisation. Il n'y a pas de modification à apporter au niveau de l'unité de communication. Quand à l'unité de controle, pour la prise ne compte du nouvel attribut il suffit d'intégrer à la réalisation des différents processus l'appel des opérateurs de l'unité de description-visualisation correspondants.

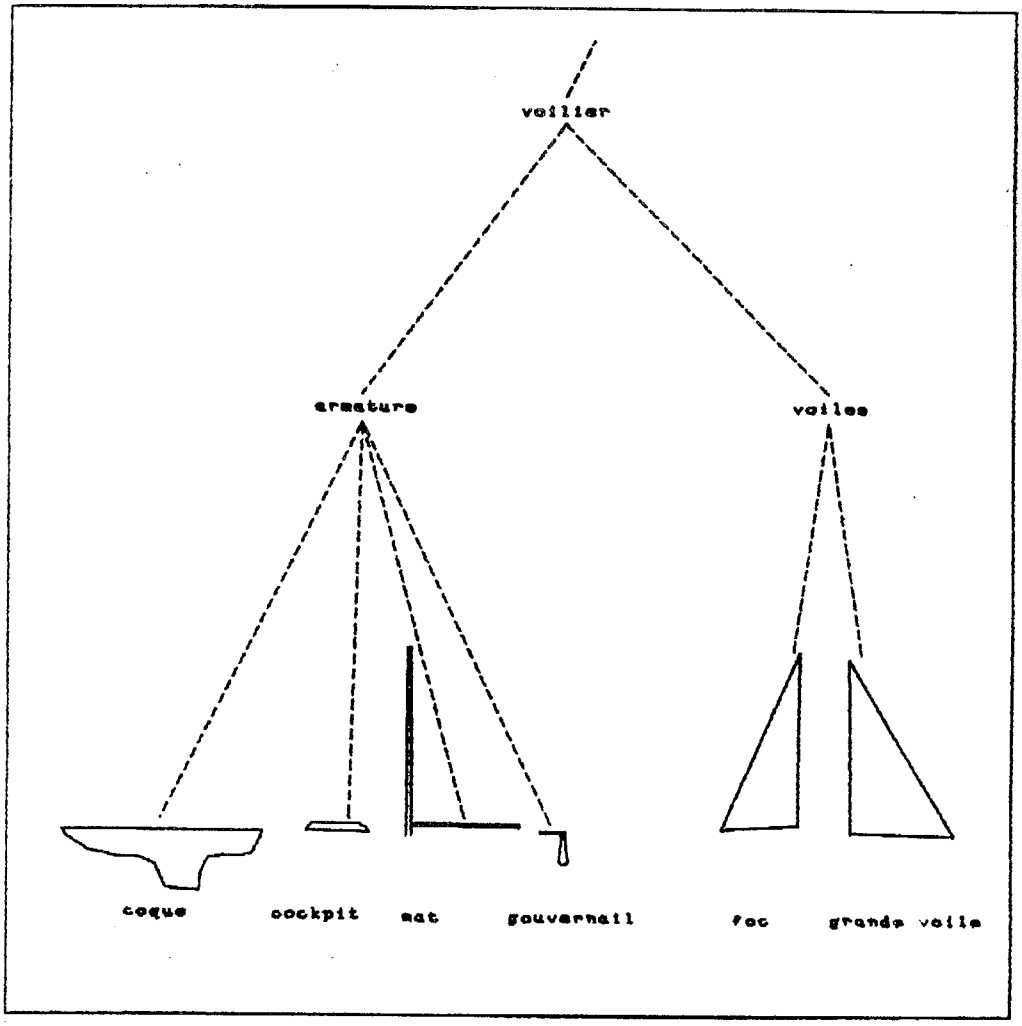
Toutefois, l'un des point faible de notre réalisation a été la prise en compte de matériels différents. Si les modules de l'unité de description et visualisation que nous avons écrits adressent correctement des matériels aux possibilités relativement limités, où la plus grande partie des processus de visualisation et description est réalisée de manière logicielle, la prise en compte de matériels évolués possédant en particulier leur propre liste de visualisation n'a pas été clairement abordée.

Quelques images produites avec CLOVIS sur un terminal Tektronix 4114





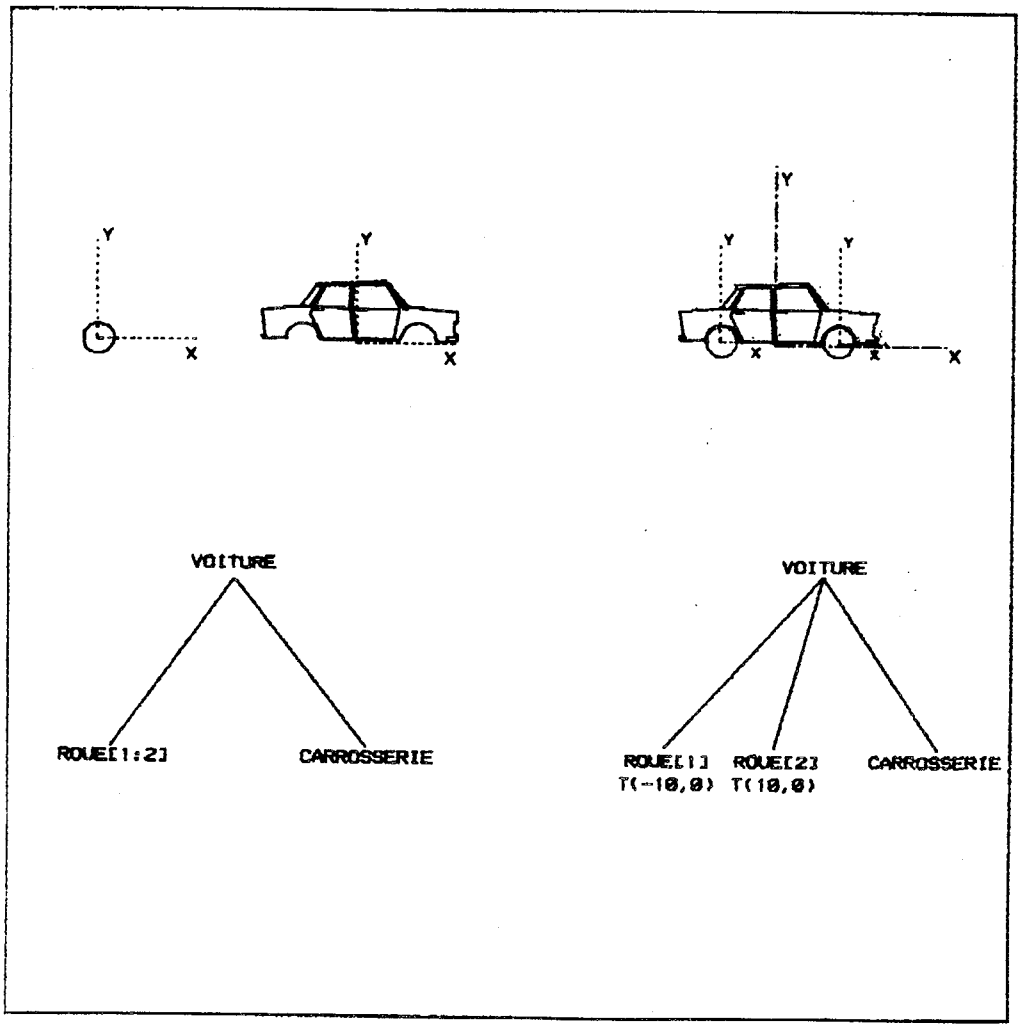
E> C>tx I>dt idvis>U A>tt D>esr V>isu id>F S>tru aR>chiv Q>uit —>



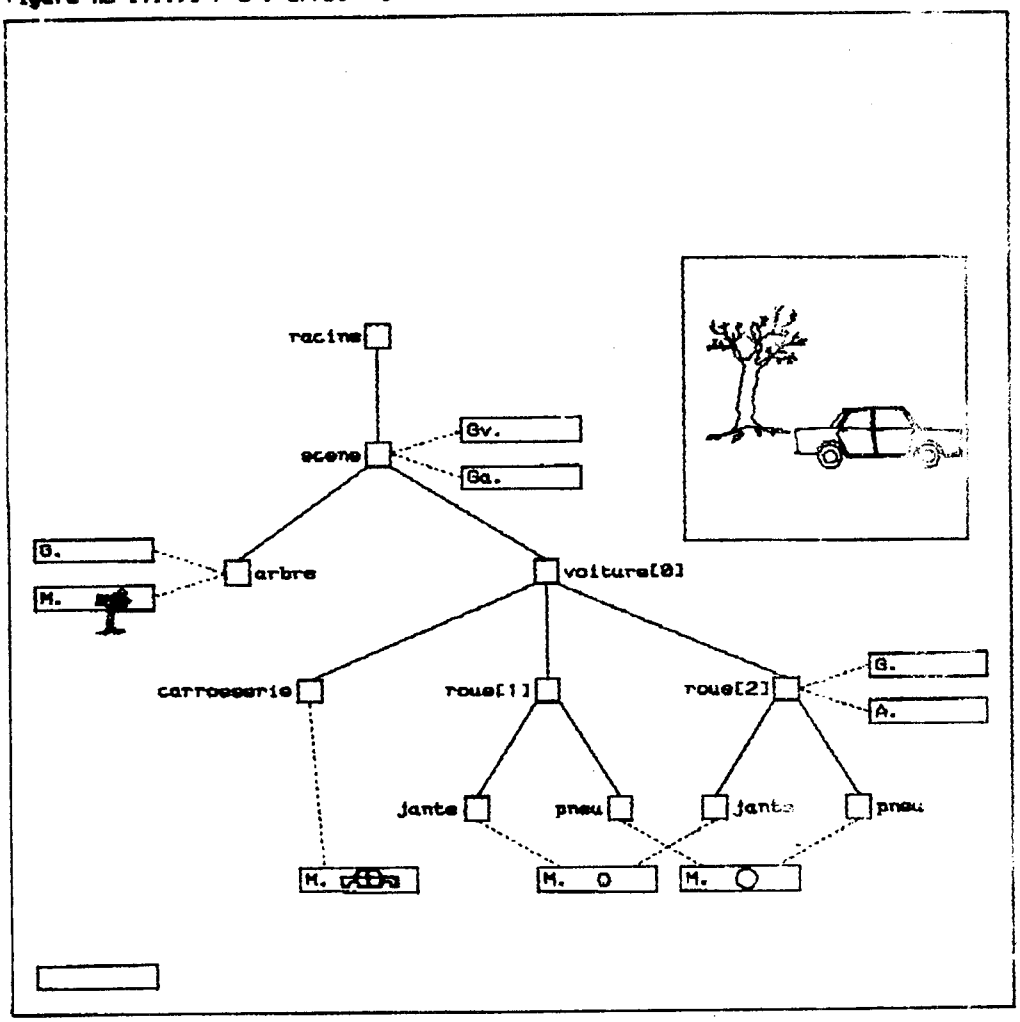
scene\texte[0])

contexte :

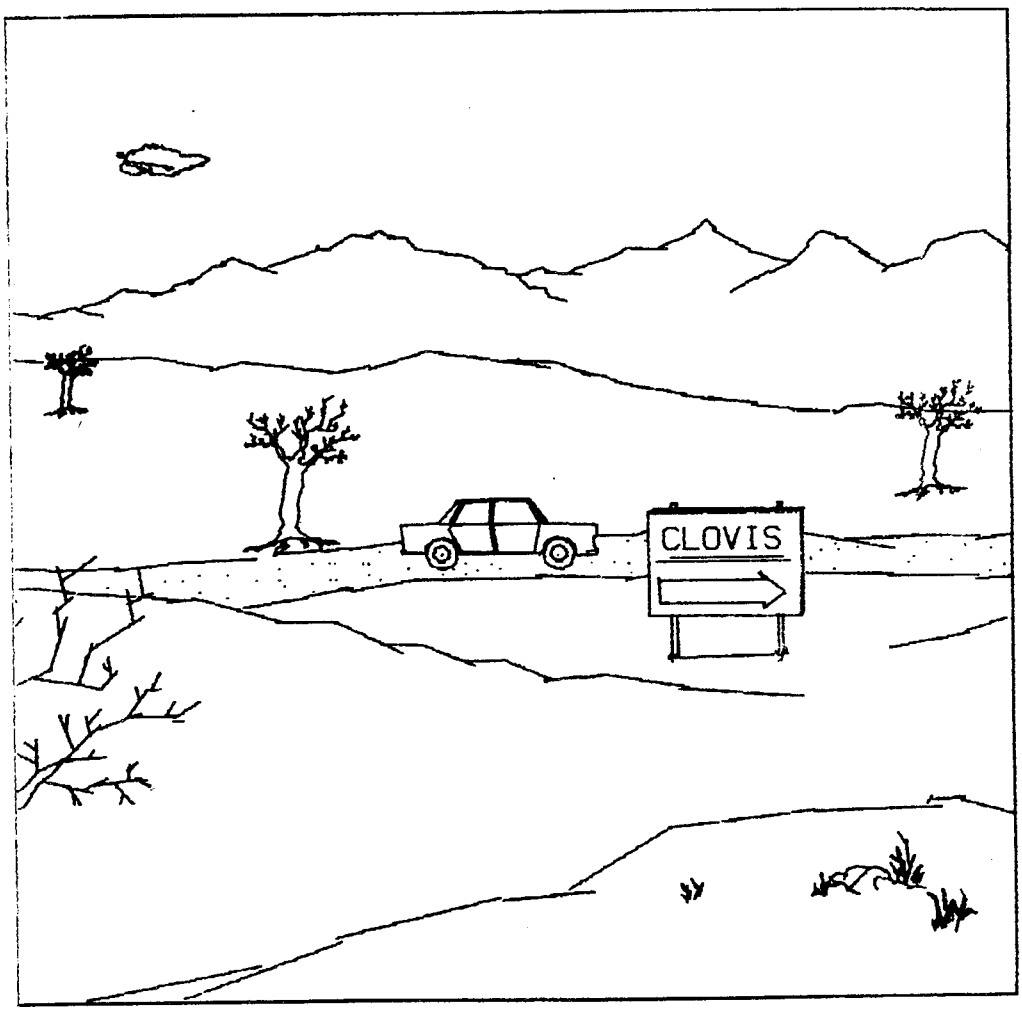
E> C>tx I>dt idvis>U A>tt D>esr V>isu id>F S>tru aR>chiv Q>uit —>

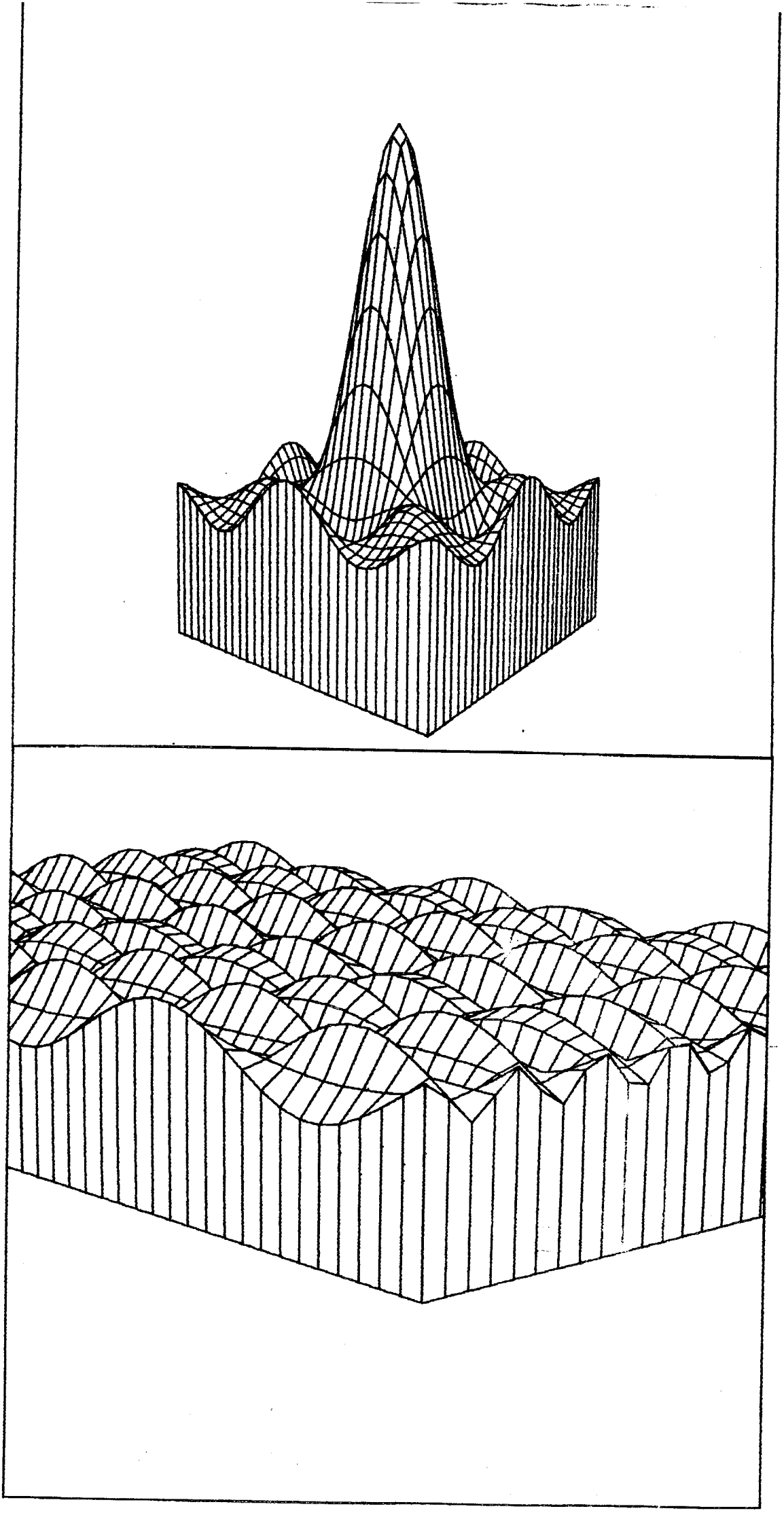


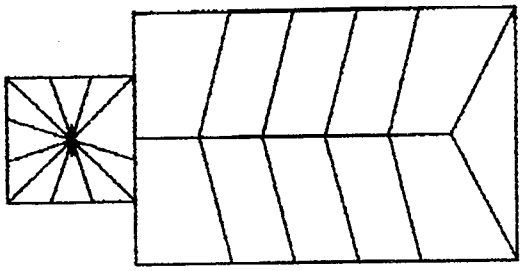
```
concat nomfich : [imag.clovis.image.morpho.m2d]micaduni7.m2d
concat nomfich : [imag.clovis.image.morpho.m2d]micadctr7.m2d
concat nomfich : [imag.clovis.image.morpho.m2d]micadarb7.m2d
figure no [1..7] / 0 : arret --> 6
figure no [1..7] / 0 : arret -->
```

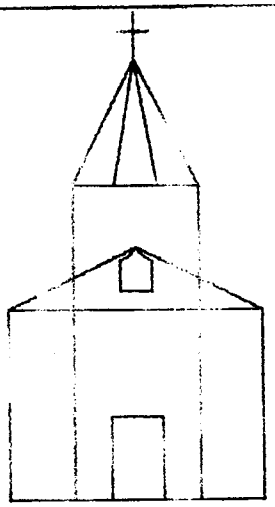


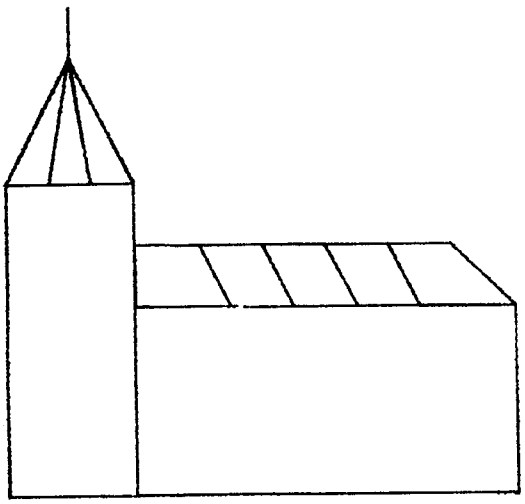

```
E> C>tx I>dt idvis>U A>tt D>esr V>ieu id>F S>tru eR>chiv Q>uit -->
```

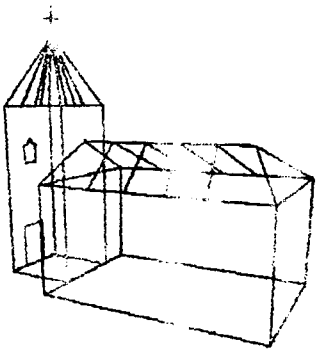


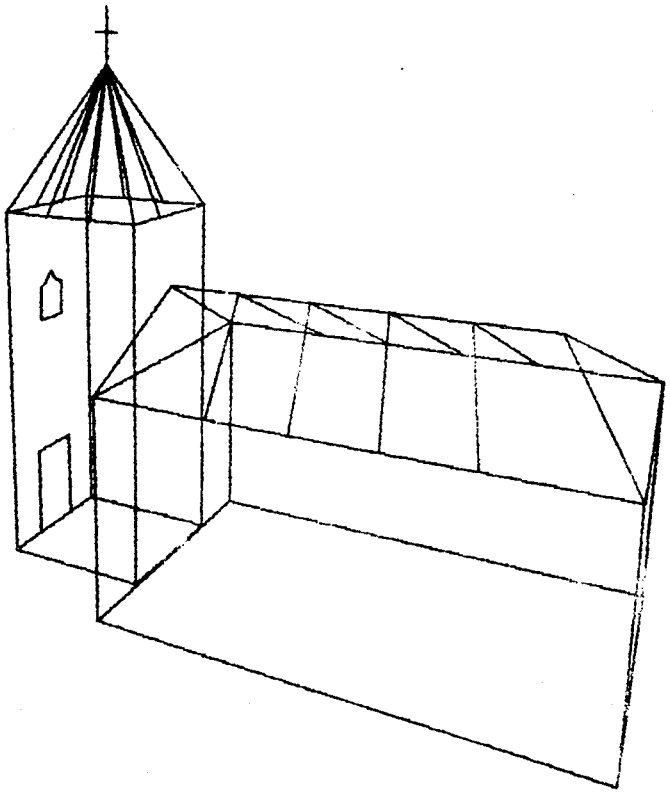


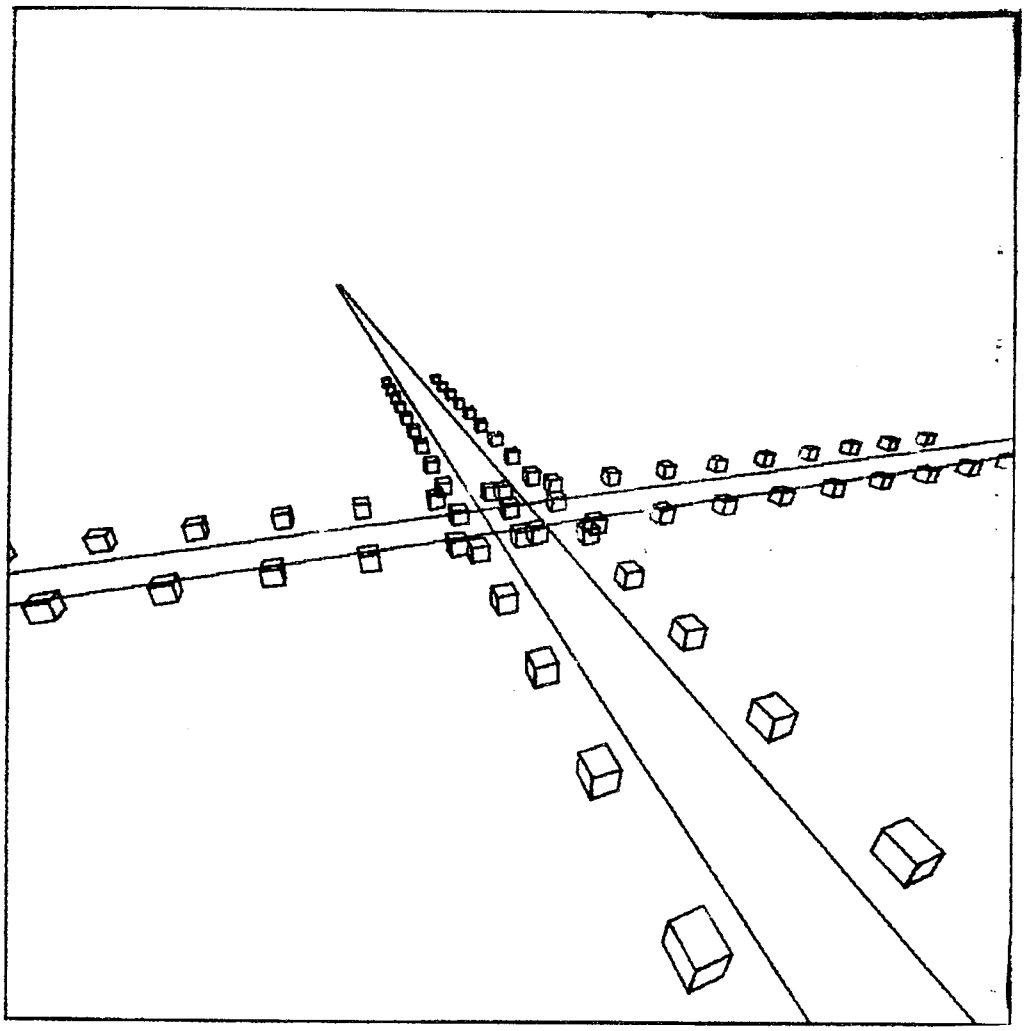




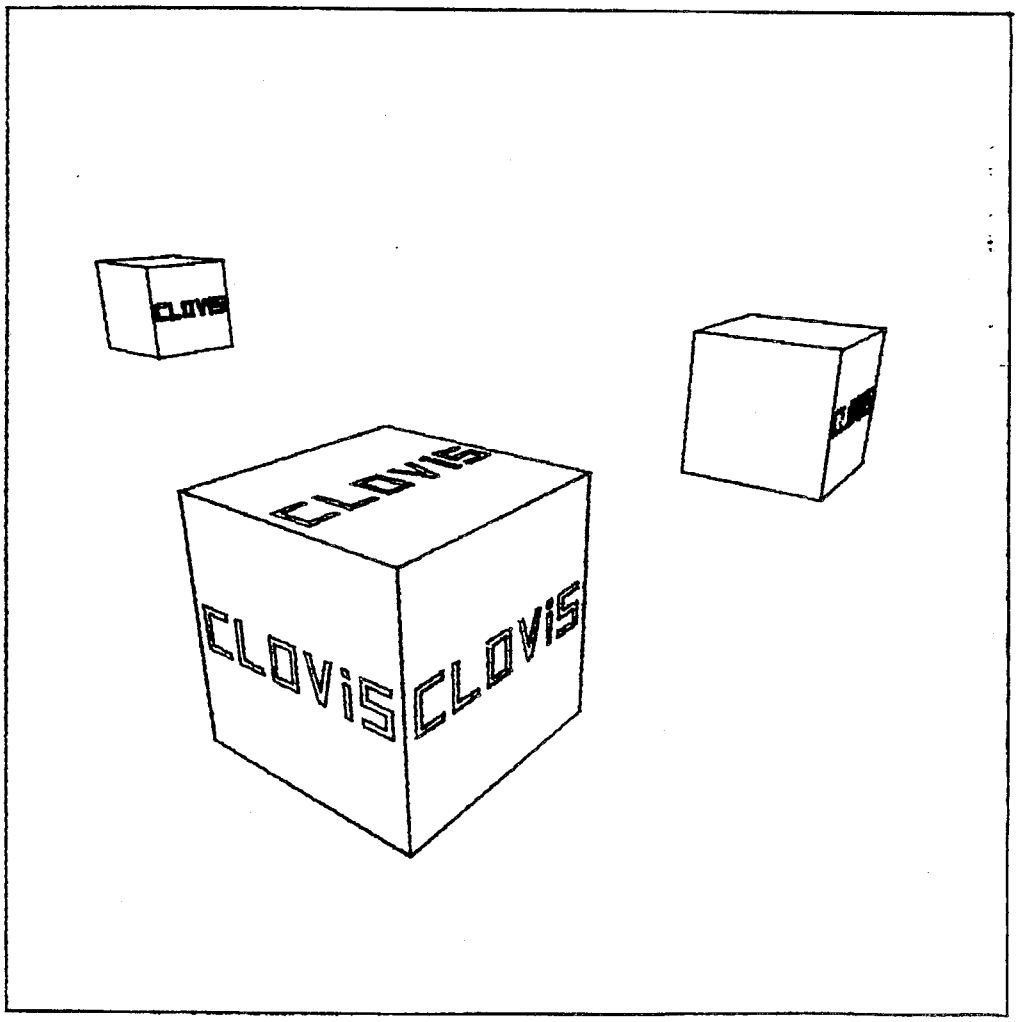








idOF SOtru aROchiv QOuit —>



3ème PARTIE

Vers un Logiciel Intégré pour la Visualisation et le Dialogue Interactifs Structurés

Chapitre VI : De CLOVIS à un système pour la Visualisation et le Dialogue Interactifs Structurés : Présentation Générale du Système

1 INTRODUCTION

Dans cette troisième partie nous présentons un second système graphique que nous avons conçu suite à l'expérience CLOVIS. Il s'agit d'un logiciel graphique destiné à piloter les matériels HELIOS - GETRIS (**G**énérateurs **E**n **T**emps **R**éel d'**I**mages **S**ynthétiques) issus de travaux effectués sosu la direction de F. MARTINEZ au laboratoire ARTEMIS [Fere 81], [Mart 84], [Zara 85],[Chib 86] et actuellement commercialisés par la société GETRIS-IMAGE. Ces systèmes se caractérisent par leur architecture matérielle modulaire et hautement reconfigurable, et qui en proposant plusieurs processus de visualisation sont capables de s'adapter aux différents besoins des utilisateurs.

Ces travaux s'appuient en grande partie sur les concepts que nous avions développés pour le projet CLOVIS. A partir des enseignements que nous avons pu tirer de cette première expérience et de ses limites (voir les conclusions du chapitre IV), trois points essentiels ont guidé notre travail :

- la nécessité de la prise en compte au niveau du système graphique des problèmes liés à la conception d'interface homme-machine graphiques. Cet aspect n'avait pas du tout été abordé dans CLOVIS, qui finalement s'était avéré mal adapté à la définition d'interfaces graphiques pour les applications (menus, ascenceurs.....). Il nous est apparu indispensable d'intégrer au système graphique des fonctionnalités permettant de définir de telles interfaces tout en tirant pleinement parti des possibilités offertes pour la visualisation d'objets graphiques.

- la nécessité d'une plus grande souplesse dans l'utilisation du système pour la définition d'objets graphiques. Rappelons que dans CLOVIS, toute visualisation d'objet nécessitait la définition d'une structure hiérarchique le décrivant, ce qui comme nous l'avons vu pouvait être parfois trop contraignant.

- la prise en compte au niveau du logiciel graphique des différentes possibilités du matériel, en particulier de son architecture banalisée entièrement reconfigurable.

Les deux premiers points répondent directement aux faiblesses que nous avions pu constater pour CLOVIS. Indépendants des problèmes matériels, ils concernent l'aspect fonctionnel du système graphique et l'architecture générale du logiciel. Ce sont essentiellement sur ces deux points qu'a porté notre travail. Nous reviendrons d'ailleurs séparément et beaucoup plus en détail sur les solutions adoptées pour ceux-ci dans les chapitres VII et VIII.

Dans ce chapitre VI, nous essayons de donner une vision d'ensemble du sytème graphique.

En premier lieu, nous nous attachons plus particulièrement à l'aspect concernant l'interface homme-machine. C'est lui qui présente le plus de "nouveautés" par rapport à ce que nous avions fait pour CLOVIS et qui a eu le plus d'influence sur la definition et l'organisation de ce second système. Cependant, la conception d'interfaces homme-machine dépasse largement le simple cadre de la synthèse d'images. Aussi, avant de présenter les propres solutions que nous avons adoptées, nous avons jugé nécessaire de dresser un rapide état de l'art des travaux effectués dans ce domaine. C'est à la lumière de cette étude que nous exposons de manière succinte l'approche que nous proposons pour la définition de l'interface graphique des applications à l'aide de notre système graphique (Le chapitre VII revient beaucoup plus en détail sur les solutions adoptées pour sa mise en oeuvre).

Ensuite, nous étudions briévement les points concernant la visualisation d'objets graphiques qui sont également développés de manière plus détaillée à l'intérieur du chapitre VIII.

Finalement, nous terminerons ce chapitre VI par une présentation rapide des matériels HELIOS-GETRIS et de leur prise en compte par le logiciel graphique. Ce bref exposé des possibilités du matériel sert à éclairer certaines des options que nous avons choisies et sur lesquelles nous reviendrons au cours des chapitre VII et VIII.

2 ETAT DE L'ART DANS LA CONCEPTION D'INTERFACES HOMME-MACHINE

La conception d'interfaces graphiques interactives est un problème fort complexe qui dépasse largement le cadre de la synthèse d'images. En effet, de par l'apparition de postes de travail aux possibilités de visualisation évoluées, et de par la popularisation des systèmes de multi-fenêtrage, les applications actuelles font de plus en plus appel des interfaces utilisateur sophistiquées présentées sous forme graphique. Les problèmes rencontrés pour répondre à de tels besoins sont loin d'être triviaux. Ils sont l'objet d'un nouveau thème de recherche qui a émergé ces dernières années : la conception d'interfaces homme-machine [Cout 87]. Aussi avant d'étudier la manière dont nous les avons abordés de façon spécifique pour notre système graphique, il nous semble important de présenter les travaux généraux effectués dans ce domaine.

2.1 Les problèmes liés à la réalisation d'interfaces utilisateur

Comme nous l'avons déjà souligné à maintes reprises, les progrès constants du matériel et des logiciels font que les systèmes informatiques actuels évoluent vers une plus grande interactivité. Parallèlement à cette évolution, les applications s'adressent de plus en plus à des utilisateurs non informaticiens, pour lesquels une bonne acceptation du système passe par la qualité de son interface. Aussi, la part accordée spécifiquement à l'interface utilisateur est -elle de plus en plus importante dans les applications actuelles (20 à 30 % du code d'après certaines études [SuSp 78]).

Malheureusement, bien souvent, le programmeur d'application ne dispose pas d'outils adéquats pour concevoir l'interface utilisateur. Il doit alors la construire entièrement à partir d'opérateurs de bas niveau. Ainsi, le code destiné à l'interface utilisateur est généralement dispersé parmi le code propre à l'application. Cet état de fait soulève plusieurs problèmes :

- la difficulté, lors de la mise en place de nouvelles applications, de **réutiliser** le travail effectué pour les applications déjà existantes.

- la difficulté d'effectuer un **prototypage** rapide de l'interface du système. En effet cette dernière conditionne de façon primordiale l'acceptation du système : il serait donc souhaitable de pouvoir la tester rapidement auprès des utilisateurs afin de pouvoir éventuellement l'adapter selon leurs réactions (conception itérative de l'interface).

- le risque de voir se développer une certaine **incohérence** entre les interfaces des différentes applications, chaque programmeur ayant ses propres techniques d'interaction et préférences. Ceci est préjudiciable à l'utilisateur final, obligé de passer par une phase d'apprentissage beaucoup plus importante lors de changements d'application.

Ces problèmes rendent la conception de bonnes interfaces utilisateurs une tâche difficile et complexe, avec comme conséquences immédiates un rallongement des **délais** et l'augmentation des **coûts de développement.** A titre d'exemple, la réalisation de l'interface homme machine de la station de travail Star de chez Xerox a nécessité pas moins de 30 homme-années [Hars 82].

De manière moins directe, ils peuvent également influer sur la **qualité des logiciels** produits (bien souvent, on constate que l'attention portée spécifiquement aux problèmes propres à l'application s'effectue au détriment de l'interface utilisateur ou inversement).

On remarquera que l'on retrouve ici, les mêmes types de difficultés que celles qui ont conduit au développement de systèmes graphiques généraux (voir chapitre II). Aussi depuis

quelques années, de nombreux efforts ont été consacrés à la définition et la réalisation de systèmes destinés à décharger les applications des problèmes liés à l'interface utilisateur. Dans les paragraphes qui suivent nous essayons de dresser un état de l'art qui mette en évidence les différentes approches en tentant de classifier un certain nombre de systèmes existants ou en cours de développement.

2.2 Classification des outils d'aide à la construction d'interfaces homme-machine

Les outils effectivement disponibles pour la construction d'interfaces utilisateur se présentent sous deux formes : les **boites à outils** ("toolbox") et les **systèmes génériques** [Cout 87].

2.2.1Les boites à outils (Toolbox)

Les boites à outils sont des bibliothèques de procédures destinées à l'écriture d'interfaces utilisateur. Elles offrent un évantail relativement large de primitives qui correspondent à différents niveaux fonctionnels. Les abstractions du toolbox ne sont pas organisées en couches : **toutes les primitives sont accessibles au programmeur**, quel que soit leur niveau. De manière très simplifiée, celles-ci peuvent être réparties selon deux catégories : la gestion du poste de travail et la gestion du dialogue.

Resource manager	Lecture, modificiation et ecriture des ressources
Quickdraw	graphique de base
Font manager	polices de caractères pour affichage de texte par Quickdraw
TextEdit	traitement de texte
Toolbox Event manager	file d'attente des événements (pression touche,bouton souris, insertion diquette...)
Window manager	Fenêtres (ouverture, deplacement, fermeture...)
Menu manager	Menus déroulants
Control manager	Contrôles : objets permettant de manipuler ou modifier information à l'aide de la souris ("boutons radio", barres de défilement...)
Dialog manager	fenêtres de dialogue et d'alerte
Desk manager	accessoires de bureau

- table VI.1 : les principaux gestionnaires du toolbox du Mac Intosh -

2.2.1.1 Gestion du poste de travail

Pour la gestion du poste de travail, les abstractions introduites ont pour but de définir un appareil logique dont le rôle est de masquer le fonctionnement du dispositif physique. Les postes de travail visés ne se limitent pas au graphique : ils sont multimédias, c.a.d. capables de gérer simultanément texte, graphique et son. Le toolbox habille et harmonise ces fonctions à l'aide de structures de données et de procédures d'accès.

D'un toolbox à un autre la nature des abstractions varie, mais de manière générale on retrouve les fonctionnalités suivantes :

- graphique de base (bitmap,viewport...)
- notion textuelles (polices de caractères...)
- notion liées au partage du terminal (fenêtre, évenement...).

La puissances des fonctions graphiques proposées à ce niveau, détermine fortement les qualités d'un toolbox. Ainsi le Mac Intosh doit en grande partie son succès à la richesse et à la qualité des fonctions graphiques de son toolbox intégré en ROM [AnDr 85].

2.2.1.2 Gestion du dialogue

Sur les fonctions de bas niveau évoquées ci-dessus, s'appuient les services de niveau dialogue. Les toolbox actuels offrent en général deux sortes d'entités :

- des entités simples (caractères, règles graduées, boutons, menus),

- des entités composées permettant de regrouper différentes entités simples (formulaires....).

Ces services évoluées, en raison de leur niveau d'abstraction tendent à définir un style d'interface homme-machine qui renforce la cohérence de présentation entre les différentes applications.

Les différences entre les toolboxs existant, se situent essentiellement sur la manière dont est effectué le contrôle, et sur les possibilités de modification des entités de dialogue :

Le contrôle peut être entièrement à la charge du programmeur "client "du toolbox, qui doit appeler explicitement les procédures de gestion des objets. C'est le cas par exemple pour le Mac Intosh [AnDr 85],[Rose 86]. A l'opposé, on peut citer X-toolkit [], où le contrôle est entièrement véhiculé par le toolbox : ce sont les objets qui invoquent automatiquement les procédures du programme client.

La modification des entités de dialogue n'est pas toujours possible (par exemple dans Andrew [Gosl 86]). Par contre, le toolbox du Mac Intosh fourni un certain nombre de points d'entrée qui permettent de modifer le comportement des entités de dialogue standards. Une autre particularité intéressante de ce dernier, est la séparation qu'il permet entre le code de l'application et la définition des entités de dialogue qu'elle utilise. Pour cela, les entités de dialogue (ou ressources) d'une application sont considérées comme des données et sont maintenues dans un fichier "ressources" distinct du fichier contenant le code de l'application. Les fichiers ressources conservent sous forme de caractères ASCII la description des différentes entités de dialogue : pour chacune d'elles sont fournis son type, son identification et les diverses informations liées à sa présentation (par exemple les textes des menu). Ces différentes descriptions peuvent ensuite être accédées à l'aide de primitives du toolbox qui permettent de reconstituer en mémoire centrale les ressources correspondantes. Cette séparation entre la description des entités de dialogue et le code des applications présente des interêts évidents : il est ainsi possible de rédéfinir et personnaliser la présentation des entités de dialogue sans avoir à recompiler l'application, cette rédéfinition pouvant être réalisée facilement à l'aide d'un éditeur interactif (éditeur de ressources).

2.2.1.3 Avantages/Inconvénients des "boites à outils"

Les toolboxs présentent l'avantage d'être facilement **extensibles**, le simple ajout de nouvelles fonctions permettant leur enrichissement. En proposant des niveaux d'abstraction très variés, ils offrent une grande **flexibilité** en laissant au programmeur le choix entre le plein contrôle de l'appareil et l'utilisation de services évolués. Par le biais d'entités de haut niveau pour la gestion du dialogue, ils améliorent de façon conséquente la **qualité** des interfaces et leur **cohérence** d'une application à une autre.

Néanmoins cette fléxibilité présente quelques contreparties. Ainsi, l'absence de structuration du toolbox risque de conduire à des architectures logicielles à la **modularité insuffisante** et d'aboutir à une mauvaise séparation entre le code d'une application et la définition des entités de dialogue qu'elle utilise. Par ailleurs, la diversité des fonctions proposées impose un **apprentissage** parfois **difficile** au programmeur.

Finalement, si les toolboxs sont un réel progrès vers la conception d'interfaces de qualité et cohérentes, ils n'intègrent pas complétement le fait que certains traitements et bien souvent le même modèle de contrôle du dialogue se retrouvent d'application en application.

2.2.2 Systèmes génériques

Les nombreuses études menées ces dernières années dans le domaine de la communication homme-machine, partent de la constatation que de très nombreuses fonctionnalités sont partagées par les interfaces de différentes applications. Il est donc tout à fait envisageable et **souhaitable** (!!) de considérer le dialogue homme-machine comme un **module indépendant** des applications et **partageable** par celles-ci.

C'est dans cette optique qu'ont été effectués des travaux en vue de proposer aux développeurs d'applications interactives des "outils" **génériques** d'aide à la réalisation d'interfaces homme-machine. Ces systèmes sont souvent conçus en dessus d'un toolbox. Mais, alors que ces derniers fournissent à l'implémenteur un ensemble de procédures standards, les systèmes génériques vont plus loin en proposant une **architecture standard**.

L'utilisation de systèmes génériques doit ainsi encourager la **séparation** entre la conception du dialogue et la conception des applications proprement dites et répondre pleinement aux problèmes soulevés au § 2.1, c'est à dire :

- améliorer la **cohérence** de dialogue interactif à l'intérieur et au travers des applications, les mêmes mécanismes étant utilisés partout,

- autoriser le **prototypage** et une implémentation rapide de l'interface utilisateur,

- faciliter la **maintenance** propre à l'interface,

- **répartir les tâches** de manière plus efficace entre les membres de l'équipe de développement,

- d'envisager la **distribution** des fonctions supportant l'interface utilisateur au travers de plusieurs systèmes ou processeurs...

2.2.2.1 Applications extensibles

Une première étape dans la réalisation de systèmes génériques a été la conception "**d'application extensibles**" qui simplifient l'usage d'un toolbox en regroupant sous la forme d'un squelette réutilisable et extensible le code implémentant les fonctions usuelles de l'interface homme-machine. Le tâche du programmeur consiste alors à remplir les "trous" et à redéfinir éventuellement les parties standards non adaptées à son application.

Dans cette catégorie de systèmes nous pouvons citer EZWin [Lieb 85] et surtout MacApp [Schm 86] environnement de développement sur MacIntosh.

En structurant les traitements communs à toutes les applications interactives, cette approche réduit sensiblement les coûts de développement et conduit à un style d'interaction très cohérent d'une application à une autre. Toutefois, le niveau d'abstraction de l'interface de programmation est encore relativement bas : la modification du squelette de base impose au programmeur la recherche dans le toolbox des fonctions nécessaires.

2.2.2.2 Systèmes de gestion de dialogue - UIMS

Alors que les applications extensibles proposent comme interface un langage de programmation usuel, d'autres systèmes vont plus loin dans l'abstraction en engendrant une application interactive à partir d'une **spécification**. Ces systèmes sont généralement désignés sous le terme de **S**ystèmes de **G**estion du **D**ialogue (SGD) et dans la litérature Anglo-Saxonne d'"**U**ser **I**nterface **M**anagement **S**ystem" (UIMS) [OBEK 84].

Dans de nombreux SGD, la spécification de l'interface est une description textuelle exprimée dans un langage spécialisé ([Jaco 85], COUSIN [Haye 85]) ou dans une notation extension de formalismes existants tels les automates d'états finis (USE [Wass 85],), les réseaux de Petri [Bart 86] ou les grammaires BNF (SYNGRAPH [Olse 83])... Cette approche, permet de séparer clairement la spécification de l'interface du développement propre à l'application, mais les formalismes proposés demeurent encore trop abstraits et se révèlent trop difficiles d'emploi pour être accessibles à des non imformaticiens.

Beaucoup plus prommetteurs sont les systèmes où la spécification est effectuée de manière interactive, tels Menulay [Buxton 83], SOS [Hullot 86], PERIDOT [MyBu 86], MIKE [OlDa 88] et ce que l'on pu voir tout récemment avec HyperCard []. Ceci permet au concepteur de voir immédiatement l'effet de ses directives, plus motivante elle permet d'atteindre plus rapidement le résultat recherché. Cependant les outils actuels de spécification interactive présentent encore de nombreuses limitations. Par exemple si le concepteur de l'interface peut construire interactivment une entité de dialogue composée, la définition d'une entité de dialogue élémentaire nécessite bien souvent une programmation traditionnelle (c'est le cas pour SOS). Plus complet est le système PERIDOT (**P**rogramming by **E**xample for **R**eal-time **I**nterface **D**esign **O**bviating **T**yping) qui utilise des techniques d'inférence basées sur des règles heuristiques permettant, à partir de techniques de manipulation directe, une programmation visuelle.

3 LA SEPARATION DES ENTITES GRAPHIQUES

Nous désignons par **entité graphique** un élément manipulable indépendamment des autres et visualisable sous une forme définie par les attributs graphiques qui lui sont associés.

Dans CLOVIS, nous ne considérions qu'**un seul type** d'entité graphique défini en regroupant à l'intérieur d'une même structure hiérarchique un ensemble d'attributs élémentaires appartenant aux classes M,G,A,Gv,Ga et E. Si cette solution rend possible un traitement des objets graphiques de manière uniforme au travers de primitives de haut niveau qui déchargent l'application de la gestion de processus de base de la synthèse d'images, elle s'est avérée toutefois inadaptée à la définition d'interfaces homme-machine. En effet, les fonctionnalités proposées par le système graphique, si elle permettent de définir une présentation visuelle pour l'interface utilisateur, ne prennent pas en compte la gestion du dialogue proprement dit (écho des dispositif d'interaction, interprétation des actions de l'opérateur correspondant aux déplacements et changements d'état de ces dispositifs....) qui est, elle, entièrement à la charge de l'application.

On retrouve donc les problèmes classiques liés à la difficulté de réalisation d'interfaces homme-machine (voir § 2.1), à la réutilisation de celles-ci au travers des applications et au manque de cohérence qui s'en suit.

C'est pourquoi, suite à cette première expérience et à la lumière des différents travaux que nous avons présentés au cours des paragraphes précédents, il nous est apparu nécessaire d'intégrer au sein du système graphique lui-même, un ensemble de fonctionnalités destinées à la définition et la gestion de l'interface des applications. Aussi, afin de répondre à cette nécessité avons-nous décidé de considérer dans ce second système deux types d'entités graphiques :

- des entités destinées à la définition de l'interface homme-machine que nous désignerons par **objets d'interface** ou **de dialogue**. Certaines de ces entités sont proposées de manière standard par le système (menus, ascenceurs...), de nouvelles peuvent être définies par les applications elles-mêmes,

- des entités destinées à la visualisation, et construites de manière spécifique par chaque application que nous désignerons par **objets graphiques**.

La distinction entre ces deux types d'entités doit ainsi permettre de répondre aux objectifs suivants :

- encourager pour les développeurs la séparation entre la définition de l'interface des applications et ce qui concerne la visualisation graphique proprement dite.

- décharger au maximum les applications des problèmes spécifiques de l'interface homme-machine par l'utilisation d'entités évoluées dediées aux échanges avec l'utilisateur (interaction ou visualisation), et leur gestion au travers d'un ensemble de primitives de haut niveau.

- encourager l'homogénéité et la cohérence des interfaces entre les applications.

L'emploi d'entités de dialogue manipulées par des primtives de haut niveau, est un approche semblable à celle proposée par les toolboxs. C'est en effet celle qui nous parait le mieux adaptée dans le cadre d'un logiciel graphique de base. Cependant, comme le lecteur aura loisir de le vérifier au cours des paragraphes qui suivent, nous avons tenu :

- à séparer clairement les fonctionnalités liées à la gestion du dialogue de celles propres à la visualisation d'objets graphiques (des primitives distinctes sont utilisées),

- à fournir un ensemble restreint de primitives pour la manipulation des objets de dialogue et assurer une certaine cohérence entre celles-ci, quel que soit le type de l'objet de dialogue,

- à assurer l'ouverture du système en permettant la définition de nouveaux objets de dialogue éventuellement composés (en regroupant hiérarchiquement différents objets de dialogue).

4 LES OBJETS D'INTERFACE

Les **objets d'interface ou de dialogue** sont le support des échanges (visualisation et dialogue interactif) entre l'application et l'utilisateur.

Il s'agit de véritables entités graphiques, dont la présentation visuelle est définie par un ensemble d'attributs qui, dans une très large mesure, reprennent ceux décrivant les objets graphiques. Mais à la différence de ces derniers, il est en plus attaché aux objets de dialogue un certain nombre d'attributs permettant de définir leur comportement lors de séances d'interaction. Ces attributs décrivent en particulier la présentation dynamique du dialogue (forme et aspect de l'écho, ...), ainsi que d'éventuelles opérations à effectuer en fonction des actions de l'opérateur (deplacement du dispositif d'interaction, changements d'état de celui-ci...). Ces attributs définissent ainsi la *sémantique* des objets destinés à l'interface utilisateur et permettent au système graphique de contrôler lui-même le dialogue.

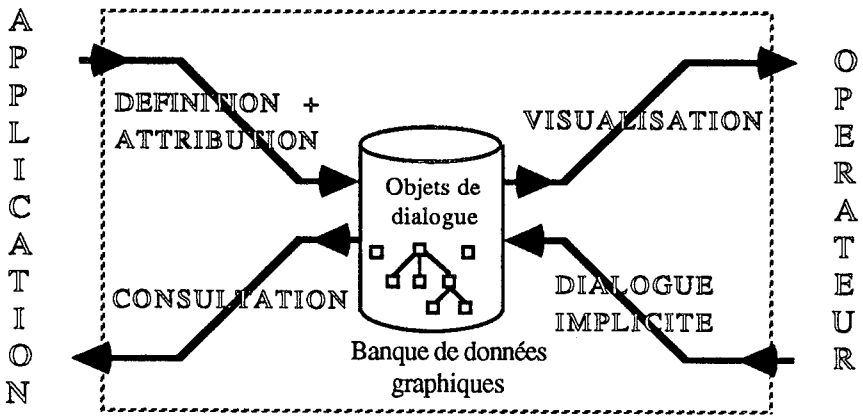
La gestion des objets de dialogue définis par une application est entièrement prise en charge par le système graphique (bien entendu sous le contrôle du programme d'application) qui conserve dans ses structures de données la description de ceux-ci. Les objets d'interfaces sont manipulés (créés, affichés, utilisés pour la visualisation d'objets graphiques ou lors d'une séance d'interaction...) par l'intermédiaire d'un ensemble de primitives de haut niveau qui déchargent au maximum l'application de leur gestion. Celles-ci peuvent être classifiées selon quatre principaux processus (fig. VI.1 p.236), analogues aux processus de base que nous avions mis en évidence pour CLOVIS. Il s'agit :

- de l'**attribution** qui permet au programme d'application de définir et modifier certains attributs d'un objet de dialogue,

- de la **consultation** qui permet au programme d'application de récupérer des attributs associés à un objet de dialogue,

- de la **visualisation** qui, sur la demande du programme d'application effectue l'affichage d'un objet de dialogue,

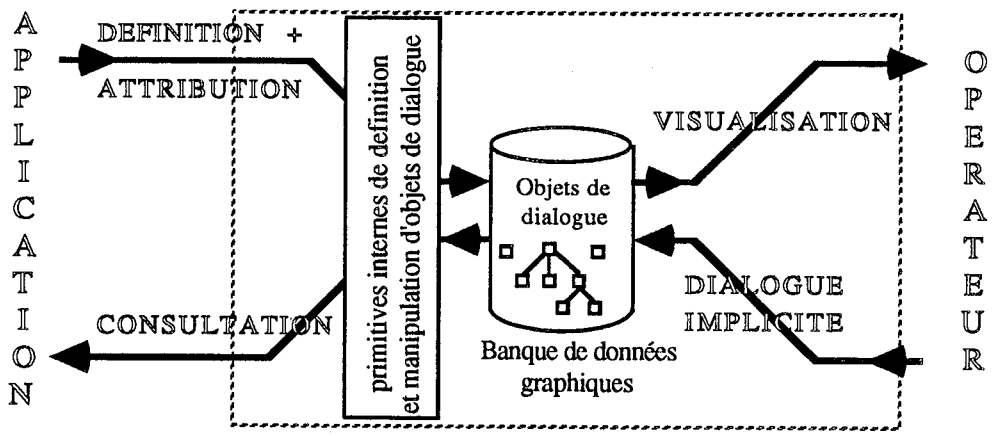
- du **dialogue implicite**, qui sur la demande de l'application consiste à l'interprétation automatique par le système graphique des actions effectuées par l'opérateur en fonction des attributs associés à un objet de dialogue.



- figure VI.1 : les processus de manipulation des objets de dialogue -

Les primitives de définition, attribution, consultation sont propres à chaque type d'objet de dialogue. La visualisation et le dialogue implicite sont effectués quand à eux de manière standard par le système graphique, indépendamment du type de l'objet de dialogue.

Un certain nombre d'objets de dialogue de base sont prédéfinis et proposés de manière standard par le système graphique. De nouveaux objets peuvent être définis soit à l'intérieur d'une application soit pour être intégrés dans les bibliothèques graphiques. Pour cela , un certain nombre de primitives de plus bas niveau sont proposées. Elles permettent la réalisation des primitives de définition, consultation, attribution associées à ces nouveaux objets (fig.VI.2). Lors de la définition de nouveaux objets de dialogue, il est possible, grâce à une structuration hiérarchique de ceux-ci, de réutiliser des objets déjà existants. Ainsi des objets d'interface complexes peuvent être construits à partir d'objets d'interface moins évolués.



- figure VI.2 : les différents niveaux de primitives pour la manipulation des objets de dialogue -

La notion d'objet de dialogue, et leur utilisation par des primitives indépendantes des applications permet de définir aisément et de manière cohérente l'interface des programmes d'application. Le dialogue implicite décharge ces dernières de l'interprétation, toujours difficile, des opérations effectuées sur ceux-ci par l'opérateur.

Les objets d'interface servent de base aux échanges entre l'application et l'opérateur. Ainsi, toutes les opérations de visualisation et de dialogue se font relativement à un objet d'interface courant. La désignation de cet objet s'effectue à l'aide d'un mécanisme de contexte qui associe les opérations ultérieures de visualisation/ou de dialogue à un objet de dialogue spécifique. De manière analogue à CLOVIS, les contextes peuvent être imbriqués permettant ainsi une programmation structurée.

Ce mécanisme de contexte est utilisé de manière interne par le système graphique, par exemple lors du dialogue implicite sur un objet d'interface, mais il sert également aux programmes d'application. Ainsi, parmi les objets de base proposés de manière standard par le système graphique, se trouvent des objets particuliers, les vues, sur lesquels le dialogue implicite n'est pas défini. Celles-ci sont à rapprocher des fenêtres des logiciels multi-fenêtres : elles servent de cadre aux opérations de visualisation d'objets graphique. Par ailleurs un dialogue de bas niveau, géré par l'application elle même, peut être effectué sur les vues. La désignitation par l'application des vues courantes pour la visualisation et pour le dialogue s'effectue au travers de la définition de contextes sur celles-ci.

5 LES OBJETS GRAPHIQUES

Les **objets graphiques** sont définis de façon spécifique par chaque application, leur construction s'effectue de manière "classique" à partir de primitives (morphologie de base) et d'attributs élémentaires. Cependant, seuls les attributs des classes Morphologie (M), Géométrie (G) et Aspect (A) sont considérés dans la définition des objets graphiques. Les attributs de Géométrie d'affichage (Ga), de Géométrie de prise de vue (Gv) et d'Eclairage (E) qui s'appliquent de manière globale à une scène sont reportés au niveau des objets d'interface, en particulier au niveau des vues. Ainsi un objet graphique sera visualisé avec les attributs globaux associés à sa vue d'affichage.

Avec CLOVIS, la manipulation des objets graphiques et de leurs attributs s'effectuait à l'aide des quatre processus de base, ATTRIBUTION, CONSULTATION, VISUALISATION et DESCRIPTION, articulés autour de la base de données graphique hiérarchique. Comme nous l'avons déjà évoqué, cette organisation s'est révélée trop contraignante et ne permettait pas toujours de répondre efficacement aux besoins des applications (cf. chapitre IV).

Aussi, ce nouveau système, s'il repose sur les mêmes concepts de base que CLOVIS, a surtout évolué vers une plus grande souplesse d'utilisation. En particulier , nous nous sommes attachés à laisser aux programmeurs d'applications une plus grande liberté dans la définition des objets graphiques afin de répondre au mieux à leurs besoins et leur niveau de compétence. Dans cette optique, le système offre **differents niveaux d'utilisation** qui autorisent plusieurs modes de définition des objets graphiques pouvant être employés de manière cohérente à l'intérieur des applications.

Dans l'utilisation la plus simple du logiciel , les attributs graphiques élémentaires sont définis sans aucune structuration. Les objets graphiques qu'ils décrivent sont **immédiatement affichés** dans le contexte de la vue courante au fur et à mesure de leur construction. Les attributs graphiques élémentaires ne sont pas mémorisés et l'espace mémoire alloué au moment de leur définition est aussitôt libéré après leur utilisation. Ce niveau autorise la définition d'images très simples ne nécessitant pas de modifications.

Un second niveau permet à l'application de **structurer** hiérarchiquement les attributs graphiques, facilitant ainsi la description d'objets complexes. Comme précédemment, de tels objets peuvent être affichés au fur et à mesure de leur construction sans être nécessairement mémorisés.

En effet, la **mémorisation** des attributs graphiques élémentaires et des objets (structurés ou non), n'est effectuée que si elle est demandée **explicitement**. Dans ce mode, tous les attributs élémentaires et les objets (structurés ou non) définis sont conservés dans la banque de données du logiciel graphique et peuvent être réutilisés par la suite pour construire de nouveaux objets et/ou pour l'affichage et/ou pour être modifiés.

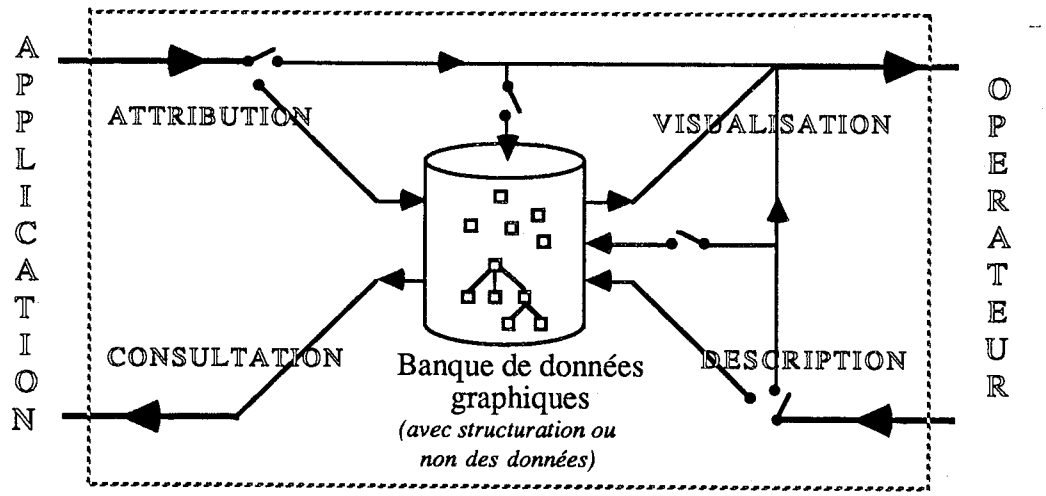
La mémorisation des objets graphiques peut être effectuée seule ou simultanément à la visualisation : dans le premier cas les attributs élémentaires et les objets définis sont uniquement mémorisés alors que dans le second cas ils sont en plus affichés au fur et à mesure de leur définition avec les mêmes règles que pour l'affichage immédiat.

Ces trois modes, visualisation immédiate, mémorisation et structuration des attributs graphiques peuvent être **combinés** entre eux (table VI.2). Leur utilisation judicieuse permet à tout moment à l'application de choisir le mode correspondant le mieux à ses besoins. Comme nous le verrons plus en détail par la suite, le passage entre ces différents modes s'effectue sous forme de contextes, la fin d'un mode provoquant le retour à l'environnement du mode précédent.

Affichage Immédiat	Mémorisation	Structuration
■		
■	■	
■		■
■	■	■
	■	
	■	■

- tablle VI.2 : Combinaisons des différents modes de définition des objets graphiques -

La possibilité d'utiliser et de combiner ces différents modes offre une plus grande souplesse au logiciel graphique. En particulier, ils introduisent une séparation moins rigide que dans CLOVIS entre la structuration des données graphiques et les processus d'attribution et de visualisation. En effet selon leur contexte d'utilisation, les mêmes primitives peuvent donner lieu à une simple **création** et **mémorisation** d'attribut ou d'objet, à une simple **visualisation**, à une **attribution** à un objet structuré,etc... Le processus d'attribution est ainsi généralisé et correspond à la définition d'attributs et d'objets graphiques par le programme d'application (fig VI.3).



- figure VI.3 : les processus pour la manipulation des attributs et objets graphiques -

De façon symétrique, la même généralisation a été apportée au processus de description : un attribut décrit peut être directement utilisé pour la visualisation et/ou mémorisé et/ou affecté à un objet structuré.

6 L'IMPLEMENTATION

L'implémentation de ce second système a été effectuée en vue de piloter le matériel HELIOS-GETRIS issus des travaux de F. MARTINEZ [Mart 84]. Comme CLOVIS, ce logiciel graphique se présente sous la forme de bibliothèques de primitives, accessibles aux applications au travers d'un langage de haut niveau.

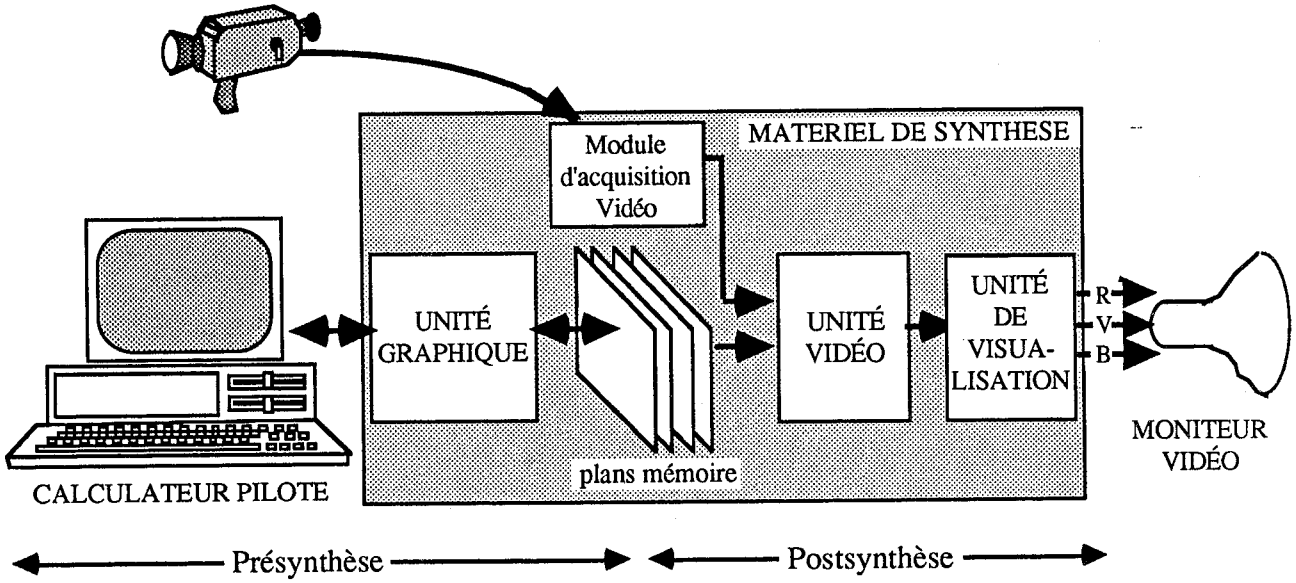
C [RiKe 84] est langage de programmation que nous avons utilisé pour le développement de ce logiciel. Notre choix s'est porté sur le langage C plutôt que PASCAL (qui lui avait servi à la réalisation de CLOVIS), car tout en offrant les possibilités des langages de programmation structurée, il permet l'emploi d'opérateurs proches des opérateurs matériels ("C est un assembleur portable" [Meye 88]). En particulier, il autorise un accès aisé et efficace aux ressources en mémoire par l'utilisation qui est faite des pointeurs et des opérateurs correspondants. Ces aspects du langage C permettent une programmation performante capable de tirer au mieux profit des capacités du processeur et du système hôte.

Le calculateur hôte sur lequel nous avons effectué ce développement est un micro-ordinateur HP Vectra compatible au standard IBM-PC-AT. Si dans un premier temps, nous avions envisagé l'utilisation de la machine au travers d'un système UNIX-like, nous nous sommes finalement porté sur la solution plus habituelle sur ce type de micro-ordinateur, à savoir l'emploi de MS-DOS.

Dans les paragraphes qui suivent, nous faisons une présentation rapide du matériel HELIOS-GETRIS. Nous exposons ensuite, de manière très rapide, la manière dont le logiciel graphique exploite les nombreuses possibilités qu'offre cette architecture banalisée.

6.1 L'architecture matérielle HELIOS-GETRIS

Les systèmes GETRIS (**G**énérateurs **E**n **T**emps **R**éel d'**I**mages **S**ynthétiques) issus en partie de travaux effectués au laboratoire ARTEMIS [Fere 81],[Zara 85],[Chib 86] et actuellement commercialisés par la société GETRIS-IMAGE se caractérisent par leur architecture matérielle modulaire et hautement reconfigurable, capable de s'adapter aux différents besoins des utilisateurs (fig. VI.4).



- figure VI.4 : l'architecture matérielle GETRIS -

Le **calculateur pilote** et **l'unité graphique** assurent la **présynthèse**, c'est à dire le remplissage des mémoires de trame. Celles-ci stockent les informations rattachées aux différents pixels constituant une image et servent de tampon aux modules de **postsynthèse**. Ces derniers, l'**unité vidéo** et l'**unité de visualisation** effectuent la conversion des informations numériques contenues dans les plans mémoire (et/ou fournies par le module d'acquisition vidéo) en l'information analogique acceptable par le moniteur TV, ceci bien entendu à la vitesse du signal vidéo pour les besoins de rafraichissement de l'image.

6.1.1 Les modules de présynthèse.

Le **calculateur pilote** (au standard IBM PC-AT) exécute le logiciel graphique de base et communique via une interface de bus spécialisée avec le matériel de synthèse.

L'**unité graphique** prend en charge les opérations de bas niveau. En particulier, elle comprend de nombreux automates cablés (pour les actions de base de lecture et ecriture dans les divers modules du système) qui permettent de très grandes vitesses d'écriture (13,5 Millions de points / s) lors du remplissage des mémoires d'image.

D'autre part l'unité graphique contient un ensemble de registres directement accessibles au calculateur pilote qui lui permettent de lire ou écrire dans les différents modules matériels et/ou de fixer des relations entre ceux-ci afin de configurer la machine.

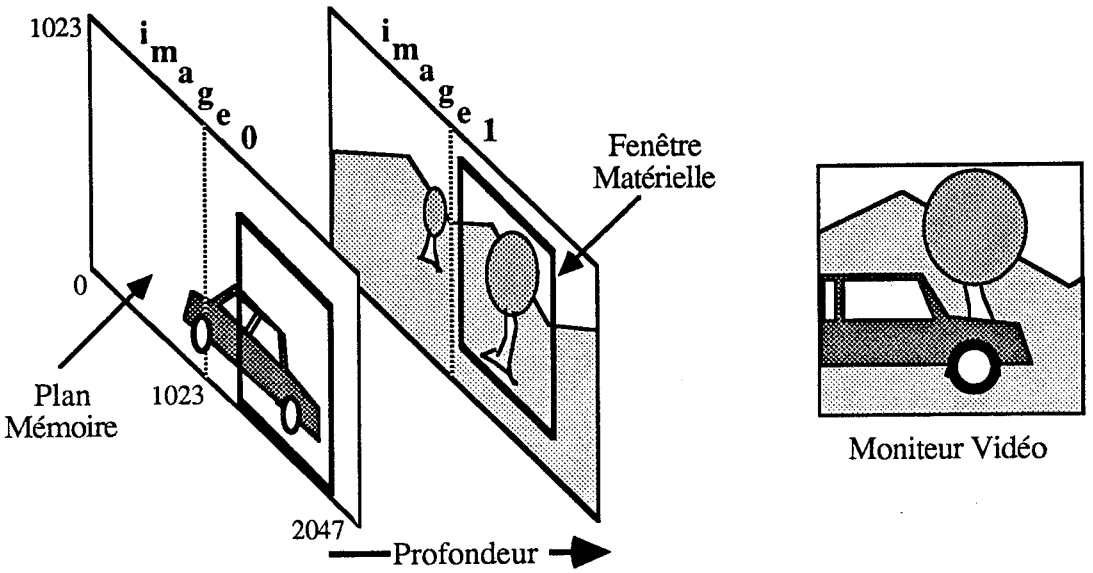
Une carte graphique spécialisée peut également être placée en option entre le calculateur pilote et l'unité graphique. Développée autour d'un processeur de traitement du signal ADSP 2100, elle prend en charge la majeure partie des opérations de base de la présynthèse (génération de droites, de cercles, de caractères, remplissage de polygônes, de tâches...) qui dans le cas standard sont effectuées par le processuer du calculateur pilote. Cette carte permet une amélioration des performances qui, selon les opérations, varie suivant un facteur compris entre 20 et 30.

6.1.2 La mémoire d'image

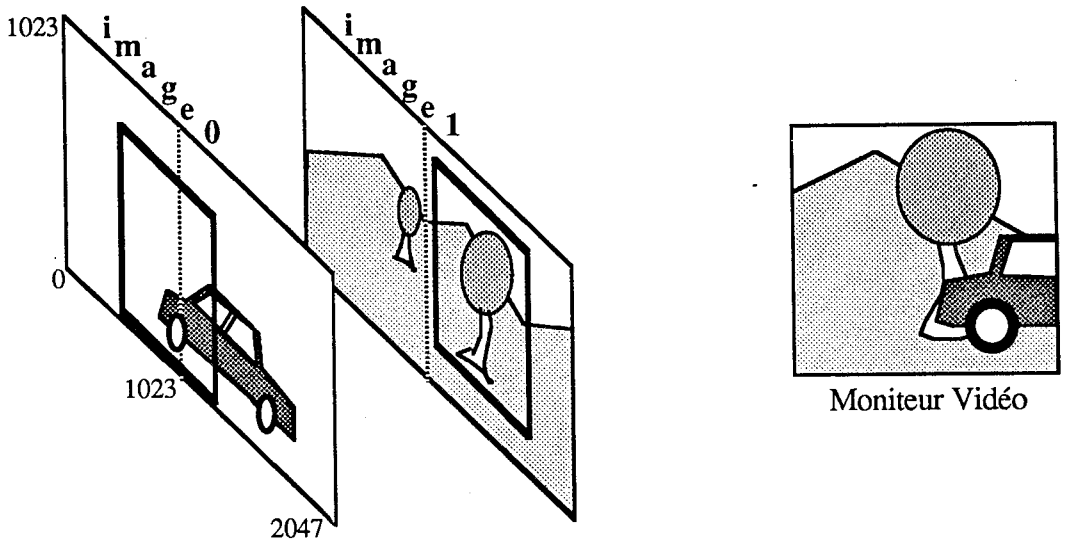
La mémoire d'image est constituée selon les besoins de l'utilisateur de un ou plusieurs modules **plan mémoire** (8 au maximum) d'une capacité de 1024x1024 points sur 12 bits chacun. La nature des informations contenues dans ces plans mémoire diffère selon la configuration du matériel et peut être par exemple une couleur (poids forts, poids faibles), un numéro interne d'identification des objets, une profondeur (Zbuffer)....

L'image finale affichée sur l'écran est constituée à partir de différentes "images logiques" (5 maximum) supportées par un ou plusieurs plans mémoire du système. Ainsi deux plans mémoires peuvent composer une "image" 2048 x 1024 x 12 ou bien deux "images" 1024 x 1024 x 12 de profondeur différente (fig VI.5). Toute une logique cablée associée aux modules plans mémoire permet de réaliser en temps réel des translations, fenêtrages, changement de visibilité, de priorité (profondeur) sur chacune de ces "images".

Grâce à un module **d'acquisition vidéo** (GAV) il est possible d'obtenir une image issue de la numérisation en temps réel d'un signal vidéo analogique. Ce module joue le même rôle que les plans mémoire à la différence que l'information qu'il fourni n'est accessible qu'en lecture.



- figure VI.5.a : Les plans mémoire et les "images" -



- figure VI.5.b : déplacement de la fenêtre matérielle -

6.1.3 Les modules de postsynthèse

Le rôle de l'**unité vidéo** est de fournir à l'unité de visualisation la couleur de chaque point de l'écran sous forme numérique. Elle contient différents modules qui selon les configurations permettent de transformer les informations fournies par les modules serveurs plans mémoire et GAV:

- des modules **tables de correspondance** d'une capacité de 4096 mots de 12 bits chacun peuvent être utilisés pour stocker, selon les configuration, des informations telles que la couleur, l'angle entre la normale d'une facette plane et une direction de source lumineuse...

- un module de **calcul d'éclairage** peut être ajouté pour les configurations 3D. C'est un opérateur cablé qui pour chaque pixel, calcule au rythme du signal vidéo sa couleur finale à partir de l'information de chrominance et de la normale de l'objet visible au point concerné, du modèle de réflexion spéculaire et diffuse associé à l'objet (16 modèles sont disponibles), de la couleur et direction d'une source lumineuse.

- un module de **calcul de textures** permet éventuellement d'associer des textures avec 4096 couleurs affichables à des objets 2D ou 3D (le pavage tridimensionnel des textures est effectué en temps réel). Les textures sont contenues à l'intérieur du module dans une banque téléchargeable et modifiable à tout instant. En version 3D, ce module fourni ses informations au module de calcul d'éclairage, les textures étant ainsi définies indépendamment de la position des objets dans l'espace.

L'intégration de ces différents modules au niveau de la postsynthèse permet un certain nombre de **modifications en temps réel** sur l'image affichée telles la substitution de couleurs, le changement de la couleur, de la texture d'un objet, le déplacement et le changement de la couleur de la source lumineuse, la modification des modèles de réflexion...

Finalement l'**unité de visualisation** se charge de la traduction des signaux numériques transmis par l'unité vidéo en signaux analogiques (suivant les composantes des couleurs primaires rouge, vert et bleu) acceptables par le moniteur TV.

6.1.4 Les différentes configurations matérielles

La grande originalité des systèmes GETRIS est la **banalisation** des modules matériels plans mémoire (GPM) et tables de correspondance (GTC) qui peuvent contenir selon les besoins des utilisateurs des informations de nature différente. Ainsi, avec un même matériel, l'utilisateur peut en le configurant différemment disposer de plusieurs processus de visualisation.

A chaque image logique (5 au maximum) qu'il défini l'utilisateur associe une configuration matérielle qui lui est propre et qui détermine le processus de visualisation utilisé. La table VI.3 ci-contre donne les différentes configurations possibles. Les ressources matérielles indiquées correspondent à une image 1024 x 1024, pour des images de taille plus importante il est nécessaire de multiplier le nombre de plans mémoires en conséquence.

Plusieures images logiques avec des configurations différentes peuvent être utilisées simultanément : les seules limites imposées sont fixées par les ressources matérielles disponibles sur la machine considérée. Il faut cependant rappeler que la grande modularité du matériel offre de nombreuses possibilités d'extension par simple ajout de cartes (cartes modules GPM, GTC, GCE (calcul d'éclairage)).

Lorsque plusieures images sont définies avec la même configuration, elles partagent par défaut les mêmes tables de correspondance. Néanmoins, l'utilisateur a la possibilité d'associer à une image sa ou ses propres tables.

6.2 La prise en compte de l'architecture banalisée par le logiciel

La gestion des différentes configurations matérielles est entièrement prise en charge par le logiciel graphique. Ceci permet au programmeur d'utiliser de manière complètement transparente les nombreuses possibilités du matériel. Les primitives de base du logiciel graphique sont indépendantes de la configuration utilisée. Ainsi, faire fonctionner une même application avec des configurations matérielles différentes ne présente aucune difficulté : il suffit de modifier à l'intérieur du programme la (les) ligne(s) correspondant à l'initialisation de la configuration courante. Par exemple, il est facile et immédiat d'utiliser un logiciel opérant avec une configuration sur 12 bits (4096 couleurs affichables simultanément) sur une configuration 24 bits (16 Millions de couleurs).

Nom de la configuration + Caractérisitiques	Memoire d'image	Autres modules matériels	Actions temps réel
COUL12 mémoire d'image avec 4096 couleurs simultanées	1 GPM : couleur sur 12 bits		- visibilité/invisibilité d'une couleur
COUL24 mémoire d'image avec 16M de couleurs simultanées	2 GPM : a) couleur poids forts b) couleur poids faible		- visibilité/invisibilité d'une couleur
COUL12+INDIR mémoire d'image avec table de couleurs (4096 couleurs)	1 GPM : index de couleur	1 GTC : table des couleurs sur 12 bits.	- modification de couleurs - visibilité/invisibilité d'une couleur
IDENT12 identification d'objets avec 4096 couleurs simultanées.	1 GPM : n° d'identification des objets	1 GTC : table des couleurs des objets sur 12 bits.	- couleur d'un objet - visibilité/invisibilité d'un objet
IDENT24 identification d'objets avec 4096 couleurs simultanées parmi 16 Millions.	1 GPM : n° d'identification des objets	2 GTC : a) couleurs poids forts b) couleurs poids faibles	- couleur d'un objet - visibilité/invisibilité d'un objet
PLANE Objets 3D définis par des facettes planes avec 4096 couleurs.	1 GPM : n° d'identification des objets (n° de facette)	2 GTC : a) table des couleurs des facettes (chrominance) b) luminance 1GCE : eclairage	- couleur d'une facette - visibilité/invisiblité d'une facette - modèle de reflexion - couleur, direction de la source lumineuse
PLANE+ZBUFF Objets 3D définis par faces planes, 4096 couleurs avec élimination des parites cachées (Z-Buffer).	2 GPM : a) n° d'identification des objets (n° de facette) b)profondeur des objets (Z-Buffer)	2 GTC : a) table des couleurs des facettes (chrominance) b) luminance 1GCE : eclairage	- couleur d'une facette - visibilité/invisiblité d'une facette - modèle de reflexion - couleur, direction de la source lumineuse
GAUCHE faces gauches 3D.	2 GPM : a) n° d'identification des objets b) normales	2 GTC : a) table des couleurs (chrominance) b) luminance 1GCE : eclairage	- couleur des objets - visbilité/invisibilité des objets - couleur, direction de la source lumineuse
GAUCHE+ZBUFF faces gauches 3D avec élimination des parties cachées.	3 GPM : a) n° d'identification des objets (n° de facette) b) normales c) profondeur des objets (Z-Buffer)	2 GTC : a) table des couleurs (chrominance) b) luminance 1GCE : eclairage	- couleur des objets - visbilité/invisibilité des objets - couleur, direction de la source lumineuse
VIDEO image vidéo analogique numérisée en temps réel avec 16 M de couleurs	1 GAV : module d'acquisition vidéo		- visibilité/invisibilité d'une couleur
VIDEO+INDIR image vidéo analogique numérisée avec indirection des couleurs (table de 4096 couleurs).	1 GAV : module d'acquisition vidéo	1 GTC : table des couleurs sur 12 bits	- visibilité/invisibilité d'une couleur - substitution de couleurs

GPM : module plan mémoire, 1024 x 1024 pixels sur 12 bits.
GTC : module table de correspondance, 4096 mots de 12 bits.
GCE : module de calcul d'éclairage en temps réel.
GAV : module d'acquisition vidéo couleur (en temps réel sur 24 bits).

- Table x : Les différentes configurations matérielles GETRIS -

- Table VI.3 : Les différentes configurations matérielles GETRIS -

Le choix des configurations matérielles associées aux "images logiques" que l'application désire manipuler, s'effectue à l'aide de la primitive

Images(conf0,conf1,conf2,conf3,conf4)

Par cette primitive le programmeur indique le nombre d'images logiques (5 au maximum numérotées de 0 à 4) qu'il veut utiliser et pour chacune d'elles la configuration souhaitée.

Le paramètre*confi* détermine la configuration de l'image *i*, son expression s'effectue par l'intermédiaire de constantes prédéfinies ***Coul12***, ***Coul24***.... (Ces constantes sont les mêmes que celles que nous avons utilisées pour désigner les configurations présentées dans la table VI.3). La constante ***Absent*** doit être utilisée pour terminer la liste des paramètres d'appel si moins de cinq images sont spécifiées.

Lorsques plusieurs images sont définies avec une même configuration, la constante ***Indep*** combinée par addition avec la constante définisant la configuration, permet de spécifier qu'une image doit posséder sa ou ses propres tables de correspondance (alors que par défaut les différentes images partagent les mêmes tables).

exemples :

Images(Coul24,Absent);

siginifie que seule l'image 0 est utilisée, ceci avec une configuration mémoire d'image avec 16 Millions de couleurs (deux plans mémoires sont nécessaires).

Images(Coul12+Indir+Indep, Coul12+Indir+Indep,Video,Absent);

indique que trois images sont utilisées. Les deux premières consistent en une mémoire d'image avec une indirection sur la couleur effectuée au travers d'une table 12 bits (4096 couleurs finales). Chacune d'elle possède sa propre table. La troisième image est obtenue à l'aide du module d'acquisition vidéo.

La primitive ***Images*** consiste en une simple déclaration des images et de leur configuration; elle ne provoque aucune initialisation effective du matériel. Ceci est réalisé suite à une demande explicite de l'application formulée au travers d'une seconde primitive

Init_Getris(Ecran,Raz)

Cette primitive permet à la fois :

- de fixer le format d'affichage utilisé sur l'écran de visualisation. Il est indiqué à l'aide du paramètre*Ecran*. (Quatre largeurs (512, 640, 704, 720) et deux hauteurs (512 et 574) sont possibles et exprimées à l'aide de constantes prédéfinies. Par exemple ***H512+L640*** signifie l'utilisation pour l'affichage d'une hauteur de 512 pixels sur une largeur de 640).

- d'initialiser (ou ré-intialiser) les divers modules matériels en fonction des configurations des images déclarées par la primitive ***Images*** qui doit avoir été invoquée au préalable. Une telle demande est indiquée à l'aide du paramètre *Raz* pourlequel la constante prédéfinie ***Image*** peut être utilisée. Outre la configuration des différents modules, l'initialisation provoque l'effacement des plans mémoires et des tables de correspondance.

- d'initialiser (ou ré-intialiser) la banque de données du logiciel graphique (le paramètre *Raz* a pour valeur la constante prédéfinie ***Liste***). Les objets et attributs précedemment définis et mémorisés sont perdus.

Il est possible d'effectuer simultanément l'initialisation des images avec celle de la banque de données du logiciel graphique en combinant par addition les constantes ***Liste*** et ***Image***.

7 CONCLUSION

Dans ce chapitre, nous avons tenté de procurer au lecteur une vision (très) globale du second système graphique que nous avons conçu suite à l'expérience CLOVIS, en mettant en avant les différents points qui ont motivés sa réalisation.

Le premier point a été la prise en compte par le système graphique d'un véritable dialogue interactif de haut niveau afin de permettre de développer facilement des applications possédant des interfaces évoluées et cohérentes. La solution que nous avons adoptée consiste en l'emploi d'**entités graphiques spécialisées pour le dialogue** (objets de dialogue ou objets d'interface) manipulées par l'intermédiaire de primitives de haut niveau permettant leur définition/attribution, consultation, visualisation et utilisation dans une séance d'interaction.

Cette solution s'apparente de par sa forme aux outils de type toolbox. Cependant, nous avons pris soin de limiter le nombre de primitives de manipulation des objets de dialogue, et de proposer par chaque type d'objet de dialogue (menu, acenseur ...) un ensemble de primitives **cohérentes**. Il est à souligner que la visualisation et l'interaction sont entièrement prises en charge par le système graphique, de manière identique quel que soit le type de l'objet de dialogue considéré. Ceci offre de larges possibilités d'**ouverture**. En effet il est relativement facile de concevoir de nouveaux objets de dialogue, pour cela il suffit de constuire les primitives les décrivant (définition de l'objet et attribution) et permettant d'accéder à l'information qu'ils véhiculent (consultation). Dans cette optique et afin de faciliter la description d'objets complexes et d'encourager la cohérence, le système propose un ensemble de primitives permettant la structuration hiérarchique d'objets de dialogue.

La définition d'objets de dialogue permet de définir la structure **statique** de l'interface d'une application. Le système graphique propose un ensemble de mécanismes, les **contextes**, destinés à expliciter **dynamiquement**, au moment de l'exécution d'un programme (application ou programme interne au système), les éléments de cette description statique concernés par les opérations d'échange avec l'opérateur (visualisation, interaction). Dérivés de la notion de contexte présente dans CLOVIS, ils permettent de définir aisément et sans risques d'effets de bords indésirables, des environnements particuliers pour les opérations de dialogue ou de visualisation. En effet, leur imbrication possible décharge l'application de la restitution de l'environnement initial au retour d'un contexte et encourage une programmation structurée.

Le deuxième point a concerné, la définition des **entités destinées à la visualisation** et désignées sous le terme d'**objets graphiques** (par opposition aux objets de dialogue dédiés aux échanges entre l'application et l'opérateur).

Tout d'abord, de par notre souci de séparer clairement leur définition de celle de l'interface , nous n'avons conservé pour leur description que les attributs qui leur sont propres : ceux des classes Morphologie, Aspect et Géométrie. Les autres attributs , Géométrie de prise de vue, d'affichage et Eclairage concernent leur présentation et ont donc été reportés au niveau des objets de dialogue.

Par ailleurs, à la lumière de l'expérience CLOVIS, nous avons également décidé de ne pas séparer aussi nettement les processus d'attribution/description et de visualisation et de ne pas imposer systèmatiquement la structuration hiérarchique des attributs graphiques afin de permettre une utilisation du **système** beaucoup **plus souple**. Pour les programmes d'application cela se traduit par l'utilisation possible et éventuellement conjointe de **trois modes** : visualisation immédiate, mémorisation et structuration hiérarchique des objets graphiques.

L'implémentation sur les systèmes HELIOS-GETRIS, nous a permi de valider cette approche. Par ces différentes évolutions, nous pensons avoir aboutit à un système graphique bien plus complet et souple que CLOVIS qui propose de manière cohérente l'ensemble des services nécessaires au développement d'applications graphiques interactives.

Chapitre VII : Les Objets d'interface

1 INTRODUCTION

Ce chapitre présente en détail l'ensemble des fonctionnalités que nous avons développées pour la prise en compte des problèmes liés à l'interface utilisateur évoqués au cours du chapitre précédent.

Dans un premier temps nous étudierons les points concernant la **description statique** de l'interface des applications c'est à dire la manière dont les entités destinées au dialogue sont décrites pour le système graphique.

Ainsi, nous présenterons tout d'abord le mécanisme qui est à la base de cette description et qui en associant à un **domaine** d'une image un ensemble d'attributs tant pour la représentation du dialogue que pour son interprétation permet la définition d'objets d'interface élémentaires.

Ensuite, nous verrons comment cette notion de domaine peut être généralisée et peut, au travers d'une structure hiérarchique, rendre possible la description d'objets de dialogue complexes.

Suivra, la présentation de l'ensemble des primitives proposées par le système graphique pour la manipulation des objets de dialogue ainsi décrits. Celles-ci se repartissent en deux groupes :
- les primitives destinées aux développeurs permettant la définition et la structuration de nouveaux objets de dialogue,
- les primitives destinées aux programmeurs d'application permettant l'utilisation d'objets de dialogue évolués. Ces primitives sont d'un niveau suffisament élevé pour décharger au maximum les applications de la gestion des objets d'interface.

La description des objets d'interface s'effectue de manière très similaire à la définition de classes dans les langages orientés objets. La structuration hiérarchique des domaines est à rapprocher de la notion d'héritage employée dans ces langages et permet une description rapide et aisée d'objets complexes. Les primitives des manipulation des objets d'interface correspondent aux méthodes associées aux objets instances d'une classe.

Pour conclure cette première partie nous étudierons de manière concrète, au travers d'exemples, la façon dont ces différents mécanismes sont utilisés. Deux types d'objets de dialogue particuliers seront présentés : les vues, objets de base dans la définition de l'interface des applications, et les choix, objets de dialogue évolués. Pour chacun d'eux nous exposerons la façon dont ils sont décrits (en particulier les choix qui utilisent la structuration hiérarchique des domaines) ainsi que l'ensemble des primitives destinées à leur manipulation par les programmes d'application.

La deuxième partie de ce chapitre présente les fonctionnalités proposées pour la **structuration dynamique** de l'interface, c'est à dire l'association, au cours de l'exécution d'un programme, des opérations d'échange avec l'utilisateur (visualisation ou dialogue) aux éléments défini par la description statique de l'interface. Ce contrôle dynamique de l'interface est basé sur la notion de contexte qui permet d'associer explicitement les opérations d'échange à un élément descriptif de l'interface. La définition et la possible imbrication de plusieurs contextes autorise une programmation structurée des applications.

Nous présenterons tout d'abord comment ce mécanisme de contexte est utilisé de façon interne par le logiciel graphique pour la réalisation des opérations de haut niveau que sont la visualisation et la gestion du dialogue implicite pour des objets d'interface. Ensuite nous verrons la manière dont les contextes sont utilisés à l'intérieur de programmes d'application, en particulier nous exposerons brièvement les opérations proposées aux utilisateurs pour la définition de leur propre dialogue. Finalement, nous étudierons comment cette notion de contexte a été généralisée pour les opérations de bas niveaux que sont les lectures et écritures en mémoire d'image.

2 OBJETS D'INTERFACE ELEMENTAIRES

2.1 Le mécanisme de base : la notion de domaine

Du point de vue du concepteur, la définition de l'interface utilisateur consiste à associer à différentes zônes de l'écran :

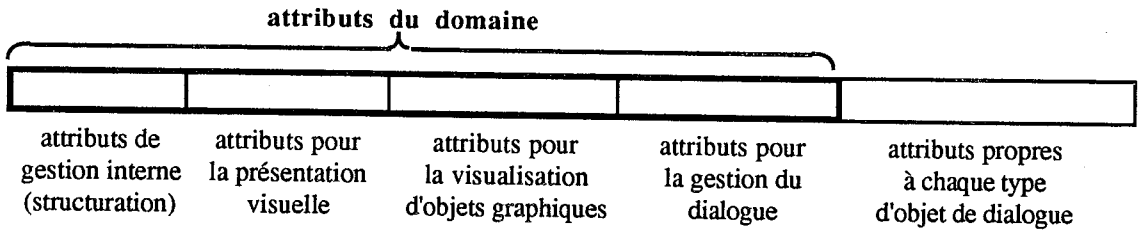
-une **présentation visuelle** matérialisant les différents éléments la composant,
-une **description de leur comportement** lors des échanges entre le programme d'application et l'utilisateur.

De cette constatation découle immédiatement la définition du mécanisme de base qui sert à la construction des objets de dialogue :

> un objet élémentaire de dialogue définit **un domaine** (région rectangulaire) sur une image logique qui **réagit de manière uniforme** lors des échanges entre l'application et l'utilisateur.

Chaque objet élémentaire de dialogue est représenté par un ensemble d'attributs regroupés à l'intérieur d'un seul descripteur (fig. VII.1). Une partie de ces attributs est commune à tous les objets élémentaires de dialogue qui partagent de nombreuses fonctionnalités. Ces attributs que nous désignerons comme étant les attributs du domaine permettent de définir :

- une représentation visuelle matérialisant l'objet de dialogue lors de son utilisation (langage de présentation),
- le comportement du domaine pour la visualisation d'objets graphiques (soit pour la propre présentation visuelle de l'objet de dialogue, soit dans le cas où comme nous nous l'avons déjà vu l'objet d'interface sert de support pour la visualisation d'objets graphiques propres à l'application)
- le comportement du domaine lors d'opérations de dialogue, c'est à dire lors de l'utilisation et de l'activation des dispostifs d'interaction sur la zône correspondante,
- des attributs pour la gestion interne et la structuration des objets élémentaires de dialogue.



- figure VII.1 : structure générale des descripteurs d'objets élémentaires de dialogue -

Aux attributs de domaine partagés par tous les objets de dialogue élémentaires s'ajoutent éventuellement d'autres attributs spécifiques à chaque **type** d'objet élémentaire dont ils complètent la description en les particularisant.

La notion de domaine est le mécanisme de base qui en regroupant de façon standard la majeure partie des attributs décrivant les objets de dialogue, permet comme nous le verrons par la suite de définir et traiter aisément et de manière **uniforme** ces derniers.

Comme nous pouvons le remarquer, le support de l'interface utilisateur, à savoir la notion de domaine est directement lié à la notion d'image logique, ce qui somme toute est naturel : les images logiques étant une description structurée de l'image finale visualisée sur l'écran qui constitue la partie visible et active de l'interface à un instant donné.

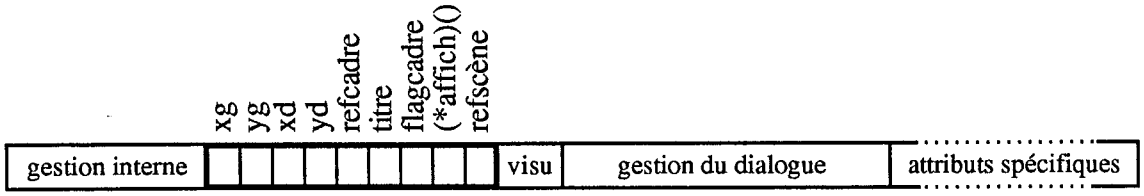
2.2 Les attributs des domaines

Dans les paragraphes qui suivent nous reprenons en détail les attributs rassemblés au sein des différentes catégories que nous venons d'exposer. Cependant nous nous intéresserons plus particulièrement aux attributs ayant trait à l'utilisation des objets de dialogue pour la définition de l'interface utilisateur. Aussi reportons nous la présentation des attributs destinés à la gestion interne des domaines au paragraphe 4.2.1.

2.2.1 Attributs pour la présentation visuelle des domaines

Une domaine défini une région rectangulaire sur une image. Les limites de cette région, **zône utile du domaine**, sont exprimées à l'aide des coordonnées (dans le repère de l'image) de son coin inférieur gauche (xg,yg) et de son coin supérieur droit (xd,yd).

Nous présentons dans ce qui suit les différents attributs qui permettent d'associer une présentation visuelle à un domaine (fig. VII.2).



- figure VII.2 : attributs pour la présentation visuelle des domaines -

2.2.1.1 Matérialisation de la zône utile (cadres).

Un attribut de type **cadre,** permet de matérialiser la zône utile d'un domaine, il défini :

- une couleur de fond pour la zône utile du domaine,
- un dessin permettant éventuellement de marquer les limites du domaine.

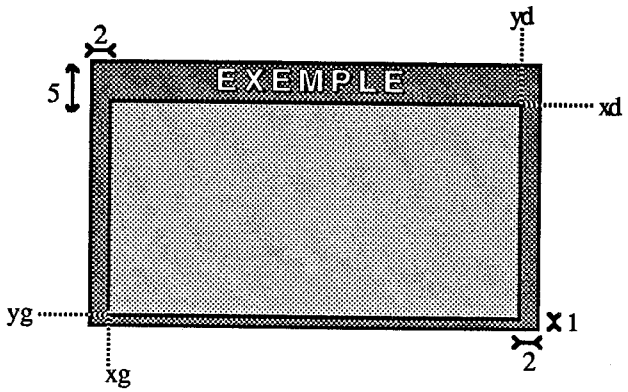
Les attributs de type cadre sont définis indépendamment des domaines, seule leur référence (**refcadre**) est conservée au sein des descripteurs de domaine (ainsi un même cadre peut être partagé par plusieurs domaines simultanément).

Un cadre est décrit à l'aide d'une primitive, ***DefCadre***, qui indique son type (l'adresse d'une fonction de génération pour son affichage) et la largeur de ses différentes marges, dessinées autour de la zône utile du (des) domaine(s) au(x)quel(s) il est associé (fig VII.3).

Bien entendu, la primitive ***DefCadre*** renvoie la référence de l'attribut ainsi créé et mémorisé.

Une seconde primitive (***AspCadre***) permet de définir et modifier à tout moment les différents aspects nécessaires à la visualisation d'un cadre et qui sont :

- un indicateur de visibilité ou non pour le fond (zône utile du domaine),
- une couleur pour le fond,
- une couleur pour les marges,
- une couleur pour le titre du (des) domaine(s) au(x)quel(s) il est associé. Le **titre** (chaîne de 30 caractères maximum) est un attribut des domaines qui est affiché avec le cadre (par exemple dans le cas des cadres du type prédifini *CadreStd*, le titre est centré dans la marge haute).



- figure VII.3 : matérialisation d'un domaine à l'aide d'un attribut cadre -

La figure VII.3 ci-dessus illustre l'utilisation d'un cadre pour la présentation d'un domaine dont les limites sont définies par son coin inférieur gauche (xg,yg) et son coin supérieur droit (xd,yd).

En plus de la référence du cadre (***refcadre***) le descripteur de domaine contient un indicateur ***flagcadre*** qui permet de spécifier quelles parties du cadre doivent être affichées lors de la visualisation du domaine. Des constantes prédéfinies sont utilisées pour le positionnement de ***flagcadre*** : ***tout*** pour demander l'affichage du cadre dans son entier, ***néant*** qui signifie pas d'affichage du cadre, ***zofond*** qui indique que seule la zône de fond du cadre doit être affichée (cette valeur de ***flagcadre*** permet d'éviter une régénération complète du cadre si uniquement des modificaton du contenu de la zône utile du domaine ont été effectuées),....

2.2.1.2 Utilisation d'objets graphiques.

Afin de compléter la présentation visuelle d'un domaine il est intéressant de tirer parti de la puissance offerte par le logiciel pour la description et la visualisation d'objets graphiques (voir chapitre VIII).

Pour cela est associé au domaine un attribut ***(*affich)()*** qui est l'adresse d'une procédure d'affichage. Celle-ci est invoquée automatiquement lors de la visualisation du domaine. Elle permet ainsi l'utilisation d'objets graphiques pour le présentation visuelle du domaine. Ceux-ci peuvent être soit des objets procéduraux (ils sont alors définis explicitement dans la procédure ***affich***), soit des objets mémorisés, élémentaires ou structurés (scènes), dont la référence (***refscène***) est également un attribut du domaine.

2.2.2 Attributs pour la visualisation d'objets graphiques.

Les attributs globaux (Gv, Ga et E) destinés à la présentation des objets graphiques ayant été retirés de la structure hiérarchique des objets il est naturel de les associer aux domaines.

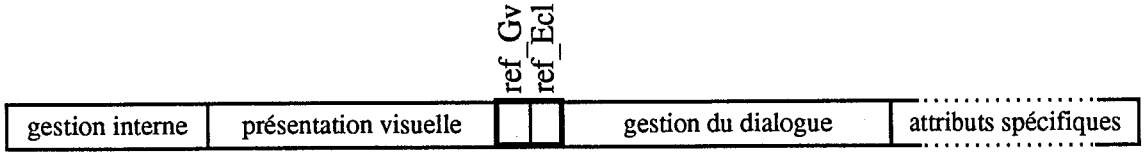
En effet, d'une part les objets d'interface, comme nous venons de le voir, peuvent utiliser des objets graphiques pour leur propre présentation visuelle, mais d'autre part, servant de support aux échanges entre l'application et l'utilisateur, ils peuvent être destinés plus particulièrement à la visualisation d'objets graphiques (par exemple les objets d'interface de type fenêtre de visualisation que nous présentons au § 5.1).

L'attribut Ga est implicitement définit avec le domaine : la zône utile du domaine constitue la cloture dans laquelle sont affichés les objets graphiques lorsqu'ils sont visualisés dans le contexte du domaine.

Les attributs de prise de vue (Gv) et d'éclairage (E) sont analogues à ceux que nous avons définis pour CLOVIS (voir § 4.4.4 et 4.4.5 du chapitre IV). Comme pour les cadres, ils sont

définis indépendamment des domaines et seules leur références (***ref_Gv*** et ***ref_Ecl***) sont conservées à l'intérieur des descripteurs de domaine (fig. VII.4).

Ces attributs s'appliquent **par défaut** à tous les objets graphiques visualisés dans le contexte du domaine, à moins que l'application ne définisse explicitement un nouvel attribut.

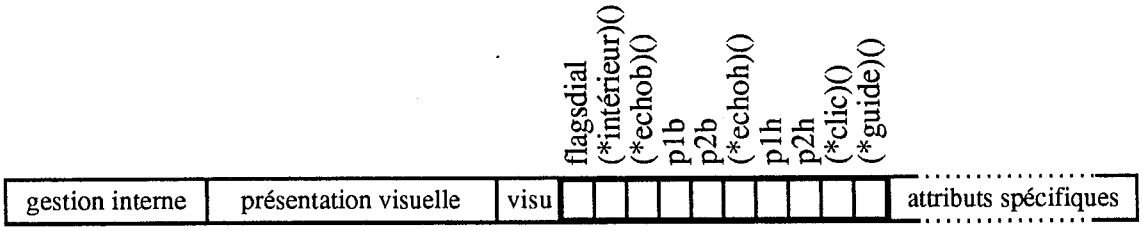


- figure VII.4 : attributs des domaines pour la visualisation d'objets graphiques -

2.2.3 Attributs pour la gestion du dialogue.

Les attributs définissant le comportement d'un domaine pour le dialogue (fig. VII.5) sont de trois types :

- des attributs permettant d'activer ou désactiver le domaine pour le dialogue.
- des attributs définissant l'apparence visuelle du dialogue : c'est à dire la forme et la couleur de l'écho du dispositif de dialogue lorsque celui-ci se situe sur la zône utile correspondant au domaine.
- des attributs définissant l'interprétation des événements du dialogue intervenant sur le domaine, c'est à dire les opérations (actions) à réaliser lors de changements d'état du dispositif d'interaction.



- figure VII.5 : attributs des domaines pour la gestion du dialogue -

2.2.3.1 Validité du dialogue

(*intérieur)() : est l'adresse d'une fonction permettant d'identifier l'intérieur du domaine (cette fonction a été prévue en particulier afin de traiter des domaines non rectangulaires), cette fonction est appelée directement par les primitives internes du logiciel graphique.

flagsdial est un indicateur qui détermine pour quels états du dispositif d'interaction le domaine intervient lors d'une séance de dialogue (c'est à dire les états pour lesquels les attributs de dialogue du domaine sont pris en compte).

La tablette à digitaliser étant le dispositif de dialogue privilégié nous ne considérons que deux états qui correspondent au stylet levé (dialogue en position haute) et au stylet enfoncé (dialogue en position basse). Nous avons étendu ces états aux autres dispostifs d'interaction disponibles, souris et clavier, où la position basse indique respectivemenent la pression du bouton de "clic" et l'enfoncement d'une touche.

Le positionnement des indicateurs de *flagsdial* peut être effectué par l'intermédiaire des constantes prédéfinies :

haut : dialogue valide lorsque le dispositif est en position haute
bas : dialogue valide lorsque le dispositif est en position basse
tout : qui correspond à ***haut*** + ***bas*** et indique que le dialogue est valide sur le domaine quelque soit l'état du dispositif d'interaction.

2.2.3.2 Présentation du dialogue

Les **attributs d'echo** lient les déplacements du dispositif d'interaction sur le domaine à un ou plusieurs témoins. Ils permettent de spécifier séparément et donc de distinguer deux formes différentes d'écho suivant que le dispositif de dialogue est en position haute ou en position basse.

(*echob)() et ***(*echoh)()*** sont les adresses de fonctions d'écho associées au dispositif d'interaction lorsqu'il est respectivement en position basse ou en position haute. Ces fonctions, chargées de l'affichage de l'écho, sont invoquées automatiquement lors des séances de dialogue.

L'utilisateur peut décrire ses propres fonctions d'écho ou utiliser des fonctions prédéfinies (EchoNul, Curseur, Fantôme, Elastique, Pinceau, Aero...).

L'appel des fonctions d'écho s'effectue avec quatre paramètres :

- l'**étape** du dialogue qui permet de distinguer le moment de l'interaction auquel correspond l'appel de la fonction d'écho et donc d'effectuer éventuellement en conséquence un traitement particulier. Ce paramètre est postionné automatiquement par le processus gérant le dialogue qui reconnait trois étapes différentes définies par les constantes :

  • ***premier*** qui correspond au début d'une séquence d'interaction sur le domaine et donc au premier appel de la fonction d'écho (par exemple suite au passage du dispositif de la position haute à la position basse sur le domaine),
  • ***interm*** lorsqu'il n'y a pas eu de changement d'état du dispositif de dialogue qui est toujours sur le (même) domaine,
  • ***dernier*** suite à un changement d'état du dispositif de dialogue (par exemple passage de la position basse à la position haute) ou bien à un changement de domaine. (Cela permet de terminer correctement l'écho (par exemple effacement du curseur) avant l'appel d'une nouvelle fonction).

- la **référence** du domaine de dialogue. Cela permet à la fonction d'écho d'aller éventuellement chercher des informations propres au domaine auquel elle est associée.

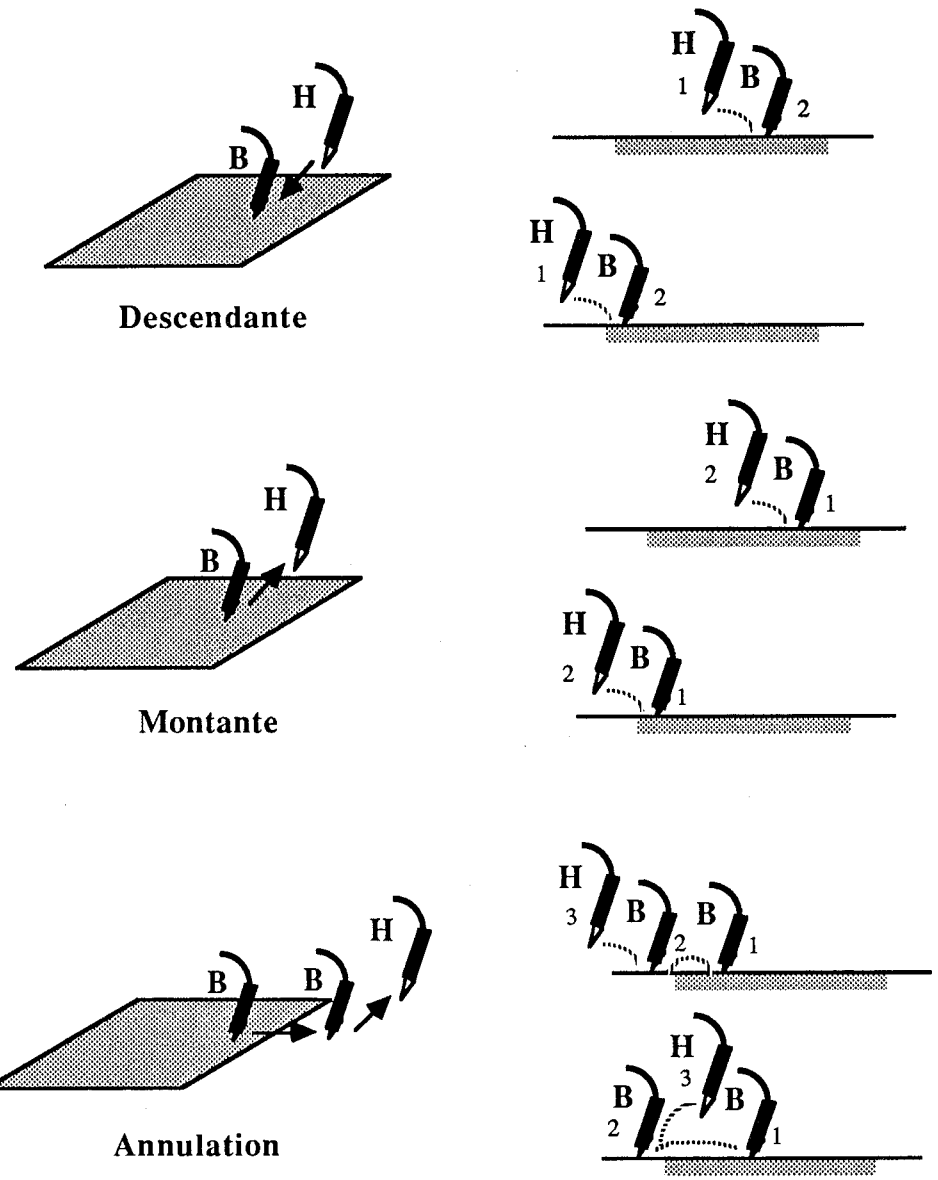
- deux autres paramètres sont prévus, leur signification est dépendante de la fonction d'écho. Par exemple ils sont utilisés par les fonctions d'écho standard proposées aux utilisateurs (Curseur, Fantôme ....) pour définir un motif et la couleur du curseur. La valeur de ces attributs est définie au niveau du descripteur de domaine dans les champs ***p1b***, ***p2b*** pour l'écho en position basse et ***p1h***, ***p2h*** pour l'écho en position haute.

(*guide)() est l'adresse d'une fonction permettant de définir des contraintes ou des fonctionnalités particulières sur l'écho et la collecte de coordonnées lorsque le dialogue a lieu sur le domaine. Par exemple il est possible de définir le déplacement de l'écho en fonction d'une grille, ou bien de déplacer l'écho horizontalement ou verticalement seulement, etc... La fonction prédéfinie ***NoGuide()*** permet des déplacements libres de l'écho (pas de contraintes sur les coordonnées récupérées).

2.2.3.3 Interprétation du dialogue

L'attribut ***(*clic)()*** est l'adresse d'une fonction qui traite la "sémantique" même du domaine pour le dialogue. Cette fonction est automatiquement invoquée lorsque des **événements** interviennent sur le domaine durant une séance de dialogue, en particulier

lorsque le dispositif d'interaction change d'état. L'appel de cette fonction s'effectue avec deux paramètres qui sont le type d'événement (ou transition) et l'adresse du descripteur du domaine à laquelle elle est rattachée.



- figure VII.6 : les transitions pour les différents événements de dialogue -

Trois types d'événements de dialogue sont considérés et désignés par les constantes prédéfinies (figure VII.6) :

- ***Descendant*** qui indique que le dispostitif de dialogue est passé de la position haute à la position basse.

Dans le cas contraire où le dispositif passe de la position basse à la position haute, on distingue deux transitions :

- ***Montant*** si le domaine sur lequel est positionné le dispositif d'interaction est le même que lors de la précédente transition descendante,

- ***Annulation*** sinon (le dispositif a été déplacé en position basse hors du domaine pour être ensuite levé).

2.2.4 Attributs spécifiques à chaque type d'objet d'interface

Les attributs précédents sont communs à tous les domaines définis par l'application. Il est nécessaire de leur ajouter un ensemble d'attributs spécifiques qui varient entre les domaines selon l'utilisation qu'il en est faite. Par exemple, pour un domaine employé pour définir un menu, il est possible d'associer un ensemble de chaînes de caractères correspondant au texte des différents choix possibles.

3 OBJETS D'INTERFACE STRUCTURES

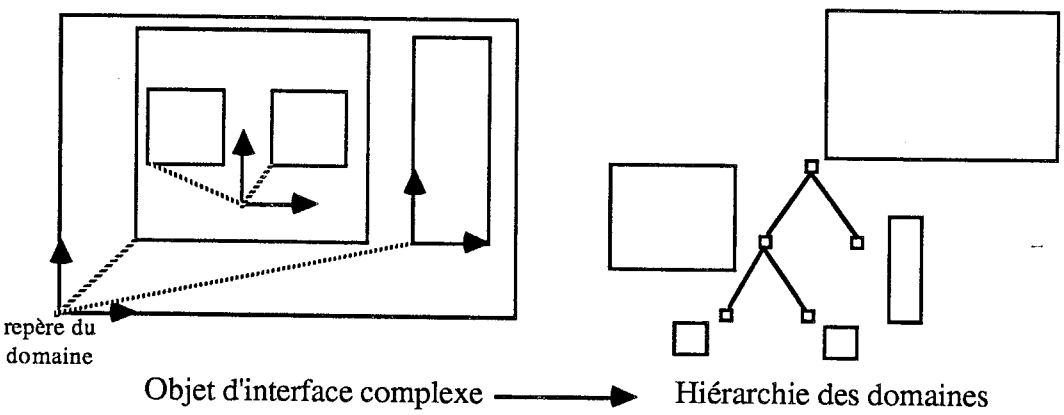
3.1 la structuration arborescente des domaines

Dans ce qui précède nous avons présenté les objets de dialogue élémentaires (définis à partir de la notion de domaine) ainsi que les attributs qui les caractérisent.

En fait, dans la réalité les objets d'interface sont en général plus complexes : bien souvent ils sont composés d'un ensemble de zônes qui réagissent différemment. La seule notion de domaine n'est plus suffisante : il est nécessaire de regrouper en une seule entité (que nous nommerons entité domaine) un ensemble de domaines qui peuvent ainsi être manipulés simultanément.

Aussi, pour définir des objets de dialogue complexes, nous avons adopté une **structuration arborescente** des domaines. Un objet d'interface est défini par une hiérarchie de domaines où chaque domaine est défini relativement à un domaine père englobant. Les coordonnées de la "zône utile" du domaine sont exprimées dans le repère du domaine père. Le repère d'un domaine est défini relativement au coin inférieur gauche de sa zône utile (par défaut il correspond à ce coin inférieur gauche, mais il peut être ensuite modifié selon les besoins de l'utilisateur).

La figure VII.7 ci-dessous montre la manière dont un objet d'interface complexe peut être modélisé par une hiérarchie de domaines.



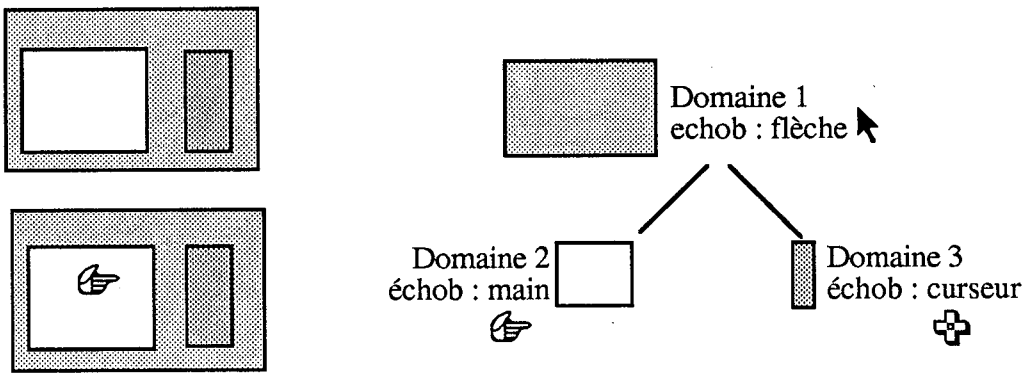
- figure VII.7 : Structuration arborescente des domaines -

3.2 Les avantages de la structuration arborescente des domaines

Outre le fait qu'elle permet de regrouper en une seule entité un ensemble de régions d'une image, la structuration arborescente des domaines présente plusieurs intérêts :

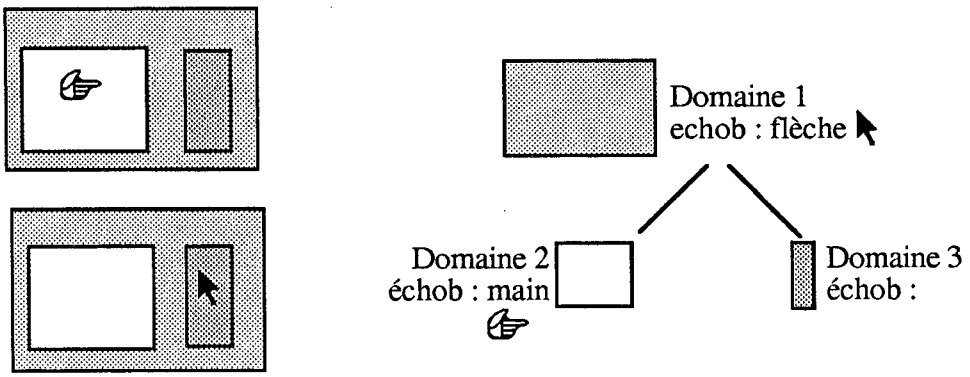
- elle permet la **réutilisation** d'objets d'interface précédemment définis pour la construction d'objets plus complexes. Il suffit d'insérer les premiers au sein de la structure arborescente des seconds.

- elle induit un ordre implicite de **priorité** pour la visualisation et surtout pour la prise en compte des domaines lors du dialogue. Les domaines sont affichés dans l'ordre correspondant au parcours infixé (racine, gauche, droite) de l'arborescence les regroupant. C'est le domaine situé le plus bas et le plus à droite qui est affiché en dernier. De manière symétrique, lors d'une séance de dialogue, c'est le domaine de niveau le plus bas sur lequel est positionné le dispositif d'interaction qui est considéré. Ainsi ce sont les attributs de dialogue de ce domaine qui sont alors pris en compte. (La figure VII.8 qui suit illustre l'interprétation dans une hiérarchie de domaines de l'attribut de dialogue définissant l'écho du dispositif d'interaction).



- figure VII.8 : Interprétation des attributs de dialogue dans une hiérarchie de domaines -

- elle permet l'**héritage** des attributs de dialogue, autorisant ainsi une définition simple et puissante d'objets d'interaction complexes. Ainsi, si le domaine sur lequel se situe le dispositif d'interaction ne possède pas certains attributs de dialogue, on utilise alors les attributs du (des) domaine(s) de niveau(x) supérieur(s) (fig VII.9).



- figure VII.9 : Héritage des attributs de dialogue dans une hiérarchie de domaines -

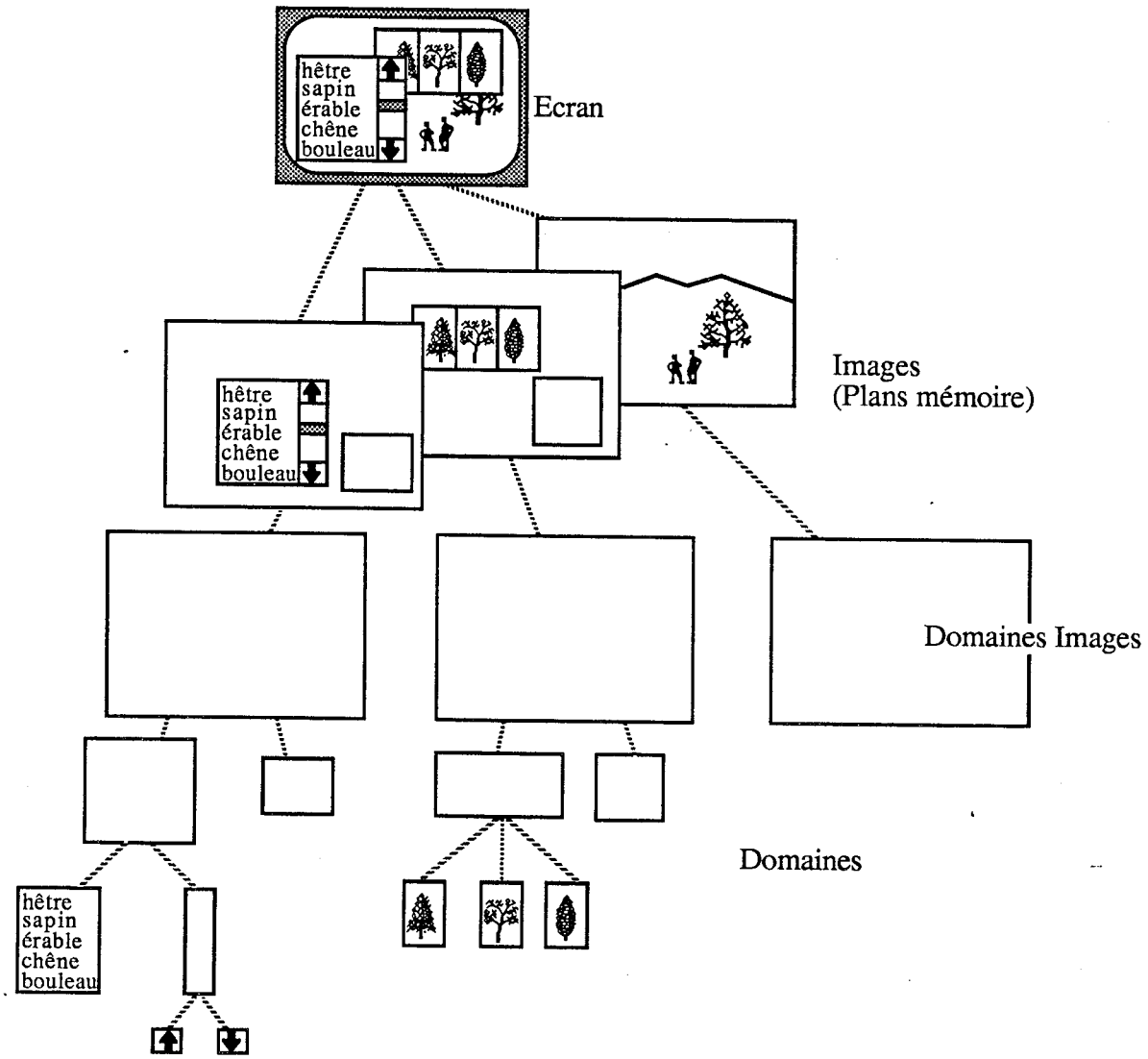
- elle permet la **modification** simple d'objets complexes. Il est ainsi possible de remplacer une structure hiérarchique par une autre, il suffit de mettre à jour les liens nécessaires. Par ailleurs les domaines étant définis relativement à un domaine père, le déplacement d'un domaine se répercute automatiquement et immédiatement sur tous les domaines hiérarchiquement dépendants.

3.3 Les domaines par défaut

Du fait de la définition hiérarchique des domaines (tout domaine doit être défini relativement à un domaine père englobant), des domaines par défaut racines des structures arborescentes des domaines sont prédéfinis.

Ainsi à chaque image "logique" (au sens GETRIS) utilisée par l'application est associé un domaine par défaut dont la zône utile correspond à l'image dans son entier.

Ces domaines par défaut sont le lien entre la notion d'image et la notion de domaines : les domaines décrivent la **structure logique des images** (fig. VII.10). Lorsqu'un domaine ou une entité domaine est utilisé il est ainsi automatiquement relié à une image, étant rattaché soit directement soit indirectement à l'un des domaines par défaut.



- figure VII.10 : Les domaines par défaut et les images -

Les domaines par défaut sont définis lors de la déclaration des images par l'application. Ils sont respectivement référencés par **image0,image1**... et possèdent les attributs par défaut suivants :

- zône utile : (**xg,yg**) = (0,0) et (**xd,yd**) = coin supérieur droit de l'image.
- repère : (**xc,yc**) = (0,0) (coin inférieur gauche).

attributs pour la présentation visuelle :

- pas de cadre (**refcadre** = null),
- pas de titre (**titre** = chaîne vide),
- pas de procédure d'affichage (**affich** = nop),
- pas d'objet graphique associé (**refscènes** = null).

attributs pour la visualisation d'objets graphiques :

- Gv : identité, projection parallèle selon Oz,
- E : lumière blanche à l'infini en z.

attributs pour le dialogue :

- **echoh** = **echob** = curseur.
- aucune action associée aux événements de dialogue (**clic** = nop).

4 MANIPULATION DES OBJETS D'INTERFACE

4.1 Les différents niveaux de primitives

Deux ensembles de primitives sont nécessaires pour la manipulation des objets de dialogue :

- des primitives permettant la description et la construction d'objets de dialogue évolués à partir du mécanisme de domaine (entités domaines).

- des primitives permettant l'utilisation de ces objets de dialogue à partir des applications pour la construction de leur interface utilisateur.

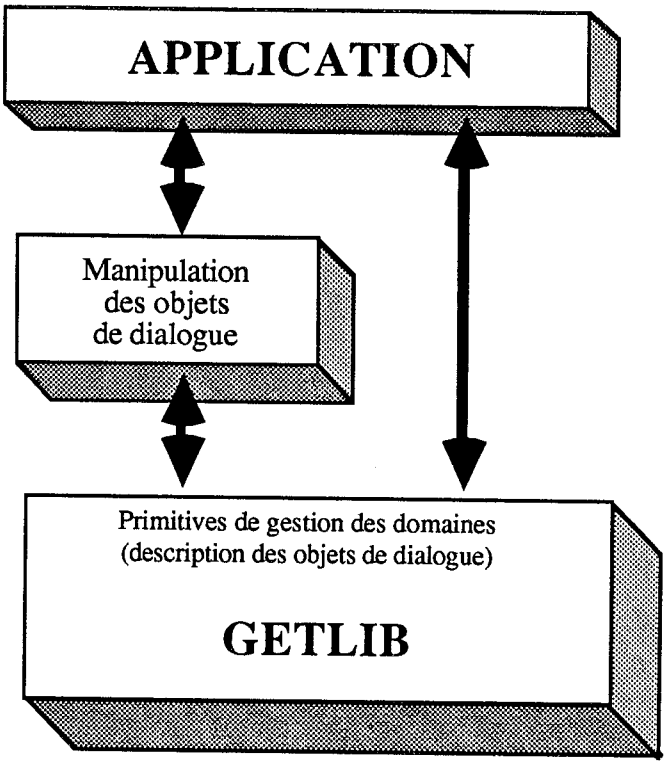
Dans la philosophie qui guide notre travail, il est clair que ces deux groupes de primitives ne s'adressent pas aux mêmes types d'utilisateurs :

- les premières, essentiellement des primitives de définition et de structuration hiérarchique des domaines, ainsi que des primitives d'attribution, concernent des "développeurs", utilisateurs avertis connaissant suffisament le système graphique et les représentations internes qu'il emploie. Elles leur permettent de définir de nouveaux types d'objets de dialogue (éventuellement à partir d'objets déjà définis) en construisant des primitives de niveau supérieur destinées aux programmeurs d'application.

- les secondes primitives permettent aux applications de manipuler aisément les objets de dialogue qu'elles traitent comme un tout (une seule entité dont il n'est pas nécessaire de connaître la structure interne). Elles libèrent au maximum l'application de la gestion des objets de dialogue qu'elle utilise pour son interface, en prenant entièrement en charge leur création (et éventuellemment leur modification), leur visualisation (affichage et effacement) et leur activation lors de séances d'interaction .

Nous pensons que la distinction de ces deux niveaux de primitives (fig VII.11) favorise la séparation entre la définition des objets d'interface et leur utilisation à l'intérieur des applications. Elle encourage la conception de techniques de dialogue puissantes pouvant ensuite être facilement réutilisées de manière cohérente au travers de différentes applications.

Cependant les deux niveaux de primitives sont accessibles aux utilisateurs du système graphique leur permettant de choisir en fonction de leur besoins et de leurs compétences. Ainsi les programmeurs d'application, simultanément à l'utilisation d'objets de dialogue évolués prédéfinis, peuvent concevoir et construire si nécessaire leurs propres objets de dialogue.



- figure VII.11 : les niveaux de primitives pour les objets de dialogue -

4.2 Les primitives destinées aux développeurs

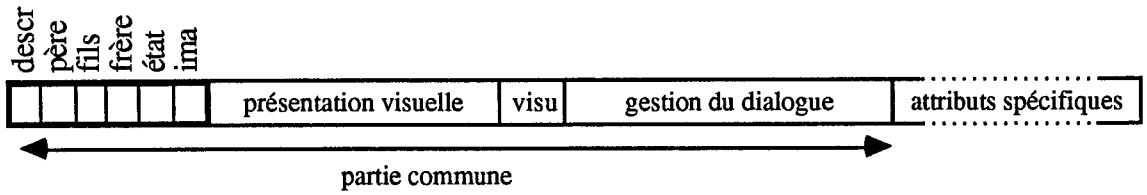
Dans les paragraphes qui suivent nous présentons les primitives destinées à la construction d'objets d'interface à l'aide de domaines. Il s'agit, comme nous l'avons déjà évoqué, essentiellement de la définition et structuration hiérarchique de domaines, ainsi que de la définition des attributs associés.

S'adressant à des développeurs avertis, ces primitives aggissent directement sur les structures de données internes du système graphique. C'est à cet effet que nous présentons dans ce qui suit la partie des descripteurs de domaine destinée à cette gestion. Nous étudierons ensuite, l'ensemble des primitives utilisées pour la construction et la structuration d'entités domaines. Suivront les diverses primitives de consultation et d'attribution qui permettent de modifier et d'accéder aux différents attributs communs aux domaines que nous avons présentés en détail au début de ce chapitre.

Pour la visualisation et le dialogue, les primitives utilisées par les développeurs sont les mêmes que celles destinées aux programmeurs d'applications. Nous les présenterons donc, au cours des paragraphes consacrés à ces dernières.

4.2.1 Attributs pour la gestion interne et la structuration des domaines

La figure VII.12, ci-contre présente la partie des descripteurs de domaine destinée à la gestion interne de ceux-ci.

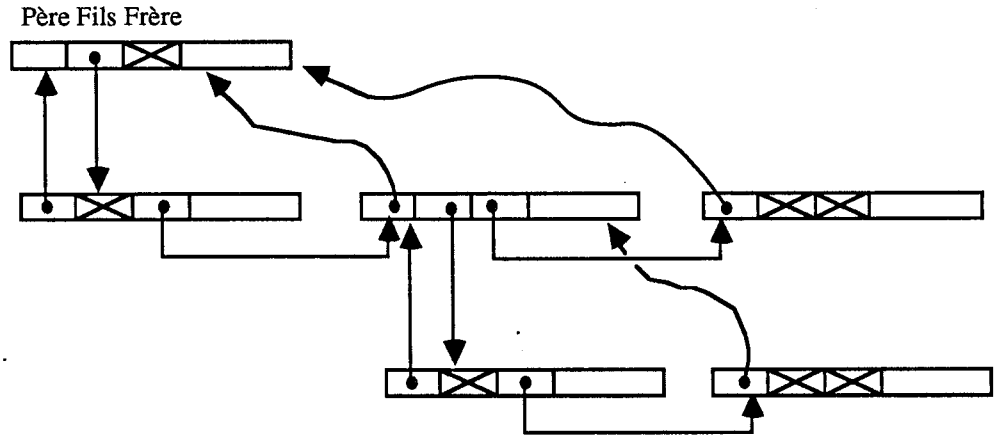


- figure VII.12 : Attributs pour la gestion interne et structuration des domaines -

Les attributs qu'elle regroupe ont les significations suivantes :

descr indique le type du descripteur (dans ce cas TYPVUE). Il est utilisé pour la gestion interne de la banque de données graphiques.

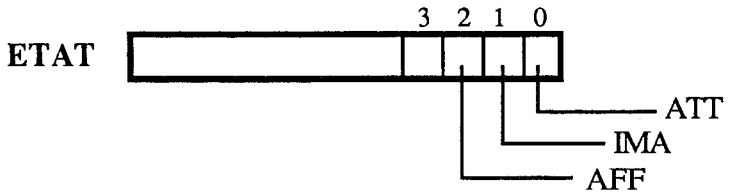
père, fils, frère sont trois pointeurs vers des descripteurs de domaine : ils définissent la structure arborescente des domaines (fig. VII.13).



- figure VII.13 : Liens hiérarchiques entre les descripteurs de domaines -

père contient la référence du domaine père. Ce pointeur est initialisé à la définition du domaine. Les deux autres pointeurs, ***fils*** et ***frère***, correspondent respectivement à la référence du premier des domaines fils éventuellement rattachés, et à la référence du domaine frère si le domaine est rattaché à son père. Ces derniers sont mis à jour lors des opérations de "rattachement" de domaines sur lesquelles nous reviendrons plus en détail au paragraphe suivant. Le double chaînage utilisé, permet d'assurer facilement la composition et l'héritage des attributs lors des séances de dialogue.

état : contient un certain nombre d'indicateurs utilisés pour les différentes opérations de gestion interne sur les domaines. Ceux-ci spécifient si le domaine et attaché ou non à une structure domaine père (ATT), si le domaine est rattaché ou non à une image (IMA), si le domaine est affiché ou non (AFF).



- figure VII.14 : Indicateurs d'état des domaines -

A un instant donné on dira qu'un domaine est :

- **libre**, s'il n'est rattaché à aucun autre (indicateur *ATT = 0*). Le domaine est alors la racine d'une sous arborescence de domaines,

- **actif**, si le domaine est un domaine image par défaut, ou s'il est rattaché à un domaine image (c'est à dire qu'en remontant les pères successifs du domaine on atteint un domaine image, domain dont l'indicateur *IMA = 1*),

- **attaché**, si le domaine est rattaché à un domaine inactif (le domaine racine de la structure à laquelle il appartient est libre).

Seuls les domaines actifs peuvent être impliqués dans les opérations de visualisation, et seuls les domaines actifs et affichés (indicateur *AFF = 1*)peuvent intervenir dans le dialogue interactif.

ima : si le domaine est actif, ce champ contient le numéro de l'image à laquelle il est rattaché. C'est sur cette image que les opérations de visualisation et de dialogue concernant le domaine seront effectivement réalisées.

4.2.2 Création et structuration arborescente des domaines

Dans ce paragraphe, nous présentons les primitives destinées à la création/suppression de domaines et à leur structuration sous forme arborescente. Il est à souligner que celles-ci ne provoquent aucun affichage ou modification de l'image, elles consistent en de simples allocations et mises à jour dans les banques de données du système graphique. Leur prise en compte sur l'image affichée n'est effectuée que suite à une demande explicite à l'aide des primitives de visualisation des domaines que nous étudierons ultérieurement (§ 4.3.2.1).

La création d'une nouvelle entité domaine s'effectue à l'aide de la fonction

Def_Domaine(ref_père,xg,yg,xd,yd,taille) ---> refdom

qui alloue dans la banque de données du système graphique l'espace nécessaire à son descripteur et renvoie la référence de celui-ci (cette référence est utilisée pour désigner le domaine dans les opérations ultérieures l'impliquant).

- *ref_père* est l'adresse du domaine père relativement auquel est défini le nouveau domaine,

- *xg,yg,xd,yd* définissent les coins inférieur gauche et supérieur droit de la zône utile du domaine (ces coordonnées sont exprimées dans le repère du domaine *ref_père*),

-*taille* défini le nombre d'octets occupés par le descripteur de l'entité domaine.

A la création d'un nouveau domaine, les attributs standards qui lui sont associés sont initialisés avec les valeurs par défaut :

- le repère *xc, yc* du domaine coincide avec le coin inférieur gauche de sa zône utile (*xg,yg*),
- un cadre par défaut (*CadreDom*) est associé au domaine,
- le domaine est marqué comme actif pour tous les état dialogue (*falgdial = tout = haut + bas*).
- tous les autres attributs du domaine sont initialisés avec des valeurs nulles.

Le domaine créé est un domaine **libre** : son descripteur n'est pas rattaché à celui de son domaine père. L'établissement de ce lien s'effectue à l'aide de la primitive

Attach_Domaine (refdom)

qui permet de construire une arborescence de domaines; elle attache le descripteur du domaine

référencé par *refdom* à celui de son domaine père (défini au moment de la création de *refdom*). Cette primitive est sans effet si le domaine *refdom* est déjà rattaché à son père. Sinon, les traitements effectués sont les suivants :

- le descripteur du domaine référencé par *refdom* est inséré en tête des fils de son père,

- un changement de repère est calculé sur le domaine *refdom* et tous les sous-domaines qui lui sont rattachés, afin d'exprimer les coordonnées définissant leur zône utile (*xg,yg,xd,yd*) et l'origine de leur repère (*xc,yc*) dans le repère du domaine racine de la structure arborescente ainsi constituée. Cette composition des coordonnées se déroule lors de l'attachement d'un domaine pour des raisons d'efficacité : ainsi lorsque un domaine est actif (c.a.d. rattaché à une image directement ou indirectement par les domaines de niveaux supérieurs) les coordonnées sont exprimées dans le repère de l'image et peuvent être utilisées directement lors d'opérations de visualisation ou de dialogue.

- si le domaine *refdom* est rattaché à un domaine actif, lui et les sous-domaines qui lui sont rattachés se voient activés (leur flag *ima* de leur champ *état* est positionné et ils héritent du numéro d'image du domaine père (champ *ima*)),

- le domaine ***refdom*** est marqué comme étant attaché (positionnement du flag *att* du champ *état*).

De manière symétrique la primitive

Detach_Domaine(refdom)

permet de retirer un domaine (référencée par *refdom*) d'une structure arborescente. La sous structure détachée reste **toujours** accessible via la référence de sa racine (*refdom*).

Lors d'un détachement de domaine, les opérations suivantes sont effectuées :

- remise à jour du chaînage des fils du père de *refdom*,

- le calcul du repère initial est fait, c'est à dire la transformation inverse de celle effectuée lors de l'attachement du domaine qui restitue des coordonnées *xg,yg,xd,yd,xc,yc* relatives est appliquée récursivement au domaine *refdom* et à tous les sous domaines qui lui sont rattachés,

- le domaine référencé par *refdom* devient libre (l'indicateur *att* du champ *état* est positionné à 0) et toute l'arborescence de domaines dont il est racine est désactivée (l'indicateur *ima* est positionné à 0).

La primitive

Sup_Domaine(refdom)

détruit la structure de domaines référencée par *refdom*. L'espace mémoire alloué au domaine *refdom* et aux sous-domaines qui lui sont attachés est entièrement libéré. Si *refdom* était attaché à un domaine père, il est préalablement détaché avant d'être détruit. Après l'application de cette primitive, la référence *refdom* n'est bien entendu plus valide.

4.2.3 Consultation et modification d'attributs de domaines

Les primitives que nous présentons dans ce paragraphe permettent la modification et la consultation des attributs standards associés aux domaines. La majeure partie d'entre elles n'effectuent aucun traitement particulier, elles accédent et/ou modifient directement le champ correspondant du descripteur de domaine. Cependant elles jouent un rôle d'encapsulation en évitant aux développeurs d'accéder directement à la structure des descripteurs de domaines.

4.2.3.1 Consultation et modification de la zône utile et du repère d'un domaine

La primitive

Int_Domaine (refdom, repère, &xg, &yg, &xd, &yd)

permet de consulter les coordonnées définissant la zône utile du domaine *refdom*, c'est à dire correspondant à l'intérieur du domaine sans tenir compte de son cadre. Les coordonnées rendues (*xg,yg* coin inférieur gauche de la zône utile et *xd,yd* coin supérieur droit) sont exprimées soit :

- dans le propre repère du domaine (*repère = Rvue*),
- dans le repère de son domaine père (*repère = Rloc*),
- dans le repère de l'image à laquelle le domaine est raccroché (*repère = Rima*), cette option n'est possible que si le domaine *refdom* est actif.

De manière analogue, la primitive

Ext_Domaine (refdom, repère, &xg, &yg, &xd, &yd)

permet de consulter les coordonnées de la zône recouverte par le domaine *refdom* dans son entier, c'est à dire en considérant son cadre si il existe. Le paramètre *repère* a la même signification que dans la primitive précédente.

Il est également possible de rédéfinir la zône utile d'un domaine par l'intermédiaire de la primitive

Zône_Domaine (refdom, repère, xg, yg, xd, yd)

refdom est la référence du domaine concerné. Comme pour la consultation, *repère* définit le référentiel dans lequel sont exprimées les coordonnées *xg,yg* et *xd,yd* des coins inférieur gauche et supérieur droit de la nouvelle zône utile du domaine (dans le repère de l'image, du domaine père, ou du domaine *refdom* lui-même).

Lors de la modification de la zône utile d'un domaine, le repère associé à ce dernier est automatiquement mis à jour afin que sa position relative par rapport au coin inférieur gauche de la zône utile demeure inchangée. La zône utile et le repère de tous les sous-domaines attachés au domaine *refdom* sont également modifés de manière analogue.

La position de l'origine du repère associé à un domaine, peut quand à elle, être explicitement modifiée à l'aide de la primitive

Orig_Domaine(refdom, xc, yc)

refdom est la référence du domaine et *xc,yc* définissent la nouvelle origine du repère du domaine relativement au coin inférieur gauche de sa zône utile. De la même manière que pour la primitive *Zone_Domaine*, les sous-domaine de *refdom* sont translatés en conséquence afin de conserver les mêmes positions relatives.

Les deux primitives ci-dessus ne provoquent aucune modification de l'image, la nouvelle zône utile ou le nouveau repère ne seront pris en compte que pour les affichages ultérieurs concernant le domaine.

4.2.3.2 Consultation et modification des attributs pour la visualisation des domaines

Pour la gestion des cadres, permettant de matérailiser la zône utile d'un domaine, le

programmeur d'application dispose du jeu de primitives suivantes :

Valid_Cadre(refdom,mode)

qui pour le domaine de référence *refdom* positionne les indicateurs *flagcadre* indiquant les portions du cadre à afficher. Celles-ci sont définies à l'aide du paramètre *mode* pour lequel des constantes ont été prédéfinies (***tout*** toutes les partie du cadre, ***zofond*** le fond uniquement, ***neant*** pas d'affichage du cadre).

L'affectation d'un cadre à un domaine s'effectue par l'intermédiaire de la primitive

Att_Cadre(refdom,refcadre)

où *refcadre* est la référence d'un cadre défini auparavant à l'aide de la primitive *Def_Cadre* (voir § 2.2.1.1). De manière symétrique, la primitive

Cons_Cadre(refdom,refcadre)

permet de récupérer la référence du cadre associé à un domaine.

Concernant les autres attributs pour la visualisation des domaines, les primitives

Def_Guide_Domaine(refdom,fonction)
Cons_Guide_Domaine(refdom,fonction)

permettent, respectivement, d'affecter et de consulter la fonction d'affichage (champ *(*affich)()* du descripteur) d'un domaine. *fonction* est l'adresse de la fonction d'affichage. De manière analogue

Def_Obj_Domaine(refdom,refobj)
Cons_Obj_Domaine(refdom,refobj)

sont destinées à l'attribution et la consultation de l'objet graphique mémorisé éventuellement associé au domaine (attribut *refscène* du descripteur). *refobj* est la référence de l'objet à affecter ou de l'objet consulté.

4.2.3.3 Consultation et modification des attributs pour le dialogue

La primitive

Valid_Dial(refdom,Cas)

positionne les indicateurs du champ *flagdial* afin de valider ou invalider partiellement le dialogue sur le domaine. Rappelons que *flagdial* définit les états du dispositif d'interaction pour lesquels le domaine *refdom* est actif (et donc ses attributs considérés) lors d'une séance de dialogue. Les états pour lesquels le dialogue est validé sont définis par le paramètre *cas* pour lequel les constantes prédéfinies ***haut,bas,tout*** et ***néant*** (invalidation totale du dialogue) peuvent être utilisées en étant éventuellement combinées.

La définition et consultation des autres attributs de dialogue associés au descripteur de domaine s'effectue à l'aide des primitives

Def_Guide_Domaine(refdom,fonction)
Cons_Guide_Domaine(refdom,fonction)
Def_Interieur_Domaine(refdom,fonction)
Cons_Interieur_Domaine(refdom,fonction)
Def_Clic_Domaine(refdom, fonction)
Cons_Clic_Domaine(refdom, fonction)

qui concernent respectivement les fonctions *intérieur* permettant de déterminer l'appartenance

d'un point au domaine, *guide* permettant de contraindre l'echo et *clic* qui est invoquée lors des changements d'état du dispositif sur le domaine. Le paramètre *fonction* est l'adresse de la fonction correspondante à affecter au domaine ou à consulter.

Les attributs définissant l'écho lorsque le dispositif de dialogue se situe sur le domaine sont quand à eux manipulés à l'aide des primtives

Def_Echo_Domaine(refdom, cas, fecho, motif, expcoul)
Cons_Echo_Domaine(refdom, cas, fecho, motif, expcoul)

Le paramètre *cas* indique s'il s'agit des attributs pour l'écho du dialogue en position haute ou basse (défini par les constantes ***bas*** et ***haut***), *fecho* est l'adresse de la fonction d'écho, *motif* et *expcoul* le motif de curseur et le numéro de l'expression de couleur qui lui sont éventuellement associés.

4.3 Les primitives destinées aux programmeurs d'application

L'ensemble des primitives dédiées à la manipulation des objets de dialogue à l'intérieur des programmes d'application peut se scinder en deux groupes :

- les primitives **dépendantes** du type de l'objet de dialogue auquel elles se rapportent. Elles concernent la création, la modification et la consultation des objets d'interface, opérations pour lesquelles un ensemble de primitives spécifiques à chaque type d'objet doit bien entendu être défini afin de pouvoir paramétrer les objets selon leur nature.

- les primitives proposées de façon **standard** par le logiciel graphique et indépendantes du type des objets de dialogue qu'elles manipulent. En effet, du fait de l'unicité de la représentation interne des objets d'interface (arborescence de domaines) un seul jeu de primitives est nécessaire pour leur visualisation et leur utilisation lors de séances de dialogue.

4.3.1 Primitives spécifiques à chaque type d'objet de dialogue

Si à chaque type d'objet de dialogue sont associées ses propres primitives de création, modification et consultation, celles-ci partagent néanmoins dans une très large mesure les mêmes fonctionnalités. Aussi pouvons nous, dans ce qui suit, donner la forme générique de l'ensemble des primitives de cette nature associées à chaque type d'objet de dialogue (ces primitives diffèrent par le nombre et la nature de leurs paramètres). Cette liste fournit ainsi la trame de ce qui est doit être développé (à partir des primitives internes de manipulation de domaines exposées précédemment) lors de la définition d'un nouveau type d'objet de dialogue.

4.3.1.1 Définition d'un objet de dialogue

La primitive

refdom <-- Def_*X*(refpère,....)

permet la création d'un exemplaire d'objet de dialogue de type X. Pour respecter la hiérarchie images/domaines, l'objet de dialogue ainsi créé est défini relativement à un domaine père englobant désigné par *refpère*. Cette primitive retourne la référence (*refdom*) de l'objet créé. Celle-ci est utilisée par la suite pour l'identification de l'objet de dialogue dans les primitives de manipulation l'impliquant.

Un certain nombre de paramètres, propres à chaque type d'objet de dialogue permettent de particulariser l'objet lors de sa création (taille, aspect, contenu (par exemple les différents textes d'un menu)....).

Il est à souligner que la primitive de création (ou définition) d'un objet de dialogue, consiste

uniquement en sa déclaration. Elle a pour seul effet d'allouer les structures de données nécessaires à la représentation de l'objet de dialogue et n'a, en particulier, aucune répercussion sur l'image affichée. La visualisation de l'objet de dialogue doit être demandée explicitement à l'aide de la primitive appropriée.

4.3.1.2 Modification d'un objet de dialogue

Le programme d'application a la possibilité de modifier certaines des caractéristiques (attributs) des objets de dialogue qu'il a définis. Pour cela, plusieurs primitives lui sont proposées, chacune concernant un ensemble d'attributs plus ou moins liés, susceptibles d'être modifiés simultanément et qu'il est donc plus aisé de manipuler de façon groupée et indépendante des autres. Le nombre de ces primitives et leur paramètres (reprennant dans une large mesure ceux de la primitive de création), dépendent bien entendu du type de l'objet de dialogue, cependant on retrouvera en règle générale les primitives suivantes :

Maj_*X*(refdom,.....)

où *refdom* est la référence qui désigne l'objet de dialogue (de type X) à modifier. Les attributs que cette primitive permet de modifier sont ceux qui dépendent directement du type de l'objet de dialogue et qui contiennent l'information qu'il est chargé de transmettre, par exemple le texte des différents choix pour un menu... L'application de cette primitive entraîne **une mise à jour automatique** de l'image si l'objet de dialogue est affiché.

La seconde primitive

Asp_*X*(refdom,.....)

concerne quand à elle la modification des attributs d'aspect qui permettent de particulariser l'objet de dialogue (couleur, polices de caractères....). A l'inverse de la primitive précédente, la modification n'intervient que sur les structures de données de l'objet de dialogue et ne provoque aucune mise à jour de l'image. Celle-ci doit être demandée explicitement à l'aide de la primitive de visualisation d'objets de dialogue.

Une dernière primitive permet la modification des paramètres décrivant le comportement de l'objet de dialogue lors des séances d'interaction :

Def_Exec_*X*(refdom,&fct_trt,...)

Elle permet en particulier l'association à l'objet de dialogue référencé par *refdom* d'une fonction de traitement pour les événements de dialogue (*&fct_trt* l'adresse de cette fonction).

4.3.1.3 Consultation

Le programme d'application peut récupérer certains paramètres caractéristiques d'un objet de dialogue à l'aide de la primitive :

Cons_*X*(refdom,...)

refdom est la référence de l'objet de dialogue de type X concerné. Parmi les autres paramètres qui recoivent l'information consultée certain peuvent correspondre :

-soit à des caractéristiques de l'objet utilisées lors de sa création (le programme d'application peut ainsi les récupérer si il ne les a pas mémorisées par ailleurs),

- soit à des informations qui ont pu être associées à l'objet de dialogue lors du déroulement d'une séance d'interaction au cours de laquelle il a été impliqué. Les données ainsi consultées permettent à l'application de prendre compte, si nécessaire, du résultat d'une séance de dialogue.

4.3.2 Primitives standards de manipulation d'objets de dialogue

Les primitives qui suivent traitent les objets de dialogue de manière **banalisée** : elles sont totalement indépendantes du type de l'objet d'interface.

4.3.2.1 Visualisation

La primitive

Aff_ObjDial (refdom,Mode)

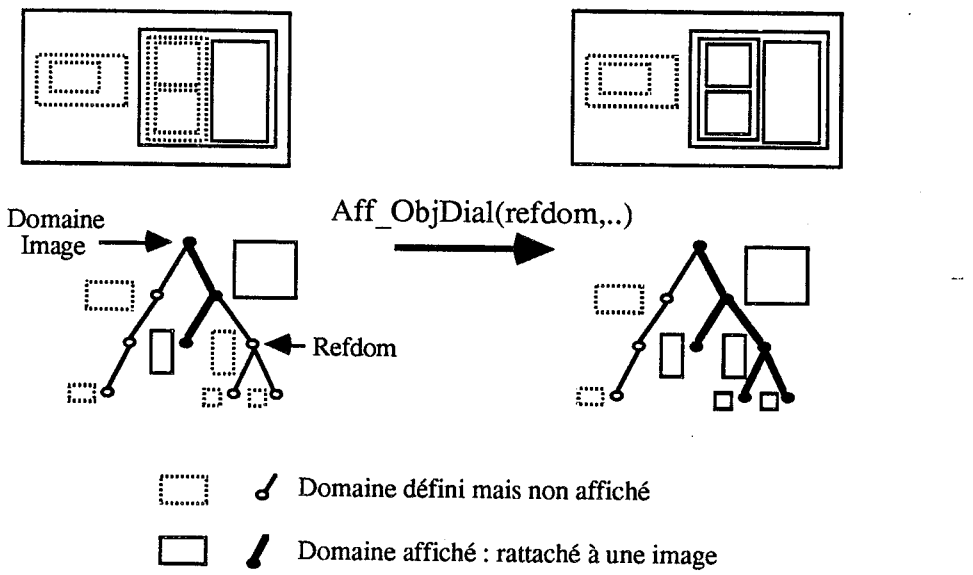
affiche un objet de dialogue quelconque référencé par *refdom.*

Le paramètre *mode* indique les opérations qui doivent être effectuées lors de la visualisation de l'objet en combinant par addition les constantes :

- *cadre* provoque l'affichage du cadre du domaine,
- *presse* indique l'utilisation du presse papier lors de l'affichage du cadre. C'est à dire la sauvegarde (si la mémoire disponible le permet) de la région de l'image correspondant à la zône utile et au dessin du cadre du domaine .
- *entité* provoque l'invocation automatique de la fonction d'affichage *affich()* , attribut du domaine racine de la hiérarchie des domaines définissant l'objet. Comme nous l'avons vu au § 2.2.1, c'est cette fonction qui assure la visualisation des objets graphiques éventuellement associés au domaine ainsi que la visualisation des domaines de niveaux inférieurs.

L'affichage d'un objet de dialogue s'effectue sur l'image à laquelle il a été implicitement associé lors de sa création (voir § 3.3).

Cependant, la visualisation n'a de sens et n'est possible que si l'objet (le domaine) père par rapport auquel l'objet de dialogue a été défini est lui même déjà affiché. Dans le cas contraire la primitive ***Aff_ObjDial*** est sans effet. (Remarque : les domaines images par défaut ***Image0***, ***Image1***.... racines des structures arborescentes des domaines sont toujours considérés comme affichés).



- figure VII.15 : rattachement des objets de dialogue (domaines) à une image -

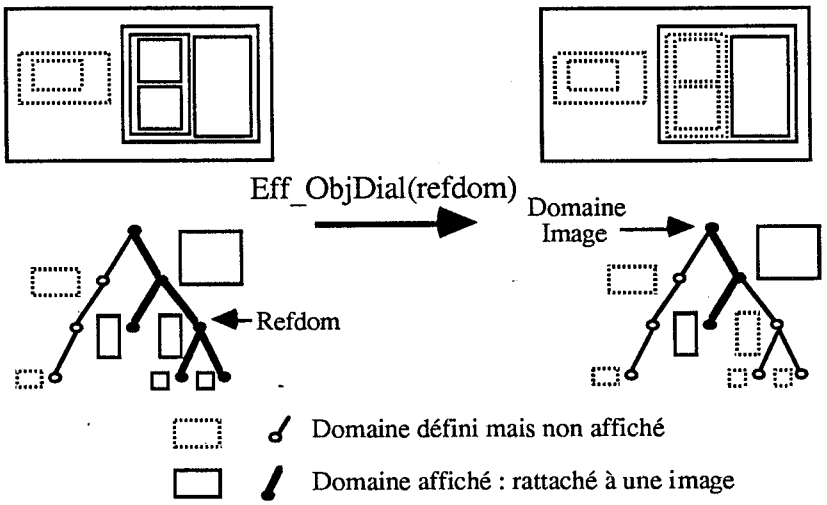
En fait, la définition des objets de dialogue crée implicitement une hiérarchie image/domaine sur laquelle on peut définir une seconde hiérarchie qui décrit la structure de l'image affichée (fig. VII.15). Le rattachement effectif d'un objet de dialogue (c.a.d. du domaine racine de la

hiérarchie des domaines le définissant) à une image a lieu lors de son affichage et n'est possible que si son père est lui même déjà rattaché à une image (ou est une image domaine par défaut).

De manière symétrique à la visualisation, la primitive :

Eff_ObjDial (refdom)

permet d'effacer l'objet de dialogue référencé par *refdom* . L' objet de dialogue effacé n'est plus associé à une image, son domaine racine est automatiquement détaché (fig. VII.16). Si le mode *Presse* a été utilisé lors de son affichage , l'effacement de l'objet de dialogue restitue la zone du presse papier. Dans le cas contraire cette opération n'a aucun effet direct sur l'image visible.



- figure VII.16 : effacement d'un objet de dialogue -

L'effacement d'un objet de dialogue n'a pas d'effet sur sa mémorisation : il est conservé dans la banque de données du logiciel graphique et peut toujours être accédé par sa référence et être réutilisé par la suite.

4.3.2.2 Le dialogue

L'utilisation d'un objet de dialogue pour une séance d'interaction s'effectue à l'aide de la primitive :

Séance(refdom)

Une séance de dialogue ne peut avoir lieu que si le domaine est effectivement affiché, dans le cas contraire cette primitive est sans effet. Lorsqu'un objet de dialogue est ainsi rendu actif pour une séance d'interaction, les attributs de la hériarchie de domaines définissant l'objet sont automatiquement pris en compte (avec une éventuelle composition) : ce sont les fonctions d'écho et de traitement des événements de dialogue associées à ces domaines qui sont exécutées lorsque le dispositif d'interaction est déplacé sur la (les) zône(s) correspondante(s).

La manière dont la séance d'interaction se termine dépend bien entendu du type de l'objet de dialogue. Elle correspond à une ou plusieurs séquences d'interaction reconnues par les primitives de gestion du dialogue (attribut des domaines) de l'objet et dont le traitement provoque l'arrêt de la boucle d'interaction. (A cet effet une primitive **FinSéance()** est proposée aux développeurs et doit donc être insérée dans l'une des fonctions de traitement associées à l'objet de dialogue).

De plus amples détails sur le fonctionnement des primitives ***Seance()*** et ***FinSéance()*** sont donnés au paragraphe 6.3, où nous présentons la manière dont le contrôle dynamique du dialogue lors de l'execution d'un programme est effectué. La façon dont ces deux primitives sont réalisées sera alors explicitée.

5 EXEMPLES D'OBJETS D'INTERFACE

Dans ce qui suit, nous présentons deux exemples d'objets de dialogue prédéfinis et proposés de manière standard aux programmeurs d'applications :

- les vues qui permettent la visualisation d'objets graphiques,
- les menus avec défilement vertical, qui permettent de selectionner une entité parmi un ensemble.

Ces deux exemples nous permettront de montrer la manière dont des objets d'interface peuvent être définis à partir de domaines et des attributs associés (§ 2), et d'illustrer également les primitives qui permettent leur manipulation (que nous avons présentées sous forme générique au § 4.2).

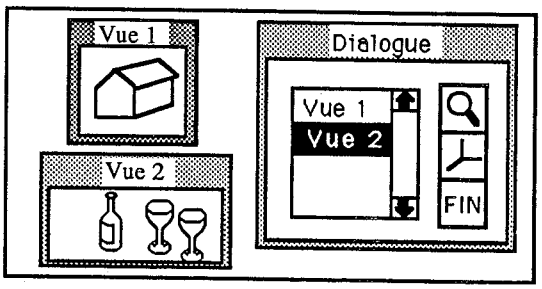
5.1 Les vues

5.1.1 Fonctionnalités des Vues

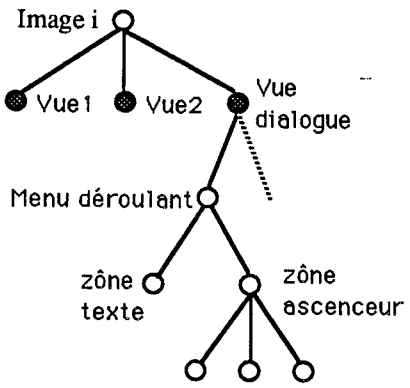
Les vues sont des objets d'interface uniquement destinés à la visualisation d'autres objets (objets graphiques ou objets de dialogue). Elles ne possèdent pas d'attributs (actions) pour le traitement du dialogue qui s'il a lieu sur la zône utile d'une vue doit être entièrement géré par l'application (la primitive Séance présentée précédemment ne peut être utilisée).

Les vues représentent les objets d'interface de niveau le plus élevé dans la hiérarchie des domaines. Elles constituent le premier niveau de structuration d'une image et ne peuvent être imbriquées. Les vues sont à rapprocher de la notion de fenêtre popularisée par les désormais classiques logiciels possédant une interface multifenêtres.

La figure VII.17 ci dessous illustre l'utilisation de trois vues définies sur une image. Les deux premières vues sont destinées à la visualisation d'objets graphiques, la troisième étant quand à elle dédiée à la gestion de l'interaction avec l'utilisateur. Dans cette dernière vue sont définis d'autres objets d'interface pour la prise en charge du dialogue dont par exemple un menu sur lequel nous reviendrons par la suite.



Image



Structure des domaines

- figure VII.17 : utilisation des vues pour la structuration des images -

Par souci de cohérence, les domaines par défaut (*image0, image1...*) définis lors de la déclaration des images logiques (voir § 3.3) sont en fait des objets d'interface de type vue (sans titre ni dessin de cadre). En effet la visualistion d'objets graphiques ou l'utilisation d'objets d'interface peut s'effectuer directement par rapport à ces domaines sans que l'utilisateur ait à définir ses propres vues...

5.1.2 Les primitives de manipulation des vues

Dans ce qui suit nous présentons brièvement les primitives proposées à l'application pour la gestion des vues.

5.1.2.1 Définition-création d'une vue

La primitive

Def_Vue(image,titre,xg,yg,xd,yd) --> référence

défini une vue et renvoie sa référence. La vue créée est rattachée à l'image de la vue par défaut référencée par *image*. Les autres paramètres de *Def_vue* indiquent son titre et le coin inférieur gauche et supérieur droit de sa zône utile (en coordonnées image).

Au domaine de la vue ainsi créée sont associés les attributs par défaut suivants :

- cadre standard avec *flagcadre* positionné à *tout*,
- *flagdial* positionné à *tout* (la vue est active pour tous les états du dialogue), tous les autres attributs de dialogue sont nuls.

5.1.2.2 Modification d'une vue

Les primitives de modification des vues sont :

Zone_Vue(refvue,xg,yg,xd,yd)

qui permet de modifier l'emplacement sur l'image de la vue *refvue* . Les coordonnées de la nouvelle zône utile de la vue sont exprimées dans le repère de l'image.

A l'heure actuelle cette primitive ne provoque aucune modification de l'image qui doit si nécessaire être explicitement régénérée par l'application. Cependant il est prévu d'étendre le logiciel afin qu'il effectue automatiquement la régénération de l'image, en traitant en particulier les problèmes de recouvrement entre les vues.

Orig_Vue(refvue,xc,yc)

provoque la modification de l'origine du repère associé à la vue *refvue* . Les coordonnées de la nouvelle origine du repère sont exprimées relativement au coin inférieur gauche de la zône utile de la vue. Cette modification n'affecte pas les objets déjà affichés dans la vue, elle ne concerne que les affichages ultérieurs sur la vue.

Titre_Vue(refvue,titre)

permet de modifier le titre de la vue *refvue* . Le cadre de la vue doit être réaffiché pour que cette modification soit prise en compte sur l'image.

5.1.2.3 Consultation des attributs d'une vue

Les primitives de consultation sont :

Int_Vue(refvue,xg,yg,xd,yd)
Ext_Vue(refvue,xg,yg,xd,yd)

qui permettent respectivement de récupérer les coordonnées image des coins inférieur gauche et supérieur droit de la zône utile et du cadre d'une vue référencée par *refvue* .

5.1.2.4 Utilisation des vues : la notion de contexte

Les vues servent de support aux échanges entre l'application et l'utilisateur. Pour chaque type d'échange (visualisation ou écriture, dialogue) il existe une **vue courante** permettant de définir le contexte dans lequel les actions relatives à ces échanges vont être effectuées. Ces contextes peuvent être définis et imbriqués **dynamiquement** lors de l'execution d'un programme à l'aide des primitives :

DebEcr(refvue)	**DebDial(refvue)**
...	...
FinEcr()	**FinDial()**
pour la visualisation (l'écriture)	pour le dialogue

Il existe toujours un contexte par défaut créé à l'initialisation du système (primitive *InitGetris*) qui pour la visualisation et le dialogue est la vue par défaut *Image0*.

Tous les objets affichés dans le contexte d'une vue sont visualisés en fonction d'attributs par défaut sauf si d'autres attributs sont définis explicitement à l'aide des primitives appropriées (voir chapitre VIII). Les attributs par défaut associés à toutes les vues sont extraits d'un fichier "ressource" que l'utilisateur peut éventuellement modifier.

Pour le dialogue un certain nombre de primitives de bas niveau permettent à l'application de gérer elle même l'interaction sur une vue si elle n'utilise pas d'objets d'interface évolués.

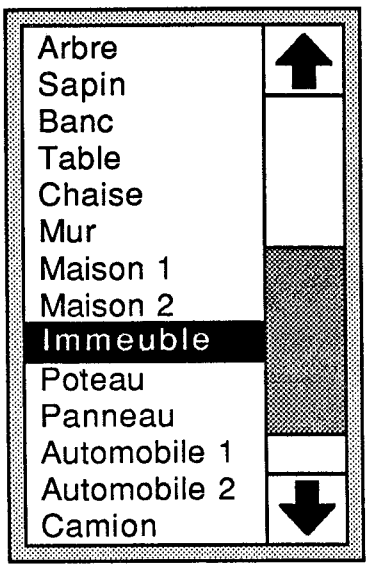
Si pour l'utilisateur (programmeur d'application) la notion de contexte est directement liée à la notion de vue (ce qui d'ailleurs est souhaitable), celle-ci n'en demeure pas moins beaucoup plus générale. En effet, les contextes peuvent être non seulement associés aux vues, mais également aux domaines définissant des objets d'interface quelconques et ceci quel que soit leur niveau dans la structure arborescente des domaines. Aussi ne détaillons-nous pas plus la définition et l'utilisation des contextes sur laquelle nous revenons par la suite (§ 6).

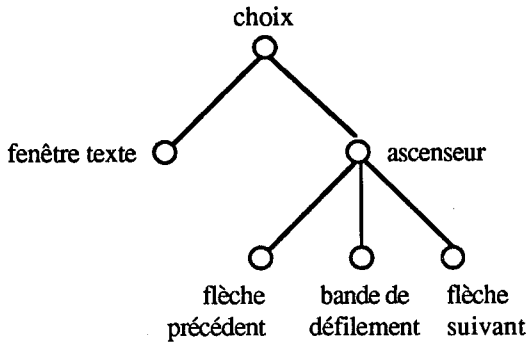
5.2 Les Menus ou Choix

5.2.1 Fonctionnalités des choix (menus avec défilement vertical)

Dans l'interface d'applications interactives, il est très souvent nécessaire de définir des menus permettant de sélectionner une entité parmi un ensemble possible. A cet effet, un second type d'objet de dialogue prédéfini , le ***choix***, est proposé aux programmeurs. Ceci rend immédiat l'emploi de menus avec défilement vertical (il suffit de spécifier leur position et contenu), qui grace au mécanisme de domaines sont entièrement pris en charge par le logiciel graphique.

Un objet de dialogue de type *choix* est défini par une hiérarchie de domaines à trois niveaux (fig. VII.18).





- Hiérarchie des domaines -

- figure VII.18 : structure de domaine d'un objet de type *choix* -

Le domaine du menu dans son ensemble (domaine racine) est divisé en deux sous-domaines, correspondants respectivement :

- à une "fenêtre" d'affichage pour le texte identifiant les différentes entités selectables à l'aide du menu,

- à un "ascenseur" permettant de déplacer la fenêtre d'affichage du texte sur l'ensemble des entités associées au menu. (Ce domaine ascenseur n'existe en fait que si l'ensemble des entités associées au menu est trop important pour pouvoir être visualisé dans son entier dans la fenêtre d'affichage du texte).

L'ascenseur est lui-même subdivisé en trois domaines :

- aux extrémités de l'ascenseur, deux régions permettent lorsque l'on "clique" dessus, de déplacer la fenêtre texte du menu d'une ligne vers le haut ou d'une ligne vers le bas. La flèche dessinée dans ces régions symbolise la direction de ces déplacements.

- la région centrale de l'ascenseur (bande de défilement) représente l'ensemble des entités associées au menu, le rectangle gris indiquant la position actuelle de la fenêtre texte sur cet ensemble. Il est également possible de faire défiler le texte du menu en "cliquant" sur cette région : la fenêtre est positionnée directement à l'endroit indiqué.

5.2.2 Les primitives de gestion des choix

Les primitives spécifiques pour la manipulation des objets d'interface de type *choix* (menu avec défilement vertical) sont les suivantes :

5.2.2.1 Définition/création d'un choix

refchoix <--- Def_Choix(refvue,refcadre,xg,yg,xd,yd,descr)

permet de créer un nouveau menu sur la vue *refvue* (c'est sur cette vue qu'il sera affiché). Les autres paramètres de cette primitive sont :

- *xg*, *yg* et *xd*, *yd* qui définissent les coins inférieur gauche et supérieur droit de la zône occupée par le menu sur l'image (zône utile du domaine menu). Ces coordonnées sont exprimées dans le repère de la vue *refvue*.

- *refcadre* qui est la référence de l'attribut cadre attaché au menu pour éventuellement matérialiser cette zône.

- *descr* qui est une chaîne de caractères contenant la liste des noms des différentes entités devant apparaître au menu. Le premier caractère de ce descripteur joue le rôle de séparateur entre les noms. Par exemple pour un menu contenant cinq noms de fichiers on utilisera la chaîne :

"/Fichier1/Fichier2 /Fichier3 /Fichier4 /Fichier5"

Il est également possible de spécifier la validité de chacune des entités à l'intérieur du descripteur (par défaut elles sont valides) en faisant précéder leur nom par le caractère de contrôle ^ suivi de :

- **i** pour invalide
- **s** pour sélecté (possible sur une seule entité).

5.2.2.2 Modification d'un choix

Les primitives de modification servant à changer certains des attributs propres aux objets de dialogue de type *choix*, sont les suivantes :

Asp_Choix(refchoix,police,expaval,expinv,expsel)

permet de spécifier l'aspect des différentes entités du menu déroulant référencé par *refchoix* en indiquant la *police* de caractères et des expressions de couleur pour les entités valides (*expval*), invalides (*expinv*) ou l'entité sélectée (*expsel*). Ces modifications ne provoquent aucune mise à jour de l'image qui doit être demandée explicitement en réaffichant le menu déroulant si nécessaire.

DefExec_Choix(refchoix,adr_trt)

permet d'associer une fonction de traitement au choix (d'adresse *adr_trt*) qui sera exécutée chaque fois qu'une entité du menu sera sélectée (par exemple l'ouverture d'un fichier si les entités sont des noms de fichier).

Maj_Choix(refchoix,nument,etat,texte)

autorise la modification du nom de l'une des entités apparaissant dans le choix référencé par *refchoix* . L'entité de rang *nument* voit son nom remplacé par la chaîne contenue dans *texte* et la validité indiquée par *etat* lui est associée. Si le choix est affiché, la mise à jour automatique de l'image est effectuée.

Modif_Choix(refchoix,descr)

remplace les entités associées au menu *refchoix* par les nouvelles entités contenues dans le descripteur *descr*. Pour cette primitive, la mise à jour de l'image n'est pas automatique, elle est doit être demandée explicitement à l'ide de la primitive *Aff_ObjDial.*

5.2.2.3 Consultation des choix

Suite à une séance de dialogue, il peut être intéressant pour l'application de connaître quelle entité du menu a été selectionnée. A cet effet, deux primitives de consultation lui sont proposées :

numsel <-- Cons_Choix(refchoix)

retourne le numéro de l'entité actuellement sélectée à l'intérieur du menu *refchoix*. Si aucune entité n'est sélectée la constante *absent* est renvoyée.

Cons_Ent_Choix(refchoix,nument,etat,texte)

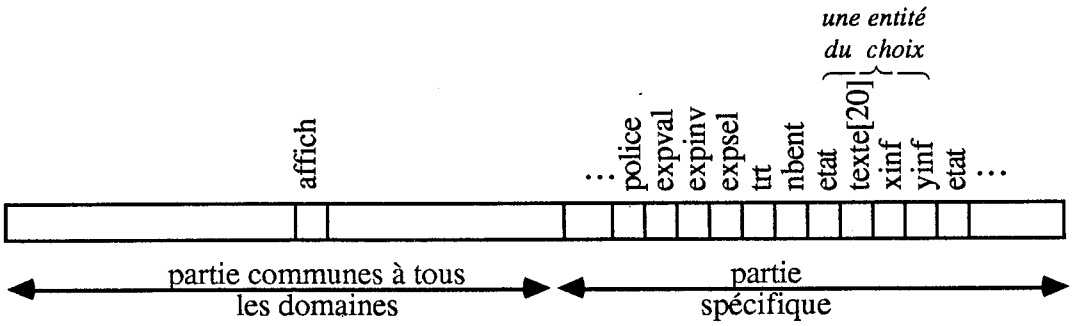
permet de récupérer l'etat (*etat*) et le nom (*texte*) de l'entité de rang *nument* dans le menu. Si *nument* est plus grand que le nombre d'entités effectivement présentes dans le menu, une chaîne vide avec l'état invalide est retournée.

5.2.3 Représentation des choix à l'aide des domaines

Dans les paragraphes qui suivent nous étudions plus en détail la manière dont les objets de type *choix* sont représentés à l'aide d'une hiérarchie de domaines. En particulier nous nous attacherons aux attributs associés aux différents domaines composant un *choix* et qui permettent la prise en compte automatique du dialogue pour un tel objet d'interface.

5.2.3.1 Le domaine choix

Le domaine choix est la racine de l'arborescence de domaines (c.f. fig. VII.18 p. 272) créée lors de la définition d'un nouveau menu déroulant. Le descripteur de ce domaine comprend une partie spécifique qui regroupe les attributs propres aux objets de dialogue de type choix. La figure VII.19 qui suit, présente une partie de ces attributs que nous avons déjà évoqués au § 5.2.2.2. (nous ne présentons que les attributs que nous jugeons nécessaires à une bonne compréhension du fonctionnement des menus déroulants, aussi un certain nombre d'attributs destinés à la gestion interne des choix ont volontairement été omis).



- figure VII.19 : descripteur du domaine choix -

Les principaux attributs spécifiques aux *choix* sont donc :

- *police,expval,expinvetexpsel*, attributs d'aspect pour l'affichage des entités dans la fenêtre texte,
-*trt*, adresse d'une fonction de traitement exécutée lors de la selection d'une entité, (elle est spécifique à chaque *choix* et est définie comme nous l'avons vu au niveau du programme qui crée le choix)
-*nbent*, le nombre total d'entités associées au menu et fournies à l'aide de *descr* dans la primitive *Def_Choix*.

pour chacune des entités du menu, sont conservées les informations suivantes :

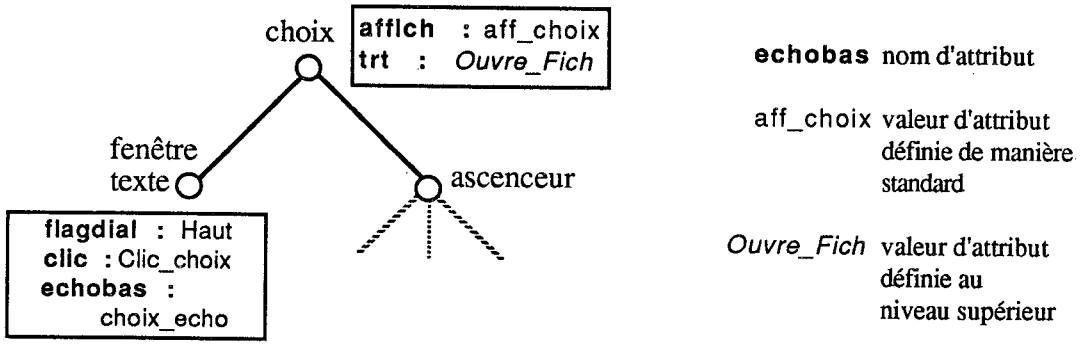
- *état* qui indique l'état *sélecté*, *valide* ou *invalide* de l'entité,
-*texte[20]* chaîne de 20 caractères qui est le nom de l'entité,
-*xinf,yinf* qui sont les coordonnées du coin inférieur gauche de l'entité.

L'attribut *affich* (que tous les objets de dialogue possèdent) est pour les objets de type *choix* initialisé avec l'adresse d'une fonction *aff_choix* qui assure la visualisation des sous domaines du choix que sont la fenêtre texte et l'ascenseur. Elle effectue en particulier l'affichage des différents noms des entités apparaissant dans la fenêtre texte (objets graphiques de type texte).

Par contre aucun attribut de dialogue n'est associé à ce domaine choix, la gestion de l'interaction s'effectuant entièrement sur les domaines de niveau inférieur.

5.2.3.2 La fenêtre texte

Le sous-domaine fenêtre texte, rattaché au domaine choix, (fig. VII.20) est créé à la définition du choix avec les attributs suivants (il ne possède pas d'attributs particuliers, son descripteur est un descripteur de domaine sans partie spécifique):



- figure VII.20 : attributs de la fenêtre texte d'un menu -

- *flagdial* = ***haut*** qui signifie que le dialogue sur ce domaine n'est initialement valide que pour le dispositif de dialogue en position haute.

- *echobas* = ***echo_choix*** est la fonction d'écho pour le dialogue en position basse sur le domaine. Elle calcule selon les déplacements du dispositif de dialogue sur ce domaine, le numéro de l'entité selectée et affiche son nom avec la couleur appropriée (couleur définie par l'attribut *expsel*. *expval* et *expinv* s'appliquent aux autres entités valides ou invalides).

- *clic* = ***clic_choix*** est l'adresse de la fonction qui est appelée automatiquement à chaque changement d'état du dispositif d'interaction.

Lorsqu'une transition descendante est enregistrée sur le domaine, elle permet de l'activer pour une séquence de dialogue en modifiant *falgdial* à ***tout***. (Ce n'est qu'alors que la fonction *echo_choix* peut être prise en compte).

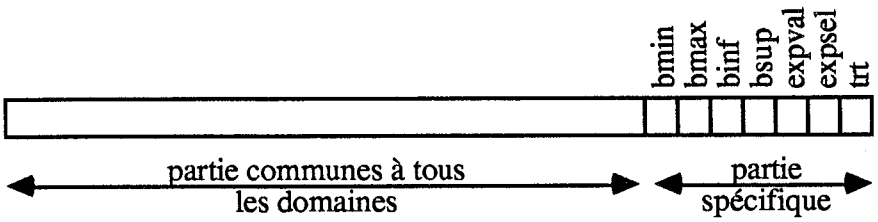
A la fin d'une séquence (transition *montante* ou *annul*) le dialogue en position basse est à nouveau invalidé sur le domaine (*flagdial* = ***haut***).

A chaque appel de la fonction *clic_choix*, la fonction *trt*, attribut du domaine père (domaine choix), est exécutée. Cette fonction *trt* permet d'associer directement aux actions effectuées par l'opérateur sur la fenêtre texte, un traitement propre au programme utilisant le menu. Ainsi on peut imaginer une fonction *trt* qui lors de la sélection d'une entité (transition montante sur la fenêtre texte) ouvre automatiquement le fichier dont le nom a été désigné.

5.2.3.3 L'ascenseur

L'ascenseur constitue lui même un objet de dialogue évolué : il correspond, en fait, à un type d'objet d'interface **indépendant** des menus déroulants qui du point de vue fonctionnel permet de sélectionner un intervalle à l'intérieur d'un domaine de valeurs (entières). L'étendue de ce domaine est symbolisée par la bande de défilement centrale de l'ascenseur. La valeur de l'intervalle courant est indiquée par la position et la taille (proportionnelle à la largeur de l'intervalle) d'un rectangle à l'intérieur de cette région.

Les bornes supérieure (*bmax*) et inférieure (*bmin*) du domaine de valeur, ainsi que les bornes (*bsup* et *binf*) de l'intervalle courant constituent à ce titre des attributs spécifiques des objets de dialogue de type ascenseur (fig VII.21).



- figure VII.21 : descripteur d'un ascenceur -

A ces informations s'ajoutent d'autres attributs propres aux ascenseurs sur lesquels nous reviendrons par la suite et qui sont :

- *expval* et *expsel*, expressions de couleur permettant de modifier la couleur des flèches ou du rectangle central de l'ascenseur afin de signaler (écho) à l'utilisateur l'étape d'interaction où il se situe,
-*trt*, adresse d'une fonction de traitement permettant de paramétrer les fonctions d'écho associées de façon standard aux ascenseurs.

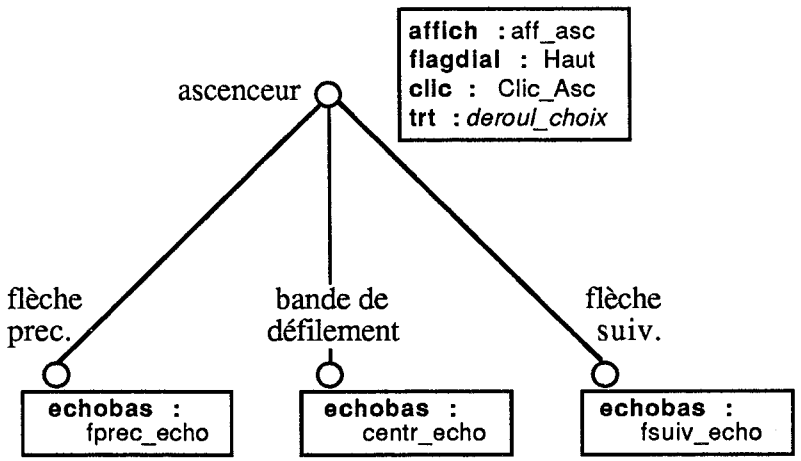
Les ascenseurs formant un type d'objet de dialogue, ils sont directement utilisables par une application ou pour la construction d'objets d'interface de niveau supérieur via un ensemble de primitives de haut niveau analogues à celles pour les menus.

La primitive

refasc <-- Def_Asc(refdom,xg,yg,xd,yd,valmin,valmax,valinf,valsup)

défini un ascenseur rattaché au domaine *refvue*. *xg*, *yg*, *xd*,*yd* indiquent la position et la taille de l'ascenseur dans le domaine de référence *refdom*. *Valmin* et *valmax* sont les bornes du domaine de valeurs associé à l'ascenseur, et *valinf* et *valsup* les valeurs d'initialisation de l'intervalle.

Cette primitive construit directement la hiérarchie des domaines composant un ascenseur en leur affectant les attributs standards (communs à tous les objets de type ascenseur) indiqués sur la figure VII.22.



- figure VII.22 : les attributs des ascenseurs -

La fonction d'affichage (*affich* = ***aff_asc***) associée au domaine racine de l'ascenseur assure la visualisation des flèches de déplacement et du rectangle dans la bande de défilement vertical (ce sont des objets graphiques de type polygône).

Initialement, le dialogue sur un ascenseur est uniquement valide pour le dispositif d'interaction en position haute (*flagdial* = ***haut***). Cette validité peut être modifiée si l'on "clique" sur l'ascenseur. A cet effet, l'attribut *clic* = ***clic_asc*** est une procédure appelée à chaque changement d'état du dispositif d'interaction sur l'ascenseur et qui positionne *flagdial* à ***tout*** si la transition est descendante et désactive l'ascenseur (*flagdial* = ***haut***) sinon.

Les échos associés aux différentes régions de l'ascenseur sont destinés au traitement du dispositif d'interaction en position basse (attribut *echobas*).

Les fonctions ***fprec_echo*** et ***fsuiv_echo*** sont appelées lorsque le dispositif (en position basse) se trouve sur les domaines situés aux extrémités de l'ascenseur (flèche vers le haut ou vers le bas). Elles affichent la flèche correspondante en utilisant l'une des expressions de couleur attributs de l'ascenseur :

- *expsel* (flèche sélectée) tant que le dispositif d'interaction reste en position basse sur le domaine,
- *expval* qui à la fin de la séquence d'interaction (passage du dispositif en position haute ou sortie hors du domaine) restitue à la flèche sa couleur originale (flèche valide).

A chaque appel de ces fonctions d'écho (dispositif en position basse sur le domaine) les bornes de l'intervalle courant (*binf* et *bsup*) sont incrémentées (ou bien décrémentées selon la direction de la flèche) d'une unité. Le rectangle symbolisant l'intervalle dans la bande de défilement (domaine central de l'ascenseur) est automatiquement déplacé en conséquence.

La fonction d'echo ***centr_echo*** associée au domaine central de l'ascenseur a un comportement analogue. Quand le dispositif de dialogue est en position basse sur cette région, le rectangle indiquant l'intervalle est déplacé au point désigné (avec la mise à jour des bornes de l'intervalle *binf* et *bsup*) et est affiché avec la couleur *expsel* . A la fin de la séquence d'interaction (dispositif en position haute ou sortie du domaine) le rectangle retrouve sa couleur initiale (*expval*).

En général un ascenseur est utilisé en liaison avec une autre entité graphique qu'il permet de modifier interactivement (par exemple, modification d'une grandeur en utilisant l'ascenseur comme "potentiomètre", déplacement dans une fenêtre...). Aussi, bien souvent est-il souhaitable de répercuter immédiatement les modifications de l'intervalle de l'ascenseur à cette entité graphique.

C'est pourquoi les fonctions d'écho standard des ascenseurs sont **paramétrables** à l'aide de l'attribut *trt*, adresse d'une fonction de traitement affectée au domaine racine de l'ascenseur. Cette fonction, attribut non standard défini au niveau du programme créant un ascenseur, est exécutée à chaque appel de l'une des fonctions d'écho *fprec_echo*, *fsuiv_echo* ou *centr_echo*.

Ainsi, dans notre exemple pour les objets de type choix (menus avec défilement vertical), l'ascenseur est directement lié à la fenêtre texte pour l'affichage des entités selectables. Le domaine de valeur de l'ascenseur correspond au nombre total d'entités selectables du menu, les bornes de son intervalle étant le nombre d'entités visibles dans la fenêtre texte à un instant donné. Au moment de la création d'un *choix*, lors de la définition de son ascenseur, l'attribut *trt* de ce dernier est initialisé avec l'adresse d'une fonction *deroul_choix* qui à chaque modification de l'intervalle de l'ascenseur déplace la fenêtre texte du menu du nombre de lignes correspondant.

La primitive **Def_Exec_Asc(refasc,adr_trt)** permet de modifier la fonction de traitement d'un ascenseur référencé par *refasc*.

Les autres primitives concernant les ascenseurs sont :

Asp_Asc(refasc, expval,expsel)

qui permet de modifier les aspects utilisés par les fonctions d'écho selon les étapes d'interaction. (Si l'ascenseur est affiché il n'y a pas de modification immédiate de celui-ci).

Maj_Asc(refasc,inf,sup)

qui permet de modifier l'intervalle courant de l'ascenseur *refasc*. Au contraire de *Asp_Asc*, si l'ascenseur est affiché l'image est automatiquement mise à jour.

Cons_Asc(refasc,inf,sup)

qui permet de récupérer les bornes de l'intervalle courant de l'ascenseur *refasc*.

6 STRUCTURATION DYNAMIQUE DE L'INTERFACE - LES CONTEXTES

La hiérarchie des domaines décrit la structure **statique** de l'interface utilisateur (et de manière plus générale de l'image). Le mécanisme de **contexte** que nous avons évoqué au § 5.1.2.4, permet de définir sur celle-ci une structure **dynamique d'exécution** : c'est à dire d'**associer** et de **limiter** lors du déroulement d'un programme, les échanges entre l'application et l'opérateur à l'un des éléments de cette structure arborescentes de domaines.

6.1 La notion de contexte d'exécution

Les contextes décrivent l'**environnement d'exécution** pour les opérations de visualisation et de dialogue : ils regroupent l'ensemble des paramètres qui les régissent (par exemple les attributs par défaut pour la visualisation d'objets graphiques, le dispositif d'interaction pour une séance de dialogue...).

Aussi, afin d'offrir le maximum de généralité et de souplesse, deux types de contextes destinés l'un à la visualisation, l'autre au dialogue, sont définis **séparément** et de façon **indépendante**.

La définition d'un nouveau contexe (de visualisation ou de dialogue) s'effectue par référence à l'un des éléments de la structure arborescente des domaines (primitive **Deb_Ecr(refdom)** pour la visualisation, **Deb_Dial(refdom)** pour le dialogue). Le domaine désigné sert alors de "support" aux échanges entre l'application et l'utilisateur, et une partie des attributs qui lui sont associés interviennent dans l'environnement d'exécution des opérations de visualisation ou de dialogue.

D'autres paramètres, non directement liés aux domaines sont regroupés à l'intérieur des contextes (par exemple, la nature du dispositif d'interaction utilisé pour le dialogue, les attributs élémentaires pour la visualisation d'objets graphiques). Lors de la définition d'un nouveau contexte, ces attributs sont initialisés avec des valeurs par défaut, ils peuvent être modifiés par la suite à l'aide de primitives appropriées que nous étudierons ultérieurement.

(remarque : les valeurs des attributs par défaut associés aux contextes, sont définies dans un fichier "ressources". En modifiant ce fichier, une application peut adapter ces valeurs par défaut à ses propres besoins sans recompilation).

La définition d'un nouveau contexte ne détruit pas le précédent : les contextes sont **préservés** grâce à un mécanisme de pile, la définition d'un nouveau contexte correspondant à son empilement. La limite de validité du contexte courant (sommet de pile) est indiquée à l'aide d'une primitive fin de contexte (**Fin_ecr()** pour la visualisation, **Fin_dial()** pour le dialogue) qui provoque son dépilement et le retour au contexte précédent.

L'**imbrication des contextes** qui correspond à leur empilement, présente un intérêt **majeur**. De la même manière que le mécanisme de blocs des langages de programmation structurés, elle permet une meilleure structuration et modularité des applications. En particulier, il est ainsi possible de **sous traiter "proprement"** des opérations de visualisation ou de dialogue à des sous programmes : ceux-ci en définissant leur propre contexte qu'ils détruisent avant le retour au programme appelant permettent de préserver ce dernier contre tout effet de bord indésirable. L'utilisation des contextes garanti ainsi un maximum d'indépendance pour les traitements effectués par de tels sous-programmes qui peuvent ainsi être réutilisés très facilement et sans risques au travers des applications (l'exemple ci-dessous, montre un programme d'affichage de palette de couleurs).

```
pallette(ref_pal); /* sous programme pour l'affichage d'une pallette et le choix d'une couleur */
{
    Deb_Ecrl(ref_pal); /* affichage sur la vue refpal */
    Deb_Dial(ref_pal); /* dialogue sur la vue où est affichée la palette */
        ....
        affichage de la palette et selection d'un couleur
        ....
    FinDial(); /* retour au contextes précedents */
    FlinEcr(); /* contextes du programme appelant */
}
```

La définition de contextes est au coeur de tous les échanges entre l'application et l'utilisateur. Elle est utilisée de façon interne par le logiciel graphique pour l'affichage d'objets d'interface et les séances de dialogue sur ceux-ci. Ainsi, lors de l'invocation des primitives de visualisation et de dialogue sur les objets d'interface, les contextes correspondants sont définis automatiquement et l'utilisateur n'a pas à s'en préoccuper. (Nous reviendrons plus en détail sur ces points aux § 6.2.2.2 et 6.3.3.2)

Cependant, cette notion de contextes est néanmoins nécessaire au programmeur d'applications afin de pouvoir gérer efficacement son propre affichage et/ou son dialogue par rapport aux objets de type *vue* qu'il peut être amené à définir (§ 5.1.2.4).

6.2 Contextes de visualisation

6.2.1 Définition de contextes de visualisation

La primitive

DebEcr(refdom)

désigne le domaine *refdom* comme nouveau contexte de visualisation (ou d'écriture). Elle défini un nouvel environnement d'exécution (associé au domaine *refdom*) qui, pour les opérations de visualisation subséquentes, détermine :

- l'image sur laquelle s'effectuent les actions d'écriture liées à l'affichage. Définie **implicitement** , cette image est celle à laquelle est rattaché le domaine contexte (*refdom*),
- les attributs globaux Ga, Gv et E utilisés pour l'affichage d'objets graphiques et qui sont ceux associés au domaine contexte (*refdom*),

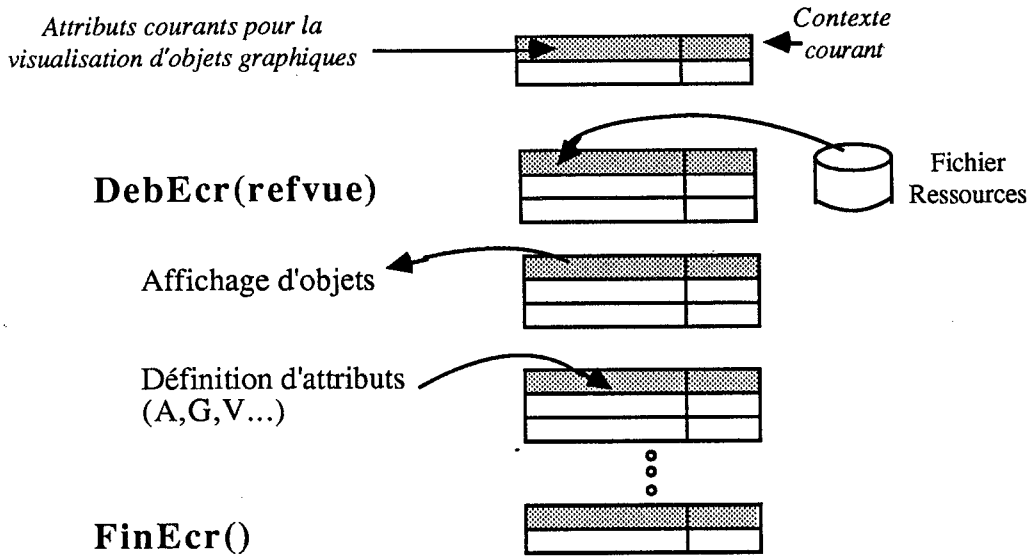
- l'ensemble des attributs utilisés pour l'évaluation et l'affichage d'objets graphiques (Couleur, Aspect, Visibilité, Reflexion). A la définition d'un nouveau contexte, ces attributs sont initialisés à partir de valeurs par défaut issues d'un fichier ressources paramétrable par l'application. Cet ensemble d'attributs, les attributs **courants** du contexte, peut évoluer au cours de la durée de vie du contexte. L'application peut si nécessaire les modifier par la suite, la définition d'un nouvel attribut remplaçe l'attribut courant. L'affichage d'un objet s'effectue

ainsi avec le dernier attribut spécifié. Nous reviendrons plus en détail sur ces attributs et la manière dont ils sont définis au chapitre suivant.

La définition d'un nouveau contexte d'écriture ne détruit pas le précédent, les informations qu'il regroupe sont sauvegardées à l'aide d'un mécanisme de pile (fig. VII.23). La primitive

FinEcr()

restitue le contexte précédent, dans l'état où il était avant le dernier appel de *DebEcr*. En particulier les attributs courants pour l'évaluation des objets graphiques pour l'affichage sont les derniers qui ont été spécifiés dans ce contexte.



- figure VII.23 : l'empilement des contextes et l'utilisation des attributs pour la visualisation -

Il existe un contexte de visualisation par défaut créé à l'initialisation du système et qui est toujours présent (il ne peut être dépilé). Ce contexte est associé à la racine de l'arborescence des domaines, à savoir le domaine correspondant à la vue par défaut *Image0*.

6.2.2 Utilisation des contextes de visualisation

Dans l'utilisation des contextes de visualisation que nous présentons ici, nous ne parlons pas de la définition et de la visualisation d'objets graphiques que nous étudions en détail au chapitre suivant. Nous présentons uniquement des éléments complémentaires liés à la création de contextes de visualisation.

6.2.2.1 Visualisation des cadres

Comme nous l'avons vu au § 2.2.1.1, la matérialisation d'un domaine s'effectue à l'aide d'un attribut de type cadre qui lui est associé. Deux primitives sont proposées pour afficher et effacer la zône utile et/ou le cadre du domaine contexte de visualisation courant.

Affcadre(pressepapier)

provoque l'affichage des différents éléments (zône utile, marges gauche, droite, haute et basse, titre) associés au cadre attaché au domaine contexte. L'affichage de ces différents éléments peut être sélectif, ceux-ci pouvant être invalidés à l'aide de la primitive **ValidCadre(refdom,mode)** qui positionne l'indicateur *flagcadre* du domaine *refdom* (voir § 2.2.1.1).

Avant l'affichage de son cadre, l'emplacement du domaine contexte peut être préalablement sauvegardé dans un "presse papier" (*pressepapier = presse*). Dans le cas, contraire (*pressepapier = non*) aucune sauvegarde n'est effectuée. Le "presse papier" est une zône de la mémoire d'image destinée à la sauvegarde de portions d'images, sa taille et sa position est définie par le programme d'application.

L'effacement de la zône utile et/ou du cadre du domaine contexte de visualisation est réalisé à l'aide de la primitive

Effcadre()

Cette primitive n'a de signification et d'effet que si le cadre du domaine contexte a été auparavant affiché avec l'option presse-papier. Dans ce cas, elle provoque la restitution du contenu précédent de l'emplacement du domaine.

6.2.2.2 Affichage d'objets d'interface

Lors de l'affichage d'objets d'interface à l'aide de la primitive *Aff_ObjDial* (voir § 4.2.2.1), un contexte sur le domaine racine de l'objet visualisé est automatiquement créé. Il permet d'effectuer toutes les opérations d'affichage liées à la visualisation de l'objet de dialogue dans un environnement qui lui est propre. A la fin de cette visualisation, le contexte ainsi créé est détruit afin de restaurer le contexte du programme appelant.

```
Aff_ObjDial(refdom,mode);
debut
   DebEcr(refdom);
      si mode cadre alors AffCadre(mode)
      si mode entité alors exécution de la fonction d'affichage du domaine.
   FinEcr();
fin;
```

De manière symétrique, la primitive d'effacement d'un objet de dialogue est réalisée comme suit :

```
Eff_ObjDial(refdom);
debut
   DebEcr(refdom);
      EffCadre();
   FinEcr();
fin;
```

6.3. Contextes de dialogue

6.3.1 La définition de contextes de dialogue

La primitive de définition d'un nouveau contexte de dialogue

DebDial(refdom)

désigne le domaine *refdom* comme support de l'interaction utilisateur et le rend **actif** pour le dialogue ainsi que **tous** les domaines qui lui sont hiérarchiquement dépendants. Cela siginifie que les attributs de dialogue de ces différents domaines sont pris en compte automatiquement lors des séquences d'interaction durant toute la durée de ce contexte (voir § 6.3.3). Les attributs des autres domaines de la structure arborescente sont ignorés.

remarque : Un domaine ne peut être utilisé comme contexte de dialogue que si il est actif (affiché et rattaché à un image). Dans le cas contraire une erreur le signale à l'opérateur.

Les autres paramètres regroupés dans le contexte de dialogue et indépendants du domaine désigné sont :

- la nature du dispostif d'interaction utilisé (tablette, souris ou clavier). Cet attribut est en fait l'adresse d'une fonction gérant le dispositif concerné, et qui est appelée automatiquement lors d'une séquence d'interaction (voir primitive *Collecte* au paragraphe suivant) (cette fonction retourne les coordonnées écran du point désigné ainsi que le numéro de la touche enfoncée).

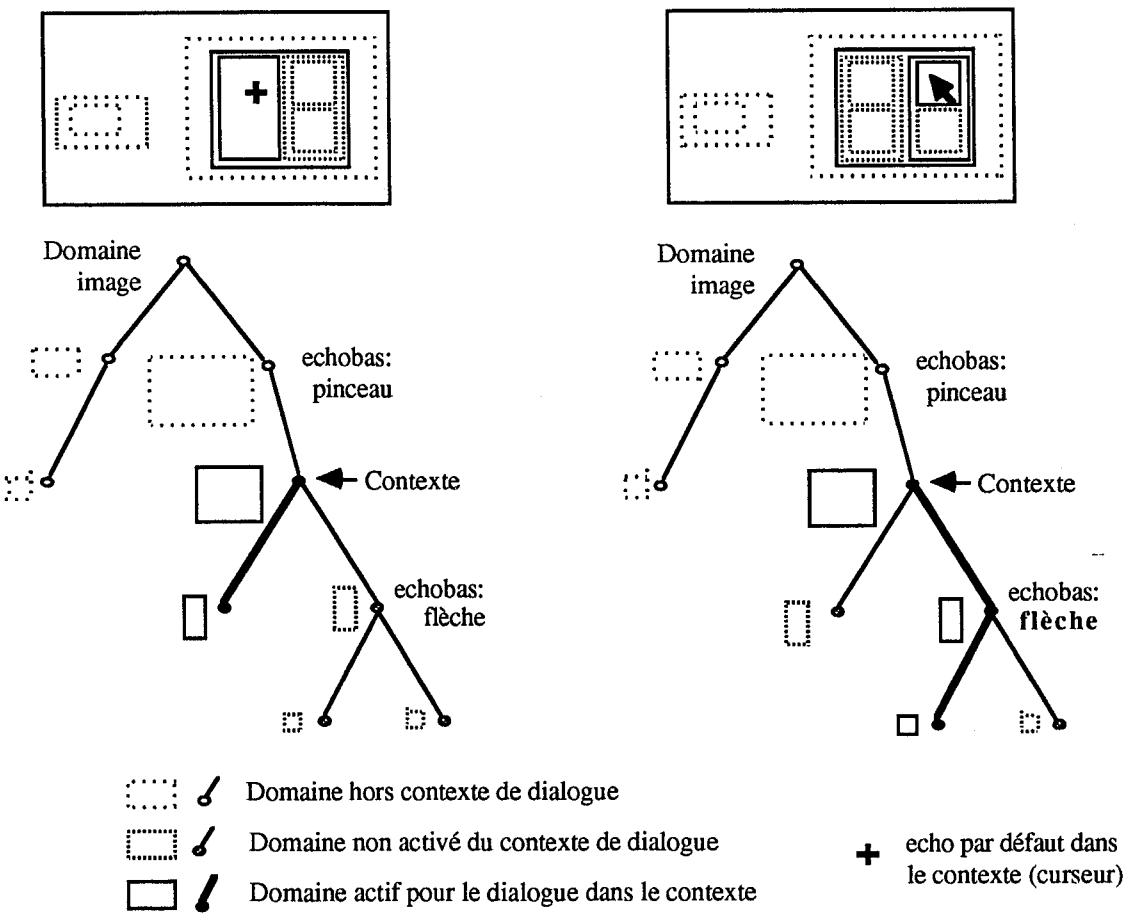
- l'adresse d'une fonction d'écho (et ses deux paramètres associés) utilisée lorsque le dispositif d'interaction se situe en dehors de la zône définie par le domaine contexte.

- des attributs utilisés par défaut, dans le cas où le domaine contexte et les domaines de sa sous-aborescence ne possèdent pas le (les) attribut(s) de dialogue correspondant. Il s'agit :

  • de l'adresse d'une fonction d'écho par défaut et ses deux paramètres associés (en général un motif de curseur et une expression de couleur). C'est cette fonction qui est appelée en dernier ressort si aucun des domaines (de la hiérarchie de domaines sous le contexte) dans lesquels se trouve le point désigné par le dispositif d'interaction ne possède d'attribut d'écho (fig. VII.24),

  • de l'adresse d'une fonction guide pour contraindre éventuellement la récupération des coordonnées (déplacements discrets, orthogonaux....),

  • de l'adresse d'une fonction définissant l'intérieur du domaine de dialogue.



- figure VII.24 : attributs par défaut dans un contexte de dialogue -

A la création d'un nouveau contexte ces paramètres sont initialisés avec les valeurs par défaut :

- *tablette* pour le dispositif d'interaction,

- *curseur* comme fonction d'écho par défaut et pour l'écho à l'exterieur du domaine contexte,

- une fonction guide inopérante (*nop*),

- *domaine_std()* une fonction qui défini l'intérieur de la région de dialogue comme étant la zône utile du domaine contexte.

Ces attributs du contexte courant peuvent être modifiés à tout moment par le programme d'application. (Les primitives permettant ces modifications sont présentées au paragraphe suivant)

Comme pour les contextes de visualisation, la définition des contextes de dialogue est possible jusqu'à seize niveaux d'imbrication. La terminaison (dépilement) du contexte courant s'effectue avec la primitive

FinDial()

qui restitue le contexte précédent. Comme pour la visualisation, un contexte de dialogue par défaut, associé au domaine *Image0*, est créé à l'initialisation du logiciel et ne peut jamais être fermé.

6.3.2 Modification des attributs du contexte de dialogue

La modification des paramètres liés au contexte de dialogue courant peut être effectuée à tout moment (après la définition du contexte) à l'aide des primitives suivantes :

DefDispo(dispo)

permet de spécifier le dispositif d'interaction utilisé lors des séances de dialogue. *dispo* est l'adresse d'une fonction gérant le dispositif concerné. Des fonctions prédéfinies *tablette, souris, clavier* sont à la disposition de l'utilisateur qui peut également créer ses propres fonctions.

DefEcho(zône,fecho,p1,p2)

permet la modification des fonctions d'écho du contexte. *zône* indique si il s'agit de la fonction pour le traitemnt de l'écho à l'extérieur du domaine (*zône = ext*) ou pour l'écho par défaut (*zône = int*). *fecho* est l'adresse de la fonction d'écho, là aussi sont proposées un certain nombre de fonctions prédéfinies (*EchoNul, Curseur, Pinceau, ...*). *p1* et *p2* sont des paramètres de la fonction d'écho, leur l'interprétation dépend bien entendu de cette dernière (par exemple *p1* est un numéro de motif de curseur et *p2* une expression de couleur pour l'echo *Curseur*).

DefGuide(guide)

modifie la fonction de guide du contexte. *guide* est l'adresse d'une fonction qui permet d'imposer des contraintres sur les coordonnées récupérées.

DefDomaine(fdom)

remplace la fonction courante définissant l'intérieur domaine du contexte (i.e. la région considérée comme zône de dialogue) par la fonction d'adresse *fdom* .

6.3.3 Les opérations dans le contexte de dialogue

6.3.3.1 La primitive Collecte()

A la base de la gestion du dialogue se trouve la primitive **Collecte()** qui provoque la collecte d'informations à partir du dispositif de saisie. Elle doit être invoquée avant tout autre récupération d'informations lors d'une séquence d'interaction. Cette fonction retourne 0 si le dispositif de saisie est en position haute, le numéro de la touche enfoncée dans le cas contraire. Si le dispositif d'interaction utilisé est une tablette à digitaliser fonctionnant avec un stylet, le code 0 correspond au stylet levé, le code 1 au stylet appuyé.

C'est cette primitive **Collecte()** qui, en fonction de l'environnement défini par le contexte de dialogue courant, récupère les données à partir du périphérique d'entrée, gère les déplacements de l'écho, assure l'interprétation des événements de dialogue. Page 288, nous donnons sous une forme pseudo-algorithmique son schéma de fonctionnement général, en particulier la manière dont les attributs (de dialogue) des domaines de l'arborescence définie sous le contexte courant sont pris en compte.

On distingue trois traitements selon qu'il s'agit du premier appel de **Collecte()** dans le contexte, qu'il y a changement ou non du domaine courant de dialogue (domaine situé le plus bas dans l'arborescence des domaines à partir du domaine contexte et qui contient le point désigné par le dispositif d'interaction).

Afin d'effectuer ces derniers traitements, il est nécessaire de conserver l'état du dialogue entre deux appels successifs de **Collecte()**. Pour cela sont mémorisés dans les structures de données du contexte un certain nombre de paramètres : n° de la dernière touche du dispositif d'interaction actionnée, domaine de dialogue, fonction d'echo lors du dernier appel de collecte, dernière fonction clic utilisée...

On trouvera pages suivantes l'algorithme détaillé de la primitive **Collecte()** dont nous explicitons certains points ici :

1) La sélection de la fonction d'écho courante s'effectue en parcourant la hiérarchie des domaines situés entre le domaine contexte de dialogue et le domaine courant (*domcour*). Est selectionnée la fonction de niveau le plus bas parmi ces domaines. Si aucun de ces domaines ne possède d'attribut d'écho, c'est la fonction d'écho par défaut associée au contexte qui est retournée (voir § 6.3.1). De la même manière, dans le cas où le point courant de dialogue est à l'extérieur du domaine contexte (*domcour* = nil), ce sont les attributs d'écho par défaut correspondant associés au contexte qui sont pris en compte.

2) Dans le cas où l'on change de domaine courant de dialogue ou de touche, l'exécution de la fonction d'écho de l'appel précédent de contexte (*echoprec*) avec la transition dernier sert à terminer correctement l'écho (par exemple effacer le curseur) avant de passer à une nouvelle fonction d'écho.

3) Lorsque dans **Collecte()** un changement d'état du dispositif d'interaction est enregistré (n° touche cour ≠ n° touche prec) il y a prise en compte des attributs de traitement du dialogue (fonctions *clic*) associés aux domaines rattachés au domaine contexte (voir § 2.2.3.3).

Si la transition est **descendante** (n° touche prec = 0), la fonction *clic* située le plus bas entre le domaine contexte et le domaine de dialogue courant est selectionnée. (Si aucun de ces domaine ne possède d'attribut *clic*, ou si le domaine courant est hors contexte (*domcour* = Nil) une fonction nulle (nop) est selectée). Une fois déterminée, cette fonction *clic* est exécutée avec les deux paramètres qui lui sont associés, à savoir : le type de transition (ici descendante) et la référence du domaine auquel elle est rattachée.

C'est cette même fonction qui sera exécutée lors du prochain changement d'état du dispositif d'interaction dans le contexte. Aussi, afin de pouvoir la réutiliser lors d'appels

```
Collecte()
debut
   (* domprec = nil si premier appel de collecte dans le contexte courant
            = référence du domaine de dialogue lors du dernier appel de collecte  sinon.
      n° touche prec = 0  (position haute) si premier appel de collecte dans le contexte courant,
                  = n° touche enfoncée lors du dernier appel de collecte sinon.
      echoprec = nop si premier appel de collecte dans le contexte courant
               = adresse de la fonction d'echo invoquée au dernier appel de collecte sinon*)

   exécution de la fonction dispositif du contexte courant;
   (* récupère les coordonnées écran du point courant + le numéro de la touche enfoncée
    (n° touchecour) */

   domcour <- domaine du dialogue;
   (* domcour <--  référence du domaine situé le plus bas dans l'arborescence sous le domaine
   contexte et qui contient le point désigné par le dispositif d'interaction, Nil si le dispositif est
   situé en dehors de la zône utile du contexte *)
   cas

      1er appel de collecte :

            si dispositif en position basse (n° touche ≠ 0) alors
               maj de la fonction clic;
               exécution de la fonction clic;
            finsi
            echocour <- selection de la fonction d'écho; 1
            exécution de la fonction d'echo echocour avec la transition premier;

      domprec ≠ domcour :

            exécution de la fonction d'écho echoprec avec la transition dernier 2
            si n° touche ≠ n° touche prec alors
               si transition descendante alors
                  maj de la fonction clic;
               finsi
               exécution de la fonction clic;
            finsi
            echocour <- selection de la fonction d'écho; 1
            exécution de la fonction d'echo echocour avec la transition premier;

      domprec = domcour :

            si n° touche ≠ n° touche prec alors
               exécution de la fonction d'écho echoprec avec la transition dernier 2
               si transition descendante alors
                  maj de la fonction clic;
               finsi
               exécution de la fonction clic;
               echocour <- selection de la fonction d'écho;1
               exécution de la fonction d'echo echocour avec la transition premier;
            sinon
               exécution de la fonction d'echo echocour avec la transition interm;
            finsi
   fincas
   domprec <- domcour;
   n° touche prec <- n° touche;
   echoprec <- echocour;
   retour(n° touche);
fin
```

ultérieurs à ***Collecte()***, son adresse ainsi que la référence du domaine auquel elle est associée sont mémorisées dans les structures de données du contexte de dialogue.

Ainsi, quand ***Collecte()*** détecte le passage du dispositif d'interaction de position basse à position haute (*n° touche cour* ≠ *n° touche prec* et *n° touche cour* = 0) la fonction clic du contexte est exécutée avec la référence du domaine auquel elle est attachée avec la transition :

- **montante** si lors du dernier appel de collecte le domaine de dialogue (domprec) était celui auquel est rattaché la fonction clic,

- **annulation** sinon.

6.3.3.2 Utilisation de la primitive Collecte()

Comme pour les contexes, la primitive ***Collecte()*** peut être utilisée à deux niveaux différents :

- de manière implicite et transparente via la primitive ***Séance(refdom)*** sur un objet d'interface (voir § 4.2.2.2),

- de manière explicite dans une séquence d'interaction au cours de laquelle l'application effectue elle-même sa propre gestion du dialogue.

a) Séance de dialogue sur un objet d'interface.

Pour les objets d'interface les séquences d'interaction sont entièrement définies par les attributs de dialogue associés aux domaines qui les constituent. En particulier le traitement des évenements de dialogue est effectué au travers des fonctions *clic*, *exec* dont l'appel est comme nous vennons de le voir pris en charge par la primitive ***Collecte()***. Aussi la primitive ***Séance(refdom)*** qui initialise une séquence d'interaction sur un objet d'interface se résume à :

```
Seance(refdom);
debut
   DebDial(refdom);
      Fin_de_Seance := faux;
      repeter
         tant que (non Fin_de_Seance) et (non Collecte()) faire ;
         tant que (non Fin_de_Seance) et (Collecte()) faire ;
      jusqu'à Fin_de_Seance ;
   FinDial();
fin;
```

Le domaine "racine" de l'objet d'interface (de référence *refdom*) devient le contexte de dialogue courant. La séance de dialogue consiste ensuite à "boucler" sur la primitive ***Collecte()*** jusqu'à ce que l'indicateur *Fin_de_Seance* soit positionné à vrai. Cet indicateur est intégré aux structures de données définissant le contexte de dialogue courant, il est positionné à vrai par l'intermédiaire de la primitive ***FinSeance()*** qui, comme nous l'avons déjà vu au § 4.2.2.2, doit être invoquée au moins une fois dans l'une des fonctions de traitement du dialogue des domaines constituant l'objet d'interface.

A la fin de la séance de dialogue, le contexte est bien entendu dépilé afin de restituer le contexte du programme appelant.

b) Utilisation explicite de *Collecte()* et primitives complémentaires.

Il est possible de gérer directement le dialogue au niveau de l'application, sans passer par des objets d'interface et la primitive ***Séance()***. La primitive ***Collecte()*** doit alors être

invoquée avant toute autre opération. Typiquement une séquence d'interaction se déroule alors de la façon suivante :

```
DebDial(refvue); (* contexte de dialogue sur la vue support de l'interaction *)
   ....
   tant que (non Collecte()) faire ; (* boucle d'attente d'activation du dispositif *)
   ...
   tant que Collecte() faire
   debut
      ...
      (* actions de récupération d'informations et de traitement *)
      ...
   fin
   ...
FinDial();
```

Un certain nombre de primitives sont proposées pour récupérer des informations sur l'état courant du dialogue.

CollXY(refdom,x1,y1)

permet de récupérer les coordonnées du point courant de la séance de dialogue (point récupéré lors du dernier appel de ***Collecte()*** et mémorisé dans les structures de données du contexte). Ces coordonnées sont exprimées dans le repère du domaine indiqué comme paramètre (en général une vue).

CollZône(refdom,x1,y1,x2,y2)

permet de récupérer les coordonnées de deux points définissant une zône rectangulaire : le point où a eu lieu le dernier changement d'état d'une touche et le point courant (les coordonnées sont exprimées dans le repère du domaine *refdom*)

CollVidéo(Rouge,Vert,Bleu) --> numimage

permet de récupérer la couleur finale (au niveau des signaux vidéo) aux coordonnées du point courant de la séance de dialogue et retourne le numéro (0..4) de l'image visible en ce point.

6.4 Lecture et Ecriture d'informations en mémoires d'image

Nous présentons brièvement dans ce paragraphe les fonctionnalités proposées par le logiciel graphique pour les transferts rapides d'information directement à partir et/ou vers les mémoirs d'image. En effet, le matériel est pourvu d'opérateurs cablés qui autorisent des lectures écritures en mémoire de trame à très grande vitesse (13,5 M de pixels 12 bits par seconde).

Nous avons placé l'étude de ces primtives à ce niveau car elles complètent à la fois les opérations de collecte d'informations et et les opérations de visualisation et pour ce fait utilisent le même mécanisme de contextes.

6.4.1 Lecture des informations en mémoire d'image - Les contextes de lecture.

Après avoir récupéré les coordonnées du point courant à l'aide de l'une des primitives de collecte présentées ci-dessus, il est possible d'aller lire directement les informations enregistrées au point correspondant en mémoire d'image (plan mémoire). La nature des informations ainsi récupérées dépend bien entendu de la configuration de l'image

correspondante : selon les cas il peut s'agir d'une couleur, d'un numéro d'identification, d'une normale etc.....

Afin de faciliter la définition des zônes de lecture dans les plans mémoire, celles-ci sont spécifiées à l'aide de coordonnées exprimées dans les repères des domaines. Ainsi, la lecture peut être effectuée directement après avoir recupéré les coordonnées d'un point désigné par l'utilisateur au cours d'une séquence d'interaction. En effet ces coordonnées sont définies dans le repère d'un domaine spécifié par l'application (voir § 6.3.2.2.3). C'est pourquoi, comme pour la visualisation et le dialogue, les opérations de lecture se déroulent relativement à un domaine courant (qui définit implicitement l'image et le repère dans lequel sont évaluées les coordonnées). De manière analogue, la définition du domaine courant de lecture s'effectue à l'aide d'une déclaration de contexte qui permet ainsi de préserver l'environnement d'exécution des opérations de lecture et de structurer les applications.

DebLec(refdom); (* contexte de lecture sur le domaine *refdom* *)
...
actions de lecture élémentaires sur le domaine *refdom*
...
FinLec(); (* retour au contexte précédent *)

(remarque : comme pour la visualisation et le dialogue, lors de l'initialisation du logiciel un contexte de lecture est ouvert par défaut sur la vue *Image0* et ne peut être fermé.)

Outre l'identification du domaine sur lequel les lectures sont réalisées (repère pour l'évaluation des coordonnées), le contexte de lecture défini également la manière dont elles sont effectuées : direction de lecture, masque de lecture (lecture en continu, en pointillée, etc...). La modification des paramètres du contexte courant est réalisée à l'aide de la primitive

ParamLec(direction,masque)

La lecture effective s'effectue au travers de la primitive

Lec(typeinfo,xdeb,ydeb,nbpix,buffer)

qui permet de lire l'information associée à *nbpix* pixels pris à partir du point de coordonnées *xdeb*, *ydeb* (dans le repère du domaine contexte de lecture) selon la direction courante du contexte (définie par *ParamLec*). Cette information est stockée dans le tableau d'entier *buffer* de longueur supérieure ou égale à *nbpix*. Le paramètre *typeinfo* est un code indiquant la nature des informations à lire (Couleur poids faibles, Couleur poids forts, numéro d'identification, vecteur normal, profondeur...). C'est ce paramètre qui associé avec la référence du domaine contexte de lecture détermine le plan mémoire où les informations sont récupérées.

6.4.2 Ecriture des informations en mémoire d'image - Transferts de blocs de Pixels

De façon totalement symétrique à la primitive *Lec*, la primitive

Ecr(TypeInfo,xdeb,ydeb,nbpix,buffer)

permet l'écriture bufferisée directement en mémoire d'image de l'information 12 bits sur les pixels d'une ligne de direction donnée.

Comme pour les lectures, l'écriture s'effectue par rapport à un domaine donné (*xdeb* et *ydeb* sont des coordonnées exprimées dans son repère). Pour la définition de ce domaine courant d'écriture, on utilise le même mécanisme que pour les contextes de visualisation d'objets (§ 6.2.1). Ainsi, la primitive *DebEcr(refdom)* défini simultanément un contexte de visualisation et un contexte d'écriture associé au domaine *refdom*. En plus des informations

nécessaires à l'affichage (voir § 6.2.1), les contextes de visualisation/écriture regroupent les paramètres propres à l'écriture. Ceux-ci, anlogues aux paramètres liés au contexte de lecture (direction de lecture, masque de lecture) peuvent à tout moment être modifiés par la primitive

ParamEcr(direction,masque)

Finalement, complétant les primitives de lecture/écriture bufferisées, une dernière primitive permet le transfert immédiat d'informations d'un domaine à un autre sans passer par une mémorisation intermédiaire, utilisant ainsi de façon optimale les mécanismes cablés de lecture et écriture. Ainsi

Transfert(xglec,yglec,xdlec,ydlec,xgerc,ygecr)

effectue un transfert rapide d'un bloc rectangulaire du domaine contexte courant de lecture directement vers le domaine contexte courant d'écriture. *xglec, yglec* et *xdlec, ydlec* sont les coordonnées des coins inférieur gauche et supérieur droit de la zône à recopier, exprimées dans le repère du domaine contexte de lecture. *xgecr, ygecr* sont les coordonnées du coin inférieur gauche du rectangle à écrire, exprimées dans le repère du domaine contexte d'écriture.

(remarque : seules les informations contenues dans les plans mémoires et présentes à la fois dans la configuration de l'image de lecture et celles de l'image d'écriture sont transférées.)

7 CONCLUSION

Au travers des différentes fonctionnalités que nous avons présentées au cours de ce chapitre, nous pensons avoir posé les bases nécesaires à la prise en compte par le système graphique des problèmes liés à l'interface utilisateur.

La notion de domaine, leur structuration hiérarchique, permettent de décrire des objets de dialogue évolués. Les primitives de visualistion *(Aff_ObjDial)* et de séance d'interaction (*Seance*) déchargent totalement l'application des problèmes liés à l'affichage de tels objets et à leur utilisation lors d'une séquence d'interaction. En particulier, c'est la primitive *Collecte()* que nous avons étudiée en détail qui assure l'interprétation des actions effectuées par l'opérateur à l'aide des attributs associés aux domaines supports du dialogue interactif.

Un deuxième point à souligner, qui découle de ce qui précède, est la possibilité offerte par le système pour la définition de nouveaux objets de dialogue. Le système en proposant des primitives standard pour la définition et la strucutration des domaines permet de définir aisément les primitives de haut niveau pour la manipulation d'objets de dialogue d'un type spécifique.

Le système encourage une programmation extrêmement modulaire pour la définition des objets de dialogue, un objet de dialogue correspond à un module, seules les primitives de haut niveau pour sa manipulation (définition, attribution, consultation) sont accessibles à l'utilisateur, elles permettent un encapsulation masquant les structures de données décrivant l'objet. On retrouve ici les fonctionnalités associées à la programmation orientée objet, qui d'ailleurs trouve un domaine d'application privilégiée dans la définition d'interfaces homme-machine [Cox 86]... Les domaines constitueraient ainsi la classe de base à partir de laquelle seraient construits par héritage les objets de dialogue.. Cependant, les langages orientés objets s'ils ne reposent pas sur des concepts nouveaux ne sont pas encore largement diffusés et nécessitent un environnement matériel relativement important (stations de travai), d'autre part l'intégration des fonctionnalités liées à la definition de l'interface utilisateur à l'intérieur d'un système graphique complet, nécessite un degré de performances que la communication par messages propre aux langages orientés objets risque d'affecter. Ce les raisons pour lesquelles nous avons développé tout le système en langage C. Toutefois, cette voie mériterait d'être explorée plus en avant, la programmation orientée objet permettrait en paticulier un prototypage rapide et sur, de nombreuses fonctionnalités étant intégrées au langage lui-même.

Un autre aspect important des fonctionnalités que nous avons présentées dans ce chapitre, est la notion de contexte qui permet de structurer dynamiquement l'interface des applications en associant les opérations d'échange (dialogue, visualisation, lecture ou écriture d'informations sur une image) aux éléments de la description statique de l'interface. L'existence simultanée de trois types de contexte (Ecriture, Dialogue, Lecture) , permet à l'application de contrôler séparément l'environnement dans lequel s'effectuent les opérations d'affichage, d'interaction, ou de lecture d'informations en mémoire d'image. Leur imbrication possible décharge l'application des problèmes liés aux changement d'environnement en permettant le retour immédiat à un contexte initial (fin d'un contexe et retour au contexte précédent).

Beaucoup, plus général que dans CLOVIS, le mécanisme de contexte permet de couvrir l'ensemble des besoins des applications. La définition de contextes est à la base de la programmation des applications. En particulier, nous verrons dans le chapitre qui suit, que l'emploi des contextes d'écriture (que nous avons présentés ici pour la visualisation et l'écriture en mémoire d'image) joue un rôle primordial dans la définition des objets graphiques et permet de combiner aisément l'affichage immédiat d'objets avec leur mémorisation comme nous l'avons déjà évoqué au chapitre VI.

Chapître VIII : Les Objets Graphiques

1 Introduction

2 Mémorisation et affichage immédiat

2.1 Modes visualisation et mémorisation - Contexte d'écriture
2.2 Mémorisation temporaire et récupération de l'espace mémoire
2.3 Accès aux objets mémorisés - Références

3 Attribution

3.1 Définition d'objets et d'attributs élémentaires libres
3.1.1 Définition d'objets élémentaires - Morphologie
3.1.2 Définition d'attribus élémentaires - Aspect, Couleur, Visibilité, Géométrie, Reflexion
3.1.2.1 Prise en compte des attributs élémentaires libres
3.1.2.2 Modelisation des attributs dans les différentes classes
3.1.2.2.1 Couleur
3.1.2.2.2 Aspect
3.1.2.2.3 Visibilité
3.1.2.2.4 Reflexion
3.1.2.2.5 Géométrie
3.2 Definition d'objets structurés - STRUCTURE - IDENTITE
3.2.1 La structuration des attributs graphiques - notion de scène
3.2.2 Le Partage d'Objets
3.2.3 Composition des attributs graphiques
3.2.3.1 Règles de composition des attributs élémentaires
3.2.3.2 Les attributs groupés.
3.2.4 Identification des objets
3.2.4.1 Nomination des objets
3.2.4.2 Primitives d'accès aux objets structurés
3.2.5 Création d'Objets Structurés
3.2.5.1 Construction progressive de structure
3.2.5.1.1 Définition de scène - Scène courante
3.2.5.1.2 Nomination des objets
3.2.5.1.3 Réutilisation d'attributs et d'objets mémorisés - Attribution Implicite
3.2.5.1.3.1 Réutilisation d'attributs graphiques élémentaires
3.2.5.1.3.2 Réutilisation d'objets graphiques
3.2.5.1.4 Intêret de la construction progressive de structure
3.2.5.2 Construction de structure à partir d'expressions de noms
3.3 Réutilisation d'attributs et d'objets mémorisés - Attribution Explicite
3.3.1 Affectation d'attributs élémentaires
3.3.2 Affectation d'objets
3.4 Modification d'attributs et d'objets élémentaires mémorisés.

4 Consultation

4.1 Primitives de consultation de structure
4.2 Primitives de consultation d'attributs élémentaires

5 Description

6 Visualisation

7 Conclusion

1 INTRODUCTION

De la même manière que le chapitre VII détaillait les mécanismes proposés pour la définition de l'interface des applications, ce chapitre VIII précise les fonctionnalités destinées à la définition et la visualisation des objets graphiques que nous avons déjà présentées rapidement au cours du chapitre VI.

Comme nous l'avons vu alors, le point essentiel qui distingue celles-ci par rapport à ce qui avait été fait pour CLOVIS, concerne l'utilisation possible de plusieurs modes qui permettent :

- la **visualisation immédiate** des objets graphiques, sans aucune mémorisation des attributs les définissant,

- la **mémorisation éventuelle** des attributs et objets graphiques dans la banque de données gérée par le système graphique,

- l'**utilisation directe** des attributs graphiques de base pour la définition d'objets élémentaires **ou bien** leur **structuration hierarchique** pour la définition d'objets complexes.

Dans ce chapitre nous essayons de montrer la richesse et la souplesse qu'offre le système graphique par l'utilisation de ces différents modes qui rappelons-le peuvent être combinés entre eux (table VIII.1).

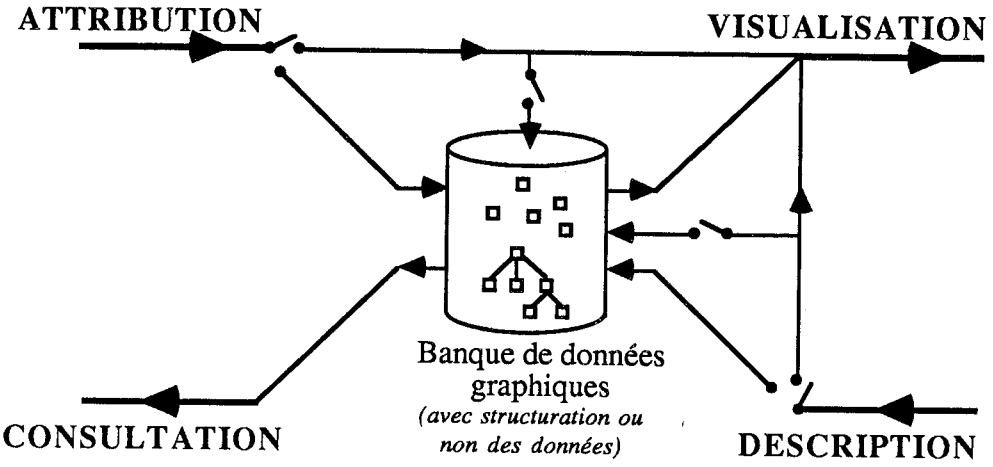
Affichage Immédiat	Mémorisation	Structuration
■		
■	■	
■		■
■	■	■
	■	
	■	■

- table VIII.1 : combinaison des différents modes de définition des objets graphiques -

Tout d'abord nous présentons la manière dont l'application peut choisir entre les modes de visualisation immédiate et/ou de mémorisation lors de la définition des attributs graphiques. Ce choix s'effectue à l'aide du mécanisme de définition de contextes d'écriture que nous avons présenté au chapitre VII et qui a été généralisé et entendu à la mémorisation.

Ensuite, nous présentons les fonctionnalités propres à la définition et à la manipulation des attributs et objets graphiques. Comme nous l'avons fait tout au long de cette thèse, nous avons tenté de les regrouper selon les processus de base que sont l'attribution, la consultation, la description et la visualisation.

Cependant, de par l'utilisation conjointe des différents modes, il est clair que la distinction entre les processus est beaucoup moins nette qu'elle ne l'était dans CLOVIS (fig VIII.1). En particulier, l'attribution (et de manière symétrique la description) n'est plus nécessairement séparée de la visualisation.



- figure VIII.1 : les processus pour le logiciel de visualisation d'objets graphiques -

Sous le terme d'attribution, nous avons regroupé tout ce qui concerne la définition par le programme d'application d'attributs élémentaires, leur possible structuration en hiérarchies d'objets et la réutilisation d'attributs et d'objets mémorisés.

La définition des attributs élémentaire est présentée, de manière désormais "classique", en rassemblant les primitives selon la classe d'attributs concernée (Morphologie, Aspect, ou Géométrie).

Pour le second point, si l'emploi de la structure hiérarchique présente de nombreuses analogies avec CLOVIS, le traitement des attributs morphologiques de base en tant qu'objets graphiques à part enitière a entrainé un certain nombre de modifications que nous exposons. Par ailleurs, nous avons également introduit une plus grande souplesse dans la définition de la structure hiérarchique qui peut éventuellement être effectuée de manière progressive (en particulier en liaison avec un affichage immédiat).

Le dernier point présente la réutilisation d'attributs et d'objets mémorisés dans la banque de données du système graphique. Ceci peut intervenir soit pour une visualisation immédiate, soit pour la définition d'objets structurés.

En ce qui concerne la Consultation et la Description nous sommes beaucoup plus bref. En effet la prise en compte de ces deux processus ne présente que peu de nouveautés par rapport aux fonctionnalités offertes par CLOVIS.

Finalement, pour le processus de visualisation, il est défini de manière implicite par la configuration matérielle utilisée. La visualisation d'objets graphiques s'effectue par la combinaison du mode d'affichage immédiat et l'emploi des primitives d'attribution présentées précedemment. Le système ne propose donc pas de fonctionnalités particulières, autres que celles offertes pour la définition des modes graphiques et l'attribution.

2 MEMORISATION ET AFFICHAGE IMMEDIAT

2.1 Modes visualisation et mémorisation - Contexte d'écriture

Comme nous l'avons déjà évoqué, à la différence de CLOVIS, ce second système permet aux applications de travailler selon plusieurs modes :

Dans le mode **visualisation simple** (ou affichage immédiat), les attributs et objets graphiques sont utilisés et affichés dans le contexte de la vue courante (par défaut la vue

Image0) dès leur définition et l'espace mémoire qui leur est alloué est libéré aussitôt leur affichage terminé.

Dans le mode **mémorisation simple**, tous les attributs et objets définis sont mémorisés dans la banque de données du logiciel graphique sans qu'aucun affichage ne soit effectué.

Le mode **visualisation et mémorisation simultannées** combine les deux modes précédents : les objets définis sont non seulement mémorisés, mais également affichés avec les mêmes régles que pour l'affichage immédiat.

Ainsi les mêmes primitives de définition d'attributs et d'objets graphiques peuvent, selon le mode dans lequel elles sont invoquées, provoquer un affichage et/ou une mémorisation.

La spécification des modes s'effectue sous forme de contexte, la fin d'un mode provoquant le retour au mode précédent. Elle est réalisée avec les mêmes primitives ***DebEcr*** et ***FinEcr*** que pour la définition de contextes de visualisation (cf § 6 du chapitre VII). Ces primitives délimitent un **contexte d'écriture** au sens large, qui selon la valeur du paramètre de *DebEcr* peut être :

- soit un contexte **visualisation simple** (le paramètre est la référence du domaine support de la visualisation),

- soit un contexte de **mémorisation simple** (le paramètre est la constante prédéfinie *Banque*),

- soit un contexte de **mémorisation et visualisation simultannées** (le paramètre est la constante *Banque* combinée avec la référence d'un domaine).

Les contextes d'écriture peuvent être imbriqués avec les règles que nous avons présentées au chapitre VII, indépendamment du fait qu'ils mentionnent ou non une mémorisation. La primitive ***FinEcr()*** restitue automatiquement le contexte précédent. A son initialisation, le système se trouve dans le mode de visualisation immédiate dans le contexte de la vue par défaut ***Image0***. Ce contexte d'écriture par défaut est toujours présent et ne peut être supprimé. L'exemple qui suit, illustre l'emploi de contexte pour le passage dans les différents modes possibles.

```
DebEcr(Banque);
   .... /* les objets et les attributs définis sont simplement mémorisés*/
   DebEcr(refvue);
      .... /* les objets et attributs définis sont simplement affichés */
      DebEcr(Banque);
         ... /* les objets et les attributs définis sont simplement mémorisés*/
      FinEcr();
      .... /* les objets et attributs définis sont simplement affichés */
   FinEcr();
   .... /* les objets et les attributs définis sont simplement mémorisés */
   DebEcr(Banque + Refvue);
      /* les attributs et les objets définis sont mémorisés et affichés
         dans le contexte de la vue RefVue */
      ...
   FinEcr();
   ....
FinEcr();
```

2.2 Mémorisation temporaire et récupération de l'espace mémoire

Il n'est pas toujours nécessaire de mémoriser certains attributs ou objets durant toute la durée d'une application. Les primitives ***Bloc()*** et ***FinBloc()*** permettent de délimiter un bloc de mémorisation **temporaire**.

```
Bloc();  /* début d'un bloc de mémorisation temporaire */
   ...
   ...
FinBloc(); /* fin du bloc de mémorisation temporaire */
```

Les objets et attributs définis et mémorisés à l'intérieur d'un tel bloc ne sont accessibles que dans celui-ci. Après l'exécution de la primitive ***FinBloc()*** , l'espace qui leur a été alloué dans la banque de données graphique est libéré, et les variables de l'utilisateur contenant les références vers ces éléments n'ont plus de signification.

Le fonctionnement des blocs de mémorisation est analogue à la définition de variables locales utilisée dans les langages de programmation structurée style PASCAL. Ils peuvent donc être imbriqués les uns dans les autres (jusqu'à seize niveaux d'imbrication sont autorisés), permettant ainsi une programmation modulaire et structurée. Ainsi, un sous programme effectuant un traitement temporaire, peut définir ses propres attributs et objets, les afficher et restituer en sortie la banque dans l'état précédent son appel.

```
sousprog1(...)
{
   Bloc();
      .....
      /* définition et affichage d'attributs et d'objets propres à sousprog1 */
      .....
   FinBloc();
}
```

remarque : les primitives de définition d'objet d'interface préfixées par *Def_* (c.f. chapitre VII) provoquent automatiquement une mémorisation quel que soit le contexte d'écriture dans lequel elles sont exécutées. Cependant, la durée de vie de ces objets correspond au bloc de mémorisation dans lequel ils sont créés. La fin du Bloc, entraine la libération de l'espace qui leur est alloué de la même manière que pour les attributs et objets graphiques.

2.3 Accès aux objets mémorisés - références

Afin de permettre de réutiliser par la suite les objets et attributs mémorisés, toutes les primitives de définition retournent une **référence** vers ceux-ci. Ces références doiventt être conservée au niveau du programme d'application. (La référence d'une entité (attribut élémentaire ou objet) est en fait un entier redéfini par le type *refer* qui représente une adresse dans la banque de données graphique).

```
refer coul1,coul2,ligne,ligne2;
....
DebEcr(Banque);
   ...
   coul1 = Couleur(....);
   ligne1 = Ligne2D(....);
   ...
   coul2 = Couleur(....);
   ligne2 = Ligne2D(....);
   ...
FinEcr();
```

Les références sont le mécanisme uniforme pour accéder aux entités enregistrées dans la banque de données du logiciel graphique (objets d'interface, attributs graphiques, objets élémentaires ou structurés). Cependant en ce qui concerne les objets structurés, un mécanisme d'accès à base de noms et d'expressions de noms analogue à celui employé dans CLOVIS est également proposé comme nous le verrons plus en détail au paragraphe 3.2.4.

3 ATTRIBUTION

L'attribution peut prendre des formes multiples, et les mêmes primitives peuvent donner lieu à des effets différents selon leur contexte d'utilisation (simple visualisation, simple mémorisation, combinaison des deux, structuration....). Aussi, dans un premier temps, dans un souci de clarté et de simplicité, nous nous contenterons de présenter la définition d'objets et d'attributs élémentaires sans aucune structuration. Ces objets et attributs, définis en dehors de toute structure, sont désignés sous le terme d'objets et d'attributs élémentaires **libres**.

Après avoir passé en revue les différents primitives de définition et d'objets et d'attributs élémentaires et avoir étudié leur comportement selon la nature du contexte d'écriture, nous étudierons les mécanismes offerts à l'application pour la structuration des objets graphiques.

Nous présenterons ensuite la manière de réutiliser les attributs élémentaires et objets mémorisés afin de construire ou modifier des objets (structurés), ou bien directement pour l'affichage. Nous verrons qu'il est également possible de modifier l'information qui caractérise les attributs et objets élémentaires.

3.1 Définition d'objets et d'attributs élémentaires libres

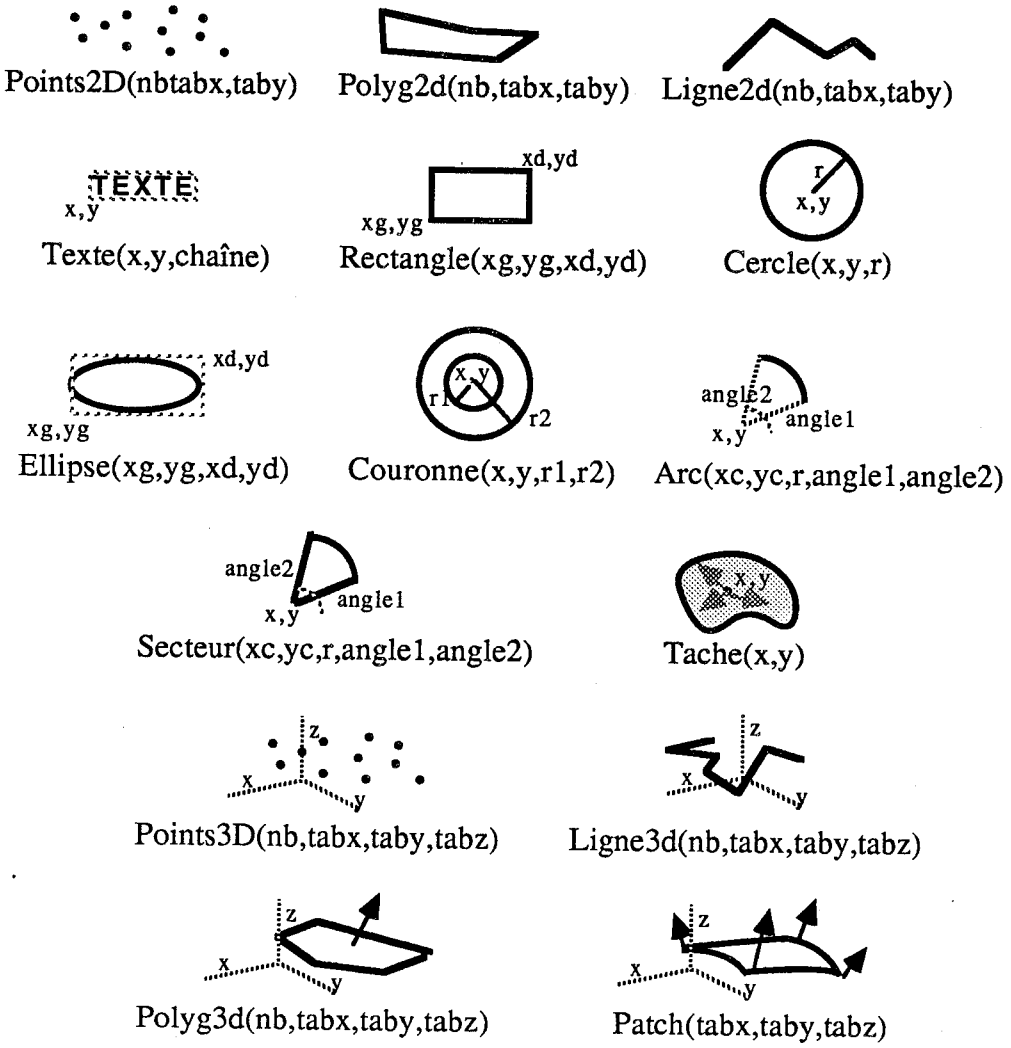
3.1.1 Définition d'objets élémentaires - Attributs Morphologiques

Les attributs morphologiques sont définis à partir d'un certain nombre de primitives qui construisent des objets de base ou objets élémentaires. La figure VIII.2, page 302, donne une liste de celles-ci, elles reprennent dans une large mesure celles que nous avions développées pour CLOVIS.

Lorsque des objets élémentaires sont définis à l'interieur d'un contexte de visualisation, ils sont **immédiatement** affichés avec les derniers attributs d'Aspect, de Géométrie de Visibilité spécifiés (voir paragraphe suivant), et en l'absence de toute spécification d'attribut, avec les attributs par défaut attachés au contexte.

Dans un contexte de mémorisation, les primitives de définition de morphologies élémentaires renvoient la référence de l'objet créé que l'application peut conserver.

```
refer  ref1; /* variable référence pour conserver l'adresse d'un objet élémentaire */
....
DebEcr(refvue);
   Cercle2D(0,0,20); /* affiche un cercle de centre 0,0 et de rayon 20 */
   ...
   DebEcr(banque);
      ...
      ref1 = Poly2D(n,tx,ty); /* mémorisation d'un polygône 2d */
      ...
   FinEcr()
   ...
FinEcr()
```



- figure VIII.2 : primitives de définition de morphologies élémentaires -

3.1.2 Définition d'attributs élémentaires - Couleur, Aspect, Géométrie, Visibilité, Reflexion.

Les attributs élémentaires associés aux objets graphiques de base pour définir leur présentation visuelle sont définis à l'aide de primitives de modélisation dont la liste est donnée dans la table VIII.2. Ces attributs sont très proches de ceux que nous avons utilisés pour CLOVIS, néanmoins quelques améliorations ont été apportées, en particulier en ce qui concerne la prise en compte de l'aspect et de la couleur.

Cependant avant d'étudier plus en détail la modélisation des différents attributs de chacune des classes présentées dans la table VIII.2, nous allons exposer la manière dont les attributs sont pris en compte au moment de leur définition selon que l'on se trouve dans un contexte de visualisation ou de mémorisation.

CLASSE	PRIMITIVES
COULEUR	Couleur(rouge,vert,bleu) ExpCoul(numexp,opérateur,ref_rouge,ref_vert,ref_bleu);
VISIBILITE	Visibilité(indic)
ASPECT	AspSurf(motif,numexp) AspLigne(motif,numexp,style) AspTexte(motif,numexp,police,taille,style) Aspect(ref_surf,ref_text,ref_texte)
REFLEXION	Reflex(num_modèle)
GEOMETRIE **Geom2D**	Geom2d(repère,m11,m12,m21,m22,m31,m32); Trans2D(repère,x,y) Rota2D(repère,x,y,angle) Homoth2d(repère,x,y,echx,echy)
Geom3D	Geom3d(repère,m11,m12,m13,m21,m22,m23,m31,m32,m33, m41,m42,m43); Trans3D(repère,x,y,z) Rota3D(repère,x1,y1,z1,x2,y2,z2,angle) Homoth3d(repère,x,y,z,echx,echy,echz)

- table VIII.2: les attributs graphiques élémentaires -

3.1.2.1 Prise en compte des attributs élémentaires libres

Si une primitive de définition d'attribut élémentaire est invoquée au cours d'un affichage immédiat (à l'intérieur d'un contexte de visualisation), l'attribut créé devient **attribut courant** du contexte de visualisation. Il s'applique alors à tous les objets libres définis par la suite dans ce contexte, ceci jusqu'à ce qu'il y ait une nouvelle définition d'un attribut de la même classe. Ainsi les morphologies élémentaires définies hors structure (objets élémentaires libres) sont affichées avec les derniers attributs spécifiés dans le contexte, ou sinon avec les attributs par défaut du contexte.

Ce mode de définition d'objets et d'attributs élémentaires à l'intérieur d'un contexte de visualisation, permet la construction immédiate d'images simples ne nécessitant plus de modifications ultérieures, ce qui n'était pas possible dans CLOVIS, le passage par la structure hiérarchique étant alors obligatoire. Il correspond à une définition **procédurale** des objets graphiques. On retrouve ainsi les fonctionnalités de bas niveau (segments non retenus) proposées par les logiciels de type GSPC ou GKS avec l'utilisation de primitives de sortie (morphologies élémentaires) et une spécification **modale** des attributs graphiques.

Comme pour les morphologies, les primitives de définition d'attributs élémentaires invoquées dans un contexte de mémorisation, retournent une référence vers l'attribut créé. L'application peut conserver cette référence et l'utiliser afin d'accéder par la suite à l'attribut. Si la définition

de l'attibut est effectuée dans un contexte de mémorisation simple, l'attribut créé est uniquement stocké dans la banque de données graphique et ne concerne en rien l'affichage. Il devra être réutilisé par la suite à l'aide d'une primitive d'attribution (voir § 3.2.5.1.3.1 et § 3.3.1). Dans un contexte de visualisation et mémorisation simultanées, l'attribut créé et mémorisé devient également attribut courant du contexte de visualisation. L'exemple ci-dessous présente l'effet que produit la définition d'un attribut (ici la couleur) selon le contexte dans lequel elle a lieu.

```
refer coul1,coul2,coul3,ligne1,ligne2,ligne3;

....
DebEcr(Banque+refvue); /* mémorisation plus affichage immédiat */
   Cercle(0,0,20); /* affiche un cercle avec la couleur par défaut */
   coul1 = Couleur(255,0,0); /* rouge devient la couleur courante du contexte */
   ...
   ligne1 = Ligne2D(....); /* affiche une ligne 2D avec la couleur rouge */
   ...
   DebEcr(Banque); /* mémorisation simple */
      coul2 = Couleur(0,0,255);  /* définition d'une couleur bleue */
      ....
   FinEcr();
   ....
   ligne2 = Ligne2D(....); /* affiche une ligne 2D avec la couleur rouge */
   coul3 = Couleur(0,255,0); /* vert devient la couleur courante du contexte */
   ...
   ligne3 = Ligne2D(....); /* affiche une ligne 2D avec la couleur vert */
   ...
FinEcr();
```

3.1.2.2 Modélisation des attributs des différentes classes

3.1.2.2.1 COULEUR

La couleur avec laquelle les objets sont visualisés est spécifiée par deux paramètres :

- une couleur de base ou **couleur courante**, attribut défini à l'aide de la primitive **Couleur(R,V,B)** , dans laquelle on indique ses composantes primaires Rouge, Vert et Bleu (entiers compris entre 0 et 255).

- une "expression de couleur" référencée dans l'attribut d'aspect courant (voir paragraphe suivant), et qui indique la manière dont cette couleur de base est ensuite utilisée pour l'affichage des objets. Les expressions de couleurs permettent de modifier, moduler, compléter ou remplacer la couleur courante (couleur de base) à l'aide d'une seconde couleur (**couleur de référence**).

Jusqu'à 95 expressions de couleurs peuvent être définies, ceci à l'aide de la primitive

ExpCoul(numexp,ref_rouge,ref_vert,ref_bleu,opérateur)

où :

- ***numexp*** est un entier qui désigne l'expression de couleur à définir ou redéfinir,

- ***ref_rouge***, ***ref_vert***, ***ref_bleu*** sont les composantes de la couleur de référence (entiers compris entre 0 et 255),

- ***opérateur*** est l'adresse d'une fonction effectuant le traitement de la couleur de base (couleur courante) à l'aide de la couleur de référence. Des fonctions standards sont proposées à l'opérateur :

- ***C_Cour*** qui utilise directement, sans la modifier, la couleur courante,
- ***C_Ref*** qui utilise directement la couleur de référence au lieu de la couleur courante,
- ***C_Degr*** qui effectue un dégradé de couleurs entre la couleur courante et la couleur de référence (interpolation linéaire entre les composante R,V,B des deux couleurs),
- ***C_Modu*** qui module (multiplie) la couleur courante à l'aide de la couleur de référence.

3.1.2.2.2 ASPECT

Comme dans CLOVIS, il est possible de définir séparément des aspects différents pour chaque type de morphologie ligne, surface ou texte :

- l'**aspect de ligne**, qui défini l'apparence des morphologies "fil de fer" (Ligne2d,points...) et des contours des morphologies surfaciques (Rectangle,Couronne....).

- l'**aspect de surface**, qui défini l'apparence des morphologies délimitant une portion de surface (Rectangle,Polyg2d...).

-l'**aspect du texte**, qui décrit l'apparence des morphologies texte.

Ces différents attributs d'aspect élémentaires sont construits et affectés (dans le cas par exemple d'une visualisation immédiate) à l'aide des primitives :

- **Aspligne(style,motif,numexp)** pour l'aspect des lignes :

-***style*** indique si les traits doivent être tracés aux crayon (épaisseur de 1 pixel) ou au pinceau (l'épaisseur du trait est alors définie par le motif de tracé),

- ***motif*** est un numéro de motif de "crayon" ou de "pinceau" avec lequel les lignes et les points sont tracés. Pour un style de tracé au "crayon", le motif indique la texture du trait (continu, pointillé, mixte...). Dans le cas d'un tracé au "pinceau," le motif indique la forme et la taille du "pinceau". Un motif nul (*motif* = 0) signifie que le tracé ne doit pas être effectué,

-***numexp*** est le numéro de l'expression de couleur à utiliser pour déterminer la couleur du trait (*numexp* = ***Neant***, indique que l'on utilise directement la couleur courante).

- **Aspsurf(motif,numexp,police,taille,style)** pour l'aspect des surfaces :

-***motif*** est le numéro de trame (motif de remplissage) avec laquelle la surface doit être remplie. Un motif nul (*motif* = 0) indique que le remplissage ne doit pas être effectué, le motif ***Uni*** (*motif* = 1) signifie un remplissage uniforme,

- ***numexp*** est le numéro de l'expression de couleur à utiliser pour déterminer la couleur de remplissage des objets (*numexp*= ***Neant***, indique que l'on utilise la couleur courante).

- **Asptexte (style,motif,numexp)** pour l'aspect du texte :

- ***motif*** est un numéro de trame pour le remplissage des caractères identique à celui utilisé pour l'aspect des surfaces,

-***numexp*** , le numéro de l'expression de couleur à utiliser pour le remplissage des caractères,

-***police*** , le numéro de la police de caractères à utiliser,

- ***taille*** , facteur de zoom à appliquer aux caractères,

- ***style*** indique un mode de génération des caractères (orientation, italique, gras .....).

Les motifs de tracé (crayon) et de remplissage, ainsi que les polices de caractères, sont définis dans des fichiers "ressources", chargés à l'initialisation du système. Ces fichiers peuvent

être changés au cours de l'exécution du programme, permettant à l'application d'accéder à d'autres motifs ou polices.

La primitive

Aspect(refligne,refsurf,reftexte)

permet d'affecter globalement les trois types d'aspect élémentaires. Les aspects élémentaires utilisés sont désignés à l'aide de leur référence et doivent avoir été définis (et mémorisés) précédemment à l'aide des primitives correspondantes.

A la place d'une référence vers un attribut d'aspect élémentaire, il est possible d'utiliser les constantes prédéfinies :

- ***Absent***, qui signifie que l'aspect précédemment défini est conservé pour les morphologies de type correspondant,

- ***Neant***, qui signifie que les morphologies correspondantes ne doivent pas être affichées.

3.1.2.2.3 REFLEXION

L'attribut de réflexion permet d'associer aux objets tridimensionnels un "modèle de reflexion" qui exprime la manière dont il réagissent aux paramètres d'éclairage (matériau brillant, mat...). Rappelons que GETRIS permet de gérer simultanément 16 modèles de reflexion à l'aide d'opérateurs cablés assurant des modifications temps réel. Lors de l'initilisation de la machine en configuration 3D, ces 16 modèles sont chargés en mémoire à partir d'un fichier ressources. Le choix d'un modèle de reflexion est ensuite exprimé à l'aide d'un entier (0..15).

3.1.2.2.4 VISIBILITE

Dans certaines configurations matérielles, grâce à des tables de visibilité cablées, l'utilisateur peut rendre instantanément visibles ou invisibles des objets déjà affichés. A cet effet un attribut de visibilité peut être associé aux objets (il a surtout une signification pour les objets structurés mémorisés que nous verrons au § 3.2).

3.1.2.2.5 GEOMETRIE

Deux sous classes d'attributs géométriques sont considérées : **GEOM2D** destinée aux objets bidimensionnels et **GEOM3D** pour les objets tridimensionnels. Les attributs de ces deux classes sont définis séparément et peuvent être utilisés simultanément.

Les attributs géométriques définissent des transformations géométriques appliquées, au moment de l'affichage, aux coordonnées décrivant la morphologie des objets. Pour le calcul de ces transformations géométriques on utilise les coordonnées homogènes. Comme dans CLOVIS, des primitives permettent de construire les transformations standards rotation, translation, homothétie. Cependant, l'utilisateur peut si il le désire spécifier une transformation quelconque en fournissant les coefficients de sa matrice en coordonnées homogènes.

Le paramètre ***repère*** utilisé dans les primitives de définition d'attributs géométriques indique dans quel repère la transformation est exprimée, trois constantes prédéfinies sont destinées à cet effet :

- ***Abs***: la transformation est exprimée dans le repère de la vue (transformation absolue),

- ***Rel*** : la transformation est relative au repère courant issu des transformations précédentes (composition des transformations),

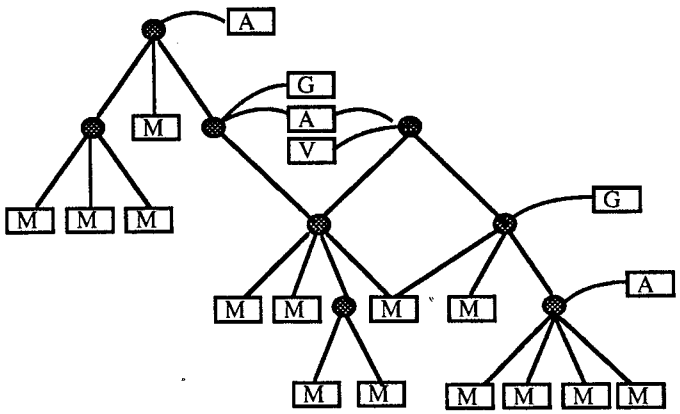
- ***Rloc*** : transformation locale au repère du niveau supérieur de la structure (composition des matrices). Cette option est utilisable uniquement dans le cas d'objets structurés (voir § 3.2.3.1).

3.2 Definition d'objets structurés - STRUCTURE - IDENTITE

3.2.1 La structuration des attributs graphiques - notion de scène

Ce second système autorise, de façon analogue à CLOVIS, la structuration hiérarchique des attributs graphiques élémentaires afin de définir des objets complexes. L'entité de base pour la structuration des attributs graphiques est la **scène** définie comme suit :

Une **scène** (figure VIII.3) est un objet structuré qui permet de regrouper un ensemble d'objets "homogènes", c'est à dire associés aux mêmes attributs d'Aspect, Couleur, Géométrie, Visibilité et de Reflexion. Les objets qui composent la scène peuvent être eux-mêmes des objets structurés (scènes) ou bien des objets élémentaires (morphologies élémentaires).



● Objet structuré (scène)

[M] Attribut élémentaire (M : Morphologie, A : Aspect, G : Géométrie, V : visibilité)

- figure VIII.3 : La structuration des attributs graphiques -

De cette définition, découlent plusieurs différences fondamentales avec CLOVIS qu'il est important de souligner :

- **le retrait de la structure hiérarchique des attributs globaux Ga, Gv et E** : seuls les attributs des classes M, G et A (qui regroupe les sous-classes Aspect proprement dit, Visibilité et Eclairage) sont conservés au niveau des scènes.

Ce choix correspond au souci de séparer la définition des objets graphiques de la définition de leur présentation. En effet, les attributs des classes Gv, Ga et E s'appliquent de façon globale à un ensemble d'objets (scène) et permettent d'en définir une image (ils décrivent les conditions d'observation de la scène). Ces attributs sont donc plus naturellement associés à des objets d'interface (les vues) particuliers, destinés à la présentation d'objets graphiques et définis **indépendament** par l'application. La visualisation d'objets graphiques s'effectue alors en les associant à un ou plusieurs de ces objets d'interface. (Dans CLOVIS, le traitement des attributs Ga, Gv et E de manière identique aux attributs M, A et G, si il offrait une "belle" homogénéité posait parfois certains problèmes. Par exemple, la présentation simultanée de plusieurs vues différentes d'un même groupe d'objets (scène) nécessitait plusieurs manipulations (redéfinition des attributs Ga,Gv et E, revisualisation de la structure de la scène) qui à l'usage se sont révélées lourdes et inappropriées).

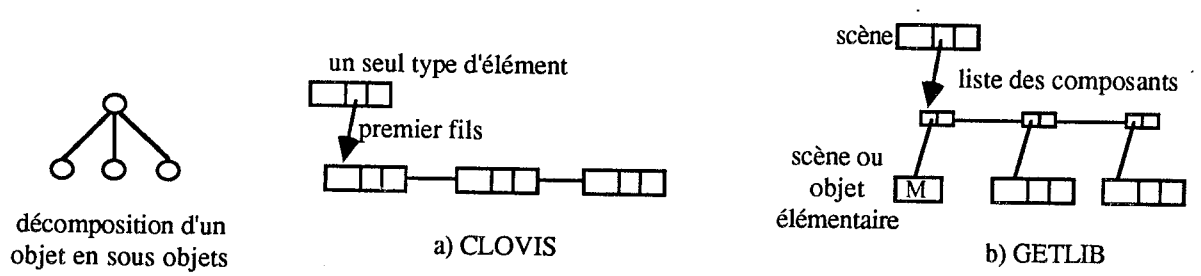
- **le traitement particulier des attributs morphologiques** : à un élément de la structure hiérarchique des objets graphiques (scène) peut être associé un nombre quelconque de morphologies élémentaires alors qu'il ne peut posséder qu'un seul type d'attribut à l'intérieur des autres classes. (Rappelons que dans CLOVIS, au plus un seul type d'attribut par classe pouvait être associé à un noeud de l'arborescence, morphologie y compris). Ceci permet de définir simplement des objets complexes à partir de plusieurs morphologies élémentaires sans avoir à multiplier le nombre d'éléments dans la structure hiérarchique.

Les morphologies élémentaires sont considérées comme des objets de manière analogue aux scènes (objets structurés), à la seule différence qu'il n'est pas possible de leur associer directement des attributs. Ainsi, comme nous le verrons par la suite, les mêmes primitives permettent de traiter indifféremment les objets élémentaires et les objets structurés. Les morphologies élémentaires définissent des objets manipulables et visualisables indépendamment des autres. En particulier, il est ainsi permis, comme nous l'avons vu au paragraphe 3.1, de définir et d'afficher des objets élémentaires sans avoir à passer par la structure hiérarchique, ce qui dans CLOVIS était impossible. L'intérêt et l'avantage de cette solution est évident pour la construction d'images simples ne necessitant pas ou peu de modifications.

- le troisième point, lié au précédent, concerne les **possibilités de partage d'objets** entre plusieurs scènes beaucoup plus variées que dans CLOVIS. Nous le développons en détail dans le paragraphe qui suit.

3.2.2 Le Partage d'Objets

Dans ce nouveau système, la représentation de la structure hiérarchique est quelque peu différente de celle que nous avions adoptée pour CLOVIS : la décomposition d'une scène n'est plus représentée par la liste des objets qu'elle regroupe (fig. VIII.4 a), mais par la liste des **adresses** (références) de ces objets (fig. VIII.4 b).



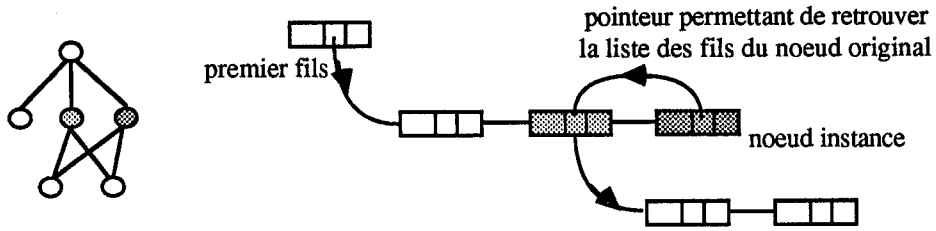
- figure VIII.4 : représentation de la structure hiérarchique -

Outre le fait qu'elle permet de traiter les attributs morphologiques comme des objets à part entière (c.f. § précédent), cette représentation de la structure hiérarchique offre une beaucoup plus grande souplesse que celle utilisée pour CLOVIS, en particulier en ce qui concerne le **partage** d'objets entre différents éléments de la structure.

Dans CLOVIS, un objet ne pouvait être directement réutilisé (dans une même structure ou dans plusieurs structures différentes) : en effet, de par le type de chaînage employé (premier fils, frère précédent, frère suivant), un élément ne saurait apparaitre plusieurs fois dans la liste des descendants d'un noeud ou appartenir simultanément aux listes des descendants de noeuds différents.

Cependant, nous avions introduit un mécanisme, l'**instanciation**, autorisant le partage d'une même "sous-structure" par des éléments différents et évitant ainsi la duplication d'objets ayant une décomposition identique. Ainsi, pour partager la sous structure d'un noeud (noeud original) un noeud de type particulier (noeud instance) est créé. La liste des descendants des noeuds instances est accédée **indirectement** en passant par le noeud orignal : elle est ainsi identique à celle de ce dernier (fig. VIII.5). Si cette sous-structure est la même pour tous les noeuds qui la partagent (en particulier toutes les modifications qui lui sont apportées

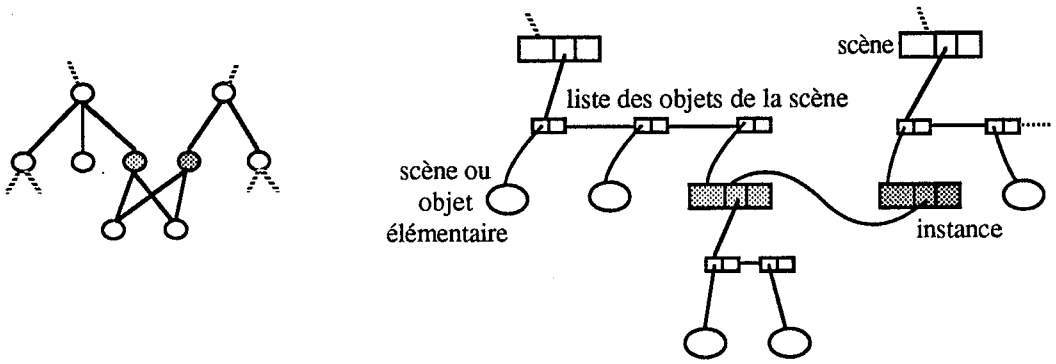
s'appliquent de manière identique pour ceux-ci), ses différentes occurences peuvent être néanmoins caractérisées au moment de l'affichage par les attributs évalués au niveau des noeuds instances. (possibilité de particulariser les différentes instances de la sous structure en affectant des attributs propres aux noeuds instances).



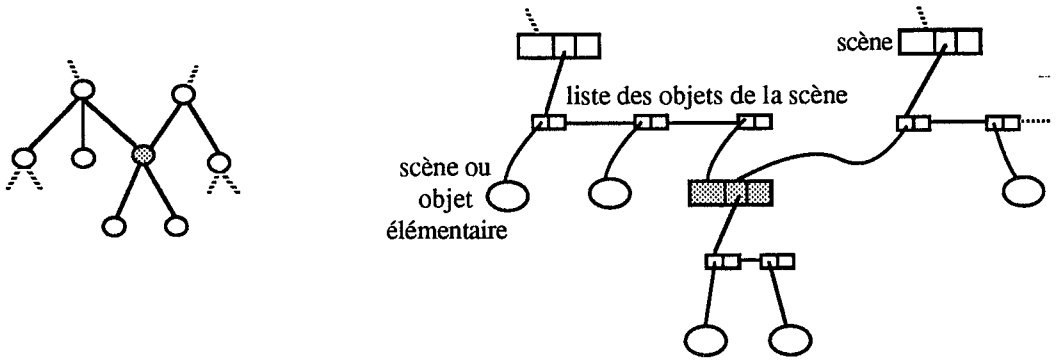
- figure VIII.5 : partage de sous structures dans CLOVIS -

Dans ce second système, nous avons conservé la notion d'instance introduite dans CLOVIS, qui permet à des **scènes** différentes de partager la même sous structure (c'est à dire la même liste de sous objets) (fig. VIII.6 a).

Cependant le mode de représentation de la structure hiérarchique offre une plus grande richesse. Il est ainsi possible de partager **directement** des objets **dans leur intégralité** (et non plus uniquement leur sous-structure) entre plusieurs scènes différentes (fig. VIII.6 b) et/ou de réutiliser le même objet à l'intérieur d'une même scène. Au moment de la visualisation, les différentes occurences du même objet se distinguent uniquement par les attributs qu'elles héritent des scènes de niveau hiérarchique supérieur.



a) partage de la même sous structure par deux scènes



b) partage d'un même objet par deux scènes

- figure VIII.6 : Partage d'objets -

Cette seconde approche permet de partager très simplement des objets quelconques (élémentaires ou structurés). Elle ne nécessite pas de traitement particulier (au niveau du parcours de la structure pour l'affichage, rien ne distingue un objet ainsi partagé des autres objets) et évite la création de noeuds intermédiaires (noeuds instances), qui introduit un niveau de structuration supplémentaire inutile quand les objets à partager n'ont pas d'attributs propres. Par ailleurs des scènes différentes peuvent posséder des objets en commun sans pour autant avoir des sous structures totalement identiques (fig. VIII.6 b), chose qui n'était pas possible dans CLOVIS (à moins d'introduire des niveaux d'instances supplémentaires).

3.2.3 Composition des attributs graphiques

3.2.3.1 Règles de composition des attributs élémentaires

Lors de la visualisation d'un objet structuré, une composition des attributs associés aux différentes scènes qui le définissent, est effectuée durant le parcours de structure afin de déterminer les caractéristiques propres aux différents objets élémentaires rencontrés et affichés. Ces règles de composition sont les suivantes :

- si pour une classe donnée aucun attribut n'est défini dans une scène, les objets qu'elle regroupe sont affichés avec l'attribut correspondant évalué au niveau d'imbrication supérieur. En l'absence de toute définition d'attribut dans la structure hiérarchique, c'est l'attribut courant du contexte de visualisation et en dernier ressort son attribut par défaut qui s'applique.

- quand un attribut est défini pour une scène, il s'applique à tous les objets définis dans le sous arbre dont elle est racine.

Pour les classes COULEUR, VISIBILITE et REFLEXION, l'attribut se substitue tout simplement à la valeur évaluée au niveau supérieur. Pour la classe ASPECT, cette substitution n'a lieu que lorsque la référence d'aspect est différente de ***Absent***.

Pour les classes GEOM2D et GEOM3D, si l'attribut a été défini avec la mention ***Rloc*** , il y a composition avec la transformation géométrique définie au niveau supérieur (composition par produit matriciel en coordonnées homogènes). Cette composition des géométries permet d'associer des repères propres aux objets et d'exprimer leur position par rapport au repère de niveau supérieur à l'aide de transformations géométriques "locales". Si l'attribut géométrique est une géométrie absolue (mention ***Abs***), il n'y a aucune composition et les mêmes règles de substitution que pour les autres attributs sont appliquées.

2.2.3.2 Les attributs groupés

De par les règles de composition des attributs graphiques présentées dans le paragraphe précédent, si l'on désire particulariser certains objets à l'intérieur d'une scène il est nécessaire d'introduire des niveaux de structure supplémentaires auxquels sont associés les attributs correspondant.

Pour éviter cette décomposition supplémentaire, qui parfois peut entrainer une certaine lourdeur, il est possible d'associer à une scène non pas un, mais un ensemble d'attributs élémentaires d'une même classe, qui s'appliquent séparément et individuellement aux objets qu'elle regroupe.

L'affectation à une scène d'un ensemble d'attributs d'une même classe s'effectue par l'intermédiaire d'un **attribut groupé**. Un tel attribut est créé à l'aide de deux primitives (l'une de création d'un groupe d'attributs d'une classe donnée, l'autre de terminaison de groupe) qui "encadrent" des définitions d'attributs élémentaires. Un groupe d'attributs rassemble tous les attributs élémentaires de la classe correspondante, définis suite à l'appel de sa primitive de déclaration et avant l'appel de la primitive de fin de groupe.

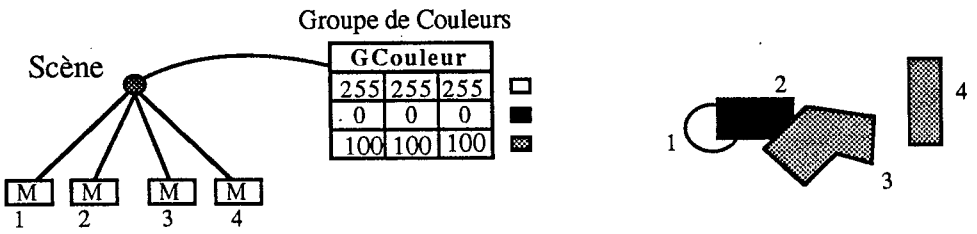
```
GCouleur(3); /* définition d'un groupe de 3 couleurs */
      Couleur(255,0,0);  /* rouge */
      Couleur(0,255,0);  /* vert */
      Couleur(0,0,255);  /* bleu */
FinG();      /* fin du groupe */
```

L'exemple ci-dessus, présente la définition d'un groupe de trois attributs de couleur. De façon analogue, à chacune des primitives de définition d'un attribut simple , correspond une primitive de création d'un groupe d'attributs (primitive de définition de l'attribut élémentaire préfixée par G).

Lors de la définition d'un groupe d'attributs, l'utilisateur indique le nombre maximum d'attributs le composant. Si des attributs sont définis en surnombre avant la fin du groupe (*FinG()*) , ils sont tout simplement ignorés.

Quand au moment de la visualisation, un attribut groupé est évalué, les attributs élémentaires qui le composent sont associés séparémment et successivement aux objets affichés par la suite : le premier attribut pour le premier objet, le second pour le deuxième.... Le dernier attribut d'un groupe s'applique à l'objet de rang correspondant et à tous ceux qui suivent jusqu'à la définition d'un nouvel attribut. Au contraire, si le nombre d'attributs du groupe est supérieur au nombre d'objets présents au moment de l'affichage, les attributs excédentaires sont ignorés.

La figure VIII.7 ci-dessous, présente un exemple d'utilisation d'un attribut de couleur groupé affecté à une scène.



- figure VIII.7 : utilisation d'attributs groupés -

La scène regroupe quatre objets élémentaires, un cercle, un rectangle un polygône et un rectangle. Le premier objet de la scène est affiché avec la couleur blanche, le deuxième en noir et les suivant en gris.

Si l'utilisation d'attributs groupés est surtout intéressante pour la construction d'objets structurés, leur définition peut, comme pour les attributs élémentaires et avec les mêmes règles, intervenir à tout moment (dans un contexte de visualisation et/ou mémorisation et/ou de création de structure (voir § 3.2.5.1.1). Afin de pouvoir réutiliser les attributs groupés mémorisés, une référence permettant de les désigner est retournée par leur primitive de création.

3.2.4 Identification des objets

3.2.4.1 Nomination des objets

Comme pour les attributs et objets élémentaires , les objets structurés (scènes) peuvent être directement accédés par leur référence (addresse vers leur descripteur dans la banque de données graphique) retournée au moment de leur construction. Cependant, si ce mécanisme est simple et permet de traiter de manière uniforme tous les éléments mémorisés, il présente néanmoins quelques inconvénients en ce qui concerne les objets structurés :

- la mémorisation et la gestion des références est à la charge de l'application,

- les références constituent un accès direct aux objets sans refléter la structuration qui existe entre ceux-ci,

- les références peuvent se révéler un moyen d'identification inadapté lors d'un dialogue interactif au cours duquel l'utilisateur est amené à désigner le ou les objets qu'il désire manipuler (particulièrement si les objets doivent être désignés autrement que par un dispositif de dialogue, ou bien si l'identification de l'objet doit être communiquée à l'opérateur).

Ces remarques font ressortir la nécessité d'offrir à l'utilisateur, en plus des références, un moyen d'identifier les objets qu'il crée et d'acceder ensuite facilement à ceux-ci. Comme nous l'avons vu pour le système CLOVIS , nous pensons que le moyen le plus adapté est à base d'identificateurs **alphanumériques** associés à chaque objet qui combinés entre eux dans des **expressions de référence**, permettent de désigner des éléments de la structure hiérarchique.

Cependant, si nous reprenons dans une large mesure les principes énoncés pour CLOVIS , le mécanisme de désignation des objets à l'aide de noms alphanumériques a été adapté à ce nouveau système en le rendant beaucoup moins contraignant :

- Ainsi, de la même manière qu'il peut ou non conserver les références des objets et attributs, l'utilisateur est **libre** de nommer les objets qu'il crée. Cette liberté laissée à l'application, offre la possibilité de nommer uniquement les parties de structure sur lesquelles on agit par la suite, sans se soucier de donner un nom aux éléments auquels on n'accédera plus.

- Alors que dans CLOVIS, la nomination des objets ne pouvait s'effectuer qu'au moment de la construction d'une structure par l'intermédiaire d'une expression de création, ce second système s'il offre toujours cette possibilité (§ 3.2.5.2) permet par ailleurs de nommer à tout moment un objet dont on possède la référence. Grâce à ce second mode, nous verrons qu'il est possible de nommer les scènes et objets lors d'une construction progressive de structure ainsi que de modifier l'identification d'un noeud (§ 3.2.5.1).

- Aucune contrainte n'est plus imposée sur les noms des éléments. Alors que dans CLOVIS, l'identification d'un élément devait être sans ambiguité, c'est à dire qu'un nom devait être unique à un niveau de structure, des objets différents peuvent éventuellement partager le même nom. Si ces objets se trouvent à un même niveau de structure, un indice en plus de leur nom permet de les distinguer. Cette possibilité, outre qu'elle simplifie amplement la gestion des identificateurs*, offre une plus grande souplesse à l'application (pourquoi ne pas nommer de façon identique des objets différents mais qui ont la même fonctionnalité ?) même si elle accroit ses "responsabilités".

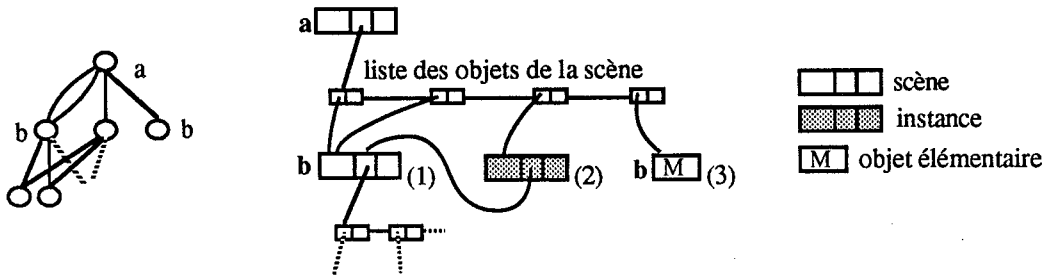
** un traitement pour distinguer les homonynes à un même niveau est lourd, et l'on peut supposer que ce problème intervient suffissament rarement pour laisser sa gestion éventuelle à l'application et pour ne pas pénaliser l'efficatié de l'identification des noeuds par sa prise en compte par le système graphique.*

3.2.4.2 Primitives d'accès aux objets structurés

Comme dans CLOVIS, une expression de référence simple permet de désigner **un** élément de structure existant à l'aide de son nom et de celui des scènes de niveau supérieur auxquelles il est rattaché. La syntaxe des expressions de référence simple est la suivante :

< référence simple > ::= < référence absolue > | < référence relative >
< référence absolue > ::= "/" < identificateur > {"[" < indice > "]"}{"/" < référence relative >}
< référence relative > ::= < identificateur > {"[" < indice > "]"}{"/" < référence relative >}
| < rang > { "/" < référence relative > }
< rang > ::= < entier non signé >
< indice > ::= < entier non signé >
< identificateur > ::= < lettre > { < lettre > | < chiffre > } *
< entier non signé > ::= < chiffre non nul > { < chiffre > } *

L'utilisation d'indices permet de distinguer les éléments d'un même niveau qui partagent le même identificateur : *nom_obj[i]* indique le $i^{ème}$ objet de nom *nom_obj* au niveau de structure considéré.



a) a/b[1] et a/b[2] désignent le même objet (1)
b) a/b[3] désigne l'objet (2) instance de l'objet (1)
c) a/b[4] désigne l'objet (3) de même nom que l'objet (1)
d) a/b désigne l'objet (1)

- figure VIII.8 : utilisation de références simples indicées -

Mais, alors que dans CLOVIS il s'agissait uniquement de désigner les différentes instances d'une même sous structure, le mécanisme d'indiçage recouvre ici de manière uniforme tous les cas possibles "d'homonymie" entre éléments d'un même niveau, à savoir :

- l'utilisation directe de plusieurs références à un même objet à l'intérieur d'une scène (fig. VIII.8.a),

- le partage indirect de la sous structure d'une scène à l'aide de noeuds intermédiaires de type instance qui, comme avec CLOVIS, possèdent le même identificateur que le noeud original racine de la sous-structure partagée (fig VIII.8.b),

- la possibilité laissée à l'utilisateur d'employer le même identificateur pour désigner des objets différents (fig. VIII.8.c)

Dans le cas où un identificateur apparait plusieurs fois au même niveau de structure et en l'absence d'indice, l'objet considéré est le premier dont le nom correspond (fig. VIII.8.d).

Il est également possible de désigner les objets d'une branche, non par leur nom mais par leur rang dans la structure : le premier fils d'une scène étant l'objet de rang 1, le second l'objet de rang 2, etc.... Ainsi sur l'exemple de la figure VIII.8, ci-dessus, la référence *a/4* désigne l'objet (3).

L'emploi de son rang pour désigner un objet peut s'avérer utile, par exemple dans le cas où tous les objets n'ont pas été nommés.

(remarque : dans le cas d'objets référencés directement plusieurs fois par une ou des scènes, des expressions de référence distinctes peuvent désigner le même objet.)

Les expressions de référence peuvent être soit absolues, soit relatives à une scène servant de **contexte d'identification**. La syntaxe utilisée est analogue à celle employée par le système UNIX pour la définition de chemin dans une hiérarchie de répertoire.

Une référence absolue débute par un / et l'identificateur qui suit doit désigner un objet **libre**, c'est à dire un objet qui soit a été défini hors de toute structure, soit n'est plus référencé par aucune scène.

Pour une référence relative, l'identificateur qui débute éventuellement l'expression doit être celui de l'un des objets que regroupe la scène contexte d'identification. En l'absence de contexte

d'identification (par défaut à l'initialisation du système), cet identificateur doit comme pour les expressions de référence absolues, désigner un objet libre.

La définition des contextes d'identification, s'effectue de manière analogue à la définition de contextes dans CLOVIS (§ 3.4 du chapitre IV). Suite à l'appel de la primitive

Contexte_Idt(ref_simple)

l'objet désigné par la référence simple (absolue ou relative) passée en paramètre devient nouveau contexte d'identification courant. Les contextes d'identification sont imbriqués et la primitive

Fin_Contexte_Idt()

restitue le contexte d'identification précédent.

L'accès aux objets mémorisés et précédemment nommés peut être effectué par l'intermédiaire de deux primitives qui au travers d'expressions de référence simple, permettent de retrouver la référence d'un objet.

Ces deux primitives sont :

Ref_Identité(ref_simple) --> référence

qui retourne la référence de l'objet désigné par la chaîne de caractères *ref_simple* qui doit contenir une expression de référence simple. En cas d'erreur de syntaxe ou de la non existence d'un objet pour un identificateur donné, un message le signale à l'opérateur et une référence nulle est retournée.

Ref_Identité_Structure(ref_simple,ref_scène,ref_obj,nb_occ)

qui en plus de la référence *ref_obj* de l'objet désigné par l'expression de référence simple contenue dans la chaîne *ref_simple*, retourne :

- la référence *ref_scène* de la scène de niveau supérieur à laquelle l'objet *ref_obj* appartient,
- le nombre d'occurences, *nb_occ*, de l'objet *ref_obj* dans cette scène.

3.2.5 Création d'Objets Structurés

La définition d'objets structurés peut être réalisée de deux manières différentes :

- en construisant l'objet à partir d'une **expression de construction** définissant sa structure et l'identité des éléments qui le composent,

- en construisant l'objet de façon **progressive**, c'est à dire en mélant à la construction des éléments de sa structure hiérachique (scènes), la définition et l'affectation des attributs élémentaires qui les caractérisent.

3.2.5.1 Construction progressive de structure

3.2.5.1.1 Définition de scène - Scène courante

La définition d'un objet structuré peut s'effectuer comme suit :

```
Scene();
   ...
   /* les attributs et objets définis sont affectés directement à la scène */
   ...
Fin();
```

La primitive ***Scene()*** crée une **scène courante** à laquelle sont affectés **implicitement** tous les attributs et objets définis par la suite et ceci jusqu'à l'appel de la primitive **Fin()**.

Ces deux primitives ***Scene()*** et ***Fin()*** délimitent ce que nous pouvons appeler un "contexte d'attribution implicite" ou "**contexte de création de structure**" :

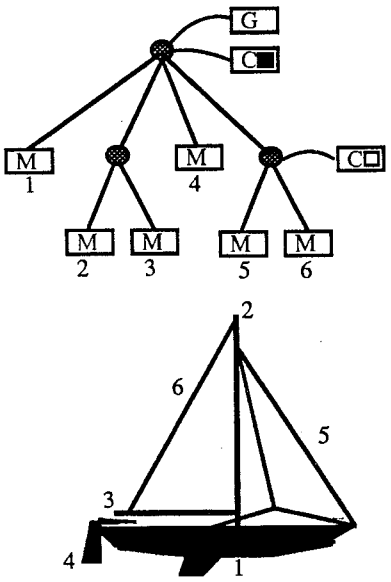
- la définition d'un nouvel attribut provoque son affectation à la scène courante. Si un attribut de même type à déjà été attaché à la scène courante, il est remplacé par le nouvel attribut.

- les nouveaux objets sont affectés à la scène courante dans l'ordre dans lequel ils sont définis. Ceux-ci peuvent être indifféremment des objets élémentaires ou des objets structurés (scène) : l'imbrication d'appels successifs des primitives ***Scene()*** et ***Fin()*** permet de définir une structuration hiérarchique des scènes (fig VIII.9).

En cas de mémorisation, la primitive ***Scène()*** retourne la référence de l'objet créé, qui peut être ainsi conservée par le programme d'application.

Lors d'une construction de structure dans un contexte de visualisation immédiate, les objets élémentaires sont affichés et affectés à la scène courante au fur et à mesure de leur définition. Les règles de composition définies précédemment (§ 3.2.3) sont appliquées pour l'évalution des attributs utilisés pour la visualisation de ces objets, ceci même en l'absence de mémorisation. Dans ce cas, les attributs définis à l'intérieur d'une scène sont évalués et conservés jusqu'à sa terminaison. Il y a empilement des attributs évalués en liaison avec l'imbrication des scènes, la fin d'une scène correspondant au dépilement des attributs afin de restituer comme attributs courants les attributs du niveau précédent.

```
Scene();
     Couleur(0,0,0);  /* noir */
    Geom2d(....);     /* position du bateau */
     Polyg2d(...);     /* la coque (1) */
    Scene();
       Rectangle(...); /* le mat (2) */
        Ligne2d(....);  /* la baume (3) */
   FinScene();
   Polyg2d(...);      /* le gouvernail */
   Scene();           /* les voiles */
     Couleur(255,255,255); /* blanc */
      Polyg2d(...);  /* le foc (5) */
     Polyg2d(...);   /* la grande voile (6) */
    FinScene()
FinScene()
```



● Objet structuré (scène)

[M] Attribut élémentaire (M : Morphologie, C : Couleur, G : Géométrie)

- figure VIII.9 : Construction progressive de structure -

3.2.5.1.2 Nomination des objets

La primitive

Nommer(ref_obj,identif)

permet de définir ou modifier le nom d'un objet mémorisé quelconque (objet structuré ou

élémentaire). *ref_obj* est la référence de l'objet à nommer, *identif* est une chaîne de caractères qui constitue l'identificateur à associer à cet objet. Aucune contrainte n'est imposée à cet identificateur qui peut éventuellement être utilisé pour d'autres objets, le seul contrôle effectué concerne sa syntaxe (< identificateur > ::= < lettre > { < lettre > | < chiffre > } *). En cas d'erreur de syntaxe dans l'identificateur, cette primitive est sans effet.

La primitive ***Nommer*** peut être appelée à tout moment dans le bloc de mémorisation de l'objet *ref_obj*.

3.2.5.1.3 Réutilisation d'attributs et d'objets mémorisés - Attribution Implicite

Bien entendu, des objets et attributs élémentaires précédemment définis et mémorisés peuvent être réutilisés lors d'une construction progressive de structure.

Les primitives qui permettent d'affecter un attribut élémentaire ou un objet mémorisé à la scène courante, constituent ce que nous appelerons des primitives d'**attribution implicite**. Ces primitives ne mentionnent que la référence de l'attribut ou de l'objet réutilisé. Comme les primitives de définition d'attributs ou d'objets élémentaires, elles ont un effet qui diffère selon leur contexte d'utilisation :

- dans un contexte de construction de scène l'attribut ou l'objet est affecté à la scène courante,

- dans un contexte de visualisation l'attribut ou l'objet spécifié est immédiatement pris en compte pour l'affichage (ceci même s'il s'agit d'un contexte de visualisation simple),

- dans un contexte de mémorisation simple, ces primitives sont sans effet.

3.2.5.1.3.1 Réutilisation d'attributs graphiques élémentaires

La réutilisation d'un attribut memorisé s'effectue par l'intermédiaire de la primitive

Attrib(ref_att)

ref_att est la référence de l'attribut méorisé qui peut appartenir à une classe quelconque (Couleur, Aspect, Géométrie, Visibilité) et peut être indifféremment simple ou groupé.

L'effet de cette primitive est identique à celui des primitives de définition d'attributs élémentaires :

- dans un contexte de visualisation simple , l'attribut *refatt* devient l'attribut courant pour la visualisation,

- dans un contexte de construction de scène, l'attribut *ref_att* est affecté à la scène courante. Si la construction de scène s'effectue dans un contexte de visualisation, l'attribut est immédiatement utilisé pour l'évaluation des objets ultérieurement définis dans la scène .

remarque : dans un contexte de mémorisation simple, sans visualisation ni construction de structure, l'utilisation de la primitive *Attrib* n'a pas de signification et est sans effet.

Exemple :

```
DebEcr(Banque);
   /* mémorisation des attributs et des objets */
   vert = Couleur(0,255,0); /*couleur verte*/
   ...
   arbre = Scene();
      ....
      feuilles = Scene()
```

```
            Attrib(vert); /* les objets de la scène feuilles auront une couleur verte */
            ....
        Fin(); /* feuilles */
        ...
    Fin(); /* arbre */
    ...
FinEcr(); /* contexte de mémorisation */
/* visualisation immédiate */
...
Couleur(255,0,0); /* couleur courante : rouge */
Ligne2D(....); /* affichage d'une ligne brisée  avec la couleur rouge */
Attrib(vert);
Cercle2D(...); /* affichage d'un cercle avec la couleur verte */
...
```

3.2.5.1.3.2 Réutilisation d'objets graphiques

a) référence directe à un objet existant

De manière similaire à *Attrib*, la primitive

Objet(ref_obj)

permet de réutiliser un objet graphique (élémentaire ou structuré) précedemment mémorisé :

- dans un contexte de visualisation simple, l'objet de référence *ref_obj*, est affiché en fonction des attributs courants du contexte.

- dans un contexte de construction de scène, l'objet *ref_obj* est affecté à la scène courante. Si la construction de scène s'effectue dans un contexte de visualisation, l'objet *ref_obj* est affiché avec les mêmes règles que pour la définition de nouveaux objets. Il est à souligner que la scène courante référence **directement** l'objet qui est ajouté à la liste de ses éléments.

Exemple:

```
DebEcr(Banque); /* mémorisation des attributs et des objets */
    pneu = Couronne(...);
    jante = Cercle(...);
    roue = Scene()
        Gcouleur(2);
            Couleur(100,100,100); /* gris pour la jante */
            Couleur(0,0,0); /* noir pour le pneu */
        FinG();
        Objet(jante);
        Objet(pneu);
    Fin();
    ...
    Ferrari = Scene();
        Couleur(255,0,0);
        Polyg2d(...);  /* la carrosserie */
        Scene(); /* les roues */
            GGeomd2D(2); /* position des roues relativement à la voiture */
                Trans2D(rloc,....); /* 1ère roue */
                Trans2D(rloc,....); /* 2ème roue */
            FinG();
            Objet(roue);
            Objet(roue);
        Fin();
    Fin();
```

```
...
FinEcr(); /* contexte de mémorisation */
...       /* visualisaton immédiate */
...
Couleur(0,255,0,0);
Ligne2D(....); /* affichage d'une ligne brisée  avec la couleur verte */
Objet(Ferrari); /* affichage de l'objet Ferrari */
...
...
```

remarque : dans un contexte de mémorisation simple, sans visualisation ni construction de structure, l'utilisation de la primitive *Objet* n'a pas de signification et est sans effet.

b) instance de scène

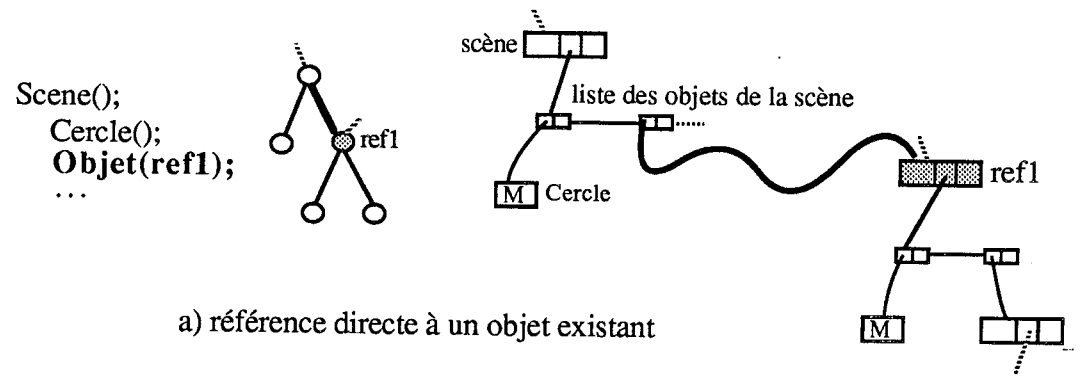
Si l'on ne désire pas réutiliser directement une scène mémorisée, mais uniquement sa sous structure, la primitive

Instance(ref_scene) --> refinst

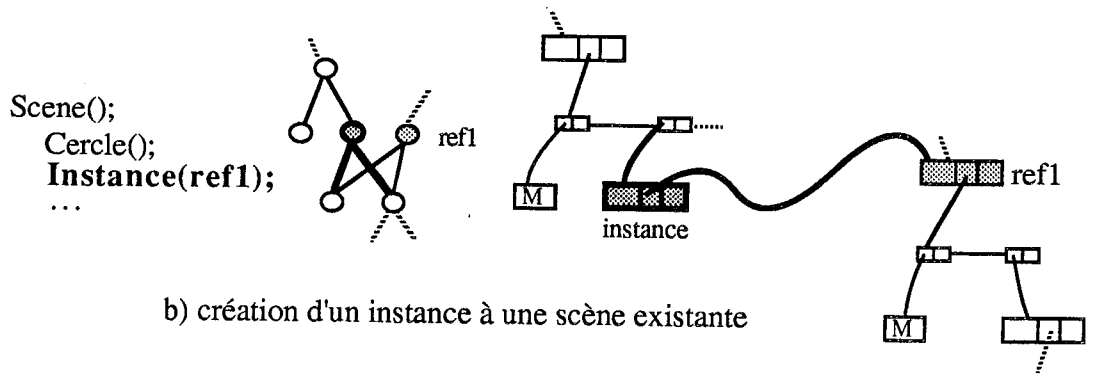
permet de créer un élément de type instance. La valeur retournée est la référence de cette instance (fig VIII.10). Cet objet partage la sous-structure de la scène originale indiquée par *ref_scène*, il possède initialement les mêmes attributs que celle-ci, mais ces derniers peuvent être par la suite modifiés pour particulariser l'instance.

Dans un contexte de visualisation simple, l'appel de ***Instance(ref_scene)*** est totalement équivalent à ***Objet(ref_scene)*** , la scène *ref_scène* est affichée, l'instance n'est pas mémorisée.

Dans un contexte de construction de structure, l'instance créée est ajoutée aux éléments de la structure courante. En cas de visualisation simultanée, elle est affichée avec les mêmes règles que pour les objets.



a) référence directe à un objet existant



b) création d'un instance à une scène existante

- figure VIII.10 : Création d'une référence directe et d'une instance à une scène mémorisée -

3.2.5.1.4 Intérêt de la construction progressive de structure

L'intérêt de la construction progressive de structure qui permet de créer une structure hiérarchique de scènes tout en affectant simultanément à celles-ci des attributs graphiques, est double :

- elle évite la lourdeur du mécanisme employé pour CLOVIS qui consiste à définir un objet structuré à partir d'une expression de création (cela crée une structure vide de tout attribut), puis de lui affecter, par définition successive d'entités de travail, des attributs graphiques élémentaires.

- elle permet de tirer profit de la puissance de la structuration hiérarchique des données graphiques en cas de visualisation immédiate simple ou doublée d'une mémorisation. Grâce à cette possibilité d'afficher des objets structurés au fur et à mesure de leur construction, on peut facilement envisager de décrire interactivement certains attributs de l'objet au cours de son élaboration et en s'appuyant sur la présentation partielle déjà affichée.

3.2.5.2 Construction de structure à partir d'expressions de noms

Comme dans CLOVIS, il est également possible de créer une structure à partir d'une expression définissant les noms de ses éléments. Ceci est effectué par l'intermédiaire de la primitive

Structure(expr_création) --> reference

qui construit la structure de scènes définie par l'expression de création contenue dans la chaîne de caractères *expr_création*. Si cette primitive est appelée à l'intérieur d'un contexte de construction de structure, la structure créée est ajoutée à la scène courante. En l'absence de scène courante, la structure créée est un objet libre. Quel que soit le contexte d'appel de ***Structure*** , la structure créée est mémorisée et la référence de sa scène racine est retournée.

La syntaxe des expressions de création de structure, inspirée de celle de CLOVIS, est la suivante :

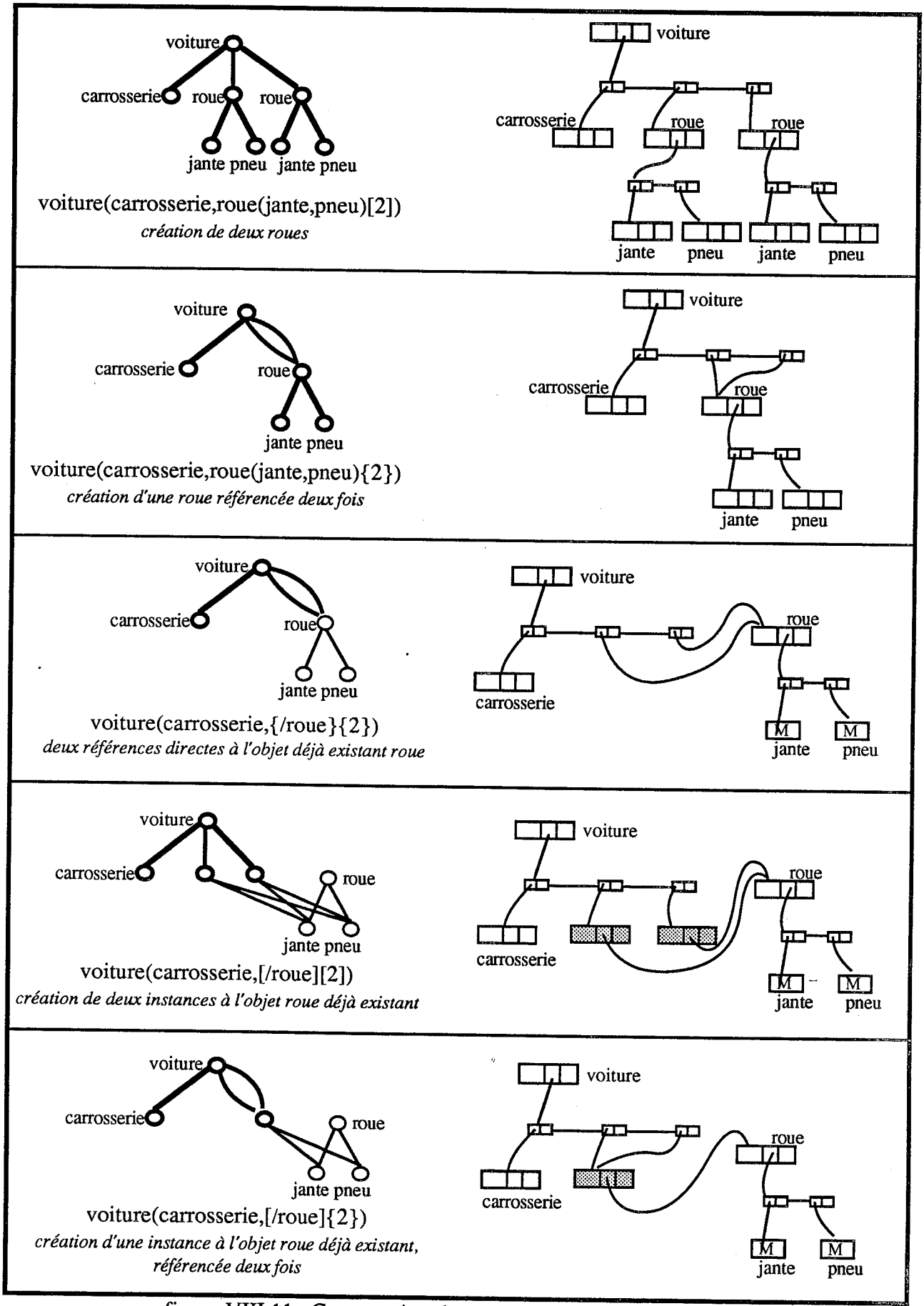
< expression de création > ::= < nom de scène > { "(" < sous-structure > ")" }
< sous-structure > ::= < structure > { < nbre d'exemplaires > | < nbre de références > }
{ "," < sous-structure > }
< structure > ::= < nouvelle structure > | < référence directe > | < instance >
< nouvelle structure > ::= < nom de scène > { "(" < sous-structure > ")" }
< référence directe > ::= "{" < référence simple > "}"
< instance > ::= "[" < référence simple > "]"
< nom de scène > ::= < identificateur > | "?"
< nbre d'exemplaires > ::= "[" < entier non signé > "]"
< nbre de références > ::= "{" < entier non signé > "}"
< identificateur > ::= < lettre > { < lettre > | < chiffre > } *
< entier non signé > ::= < chiffre non nul > { < chiffre > } *

Bien entendu, seule une structure de scènes peut être définiee à l'aide d'une telle expression, la création d'objets élémentaires à partir de simples identificateurs étant impossible.

Cependant, et il s'agit d'une nouveauté importante par rapport à CLOVIS, il est possible d'utiliser des objets déjà existants (élémentaires ou structurés) dans la définition de la nouvelle structure. Ceux-ci sont désignés à l'aide d'expressions de référence simple et peuvent être affectés à la scène créée au niveau supérieur soit :

- en les référençant directement (< référence directe >)

- soit en créant un noeud instance qui permet le partage de la sous structure de la scène désignée par l'expression de référence simple (< instance >).



- figure VIII.11 : Construction de structures à l'aide d'expressions -

D'autre part, il est possible de créer simultanément plusieurs exemplaires d'une même structure (< nbre d'exemplaires >) ou de créer une seule structure qui est référencée directement plusieurs fois (< nbre de références >) par la scène de niveau supérieur.

De plus, il n'est pas nécessaire de nommer toutes les scènes construites à l'aide de l'expression de création. L'emploi du ? permet de définir des scènes sans pour autant leur attribuer d'identificateur.

La figure VIII.11, ci-contre, résume les différentes possibilités offertes par les expressions de création de structure.

3.3 Réutilisation d'attributs et d'objets mémorisés - Attribution Explicite

Nous appelons primitives d'**attribution explicite**, des primitives qui permettent d'affecter un attribut ou objet à une scène mémorisée quelconque. Au contraire des primitives d'attribution implicite ***Attrib*** et ***Objet*** vues aux § 3.2.5.1.3, celles-ci mentionnent explicitement la scène concernée. Elles sont indépendantes de leur contexte d'utilisation (visualisation, mémorisation, création de structure) et n'ont en général pas d'effet sur l'image affichée. Les primitives d'attribution explicite sont très proches des primitives du processus d'Attribution tel que nous l'avions défini dans CLOVIS .

3.3.1 Affectation d'attributs élémentaires

L'association d'un attribut (simple ou groupé et de classe quelconque) à une scène mémorisée quelconque s'effectue à l'aide de la primitive

Attrib_att(ref_scène,ref_att)

où *ref_scène* est la référence de la scène concernée, et *ref_att* celle de l'attribut.

Cette primitive, similaire à la primitive d'attribution de CLOVIS, n'a aucun effet sur l'image. Si la scène concernée est affichée, elle doit être explicitement revisualisée afin que les modifications éventuelles soient prises en compte.

L'utilisation de cette primitive à l'intérieur d'un contexte de construction de structure a peu de signification. Toutefois, si la scène *ref_scène* est en cours de construction (scène courante ou scène de niveau supérieur non encore "fermée") à l'intérieur d'un contexte de visualisation, l'affectation a effectivement lieu mais l'attribut *ref_att* n'est pas pris en compte pour l'affichage des objets élémentaires.

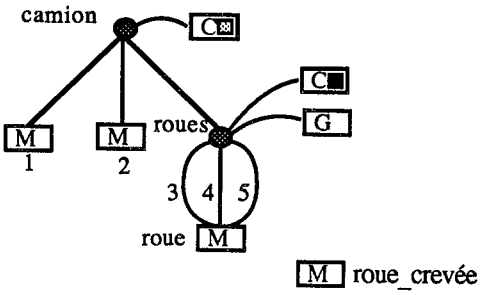
Une seconde primitive, au comportement analogue, permet de supprimer l'un des attributs d'une scène mémorisée :

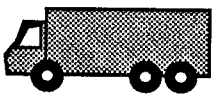
Sup_Att(ref_scène,classe_Att)

classe_att désigne la classe (ASPECT, GEOM2D, GEOM3D,VISIBILITE, COULEUR, REFLEXION) de l'attribut à supprimer dans la scène de référence *ref_scène.*

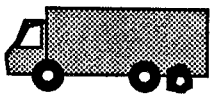
remarque : cette suppression consiste uniquement au retrait du lien entre le descripteur de la scène et le descripteur de l'attribut. Si la référence de l'attribut est conservée, il peut être réutilisé par ailleurs. A l'heure actuelle, dans le cas où l'attribut n'est plus utilisé, la récupération de l'espace mémoire ne s'effectue que par le mécanisme de définition de blocs de mémorisation présenté au § 2.3.

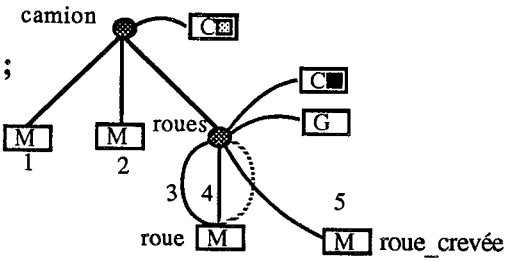
```
roue = Couronne(0,0,r1,r2);

camion = Scene();
    Couleur(50,50,50);  /* gris */
    Polyg2d(...);      /* la cabine (1) */
    Polyg2d(...);      /* la remorque (2) */
      roues = Scene();       /* les roues */
        Couleur(0,0,0);   /* noir */
        GGeom2D(3);  /* position des roues */
            Trans2D(...);
            Trans2D(...);
            Trans2D(...);
        FinG();
        Objet(roue);    /* la roue avant (3) */
        Objet(roue);     /* 1ère roue arrière (4) */
        Objet(roue);    /* 2ème roue arrière (5) */
    FinScene();
FinScene();

roue_crevée = Polyg2d(....);
```



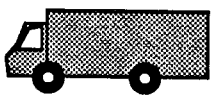


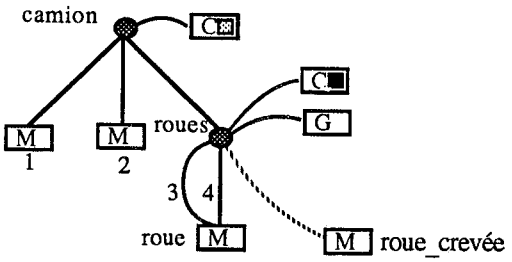
Substit_Obj(roues,roue,3,roue_crevée);



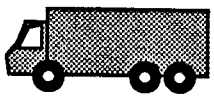


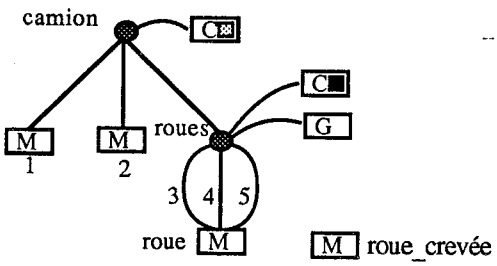
Sup_Obj(roues,roue_crevée,1);





Ins_Obj(roues,roue,2,roue);





● Objet structuré (scène)

[M] Attribut élémentaire (M : Morphologie, C : Couleur, G : Géométrie)

..... lien supprimé

- figure VIII.12 : modification de la liste des objets d'une scène -

3.3.2 Affectation d'objets

La primitive

Attrib_Obj(ref_scène,ref_obj)

permet d'affecter l'objet de référence *ref_obj* à la scène mémorisée *ref_scène*. *ref_obj* peut être une référence vers un objet de type absolument quelconque, que ce soit un objet élémentaire, un objet structuré (scène) ou une instance de scène.

Si la scène *ref_scène* est une scène mémorisée définie précédemment et est déjà affichée, cette attribution n'a pas d'effet sur l'image. Afin de tenir compte des modifications apportées à la scène, cette dernière doit être explicitement réaffichée à l'aide de la primitive *Objet* appelée dans le contexte de visualisation approprié.

Dans le cas de l'appel de *Attrib_Obj* dans un contexte de construction de structure :

- si la scène *ref_scène* est la scène courante, *Attrib_Obj* a le même effet que la primitive *Objet*,
- si la scène *ref_scène* est une scène de niveau supérieur (donc non encore terminée), l'objet *ref_obj* lui est tout de même affecté. Si le contexte est également un contexte de visualisation (affichage immédiat), l'objet *ref_obj* sera affiché dès le retour à la scène *ref_scène* . Si plusieurs objets ont été ainsi affectés à la scène *ref_scène*, ceux-ci sont affichés dans leur ordre d'attribution.

D'autres primitives, permettent de modifier la liste des objets regroupés dans une scène en supprimant, remplaçant, insérant des objets (fig. VIII.12 ci-contre).

Sup_Obj(ref_scène,ref_obj,no_inst)

retire l'objet de référence *ref_obj* de la liste des objets de la scène *ref_scène*. Il se peut qu'une même scène fasse plusieurs fois référence au même objet, à cet effet le paramètre *no_inst* est un entier qui permet de désigner laquelle de ces références doit être supprimée (*no_inst* indique son rang parmi celles-ci). L'emploi de la constante prédéfinie ***toutes*** permet de retirer d'un seul coup toutes les occurences de l'objet *ref_obj* dans la scène *ref_scène*.

Substit_Obj(ref_scène,ref_old_obj,no_inst,ref_new_obj)

permet la substitution de l'un des objets d'une scène (*ref_scène*) par un autre (*ref_new_obj*). L'objet à remplacer est désigné de manière identique que pour *Sup_Obj* par sa référence *ref_old_obj* et son rang *no_inst* parmi ses occurences dans la scène.

Ins_Obj(ref_scène,ref_prec,no_inst,ref_obj)

permet d'inserer un objet (*ref_obj*) dans la liste des objets de la scène *ref_scène*. La position d'insertion est indiquée en désignant l'objet (par *ref_prec* et *no_inst*) après lequel *ref_obj* doit être ajouté. Si on désire effectuer une insertion en tête de liste, la constante prédéfinie ***nul*** doit être utilisée en lieu et place de *ref_prec*. La constante prédéfinie ***fin*** peut être employée pour une insertion en fin.

Dans tous les cas, pour ces trois dernières primitives, l'image n'est pas mise à jour et la scène modifiée doit si nécessaire être explicitement réaffichée à l'aide de la primitive *Objet* , appelée dans un contexte de visualisation.

3.4 Modification d'attributs et d'objets élémentaires mémorisés

A tout moment, il est possible de modifier directement la valeur d'un attribut ou d'un objet élémentaire mémorisé. Cette modification s'effectue en deux étapes :

- en premier lieu la désignation de l'objet ou de l'attribut à modifier, par l'appel de la primitive

Modif(ref)

où *ref* est la référence de l'objet ou de l'attribut élémentaire concerné.

- ensuite, doit immédiatement suivre l'invocation d'une primitive de définition d'un attribut ou d'un objet élémentaire. Cet attribut remplace l'attribut précédement désigné par *Modif*, il doit être du **même type** que ce dernier. Une telle modification n'a aucun effet sur l'image affichée, elle ne concerne que la banque de données du logiciel graphique.

Exemple :

```
DebEcr(Banque);
   ...
   cercle1 =Cercle2D(0,0,50); /* cercle de centre 0,0 de rayon 50 */
   ...
FinEcr();
...
Objet(cercle1); /* affiche un cercle de centre 0,0 de rayon 50 */
...
Modif(cercle1); /* modification de cercle1 remplacé par un cercle */
Cercle2D(0,0,100);  /* de centre 0,0 et de rayon 100 */
...
Objet(cercle1); /* affiche un cercle de centre 0,0 de rayon 100 */
```

remarque : Dans le cas de la modification d'un objet dont la longueur peut être variable, par exemple le nombre de sommets pour une ligne brisée, l'utilisateur doit s'assurer que la taille du nouvel objet est inférieure ou égale à celle de l'objet à remplacer. Ceci, pour la simple raison qu'aucune nouvelle allocation n'est effectuée dans la banque de données graphiques, le nouvel objet ou attribut prenant la place de celui désigné par la référence *ref* dans la primitive *Modif*.

4 CONSULTATION

La consultation permetau programme d'application de retrouver l'information conservée dans la banque de données du logiciel graphique. Les primitives liées à ce processus se répartissent en deux groupes :

- des primitives de consultation de structure et d'identité qui s'appliquent aux objets,

- des primitives de consultation d'attributs élémentaires qui permettent de retrouver les attributs associés à des objets et/ou l'information qui les définis.

4.1 Primitives de consultation de structure

Ces primitives complètent les primitives ***Ref_Identité*** et ***Ref_Identité_struct*** présentées au § 3.2.4.2. Elles permettent au programme d'application de récupérer l'information concernant la structure et l'identité d'éléments dont la référence est connue.

Identité(ref_obj) --> identificateur

renvoie l'identificateur de l'objet de référence *ref_obj*. Si celui-ci ne possède pas de nom, une chaîne vide est retournée.

Identité_Elt_Scène(ref_scène,rang) --> identificateur

retourne l'identificateur d'un objet de la scène *ref_scène* désigné par son rangd dans celle-ci. Comme pour la primitive ***Identité***, si l'objet ne possède pas de nom ou si le rang est incorrect (supérieur au nombre d'éléments de la scène *ref_scène*), une chaîne vide est retournée.

Rang(ref_scène,ref_obj,i) --> entier

permet de retrouver parmi les éléments de la scène de référence *ref_scène* , le rang de la $i^{ème}$ occurence de l'objet de référence *refobj* . Cette fonction retourne 0 si l'objet *ref_obj* est absent de la scène *ref_scène* ou si dans la scène *ref_scène* il y a moins de *i* références à l'objet *ref_obj*.

Nbre_Occ(ref_scène,ref_obj) --> entier

fourni le nombre de références directes à l'objet *ref_obj* présentent dans la scène *ref_scène*. Cette fonction retourne 0 si il n'existe aucune référence à *ref_obj* dans *ref_scène*.

De façon similaire la primitive

Nbre_Nom(ref_scène,identificateur) --> entier

indique le nombre de fois où un identificateur est utilisé pour désigner des objets différents à l'intérieur d'une scène.

Type_Objet(ref_obj) --> type

permet de connaître la nature de l'objet *ref_obj* . Le type de l'objet est indiqué par l'une des constantes prédéfinies :

- ***elem*** pour les objets élémentaires (morphologie de bas),
- ***sce*** pour les objets structurés (scènes),
- ***inst*** pour les instance de scènes.

A partir de la référence d'un objet de type instance, il est possible de retrouver la référence du noeud "original" racine de la sous-structure partagée avec l'instance, à l'aide de la primitive

Ref_Orig(ref_inst) --> ref_scène

(remarque : *ref_inst* doit être une référence vers un noeud de type instance, dans le cas contraire une référence nulle est retournée.)

Nbre_Ref(ref_obj) --> entier

indique le nombre de fois ou l'objet *ref_obj* est référencé par des scènes.

Identité_Structure(ref_obj) --> ref_multiple

donne sous forme d'une chaîne de caractères, expression de référence multiple, la structure de l'objet de référence *ref_obj* et les identificateurs des éléments qui le constituent.

4.2 Primitives de consultation d'attributs élémentaires

La consultation des attributs élémentaires d'une scène s'effectue à l'aide de la primitive

Consulter_Att(ref_scène,typatt) --> ref_att

qui retourne la référence de l'attribut de type *typatt* associé à la scène *ref_scène*. Si la scène ne possède pas d'attribut du type indiqué une référence nulle est retournée.

Afin de compléter cette primitive de consultation et de permettre au programme d'application d'accéder au contenu des descripteurs d'attributs, il existe pour chaque type d'attribut élémentaire une primitive de "démodélisation" dont le rôle est symétrique de celui des primitves de définition (construction ou modélisation) d'attributs. Par exemple alors qu'un attribut de couleur peut être défini par la primitive ***Couleur(r,v,b)*** , la primitive

Cons_Couleur(ref_att,r,v,b)

fourni les composantes rouge, vert et bleu de l'attribut de couleur de référence *ref_att*.

Pour chacune des primitives de définition d'attribut et d'objet élémentaire (voir § 3.1), il existe de manière analogue une telle primitive de consultation dont les paramètres sont équivalents.

Une dernière primitive complète la consultation des attributs élémentaires, elle permet de connaître le type d'un attribut ou d'un objet élémentaire:

Typ_att(ref) --> typatt

Cette primitive est surtout utile pour retrouver la nature d'un attribut élémentaire pour, par exemple, recupérer ensuite l'information qui le caractérise à l'aide de la primitive de démodélisation appropriée.

5 DESCRIPTION

Les primitives de dialogue élémentaires (collecte d'un point, d'une couleur en un point...) et les primitives de définition de contextes de dialogue présentées au cours du chapître VII, permettent aux programmes d'application de construire aisément leurs propres séquences d'interaction.

Cependant, il nous parait souhaitable d'intégrer au logiciel graphique, des primitives de haut niveau pour la description interactive des différents attributs élémentaires manipulés de la même manière que nous l'avions fait dans CLOVIS.

Le mécanisme de contexte permet de réaliser sans difficultés majeures la description dont la forme pourrait être

```
Decrire(refdom,typatt,proc) -> reffatt;
(* description d'un attribut de type typatt à l'aide du processus proc, le support de l'interaction
    étant le domaine refdom *)
debut
   Debdial(refdom);
      ....
      Opérations d'interaction pour la description de l'attribut en fonction du type et du
      processus
      ....
   FinDial()
   retour(reference de l'attribut créé)
fin;
```

De la même manière que pour l'attribution, selon le contexte dans lequel le processus de description est invoqué, l'attribut décrit interactivement est soit :

- immédiatement utilisé pour la visualisation, il devient l'attribut courant de sa classe (contexte de visualisation immédiate),

- conservé dans la banque de données graphiques pour pouvoir être utilisé ultérieurement (contexte de mémorisation seule),

Les deux modes peuvent être combinés. Par ailleurs, si l'attribut est décrit dans un contexte de création de structure, il est affecté à la scène courante. Ce dernier point, combiné avec une visualisation immédiate, permet de définir interactivement certains des attributs d'un objet au fur et à mesure de son élaboration.

L'exemple qui suit illustre la description en liaison avec la construction d'une structure, il s'agit de la construction d'une voiture. Tout d'abord, sa carrosserie est affichée, ensuite pour les roues, l'utilisateur les positionne interactivement en décrivant une translation dans le repère de la carrosserie.

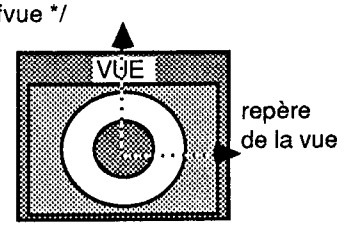
```
vue0 = DefVue(image0,"voiture",0,0,512,512);
....
DebEcr(vue0 + Banque);
   ...
   voiture = Scene(3);
      Couleur(255,0,0);
      Objet(carrosserie);
      Scene(1);
         (* description d'une translation permettant de positionner la roue
            avant relativement à  la carrosserie * )
         Decrire(vue0,Trans2d,rel);
         Objet(roue);
      Fin();
      Scene(1);
         (* description d'une translation permettant de positionner la roue
            arrière relativement à  la carrosserie * )
         Decrire(vue0,Trans2d,rel);
         Objet(roue);
      Fin();
   Fin();
   ...
FinEcr();
```

6 VISUALISATION

Dans ce second système, la visualisation est entièrement controlée par le mécanisme de définition de contexte d'écriture. Comme nous l'avons vu tout au long de ce chapitre, ce sont les mêmes primitives qui permettent selon la nature du contexte dans lequel elles sont invoquées d'effectuer une attribution, une simple visualisation ou la combinaison des deux processus. Ainsi, pour la visualisation d'objets graphiques l'utilisateur a le choix entre de nombreuses possibilités :

- un visualisation **immédiate d'objets élémentaires**. Les objets élémentaires sont affichés au fur et à mesure de leur définition avec les derniers attributs spécifiés dans le contexte de visualisation courant. Des attributs mémorisés peuvent être utilisés.

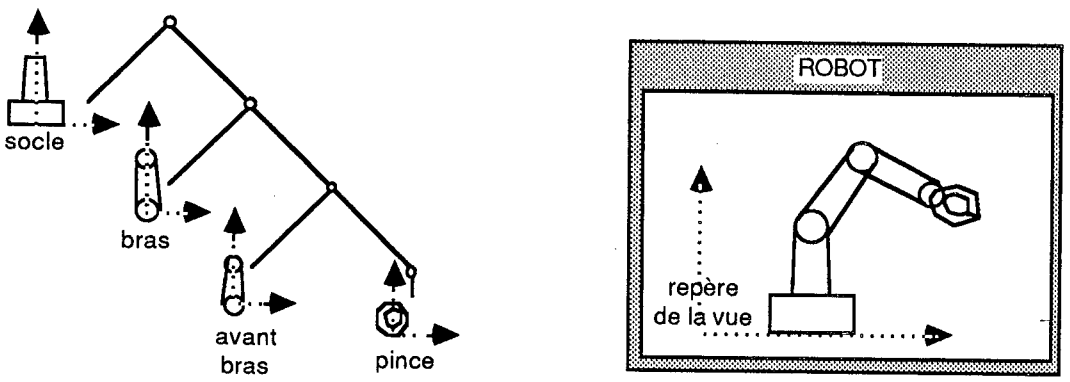
```
DebEcr(refvue);  /* visualisation immédiate dans la vue de référence refvue */
  Couleur(255,255,255);
    Cercle(0,0,10);  /* affichage d'un cercle avec la couleur blanche */
    Ligne2d(....); /* affichage d'une ligne brisée avec la couleur noire */
    Attrib(gris); /* gris est la référence d'une couleur définie dans un
      contexte de      mémorisation : gris = Couleur(100,100,100) */
     Cercle(0,0,5);  /* affichage d'un cercle avec une couleur grise */
  FinEcr();
```



- figure VIII.13 : visualisation immédiate d'objets élémentares -

- un visualisation **immédiate d'objets structurés**. Les objets élémentaires sont affichés au fur et à mesure de leur définition avec les attributs évalués au niveau de la scène courante. Cette possibilité est particulièrement intéressante pour l'affichage d'objets regroupant différents constituants dont il est parfois plus aisé de décrire la morphologie dans des repères locaux. L'assemblage des différents composants s'effectue à l'aide de transformations géométriques qui positionnent chaque sous-objet dans le repère de l'objet de niveau supérieur. La structuration sous de forme de scènes permet de définir facilement cette hiérarchie, les attributs, en particulier les attributs géométriques, sont composés et évaluées atomatiquement pour l'affichage des objets. Ainsi, dans l'exemple de la figure VIII.14 ci-dessous, chaque constituant du robot est solidaire de celui de niveau supérieur, ce qui s'exprime simplement à l'aide d'une structure de scènes.

```
DebEcr(refvue); /* affichage dans la vue de reference refvue */
  Trans2d(Rloc,...); /* positionne le robot dans le repère de la vue */
  Scene();
    Couleur(0,0,0); /* noir pour les traits */
    Aspsurf(0,0); /* pas de remplissage des surfaces*/
    Polyg2d(...);  /* le socle */
    Scene();
      Geom2d(Rloc,...); /* positionne le bras par rapport au socle */
      Cercle(0,0,5);
      Polyg2d(...);
      Cercle(0,10,3);
      Scene();
        Geom2D(Rloc...); /* positionne l'avant bras par rapport au bras */
        Cercle(0,0,3);  /* l'avant bras */
        Polyg2d(...)
        Scene();
          Geom2D(Rloc...); /* positionne la pince par rapport au socle */
          Polyg2d(...)
        Fin();
      Fin();
    Fin();
  Fin();
FinEcr();
```

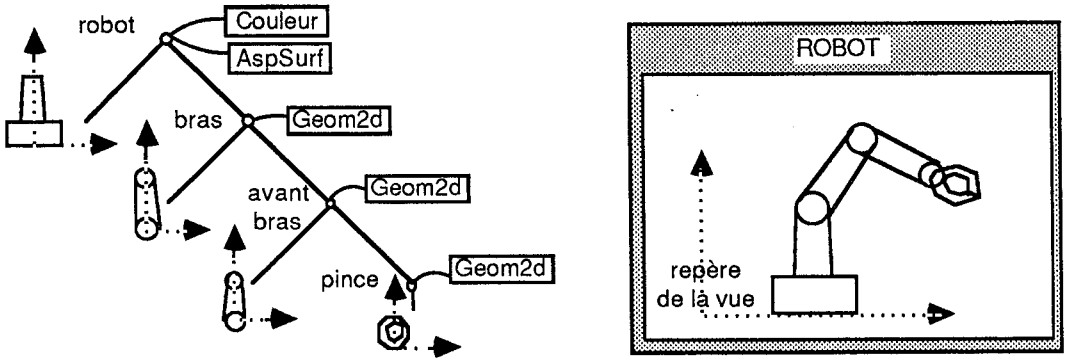


- figure VIII.14 : visualisation immédiate d'un objet structuré -

Les deux cas précédents correspondent à une définition procédurale des objets graphiques. Les objets décrits ne sont pas mémorisés, aussi leur réaffichage nécessite la réexécution des primitives les définissant.

Il est possible de combiner affichage immédiat et mémorisation. Comme précédemment les objets sont affichés au fur et à mesure de leur définition avec les mêmes règles d'évaluation des attributs, mais en plus les objets et attributs spécifiés sont conservés dans la banque de données du système graphique (fig. VIII.15).

```
refer  robot,bras,avant_bras,pince;
DebEcr(refvue + Banque); /* affichage dans la vue de reference refvue + mémorisation */
   Trans2d(Rloc,...); /* positionne le robot dans le repère de la vue */
   robot = Scene();
      Couleur(0,0,0); /* noir pour les traits */
      Aspsurf(0,0); /* pas de remplissage des surfaces*/
      Polyg2d(...);  /* le socle */
      bras = Scene();
         Geom2d(Rloc,...); /* positionne le bras par rapport au socle */
         Cercle(0,0,5);
         Polyg2d(...);
         Cercle(0,10,3);
         avant_bras = Scene();
            Geom2D(Rloc...); /* positionne l'avant bras par rapport au bras */
            Cercle(0,0,3);  /* l'avant bras */
            Polyg2d(...)
            pince = Scene();
               Geom2D(Rloc...); /* positionne la pince par rapport au socle */
               Polyg2d(...)
            Fin();
         Fin();
      Fin();
   Fin();
FinEcr();
```



Structure construite et **mémorisée**

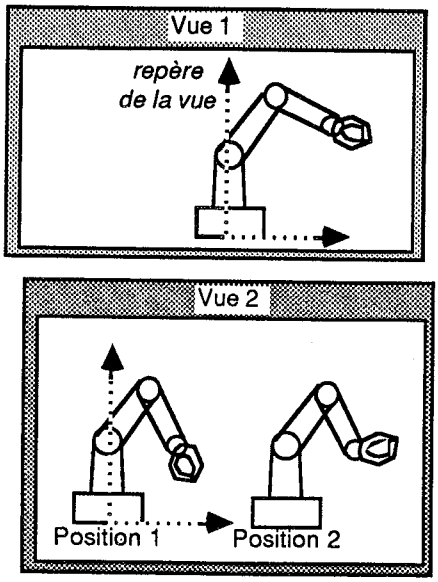
Affichage dans la Vue*refvue*

- figure VIII.15 : affichage immédiat et mémorisation simultanée -

Les objets structurés mémorisés peuvent être modifiés à l'aide des primitives d'attribution explicite. Celles-ci n'ont pas d'effet immédiat sur l'image, même si l'objet concerné est affiché. Elle n'agissent que sur la banque de données. Le réaffichage des objets mémorisés doit être explicitement demandé par l'appel d'une primitive d'attribution implicite dans un contexte de visualisation.

La figure VIII.16, page suivante, donne un exemple de visualisation d'objets mémorisés. L'objet affiché est le robot défini dans l'exemple de la figure VIII.15 ci-dessus. Trois visualisation de cet objet sont effectuées dans deux vues différentes. La première consiste au simple affichage de l'objet tel qu'il a été défini. Les deux suivantes, effectuées dans la même vue, réaffichent le robot après voir respectivement effectué une rotation de son "avant-bras" et de sa "pince".

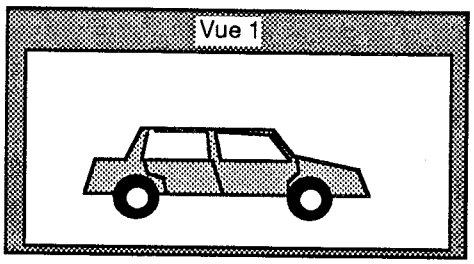
```
DebEcr(Banque); /* memorisation simple d'attributs */
   rot1 = Geom2d(Rel,0,0,-30); /* rotation pour l'avant bras */
   rot2 = Geom2d(Rel,0,0,90);  /* rotation pour la pince */
   ...
FinEcr();
....
DebEcr(Refvue1);  /* affichage dans la vue 1 */
   ...
   Objet(robot); /* affiche le robot dans la vue 1 */
   ...
   /* rotation de l'avant bras du robot */
   Attrib(avant_bras,rot1);
   ...
   DebEcr(Refvue2);   /* affichage dans la vue 2 */
      Texte(-10,-10,"Position 1");
      ...
      Objet(robot); /* affiche le robot dans la vue 2 */
      ...
      /* rotation de la pince du robot */
      Attrib(pince,rot2);
      ....
      Texte(50,-10,"Position2");
      ....
      Trans2d(Rel,60,0);
      Objet(robot); /* affiche le robot dans la vue 2 */
   FinEcr();
   ...
```



- figure VIII.16 : Visualisation d'objets mémorisés -

La visualisation d'objets mémorisés peut être combinée avec une construction de structure, dans ce cas l'objet est affiché à l'aide des attributs évalués au niveau de la scène courante et est également affecté à celle-ci (fig VIII.17).

```
refer roue, voiture;
DebEcr(refvue1 + Banque); /* visualisation + mémorisation */
   voiture = Scene();
      Couleur(100,100,100);
      Polyg2d(...); /* carrosserie */
      Scene();
         Trans2d(Rel,-40,0);
         roue = Scene();
           Couleur(0,0,0);
           Cercle(0,0,10);
           Scene();
             Couleur(255,255,255);
             Cercle(0,0,7);
           Fin();
         Fin();
      Fin();
      Scene();
         Trans2d(Rel,-40,0);
         Objet(roue);   /*visualisation et affectation de la roue */
      Fin();
   Fin();
FinEcr();
```



- figure VIII.17 : visualisation et affectation d'objets mémorisés -

Les exemples qui précèdent montrent la richesse des différentes méthodes proposées par le système graphique pour la visualisation d'objets. De par leur diversité, elles permettent de satisfaire pleinement les besoins des applications. Il est à noter que les concepteurs d'objets d'interface (voir chapitre VII), ont accès à toutes ces fonctionnalités pour la définir la

présentation visuelle de ces objets. L'attribut *affich* associé aux objets d'interface est une fonction dont le contenu est laissé à la totale liberté du concepteur. Elle peut ainsi soit réaliser un affichage immédiat sans mémorisation, soit afficher un objet mémorisé (dont la référence peut être conservée au niveau de l'objet d'interface), soit combiner les deux...

Le processus de visualisation effectivement appliqué dépend quand à lui de la configuration de l'image sur laquelle l'affichage a lieu. Cependant, il demeure totalement transparent à l'utilisateur, le logiciel graphique prenant totalement en charge la gestion de l'architecture banalisée. La configuration d'une image défini automatiquement les opérateurs de bas niveau qui seront invoqués lors de l'affichage dans le contexte d'une vue située sur-celle-ci.

7 CONCLUSION

Dans ce chapitre nous avons essayé de donner une vision globale des fonctionnalités proposées par ce second logiciel pour la visualisation d'objets graphiques. Par la combinaison des modes de visualisation immédiate, mémorisation et structuration hiérarchique des attributs graphiques, nous pensons avoir avoir atteint l'objectif que nous étions fixé et qui consistait à offrir aux programmeurs d'applications une plus grande souplesse dans l'utilisation du système graphique. En particulier l'emploi conjoint de ces modes introduit une séparation beaucoup moins forte que dans CLOVIS entre les différents processus de base. Ainsi, il est par exemple possible de mêler l'attribution et la structuration des données graphiques à la visualisation ou bien à la description.

La définition de contextes d'écriture est un mécanisme puissant qui permet aux applications de contrôler de manière efficace et structurée les passages entre ces différents modes. En fonction de ce contexte, les mêmes primitives permettent la visualisation et/ou la mémorisation et/ou la structuration d'objets graphiques. A partir d'un jeu de primitives restreint et cohérent, l'utilisateur dispose de toute un gamme de possibilités qui vont du simple affichage d'objets élémentaires à la définition d'objets complexes décrits à l'aide d'une structure hiérachique dont la mémorisation et la gestion est confiée au système graphique.

Si la structuration hiérarchique des attributs graphiques introduite dans CLOVIS a été en grande partie conservée, là aussi une plus grande souplesse d'utilisation a été introduite. Le report des attributs globaux des classes Gv, Ga et E au niveau des objets d'interface, assure l'indépendance entre la description des objets graphiques et leur présentation visuelle qui sont clairement séparés. Le traitement des morphologies de base en tant qu'objets élémentaires, le passage non obligatoire par la structure hiérarchique pour leur visualisation, les différents modes de construction et d'accès aux structures sont autant d'améliorations qui répondent aux limitations que nous avions rencontré avec CLOVIS, sans pour autant remettre en cause la puissance du système graphique.

CONCLUSION

Au cours de la dernière décennie, nous sommes entrés de plein pied dans "l'ère" du graphique interactif et de la synthèse d'images.

Cependant, si du point de vue technique la plupart des problèmes posés ont été résolus, il semble que l'on se dirige à l'heure actuelle vers une spécialisation de plus en plus marquée. Ce phénomène est valable tant du point de vue du matériel, avec en particulier la multiplication des stations de travail qui allient puissance de calcul à des capacités graphiques haute résolution et sont distribuées autour d'un réseau local, que du point de vue des algorithmes qui s'attaquent à des problèmes très spécifiques (comme par exemple la prise en compte très précise de modèle physiques décrivant les phénomènes à visualiser (vagues, effets spéciaux d'éclairage... etc...)). Certaines machines, se spécialisent ainsi pour tel ou tel type d'algorithme, par exemple le lancer de rayons (Ray Tracing) pour la visualisation réaliste...

Mais si l'évolution des techniques a été aussi importante, et permet de produire aujourd'hui des images au réalisme frappant, notamment pour les spots publicitaires et le cinéma, les outils permettant de les intégrer à un système graphique et de les rendre accessibles à des utilisateurs non spécialistes demeurent encore en partie à développer.

Ainsi, il est clair que le logiciel n'a pas toujours su suivre l'évolution foudroyante des matériels (mais est-ce un problème propre au graphique ?). Certes, l'effort de normalisation pour des systèmes graphiques généraux de haut niveau, indépendants du matériel a été une contribution importante dans la définition et la compréhension des concepts de base. Mais ces premières tentatives, pour des raisons tant techniques que historiques n'ont pas complètement porté leurs fruits et révèlent une certaine obsolescence aussi bien du point de vue de l'évolution des matériels que de l'évolution des besoins des utilisateurs. Cette évolution, remet en particulier en cause les fonctionnalités essentielles des logiciels graphiques de base qui demeurent encore aujourd'hui une question largement contreversée.

C'est notre contribution dans ce domaine, que nous avons tenté d'exposer tout au long de ce rapport relatant nos travaux au sein de l'équipe graphique du laboratoire ARTEMIS.

Dans un premier temps, nous avons essayé de dégager un ensemble de concepts de base afin de pouvoir mener notre étude de manière systématique. A la lumière de ceux-ci, nous nous sommes ensuite efforcés de mettre en évidence les principaux problèmes soulevés pour le logiciel graphique et les différentes approches qui ont été adoptées pour les résoudre. Dans la séparation que nous avons adoptée au cours de notre exposé, tout d'abord la visualisation puis la modélisation, nous avons voulu refléter une certaine évolution historique qui explique en partie les problèmes actuels rencontrés dans le logiciel graphique.

Le système CLOVIS que nous avons présenté dans la seconde partie de cette thèse représente quand à lui la première étape vers un système graphique de haut niveau. D'une organisation logicielle originale (organisation hiérarchique (voir § chapitre V), prise en charge et ordonnancement des processus, structuration hiérarchique des données graphiques), il nous a permis de tester et valider une approche relativement nouvelle et de présenter une alternative intéressante aux techniques logicielles de l'époque. Mais, là encore il nous a fallu faire face l'évolution des matériels et des besoins des utilisateurs. En particulier, la conception au sein de l'équipe graphique du matériel HELIOS-GETRIS, architecture spécialisée pour la synthèse d'images, banalisée et reconfigurable, a nécessité pour la réalisation du logiciel de base permettant son exploitation la refonte d'un certain nombre de points de CLOVIS.

A cette occasion, nous avons finalement décidé d'abandonner le projet CLOVIS, pour concevoir un second logiciel tirant parti de cette première expérimentation mais résolument tourné vers la conception d'applications interactives en synthèse d'images et exploitant au mieux les possibilités multiples des matériels HELIOS-GETRIS. Nous avons ainsi essayé de définir un logiciel graphique de haut niveau en conservant néanmoins une très grande souplesse d'utilisation (chapitre VI). Par ailleurs, la nécessité de prendre également en compte les problèmes spécifiques à la définition d'interfaces graphiques homme-machine, nous est apparue. En effet, il ne nous a pas semblé souhaitable de dissocier ceux-ci de la visualisation d'objets graphiques, ces deux points étant intimement liés.

Nous pensons, avec ce second système, avoir dégagé un ensemble de fonctionnalités qui permettent de répondre de manière cohérente aux besoins d'applications interactives de synthèse d'images.

La distinction entre objets graphiques et objets de dialogue, nous a permis de mieux séparer les problèmes ayant trait à l'interface utilisateur des problèmes uniquement liés à la visualisation graphique. La prise en charge du dialogue interactif par le système au travers de primitives de haut niveau, permet la définition aisée et cohérente de l'interface homme-machine des applications. La représentation que nous avons adoptés pour les objets de dialogue (chapitre VII), garantit l'ouverture du système en permettant la définition de nouvelles entités.

Le deuxième point important, est la notion de contexte qui, généralisée à l'écriture (visualisation, mémorisation), au dialogue et à la lecture (collecte d'informations en mémoire d'images), va dans le sens d'une plus grande souplesse d'utilisation du système, sans pour autant reporter le contrôle au niveau des applications. A cet effet, la distinction et l'imbrication des contextes facilite la définition dynamique par l'application des environnements pour l'exécution des opérations qu'elle réalise à l'aide du système graphique et encourage une programmation structurée. La gestion de ceux-ci par le logiciel graphique protège l'application d'effets de bords indésirables.

C'est en grande partie par l'utilisation de cette notion de contexte (en particulier de contexte d'écriture) que nous avons pu définir un ensemble de fonctionnalités extrêmement souples pour la définition et la visualisation d'objets graphiques (chapitre VIII). C'est autour d'elle que nous avons fait évoluer les fonctionnalités que nous proposions avec CLOVIS vers une meilleure adéquation aux besoins des applications.

Le logiciel que nous avons ainsi conçu permet une utilisation efficace au travers d'une interface de haut niveau des matériels HELIOS-GETRIS. L'ensemble de ces travaux, nous a rapproché d'un système graphique prenant en charge de manière ***structurée*** *et* ***cohérente*** *la* ***visualisation*** *interactive* ***et*** *le* ***dialogue*** *interactif de haut niveau. Développé initialement pour HELIOS-GETRIS, ce logiciel pourrait être adapté à d'autres matériels. Nous pensons qu'il ne s'agit que d'une étape, comme CLOVIS et les autres systèmes auparavant, vers la définition d'un système graphique complet. Dans cette direction plusieurs axes de recherche sont à explorer :*

- une meilleure prise ne compte de l'interface entre le logiciel graphique et l'architecture matérielle, la définition de la nature et du niveau de cette interface restant un problème loin d'être résolu de manière satisfaisante,

- au niveau des problèmes de l'interface homme-machine, les systèmes de gestion d'interface utilisateur ("User Interface Managment Systems") permettant une description graphique interactive de celle-ci, nous semblent être une voie très prometteuse. Les techniques d'intelligence artificielle ont, dans ce domaine, un rôle très important à jouer.

- Le graphique interactif, et les problèmes d'interface Homme-Machine, ont été des éléments moteurs dans la définition des nouvelles techniques de génie logiciel qui ont donné lieu à l'émergence de nouveaux langages que l'on désigne aujourd'hui sous le terme de "programmation orientée objet". La formalisation des concepts d'objet et le développement de tels langages, doivent permettre d'améliorer encore l'abstraction et l'interface des systèmes graphiques.

Voici donc quelques unes des voies qu'il nous semblerait important d'explorer, et qui devraient nous permettre d'aboutir à une meilleur compréhension et méthodologie dans la conception des systèmes graphiques. Néanmoins, il faut être conscient que l'irruption de plus en plus grande du parallélisme dans les machines actuelles et surtout à venir, risque de bouleverser de manière fondamentale un certain nombre de concepts et techniques qui semblaient bien établis. Ce sera à nous de faire le chemin permettant d'effectuer la jonction entre les besoins des applications et les possibilités des matériels, afin d'exploiter efficacement ces nouvelles techniques.

BIBLIOGRAPHIE

[AbBu 86] Abi-Ezzi, S., Bunshaft, A. J.
An Implementer's View of PHIGS.
IEEE - Computer Graphics & Applications, Février 1986.

[AdLe 86] Adair, W. B., Ledford, V. G.
Using GKS in an application.
proc. NCGA, Mai 1986.

[Adob 85] Adobe
Poscript Language Reference Manual
Addison Wesley, Reading, Mass, 1985

[Aman 87] Amanatides, J.
Realism in Computer Graphics: A Survey.
IEEE - Computer Graphics & Applications, Janvier 1987.

[AnDr 85] Andrieux, A.Drouler, C.
Programmez votre Macintosh
Ed Mac Graw-Hill, 1985

[ANSI 83] Draft proposed American National Standard For the Virtual Device Metafile.
American National Standard Institute X3H3 83-15 R2, 1983.

[ANSI 85] Computer Graphics Virtual Device Interface
American National Standard Institute, Document X3H3 85-47, 1985.

[ANSI 85] Draft American National Standard for the Functionnal Specification of the Programmer's Hierarchical Interactive Graphic System (PHIGS).
American National Standard Institute X3H3/85-21 R2, Août 1985

[Arno 81] Arnold, D.
The Requirement for Process Structured Graphics Systems.
Computer Graphics, Vol 15, n°2, Juillet 1981.

[Athe 83] P.R. Atherton
A scan-line hidden surface removal algorithm for constructive solid geometry
Computer Graphics, vol 17, n°3, juillet 1983

[BaSi 86] Barthet , M.F, Sibertin-Blanc, C.
La modélisation d'applications interactives adaptées aux utilisateurs par des réseaux de Petri à structure de données;
Actes du Troisième colloque-Exposition de Génie Logiciel
Versailles, Mai 1986.

[BEHH 82] Bono, P. R., Encarnaçao, J. L., Hopgood, F. R., Hagen, T. J.
GKS : The First Graphics Standard.
IEEE - Computer Graphics & Applications, Juillet 82.

[Benn 86] Bennett, J.L.
Tools for building advanced user interfaces.
IBM Systems Journal, Vol 25, n° 3/4, 1986

[BBFo 78] Bergeron, R. D., Bono, P. R., Foley, J. D.
Graphics programming using the Core System.
Computing Surveys, Vol 10, n° 4, Décembre 1978.

[Bert 86] Berton, A.
Une station de multiposte pour applications techniques et CAO.
Electronique Industrielle, N° 103, Mars 1986.

[BCVD 77] Boos, J., Caruthers, L. C., Van Dam, A.
A GPGS, a device independent General Purpose Graphic System for stand alone and satellite graphics.
Computer Graphics, Proc. SIGGRAPH 77, Vol 11, N° 2, Août 1977.

[Bono 85] Bono, P. R.
A survey of graphics standards and their role in information interchange.
IEEE - Computer , Octobre 1985.

[Boul 85] Boulze, A.
Dans le cadre de CLOVIS : un logiciel pilote du terminal de synthèse d'images réalistes HELIOS.
Memoire d'ingénieur CNAM. Grenoble, Octobre 1985.

[Brun 82] Bruns, B.
A discussion of graphic software standards.
Computer graphics World, n° 8, 1982

[CaGi 86] Carson, G.S., Mc Ginnis, E.
The Reference Model for Computer Graphics.
IEEE - Computer Graphics & Applications, Août 1986.

[CCVa 85] Carlbom, I., Chakravaty, I., Vanderschel, D.
A hierarchical Data Structure For representing the Spatial Decomposition of 3D objects.
IEEE - Computer Graphics & Applications, Avril 1985.

[CaKu 86] Campbell, R.H., Kubitz, W.J.
The Professional Workstation Research Project.
IEEE - Computer Graphics & Applications, Mai 1986.

[CaPi 78] Carlbom, I., Piacoreck, J.
Planar Geometric projections and viewing transformations.
Computing Surveys, Vol 10, N° 4, Decembre 1978.

[Cars 83] Carson, G.S.
The Specification of Computer Graphics Systems.
IEEE - Computer Graphics & Applications, Septembre 1983.

[Cars 84] Carson, G.S.
An Approach to the Formal Specification of Computer Graphics Systems.
Computer & Graphics, Vol 8, N° 1, 1984.

[CGPS 84] Cahn, D.U., Mc Ginnis, E., Puk, R.F., Seum, C.S.
The PHIGS System.
Computer Graphics World, Février 1984.

[Chib 86] Chibane K.
Etudes et Evaluations d'Architectures de Pré-Synthétiseurs d'Images Réalistes. HELIOS-GETRIS.
Thèse de Docteur Ingénieur de l'Institut National Polytechnique de Grenoble, Novembre 1986.

[Clar 76] Clark J.M.
Hierarchical Geometric Models for Visible Surface Algorithms
Communication of the ACM - Vol 19, N° 10, Octobre 1976.

[Clar 85] Clarkson, T. B.
Standardized Graphics on the IBM Personal Computer.
IBM System Journal, Vol 24, N° 1, 1985.

[Cout 87] Coutaz, J.
Interfaces Utilisateur Evoluée.
Cours INRIA - Sophia Antipolis - 20-23 Octobre 1987.

[Cox 86] Cox, B. J.
Object Oriented Programming.
Ed. Addison Wesley, 1986.

[Dam 84] van Dam, A.
Computer Graphics comes of Age.
Communications of the ACM, Vol 27, n° 7, Juillet 1984.

[EEKP 79] Eckert, R., Enderle, G., Kansy, K., Prester, F. J.
GKS 79, Proposal for a Standard for a Graphic Kernel System.
proc. Eurographics 79, Bologna, 1979.

[EEKN 80] Encarnacao, J., Enderle, G., Kansy, K., Nees, G., Schlechtendahl, E.G., Weiss, J. and Wibkirchen, P.
The Workstation Concept and the resulting conceptual differences to the GSPC proposal.
Computer Graphics, Proc. SIGGRAPH 80, Vol 14, N° 2, Août 1980.

[EnKP 84] Enderle, G., Kansy, K., Pfaff,P.
Computer Graphics Programming, GKS the Graphics Standard.
Ed. Springer Verlag, 1984.

[Ende 85] Enderle, G.
Guest Editor's Introduction Computer Graphics Standards.
Computers & Graphics, Vol 9, N° 1, 1985.

[Ever 52] Everett, R.
The Whirlwind I Computer.
Joint AIEEE-IRE, Electr. Digit. Computer, 1952.

[Exco 88] Excoffier, T.
Le lancer de rayon: Fondements et méthodes d'accélération
Modèles géométriques et Images numériques
Lyon, 26 et 27 Avril 1988

[FAGr 83] Fuchs, H., Abram, G.D., Grant, E.D.
Near Real-Time Shaded Display of Rigid Objects.
Computer Graphics, Proc. SIGGRAPH 83, Vol 17, n° 3, Juillet 1983.

[Feir 81] Ferreira, F. N.
Conception et Réalisation d'un Système Interactif pour la Synthèse d'Images Réalistes : HELIOS.
Thèse de Docteur Ingénieur de l'Institut National Polytechnique de Grenoble, Septembre 1981.

[FKNa 80] Fuchs, H., Kedem, Z.M., Naylor, B.F.
On Visible Surface Generation By A Priori Tree Structures.
Computer Graphics, Proc. SIGGRAPH,

BIBLIOGRAPHIE

[Fole 82] Foley, J.D.
The SIGGRAPH Core System Today.
Computer Graphics World, Août 1982.

[FoDa 82] Foley, J.D., Van Dam, A.
Fundamentals of Interactive Computer Graphics.
Ed. Addison Wesley, 1982.

[FoWe 81] Foley, J.D., Wenner, P.A.
The George Washington University Core System Implementation.
Computer Graphics, Vol 15,N° 3, Août 1981.

[GaOs 85] Gallop, J. R., Osland, D. C.
Experiences with implementing GKS on a Perk and others computers.
Computer Graphics Forum, N° 4, 1985.

[GAWh 86] Grant, E., Amburn, P., Whitted, T.
Exploiting Classes in Modeling and Display Software.
IEEE - Computer Graphics & Applications, Novembre 1986.

[GeGr 85] Genoud, Ph. ; Grabowiecki, J.F.
A general purpose graphic environment : CLOVIS
Congrès I.A.S.T.E.D., Paris, Juillet 1985.

[GeGr 86] Genoud, Ph. ; Grabowiecki, J.F.
Vers un logiciel graphique interactif de Haut Niveau
5ème conférence européenne CAO et Infographique, MICAD, 1986.

[GiDu 85] Gibson, D. R. , Duce, D. A.
GKS Graphics and Text Processing.
Computer Graphics Forum, N° 4, 1985.

[GKS 84] Special GKS issue.
Computer Graphics - SIGGRAPH-ACM, Février 1984.

[Gnat 84] Gnatz, R.
Approaching a Formal Framework for Software Standards.
Computer & Graphics, Vol 8, n° 1, 1984.

[GoRo 84] Goldberg, A., Robson, D.
Smalltalk-80: The interactive Programming Environment;
Addison-Wesley Publ, 1984

[Grim 86] Grimes, J.P.
A VLSI Implementation of ANSI CG-VDI.
Proc. Computer Graphics' 86, NCGA, Mai 1986.

[Gros 86] Gross, C.
Stations personnelles et accélérateurs graphiques à la norme Phigs chez Apollo.
Electronique Industrielle, n° 105, Avril 1986.

[GSPC 77] Status Report of the Graphics Standard Planning Commitee.
Computer Graphics - SIGGRAPH-ACM, Vol 11, n° 2, Juillet 1977.

[GSPC 79] Status Report of the Graphics Standard Planning Commitee.
Computer Graphics - SIGGRAPH-ACM, Vol 13, n° 3, Août 1979.

[GuTu 76] Guedj, R.A., Tucker, H.A. (Eds.)
Methodology in Computer Graphics.
Proc. IFIP WG5.2 Workshop, SEILLAC I, Mai 1976.
North Holland, Amsterdam (1979).

[HaHe 82] Hatfield, L., Herzog, B.
Graphics Software - from Techniques to Principles.
IEEE - Computer Graphics & Applications, Janvier 1982.

[Heck 86] Heck, M.
PHIGS Hits the Market.
Computer Graphics World, Janvier 1986.

[Herb 83] Herbert, F.
Solid Modelling using a Volume encoding Data Structure.
Computer & Graphics, Vol 7, n° 1, 1983.

[HeRe 85] Herman, I., Reviczky, J.
A General Device Driver for GKS.
Computer Graphics Forum, n° 4, 1985.

[Hind 84] Hindin, H.J.
Graphics standards finally start to sort themselves out.
Computer Design, Mai 1984.

[HJOs 86] Henderson, L., Journey, M., Osland, C.
The Computer Graphics Metafile.
IEEE - Computer Graphics & Applications, Août 1986.

[Hurw 67] Hurwitz, A., Citron, J. P. Yeaton, J. B.
GRAF : graphic additions to FORTRAN.
Proc. 1967 AFIPS Spring Jt Computer Conf, 1967.

[Hvis 79] Hvistendhal, H.
IGL - A structured language for interactive graphics.
EUROGRAPHIC'S 1979, Bologna, Août 1979.

[ISO 85a] GKS, Functionnal Description.
ISO IS 7942, Aôut 1985.

[ISO 85b] GKS-3D Functionnal Description.
ISO, Second D.P 8805, Octobre 1985.

[Jaco 85] Jacob, R.J.K.
A State Transition Diagram Language for Visual Programming.
IEEE Computer, Aôut 1985.

[Jete 86] Jeter, J.P.
Silicon Support for Graphics Standard
proc. NCGA, Mai 1986.

[JaWi 84] Jansen, F.W., van Wijk, J.J.
Previewing Techniques in Raster Graphics.
Computer & Graphics, Vol 8, n° 2, 1984.

[KeRi 84] Kernighan, B.W., Ritchie, D.M.
Le langage C.
Ed. Masson, 1984

[Krze 86] Krzewina, T.
Une étude comparative de PHIGS et GKS.
Actes de la conférence MICAD 86, ed. Hermes, Février 1986.

[KTSa 85] P.Koistinen, M.Tamminen, H.Sammet.
Viewing solid models by bintree conversion
Eurographics'85, Nice, September 1985, 147-157.

[LaNo 84] Lantz, K.A., Nowicki, W.I.
Structured Graphics for Distributed Systems.
ACM Transactions on Graphics, Vol 3, n° 1, Janvier 1984.

[LeCo 83] Lewitt, M., Cohen, B.
Computer graphics needs standards as a foundation for future growth.
Electronics, Fevrier 1983.

[Ledu 77] Leduc-Leballeur, A.
Conception et Réalisation d'un Logiciel Graphique de Base Indépendant de son Contexte d'Utilisation : Application au Logiciel GRIGRI.
Thèse de Docteur-Ingénieur, I.N.P.Grenoble, Septembre 1977.

[LLMa 78] Leduc-Leballeur, A., Lucas, M., Martinez, F.
Conception et Réalisation d'un Logiciel Graphique Interactif Indépendant du Contexte d'Utilisation : le Logiciel de Base GRIGRI.
RAIRO Informatique, Vol 12, n° 2, 1978.

[LuHu 84] Lubinski, T., Hutzel, I.
An Object-Oriented Graphical Kernel System.
Computer Graphics World, Vol 7, N° 7, Juillet 1984.

[Mart 80] Martinez, F.
Clovis : complexe logiciel pour la visualisation
Interactive structurée
Congrès AFCET-TII, Nancy, Novembre 1980.

[Mart 82] Martinez, F.
Vers une approche Systématique de la synthèse d'Images
Aspects logiciels et Matériel
Thèse doctorat d'Etat,
Institut National Polytechnique de Grenoble, Novembre 1982.

[Mart 86] Martinez, F.
Comparaison et évaluation des architectures de systèmes de synthèse d'images.
Revue de C.F.A.O. et d'Infographie, 1 (1), 1986.

[Mart 84] Marty, J. C.
Un editeur graphique pour le système CASCADE : EDICAS
Thèse de 3ème cycle de l'Institu Nationnal Polytechnique de Grenoble, 1984

[MaTh 81] Magnenat-Thalmann, N., Thalmann, D
A graphical pascal extension based on graphical types.
Software (Practice and Experience) 11.53.12 1981

[Mead 84] Meads, J.A.
The Graphics Standards Battle.
Datamation, n° , 1984

[Mead 86] Meads, J.A.
The Standards Pipeline.
Computer Graphics, Juillet 1986.

[Meag 82] Meagher, D.
Geometric Modeling Using Octree Encoding.
Computer Graphics and Image Processing, n° 19, 1982.

[MeNo 84] Mehl, M. E., Noll, S. J.
A VLSI support for GKS.
IEEE - Computer Graphics & Applications, Août 1984.

[Mich 78a] Michener, J. C., Van Dam, A.
A functionnal Overview of the Core System with glossary.
Computing Surveys, Vol 10, n° 4, Décembre 1978.

[Mich 78b] Michener, J. C., Foley, J. D.
Some major issues in the design of the Core graphic system.
Computing Surveys, Vol. 10, n° 4, Décembre 1978.

[MTLa 82] Magnenat-Thalmann, N., Thalmann, D., Larouche, A.
A Multilevel Graphics System based on top-down Methodology.
Computer & Graphics, Vol 6, n° 3, 1982.

[MyBu 86] Myers, B.A., Buxton, W;
Creating Highly-Interactive and Graphical User Interfaces by Demonstration.
Computer Graphics, Proc. SIGGRAPH 86, Vol 20, n° 4, Aôut 1986.

[Myer 84] Myers, B.A.
The User Interface for Sapphire.
IEEE Computer Graphics and Applications, Décembre 1984.

[NeMu 85] Newman, H., Muscatti, Z.
Coding aspects of the virtual device metafile and other picture standards.
Computer & Graphics, Vol. 9, n° 1, 1985.

[Newm 71] Newman, W. M.
Display procedures.
Communication of ACM, Vol 14,N° 10, Octobre 1971.

[Newm 78] Newman, W. M., Van Dam A.
Recent efforts toward graphics standardization.
Computing Surveys, Vol 10, N° 4, Décembre 1978.

[NNSa 72] Newell, M.E., Newell, R.G., Sancha, T.L.
A New Approach to the Shaded Picture Problem
Proc.A.C.M. National Conference, 1972.

[OBEK 84] Olsen, D.R., Buxton, W., Ehrich, R., Kasik, D.J., Rhyne, J.R., Sibert, J.
A Context for User Interface Management.
IEEE Computer Graphics and Applications, Décembre 1984.

[OUTo 85] T.Ohashi, T.Uchiki, M.Tokoro
A three-dimensional shaded display method for voxel-based representation
Eurographics '85, September 1985, 221-232.

[Olen 83] Olenchuk, B.
Graphics Standards
Computer Graphics World, N° 8, Août 1983.

[OlDa 88] Olsen, Dan. Dance, John.
Macros by Example in a Graphical UIMS
IEEE Computer Graphics & Applications, January 1988.

[OlDe 83] Olsen, D.R, Dempsey, E.P.
Syngraph: A Graphical User Interface Generator;
Computer Graphics, July 1983.

[PAGM 88] Peroche, B., Argence, J., Ghazanfarpour, D., Michelucci, D.
La Synthèse d'Images.
Ed. Hermes, Fervrier 1988.

[Pfaf 83] Pfaff, G. E.
The Construction of Operator Interfaces Based on Logical Input Devices.
Acta Informatica, Vol 19, N° 2 , Avril 1983.

[PFAr 86] Powers, T., Frankel, A., Arnold, D.
The Computer Graphics Virtual Device Interface.
IEEE - Computer Graphics & Applications, Août 1986.

[PMKi 85] Pfaff, G.E., Müller, R., Kirsch, B.
A Verifier for Checking the Conformance of Programs with the GKS Standard.
Computers & Graphics, Vol 9, N°1, 1985.

[Prei 78] R.B. Preiss
Storage CRT Display Terminals: Evolution and Trends
Computer, November 1978

[PuMc 86] Puk, R.F., McConnell, J.I.
GKS-3D : A Three-Dimentional Extension to the Graphical Kernel System.
IEEE, Août 1986.

[Reed 85] Reed, Th. N.
Standardization of the Virtual Device Metafile and the Virtual Device Interface
Computer and Graphics, Vol 9, n° 1, 1985.

[Reis 86] Reiss, S.P.
AN Object-Oriented Framework for Graphical Programming.
SIGPLAN Notices, Vol 21, n° 10, Octobre 1986.

[Requ 80] Requicha, A.G.
Reprsentation for Rigid Solids : Theory, Methods and Systems
Computing Surveys, Vol 12, n° 4, Decembre 1980.

[Rix 84] Rix, J.
On Developing a GKS Driver Architecture for Raster Workstations.
Computer & Graphics, Vol 8, n° 2, 1984.

[RoAd 76] Rogers, D.F., Adams, J.A.
Mathématicals Elements for Computers Graphics.
Mc Graw Hill Book Company.

[Rose 86] Rose, C. et Al
Inside Macintosh
Addison Wesley Publ 1986.

[Rose 81] Rosenthal, D.S.H.
Methodology in Computer Graphics Re-examined.
Computer Graphics, Vol 15, n° 2, Juillet 1981.

[Roth 82] S.D.Roth
Ray casting for modelling solids
Computer graphics and Image Processing, n°18, 1982

[RMPK 82] Rosenthal, D. S., Michener, J. C., Pfaff, G. E., Kessener, R., Sabin, M.
The Detailed Semantics of Graphics Input Devices.
Computer Graphics, Proc. SIGGRAPH 82, Vol 16, n° 3, Juillet 1982.

[RuWh 80] Rubin, S.M., Whitted, T.
A 3-Dimensional Representation for Fast Rendering of Complex Scenes.
Computer Graphics, Proc. SIGGRAPH 80,

[SaWe 86] Samet, H., Weber, R. E.
Hierarchical Data Structures.
Semaine Internationale de l'Image électronique, deuxième colloque image.
Nice - Avril 1986.

[Schm 86] Schmucker, K.
Mac App: An Application Framework;
Byte, 1986.

[Scot 84] Scott, G.M.
Pratical implementation of GKS.
Computer Graphics World, Vol 7, N° 12, Décembre 1984.

[Scot 86] Scott, J.
Drivers Free Users From Device Dependencies.
Mini-Micro Systems, Avril 1986.

[SBGS 69] Schumacker,R.A , Brand, B, Gilliland, M, Sharp, W.
Study for Applying Computer Generated Images to Visual Simulation.
Tech. Report No. AEHRL TR 69-14 (AD 700375)
US Air Force Human Resources Lab 1969

[SBMo 86] Shuey, D., Bailey, D., Morrissey, T.P.
PHIGS: A Standard, Dynamic, Interactive Graphics Interface.
IEEE - Computer Graphics & Applications, Août 1986.

[SeMi 85] Sears, K.H., Middleditch, A.E.
Raster Applications with GKS.
Computer-aided design, Vol 17, N° 1, Janvier/Février 1985.

[SHBl 86] Sibert, J.L., Hurley, W.D., Bleser, T.W.
An Object-Oriented User Interface Management System.
Computer Graphics, Proc. SIGGRAPH 86, Vol 20, n° 4, Août 1986.

[Shep 84] Shepherd, B.J.
A Pragmatic Look at GKS and Core.
Computer Graphics World, Février 1984.

[Shep 86] Shepherd, B.J.
Standard Cognizant Graphics System Architecture.
proc. NCGA, Mai 1986.

[Sieb 86] Siebers, G.R.
An introduction to computer graphics.
Computer-aided design, Vol 18, n° 3, Avril 1986.

[Simo 83] Simons, R. W.
Minimal GKS.
Computer Graphics, Proc. SIGGRAPH 83, Vol 17, n° 3, Juillet 1983.

[Skal 86] Skall, M.W.
NBS's Role in Computer Graphics Standards
IEEE Computer Graphics & Applications, 1986.

[Smit 71] Smith, D. N.
GPL/I - A PL/I extension for computer graphics.
Proc. 1971 AFIPS Spring Jt. Computer Conference.

[Sond 84] Sonderegger, E. L.
The Case for Core System Standardization.
Computer Graphics World, Fevrier 1984.

[SpGa 86] Sparks, M.R., Gallop, J.R.
Language Bindings for Computer Graphics Standards.
IEEE Computer Graphics & Applications, 1986.

[Spie 86] Spiers, R. G.
Realization and Application of an Intelligent GKS Workstation.
IEEE - Computer Graphics & Applications, Mai 1986.

[Srih 81] Srihari, S.N.
Representation of Three-Dimensional Digital Images.
Computing Surveys, Vol 13, n° 4, Décembre 1981.

[SSSc 74] Sutherland, I.E., Sproull, R.F., Schumacker, R.A.
A Characterization of Ten Hidden-Surface Algorithms.
Computing Surveys, Vol 6, n° 1, Mars 1974.

[Stol 84] Stoll, C.
GKS for Imaging.
Computer Graphics, Proc. SIGGRAPH 84, Vol 13, n° 3, Juillet 1984.

[Sun 86] Sun Microsystem. 1986
Sun CORE Reference Manual

[Suth 63] Sutherland, I.E.
SKETCHPAD : A Man-Machine Graphical Communication System.
AFIPS Cont. Proc., Vol 23, SJCC 1963.

[Suyd 85] Suydam, W.E.
Graphics Standards Emerge as Firms strive for Viability.
Computer Design, Decembre 1985.

[Stra 85] Straayer, D.H.
The Standardization of three-dimensional graphics systems.
Computer & Graphics, Vol. 9, n° 1, 1985.

[Vand 87] Vanderbroucke, L.
Computer Graphics from the Seventies to the Nineties
EUROGRAPHICS'87, Septembre 1987.

[Wagg 84] Waggoner, C.N.
The GKS Advantage.
Computer Graphics World, Vol 7, nº 10, Octobre 1984.

[Warn 81] Warner, J. R.
Principles of device-independent computer graphics software.
IEEE - Computer Graphics & Applications, Octobre 1981.

[Warn 82] Warner, J. R.
A discussion of graphic software standards.
Computer graphics World, n° 8, 1982.

[Warn 84] Warner, J. R.
Device-Independent Tools Systems.
Computer Graphics World, Février 1984.

[Warn 85] Warner, J.R.
Standard Graphics Software For High Performance Applications.
IEEE - Computer Graphics & Applications, Mars 1985.

[Wass 85] Wasserman, A.
Extending State Transition Diagrams for the Specification of Human-Computer Interaction
IEEE Transactions on Software Engineering, August 1985.

[Will 85] Williams, T.
Graphics Standards Gain Ground Slowly
Computer Design, Avril 1985.

[Will 85] Williams, T.
Graphics Processing Migrates from Host to Workstation.
Computer Design, Juillet 1985.

[Wiss 85] Wisskirchen, P.
Towards Object-Oriented Graphics Standards.
EUROGRAPHICS'85.

[Wrig 84] Wright, T.
Is GKS powerful enough for the application ?
Computer Design, Mai 1984.

[Wrig 84] Wright, T.
GKS Versus Core.
Computer Graphics World, Février 1984.

[WTNe 84] Waggoner, C.N., Tucker, C., Nelson, C.J.
Nova GKS, A Distributed Implementation of the Graphical Kernel System.
Computer Graphics, Vol 18, N° 3, Juillet 1984.

[WWGr 85] Westrup, T.D., Kegel, W., Gras, J.
User Interaction with an Environment for Image Processing and Graphics.
Computer Graphics Forum n° 4, 1985.

[YaSr 83] Yau, M.M., Srihari, S.N.
A Hierarchical Data Structure for Multidimensional Digital Images.
Communications of the ACM, Vol 26, n° 7, Juillet 1983.

[Zara 85] Zarate Silva, V. H.
Etude et Réalisation d'un synthétiseur d'images basé sur une architecture banalisée.
Thèse de Docteur Ingénieur de l'Institut National Polytechnique de Grenoble, Novembre 1985.

[Zimm 87] Zimmerman, H.
Qui fait quelles normes ?
01 Informatique Magazine, Février 1987.

ANNEXE 1 : L'analyse des expressions dans CLOVIS

1 Le rôle de l'analyse

Nous présentons ici le détail de l'analyse des expressions utilisées dans les primitives de structuration de CLOVIS (construction de structure, définition d'entité logique) que nous avons évoquée au chapitre V.

Dans le second système qui fait l'objet de la troisième partie de ce rapport, interviennent également des expressions alphanumériques pour la définition et la manipulation de structures hiérarchiques. Celles-ci sont très proches de celles utilisées pour CLOVIS, et leur analyse ne diffère que sur quelques points de détail par rapport à celles que nous étudions ici.

Nous rappelons que l'analyse des expressions (de création ou de référence) remplit plusieurs rôles :

- elle vérifie leur syntaxe,
- elle vérifie leur sémantique en fonction de leur contexte d'utilisation,
- dans le cas d'expressions de référence, elle assure la génération dans les structures de l'unité de communication des séquences d'accès aux noeuds correspondants à la chaîne étudiée,
- dans le cas d'expressions de création, elle **construit** effectivement la structure définie.

2 Réalisation de l'analyse

L'analyse des différentes expressions est réalisée à partir d'un automate d'états finis dont le diagramme de transition est représenté sur la figure V.14.

Les différentes entités extraites de la chaîne étudiée et reconnues par l'analyseur sont mentionnées sur les arcs indiquants les transitions d'états. Celles ci sont :

- soit des symboles terminaux : [,], (,), >, ., ,, +, -
- soit un identificateur de noeud (< lettre > { < lettre > | < chiffre > }) représenté par "idtf",
- soit un entier non signé correspondant à une borne d'indiçage et représenté par "ent".

- Le symbôle % indique la fin de la chaîne à étudier.

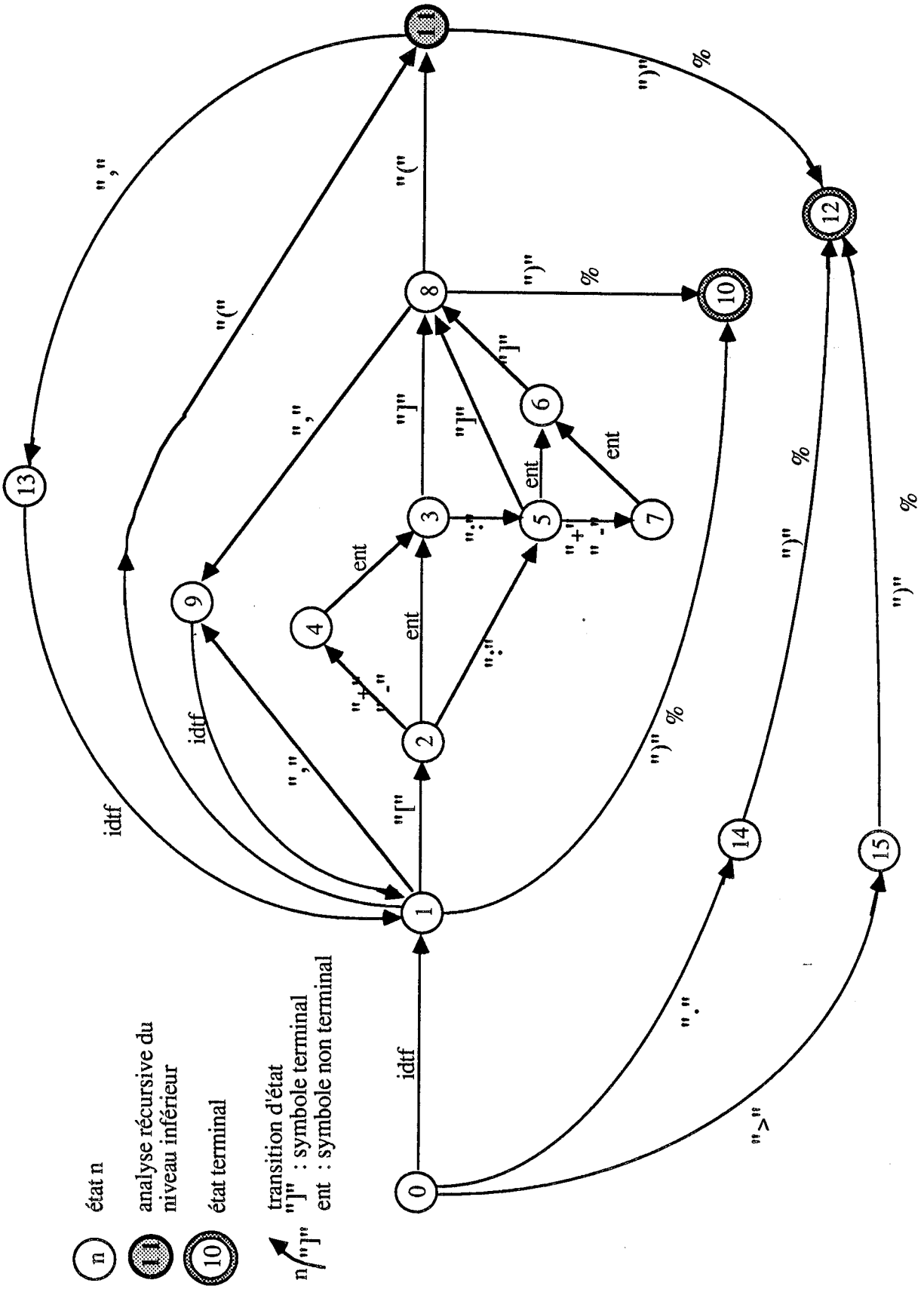
Comme on peut le constater sur la figure A.1, l'analyse des différentes expressions s'effectue de manière récursive. Ainsi l'automate représente l'analyse de la partie de la chaîne correspondant aux éléments d'un même niveau, l'étude d'un niveau inférieur est réalisée par un appel récursif du programme d'analyse (état 11).

D'autre part l'automate réalise la synthèse des différents types d'expressions (de création, de référence simple ou multiple). Cela permet d'avoir un seul programme d'analyse et d'éviter la duplication de code, même si cela introduit quelques tests suplémentaires pour vérifier la validité de certaines transitions selon la nature de l'expression à analyser.

Ainsi seront interdites les transitions d'états suivantes :

- pour les expressions de création :

état 0 ----> état 14, état 15
état 8 ----> état 9 (au niveau 0 d'analyse uniquement)



- figure A.1 : L'automate d'Analyse des expressions -

- pour les expressions de référence simple :

état 0 ----> état 14, état 15
état 1 ----> état 9
état 8 ----> état 9
état 11 ----> état 13

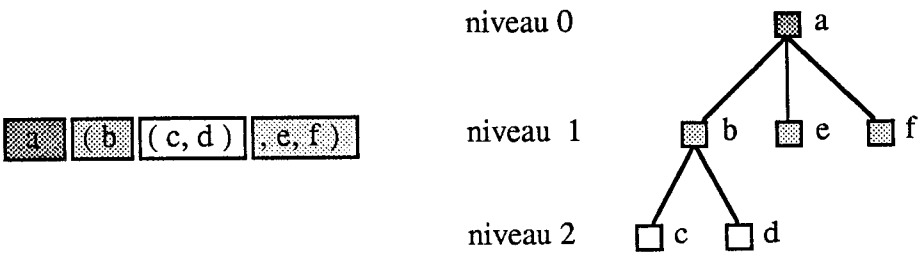
Dans ce qui suit nous présentons la procédure d'analyse réccursive des différents niveaux d'une expression, avec une explication détaillée des traitements effectués pour les principaux états de l'automate. Celle-ci est réalisée à l'aide de la primitive

ANALYSE_NIV(chaîne, type_expr, niv, ideb) --> err

-*chaîne* contient l'expression alphanumérqiue à analyser.

- *type_expr* défini le type de l'expression qui peut être soit une expression de création (*type_expr* = CREA), soit une expression de référence simple (*type_expr* = RSIMPLE) ou multiple (*type_expr* = RMULTI).

- *niv* et *ideb* sont des variables destinées aux appels successifs de ANALYSE_NIV et qui indiquent respectivement le niveau d'expression analysée (figure A.2), et la position dans *chaîne* du début de ce niveau (à l'appel initial niv = 0 et ideb = 1).



- figure A.2 : Les différents niveaux d'une expression -

ANALYSE_NIV est une fonction qui renvoie un code erreur *err*, entier qui indique toute erreur de syntaxe ou de sémantique détectée lors de l'étude du niveau. Si l'analyse du niveau s'est déroulée correctement, la valeur retournée est 0 (voir table A.1 p A1.7).

a) présentation des principaux états

Les traitements effectués pour les principaux états de l'automate d'analyse définis sur la figure A.1 sont :

état 0 :

Cet état correspond au début d'un niveau d'expression. Le programme d'analyse utilise la pile PILE (définie pour le parcours de sous-structure) afin de conserver le noeud courant, père du niveau sur lequel porte l'analyse. Ainsi au début de l'analyse d'une expression de référence (analyse du niveau 0), le noeud courant sommet de PILE est le noeud correspondant au contexte courant. Dans le cas d'une expression de création, lors du premier appel de l'analyseur ce noeud courant est indéfini (NIL), la structure créée étant indépendante de la structure existante.

état 1 :

L'identificateur récupéré est stocké dans une variable NOMCOUR. L'analyseur effectue alors une recherche parmi les fils du noeud courant afin de déterminer le(s) noeud(s) dont le nom est identique à NOMCOUR. Les adresses de ces noeuds sont conservées dans une liste (LISTREF : liste des noeuds du niveau référencés par NOMCOUR).

Dans le cas d'une expression de référence si la recherche échoue (LISTREF vide) l'analyse est stoppée et un message d'erreur signale que l'identificateur ne désigne aucun élément existant de la structure hiérarchique.

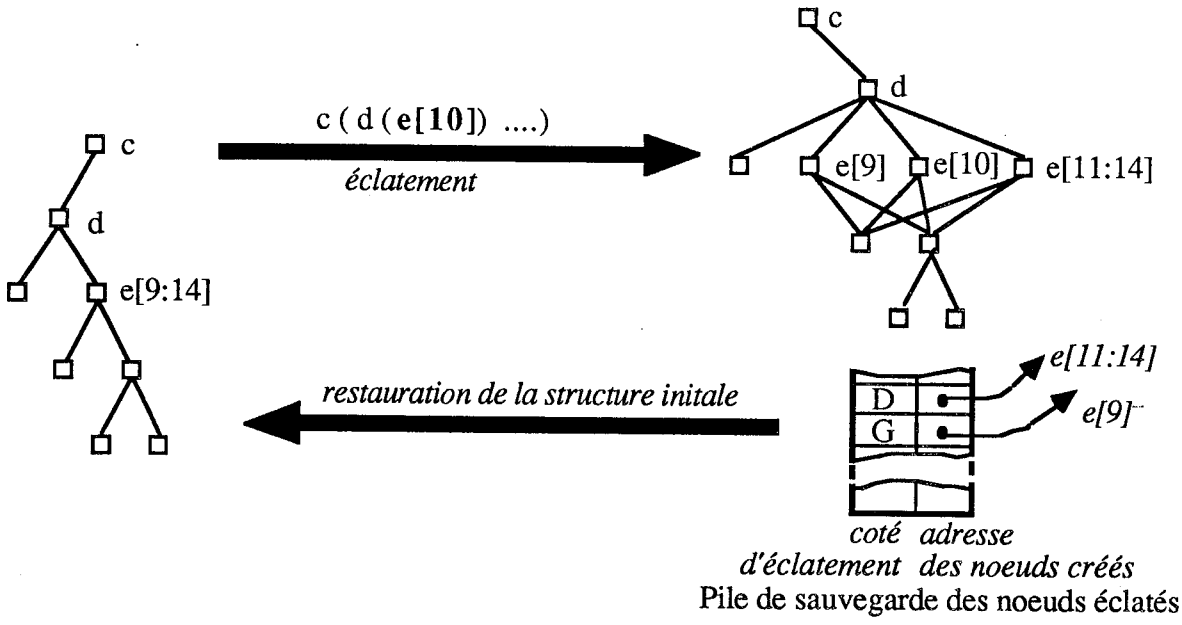
états 2,3,4,5,6,7 :

Ces différents états correspondent à la récupération éventuelle des bornes d'indiçage associées à l'identificateur NOMCOUR. Celles ci sont stockées dans les variables IDEB et IFIN. (Des tests sont effectués sur la validité de ces indices, par exemple IDEB <= IFIN).

état 8 :

Le symbôle qui provoque l'entrée dans cet état est le caractère "]" qui coïncide avec la terminaison de l'acquisition d'un identificateur indicé. L'analyseur recherche le(s) éventuel(s) élément(s) du niveau désignés par NOMCOUR[IDEB,IFIN]. Cette recherche est effectuée dans LISTREF initialisée à l'état 1 et qui contient les adresses de tous les noeuds du niveau dont l'identificateur est NOMCOUR. Parmi ces noeuds, seuls sont conservés ceux dont les bornes d'indiçage intersectent l'intervalle [IDEB,IFIN].

Dans le cas d'une expression de référence, la désignation NOMCOUR[IDEB,IFIN] peut éventuellement conduire à l'éclatement d'un noeud si ses bornes d'indice *ideb,ifin* sont telles que [*ideb,ifin*] ∩ [IDEB,IFIN] ≠ ∅ et [*ideb,ifin*] ⊈ [IDEB,IFIN]. L'éclatement de structure est effectivement réalisé à ce stade. Toutefois afin de restaurer la structure dans son état intial en cas d'erreur ultérieure dans l'analyse, on conserve dans une pile l'adresse des noeuds instance ainsi crées (voir fig. A.3). Par ailleurs, avant de procéder à tout éclatement, on vérifie que le noeud à "éclater" ne fait pas partie d'une autre entité logique (de visualisation si l'expression concerne le travail ou inversement). Pour ce faire on vérifie le champ *typ_log* du descripteur du noeud, et si l'élément est marqué comme composant d'une entité logique l'éclatement n'a pas lieu et l'analyse est interrompue.



- figure A.3 : Eclatement des noeuds lors de l'analyse -

A la fin du traitement de l'état 8, LISTREF contient la liste des noeuds référencés par NOMCOUR[IDEB,IFIN]. Dans le cas d'expressions de référence, si cette liste est vide l'analyse est interrompue comme pour l'état 1.

Remarque : Si suite à l'état 1, on ne passe pas par les étapes 2,3,4,5,6,7 et 8, NOMCOUR est un indentificateur non indicé, lui sont associées alors les bornes d'indiçage par défaut BORNEMIN et BORNEMAX.

état 9 :

Cet état n'est autorisé que pour les références multiples ou les expressions de création. Son symbole d'entrée est ",", l'identificateur (éventuellement indicé) acquis lors des étapes précedentes correspond donc à un élément "terminal" de l'expression à analyser.

Pour les expressions de référence, avant de générer la séquence d'accès définie par NOMCOUR[IDEB,IFIN] (accès de type FEU pour chacun des noeuds de LISTREF) l'analyseur vérifie que cet identificateur n'interfère pas avec ceux déjà traités pour le niveau. (C'est à dire que NOMCOUR[IDEB,IFIN] ne désigne pas un(des) élément(s) référencé par la partie de l'expression déjà analysée). C'est pourquoi l'analyseur conserve une liste des noeuds déjà traités pour le niveau, que l'on peut alors comparer avec celle des éléments référencés par NOMCOUR[IDEB,IFIN] (LISTREF).

Pour les expressions de création, cette vérification de l'interférence de l'identification du noeud à traiter avec les noeuds déjà créés pour le niveau s'effectue plus simplement en testant LISTREF. Si celle-ci est vide, le noeud est alors créé, sinon l'analyse est interrompue.

état 10 :

Cet état termine le traitement d'un niveau, le niveau 0 si le symbôle d'entrée est "*%*", un niveau traité récursivement si le symbôle d'entrée est ")". L'expression analysée est du type *.....(....identificateur)....* ou *.....identificateur%*. A ce niveau, il est donc nécessaire de traiter l'identificateur (qui peut être indicé) acquis durant les étapes précédentes.

Pour les expressions de création ou de référence multiple, ce traitement est en tous points identique à celui de l'état 9 présenté ci dessus.

Dans le cas d'une expression de référence simple, un test supplémentaire est réalisé afin de vérifier que NOMCOUR[IDEB,IFIN] correspond bien à un et un seul élément de la structure hiérarchique (LISTREF établie au états 1 et/ou 8 contient un seul élément).

état 11 :

Le symbôle d'entrée est le caractère "(" : on débute un nouveau niveau. Avant de l'analyser il convient de traiter l'identificateur acquis aux étapes précédentes (l'expression est de type *....NOMCOUR(....* ou *NOMCOUR[IDEB,IFIN](....)*.

Dans le cas d'une expression de référence, le traitement est le suivant :

```
pour chaque noeud de LISTREF
   GENERE(VISU/TRAV,noeud,DSC)    (*Générer dans PILEPAR un accès
                                      noeud de type DSC*)
   EMPILE(noeud),
   Analyser récursivement le niveau inférieur débuté par "(",
   noeud = DEPILE()
   GENERE(VISU/TRAV,noeud,ASC)  (*Générer dans PILEPAR un accès
                                    noeud de type ASC*)
fin pour
```

En ce qui concerne les expressions de création, le traitement s'effectue comme suit :

```
Créer le noeud correspondant à l'identificateur courant (NOMCOUR[IDEB,IFIN]),
Le chaîner dans la liste des éléments du niveau (fils du noeud courant),
EMPILE(noeud)  (*Empiler le noeud créé*),
Analyser récursivement le niveau inférieur débuté par "(",
noeud = DEPILE().
```

La detection des erreurs est analogue à celle de l'état 10 vu précédemment.

état 12 :

Cet état correspond à la terminaison d'un niveau (le symbôle d'entrée est le caractère "%" ou le caractère ")"), mais, à la différence de l'état 10, il ne comporte aucun traitement. En effet, l'expression est du type *identificateur(.....)%* ou *.....(..... identificateur(.....))*..... ; le dernier identificateur du niveau, c'est à dire celui qui précédait le sous niveau dont l'analyse vient de se terminer, a déjà été traité.

Il s'agit donc d'un état terminal dont le rôle est uniquement de dépiler les appels récursifs de l'analyseur.

état 14 :

Cet état n'est valide que pour les références multiples. Son symbôle d'entrée, le caractère ".", désigne tous les noeuds d'un niveau. Son traitement consiste à rechercher le premier fils du noeud courant et à générer un accès à celui-ci de type NIV.

état 15 :

Cet état correspond au traitement du symbôle ">" (valide lui aussi uniquement pour les références multiples) qui référence toute une sous arborescence. Celui-ci s'effectue de manière simple : l'analyseur génère un accès de type SSA au noeud courant père de cette sous structure.

b) les erreurs détectées au cours de l'analyse

Comme on peut le constater de très nombreux contrôles sont effectués lors de l'analyse des expressions. La table A.1, ci -contre, donne une liste exhaustive des erreurs ainsi détectées par l'analyseur.

En cas d'erreur lors de l'analyse d'une expression CLOVIS conserve la position de début et la position de fin dans la chaîne de l'entité ayant provoqué l'erreur. L'application pourra ainsi, si elle le désire, récupérer ces valeurs pour effectuer son propre traitement d'erreur (par exemple impression de l'expression avec la sous-chaîne incriminée en surbrillance).

3 Utilisation de l'analyse par les primitives de structuration

La primitive ANALYSE_NIV que nous venons de présenter n'est pas directement utilisée par les primitives de structuration (création de structure, définition d'entités logiques) qui nécessitent l'analyse des expressions qu'elles utilisent. Pour cela, elles passent par une primitive intermédiaire

ANALYSE(chaîne, typ expr) --> err

dont le rôle supplémentaire est l'initialisation des structures de données globales à l'analyse (en particulier la table de sauvegarde des noeuds éclatés) et la restauration éventuelle de la structure initiale en cas d'erreur.

Ce second point concerne la suppression des noeuds instances qui ont éventuellement été créés lors d'éclatement (cf. état 8 de l'automate) et la remise des bornes d'indiçage des noeuds à leur valeur de départ. Pour cela, chaque noeud de la pile PECLAT est détruit. Afin de restituer la valeur initiale des indices du noeud "éclaté", on conserve dans PECLAT l'indication du côté d'éclatement, c'est à dire si le noeud instance a été inséré à gauche ou à droite du noeud original. Ainsi, si l'éclatement a eu lieu gauche, avant de supprimer le noeud instance, on remplace la valeur de la borne d'indice inférieure du noeud situé à sa droite par celle du noeud à détruire. Le traitement de l'éclatement à droite s'effectue de manière totalement symétrique.

NATURE DE L'ERREUR	Type d'expression	Etat(s) de détection
Identificateur (nom) de noeud inconnu pour le niveau.	RS, RM	1
Indiçage incorrect IDEB > IFIN.	RS, RM, CR	6
Il n'existe pas d'élément dont les bornes d'indiçage correspondent à celles associées à l'identificateur courant.	RS, RM	8
Eclatement de structure impossible, il invalide une entité logique	RS, RM	8
Plusieures références à un même noeud à un même niveau.	RM	9, 10, 11
Création impossible, noeud déjà existant.	CR	9, 10, 11
Référence à plusieurs noeuds alors que référence simple requise.	RS	9, 10, 11
"," interdite dans ce type d'expression.	RS, CR	9, 13
"." interdit dans ce type d'expression.	RS, CR	14
")" et "." interdits au même niveau.	RM	13
">" interdit dans ce type d'expression	RS, CR	15
Symbole non autorisé	RS, RM, CR	-1

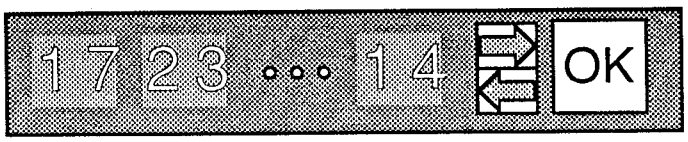
RS : référence simple, RM : référence multiple, CR : expression de création

- table A.1 : Les Erreurs détectées lors de l'Analyse -

ANNEXE 2 : Un exemple d'objet d'interface : Les incrementeurs de valeurs

1 Présentation générale des incrémenteurs de valeurs

Les incrementeurs de valeurs permettent de définir graphiquement et interactivement un ensemble de valeurs entières. Ces objets d'interface se présentent sous la forme suivante (fig. A.2.1):

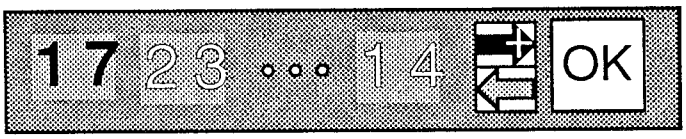


- figure A.2.1 : Incrémenteur de valeurs -

La liste des valeurs entières à définir est affichée, le nombre de ces valeurs est quelconque et est fixé par l'application à la définition de l'incrémenteur. Ces valeurs peuvent varier chacune dans un intervalle donné (défini par l'application), c'est la valeur courante qui est affichée.

Les flèches, sont des "boutons incrémenteurs" qui permettent de faire varier l'une des valeurs précédentes à l'intérieur de son intervalle de définition.

Pour modifier une valeur, l'opérateur la sélectionne en "cliquant" sur celle-ci. Elle devient alors active (un changement de couleur le signale); chaque fois que l'opérateur clique sur l'une des flèches, cette valeur est incrémentée ou décrémentée d'un pas donné (fig. A.2.2). La nouvelle valeur mise à jour est affichée à chaque étape. La sélection d'une valeur a pour effet de désactiver la valeur précédemment active (la restitution de sa couleur initiale le signale).



- figure A.2.2. Modification d'une valeur -

Lorsque l'utilisateur a défini toutes les valeurs qui l'intéressent, il termine la séance d'interaction en cliquant sur le "bouton" OK.

2 Primitives de manipulation des incrémenteurs de valeurs

Les incrémenteurs de valeurs sont des objets de dialogue évolués manipulés au travers d'un jeu de primitives de haut niveau analogue à celui que nous avons présenté au chapitre VI. Nous donnons ci-dessous, l'ensemble des primitives spécifiques aux incrémenteurs.

a) définition

```
refer  Def_IncrVal(refdom,refcadre,xg,yg,descr,expval,expsel)
refr  refdom,refcadre;
int  xg,yg;
char  *descr;
int  expval,expsel;
```

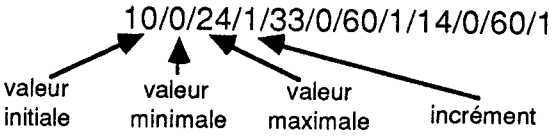
Cette primitive permet la définition d'un incrémenteur de valeurs.

refdom est la référence du domaine père auquel l'incrémenteur de valeurs est rattaché.

refcadre est la référence du cadre permettant éventuellement de délimiter la zône utile du domaine définissant l'incrémenteur de valeurs.

xg,yg sont les coordonnées (dans le repère du domaine père) du coin inférieur gauche du domaine associé à l'incrémenteur de valeurs.

descr est une chaîne de caractères qui défini l'ensemble des valeurs qui seront manipulées à l'aide de l'incrémenteur. Pour chacune de celles-ci sont indiquées dans l'ordre une valeur intiale, une valeur minimale, une valeur maximale, un incrément. Les différentes valeurs sont séparées par un caractère qui est le /. Par exemple pour construire un incrémenteur permettant de spécifier interactivement une heure, on utilise trois valeurs qui indiquent respectivement les heures, les minutes et les secondes. Celles-ci sont définies avec le descripteur suivant :



expval, expsel sont les numéros de expressions de couleur utilisées pour afficher les entités de l'incrémenteur (les valeurs et les flèches d'incrémentation-décrémentation). *expval* sert pour les entités valides, *expsel* concerne les entités sélectées (la valeur active et la flèche en cours d'utilisation).

b) modification

```
Maj_IncrVal(ref,no_val,vi,vmin,vmax,increm)
refer ref;
int no_val,vi,vmin,vmax,increm;
```

Cette primitive permet de modifier l'une des valeurs associées à l'incrémenteur.

ref est la référence de l'objet incrémenteur.

no_val est le numéro de la valeur à modifier.

vi,vmin,vmax sont respectivement la nouvelle valeur initiale, la nouvelle valeur minimale et la nouvelle valeur maximale.

increm est le nouvel incrément qui sera utilisé pour modifier cette valeur.

Si l'incrementeur est affiché lorsque cette primitive est invoquée, l'image est automatiquement mise à jour.

```
Asp_IncrVal(ref,expval,expsel)
refer ref;
int expsel,expval;
```

permet de modifier les aspects d'un incrémenteur.

ref est la référence de l'objet incrémenteur.

expval est le numéro de l'expression de couleur utilisée pour afficher les entités valides (valeurs non selectées et flèches non actives).

expval est le numéro de l'expression de couleur utilisée pour afficher les entités selectées.

L'appel de cette primitive ne provoque pas de modification de l'image. Si l'incrémenteur est affiché, il doit être explicitement revisualisé.

c) consultation

```
Cons_Incrval(ref,no_val,vcour,vmin,vmax,increm)
refer ref;
int  no_val,*vcour,*vmin,*vmax,*increm;
```

Cette primitive permet à l'application de récupérer les valeurs associées à un incrémenteur.

ref est la référence de l'objet incrémenteur.

no_val est le numéro de la valeur à consulter. (Les valeurs sont numéroté de 1 à N selon leur ordre de définition dans le descripteur utilisé lors de la définition de l'incrémenteur).

vcour est sa valeur courante.

vmin et ***vmax*** définissent son intervalle de variation et ***increm*** l'incrément associé

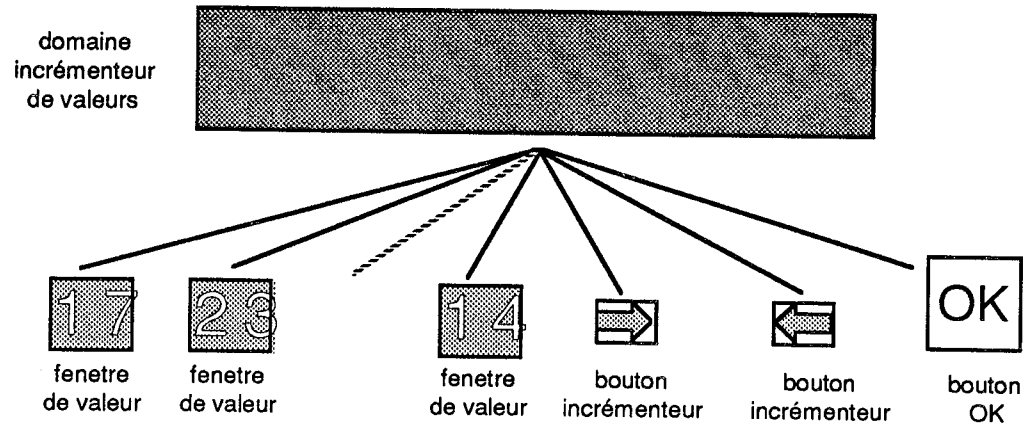
3 Exemple d'utilisation d'un incrémenteur de valeurs

Nous donnons ici, un exemple d'utilisation, au travers des ses primitives de manipulation, d'un objet d'interface du type incrémenteur de valeurs. Dans cet exemple, l'incrémenteur de valeur sert à définir une horloge et permet de positionner interactivement les heures, les minutes et les secondes.

```
refer  refh;  /* réference de l'objet de dialogue horloge */
int expsel,expval; /* no des expressions de couleurs */
int hcour,mcour,scour; /* pour la récupération de l'heure courante */
...
...
ExpCoul(expval,C_Ref,32,128,0);
ExpCoul(expsel,C_Ref,240,48,0);

/* définition de l'horloge */
refh = Def_IncrVal(Image0,NoCadre,xg,yg,"12/0/24/1/30/0/60/1/40/0/60/1,expval,expsel);
...
AffVue(refh,Tout+Presse); /* affiche l'horloge sur l'image 0 */

Seance(refh); /* séance de dialogue implicite sur l'horloge */

Cons_IncrVal(refh,1,&hcour,&hmin,&hmax,&incrh); /* consultation de l'heure */
Cons_IncrVal(refh,1,&mcour,&mmin,&mmax,&incrm); /* consultation des minutes*/
Cons_IncrVal(refh,1,&scour,&smin,&smax,&incrs); /* consultation des secondes*/

EffVue(refh); /* efface l'horloge */
....
```

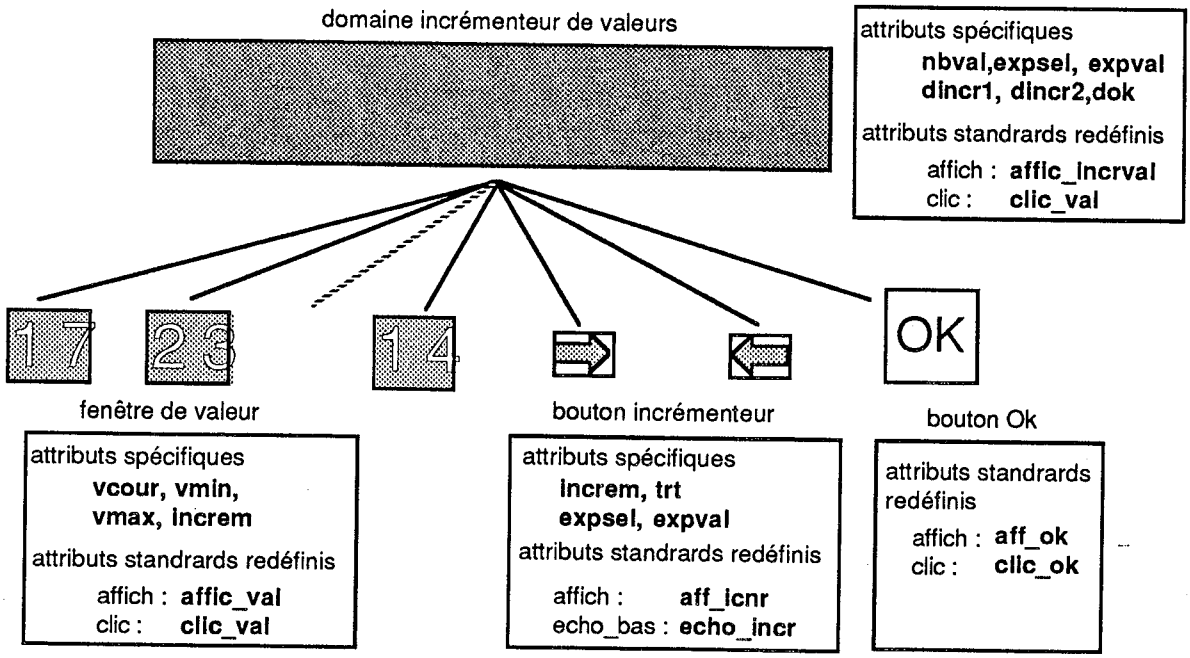
4 Implémentation des incrémenteurs de valeurs

Les incrémenteurs de valeurs sont des objets d'interface évolués définis à l'aide d'une hiérarchie de domaines que nous étudions dans ce qui suit (fig. A.2.3).



- figure A.2.3 : hiérarchie des domaines définissant un incrémenteur de valeurs -

Un incrémenteur de valeurs est défini en regroupant en une seule entité (à l'aide d'une hiérarchie de domaines) plusieurs objets de dialogues. Le lecteur trouvera dans les pages qui suivent, les programmes permettant la définition d'un tel objet d'interface, c'est à dire réalisant les primitives pour sa manipulation telles que nous les avons présentées au § 2. Pour une meilleure compréhension de ces programmes la figure A.2.4, ci-dessous, indique les différents attributs particularisant chacun des différents domaines regroupés dans un incrémenteur de valeurs.



- figure A.2.4 : attributs des domaines d'un incrémenteur de valeurs -

Le **bouton OK** qui est un objet d'interface élémentaire. Ses attributs sont les attributs standards des domaines (mis à jour par sa fonction d'affichage ***aff_ok*** qui lui est propre et qui visualise le texte **OK**). Le seul attribut qui le particularise, du point de vue du dialogue, est sa fonction ***clic*** qui quand elle est activée (changement d'état du dispositif de dialogue sur le domaine du bouton OK) provoque la fin de la séance de dialogue en cours.

Les **boutons incrémenteurs** sont des objets d'interface particularisés à l'aide des attributs suivants :

increm qui est un entier et qui est le pas d'incrémentation associé au bouton,

expval,expsel sont des numéros d'expression de couleur, qui sont utilisés pour l'affichage des flèches du bouton incrémenteur. ***expsel*** sert quand le dialogue et actif sur le domaine (dispositif en position basse), ***expval*** est employé sinon.

Parmi les attributs standards du domaine, seuls sont redéfinis la fonction d'affichage et la fonction pour l'écho en position basse; tous les autres attributs ont les valeurs par défaut.

aff_incr est la fonction d'affichage du bouton incrémeteur. En particulier cette fonction dessine la flèche vers la gauche ou la droite selon le signe de l'incrément *increm.*

echo_incr est la fonction d'écho automatiquement invoquée quand le dispositif est en position basse sur le domaine du bouton incrémenteur. Elle effectue les traitements suivants :

- lors de son premier appel, elle affiche la flèche de l'incrémenteur avec la couleur définie par l'expression *expsel.* De manière symétrique, lors du dernier appel, elle restitue à la flèche sa couleur initiale définie par l'expression *expval.*

- chaque fois que cette fonction d'écho est appelée, elle exécute une fonction de traitement dont l'adresse est conservée dans l'attribut***trt*** associé au domaine du bouton incrémenteur. Cette fonction est en général définie au niveau du domaine père, elle permet de répercuter les actions effectuées sur le bouton incrémenteur à d'autres entités. Dans le cas de notre incrémenteur de valeur, elle se charge de la mise à jour et de l'affichage de la valeur active. Cette fonction de traitement est invoquée avec pour paramètres l'étape du dialogue (*Premier, Interm, Dernier*) et la valeur de l'incrément (*increm*) associé au bouton.

Les boutons OK, et les boutons incrémenteurs définissent deux classes d'objets de dialogue qui peuvent être manipulés au travers d'un ensemble de primitives de haut niveau similaires à celles que nous avons présentées précédemment pour les incrémenteurs de valeurs. Ils ne sont pas spécifiques aux incrémenteurs de valeurs et peuvent être directement utilisés par une application (c'est le cas en particulier des boutons OK), ou servir à construire d'autres objets de dialogue évolués. Le lecteur trouvera à partir de la page A.2.7 le listing des programmes C (modules **Ok.c** et **Increm.c**) qui réalisent les primitives de manipulation propres à ces objets de dialogue à partir des primitives standards de définition de domaine que nous avons présentées au chapitre VI §4.2.

Les **fenêtres de valeurs** sont des domaines définis de manière spécifique pour les incrémenteurs de valeurs. Les primitives pour leur manipulation sont définies dans le même module que celui implémentant les primitives de manipulation des incrémenteurs de valeurs, et ne sont accessibles qu'à l'intérieur de celui-ci (module **incr_val.c** p. A.2.10).

Au domaine fenêtre de valeurs sont associés les attributs suivants :

val,vmin,vmax qui définissent la valeur courante et son intervalle de variation,

increm l'incrément associé à cette valeur,

Les attributs standards du domaine, ont tous les valeurs par défaut sauf, la fonction d'affichage et la foncton clic qui sont respectivement :

aff_val qui permet de visualiser la valeur courante,

clic_val qui, quand une transition descendente est effectué sur le domaine, permet :

- d'afficher la valeur avec la couleur définie par l'attribut *expsel* situé au niveau du domaine père (domaine incrémenteur de valeur), (si une valeur était déjà selectée, cette dernière est réaffichée avec la couleur défine par l'attribut *expval* du domaine incrémenteur de valeur).

- d'initialiser les boutons incrémenteurs avec la valeur +- *increm.*

La fonction de traitement associée aux boutons incrémenteurs est invoquée par la fonction d'écho sur ces domaines(***echo_incr***), est ***trt_val*** qui en fonction de l'incrément du bouton activé, modifie la valeur courante attachée à la fenêtre de valeur selectée et la réaffiche.

Au niveau du domaine racine de la hiérarchie définissant les objets de type incrémenteurs de valeurs les attributs spécifiques définis sont les suivants :

nbval : le nombre de valeurs regroupées dans l'incrémenteur,

expval,expsel : sont des numéros d'expressions de couleurs, *expsel* est utilisée pour particulariser le bouton incrémenteur et la valeur actifs, *expval* est employée pour les autres entités.

dincr1 ,dincr2 sont les références des deux "boutons incrémenteurs" rattachés à ce domaine. Leur mémorisation explicite permet d'accéder directement à ceux-ci, en particulier dans la fonction ***clic_val*** invoquée quand une valeur est selectée.

dok est la réference du "bouton Ok" rattaché au domaine incrémenteur de valeurs.

Tous les attributs standards du domaine ont les valeurs par défaut, sauf bien entendu la fonction d'affichage ***affich*** qui ici est la fonction ***aff_incrval*** qui assure la visualisation de toute la hiérarchie des domaines.

```
/*****************************************************************/
/* MODULE       : ok.h                                           */
/* ROLE         : constantes predefinies et structures de donnees */
/*                concernant les boutons OK                      */
/*****************************************************************/

/*****************************/
/* Descripteur de domaine OK */
/*****************************/

typedef struct
{
  #include <infodom.h>  /* partie standard des descripteurs de domaine */
} dok;

/**********/
/* Macros */
/**********/

/* conversion reference domaine OK --> adresse du descripteur */
#define adrok(ref)   ((dok *) *(_banque + ref))
```

```
/*****************************************************************/
/* MODULE       : increm.h                                       */
/* ROLE         : constantes predefinies et structures de donnees */
/*                concernant les BOUTONS INCREMENTATION          */
/*****************************************************************/

/************************************************/
/* Descripteur de domaine BOUTON INCREMENTATION */
/************************************************/

typedef struct
{
  #include <infodom.h>/* partie standard des descripteurs de domaine */
  int incr;           /* valeur de l'increment */
  int (*trt)();       /* adresse de la procedure de traitement */
  int expval,expsel; /* expressions de couleur utilisees */
} dincr;

/**********/
/* Macros */
/**********/

/* conversion reference domaine --> adresse du descripteur */
#define adrincr(ref)   ((dincr *) *(_banque + ref))
```

```
/*****************************************************************/
/* MODULE       : ok.c                                           */
/* ROLE         : Actions de base concernant les boutons Ok      */
/*****************************************************************/

/****************/
/* IMPORTATIONS */
/****************/

/*---- importations de modules ------*/
#include <stdio.h>

#include <getlib.h>   /* constantes de la bibliothèque graphique */
#include <gcte.c>
#include <util.h>
#include <dial.h>
#include <dom.h>

#include "ok.h"       /* descripteur de bouton OK */

/*************************************************/
/* Primitives de gestion interne des boutons OK */
/*************************************************/

/*--------- affichage d'un bouton OK -----------*/
static aff_ok(ref,mode)
refer ref;
int mode;
{
    dok *pok;

    pok = adrok(ref);
    if (mode & Entites)
    { DebEcr(ref);
        Couleur(192,240,224);
        Rectangle(0,0,pok->xd - pok->xg,pok->yd - pok->yg);
        Couleur(0,0,0);
        AjustTexte(0,0,pok->xd - pok->xg,pok->yd - pok->yg,"OK",AjCentre);
      FinEcr();
    }
}

/* ---fonction clic appelee a chaque changement d'etat ----*/
/* du dispositif de dialogue sur le domaine du bouton OK --*/
static clic_ok(transit,ref)
int transit,ref;
{
  if (transit == Desc ) FinSeance();
}

/*********************************************/
/* Primitives de Manipulation des boutons OK */
/*********************************************/

/*------- definition d'un bouton Ok ---------*/

refer Def_Ok(refvue,refacdre,xg,yg,xd,yd)
refer refvue,refacdre;
int xg,yg,xd,yd;
{
  refer refok;
  refok = Def_Domaine(refvue,xg,yg,xd,yd,sizeof(dok));
  Def_Affich(refok,aff_ok);
  Def_Clic(refok,clic_ok);
  return(refok);
}
```

```
/*************************************************************************/
/* MODULE      : increm.c                                                */
/* R0LE        : Actions de base concernant les boutons incrementeurs */
/*************************************************************************/

/****************/
/* IMPORTATIONS */
/****************/

/*---- importations de modules ------*/

#include <stdio.h>
#include <stdlib.h>

#include <getlib.h> /* constantes de la bibliothèque graphique */
#include <util.h>
#include <dial.h>
#include <dom.h>
#include <gcte.c>

#include "increm.h"

/*---- importation de variables ------*/

extern dvue *vecr;  /* vue courante d'ecriture */

/**************/
/* Constantes */
/**************/

#define Premier -1
#define Interm  0
#define Dernier 1

/**********************/
/* Variables globales */
/**********************/

static dincr *pincr;
static refer refincrem;

/*************************************************************/
/* Primitives de gestion interne des boutons incrementeurs */
/*************************************************************/

/*--- procedures d'affichage d'un incrementeur ---*/

static aff_fleche(etat)
int etat;
{
  int larg,haut,dx,dy;
  int xg,xd,yg,yd;
  int tx[7],ty[7];
  int exp;

  DebEcr(refincrem);
```

```
    Int_Domaine(reference(vecr),Rvue,&xg,&yg,&xd,&yd);
    larg = xd - xg;
    haut = yd - yg;
    dx = larg / 2;
    dy = haut / 2;
    if ( pincr->incr > 0 )
    {  /* fleche droite */
     tx[0] = xd;       ty[0] = yg + dy; /* extremite de la fleche */
     tx[1] = xd - dx;  ty[1] = yg;
     tx[2] = xd - dx;  ty[2] = yg + dy / 2;
     tx[3] = xg;       ty[3] = yg + dy / 2;
     tx[4] = xg;       ty[4] = yd - dy / 2;
     tx[5] = xd - dx;  ty[5] = yd - dy / 2;
     tx[6] = xd - dx;  ty[6] = yd;
    }
    else
    {  /* fleche gauche */
     tx[0] = xg;       ty[0] = yg + dy; /* extremite de la fleche */
     tx[1] = xd - dx;  ty[1] = yg;
     tx[2] = xd - dx;  ty[2] = yg + dy / 2;
     tx[3] = xd;       ty[3] = yg + dy / 2;
     tx[4] = xd;       ty[4] = yd - dy / 2;
     tx[5] = xd - dx;  ty[5] = yd - dy / 2;
     tx[6] = xd - dx;  ty[6] = yd;
    }

    switch (etat)
    {
     case Valide :   exp = pincr->expval;
                     break;
     case Selecte :  exp = pincr->expsel;
                     break;
    }
    AspSurf(Uni,exp);
    Polyg2d(7,tx,ty);
  FinEcr();
}

static aff_incr(refincr,mode)
/*-- affichage d'un bouton incrementeur --*/
refer refincr;
int mode;
{

   if ( ! (mode & Entites)) return;

   refincrem = refincr;
   pincr = adrincr(refincrem);

   DebEcr(refincr);
     Couleur(192,240,224);
     Rectangle(0,0,pincr->xd - pincr->xg,pincr->yd - pincr->yg);
   FinEcr();
   aff_fleche(Valide);
}
```

```
/*-------------------------*/
/*------- dialogue -------*/
/*-------------------------*/

static echo_incr(etape,p1,p2,ref)
/* fonction d'echo appelee lorsque le dispositif est en position basse sur */
/* le domaine du bouton incrementeur */
int etape,p1,p2;
refer ref;
{
  refincrem = ref;
  pincr = adrincr(refincrem);
  switch (etape)
  {
     case Premier : DebEcr(refincrem);
                    aff_fleche(Selecte);
                    break;
     case Interm :  break;
     case Dernier :  aff_fleche(Valide);
                    FinEcr();
                    break;
   }
   /* appel de  la fction de traitement associee au bouton incrementeur */
   /* elle affiche la valeur modifiee */
   (*(pincr->trt))(etape,pincr->incr);
}

/*********************************************************/
/* Primitives de Manipulation des boutons incrementeurs */
/*********************************************************/

/*-------------------------*/
/*------ definition ------*/
/*-------------------------*/

refer Def_Increm(refvue,xg,yg,xd,yd,pas,expval,expsel)
/*------- definition d'un bouton incrementeur ----------*/
refer refvue;
int xg,yg,xd,yd;     /* taille du bouton */
int pas;             /* pas d'incrementation */
int expval,expsel;   /* expressions de couleur */
{
   refincrem = Def_Domaine(refvue,xg,yg,xd,yd,sizeof(dincr));
   pincr = adrincr(refincrem);
   pincr->affich = aff_incr;
   pincr->vcour = vi;
   pincr->vmax = vmax;
   pincr->incr = pas;
   pincr->expval = expval;
   pincr->expsel = expsel;
   pincr->expinv = expinv;
   Valid_Dial(refincrem,Haut);
   Def_Echo_Domaine(refincrem,Bas,echo_incr,refincrem,0);
   return(refincrem);
}
```

```
/*-------------------------*/
/*---- modification ------*/
/*-------------------------*/

Def_Exec_Incr(refincr,adr_trt)
/*-- fonction de traitement du dialogue associee au bouton incrementeur --*/
refer refincr;
int (*adr_trt)();
{
  pincr = adrincr(refincr);
  pincr->trt = adr_trt;
}

Maj_Increm(refincr,pas)
/*-- valeurs associees à un bouton incrementeur --*/
refer refincr;
int pas;
{
   pincr = adrincr(refincr);
   pincr->incr = pas;
}

Asp_Increm(ref,expval,expsel)
/*--- aspects associes aux entites d'un bouton incrementeur ---*/
refer ref;
int expval,expsel;
{
   pincr = adrincr(refincr);
   pincr->expval = expval;
   pincr->expsel = expsel;
}

/*-------------------------*/
/*---- consultation ------*/
/*-------------------------*/

Cons_Increm(refincrincr)
/*--- consultation des valeurs associees a un bouton incrementeur ---*/
refer refincr;
int *incr;
{
   pincr = adrincr(refincr);
   *incr = pincr->incr;
}
```

```
/*************************************************************/
/* MODULE : val.h                                            */
/* ROLE   : constantes predefinies et structures de donnees  */
/*                concernant les fenetres de valeur          */
/*************************************************************/

/*********************************************/
/* Descripteur de domaine fenetre de valeurs */
/*********************************************/

typedef struct
{
  #include <infodom.h>
  int val;              /* valeur courante */
  int vmin,vmax;        /* intervalle de définition de la valeur */
  int increm;           /* valeur absolue de l'increment */
} dval;

/**********/
/* Macros */
/**********/

/* conversion reference du domaine --> adresse du descripteur */
#define adrval(ref)   ((dval *) *(_banque + ref))
```

```
/*************************************************************/
/* MODULE : incrval.h                                        */
/* ROLE   : constantes predefinies et structures de donnees  */
/*          concernant les INCREMENTEURS de valeurs          */
/*************************************************************/

/*************************************************/
/* Descripteur de domaine INCREMENTEUR DE VALEURS */
/*************************************************/

typedef struct
{
  #include <infodom.h>     /* attributs communs a tous les domaines */
                           /* attributs spécifiques */
  refer dincr1,dincr2;     /* boutons d'incrementation et de decrementation */
  refer dok;               /* bouton ok */
  int nbval;               /* nbre de valeurs */
  int expval,expsel;       /* expressions de couleur utilisees */
} dincrv;

/**********/
/* Macros */
/**********/

/* conversion reference domaine incrementeur --> adresse descripteur d'incrementeur */
#define adrincrv(ref)   ((dincrv *) *(_banque + ref))
```

```
/*************************************************************************/
/* MODULE       : incrval.c                                              */
/* R0LE         : Actions de base concernant les incrementeurs de valeurs */
/*************************************************************************/

/****************/
/* IMPORTATIONS */
/****************/

/*----- importations de modules ------*/

#include <stdio.h>
#include <stdlib.h>

#include <getlib.h>
#include <gcte.c>
#include <util.h>
#include <dial.h>
#include <dom.h>

#include "incrval.h"    /* domaines incrementeurs de valeurs */
#include "val.h"      /* domaines fenetres de valeurs */


/***************/
/* Constantes */
/***************/

#define Premier  -1
#define Interm    0
#define Dernier   1

#define ACTIF     1
#define INACTIF   0

#define DX   10 /* ecart en pixels entre les boutons */
#define DY   10
#define HCAR 16 /* Hauteur des caracteres */
#define LCAR  9 /* Largeur */

#define SEPAR '/' /* separateur des valeurs dans le descripteur */

/***********************/
/* Variables globales */
/***********************/

static dincrv *pincrv;

static dval  *pval;       /* adr. descr. fenetre de valeur selectee */
static refer refval;        /* ref fenetre de valeur selectee */
static int   valselect = Non; /* une fenetre de valeur selectee */
static char  txtval[8];       /* chaine pour affichage des valeurs */


/*************************************************************************/
/* PRIMITIVES INTERNES DE MANIPULATION DES DOMAINES FENETRES DE VALEURS */
```

```
/*************************************************************************/

/*--------- affichage ----------------------*/

static aff_valeur(etat)
/* affichage de la valeur courante de la fenetre PVAL */
int etat;
{
     DebEcr(refval);
       /* le fond */
       Couleur(192,240,224);
       Rectangle(0,0,pval->xd - pval->xg,pval->yd - pval->yg);
       /* le texte */
       switch (etat) {
         case ACTIF   : AspTexte(Uni,pval->pere->expsel,0);
                /* Couleur(240,48,0); */
                        break;
         case INACTIF : AspTexte(Uni,pval->pere->expsel,0);
                /* Couleur(32,128,0); */
                        break;
       }
       itoa(pval->val,txtval,10);
       AjustTexte(0,0,pval->xd - pval->xg,pval->yd - pval->yg,txtval,AjCentre);
     FinEcr();
}

static aff_val(ref,mode)
/* fonction d'affichage d'un domaine fenetre de valeurs */
refer ref;
int mode;
{
   if (mode & Entites)
   { refval = ref;
     pval = adrincrv(refval);
     aff_valeur(INACTIF);
   }
}

/*--------- dialogue ----------------------*/

static clic_val(transit,ref)
/*-- fonction clic executée quand transition descendante sur le
  domaine fenetre de valeur de reference ref --*/
int transit,ref;
{
  if (transit == Desc )
  {
     if (valselect)       /* desactivation de la fenetre precedemment */
       aff_valeur(INACTIF); /* selectee */

     refval = ref;
     pval = adrval(ref);

     /* mise a jour des incrementeurs */

     Maj_Increm(pval->pere->dincr1,pval->icrem);
```

```
      Maj_Increm(pval->pere->dincr2,-(pval->increm));

      if ( ! valselect)      /* premiere selection de valeur */
      { valselect = Oui;     /* validation du dialogue sur les incrementeurs */
        Valid_Dial(pval->pere->dincr1,Tout);
        Valid_Dial(pval->pere->dincr2,Tout);
      }

      /* changement de couleur de la fenetre de valeurs selectee */
      aff_valeur(ACTIF);
   }
}

static trt_val(etape,incre)
/* fonction de traitement invoquee à par l'echo en position basse des
   incrementeurs : mise a jour et affichage de la valeur -----*/
int etape,incre;
{
  switch (etape)
  {
     case Premier : break;

     case Interm  : pval->val += incre;
                    if (pval->val > pval->vmax)
                       pval->val = pval->vmin;
                    else if (pval->val < pval->min)
                       pval->val = pval->max;
                    break;

    case Dernier :  break;

   }
   aff_valeur(ACTIF);
}

/*------------- definition ------------------*/

static refer Def_Val(ref,xg,yg,xd,yd,val,vmin,vmax,incr)
refer ref;
int xg,yg,xd,yd;     /* zone utile du domaine */
int val,vmin,vmax;   /* valeur initiale et intervalle de variation */
int incr;        /* valeur absollue du pas d'incrementation */
{

   refval = Def_Domaine(ref,xg,yg,xd,yd,sizeof(dval));
   pval = adrval(refval);
   Att_Cadre(refval,NoCadre);
   pval->val = val;
   pval->vmin = vmin;
   pval->vmax = vmax;
   pval->increm = incr;
   pval->affich = aff_val;
   pval->clic = clic_val;
   return(refval);
 }
```

```
/*********************************************************************/
/* PRIMITIVES INTERNES DE MANIPULATION DES INCREMENTEURS DE VALEURS */
/*********************************************************************/

/*----- affichage -------------------*/

static aff_IncrVal(ref,mode)
/* fonction d'affichage d'un objet de type incrementeur de valeurs */
refer ref;
int mode;
{
   refer refv;
   dval *pv;
   int i;

   if (mode & Entites)
   { pincrv = adrincrv(ref);

     DebEcr(ref);
       Couleur(192,240,224);
       Rectangle(0,0,pincrv->xd - pincrv->xg,pincrv->yd - pincrv->yg);
       Couleur(32,128,0);
       /* affichage des sous domaines fenetres de valeur */
       refv = pincrv->fils;
       for (i = 0; i < pincrv->nbval; i++)
       {
          Aff_ObjDial(refv,mode);
          pv = adrval(refv);
          refv = pv->frere;
       }

     FinEcr();

    /* affichage des boutons incrementeurs */
     Aff_ObjDial(pincrv->dincr1,mode);
     Aff_ObjDial(pincrv->dincr2,mode);
    /* affichage du bouton OK */
     Aff_ObjDial(pincrv->dok,mode);
   }
}

static int extract_val(ch,nbdigit)
/* ------------------------------------------------------------------------*/
/* extraction des valeurs contenues dans le descripteur pour la definition */
/* d'un incrementeur de valeurs  : renvoie la valeur et le nombre de chiffres */
(NBDIGIT) extraits du descripteur de valeurs CH */
/*-----------------------------------------------*/
char *ch;
int  *nbdigit;
{
   int n = 0;
   int signe = 1;
   char *ad = ch;

   if (*ad == SEPAR)
     ad++;
```

```
    if (*ad == '+' || *ad == '-')
      signe = (*ad++ == '+') ? 1 : -1;
    *nbdigit = (signe == -1) ? 1 : 0;

    for (n = 0 ; (*ad != SEPAR) && (*ad != '\0')  ; ad++ )
    {
      n = 10 * n + (*ad - '0');
      (*nbdigit) ++;
    }

    return(signe * n);
}

#define FIN_VI    1
#define FIN_VMIN  2
#define FIN_VMAX  3
#define FIN_INCR  4

static analyse_descr(d,nbval,nbc)
/* analyse un descipteur D de valeurs vi/vmin/vmax/incr/.../vi/vmin/vmax/incr
  et retourne : NBVAL le nombre quadruplets vi/vmin/vmax/incr contenus dans D
  et NBC le nombre maximum de caracteres necessaire a l'affichage de toutes
  ces valeurs */
char *d; /* descripteur */
int *nbval,*nbc;
{
  int typ_separ = FIN_VI;
  int nbc_vmin = 0;
  int nbc_cour = 0;

   *nbc = 0;
   for (*nbval = 1 , d = descr; *d != '\0'; d++)
      if (*d == SEPAR)
      {
       (*nbval)++;
       switch (typ_separ) {
         case FIN_VI  : /* la prochaine valeur est une vmin */
                    typ_separ = FIN_VMIN;
                    nbc_cour = 0;
                    break;
        case FIN_VMIN   : /* la prochaine valeur est une vmax */
                    nbc_vmin = nbc_cour;
                    typ_separ = FIN_VMAX;
                    nbc_cour = 0;
                    break;
        case FIN_VMAX   : /* la prochaine valeur est un incr */
                    (*nbc) += (nbc_vmin > nbc_cour) ? nbc_vmin : nbc_cour;
                    typ_separ = FIN_INCR;
                    break;
        case FIN_INCR   : /* la prochaine valeur est une vi */
                    typ_separ = FIN_VI;
                    break;
       }
       }
       else
```

```
        if ((typ_separ == FIN_VMIN) ||(typ_separ == FIN_VMAX))
          nbc_cour++;

   /* nbval == nbre de separateurs + 1 */
   (*nbval) /= 4; /* nbre de quadruplets vi/vmax/vmin/incr */
}

/*****************************************************************/
/**   PRIMITIVES DE MANIPULATION DES INCREMENTEURS DE VALEURS **/
/*****************************************************************/

/*-------------------------*/
/*------ definition ------*/
/*-------------------------*/

/*--------- definition d'un incrementeur de valeur -------*/
refer Def_IncrVal(refvue,refcadre,xg,yg,descr,expval,expsel)
/* definition d'un incrementeur de valeurs */
refer refvue,refcadre;  /* domaine pere, cadre */
int xg,yg;              /* coin inf. gauche */
char *descr; /* descripteur des valeurs vi/vmin/vmax/incr/vi/...vi/vmin/vmax/incr */
int expval,expsel;
{
   refer refincrv,refincr1,refincr2,refok,refv;
   int xd,yd,dx,dy,x1,x2,y1,y2;
   char *d;
   int nbval,vi,vmax,nbc;

   /* calcul du nombre de valeurs NBVAL et du nombre
        de caracteres occupes NBC */
   analyse_descr(d,&nbval,&nbc)

   /*-- definition du domaine global (racine) ----*/

   xd = xg + (nbval+3) * DX + (nbc+4) * LCAR;
   yd = yg + 2 * DY + HCAR;
   refincrv = Def_Domaine(refvue,xg,yg,xd,yd,sizeof(dincrv));
   Att_Cadre(refincrv,refcadre);
   pincrv = adrincrv(refincrv);
   pincrv->affich = aff_IncrVal;
   pincrv->nbval = nbval;
   pincrv->expval = expval;
   pincrv->expsel = expsel;

   /* ------ definition des sous domaines ------- */

   dx = (xd - xg);
   dy = (yd - yg);

   /* sous domaine bouton OK */

   x2 = dx - DX; ; y2 = dy - DY;
   x1 = x2 - 2*LCAR; y1 = y2 - HCAR;

   pincrv->dok = refok = DefOk(refincrv,0,x1,y1,x2,y2);
   Attach_Domaine(refok);
```

```
    /* sous domaine incrementeur */

    x2 = x1 - DX; ;
    x1 = x2 - 2*LCAR; y1 = y2 - HCAR/2;

    refincr1 = Def_Increm(refincrv,x1,y1,x2,y2,1,expval,expsel);
    Def_Exec_Incr(refincr1,trt_val);
    pincrv->dincr1 = refincr1;
    Attach_Domaine(refincr1);

    /* sous domaine decrementeur */

    y2 = y1; y1 = y2 - HCAR/2;

    refincr2 = Def_Increm(refincrv,x1,y1,x2,y2,-1,expval,expsel);
    Def_Exec_Incr(refincr2,trt_val);
    pincrv->dincr2 = refincr2;
    Attach_Domaine(refincr2);

    /* sous domaines fenetres de valeur */

    y1 = DY; y2 = y1 + HCAR;
    x2 = 0;

    d = descr;
    do
    {
       vi = extract_val(d,&nbc);
       d += nbc + 1; /* on saute la valeur et le separateur */
       vmin = extract_val(d,&nbc_vmin);
       d += nbc_vmin + 1; /* on saute la valeur et le separateur */
       vmax = extract_val(d,&nbc_vmax);
       d += nbc_vmax + 1; /* on saute la valeur et le separateur */
       incr = extract_val(d,&nbc);
       d += nbc;      /* on saute la valeur */

       nbc = (nbc_vmin > nbc_vmax) ?  nbc_vmin : nbc_vmax;
       x1 = x2 +DX; x2 = x1 + nbc * LCAR;
       refv = Def_Val(refincrv,x1,y1,x2,y2,vi,vmin,vmax,incr);
       Attach_Domaine(refv);
     } while (*d != '\0');

   return(refincrv);
}

/*-------------------------*/
/*---- modification ------*/
/*-------------------------*/

Maj_IncrVal(ref,no_val,vi,vmin,vmax,incr)
/*--- modification des valeurs de l'entite de rang NO_VAL ------*/
/*--- la nouvelle valeur maximale ne doit pas avoir un nombre --*/
/*--- de chiffre plus grand que celle qui est remplacee --------*/
refer ref;
int no_val,vi,vmin,vmax,incr;
```

```
{
  int rang;

  pincrv = adrincrv(ref);
  if (no_val > 0 && no_val <= pincrv->nbval)
  {  refval = pincrv->fils;
     pval = adrval(refval);
     /* les fils sont chaines a l'envers */
     for (rang = pincrv->nbval - no_val; rang ; rang--)
     {
          refval = pval->frere;
          pval = adrval(refval);
     }
     pval->val = vi;
     pval->vmin = vmin;
     pval->vmax = vmax;
     pval->increm = vincr;
     /* si lincrementeur est affiche mise a jour de l'image */
     if (pval->etat & AFFI)
        aff_valeur(INACTIF);
  }
}

/*---- modification des aspects standards -----*/

Asp_IncrVal(ref,expval,expsel)
refer ref;
int expsel,expval,;
{
    pincrv = adrincrv(ref);
    pincrv->expval = expval;
    pincrv->expval = expval;
    /* on modifie aussi l'aspect des incrementeurs */
    Asp_Increm(pincrv->dincr1,expval,expsel);
    Asp_Increm(pincrv->dincr1,expval,expsel);
}

/*-------------------------*/
/*---- consultation ------*/
/*-------------------------*/

Cons_IncrVal(ref,no_val,vcour,vmin,vmax,incr)
/*--- consultation  des valeurs de l'entite de rang NO_VAL ------*/
refer ref;          /* ref de l'incrementeur de valeur */
int no_val;         /* rang de l'entité consultée */
int *vcour,*vmin,*vmax; /* valeur courante et intervalle de variation de l'entité */
int *incr;          /* valeur absolue du pas d'incrementatin */
{
  int rang;

  pincrv = adrincrv(ref);
  if (no_val > 0 && no_val <= pincrv->nbval)
  {  refval = pincrv->fils;
     pval = adrval(refval);
     /* les fils sont chaines a l'envers */
     for (rang = pincrv->nbval - no_val; rang ; rang--)
```

```
    {
        refval = pval->frere;
        pval = adrval(refval);
    }
    *vcour = pval->val;
    *vmin = pval->vmin;
    *vmax = pval->vmax;
    *incr = pval->increm;
  }
  else
    *vcour = *vmin = *vmax = *incr = 0;
}
```