Un modele multicouche pour la construction d'applications graphiques interactives

Jean-Daniel Fekete

THÈSE

présentée par

Jean-Daniel Fekete

pour obtenir
Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI ORSAY

Spécialité : informatique

Un modèle multicouche pour la construction d'applications graphiques interactives

Soutenue le 12 Janvier 1996

M^me^ Froidevaux Christine	Présidente
M. Beaudouin-Lafon Michel	Directeur
M^me^ Nanard Jocelyne	Rapporteur
M. Quint Vincent	Rapporteur
M^me^ Coutaz Joëlle	Examinateur

Thèse préparée au sein du **L**aboratoire de **R**echerche en **I**nformatique

Remerciements

Je remercie Christine Froidevaux qui m'a fait l'honneur de présider mon jury de thèse.

Je remercie Michel Beaudouin-Lafon, mon directeur de thèse. Sans sa rigueur et sa patience, je n'aurais jamais pu mener à bien mon travail de thèse.

Je considère comme un honneur que Jocelyne Nanard et Vincent Quint aient bien voulu être rapporteurs et que Joëlle Coutaz ait accepté de participer à mon jury. Je les en remercie, ainsi que pour leur travail en général qui a été pour moi une source importante de motivation, d'inspiration et de référence.

Le travail décrit dans cette thèse a été réalisé en partie au LRI et principalement dans la société 2001 S.A. Je voudrais particulièrement remercier Fabrice Mourlin du LRI, sans qui cette thèse n'aurait pas pu être terminée à temps. Je remercie l'équipe informatique de 2001 S.A : Olivier Arnaud, Érick Bizouarn, Éric Cournarie, Gregory Denis, Thierry Galas, Jean-Louis Moser, Frédéric Taillefer. Ils ont eu la patience de travailler avec moi et d'essuyer les plâtres des versions préliminaires de l'architecture multicouche dans un contexte industriel. Merci aussi aux membres du groupe Interaction Homme Machine du LRI : Thomas Baudel, Yves Berteaud, Stéphane Chatty, Christophe Tronche et Mountaz Zizi, avec qui j'ai eu des échanges aussi nombreux que fructueux.

Je dois beaucoup à plusieurs personnes pour des motifs trop nombreux à décrire ici, je tiens à les remercier. J'espère que ceux que j'aurais omis, ne m'en voudront pas trop. Merci donc à Jacques André, Jean Démétro, Jean-Louis Esteve, Francis Freisz, Michel Gangnet, Henri Gouraud, Jacques Lefaucheux, Michel Leroux, Wendy Mackay, Thierry Pudet, Jean-Manuel Van Thong et Chris Weikart.

Si j'ai pu mener à bien cette thèse, c'est avant tout grâce à Nicole, qui a eu la patience de me supporter (dans le sens français et anglais) pendant les nombreuses années passées entre 2001 S.A. et le LRI. Je lui en suis reconnaissant. Merci aussi à Adrien pour sa gentillesse.

Table des matières

Introduction

Avant l'arrivée sur le marché du Macintosh de Apple en 1984, les applications graphiques professionnelles utilisaient du matériel informatique dédié et très coûteux ou exploitaient les capacités graphiques de machines généralistes de manière exclusive et spécialisée. Les premières applications de publication assistée par ordinateur (PAO) ont démontré qu'il était possible de réaliser des éditeurs de qualité professionnelle sur un système de fenêtrage généraliste. Aujourd'hui, les imprimés sont tous fabriqués à partir d'ordinateurs généralistes. Les autres domaines graphiques ont suivi, comme le montage de vidéo amateur, la cartographie ou la retouche d'image. Cependant, quelques domaines d'applications graphiques, nécessitant une grosse puissance de calcul ou des dispositifs d'entrée particuliers, ne pouvaient pas utiliser ces systèmes de fenêtrage généralistes.

Jusqu'à récemment, les applications utilisant un modèle graphique 3D ne pouvaient réafficher rapidement des scènes complexes en utilisant les fonctions graphiques des systèmes de fenêtrage. De même pour les systèmes de montage et trucage vidéo professionnels.

De même, les systèmes de fenêtrage ne géraient qu'un clavier et une souris. Aucun mécanisme n'était prévu pour ajouter un dispositif supplémentaire, parfois indispensable, comme par exemple un second clavier dans le cas d'une application de composition de texte multilingue ou deux souris pour l'interaction à deux mains [Chatty94].

Pour pallier à ces déficiences, les systèmes de fenêtrage comme X [Scheifler et al.86] et Windows [Microsoft Corporation92] ont adopté des extensions permettant à pratiquement toutes les applications graphiques de fonctionner convenablement dans les systèmes de fenêtrage.

Avec des extensions, la gestion du graphique 3D peut se faire, dans X, en utilisant GKS [American national standards institute85], PHIGS [American national standards institute88] ou OpenGL [Segal et al.93] qui est aussi disponible sous Windows. Des dispositifs d'entrées multiples sont utilisables sous X avec l'extension d'entrée Xi [Ferguson92].

L'utilisation de ces extensions pose de nouveaux problèmes pour la conception et la réalisation des applications graphiques interactives.

Nous nous intéressons dans cette thèse à ce que nous appelons des *éditeurs* et qui sont des applications graphiques interactives qui :

- sont dédiées à un *domaine* ;
- ne se limitent pas à utiliser des objets graphiques disponibles en bibliothèque ou stéréotypés ;
- permettent de créer ou de détruire interactivement des objets graphiques ;
- permettent d'agir sur ces objets par *manipulation directe* ;
- peuvent utiliser des extensions tant pour la gestion du graphique que pour la gestion des dispositifs d'entrée.

Un *domaine* est caractérisé à la fois par le profil des utilisateurs potentiels et le type de travaux réalisables à l'aide de l'éditeur. Des éditeurs connus sont : Illustrator de Adobe [Adobe Systems Incorporated91], Canvas de Deneba [Deneba software91], FontStudio de Letraset [Letraset92], PhotoShop d'Adobe [Adobe Systems Incorporated93a], ou 3D Studio de Autodesk [Autodesk92]. Ces éditeurs sont chacun spécifiques à un domaine : Illustrator est un éditeur de dessin vectoriel, destiné à être utilisé par des illustrateurs professionnels et permettant de produire des illustrations de magazines, publicité, etc. qui étaient auparavant réalisés à l'aide de carte à gratter ou d'aérographe. Canvas est destiné à être utilisé par un public moins créatif pour réaliser des graphiques schématiques comme des plans au sol ou des synoptiques. FontStudio est conçu pour numériser ou retoucher des polices de caractères vectorielles. PhotoShop permet la retouche d'images fixes ainsi que le dessin pixellaire pour l'impression professionnelle. Finalement, 3D Studio permet de modéliser des objets en trois dimensions, destinés plutôt à l'image fixe comme le design ou l'architecture.

Tous ces éditeurs ont des interfaces qui se ressemblent : une partie importante des connaissances que l'on peut avoir de l'un de ces éditeurs est transférable aux autres. Cependant, chacun de ces éditeurs offre des fonctions et des modes d'utilisation trop spécifiques pour qu'un seul éditeur configurable puisse tous les remplacer. Au contraire, nous assistons aujourd'hui à une prolifération d'éditeurs très spécialisés (systèmes d'informations géographiques, dessin animé professionnel, montage et effets spéciaux vidéo, contrôle aérien, etc.).

Dans cette thèse, nous nous intéressons à la construction d'éditeurs du point de vue du génie logiciel de l'interaction homme-machine. Nous n'abordons pas le point de vue des facteurs humains.

Dans la première partie, nous décrivons l'architecture des applications graphiques, les outils logiciels et les modèles d'architectures et analysons en quoi ils sont utilisables et adaptés aux éditeurs. Dans la seconde partie, nous présentons une architecture logicielle destinée à la construction d'éditeurs, et préconisons une méthode de réalisation. Plusieurs éditeurs graphiques commercialisés ont été conçus et réalisés avec notre architecture. Elle a donc été validée par la pratique. Ces éditeurs font partie du système professionnel de fabrication de dessins animés TicTacToon [Fekete et al.95] qui met en jeu des techniques nouvelles pour gérer le dessin à main levée. Nous décrivons l'un de ces éditeurs a la fin de la troisième partie, après avoir présenté un exemple de réalisation d'éditeur en suivant notre méthode.

Dans notre conclusion, nous présentons les perspectives ouvertes par notre travail.

Première partie

La construction d'éditeurs

Chapitre 1

Architecture des applications graphiques

Pour développer une application graphique interactive, le programmeur peut s'appuyer sur plusieurs niveaux logiciels et d'outils dont le rôle s'est clarifié et s'est stabilisé depuis une dizaine d'années. Ces niveaux et outils, du plus près du matériel jusqu'au plus abstrait, sont :

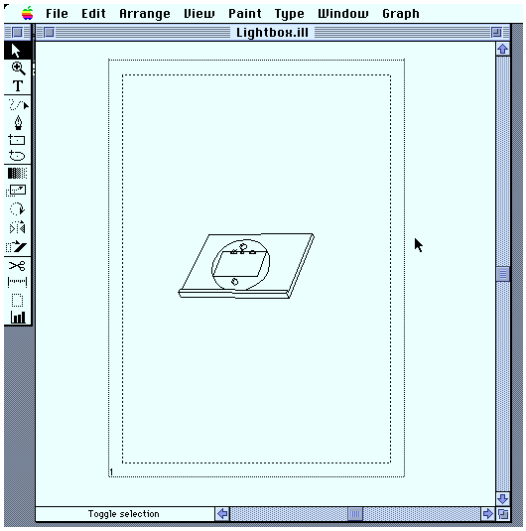
- les systèmes de fenêtrage ;
- les boîtes à outils ;
- les langages graphiques ;
- les squelettes d'applications (*application frameworks*) ;
- les éditeurs d'interface (*interface builders*) ;

Les figures 1.1 montrent quelques exemples d'éditeurs graphiques professionnels et à quel point leur présentation est semblable. On distingue une barre de menu, une boîte d'outils, et une fenêtre principale, parfois décorée d'objets de contrôle. Ce qui varie d'une application à une autre, c'est la nature des objets affichés dans la vue principale, le modèle graphique utilisé pour leur affichage, les dispositifs d'entrée et les *modalités d'interaction* (voir la définition en § 1.1).

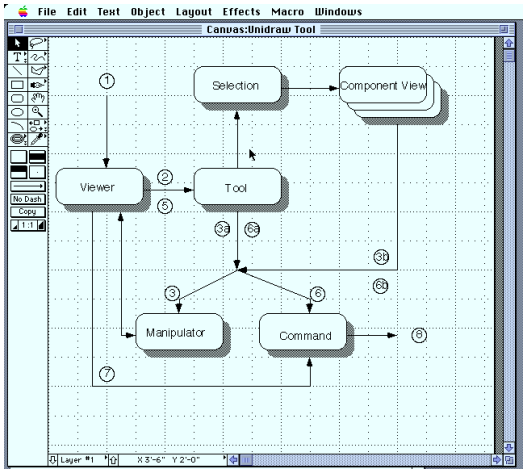
Dans cette partie, nous commençons par décrire la façon dont les extensions modifient les systèmes de fenêtrage. Nous décrivons ensuite l'impact de ces modifications dans les boîtes à outils, dans les outils de construction d'interfaces existants et enfin dans les modèles d'architectes.

1.1 Vocabulaire et concepts

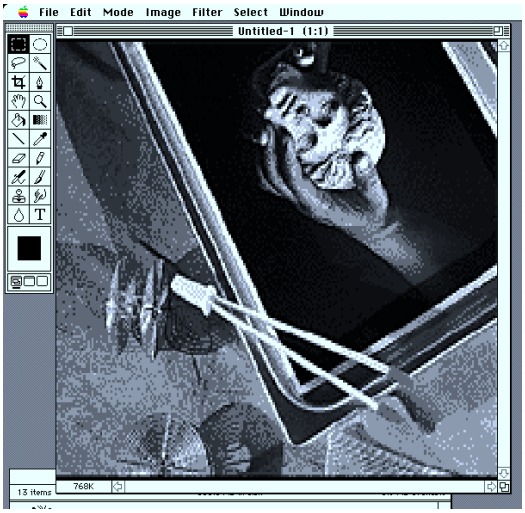
Pour décrire les éditeurs, nous employons des termes et des concepts qui sont utilisés par la communauté scientifique et technique, parfois en anglais, et dont



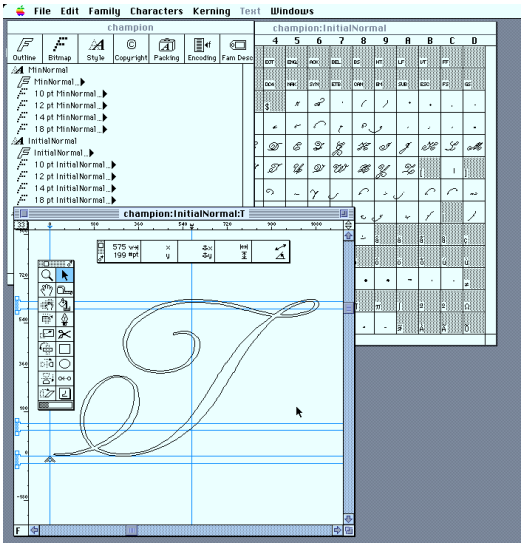
(a) Illustrator de Adobe



(b) Canvas de Deneba



(c) Photoshop de Adobe



(d) FontStudio de Letraset

FIG. 1.1 - *Éditeurs graphiques sur Macintosh*

nous donnons ici une définition.

Un *dispositif d'entrée*, ou plus simplement *dispositif*, est un objet qui a une facette physique, une facette numérique et qui gère la communication entre un utilisateur et un ordinateur. Il transforme des signaux physiques émis par des capteurs en informations numériques, permet à l'ordinateur de connaître l'état des capteurs et notifie l'ordinateur lorsque cet état physique change. Les stations de travail disposent toutes d'une souris et d'un clavier comme dispositifs d'entrée.

Un *curseur* est un petit objet graphique qui visualise la position d'un dispositif sur l'écran d'un ordinateur. Un *curseur de texte* existe dans un éditeur de texte et visualise la position où le prochain caractère sera inséré lorsque l'on tapera au clavier.

Une *action* est une modification de l'état d'un dispositif dans un contexte donné, destiné à modifier l'état d'une application graphique interactive. Le fait d'appuyer sur le bouton d'une souris (*cliquer*) lorsque le curseur est placé sur un objet graphique est une action.

Une *manipulation* ou une *interaction* est une suite d'actions destinées à spécifier une commande et ses arguments. Par exemple, appuyer sur le bouton de la souris et le relâcher lorsque le curseur est placé sur un icone dans le *Finder* du Macintosh est une manipulation qui déclenche la commande (voir ci-dessous) « sélectionner » avec comme argument l'icone sous le curseur. Une commande est initiée par une *action initiale* et est terminée par une *action terminale*.

Une *commande* est une fonction avec ses arguments, déclenchée par une action terminale. Par exemple, l'action consistant à cliquer sur le menu « Quitter » déclenche la commande « *exit*(0) » de l'application.

Un *mode d'interaction* est défini par une liste d'actions initiales et les manipulations que chaque action déclenche. Par exemple, dans l'éditeur MacDraw, lorsque l'icone représentant une flêche est sélectionné dans la boîte d'outils, le mode d'interaction permet de déplacer des objets graphiques ou de le sélectionner (ce sont deux manipulations différentes). Le mode d'interaction est parfois appelé *outil*, cependant, nous utilisons ce terme dans un autre sens, précisé en deuxième partie, § 2.

Une *modalité d'interaction* définit selon quelle interprétation une manipulation sera transformée en commande. Par exemple, la suite d'actions commençant par un bouton de souris appuyé, suivi de mouvements de la souris et terminé par le relâchement du bouton de la souris est une manipulation qui peut être interprétée comme du « clique et tire » ou comme du « dessin à main levée » ou comme de la « reconnaissance de geste ».

La *manipulation directe* a été introduite par B. Shneiderman [Shneiderman83] pour qualifier une manipulation interactive ayant les caractéristiques suivantes :

1. représentation continue des objets d'intérêt ;

2. actions physiques plutôt que syntaxe complexe ;

3. actions rapides, incrémentales et réversibles dont l'effet sur l'objet est rendu immédiatement visible.

Ces caractéristiques donnent à l'interface des propriétés intéressantes, en particulier en terme de vitesse d'apprentissage, de vitesse d'exécution des tâches et de mémorisation du processus opératoire.

Nous utilisons dans cette thèse la notion de manipulation directe dans un sens plus précis que celui de Shneiderman, et qui a été défini par Karsenty [Karsenty94] par opposition à *manipulation indirecte*. En effet, en se rapportant à la définition de Shneiderman, la manipulation d'un menu ou d'un bouton poussoir peut être interprétée comme directe, suivant le sens qu'on donne à l'« objet d'intérêt » de la définition de Shneiderman. Cependant, la manipulation étant destinée principalement à déclencher une action extérieure à l'objet graphique qu'est le menu ou le bouton, Karsenty la qualifie de manipulation indirecte. Le terme *manipulation directe* est réservé à l'interaction avec les « vrais » objets d'intérêt, c'est-à-dire ceux présentés dans la vue principale.

Chapitre 2

Les systèmes de fenêtrage

Le premier système de fenêtrage est apparu dans l'environnement Smalltalk-76 [Ingalls78] de Xerox Parc, d'après une idée de Dan Ingalls, pour permettre à plusieurs programmes d'utiliser le même écran graphique sur une station de travail. Le concept de « station de travail » a ensuite inspiré plusieurs autres équipes qui l'ont importé avec le système de fenêtrage. Les premiers systèmes de fenêtrage étaient utilisés à travers un langage de programmation dans lequel la mémoire était partagée entre tous les processus, autorisant un accès direct à la mémoire graphique. Le système de fenêtrage du Macintosh est un héritier direct de l'environnement Smalltalk, dont il a gardé le principe de fenêtres superposées, de couper/coller et d'édition non modale. Windows s'est ensuite inspiré du système de fenêtrage du Macintosh. Le premier système de fenêtrage disponible dans l'environnement Unix est SunWindows [Sun microsystems], basé sur des mécanismes de partage de la mémoire implantés dans le noyau. Pour pallier à de nombreux problèmes liés à cette architecture (nécessité de dupliquer les bibliothèques dans chaque processus utilisant le système, débogage des applications difficile, portage nécessitant des modifications du noyau), plusieurs systèmes de fenêtrage utilisant le modèle client/serveur ont été développés [Gosling86, Scheifler et al.86]. Leur originalité principale vient de l'utilisation de requêtes asynchrone pour assurer des performances acceptables sous UNIX.

Aujourd'hui, les deux familles de systèmes de fenêtrage continuent à coexister : Windows et le Macintosh utilisent toujours un système partageant la mémoire, tandis que X utilise toujours un modèle client/serveur. Cependant, dans le système Windows-NT [Custer93], Windows est implanté sous forme de client/serveur dont les requêtes sont synchrones et utilisent le mécanisme interne d'envoi de message inter-processus du système d'exploitation.

2.1 Rôles du système de fenêtrage

Le système de fenêtrage définit une couche logicielle qui :

- abstrait les fonctions graphiques du matériel ;
- autorise le partage de l'écran entre plusieurs applications ;
- simplifie la gestion des événements en les sérialisant et en les transmettant sur un seul canal ;
- offre des services utiles à toutes les applications graphiques comme la gestion de la couleur, des ressources, des polices de caractères, etc.

Au niveau le plus proche de la machine, le programmeur d'application graphique interactive doit gérer les deux sens de la communication homme-machine. Cette composante de l'interaction homme-machine a été relativement stable depuis cinq ans mais recommence à évoluer avec l'apparition d'extensions permettant de tirer profit de spécificités matérielles. Un problème actuel est de permettre l'utilisation des extensions dans un système de suffisamment haut niveau.

2.2 Sens machine-homme

Le sens machine-homme repose sur un modèle graphique qui permet de contrôler l'affichage de points sur un écran. Dans notre travail, nous ne nous intéressons pas aux autres modalités comme le son.

L'architecture des écrans est maintenant assez stable et permet à l'application de les voir comme un tableau de points (pixels) théoriquement rectangulaires. Les pixels informatiques sont caractérisés par leur dimension, le nombre de couleurs et leur intensité. Sur des contrôleurs plus évolués, l'écran physique peut combiner plusieurs sources graphiques et les pixels peuvent alors avoir des niveaux de transparence.

Les écrans les plus simples offrent deux teintes par pixel (noir et blanc par exemple) tandis que les plus sophistiqués permettent d'afficher 16 millions de couleurs différentes et parfois 256 niveaux de transparence, en autorisant la superposition de plusieurs images. La majorité des cartes graphiques gère les couleurs à travers une table des couleurs, chaque pixel contenant un index dans cette table et chaque entrée de la table contenant une valeur de couleur réelle. Dans la majorité des cas, chaque pixel est codé sur un octet qui peut donc référencer 256 couleurs différentes, chaque couleur étant codée sur trois octets.

Le système de fenêtrage gère un ou plusieurs écrans et offre deux abstractions appelées fenêtre (*Window*) et surface virtuelle (parfois *Canvas*). Comme l'illustre

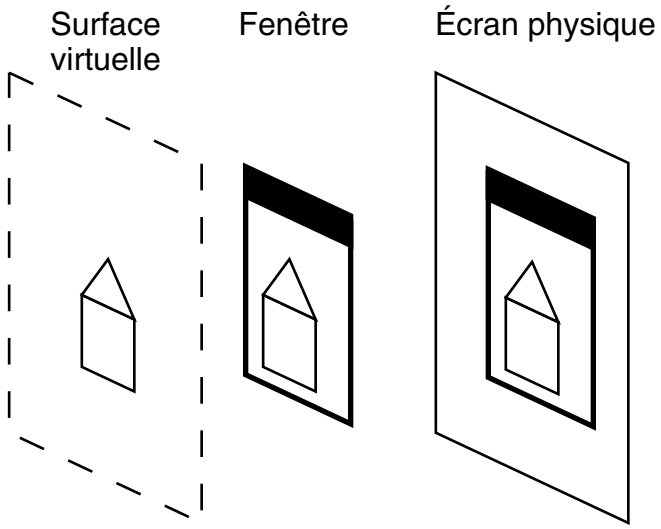


FIG. 2.1 - *Le système de fenêtrage présente à l'utilisateur une fenêtre et au programme une surface graphique virtuelle.*

la figure 2.1, la fenêtre est la zone rectangulaire visualisable sur l'écran, qui représente une partie d'une surface graphique virtuelle. Celle-ci est l'abstraction de l'écran que voit l'application graphique.

Bien que l'écran soit finalement un tableau de points, des architectures graphiques spécialisées ont optimisé certaines fonctions graphiques qui sont exécutées par le matériel à une vitesse impossible à approcher par logiciel. Un des problèmes de l'architecture des éditeurs, mais aussi des systèmes de fenêtrages, est de rendre ces optimisations utilisables sans obliger les développeurs d'éditeurs à spécialiser leur code pour un matériel particulier.

> Par exemple, certaines architectures matérielles permettent d'accélérer sensiblement la copie de zones d'images (la primitive *bitblt* de Smalltalk [Goldberg et al.83]) en utilisant de la copie de mémoire à mémoire sans passer par le processeur (DMA). Cette optimisation permet un gain de dix fois pour la copie d'image.
>
> Mais, certaines cartes graphiques ne permettent cette optimisation que pour des images de 256 pixels par 256 au maximum. Cette restriction produit une variation de performance sur les applications réelles qui obligent le programmeur à modifier son code pour s'assurer que les images qui doivent apparaître rapidement ne dépasseront pas la taille cruciale, au détriment de la portabilité.

Les optimisations matérielles sont aujourd'hui tellement répandues que tous les systèmes de fenêtrages permettent d'en tirer profit. Par exemple, Windows [Mi-

crosoft Corporation91], ainsi que X11 [Scheifler et al.86] permettent d'accéder à la bibliothèque OpenGL qui implante plusieurs optimisations 2D et 3D. C'est surtout dans le domaine du graphique 3D que ces optimisations sont indispensables, la vitesse de certaines fonctions complexes comme le plaquage de texture pouvant être jusqu'à 100 fois inférieur en logiciel qu'en matériel. Similairememt, XIE [Rogers94] est une extension pour le traitement d'images pixellaires qui permet de déléguer au serveur des opérations coûteuses. Certaines de ces opérations peuvent tirer profit d'accélérateurs matériels, comme des processeurs de compression ou décompression d'image ou des processeurs de traitement du signal. Une particularité de cette extension est que les opérations sont effectuées par le serveur de manière totalement asynchrone et parallèles ; le serveur notifie l'application lorsqu'un traitement est terminé. Enfin, la plupart des cartes graphiques implantent des plans d'*overlay*, c'est-à-dire une surface virtuelle qui s'affiche au-dessus de la surface virtuelle normale et dans laquelle une couleur particulière est transparente, ou encore certaines couleurs sont semi-transparentes. Les systèmes de fenêtrage utilisent parfois un plan pour gérer le curseur et offrent parfois des mécanismes pour utiliser ces plans, généralement en les faisant passer pour des autres écrans.

Les éditeurs professionnels doivent bien entendu tirer profit de ces optimisations et nous verrons dans les chapitres suivants que pratiquement aucune couche ou outil ne les encapsule correctement.

2.3 Sens homme-machine

L'application doit traiter des informations qui lui viennent de dispositifs d'entrée. Contrairement au sens précédent où seul le graphique doit être géré, les dispositifs d'entrée et leur mode d'utilisation peuvent être très divers. Les stations de travail disposent toutes d'un clavier et d'une souris mais cette configuration est de moins en moins la seule à exister. La souris est parfois remplacée par une tablette à numériser avec un stylet. Dans le cas des machines à stylet (*Pen-based computers*), le clavier et la souris sont remplacés par le seul stylet, qui utilise la surface de l'écran comme tablette. Par ailleurs, les dispositifs d'entrée deviennent de plus en plus nombreux (*data glove*, *flying mouse*, *knob box*, écran tactile, caméra, micro avec reconnaissance vocale, etc.) et leur mode d'utilisation de plus en plus complexe (reconnaissance de geste pour les souris, les stylets et les data glove, prise en compte de la pression et de la distance à la tablette pour les stylets, reconnaissance de posture pour les data glove, reconnaissance de sons pour les micros, etc.).

Le modèle le plus général pour la communication entre les dispositifs d'entrée et les programmes est celui des événements placés dans une file. Chaque dispositif est géré par un gestionnaire de dispositif (*device driver*). Lorsque l'état du dispositif est modifié, le gestionnaire de dispositif alloue un bloc de mémoire dans lequel

il place les informations nécessaires à décrire la modification d'état et la date de la modification. Il ajoute ce bloc à la fin de la file d'événements et permet à l'application d'accéder à cet événement.

L'application se contente donc de lire les événements dans la file et les traite, modifiant certaines données internes et l'état des dispositifs de sortie.

2.3.1 Typologie des dispositifs

Pour les extensions, des mécanismes nouveaux sont offerts pour activer un dispositif ou le désactiver. Parfois, un dispositif étendu peut remplacer un des dispositifs standards. Le modèle de [Foley et al.90, page 350] permet de décrire la plupart des dispositifs physiques. Chaque dispositif peut avoir plusieurs facettes :

positionnel s'il peut gérer une position. Il peut être *absolu*, *relatif*, *direct* ou *continu* (ces attributs ne sont pas tous exclusifs).

clavier s'il gère des codes de touches.

valuateur s'il gère des valeurs sur des *axes*. Chaque axe peut avoir une valeur *bornée* ou non, et une *résolution*.

choix s'il gère des codes successifs. Un choix peut être *exclusif* ou non.

En suivant cette typologie, une souris est un *positionnel*, un *valuateur* et un *choix*. Le *positionnel* est relatif indirect et continu, le *valuateur* a deux axes non bornés et le choix a trois codes (pour une souris à trois boutons) non exclusifs.

Les événements de type sonores et de reconnaissance de parole ne rentrent pas bien dans cette typologie car ils produisent des événements continus pendant une durée. Dans les systèmes de reconnaissance de parole, la date de début de l'événement ne correspond pas à sa date d'arrivée dans la file des événements, ce qui pose des problèmes de réordonnancement. Nous ne nous intéresserons pas à ces types de dispositifs dans cette thèse.

Dans X, l'extension Xi [Ferguson92] permet d'accéder à de nouveaux dispositifs et les décrit suivant cette organisation. Cependant, un problème important subsiste du fait qu'aucun système de fenêtrage ne sache gérer l'écho des dispositifs étendus. Un seul curseur est affiché par les systèmes, ce qui oblige l'application à gérer elle même la visualisation des dispositifs si elle le désire.

2.3.2 Stratégies de gestion des événements

Les systèmes de fenêtrages partageant l'espace mémoire comme sur le Macintosh ou Windows envoient tous les événements aux applications qui ne traitent

que ceux qui les intéressent. Pour des raisons de performance, les déplacements de la souris ne sont pas envoyés et doivent être lus dans une boucle active. Dans X, chaque application indique au serveur les événements qui l'intéressent et seulement ceux-là lui sont envoyés.

L'utilisation de dispositifs multiples est incompatible avec la première stratégie car les applications pourraient être saturées d'événements. Même la stratégie de X ne suffit pas pour suivre des dispositifs dont le débit est élevé. Le serveur reçoit les événements du dispositif mais leur gestion par le client accumule du retard dû au temps d'acheminement du serveur au client et au temps de traitement par le client. X préconise donc une stratégie utilisant le mécanisme de *motion hint* qui doit être utilisé dans les éditeurs professionnels qui gèrent des dispositifs comme un tablette ou une souris 3D.

Suivi optimisé du mouvement des dispositifs dans X Lorsque le mécanisme de *motion hint* est activé et qu'un dispositif positionnel se déplace, X n'envoie au client qu'un seul événement de mouvement. Lorsque le client traite cet événement, il est possible, et même probable, que la position du dispositif aie légèrement changé à cause du délai d'acheminement. Le client interprète alors l'événement comme une notification de changement de position et ignore ces informations positionnelles. Le client demande alors au serveur X la position courante du dispositif. Au cours de cette requête, le serveur autorise de nouveau le dispositif à envoyer un événement lors de son prochain changement de position.

Avec ce mécanisme, un éditeur reçoit les événements à la vitesse où il peut les traiter, ce qui supprime toute accumulation de retard. Cependant, pour être sûr de recevoir toutes les positions prises par un positionnel continu (pour gérer une *trace* ou reconnaître l'écriture manuscrite par exemple), un mécanisme additionnel doit être mis en œuvre : l'historique des états (*motion history*).

Lorsque le serveur implante cet historique (tous les serveurs ne le font pas), il garde les n dernières positions de tous les dispositifs associées à leur date. Lorsque le client traite un événement de notification provenant du déplacement d'un dispositif, il demande au serveur la portion d'historique entre la dernière date où le dispositif a été interrogé et la date courante ; il dispose alors de toutes les positions sans surcharger le serveur ni le client.

> Nous décrivons une utilisation de ce mécanisme dans l'exemple en troisième partie, § 2.4.5.2 avec une tablette graphique produisant jusqu'à 2000 événements par seconde. Sans ce mécanisme, même sur une station de travail de 300 Mips, du retard s'accumule. L'utilisation de ce mécanisme a supprimé totalement le retard sur la station à 300 Mips, qui traite alors environ 200 événements par seconde avec un historique contenant une dizaine d'échantillons par événement. Le même

programme sur une station à 30 Mips continue de fonctionner, traitant une trentaine d'événements par seconde avec un historique contenant une centaine d'échantillons.

2.4 Sens machine-machine

En plus de l'interaction homme-machine, une application graphique interactive peut avoir à réagir à des changements internes d'état, générés ou relayés par l'ordinateur lui-même. C'est le cas lorsque, par exemple, l'application demande à être prévenue après qu'un laps de temps se soit écoulé.

En général, ce type d'événement n'est pas unifié avec les gestions des événements issus des dispositifs, sauf sur le système MacOS où toute la communication entre les différents modules du système est à peu près unifiée. À cause de ce manque d'unification, il n'est pas toujours facile d'étendre les modes d'interprétation des dispositifs en gardant la même architecture de bas niveau. Par exemple, les événements de souris pourraient être analysés pour détecter des gestes particuliers et, une fois un geste détecté, la série d'événements serait remplacée par un nouvel événement synthétique de type « geste XXX », produit par un dispositif « capteur de geste ». Ce genre de manipulation n'est que rarement possible et l'unification entre événement de bas niveau et événement synthétisé n'est faite qu'à un niveau de programmation plus élevé.

2.5 Les modèles graphiques

Le système de fenêtrage permet donc de créer les surfaces virtuelles sur lesquelles les applications graphiques vont présenter les données. Nous décrivons ici les modèles graphiques classiques qui sont implantés directement par le système de fenêtrage ou par une extension. Trois modèles existent :

- image point par point (pixellaire),
- image vectorielle 2D,
- image 3D.

2.5.1 Modèle pixellaire

Dans le modèle pixellaire, les images sont décrites comme un tableau de pixels, c'est-à-dire de zones rectangulaires juxtaposées et de couleur constante. Chaque pixel est décrit par sa couleur et parfois par d'autres composantes. Le modèle le plus simple associe à chaque pixel une valeur binaire qui pourra être interprétée comme blanc et noir par exemple. Les autres modèles permettent d'associer

à chaque pixel un index de couleur dans une table des couleurs (*Look Up Table*) ou de décrire la couleur dans un espace à trois dimensions ou plus (RGB, CMYK, etc). Un attribut de transparence peut aussi être associé à chaque pixel, rendant possible la composition de plusieurs images pixellaires en ménageant des trous ou des zones semi transparentes. Des modèles encore plus sophistiqués permettent d'associer à chaque pixel plusieurs attributs (G-Buffer [Saito et al.90]).

Les principales primitives sont :

type Couleur, **Pixel**;
fn *largeur*(**ImagePixel**): **Entier**
fn *hauteur*(**ImagePixel**): **Entier**
fn *prend_pixel*(**ImagePixel**, **PointEntier**): **Pixel**;
proc *met_pixel*(**ImagePixel**, **PointEntier**, **Pixel**);
fn *couleur_pixel*(**Pixel**): **Couleur**;
fn *pixel_couleur*(**Couleur**): **Pixel**;

Des fonctions supplémentaires sont nécessaires lorsque l'image utilise une table des couleurs :

type Pixel = Entier;
fn *prend_couleur_indice*(**ImagePixel**,**Pixel**): **Couleur**;
proc *met_couleur_indice*(**ImagePixel**, **Pixel**, **Couleur**);
fn *couleur_max*(**ImagePixel**): **Entier**

2.5.2 Modèle vectoriel

Le modèle vectoriel de référence est celui utilisé dans le langage PostScript [Adobe Systems Incorporated90]. Une image est composée de primitives graphiques, décrites par leur topologie et par des attributs de dessin de cette topologie. Généralement, la topologie est décrite par une courbe géométrique et une règle implicite ou explicite de spécification de l'intérieur et de l'extérieur.

> Par exemple, PostScript et X définissent deux règles pour déterminer l'intérieur d'une forme décrite par une courbe géométrique. La première, appelée *paire/impaire* (*even/odd*), dit qu'un point est à l'intérieur d'une courbe géométrique fermée si une demi-droite partant de ce point et allant vers l'infini coupe la courbe un nombre impaire de fois. La deuxième, nommée *enroulement non nul* (*non zero winding*), considère l'orientation de la courbe, lorsqu'elle coupe la demi-droite et dit que le point est à l'extérieur si la courbe coupe la droite autant de fois en la traversant de gauche à droite que de droite à gauche. Une

> autre manière de l'exprimer est d'imaginer une personne parcourant la courbe d'une extrémité à l'autre, lorsqu'elle croise la demi-droite avec le point à sa droite autant de fois qu'elle la croise avec le point à sa gauche, alors le point est à l'extérieur.

Pour des primitives simples comme des ellipses ou des polygones, la règle est inutile.

Les attributs graphiques sont généralement la couleur de remplissage, la couleur du trait de contour, l'épaisseur du trait de contour. Parfois, au lieu d'une couleur, un motif peut être utilisé. Les traits peuvent être pointillés ou continus. Des modèles vectoriels plus complexes utilisent des attributs graphiques permettant de décrire des dégradés de couleurs ou des transparences. Dans leur version primitive, les modèles graphiques des systèmes de fenêtrage n'offrent pas cette richesse. Des extensions [Neider et al.93, Womack92] permettent néanmoins leur utilisation.

Dans le modèle vectoriel, l'image finale est composée comme si les primitives graphiques étaient tracées les unes par dessus les autres, comme lorsqu'un peintre donne des coups de pinceaux.

Les primitives sont :

type Nombre, **État_Graphique**, **Courbe**, **CodeCaractère**;

fn *largeur*(**ImageVecto**): **Nombre**
fn *hauteur*(**ImageVecto**): **Nombre**
proc *remplis_courbe*(**ImageVecto**, **État_Graphique**, **Courbe**);
proc *trace_courbe*(**ImageVecto**, **État_Graphique**, **Courbe**);
proc *dessine_pixels*(**ImageVecto**, **État_Graphique**, **ImagePixel**, **Point**);
proc *dessine_caractère*(**ImageVecto**, **État_Graphique**, **CodeCaractère**, **Point**);

Au lieu de passer plusieurs paramètres spécifiques à ces fonctions, le type **État_Graphique** est généralement utilisé pour plusieurs raisons :

- il évite de passer une quantité importante de paramètres aux fonctions lorsque le modèle graphique utilise beaucoup d'attributs ;
- il permet aussi d'étendre le modèle graphique en lui rajoutant des attributs tout en assurant la compatibilité ascendante ;
- il permet d'utiliser les mêmes fonctions sur des surfaces virtuelles de natures différentes, comme un écran et une imprimante, en définissant certains attributs sur l'un et pas sur l'autre.

Les attributs de l'état graphique sont généralement :

couleur_remplissage: **Couleur**;

couleur_trait: **Couleur**;
épaisseur_trait: **Nombre**;
règle_remplissage: {`paire_impaire`, `nombre_enroulement`};
pointillés: **tableau de Nombre**;
police: **PoliceDeCaractère**;

Le type **Nombre** est décrit dans la section suivante. Le type **Courbe** est généralement un polygone et peut être une courbe de Bézier [Bézier70] dans Display PostScript et le type **CodeCaractère** dépend du système de codage utilisé pour décrire la police de caractères.

2.5.2.1 Schématique et réalisme

Nous ferons la distinction entre le graphique vectoriel *schématique* et le graphique vectoriel *réaliste*. Foley [Foley et al.90, page 945] les appelle décoratifs (*cosmétique*) et géométriques. Lorsqu'une primitive graphique est agrandie, tournée ou déplacée, si un attribut graphique apparaît transformé conformément à la primitive graphique, il est géométrique, sinon, il est décoratif. Parmi les attributs graphiques, ceux qui peuvent être géométriques ou décoratifs sont l'épaisseur des traits, la taille des pointillés, leur fréquence et les motifs de remplissage. Le graphique vectoriel schématique demande beaucoup moins de ressources que le graphique vectoriel réaliste, c'est la raison pour laquelle c'est celui qui est implanté dans la majorité des boîtes à outils. Il est toujours coûteux de passer, dans un sens comme dans l'autre, du graphique vectoriel schématique au graphique vectoriel réaliste.

Les attributs concernés par cette distinction sont : l'épaisseur des traits, la taille et la fréquence des pointillés, les images pixellaires (pour les textures), les hachurages (orientation et fréquence) et les caractères. Dans la description de l'état graphique, nous avons utilisé le type **Nombre** qui est généralement un entier lorsque le modèle est schématique et un réel lorsque le modèle est réaliste.

2.5.2.2 Graphique relâché

Un modèle graphique *relâché* ignore certains attributs graphiques. Les attributs existent dans l'état graphique et doivent être spécifiés mais ils sont ignorés. Un modèle graphique schématique ou réaliste peut être relâché. Ce modèle est utilisé dans les éditeurs graphiques interactifs comme Illustrator pour éviter que les objets graphiques composés de zones remplies ne cachent les objets graphiques placés derrière eux et pour améliorer les performances d'affichage et de manipulation interactive.

2.5.2.3 Le texte

Toutes les boîtes à outils permettent de manipuler des objets graphiques représentant du texte, mais l'édition de texte nécessite des mécanismes spécifiques. Dans sa version la plus simple, utilisée entre autre par les éditeurs de texte comme EMACS [Stallman87], le texte est représenté par une chaîne de codes, certains de ces codes correspondant à un caractère et d'autres à une commande de placement. La visualisation nécessite une police de caractère et un algorithme de composition. La police de caractère associe à chaque code de caractère une image et des dimensions, tandis que l'algorithme utilise les dimensions et les codes de commande pour calculer la position de chacun des caractères. De ce point de vue, un texte n'est rien d'autre qu'un objet graphique composite contenant des objets graphiques élémentaires (les images des caractères) positionnés suivant une règle implicite (l'algorithme de composition).

2.5.3 Modèle 3D

Le modèle le plus fréquemment utilisé pour décrire des scènes 3D [Foley et al.90] distingue la topologie et les attributs des objets graphiques, positionnés dans l'espace. La topologie est décrite soit de manière constructive à partir d'opérations sur des formes primitives (CSG), soit à partir de descriptions géométriques et de règles pour définir l'intérieur et l'extérieur des formes.

Les attributs graphiques sont associés aux formes. Plus un système est photoréaliste et plus le nombre d'attributs est important. En plus des attributs liés aux objets graphiques, l'environnement dans lequel les objets sont placés peut aussi avoir des attributs, comme un éclairage ambiant ou une opacité de l'air. Enfin, le point de vue a aussi des attributs, non seulement de position mais aussi parfois de distance focale (pour imiter les défauts des vraies caméras), d'ouverture etc.

Il existe plusieurs méthodes de description des scènes 3D. La méthode utilisée par OpenGL [Neider et al.93] consiste à décrire la géométrie en associant des attributs graphiques (physiques) à cette géométrie. Les fonctions sont alors :

proc *commence_scene*(**Image3D**);
fn *termine_scene*(**Image3D**): **Scene**;
proc *commence_objet*(**Image3D**);
fn *termine_objet*(**Image3D**): **Objet**;
proc *ajoute_objet*(**Image3D**, **Objet**);
proc *commence_courbe*(**Image3D**);
fn *termine_courbe*(**Image3D**): **Courbe**;
proc *ajoute_courbe*(**Image3D**, **Courbe**);
proc *ajoute_segment*(**Image3D**, **Segment**);
proc *met_dernier_segment*(**Image3D**, **AttributDeSegment**);

proc *met_dernière_position*(**Image3D**, **AttributDePosition**);
proc *met_attribut*(**Image3D**, **AttributDObjet**);
proc *met_caméra*(**Image3D**, **AttributDeCaméra**);
proc *affiche*(**Image3D**, **Scene**);

Avec ces primitives, une image 3D est une scène, qui est composée d'une liste d'objets, eux-mêmes composés d'une liste de segments. Chaque segment peut avoir des attributs (épaisseur de trait, couleur de trait, pointillé, etc.) et peut être transformé (un des attributs définissable sur tous les objets est la transformation géométrique qui lui est appliquée). Finalement, une scène est vue à travers une caméra (point de vue), définie par des attributs spécifiques (angle d'ouverture, position, orientation, modèle de projection, etc.).

2.5.3.1 Schématique et réalisme

Comme en 2D, certains domaines 3D ne nécessitent que des attributs schématiques, permettant de distinguer les objets entre eux et d'associer des attributs symboliques à des objets 3D. Par exemple, en CAO, les objets sont modélisés pour être ensuite fabriqués réellement. Les attributs graphiques qui leur sont associés ne sont que rarement utilisés pour de l'affichage photo-réaliste. Ils sont donc définis avec une couleur uniforme par constituant. Il est ainsi possible de discerner un boulon d'une pièce. De même, un hachurage suffit à distinguer deux parties de surface.

En revanche, le graphique 3D photo-réaliste utilise énormément d'attributs graphiques qui se rapportent à des caractéristiques physiques des matériaux modélisés.

2.5.3.2 Graphique relâché

Le mode graphique relâché est beaucoup utilisé dans les modeleurs 3D qui doivent afficher les objets graphiques et permettre leur manipulation en temps réel. Une des plus communément employé est nommé « affichage en fil de fer ». Il consiste à n'afficher que les segments.

2.6 Synthèse

Bien que les systèmes de fenêtrage existent depuis plus de dix ans, de nouveaux besoins sont apparus récemment concernant la gestion des optimiseurs graphiques et des dispositifs d'entrée variés. Tous deux doivent être pris en compte dès les couches basses dans les modèles architecturaux participant à la réalisation d'éditeurs graphiques modernes.

Un éditeur peut donc utiliser le modèle graphique le plus approprié pour visualiser les objets graphiques qu'il permet d'éditer. Pour se plier aux conventions de présentation et de comportement des autres applications graphiques fonctionnant avec le système de fenêtrage, il doit aussi utiliser des objets de l'interaction qui lui sont fournis par les boîtes à outils, dont nous discutons au chapitre suivant. La cohabitation de ces deux contextes graphiques n'est généralement pas prévue, chaque modèle graphique étant hégémonique et ignorant l'existence des autres.

Chapitre 3

Les boîtes à outils

Une boîte à outils est une bibliothèque de fonctions qui :

- permet d'accéder au système de fenêtrage à travers une API (*Application Programmer's Interface*) ;
- définit des objets de gestion de l'interaction, généralement appelés *Widget*s, *Controls*, objets de contrôle ou objets interactifs.

Le système X sépare clairement le niveau API — appelé *X Intrinsics* [Nye88] — et boîte à outils— appelé *X Toolkit* [Flanagan92] — tandis que les autres boîtes à outils présentent les deux couches dans une même bibliothèque de programmation.

3.0.1 Les *Widget*s

Un *Widget* est un objet ayant une apparence graphique et un comportement interactif, dont l'implantation repose sur le système de fenêtrage. Ce sont les briques de base destinées à construire les applications graphiques interactives. Ils permettent en particulier :

- de factoriser des fonctionnalités génériques (affichage de texte, d'icones, saisie de texte, etc.) ;
- d'imposer la conformité à une charte graphique (*look*) et
- d'adopter un comportement cohérent (*feel*) dans toutes les parties de l'application et entre applications.

Ces *Widget*s peuvent être de deux catégories : conteneurs ou terminaux. Une application graphique est composée de fenêtres subdivisées en *Widget*s qui s'organisent de deux manières :

- par inclusion,

– par juxtaposition.

Les conteneurs permettent l'inclusion et organisent la juxtaposition des *Widget*s qu'ils contiennent en suivant une sémantique de placement (décrite en 3.1.2.1).

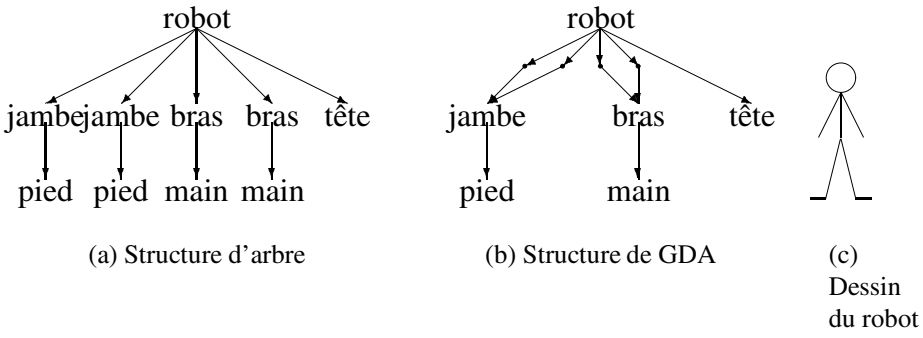
Trois types de fenêtres peuvent exister : permanentes, de dialogue ou de menu. Une application graphique a toujours au moins une fenêtre permanente. Les fenêtres de dialogue se distinguent des fenêtres permanentes par leur durée de vie qui est généralement courte. Elles servent à donner des indications temporaires comme un message d'erreur ou à recueillir des informations ponctuelles. En dehors de ces caractéristiques, elles se comportent comme des fenêtres permanentes et sont composées des mêmes *Widget*s. Les menus sont des fenêtres qui proposent généralement un choix et qui restent visibles tant que le choix n'est pas fait et tant que le curseur reste à l'intérieur de la zone de la fenêtre. Certaines boîtes à outils ne permettent pas d'utiliser tout type de *Widget* dans un menu.

3.0.2 Implantation d'un éditeur dans une boîte à outils

Les *Widget*s des boîtes à outils ne permettent pas de réaliser un éditeur, ils sont plutôt destinés à représenter des structures de données simples sous des formes stéréotypées comme des labels, des boutons poussoirs, des menus, des listes de mots etc.

Certaines boîtes à outils disposent de *Widget*s gérant des formes de graphique structuré qui peuvent être utilisées telles quelles pour des éditeurs simples. Tk [Ousterhout94] offre le *Widget Canvas* qui permet de créer et manipuler des objets graphiques simples (lignes, ellipses, polygones, petites images, etc.) et plusieurs *Widget*s Xt permettent de gérer des formes spécialisées de graphique structuré (Graphe [Beaudouin-Lafon91], Arbre, Graphique schématique, etc.). Les chances de trouver un *Widget* qui convienne à un domaine d'éditeur sont plus grandes mais la variété des domaines et les besoins spécifiques des utilisateurs nécessitent une trop grande variété de configurations pour qu'un répertoire de *Widget*s réponde à tous les besoins.

Pour réaliser un éditeur à ce niveau, deux possibilités existent : définir un nouveau *Widget* ou utiliser un *Widget* particulier, dont toutes les boîtes à outils disposent, parfois appelé « surface de dessin ». Cette surface de dessin permet de gérer les événements et l'affichage à un niveau très proche du système de fenêtrage. Au sein de la surface de dessin, le concepteur d'application graphique doit implanter entièrement les mécanismes de gestion du graphique et de l'interaction qui répondent à ses besoins en s'appuyant uniquement sur les primitives du système de fenêtrage et en respectant les contraintes imposées par la boîte à outils. Dans tous les cas, un *Widget* sert de passerelle entre la boîte à outils et la surface de dessin



(a) Structure d'arbre (b) Structure de GDA (c) Dessin du robot

FIG. 3.1 - *Représentation d'une structure graphique*

où apparaissent les objets graphiques gérés par l'éditeur.

Nous avons réalisé plusieurs passerelles de ce type sous X, initialement entre Xt et la boîte à outil Xtv [Beaudouin lafon et al.91, Beaudouin lafon et al.90], puis entre Xt et Fresco (intégrée à la distribution de Fresco). Un problème important est que les boîtes à outils ne sont pas conçues pour gérer des *Widget*s très sophistiqués. Les mécanismes et stratégies des boîtes à outils généralistes ne sont pas d'une grande utilité pour la conception et la réalisation de l'intérieur du *Widget* passerelle. Certaines boîtes à outils proposent néanmoins quelques mécanismes unificateurs entre les *Widget*s et la gestion d'objets graphiques généraux.

InterViews [Linton et al.89] et plus récemment Fresco [Linton et al.93] sont des boîtes à outils généralistes, indépendantes du système de fenêtrage, et unifiant la notion de *Widget* et d'objet graphique. Dans les sections qui suivent nous décrivons plus précisément les mécanismes de gestion du graphique utiles aux éditeurs.

3.1 Gestion du graphique à état

InterViews et Fresco définissent des mécanismes utilisables pour gérer les structures graphiques de pratiquement tout éditeur. Le modèle actuellement employé fonctionne aussi bien pour le 2D que pour le 3D et consiste à représenter les objets graphiques sous la forme d'un arbre (voir figure 1(a)) ou d'un graphe direct acyclique (GDA, voir figure 1(b)). Nous appellerons cette architecture « à structure stable » (traduction de *retained-mode graphics*).

Ce modèle est le plus couramment employé mais ne concerne pas toutes les applications. En particulier, pour la visualisation de phénomènes complexes animés, la structure graphique n'est pas gardée sous forme d'objets stables mais est directement dessinée sur la surface graphique avant d'être recalculée pour l'image suivante. Foley [Foley et al.90] cite l'exemple du calcul d'écoulement d'un fluide

autour d'un volume et sa visualisation. Nous ne prendrons pas en compte ce genre d'applications.

Ce modèle graphique à structure stable est issu de l'évolution de standards comme GKS et PHIGS. Dans ces standards, les primitives graphiques sont composées de sous-primitives suffisamment simple pour être dessinées directement par une couche matérielle dédiée. Dans les modèles graphiques actuels, le processus de calcul de l'image s'appelle le « pipeline graphique » et se décompose en plusieurs étapes qui transforment la structure graphique. La prise en compte de matériel graphique dédié se fait dans les dernières étapes du processus.

Dans cette architecture les nœuds appartiennent à deux catégories : conteneurs et terminaux. Les conteneurs servent à grouper plusieurs éléments ou à appliquer une transformation, généralement géométrique mais parfois comportementale, à leurs descendants. Les terminaux sont des objets graphiques qui gèrent une primitive du modèle graphique. La structure graphique est référencée par son nœud racine.

> Par exemple, dans la figure 1(b), chaque jambe et chaque bras du robot sont stockés sous forme d'un conteneur qui applique une transformation géométrique aux sous-nœuds. Dans le cas des jambes, la première transformation peut être l'identité tandis que la seconde applique une symétrie par rapport à l'axe du corps du robot.

Un conteneur appliquant une transformation comportementale est souvent utilisé pour la gestion de l'interaction. Il permet de modifier le comportement d'un objet graphique lors de la désignation ou de la gestion des événements (décrits en 3.1.2.7).

Les opérations définies sur les objets graphiques généraux sont les suivantes :

- création,
- placement,
- affichage,
- ajout/suppression d'une partie,
- affichage/réaffichage,
- gestion des événements.

L'architecture logicielle décrite par [Foley et al.90] spécifie que les objets graphiques, organisés sous forme hiérarchique, sont stockés dans une base de donnée centrale. Chaque objet graphique contient une transformation qui est appliquée aux objets graphiques qu'il contient (qu'ils soient terminaux ou pas). Un GDA 3D gérera généralement des matrices de transformation homogènes 4×4 tandis qu'un GDA 2D gérera des matrices 3×3. Chaque nœud contient aussi le descripteur d'une structure graphique qui est soit une primitive pour un objet graphique terminal, soit une liste de pointeurs vers d'autres nœuds pour un conteneur.

3.1.1 Modèle graphique à objets

Les méthodes de programmation par objets offrent de grands avantages pour le graphique [Laffra91]. L'unification du graphique structuré et de la construction d'interfaces a commencé à se faire à partir de 1990, lorsque le domaine des interfaces, adepte et promoteur des méthodes à objets, a intégré les architectures issues du monde graphique. Des boîtes à outils comme GEO++ [Wisskirchen91] utilisent le modèle du graphique structuré pour gérer l'affichage d'éléments de l'interface comme les boutons poussoirs ou les affichages de texte.

Déjà à partir de 1988, ET++ [Weinand et al.88] et InterViews font cohabiter des objets graphiques avec des objets de l'interface comme des boutons, des boîtes de menus ou des ascenseurs. Cependant, deux types d'objets subsistent, avec des comportements sensiblement différents.

Les objets de l'interface et les modèles graphiques 2D, textuel et 3D ont été ensuite unifiés, en 2D avec la version 3.0 d'InterViews, Fresco et en 3D avec OpenInventor [Neider et al.93, Open Inventor Architecture Group94]. Il tendent à abstraire de plus en plus la structure graphique et à l'organiser sous forme de groupes d'objets abstraits coopérants et non pas d'une structure de données dont l'implantation est publique.

Les objets graphiques s'affichent sur une surface (en 2D) ou dans un volume (en 3D). Les domaines d'unification sont:

- gestion d'un GDA en 2D,
- gestion d'objets élastiques en 3D,
- gestion pratiquement identique des objets graphiques et des objets de l'interface.

Le partage de sous-structure a été utilisé très tôt en 3D car l'occupation mémoire des objets graphiques est sensiblement plus importante qu'en 2D. Paul Calder [Calder et al.90] a montré qu'un tel partage de données pouvait être fait sans pénalités de performances en 2D. Pour cela, il définit des objets légers, les **Glyph**s, qui calculent leur information de position et de taille à la volée plutôt que de les stocker. Ils peuvent ainsi être partagés et former une structure de graphe direct acyclique plutôt que d'arbre. La boîte à outils Fresco utilise ces objets en 2D et OpenInventor [Open Inventor Architecture Group94] utilise le même type d'objet léger en 3D.

Fresco utilise aussi la structure des **Glyph**s pour gérer chaque caractère dans un éditeur de texte. En effet, le texte peut être vu comme une structure graphique 2D particulière qui utilise énormément le partage de structures graphiques : chaque caractère identique est partagé dans la structure graphique.

Fresco utilise en 3D la même architecture d'objets qu'en 2D, avec des dimensions élastiques.

```
type Glyph = classe
début
    glyphs: ListeDe(Glyph);

    fn requiert(): Dimension;
    fn alloue(Canvas, Région): Région;
    proc dessine(inout Canvas, in Région);
    proc need_resize();
    proc need_redraw();
fin;
```

```
type Axe = (X, Y);
type Dimension = classe;
début
    taille_naturelle: tableau[X..Y] de Réel;
    taille_max:       tableau[X..Y] de Réel;
    taille_min:       tableau[X..Y] de Réel;
    alignement:       tableau[X..Y] de Réel;
fin;
```

FIG. 3.2 - *L'interface nécessaire à la gestion unifiée de l'affichage dans Fresco.*

3.1.2 Gestion de l'affichage

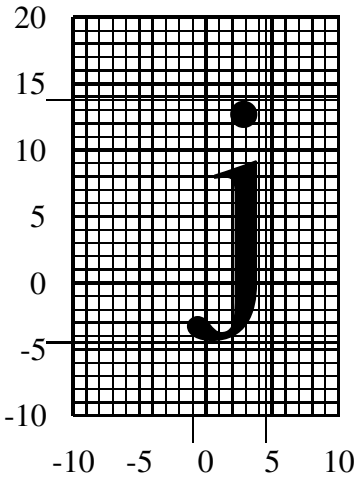
Le travail de Steven Tang [Tang et al.94] a permis d'unifier la gestion d'un GDA 2D et 3D et a permi le calcul de leur placement de manière performante. Les objets participant à l'affichage sont les suivants :

- la fenêtre (*Window*),
- la surface virtuelle (*Canvas*),
- le **Glyph** (décrit en figure 3.2),
- le **Glyph** racine.

Le schéma général est le suivant : un GDA de **Glyph**s est créé. Une fenêtre est ensuite créée pour afficher le GDA référencé par sa racine. Lorsqu'elle doit effectivement s'afficher, la fenêtre calcule la taille qu'elle doit avoir en demandant au **Glyph** racine de calculer sa dimension, puis elle crée une surface virtuelle ayant cette dimension sur laquelle elle demande au **Glyph** racine de se dessiner.

La structure d'un **Glyph** lui permet d'être considérée, selon les cas, comme ayant une taille ou une position. Dans Fresco, un **Glyph** a pour chaque axe une taille *naturelle* (un nombre réel positif), une taille minimum, une taille maximum et une position de référence sous la forme d'un rapport sur sa taille (par exemple, 0.5 indique que le point de référence est placé au milieu). La figure 3.2 décrit la façon dont Fresco code ces informations : l'idée est de représenter la taille d'une façon intrinsèque, sans utiliser de coordonnées cartésiennes inadaptées à la description d'objets graphiques élastiques autour d'une dimension naturelle. Un exemple est donné en figure 3.3.

La structure **Dimension** contient la description de la dimension élastique par rapport à un point de référence. Elle peut aussi décrire un objet rigide et le point de référence peut être interprété comme l'origine. Il répond donc aux besoins des



Coordonnées cartésiennes et largeur				
	min	max	largeur	
x	-1.0	4.5	5.5	
y	-4.5	13.8	18.3	

Requisition				
	natural	min	max	align
x	5.5	5.5	5.5	-0.18
y	18.3	18.3	18.3	-0.25

FIG. 3.3 - *le caractère « j » avec ses dimensions rigides exprimées sous forme cartésienne et de* Requisition.

objets de l'interface — qui peuvent être élastiques — et des objets du graphique structuré — qui sont le plus souvent rigides. Dans ce dernier cas, la structure n'est pas optimale mais son coût peut être réduit par l'utilisation d'un objet passerelle.

3.1.2.1 Placement

Le placement des objets graphiques est la méthode que va utiliser une application graphique pour calculer les positions et dimensions des objets graphiques qui la composent, statiquement aussi bien que dynamiquement. Il se fait par juxtaposition ou superposition des objets graphiques. Les positions de chaque objet, ainsi que la taille des objets ainsi placés, sont spécifiés de deux façons dans les systèmes actuels : explicitement ou par calcul.

Le placement peut aussi varier dynamiquement ou interactivement. Des objets graphiques peuvent apparaître ou disparaître, les fenêtres et les objets graphiques peuvent être redimensionnés et déplacés, etc.

Placement explicite Les premiers systèmes graphiques interactifs comme Smalltalk gèrent le placement des objets graphiques de façon explicite. La position des objets est spécifiée par le programmeur d'application. Les systèmes actuels comme Visual Basic perpétuent ce mode de fonctionnement dans lequel chaque objet graphique a une position et une dimension explicites. Bien entendu, ces dimensions et positions peuvent être modifiées par programmation. Un événement

est produit lors du changement de taille d'une fenêtre, ce qui peut déclencher l'appel à une procédure et le calcul de nouvelles positions et dimensions pour les objets de l'interaction.

Placement calculé Dans de nombreux cas, les objets graphiques de l'interface peuvent être placés en utilisant des règles simples, par exemple le fait qu'un objet graphique soit à gauche d'un autre ou au-dessous. Le placement déclaratif des objets graphiques de l'interface a fait l'objet de nombreuses recherches. Les plus générales ont porté sur l'expression du placement sous forme de contraintes arithmétiques [Borning79, Moloney et al.89, Leler88]. Pour les interfaces, des contraintes moins générales ont été jugées suffisantes dans Garnet [Myers89]. L'idée de ces systèmes est d'exprimer par des contraintes unidirectionnelles (nommées *formules*) les relations géométriques (au moins) entre les objets graphiques de l'interaction. Par exemple, pour décrire la juxtaposition horizontale de deux objets graphiques, la position en X minimal du deuxième objet est contrainte à avoir la valeur de la position maximale en X du premier objet, tandis que la position des points de références est contrainte à une même valeur en Y.

Le placement calculé est depuis longtemps utilisé dans le traitement du texte, où les caractères sont placés automatiquement par un algorithme qui suit les règles de composition de texte, en fonction de certains paramètres structurels (largeur de colonne, taille de la police, etc.).

Certains systèmes, comme Fresco et ET++, utilisent un mode de placement calculé qui est pratiquement identique à celui que TEX [Knuth79] utilise pour la composition de texte. Il s'avère que ce type de placement convient pour beaucoup de cas pratiques qui sont très difficiles à exprimer à partir de systèmes de contraintes généralisées.

D'autres systèmes comme la X Toolkit [McCormack88] ou Tk utilisent des spécifications de contraintes *ad-hoc* pour gérer le placement. Ilog Views [Ilog94] permet d'utiliser une grille de mise en écran de manière similaire à la mise en page.

3.1.2.2 Unification du placement avec les langages à objets

Tous les systèmes de placement utilisant des contraintes acycliques calculent le placement en trois étapes :

- demande de la dimension des objets graphiques,
- résolution des contraintes et, le cas échéant, négociations,
- attribution définitive des places.

Seule la *X Toolkit* requiert une négociation lors de la seconde étape, les autres systèmes obtiennent suffisamment d'information à la première phase pour ré-

soudre les contraintes directement à la seconde phase. Cette information est contenue dans le type de donnée utilisé pour spécifier la dimension. Par exemple, les widgets de Tk ont une taille minimale et la possibilité de s'étendre en X ou en Y. Fresco implante la gestion du placement calculé comme suit :

Demande de la dimension élastique Lors de la première étape, chaque **Glyph** doit remplir une structure de **Requisition**. Un **Glyph** peut indiquer que la taille dans une des dimensions est indéfinie, s'il ne la gère pas. En plus de la taille, un point de référence pour l'alignement doit être spécifié.

Résolution des contraintes dans les conteneurs Un **Glyph** feuille se contente de retourner une taille calculée en fonction de ses besoins, mais un **Glyph** conteneur doit calculer sa taille en fonction de la taille de ses fils et de la façon dont il place ses fils les uns par rapport aux autres.

Par exemple, un conteneur qui implante un modèle de dessin structuré traditionnel gère deux attributs : ses fils et une transformation géométrique qu'il applique à ces fils. Le calcul de la taille de ce conteneur est simplement l'union de la taille de tous les fils calculée en alignant les points de références transformés. Un conteneur qui aligne ses fils horizontalement aura comme dimension horizontale la somme des dimensions horizontales de ses fils, et comme dimension verticale le maximum des dimensions des fils, alignés sur leur point de référence vertical.

Attribution d'une place fixe Une place est définie comme une taille et une origine dans deux dimensions.

Avec la dimension élastique du **Glyph** racine d'une fenêtre, il est simple de calculer une place fixe. A priori, il suffit de fixer la taille de la place à la taille naturelle de la dimension élastique et à fixer l'origine en fonction de la référence demandée par le **Glyph**. Lors de la création d'une fenêtre, la dimension élastique sert à définir la taille par défaut de la fenêtre demandée, ainsi que les limites de réduction et d'extension que l'utilisateur peut lui affecter.

Une fois ce calcul effectué, la structure des **Glyph**s est parcourue récursivement en attribuant une place fixe à tous les **Glyph**s. Un **Glyph** terminal doit se dessiner par rapport à l'origine de la place tandis qu'un **Glyph** conteneur doit calculer la place fixe de chacun de ses fils en fonction de la place fixe qu'il a reçu, de la dimension de ses fils et de sa sémantique de placement. Il peut alors rappeler récursivement chacun de ses fils en lui passant la valeur de la portion de place qui lui est attribuée.

3.1.2.3 Optimisations pour le graphique structuré

Le graphique structuré destiné à la visualisation est un cas particulier qui ne requiert pas toutes les sophistications de placement des objets de l'interaction. Les objets graphiques structurés ont une dimension fixe (ils sont rigides) et sont affichés sur le point de vue dans l'ordre de parcours de la structure pour le 2D et suivant le modèle de projection graphique pour le 3D. Ainsi, l'étape de gestion du placement calculé peut être optimisée, mais c'est surtout la gestion de la modification de la structure de données qui est simplifiée.

3.1.2.4 Affichage

Après le placement, le **Glyph** reçoit en paramètre la surface virtuelle ainsi que la place fixe calculée et doit se dessiner sur la surface virtuelle en prenant comme origine le point de référence de la place fixe.

L'affichage se fait en utilisant les fonctions de dessin passif de la surface virtuelle. Le mécanisme d'affichage est totalement indépendant du modèle graphique implanté dans cette surface virtuelle. Il pourrait s'agir d'un modèle graphique orienté pixels et 2D, ou indépendant de la résolution comme PostScript, ou 3D comme OpenGL [Segal et al.93].

Les étapes de placement et d'affichage pourraient certainement être fusionnées avec quelques difficultés. Le gain serait vraisemblablement négligeable au vu des complications que cela entraînerait.

3.1.2.5 Réaffichage

Lorsque la structure graphique est modifiée, un réaffichage doit être provoqué. Il existe deux modes de réaffichage : immédiat et différé. Le mode de réaffichage est immédiat lorsque la commande qui modifie un objet graphique provoque son réaffichage. Dans le mode de réaffichage différé, une commande qui modifie un objet graphique ne provoque pas de réaffichage mais signale que la région sur laquelle s'affiche l'objet graphique est endommagée. À un moment qu'il considère comme opportun, le système graphique va provoquer un réaffichage de tous les objets qui apparaissent dans la région endommagée.

La gestion du réaffichage immédiat donne un résultat incorrect lorsque des objets sont opaques et peuvent être superposés. L'objet modifié pouvant être partiellement caché par un autre objet, son affichage direct provoquera une incohérence dans l'affichage. La gestion explicite de cette superposition est extrêmement coûteuse.

Le réaffichage différé est généralement préféré par les boîtes à outils qui gèrent des objets graphiques structurés. Il possède de nombreux avantages par rapport au

réaffichage immédiat :

- les objets graphiques superposés sont convenablement gérés ;
- les objets graphiques n'ont pas besoin d'offrir deux fonctions pour l'affichage et pour le réaffichage : les mécanismes d'affichage et de réaffichage sont identiques ;
- même lorsque plusieurs de ses attributs sont modifiés, un objet est réaffiché une seule fois.

En revanche, le réaffichage d'un objet graphique simple peut déclencher une chaîne de réaffichages compliqués lorsque l'objet graphique simple et des objets graphiques complexes sont superposés. Cet inconvénient est rédhibitoire pour des applications qui nécessitent une mise à jour de l'affichage en temps réel comme la vidéo ou certaines applications où le temps est critique.

Un autre défaut du réaffichage différé est qu'il provoque un *flash* sur la zone réaffichée. Celle-ci doit être effacée puis tous les objets graphiques qui apparaissent sur la zone sont réaffichés dans l'ordre, ce qui provoque une succession de changements de couleurs à l'écran. Sur une machine lente, le dessin de chaque forme est perçu distinctement tandis que sur une machine rapide, seul un flash est perçu.

Pour éviter ce désagrément, plusieurs boîtes à outils utilisent une technique mise au point pour l'animation rapide 2D et 3D : le *double buffering* [Foley et al.90, page 180]. Pour l'animation, l'idée est d'utiliser deux images, une qui est affichée et l'autre en mémoire dans laquelle l'image suivante est calculée. Le passage de l'image affichée à la suivante peut ainsi se faire instantanément. Cette méthode permet entre autre une animation plus fluide que si les images étaient calculées directement sur l'écran.

L'utilisation du *double buffering* dans les interfaces ajoute un grand confort visuel, au détriment d'une occupation supplémentaire en mémoire. Plusieurs implantations existent et le temps de changement de *buffer* varie énormément suivant le matériel et la technique utilisés. Une extension de X [Gaskins93] implante le *double buffering* et décrit quelques variantes.

3.1.2.6 Ajout/modification/suppression

La modification d'une partie d'un GDA de **Glyph**s nécessite le recalcul du placement et parfois de nombreux changements dans l'affichage. Après qu'un **Glyph** ait été modifié, une fonction — dans Fresco, la fonction *need_resize* (voir figure 3.2) — est appelée qui propage la modification vers la racine, forçant un recalcul des places si nécessaire et un réaffichage.

Dans le cas général des objets de l'interaction, la modification d'un objet quelconque peut donc avoir des répercussions sur la dimension de son ou ses conteneurs et récursivement sur la dimension de tous les objets graphiques. En revanche, la modification d'un élément dans une structure graphique traditionnelle ne modifie qu'exceptionnellement la taille de la zone de graphique structuré. De plus, cette taille est généralement visualisée à travers un point de vue dont la taille est fixe. Toute modification des dimensions à l'intérieur ne produit aucun changement de taille ou de place à l'extérieur. Pour éviter de remonter inutilement dans le GDA des **Glyph**s, Fresco traite spécialement la propagation des modifications des objets rigides du graphique structuré [Linton et al.94] en insérant un nœud particulier en racine des objets implantant le graphique structuré simple (nommé *Figure* dans Fresco).

3.1.2.7 Modèle de gestion des événements

La gestion des événements se divise en deux parties : la désignation et le traitement de l'événement.

La désignation Avant d'interagir avec les objets graphiques, il est nécessaire de déterminer l'objet avec lequel on veut interagir. L'interaction se fait à travers un dispositif qui peut être positionnel ou non. Avec les dispositifs positionnels (comme une souris, une track-ball ou un stylet de tablette), l'utilisateur de l'application pointe l'objet qu'il veut manipuler et le système doit trouver de quel objet graphique il s'agit. Avec un dispositif non positionnel comme un clavier ou un système de reconnaissance vocale, aucune information n'est produite par le dispositif pour localiser le sujet sur lequel l'interaction doit se faire. Les systèmes graphiques utilisent alors la notion de « Focus », état global de l'application qui permet d'obtenir le sujet de l'interaction. Le focus n'est pas nécessairement lié à une zone fixe de l'application ou à un objet graphique, il peut aussi utiliser comme destinataire d'événement non positionnel l'objet graphique placé sous le pointeur.

Comme on peut le voir sur la figure 3.4, il est nécessaire de trouver quel est l'objet graphique qu'un utilisateur désigne sur l'écran à partir d'un dispositif de pointage. Deux techniques existent pour réaliser cette mise en relation : la recherche dans l'arbre des objets graphiques au moment du pointage et l'utilisation d'une table d'association entre chaque pixel et l'objet qui a servi à l'affecter pour créer l'image finale.

La première technique revient à demander à chaque objet graphique s'il modifie le pixel à la position du pointeur. En le demandant aux objets dans l'ordre d'affichage, le dernier objet qui a modifié le pixel est celui qui est sous le pointeur.

La seconde technique est très simple à implanter lorsque la boîte à outils maîtrise l'affichage de chaque objet au niveau du pixel. En plus de la table des pixels, la

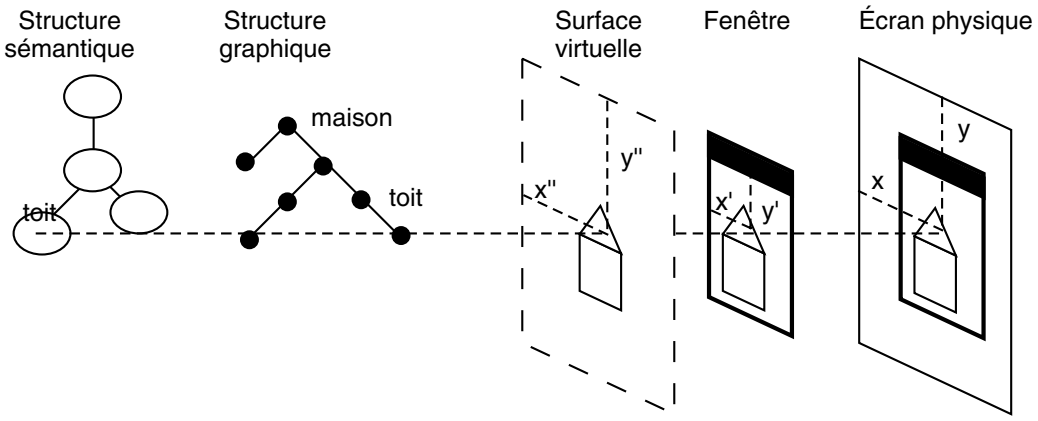


FIG. 3.4 - *Les différents niveaux structurels de la désignation (schéma de J. Coutaz).*

boîte à outils a besoin de mettre à jour une table des objets qui a la même taille que la fenêtre graphique et dont chaque entrée pointe vers le dernier objet qui a modifié la valeur du pixel. Cette technique est préconisée par le manuel d'OpenGL [Neider et al.93, page 389] 3D lorsqu'on dispose d'un accélérateur graphique.

Comme pour la gestion de l'affichage, le modèle architectural de gestion de la désignation s'est amélioré et clarifié grâce à l'utilisation des langages à objets. Dans GKS et PHIGS, ce sont les objets graphiques eux-même qui gèrent la désignation. Dans les systèmes orientés objets, les structures graphiques décident si elles interceptent un événement mais délèguent sa gestion à un autre objet, nommé *handler* dans InterViews. Ce découpage permet de découpler la structure graphique de la structure logique de gestion des événements : tout un groupe d'objets graphiques peut être géré par le même *handler* ou un objet graphique peut déléguer la gestion à divers *handles* en fonction du contexte.

Traitement de l'événement Une fois que l'objet graphique cible du pointage est trouvé, l'envoi d'événement peut se transformer en un envoi de message plus structuré, contenant le type de l'événement et ses caractéristiques.

3.1.3 Synthèse sur le graphique à état

La gestion des objets graphiques et des objets de l'interaction peut se faire de façon unifiée dans une boîte à outils comme Fresco. Les mécanismes présentés en 2D fonctionnent aussi en 3D, ce qui nous permet de penser qu'une structure de données générique peut servir de base à la grande majorité des éditeurs. Cependant, toutes les boîtes à outils ne font pas cette unification. Pour les utiliser, il est alors indispensable d'avoir recours à un *Widget* particulier qui sert de passerelle

entre le monde de la boîte à outils et le monde du graphique structuré spécialisé pour l'éditeur. Cette solution est coûteuse car elle duplique une grande partie des mécanismes de gestion des objets graphiques.

La principale faiblesse des boîte à outils concerne la gestion de l'interaction. Aucun mécanisme de haut niveau n'est proposé pour gérer des suites d'événements (d'actions) et des modalités d'interaction. Certains outils de construction d'interface, que nous étudions dans le chapitre suivant, apportent des réponses à ce problème.

Chapitre 4

Outils de contruction d'interfaces

Plutôt que de partir des mécanismes de bas niveau des boîtes à outils pour bâtir un éditeur, des outils de construction d'interface permettent de concevoir cet éditeur par composition et spécialisation de mécanismes de haut niveau. Le premier modèle architectural de conception d'éditeurs vient de Smalltalk-80. Des structures génériques spécialisables d'éditeurs ont ensuite été décrits de façon opérationnelle dans des langages à objets. Ces structures sont aussi appelées squelettes d'applications car elles poussent la réutilisation de classes à un point où quelques lignes suffisent à créer un éditeur entier. Nous allons décrire ces architectures afin de voir quels types d'éditeurs elles permettent de construire, leurs avantages et leurs limites.

Plusieurs critères peuvent être évalués pour comparer les caractéristiques respectifs de chaque architecture. Nous retiendrons les suivants :

Temps d'apprentissage : évaluation du temps requis par un professionnel pour comprendre l'architecture. Ce temps étant difficile à quantifier objectivement, nous prendrons en compte le nombre d'abstractions novatrices utilisées par l'architecture.

Temps de construction : évaluation du temps requis par un professionnel pour bâtir une application en s'appuyant sur l'architecture. Ce temps dépend de la nature de l'application et de son adéquation à l'architecture. Nous décrirons le cas où l'architecture est prévu pour l'application et le cas contraire.

Méthodologie de construction : existence d'une méthodologie prônée par les concepteurs de l'architecture.

Modèle du graphique : facilité de gérer du graphique 2D pixellaire, 2D vectoriel ou 3D avec l'architecture.

Gestion de dispositifs : possibilité de gérer des dispositifs d'entrée divers en utilisant l'architecture.

Modularité de l'architecture : facilité de modifier une partie de l'architecture sans avoir à modifier d'autres parties.

Extensibilité des applications : facilité d'étendre ou de modifier une application.

Compacité du code source : nombre de lignes de code source nécessaires pour développer une application. La variation de ce nombre en fonction de l'application est généralement plus intéressante que le nombre absolu pour une application donnée.

4.1 Architecture MVC

Un des principes de l'environnement Smalltalk est le suivant : « Choisir un petit nombre de principes généraux et les appliquer uniformément » [Ingalls83]. L'architecture MVC en est un excellent exemple.

MVC [Krasner et al.88] est une triade de classes Smalltalk destinée à uniformiser le modèle architectural des objets et de leur représentation graphique dans l'environnement de programmation Smalltalk-80.

Dans cette architecture, trois types d'objets collaborent :

– le *Modèle* est l'objet sémantique dont on veut visualiser et éditer graphiquement une facette ;

– la *Vue* est l'objet graphique, responsable de la visualisation et de l'édition graphique d'une facette du modèle ;

– le *Contrôleur* est l'objet qui gère le dialogue entre le modèle et la vue. Il assure que la vue présente une version à jour du contenu du modèle et que, après interaction graphique, le modèle soit convenablement modifié.

MVC est donc un groupe de trois classes concrètes de Smalltalk, mais ce terme désigne maintenant le modèle sous-jacent, basé sur l'existence de trois catégories d'objets pour gérer l'interaction, et qui s'avère très efficace pour la conception et la réalisation de modules d'éditeurs.

L'architecture MVC est généralement étendue afin que plusieurs vues puissent être connectées à un même modèle. Plusieurs vues peuvent alors visualiser le même modèle selon différentes représentations (une jauge et un nombre textuel par exemple).

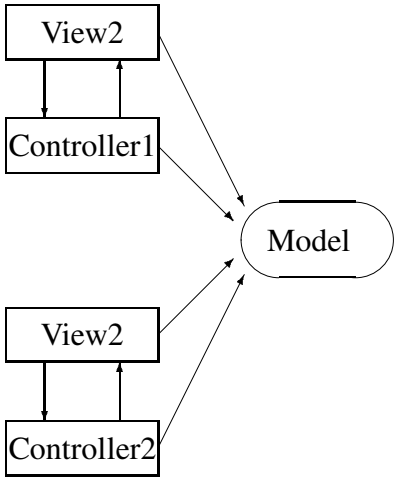


FIG. 4.1 - *Connexions explicites entre les modèles, vues et contrôleurs.*

4.1.1 Connexions entre les membres de MVC

La nature des connexions entre les trois composantes de MVC n'est pas symétrique. Dans la figure 4.1 qui décrit ces relations, on peut constater que chaque vue connaît explicitement son modèle, que le contrôleur connaît le modèle ainsi que la vue. Cependant, le modèle ne connaît ni sa ou ses vues, ni son ou ses contrôleurs. Cette propriété garantit une bonne modularité dans le développement d'applications utilisant MVC. Les modèles existent indépendamment de leur représentation et chaque application peut attacher un nombre quelconque de vues et de contrôleurs sur les modèles qu'elle veut permettre de visualiser/manipuler. Si le modèle devait connaître ses vues, la création d'un nouveau type de vue pour un modèle nécessiterait aussi la modification du modèle, cassant la modularité de MVC.

4.1.2 Implantation de MVC

Pour communiquer, un protocole est utilisé qui permet à une vue ou à un contrôleur de se connecter à un modèle. Lorsque le modèle estime qu'il a subi une modification digne d'être signalée, il s'envoie le message **changed**. Les vues et contrôleurs qui en dépendent reçoivent alors le message **update** avec en paramètre le modèle, ils peuvent alors se mettre à jour.

4.1.3 Utilisation du modèle MVC

L'environnement de Smalltalk-80 est composé de centaines de classes. Pratiquement toutes les classes responsables de la visualisation sont organisées sous

forme de vues sur un modèle plus ou moins générique. Seuls quelques objets graphiques particuliers, comme les menus, n'utilisent pas cette architecture.

Dans le cas qui nous intéresse de la fabrication d'éditeurs, l'architecture MVC implique la séparation de l'éditeur en trois parties : le modèle, qui est appelé le noyau sémantique dans les modèles d'applications graphiques, la vue, qui est une représentation du modèle, et le contrôleur, qui permet la manipulation directe. Dans ce contexte, le contrôleur est une partie extrêmement complexe qui doit être subdivisée en plusieurs sous-parties pour arriver aux briques de bases d'une architecture générique d'éditeurs.

4.1.4 Synthèse

L'architecture MVC a été reprise, sous diverses formes, dans toutes les architectures d'éditeurs. Parmi les critères d'évaluation que nous pouvons appliquer à MVC, le *temps d'apprentissage* est court car le concept est simple, le *temps de construction* d'applications utilisant MVC est raisonnable compte tenu des bénéfices de l'organisation résultante. La *méthodologie de construction* est simple puisqu'elle implique une organisation uniforme à tous les objets qui doivent être éditables. Elle est cependant limitée à quelques objets particuliers et non à la structure d'ensemble de l'application. En suivant le modèle MVC, les applications sont *modulaires* et *extensibles* sans pénalité excessive pour le *code source*. L'architecture MVC est totalement indépendante du *modèle graphique* et des *dispositifs* utilisés.

4.2 MacApp

Le squelette d'application MacApp [Schmucker87] est le premier système qui ait décrit une architecture concrète d'application graphique sous forme de classes coopérantes, devant être spécialisées pour chaque application. Les classes qui nous intéressent pour les éditeurs sont : **Application**, **Document** et **View**.

MapApp définit beaucoup d'autres classes utilitaires, ainsi que la classe **Command** qui permet d'implanter les commandes *Undo* et *Redo* que le guide de construction d'interfaces du Macintosh recommande de fournir dans toutes les applications graphiques.

La classe **Application** gère les données globales à l'application. Chaque application peut manipuler un ou plusieurs documents, gérés par la classe **Document**, et chaque document est visualisé par au moins une fenêtre qui dérive de la classe *View*.

4.2.1 La classe *Application*

C'est la classe responsable de la gestion des données partagée par tous les documents de l'application. Il n'existe qu'une seule instance de cette classe par application.

Entre autre, cette classe est responsable de la composition de la barre de menu et connaît les types des documents à ouvrir. La spécialisation de cette classe se fait par dérivation et spécialisation de certaines méthodes. En particulier, la méthode responsable de la création d'un document doit toujours être redéfinie.

4.2.2 La classe *Document*

C'est l'objet qui gère une unité de sauvegarde et de chargement, généralement un fichier. Le document est responsable de la sauvegarde et de l'ouverture d'un fichier. Il crée les fenêtres et compose leur contenu.

Une application peut parfois gérer plusieurs documents, qui peuvent être homogènes ou de natures différentes (du texte et du graphique par exemple dans les applications intégrées), et un document peut gérer une ou plusieurs vues.

> Par exemple, la figure 1(d) (page 8) montre un document — une famille de polices de caractères stockée dans un fichier — présenté sur trois vues : la liste des polices, le contenu d'une police et la description d'un caractère.

Au moins deux méthodes doivent être redéfinies pour une nouvelle applications : *DoMakeWindows* et *DoMakeViews* (d'une manière générale, les fonctions de MacApp dont le nom commence par *DoMake* doivent être spécialisées pour chaque application).

4.2.3 La classe *View*

C'est l'objet qui gère la présentation graphique des documents ou des parties d'un document. Elle s'occupe de l'affichage ainsi que de la gestion de l'interaction sur les données affichées.

4.2.4 Éléments graphiques de la vue

Toutes les architectures logicielles utilisent la notion de vue dans un sens similaire à la *View* de MacApp. Avant de présenter les autres architectures, il est utile de distinguer les éléments graphiques qui constituent une vue, qui sont de plusieurs types : *décoration*, *présentation*, *sous-vue*, *contrôle indirect* et *contrôle*

direct. Bien que ces termes ne soient pas ceux de MacApp ni de Smalltalk, nous les introduisons ici afin d'homogénéiser la terminologie.

4.2.4.1 Objets de décoration

Ce sont des objets qui ne participent pas à l'interaction et qui n'ont pas d'autre intérêt qu'esthétique. Les « filets » ou les effets d'ombre sont des objets de décoration.

4.2.4.2 Objets de présentation

Ce sont des objets qui affichent des données et sur lesquels l'utilisateur ne peut pas agir comme le nom de la vue par exemple, mais qui peuvent être modifiés lors d'un changement d'état de l'application.

4.2.4.3 Sous-vue

Il est parfois utile de faire apparaître une vue comme un élément graphique à l'intérieur d'une autre vue. Par exemple, une vue présentant un point de vue sur une scène graphique apparaît généralement à l'intérieur d'une vue qui contient deux barres de défilement et parfois d'autres *Widget*s en plus de la sous-vue.

4.2.4.4 Objets de contrôle indirect

Comme nous l'avons vu dans la figure 1.1, les éditeurs utilisent des *Widget*s autour de la vue principale. Ces *Widget*s contrôlent indirectement (dans le sens donné en 1.1) les objets graphiques de l'édition, leur apparence ou leur état.

4.2.4.5 Objets de contrôle direct

Le contrôle direct permet de modifier les structures graphiques directement à l'aide d'un dispositif de pointage. La plupart des applications graphiques proposent plusieurs modes de manipulation graphique, sélectionnés dans une palette représentée par une boîte de contrôleurs indirects : la boîte d'outils (*tool box*, à ne pas confondre avec la boîte à outils *toolkit*).

4.2.4.6 La sélection

Certaines fonctions, déclenchées par manipulation directe ou indirecte, peuvent s'appliquer identiquement sur un seul ou sur une liste d'objets représentés

graphiquement. La spécification de cette liste peut se faire de façon préfixe ou affixe, c'est-à-dire que les éléments peuvent être désignés avant ou après le choix de la fonction.

Les avantages respectifs du mode de sélection préfixe et affixe sont discutés par [Lieberman85]. Lorsqu'une fonction peut être appliquée indépendamment du rôle et de l'ordre des éléments sélectionnés, alors le mode de sélection préfixe est généralement préféré pour des raisons d'ergonomie cognitive [Shneiderman92, Tesler81, Smith et al.82]. La liste des objets est alors appelée « la sélection ». Pour des interfaces où l'ordre des éléments a une importance, alors un dialogue séquentiel doit être engagé avec l'utilisateur pour qu'il spécifie les objets dans le bon ordre, ou avec les bonnes caractéristiques, ce qui empêche d'utiliser une sélection préfixe.

La plupart des éditeurs utilisant le mode de sélection préfixe, il est nécessaire de visualiser le fait qu'un objet graphique fait partie de la sélection. Deux modes de visualisation existent : par utilisation d'attributs graphiques particuliers ou par utilisation d'objets graphiques particuliers.

Ainsi, la plupart des éditeurs de texte utilisent un mode de sélection préfixe. Comme l'affichage d'un texte se fait généralement à l'aide de deux couleurs — une pour le fond et une pour le texte —, deux autres couleurs peuvent être choisies pour représenter la couleur de fond et de texte des portions sélectionnées.

En revanche, pour un éditeur graphique pixellaire comme PhotoShop, tous les attributs graphiques peuvent être utilisés par les images affichées. Pour représenter la sélection de façon non ambiguë graphiquement, PhotoShop représente les zones sélectionnées en les entourant d'une ligne pointillée animée. La sélection est donc représentée par des objets graphiques particuliers.

4.2.4.7 La boîte d'outils

Le fait de changer le mode d'interaction est semblable au changement d'instrument que nous utilisons pour agir sur des objets physiques. Par analogie, la boîte à boutons permettant de choisir le mode d'interaction s'appelle la boîte d'outils. Certains squelettes d'application définissent une classe « outil » qui implante le mode d'interaction.

4.2.5 Synthèse

MacApp [Schmucker87] a été le premier squelette d'application (*Application Framework*) et sert de base de développement à de nombreux produits industriels comme PhotoShop [Adobe Systems Incorporated93a]. Au-dessus de la boîte à outils du Macintosh, il offre une architecture préfabriquée dans laquelle certains blocs doivent être construits et certains autres modifiés ou étendus. Il simplifie une grande partie de la construction des éditeurs graphiques mais s'arrête juste avant la

gestion des objets graphiques ; à l'intérieur de la vue principale de l'application, la gestion du graphique et de l'interaction est laissée au programmeur d'application, qui doit utiliser la boîte à outils standard du Macintosh.

Temps d'apprentissage : MacApp est long à apprendre totalement. Sa complexité vient du fait que la construction d'une application complète se fait en modifiant des détails d'un squelette d'implantation globale, prenant en compte plusieurs catégories d'éditeurs, chacun pouvant nécessiter des spécialisations très particulières. Plus l'application s'éloigne du squelette initial et plus le temps d'apprentissage peut être long.

Temps de construction : Une fois MacApp maîtrisé jusqu'au niveau requis, la construction d'une application graphique est sensiblement plus rapide qu'avec une boîte à outils. La gestion du graphique de la vue principale doit néanmoins être réalisée à partir des primitives graphiques de la boîte à outils et en suivant des contraintes supplémentaires induites par MacApp (comme la gestion de la sélection ou des objets **Command**), ce qui complexifie un peu la tâche.

Méthodologie de construction : Le fait de s'appuyer sur un squelette donne une méthode de construction très claire.

Modèle du graphique : À l'intérieur d'une vue, le programmeur doit gérer son modèle graphique. MacApp ne lui donne aucune contrainte mais ne lui est d'aucune aide non plus.

Gestion de dispositifs : Même remarque que pour le modèle du graphique.

Modularité de l'architecture : Les diverses parties d'une application graphique sont bien différenciées et les relations entre ces parties sont clairement établies.

Extensibilité des applications : Les extensions reviennent à redéfinir des méthodes de MacApp, ce qui est relativement simple une fois l'organisation générale comprise. L'extension de la sémantique des vues est du ressort du programmeur d'application et reste simple tant que l'architecture générale n'est pas bouleversée.

Compacité du code source : Les applications bâties avec MacApp requièrent sensiblement moins de code que si elles étaient bâties directement avec la boîte à outils du Macintosh.

De plus, MacApp garantit le respect du guide de style du Macintosh et offre des mécanismes de gestion des erreurs et de débogage sans dégrader les performances d'exécution ni la taille des programmes compilés.

L'utilisation de MacApp permet de réduire la complexité de la partie stéréotypée de l'application (barre de menu, *Widget*s, gestion du réaffichage globale) mais la gestion des objets graphiques à l'intérieur de la vue principale est entièrement laissée au programmeur d'application.

4.3 NeXTSTEP Interface Builder

NeXTStep [Next Computer Inc.92] implante un modèle de construction d'applications graphiques interactives extrêmement sophistiqué. Les applications stéréotypées peuvent être construites pratiquement interactivement à l'aide de l'environnement de programmation graphique. Lorsque le contenu de la vue principale n'est pas stéréotypé, le programmeur doit utiliser Display PostScript [Adobe Systems Incorporated93b] pour gérer l'affichage des objets graphiques. La gestion du graphique à état, entre Interface Builder et Display PostScript, n'est paradoxalement pas prise en compte directement dans NeXTSTEP.

4.3.1 Architecture d'une application

Les objets principaux d'un éditeur sont **Application** et **View**. Le document n'a pas d'existence concrète dans Interface Builder. Les applications sont écrites dans le langage Objective-C [Cox et al.91], qui est un langage à objets dont la sémantique est proche de Smalltalk et qui permet de charger facilement du code dynamiquement pendant l'exécution. C'est en partie grâce à ces mécanismes que les interfaces peuvent être programmées par manipulation directe. La partie liée directement à notre problématique concerne la gestion de la manipulation directe et du graphique associé. L'édition graphique à proprement parler se déroule sur une classe dérivée de **View**, nommée **DrawingView**. L'organisation structurelle d'une **View** destinée à contenir un objet à manipuler directement est décrite en figure 4.2.

4.3.2 Gestion de la manipulation directe

Pour le graphique interactif, Display PostScript doit résoudre deux problèmes : la sémantique graphique de PostScript ne permet pas d'effacer des objets sélectivement et les primitives PostScript sont relativement lentes à interpréter. Comme X11, Display PostScript est un système client/serveur. L'idée de Display PostScript est que la communication entre le client et le serveur peut être optimisée par la définition d'un protocole spécifique à chaque application. Ce protocole est, en réalité,

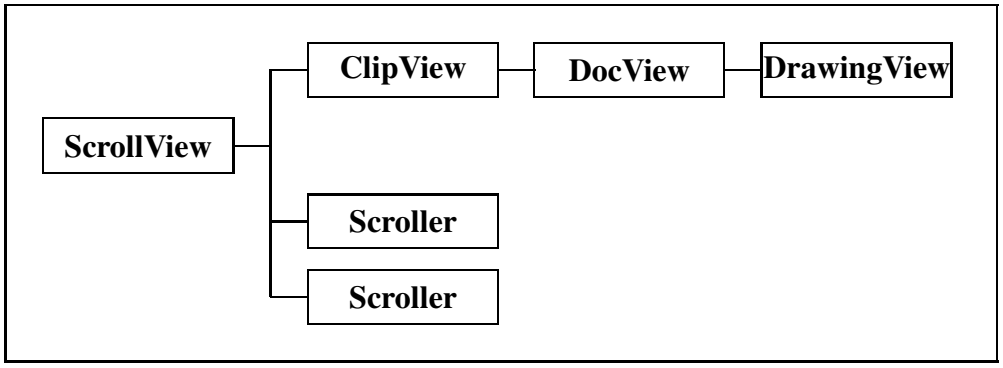


FIG. 4.2 - *Hiérarchie d'instances pour la manipulation directe dans Display PostScript.*

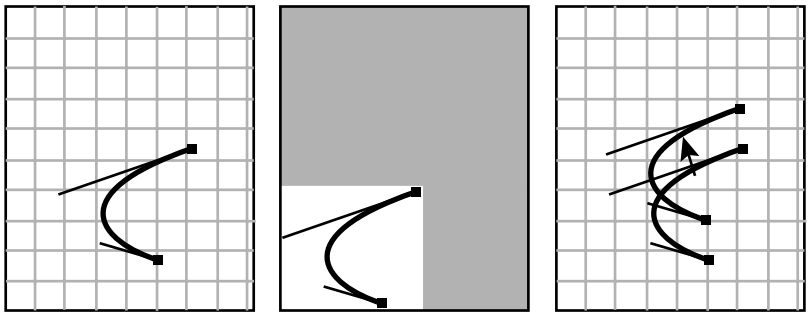


FIG. 4.3 - *Utilisation des Offscreen Buffers dans Display PostScript pour gérer la manipulation directe. L'image de gauche montre l'objet tel qu'il apparaît avant le début de la manipulation. L'image centrale montre le contenu du Offscreen Buffer utilisé pendant la manipulation. L'image de droite montre le résultat de la composition de l'image de gauche et de l'image centrale pendant la manipulation.*

un ensemble de commandes PostScript chargées dans le serveur lorsque l'application démarre et utilisées ensuite par les objets graphiques de l'application. Pour pallier aux déficiences de PostScript, Display PostScript augmente le langage de deux manières : il permet de cacher des objets PostScript dans le serveur et définit les *Offscreen Buffers*, afin de garder des dessins en mémoire plutôt que sur l'écran.

Le manuel Display PostScript explique la façon dont une application doit définir chaque catégorie d'objet pour obtenir les meilleures performances. En particulier, les images qui servent de décoration doivent être placées dans un *Offscreen Buffer*, ainsi que les objets de la manipulation directe. Les *Offscreen Buffers* peuvent être composés par Display PostScript, comme le montre la figure 4.3. Lors de sa création, on peut vouloir un buffer qui gère la transparence ou pas.

4.3.3 Synthèse

Dans la même lignée que MacApp, NextSTEP propose un squelette d'application. En plus, il décrit une méthodologie et des mécanismes de gestion du graphique 2D vectoriel. Les mécanismes ne sont pas implantés sous forme opérationnelle, obligeant le développeur d'application à les implanter lui-même.

Temps d'apprentissage : NextSTEP est moins long à apprendre que MacApp grâce à un environnement de programmation bien conçu qui permet d'accéder rapidement à la documentation.

Temps de construction : une application graphique interactive stéréotypée peut être conçue interactivement avec *Interface Builder*. Le développement d'éditeurs textuels ou graphiques vectoriels 2D est facilité par l'utilisation des primitives PostScript qui sont de haut niveau.

Méthodologie de construction : les manuels de NextSTEP et de Display PostScript ne décrivent pas de méthodologie de développement.

Modèle du graphique : le graphique vectoriel 2D est simple à gérer grâce à PostScript. Des extensions existent pour manipuler le graphique pixellaire mais le 3D n'est pas prévu. La gestion des objets graphiques de l'interaction est possible en utilisant les *Offscreen buffers*.

Gestion de dispositifs : rien n'est prévu en dehors du clavier et de la souris.

Modularité de l'architecture : même remarque que MacApp.

Extensibilité des applications : même remarque que MacApp.

Compacité du code source : même remarque que MacApp.

4.4 Garnet

Garnet [Myers91a] est à la fois une boîte à outils et un ensemble d'outils permettant de construire des applications graphiques interactives. Garnet est écrit en Common Lisp [Steele Jr.84] et se décompose en plusieurs niveaux : un solveur de contraintes (*kr*) [Myers91b, Zanden et al.91], le moteur graphique (*opal*), les gestionnaires de l'interaction (*inter*), une collection de widgets (appelés *gadgets*), un outil de construction d'interface (*gilt*) et un outil interactif de construction de gadgets (*lapidary*).

Garnet s'appuie énormément sur les possibilités du langage Common Lisp et améliore la notion de structure pour permettre aux champs de contenir non seulement des valeurs mais aussi des formules, c'est-à-dire des valeurs recalculées lors de l'accès au champ. Le langage dispose donc d'un mécanisme de propagation de dépendances qui permet une communication implicite entre les objets.

À partir de cette architecture, Garnet implante des objets graphiques qui, lorsqu'ils sont modifiés, déclenchent automatiquement leur réaffichage, ce qui rend la gestion du graphique très confortable.

Un des apports les plus importants de Garnet est l'*interacteur* [Myers89]. Un interacteur est un objet qui implante la gestion d'une modalité d'interaction dans Garnet. Il s'agit d'un raffinement du rôle du contrôleur de l'architecture MVC, bien que Garnet ne fasse pas référence à cette architecture.

4.4.1 Les Interacteurs

Un interacteur [Myers90] est un objet qui encapsule un comportement interactif. Le fait que ce soit un objet lui permet d'être manipulé et nommé. Garnet offre un programme de construction d'éditeurs qui permet de décrire interactivement que tel interacteur sera associé à tel objet à un moment donné. Un interacteur est déclenché par une condition initiale, puis il fonctionne en gérant des dispositifs d'entrée et en produisant un effet de retour ; finalement il appelle une fonction qui déclenche une action à partir des paramètres de la manipulation. Les interacteurs peuvent être paramétrés pour s'adapter à plusieurs variantes dans leur style d'interaction.

Garnet propose plusieurs types d'interacteurs (7 originellement, puis 8), qui sont :

Menu-Interactor: permet de choisir un ou plusieurs éléments d'un ensemble.

Button-Trill-Interactor: semblable au *Menu-Interactor* mais permet de répéter l'action avec une fréquence donnée tant que le bouton de la souris reste enfoncé ; utilisé par exemple pour les flêches d'une barre de défilement.

Select-and-Change-Interactor: utilisé pour déplacer ou changer la taille d'un objet graphique.

New-Point-Interactor: utilisé quand un ou plusieurs points doivent être saisis à partir de la souris.

Angle-Interactor: utilisé pour calculer l'angle que décrit le pointeur de la souris par rapport à un axe.

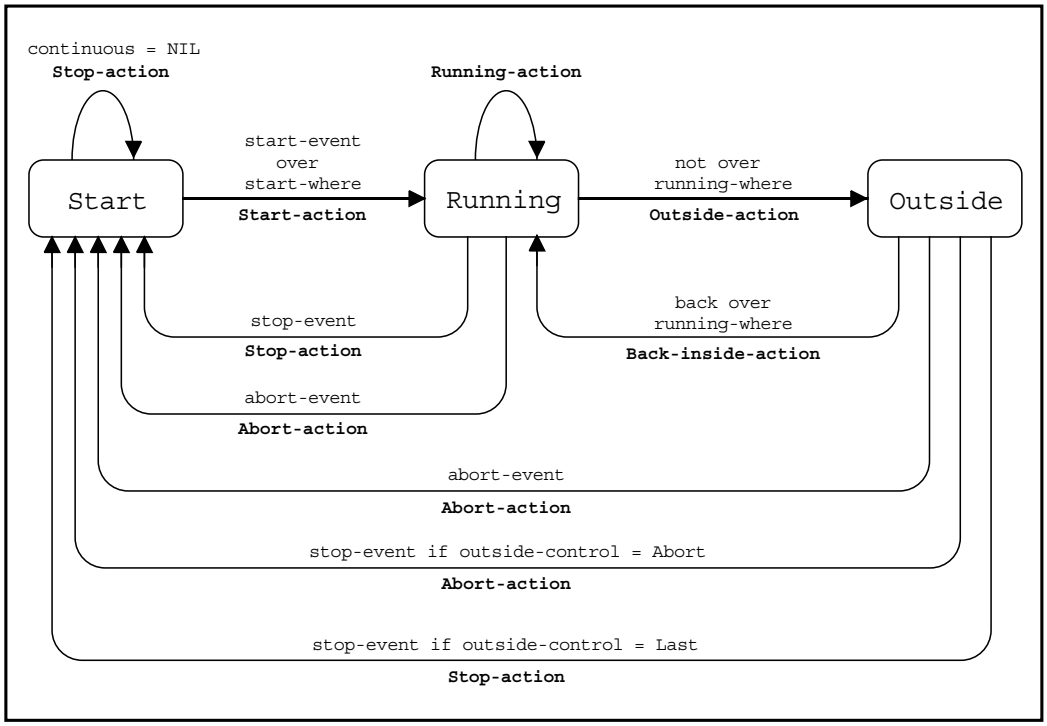


FIG. 4.4 - *Automate de gestion des événements dans les interacteurs de Garnet (Schéma tiré du manuel de référence de Garnet).*

Trace-Interactor: utilisé pour recueillir toutes les positions que parcourt la souris entre deux événements.

Text-string-Interactor: utilisé pour saisir et éditer une ligne de texte.

Gesture-Interactor: utilisé pour reconnaître un geste fait d'un seul trait.

Tous les interacteurs utilisent un automate décrit en figure 4.4. Les transitions sont provoquées par l'arrivée d'événements (marqués en penché) et peuvent déclencher l'appel d'une procédure de contrôle (marquée en gras). Les interacteurs offrent donc une assez grande flexibilité et implantent un comportement dans un objet. Cet objet peut ensuite être représenté graphiquement et manipulé dans un système interactif de construction d'interface, comme dans le cas de *Lapidary*.

Les interacteurs ne savent gérer que la souris et le clavier. Plusieurs interacteurs peuvent fonctionner en parallèle mais ils ne sont pas extensibles, seulement configurables. Il est donc impossible de rajouter de la gestion multimodale ou de nouveaux types de dispositifs d'entrée à partir des interacteurs existants.

4.4.2 Synthèse

Garnet est un environnement de construction d'applications graphiques interactives qui met en œuvre plusieurs techniques originales :

- un système de contraintes unidirectionnelles implanté dans les couches basses du système ;
- le modèle d'interacteur pour encapsuler la manipulation interactive d'objets graphiques ;
- un générateur d'éditeurs.

Contrairement à MacApp et NextSTEP, il n'est pas bâti autour de peu de concepts unificateurs mais plutôt autour de nombreux concepts *ad-hoc*. Malgré tout, il reste un des seuls environnements qui permet de construire une grande variété d'éditeurs sans avoir à écrire une seule ligne de code.

Temps d'apprentissage : pour construire des éditeurs schématiques, le temps d'apprentissage est court et l'environnement interactif de construction permet d'éviter de taper des lignes de code. Étendre les possibilités de Garnet requiert la maîtrise de nombreuses couches logicielles et de modules. Le temps d'apprentissage n'est donc pas continu suivant la complexité de l'éditeur à construire.

Temps de construction : à cause du problème de discontinuité du temps d'apprentissage, le temps de construction peut être extrêmement court — lorsque le domaine de l'éditeur et ses spécifications sont conformes à ce que Garnet propose — ou rédhibitoirement long sinon.

Méthodologie de construction : Garnet propose d'utiliser ses outils interactifs pour bâtir un éditeur. Lorsque ce n'est pas suffisant, il est possible de rajouter des modules de code Lisp. Aucune méthodologie n'est proposée pour étendre les capacités de Garnet.

Modèle du graphique : Garnet ne connaît qu'un modèle graphique vectoriel schématique.

Gestion de dispositifs : Garnet ne connaît qu'un pointeur et un clavier.

Modularité de l'architecture : Garnet est complexe mais relativement modulaire. Cependant, les langages interprétés à typage dynamique offrent des mécanismes confortables mais peu sûrs dont les programmeurs abusent souvent. Il est difficile de prédire les conséquences profondes de modifications a priori anodines sur le modèle graphique ou les dispositifs d'entrée.

Extensibilité des applications : lorsqu'un éditeur a été bâti entièrement à l'aide des outils interactifs de Garnet, il peut être aisément étendu, interactivement ou en lui rajoutant du code Lisp. Lorsque du code Lisp a été rajouté à un éditeur bâti interactivement, une partie de sa sémantique n'est pas prise en compte par les outils interactifs et le résultat n'est pas garanti de fonctionner. Étendre une application en modifiant le modèle graphique ou la structure des objets graphiques est complexe.

Compacité du code source : pour bâtir des éditeurs appartenant aux classes prévues, la taille du code est réduite. Pour le développement de nouveaux modules, le code est généralement compact grâce aux extensions syntaxiques rajoutées au langage Lisp pour faciliter l'accès aux structures de données spécifiques de Garnet. De plus, Lisp, comme Smalltalk, est un langage de haut niveau qui permet d'ignorer des problèmes comme la gestion de la mémoire, ce qui réduit encore la taille du code à implanter.

4.5 Unidraw

Unidraw [Vlissides et al.89] est un squelette d'application pour développer des éditeurs spécifiques (*A Framework for Building Domain-Specific Graphical Editors*). Par rapport à MapApp, Unidraw décrit beaucoup plus précisément les opérations qui se déroulent pour l'affichage de structures graphiques et leur manipulation directe.

La présentation des éditeurs d'Unidraw est généralement conforme à la figure 4.5. Le *Panner* gère le point de vue du **Viewer** de façon analogue à deux barres de défilement. Par rapport aux autres architectures d'éditeurs, Unidraw précise donc une architecture du **Viewer**.

La figure 4.6 décrit l'organisation des objets principaux d'une application bâtie avec Unidraw. Comme toujours, la terminologie utilisée diffère entre tous les squelettes d'applications. Il est cependant facile de reconnaître que la classe **Unidraw** est en faite la classe **Application** de MacApp et que la classe **Editor** correspond à la classe **View** (pas **Document**). Le document en tant que tel n'est pas implanté dans une classe mais existe dans un **Catalog**.

4.5.1 Modèle architectural d'un éditeur

Un éditeur représente des données structurées sous forme d'arbre, contenant des composants (*Components*). Ces composants peuvent être visualisés et manipulés dans un **Viewer**, qui édite une partie de l'arborescence, à partir d'un composant racine. Plusieurs vues peuvent coexister sur un même arbre de composants,

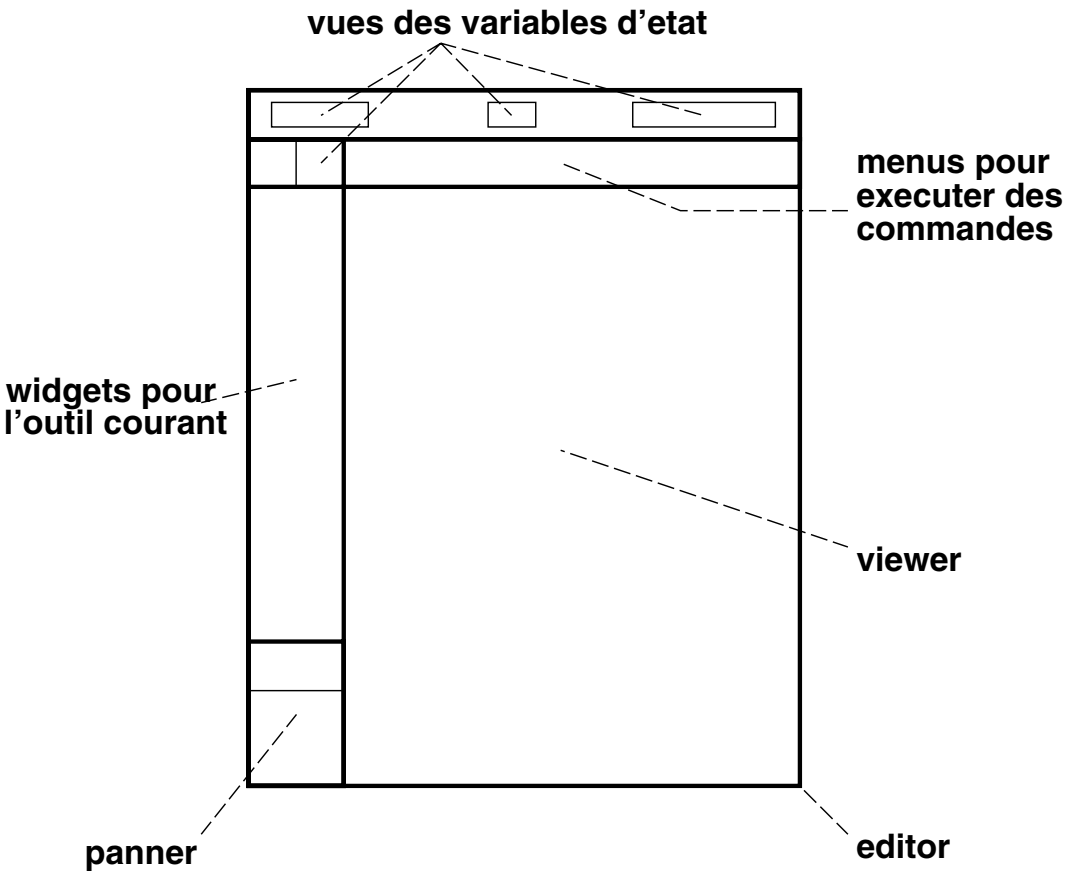


FIG. 4.5 - *Présentation générale d'un éditeur construit avec Unidraw*

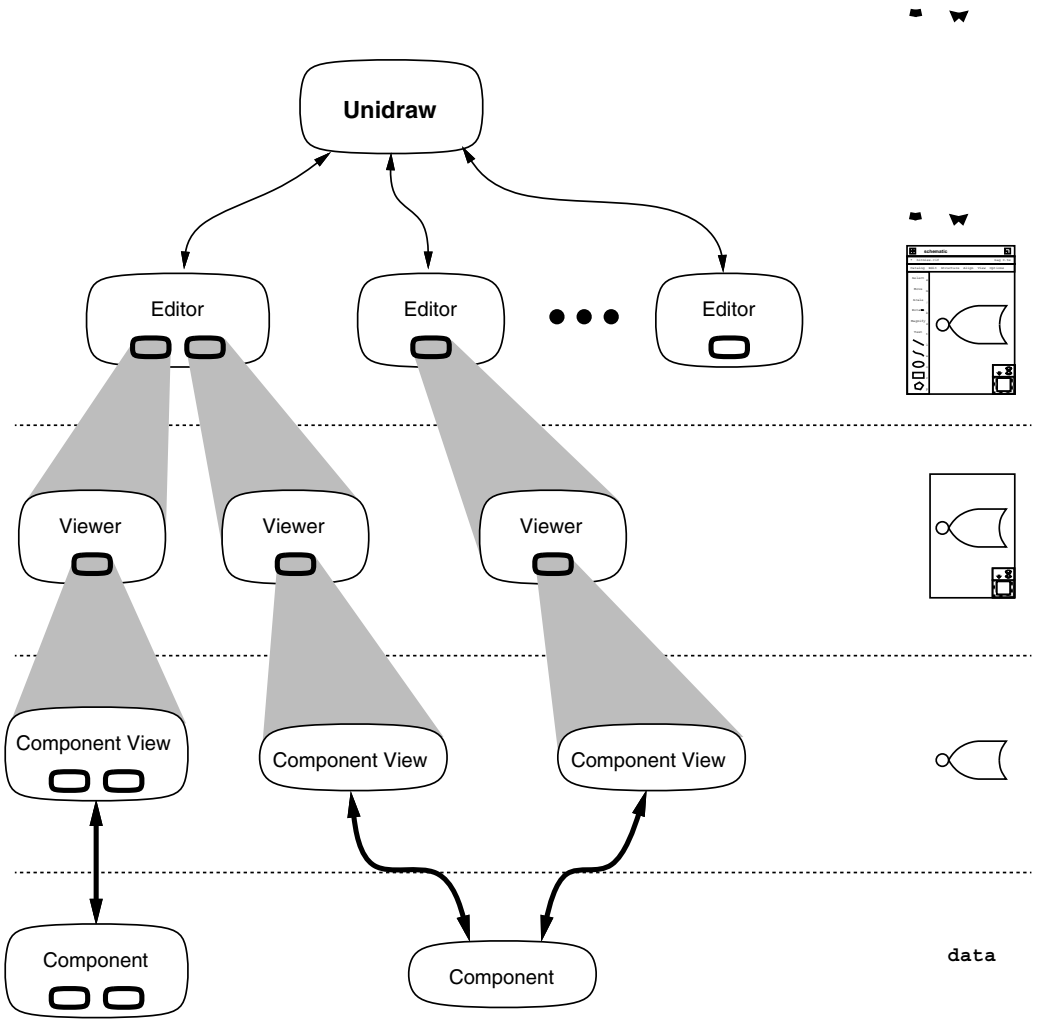


FIG. 4.6 - *Structure générale d'un éditeur fait avec Unidraw*

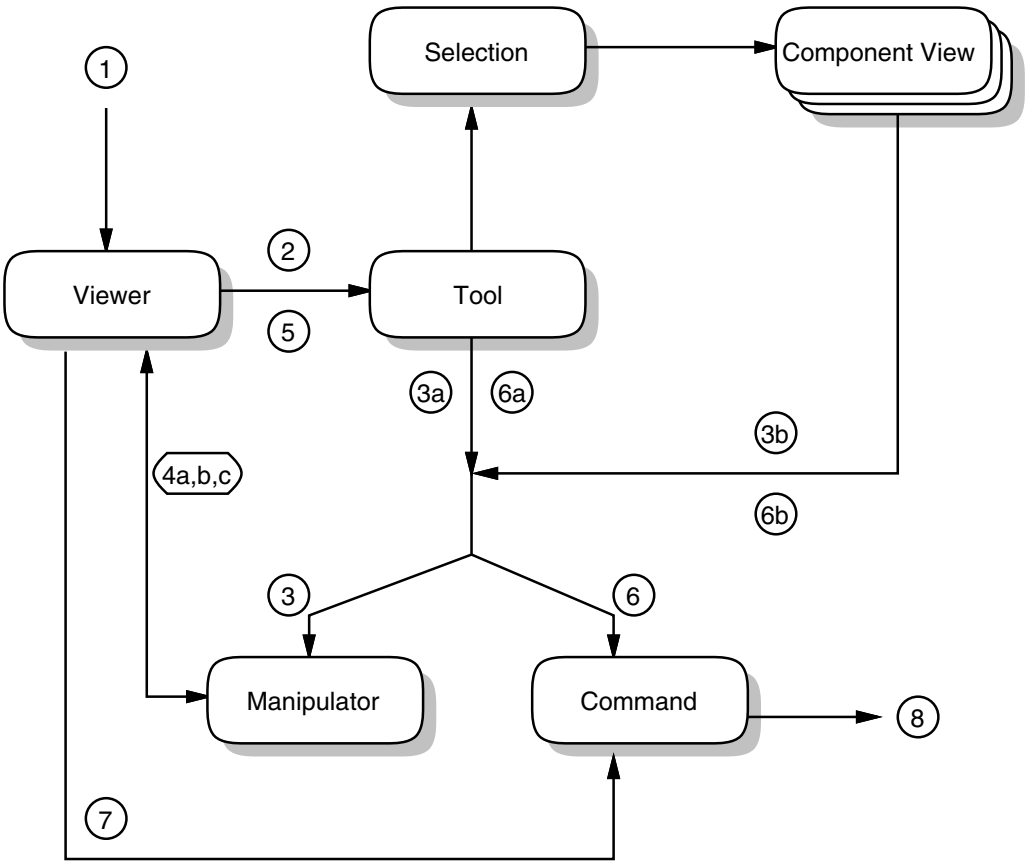


FIG. 4.7 - *La communication entre les objets pendant la manipulation directe*

partant de plusieurs racines différentes si l'application en a besoin. La face visible d'un composant est le **ComponentView**, qui s'occupe de gérer la sémantique graphique du composant.

Pour fabriquer un éditeur spécifique, Unidraw requiert que le programmeur définisse ses propres classes de composants, puis les **ComponentView** associés. En général, ces définitions se font par dérivation et nécessitent très peu de code. Pour la manipulation directe, Unidraw délègue les opérations à des objets de type « outils » (**Tool**) qui, en fonction des événements qu'ils reçoivent, font des actions directement ou indirectement à travers des objets de type **Command**. La section suivante est une traduction de la partie de la thèse de John Vlissides qui décrit la communication entre les objets.

4.5.2 Communication entre les objets pendant la manipulation directe

La figure 4.7 décrit la communication entre les objets pendant la manipulation directe et clarifie les rôles des outils (**Tool**), manipulateurs (**Manipulator**), commandes (**Command**), visualiseurs (**Viewer**) et les vues graphiques (**Graphical Views**). Les labels numériques dans le diagramme correspondent au passage en séquence :

1. le visualiseur reçoit un événement, comme l'appui sur un bouton de souris ;
2. le visualiseur demande à l'outil courant de créer un manipulateur en fonction du type de l'événement ;
3. *Création du manipulateur :* l'outil peut :
   (a) créer un manipulateur lui-même (en fonction de la sélection ou d'autres informations), ou
   (b) demander au(x) *component view(s)* de créer le(s) manipulateur(s) pour lui. L'outil doit alors combiner les multiples manipulateurs en un manipulateur composite. Chaque classe de *component view* est responsable de créer un manipulateur approprié à l'outil.
4. *Manipulation directe :* le visualiseur :
   (a) invoque la fonction *Grasp* (attrape) du manipulateur en lui passant l'événement déclenchant ;
   (b) boucle en lisant les événements suivants et en les envoyant au manipulateur avec la fonction *Manipulating* (la boucle est terminée lorsque la fonction retourne la valeur faux) ;
   (c) invoque la fonction *Effect* du manipulateur en lui passant l'événement qui a terminé la boucle.
5. Le visualiseur demande à l'outil courant d'interpréter le manipulateur (*InterpretManipulator*).
6. Interprétation du manipulateur : l'outil peut :
   (a) interpréter le manipulateur directement, créant la commande appropriée, ou
   (b) demander au(x) *component view(s)* d'interpréter le(s) manipulateur(s) pour lui. La ou les vues créent alors les commandes appropriées. L'outil doit alors combiner les commandes en une commande composite (macro).

7. Le visualiseur exécute la commande.

8. La commande mène à bien l'intention de la manipulation directe.

Cette description est compliquée, ce qui explique le temps d'apprentissage d'Unidraw. Le schéma de gestion de la manipulation directe est figé. Pour le spécialiser, le seul mécanisme disponible consiste à redéfinir le comportement des étapes parcourues. Plus le nombre d'étapes est élevé et plus le schéma peut être spécialisé.

4.5.3 Limites d'Unidraw

Unidraw souffre de trois limites importantes :

– la sémantique graphique des *component views* est celle des surfaces virtuelles d'Unidraw. Elle suffit pour tout ce qui est de la schématique mais ne convient pas à toutes les applications qui nous intéressent.

– la forme que prend la sélection est impossible à modifier.

– la manipulation directe s'appuie sur deux objets : l'outil et le manipulateur. Cette dualité est justifiée comme suit par Vlissides :

> Au début de la conception de l'architecture, il n'y avait pas de distinction entre outil en manipulateur : les outils définissaient leur propre sémantique de manipulation. Il nous est alors apparu que plusieurs outils partageaient la même sémantique, quelques uns avaient une sémantique proche, certains avaient des sémantiques différentes à certains moments, et d'autres encore avaient une sémantique unique et potentiellement idiomatique. En liant la sémantique à chaque sous-classe particulière d'outil, il était impossible de factoriser les parties communes des sémantiques pour qu'elles soit utilisables par d'autres classes d'outils.

En pratique, le fait qu'il existe deux objets rend la construction de nouveaux outils/manipulateurs très pénible. La manipulation étant modale, la gestion de l'interaction pendant la manipulation directe est très particulière, spécialisée pour un dispositif comme une souris qui ne permet pas de faire autre chose pendant la phase du clique-tire-relâche. L'extension de la gestion d'événement au multi-modal est pratiquement impossible. Les manipulateurs ont un comportement stéréotypé et ne peuvent pas être raffinés pour gérer une forme particulière de retour lexical ou syntaxique durant la manipulation directe.

4.5.4 Synthèse

Unidraw est le premier squelette d'éditeur à offrir un mécanisme pour gérer le graphique structuré et non stéréotypé des éditeurs. Pour cela, il définit une organisation générique d'objets coopérants. Les objets graphiques sont organisés en arbre et se dessinent suivant une sémantique 2D vectorielle schématique. Bien que le rôle de chacun des objets coopérant soit clair, les mécanismes implantant la manipulation directe restent peu extensibles.

Temps d'apprentissage : comme Garnet, Unidraw permet de construire certains éditeurs sans programmer. La construction d'éditeurs en programmant est rapide à comprendre par copie d'exemples. En revanche, les multiples mécanismes mis en œuvre par Unidraw sont très difficiles à comprendre. Comme MacApp, Unidraw définit un squelette générique et des mécanismes pour configurer ou modifier ce squelette, ce qui requiert pratiquement une compréhension globale du système, c'est-à-dire plus d'une centaine de classes.

Temps de construction : pour des éditeurs ne requérant aucune extension à Unidraw, la construction est très rapide, grâce au générateur d'éditeurs *Ibuild*. Pour des extensions prévues dans le schéma d'Unidraw, les mécanismes à mettre en œuvre sont simples et le code n'est pas très compliqué, bien que nécessitant la création de deux types d'objets graphiques : **Component** et **ComponentView**. Pour des extensions structurelles (ajout de dispositifs ou changement de modèle graphique par exemple), des modifications profondes, compliquées et donc longues doivent être portées au noyau d'Unidraw.

Méthodologie de construction : le fait de décomposer finement une application en objets concrets permet de systématiser le processus de construction d'un éditeur. Lorsque les mécanismes de communication conviennent à l'éditeur, cette méthodologie est rigoureuse. Cependant, la modification des mécanismes internes n'est pas toujours prévue et ne peut se faire de façon sûre.

Modèle du graphique : le graphique d'Unidraw est uniquement vectoriel schématique et les objets graphiques sont organisés sous forme d'arbre. Ces arbres peuvent être visualisés par plusieurs points de vues et chaque nœud peut servir de racine pour l'édition. Cette propriété est originale et très utile pour l'édition schématique où les éléments se décomposent souvent en blocs logiques eux-mêmes décomposables.

Gestion de dispositifs : Unidraw ne connaît qu'un pointeur et un clavier.

Modularité de l'architecture : Unidraw est modulaire pour la construction d'éditeurs mais la définition de nouveaux objets requiert une chaîne de modifications non localisées. Par exemple, chaque objet graphique et chaque outil doit pouvoir se sauvegarder et se recharger à partir d'un canal de communication abstrait (*stream*). L'objet responsable de cette sauvegarde et du chargement est le **Catalog** d'Unidraw, qui doit être dérivé pour chaque série d'objets nouveaux à une application. Chacun de ces objets doit avoir un type qui est un entier unique. Tout cela implique la définition de constantes dans un seul fichier, leur utilisation pour la dérivation du **Catalog** et pour chacune des définitions de ces objets nouveaux. L'ajout d'un type déclenche la recompilation de pratiquement toute l'application et aucune méthode ne permet de s'assurer que chaque type est bien géré par le **Catalog**…Du point de vue génie logiciel, Unidraw n'est pas assez modulaire.

Extensibilité des applications : Les applications bâties sur Unidraw sont extensibles, avec la lourdeur décrite précédemment. Comme les autres squelettes d'application, certaines extensions sont très simples tandis que d'autres remettent en cause la structure même d'Unidraw et en font perdre ses bénéfices.

Compacité du code source : Les éditeurs implantés avec Unidraw ne requièrent pas beaucoup de code source, mais certaines parties de ce code sont éclatées entre plusieurs fichiers sources et parfois fastidieuses à écrire et pratiquement automatisable.

4.6 Analyse critique

Ce chapitre a présenté une rétrospective pratiquement historique des architectures logicielles conçues pour construire des applications graphiques interactives à manipulation directe. Partant de mécanismes généraux comme MVC, les architectures ont évolué pour permettre de construire certaines classes d'applications graphiques sans même écrire une ligne de code. MacApp a montré la voie en proposant un squelette d'application disposant de mécanismes de modifications du comportement standard. NextSTEP a poussé plus loin la régularité de l'architecture et a été jusqu'à offrir une application interactive pour bâtir une classe importante d'applications graphiques. Garnet a encore poussé plus loin la construction interactive d'applications graphiques en s'appuyant sur des mécanismes de communication ajoutés au langage de programmation. Garnet a encapsulé la gestion de la manipulation directe dans un objet : l'interacteur. Unidraw a finalement étendu à certains éditeurs graphiques la classe des applications qu'il était possible de construire pratiquement interactivement.

Cependant, les squelettes d'applications souffrent de plusieurs lacunes :

- ils ne proposent qu'un modèle graphique unique et de type vectoriel schématique ;
- seul Unidraw permet de gérer du graphique structuré complexe, mais il limite la structure graphique à un arbre 2D ;
- la forme de la sélection est prédéfinie et non modifiable ;
- le schéma de gestion des événements n'est pas extensible à des éditeurs utilisant des dispositifs d'entrée particuliers (tablette, boîte à boutons, etc.), ou des modalités particulières (trace ou reconnaissance de gestes), encore moins à l'interaction multimodale.

Les avancées dans le domaine se sont donc faites avec des assertions implicites sur le modèle graphique (2D vectoriel schématique), les dispositifs d'entrée (pointeur, clavier) et la façon dont l'écho interactif durant la manipulation directe devait être faite (boîte englobante ou objet graphique modifié directement). Toutes ces assertions limitent énormément le domaine des éditeurs, leur performance, ainsi que les types d'interactions qu'on peut en attendre.

Chapitre 5

Les modèles architecturaux

Comment les modèles architecturaux peuvent-ils aider à la description des éditeurs ? On peut distinguer deux types de modèles architecturaux, que nous appellerons modèles généraux et modèles génériques. Les modèles généraux tentent de décrire les applications graphiques en les organisant en entités abstraites communicantes et en attribuant une responsabilité claire à chaque entité. Les modèles génériques décrivent les applications graphiques interactives sous forme de collections d'objets formant des motifs qui se répètent et qui peuvent décrire globalement l'application, par raffinements successifs.

Parmi les modèles généraux, les plus connus sont celui de Seeheim [Pfaff85] et de l'Arche [User Interface Developer's Workshop91]. Parmi les modèles génériques, le plus connu est certainement PAC [Coutaz87]. Le livre « *Design Patterns* » (motifs de conception) [Gamma et al.94] décrit des stéréotypes d'implantation [1] ; ce sont des méta-modèles de réalisation. Contrairement aux modèles généraux et génériques, les motifs de ne s'appliquent pas à toutes les applications graphiques interactives et interviennent rarement pendant la phase de conception, mais apportent des solutions génériques à l'implantation d'une application modélisée.

Le modèle le plus simple est certainement celui proposé par [Foley et al.90] pour les applications graphiques (voir figure 5.1). Aucune référence n'est faite à la sémantique de l'application. Le modèle gère une structure graphique hiérarchique, sa création, sa modification et son affichage. Il s'agit uniquement d'un modèle de visualisation de structure graphique qui ne prend en compte ni une sémantique possible du graphique, ni la manipulation directe.

De notre point de vue, ce modèle est révélateur de la distance entre le domaine du graphique et de l'interaction homme-machine. En infographie, seule l'image fi-

1. Le mot englais *design* a un sens intermédiaire entre conception et réalisation, nous utilisons la traduction usuelle du nom du livre, qui est *Motifs de conception*, mais qui devrait être *Stéréotypes de construction de programmes*.

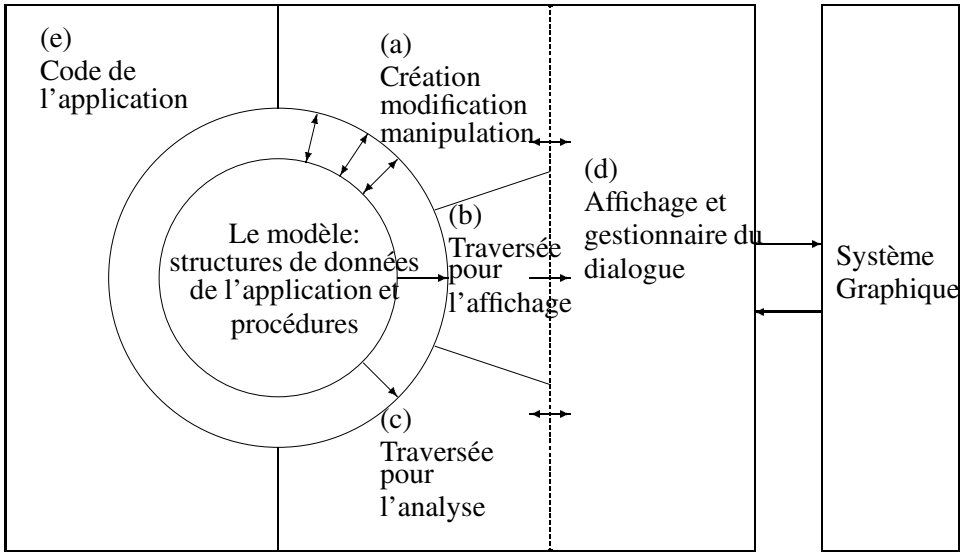


FIG. 5.1 - *Le modèle d'application avec ses modules de consultation et de modification selon [Foley et al.90]*

nale est considérée comme importante. Elle n'est généralement liée à aucun objet sémantique. L'interface se limite donc à l'ensemble des fonctionnalités destinées à manipuler le formalisme utilisé pour décrire les composantes des images. Dans le modèle de [Foley et al.90], les images sont décrites par des objets graphiques placés dans une base de données dont l'édition se limite à ajouter, modifier et retirer des objets de cette base. Bien entendu, les éditeurs généraux sont plus complexes car l'objet graphique qu'ils visualisent est une représentation d'un objet sémantique potentiellement complexe. Cette dimension n'est pas prise en compte dans le modèle de [Foley et al.90].

5.1 Modèle de Seeheim

Le premier modèle architectural d'application graphique interactive a été proposé lors d'un *workshop* à Seeheim [Pfaff85]. Il distingue trois composantes logiques : la présentation, le contrôle du dialogue et l'interface avec l'application, comme l'illustre la figure 5.2.

C'est un modèle qui s'appuie sur une analogie linguistique pour distinguer trois composantes dans les interfaces : lexicale, syntaxique, et sémantique.

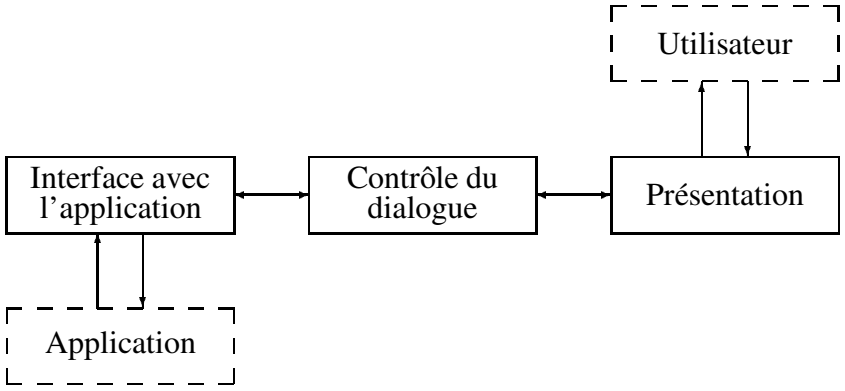


FIG. 5.2 - *Le modèle de Seeheim*

5.1.1 Présentation

La présentation gère la façon dont le système sera vu et manipulé physiquement. Dans le sens de l'entrée, elle traduit les événements envoyés par les dispositifs en entités lexicales, puis traduit les séquences d'entités lexicales en entités syntaxiques qui modifient l'état du système. Dans le sens de la sortie, elle reçoit des expressions du contrôleur du dialogue — ce qui constitue les entités syntaxiques de sortie — et les subdivise en entités lexicales qui sont les primitives graphiques.

5.1.2 Contrôleur du dialogue

Le contrôleur du dialogue sert d'interface entre la présentation et l'interface de l'application. Il ne peut pas être court-circuité. Il est responsable de l'analyse syntaxique et sémantique, c'est-à-dire :

en entrée : de vérifier que les entités syntaxiques qui lui arrivent sont correctes et d'appliquer l'opération sémantique associée à l'interface avec l'application, et

en sortie : de transformer des requêtes issues de l'interface avec l'application en expressions syntaxiques correctes qui seront envoyées à la présentation.

En plus de son rôle d'interface, le contrôleur du dialogue gère l'état de l'interaction. Toutes les expressions venues de la présentation ne sont pas valides dans tous les états. Ce rôle de gestionnaire d'état participe à la vérification sémantique et permet de filtrer des erreurs sémantiques avant qu'elles ne soient interceptées par l'application.

5.1.3 L'interface avec l'application

Plutôt qu'application, nous préférons parler de noyau sémantique. L'interface avec le noyau sémantique adapte le noyau sémantique à l'usage qui en sera fait par le contrôleur du dialogue. Il permet de traduire les données et les opérations du noyau sémantique en données et opérations acceptables par le contrôleur du dialogue.

5.1.4 Utilisation du modèle de Seeheim

Le modèle de Seeheim est avant tout fondateur car il tente d'imposer l'idée de séparation entre le noyau sémantique et sa présentation, avec la médiation du contrôleur.

Maintenir une séparation claire entre noyau sémantique et présentation est généralement difficile, surtout dans le cas des éditeurs. Même lorsque le noyau sémantique existe indépendamment de sa ou ses présentations, le fait de vouloir le présenter graphiquement requiert des mécanismes supplémentaires, par exemple pour assurer l'intégrité entre la présentation et le noyau.

> Par exemple, un système de fichiers d'ordinateur semble être un noyau sémantique clairement défini. Plusieurs sémantiques ont été définies avant que les applications graphiques interactives n'existent. Cependant, toutes les tentatives de construction de système de visualisation de système de fichiers sans modification de la sémantique du système de fichiers se sont avérées déficientes. Seules les applications fonctionnant sur Macintosh assurent l'intégrité de la représentation du système de fichiers. Cela est dû au fait que le Macintosh a prévu des mécanismes de notification destinés à prévenir le contrôleur lors d'une modification sur le système de fichier. Sans ce mécanisme, la cohérence entre le noyau sémantique et ses représentations graphiques ne peut pas être assurée interactivement.

5.2 Modèle de l'Arche

Le modèle de l'Arche [User Interface Developer's Workshop91] raffine le modèle de Seeheim en rajoutant deux composantes : l'adaptateur de domaine et l'interaction (voir figure 5.3).

Comme dans le modèle de Seeheim, la composante *interface avec l'application*, traduit les données et opérations du noyau sémantiques en données et opérations utilisables pour la visualisation et le contrôle de l'interation. La composante

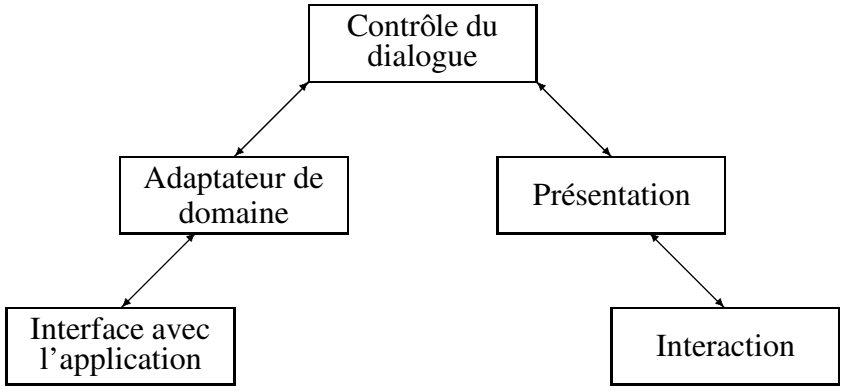


FIG. 5.3 - *Les modèle ARCH.*

adaptateur de domaine permet de gérer des objets qui sont utiles pour l'interaction mais qui n'ont pas de représentation dans le noyau sémantique de l'application, par exemple la gestion des erreurs sémantiques. Le contrôleur du dialogue fait l'interface entre les objets du domaine sémantique et le domaine graphique interactif. Il interagit avec la composante *présentation* qui reçoit de la composante *dialogue* des objets pouvant s'implanter sous plusieurs formes. Par exemple, un objet « choix multiple » peut être représenté par un menu ou par une palette de boutons. Le choix est fait dans la composante *présentation*. Enfin, la composante *interaction* est responsable de la gestion du graphique interactif au niveau bas et correspond généralement au niveau de la boîte à outils.

Pratiquement, la composante présentation gère des objets au niveau du *Widget* tandis que la composante interaction décrit l'implantation du *Widget*. Une caractéristique du modèle de l'Arche est que le rôle des composantes peut se déplacer suivant les applications. Du point de vue graphique, un même objet peut passer de la composante interaction à la composante présentation si son comportement a besoin d'être spécialisé et d'être sous le contrôle du dialogue.

> Pour saisir un nombre entier, un *Widget* de saisie de texte peut être utilisé. Si le *Widget* peut être paramétré pour n'accepter que des caractères formant un nombre valide, alors l'interaction est entièrement responsable de la saisie, le contrôleur du dialogue reçoit toujours un nombre valide de la présentation. Si le *Widget* ne peut pas être configuré, alors une partie de la gestion de l'interaction remontera à la présentation et le contrôleur du dialogue devra être appelé sur chaque caractère tapé pour vérifier sa validité.

La caractéristique du modèle de l'Arche qui lui permet de changer les attributions de la partie gauche à la partie droite suivant les besoins est à l'origine de son

qualificatif de modèle *slinky*, donné d'après le nom du jeu pour enfants.

5.2.1 Utilisation du modèle de l'Arche

Le modèle de l'Arche permet de modéliser des objets indépendants d'un domaine particulier, comme les *Widget*s pour la présentation, ainsi que des couches génériques pour l'adaptateur de domaine. Cette séparation permet de minimiser les modifications à apporter à une modélisation d'application graphique interactive lorsqu'on peut prévoir les parties qui vont être modifiées et celles qui ne le seront pas. Les parties immuables sont mises dans une des nouvelles composantes tandis que les autres sont modélisées explicitement et leur description peut être modifiée.

Pour la conception d'éditeurs, la partie importante à modéliser se trouve entre le contrôle du dialogue et la composante d'interaction. Unidraw et Garnet tentent de stéréotyper l'interaction, à l'aide des interacteurs, pour qu'une grande partie se fasse indépendamment du contrôle du dialogue donc n'ait pas à être programmée explicitement dans chaque éditeur. Cependant, il nous semble aussi important que l'interaction puisse facilement avoir accès au contrôle du dialogue pour améliorer le retour d'information pendant les manipulations. La structure fermée des interacteurs ne le permet pas.

Une des faiblesses du modèle de l'Arche est qu'il n'identifie pas finement les rôles de chacune de ses composantes. Il est donc plus descriptif qu'opérationnel.

5.3 Modèle PAC

Le modèle PAC de [Coutaz87] permet de décomposer une application graphique interactive en plusieurs agents constitués de trois facettes : la *Présentation*, l'*Abstraction* et le *Contrôle* (voir figure 5.3).

Contrairement au modèle de Seeheim, le modèle PAC n'utilise pas de métaphore linguistique. Toute application interactive est décomposée en agents PAC récursifs qui jouent un rôle aussi précis que l'on veut et collaborent à l'exécution correcte du programme. Chaque agent a trois facettes :

la présentation définit le comportement de l'agent vis-à-vis de l'utilisateur, en entrée comme en sortie ;

l'abstraction désigne les concepts et les fonctions de l'agent ;

le contrôle maintient la cohérence entre la présentation et l'abstraction.

PAC est un modèle abstrait. La décomposition d'une application en agents PAC n'est pas nécessairement identique à l'architecture logicielle que l'on choisira pour

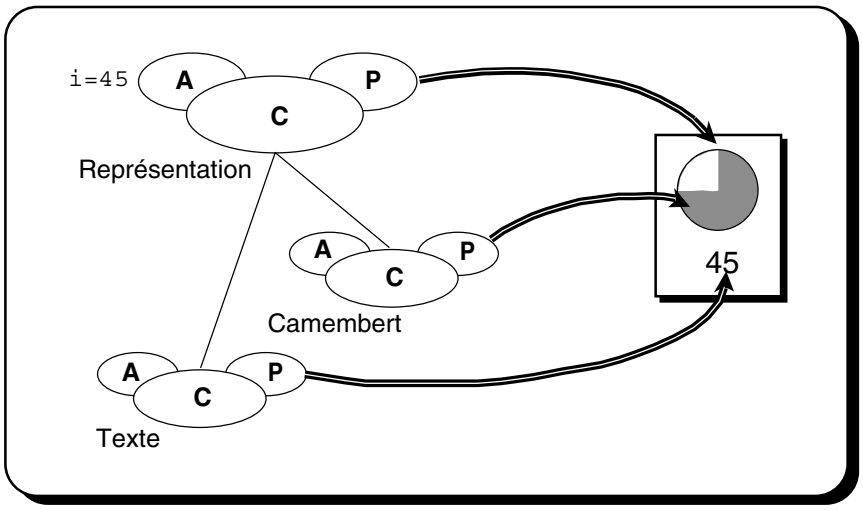


FIG. 5.4 - *Utilisation du modèle PAC (Schéma de S. Chatty)*

l'implanter. Si l'on choisit d'utiliser le modèle pour l'architecture d'une application graphique interactive, alors PAC peut être qualifié de méta-modèle d'architecture car il ne spécifie pas chaque élément constituant l'architecture mais implique que chaque élément constituant peut s'inscrire dans un schéma PAC.

5.3.1 Utilisation du modèle PAC

Pour être modélisée à l'aide de PAC, une application graphique doit être décomposée en agents. Cette décomposition peut se faire avec une granularité aussi petite que l'on veut et a pour but de faire apparaître les entités logiques collaborant à la ou aux tâches de l'application. La modélisation PAC prend énormément d'intérêt lorsqu'un répertoire d'agents est constitué et que les applications sont décrites en fonction de ces agents : c'est l'un des apports de PAC-AMODEUS.

5.4 Modèle PAC-AMODEUS

PAC-AMODEUS [Nigay94] s'appuie sur le modèle de l'Arche et décompose le contrôle du dialogue en agents PAC (voir figure 5.5). Il raffine le modèle PAC en :

– identifiant plusieurs agents et spécifiant précisément les informations qu'ils reçoivent et qu'ils produisent ;

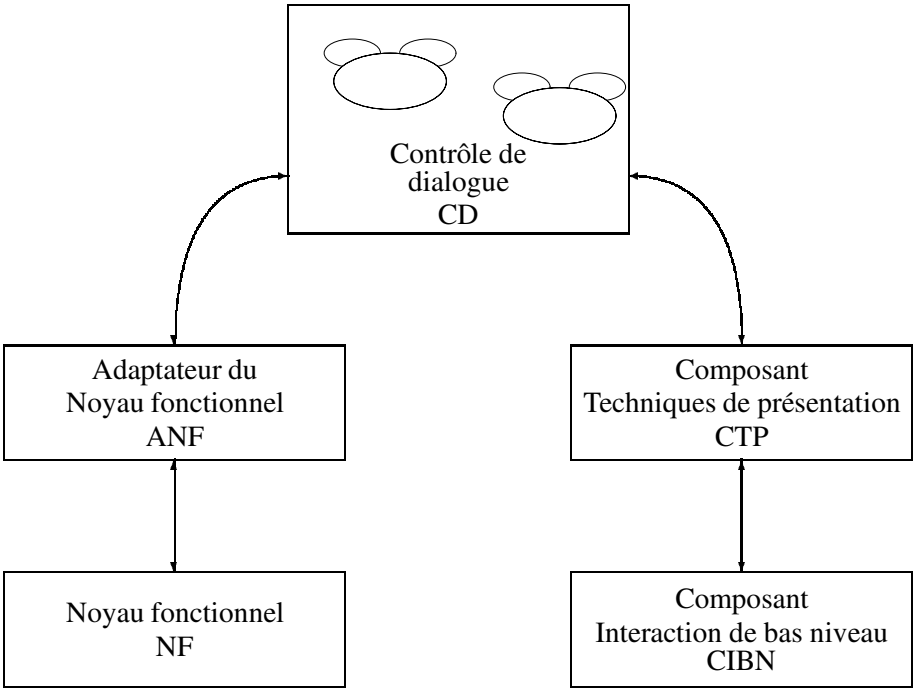


FIG. 5.5 - *Le modèle PAC-AMODEUS raffine le contrôleur du dialogue du modèle de l'Arche.*

– donnant des règles heuristiques sur la décomposition d'une application en agents ainsi que leur rôle.

PAC-AMODEUS permet de modéliser des applications multimodales en organisant des agents PAC de sorte que des informations issues de plusieurs canaux puissent être fusionnées. En ce qui concerne la conception d'éditeurs, des règles heuristiques sont décrites pour déterminer quand un agent PAC est utile et quel est son rôle. Nous allons donner un résumé de ces règles car elles sont souvent similaires au rôle des objets répertoriés dans les squelettes d'applications.

5.4.1 Règles heuristiques de mise en œuvre

– *Une fenêtre qui sert de support à un espace de travail est modélisée par un agent :* la classe « Document » dans MacApp et *Editor* d'Unidraw représente cet agent.

– *Les vues multiples d'un même concept sont gérées par un agent « vue multiple » :* cette règle s'applique lorsque la composante de présentation ne gère

pas les vues multiples. Elle est similaire à la règle d'utilisation systématique d'une classe Modèle dans le triplet MVC.

– *Une palette de classes de concepts est modélisée par un agent :* la classe gérant la boîte à boutons ou les choix exclusifs dans les squelettes, et qui est un Modèle de MVC, correspond à cette règle.

– *Une barre de menus est modélisée par un agent :* dans MacApp, une partie de la responsabilité de la classe « Document » est la gestion de la barre de menus. Chaque squelette d'application définit un mécanisme pour gérer la barre de menus lorsqu'elle peut être modifiée.

– *Une zone d'édition est modélisée par un agent :* cela correspond à la classe *Viewer* d'Unidraw.

– *Un concept complexe d'une zone d'édition est modélisé par un agent :* cela correspond à l'utilisation de certains objets graphiques disponibles dans Unidraw (*ComponentViews*) ou dans Garnet. Cependant, dans PAC-AMODEUS, les interactions nécessitent une remontée vers le noyau sémantique. Le fait d'implanter une partie de cet agent dans les objets graphiques revient à déléguer une partie du noyau sémantique dans la présentation (nous emploierons le terme de *délégation sémantique*).

– *Si, depuis une fenêtre « espace de travail » contenant un concept de donnée, il est possible d'ouvrir un espace de travail sur un autre exemplaire de la même classe, ces deux espaces sont modélisés comme des agents fils d'un agent père commun :* c'est une généralisation de la première règle. Elle signifie que la règle de MVC doit être suivie dès qu'on peut avoir plusieurs vues sur un même objet.

– *Si, depuis une fenêtre d'édition qui restitue, de manière synthétique, un ou plusieurs concepts composés, il est possible d'ouvrir une nouvelle fenêtre représentant plus en détail le contenu d'un de ces concepts, l'agent qui modélise la nouvelle fenêtre est fils de l'agent source :* cette règle est spécifique aux agents qui se combinent très facilement en cascade. Dans une architecture basée sur les objets et MVC, une vue composite contient plusieurs vues qui sont toujours liées à un modèle ; comme les rôles du modèle et de la vue ne sont pas symétriques, il est naturel de revenir au modèle lié à la vue synthétique lors de la création d'une nouvelle vue détaillée.

– *Si la spécification d'une commande implique des actions distribuées sur plusieurs agents, ceux-ci doivent être placés sous le contrôle d'un agent qui cimente les actions réparties en une commande :* cette règle n'a pas d'équi-

valent direct dans les squelettes d'applications mais elle en a dans les motifs de conception : le motif *mediator* (page 78).

– *Un agent PAC doit être dédié à la gestion des erreurs sémantiques :* cette règle est intéressante car elle n'a pas d'équivalent dans les squelettes d'application, qui peuvent au mieux offrir des mécanismes de gestion de boîtes de dialogue. Lorsque la gestion des erreurs ne se limite pas à afficher une boîte de dialogue et annuler la commande, les stratégions de gestion des erreurs sémantiques sont typiquement du domaine du concept et non pas des outils.

– *Utiliser des références symboliques pour la coopération entre agents afin d'assurer leur réutilisation :* cette règle est un peu éloignée des langages à objets dans lesquels les références sont faites sur des objets appartenant à une classe. Un nom n'est donc pas suffisant pour qualifier une information. Les agents sont potentiellement plus modulaires de ce point de vue. Nous discutons plus précisément de ce point à la fin de cette section.

– *Un agent PAC et un agent fils unique peuvent être regroupés en un seul agent :* cette règle est similaire à celle d'unification structurelle utilisée par InterViews (décrit en deuxième partie, § 4.2) et ET++ [Weinand et al.88].

– *Un agent dont le rôle peut être encapsulé par un objet de présentation ou d'interaction peut être éliminé et apparaître comme un composant de la présentation de son agent père :* cette règle met en évidence le point fort de l'aspect variable du modèle de l'Arche : ce qui est encapsulé par la présentation ou l'interaction peut être ignoré par le contrôleur du dialogue. D'un point de vue génie logiciel, cette règle rend imprécise la sémantique du modèle représenté dans le contrôleur. En effet, une partie de la sémantique est alors prise en charge par les objets de la présentation et de l'interaction, et cette partie n'est pas décrite explicitement dans le modèle. Cela laisse des ambiguïtés dans la spécification et le comportement.

5.4.2 Analyse critique

PAC-AMODEUS décrit de façon assez précise la nature des agents à créer dans le contrôleur de dialogue pour construire une application graphique quelconque, et en particulier un éditeur. À partir des règles énoncées, il est possible de concevoir l'organisation du contrôleur de dialogue lié à une application graphique interactive.

Cependant, ces règles ont été utilisées dans le cadre d'applications graphiques interactives où la présentation est principalement faite par une configuration stable

de *Widget*s et non un nombre potentiellement important d'objets graphiques complexes. Lorsque le nombre d'objets graphiques manipulés peut être très grand ou que le contrôle de la manipulation directe doit passer par le contrôleur de dialogue, alors plus de détails doivent être donnés sur l'organisation du contrôleur et ses relations fines avec les objets de la présentation et de l'interaction.

La règle concernant la réutilisation des agents par l'utilisation de références symboliques nous paraît similaire au polymorphisme de sous-types [Cardelli et al.85]. D'un point de vue génie logiciel, la flexibilité des agents est plus difficile à contrôler que les classes. Le fait de vouloir abstraire certaines fonctions des agents, de vouloir les rendre réutilisables, revient à reconnaître l'existence de motifs de collaboration abstraits comme ceux décrits à la section suivante.

5.5 Motifs de conception

PAC permet de décomposer une application en plusieurs agents. Cette décomposition uniforme permet une modélisation intéressante opérationnellement car elle fait apparaître des similarités entre classes d'applications, comme on peut le voir dans l'architecture PAC-AMODEUS. Les motifs de conception (*Design Patterns*) [Gamma et al.94] sont des règles méthodologiques qui décrivent l'agencement de plusieurs objets collaborant à la résolution d'un problème de conception.

Contrairement à PAC, les motifs de conception ne décrivent pas une application entière mais sont plutôt des modèles génériques qui offrent des schémas de solutions à des problèmes d'architectures. L'argument est que, si le programmeur se trouve dans la configuration où il doit gérer tel type de relation entre des objets, alors l'utilisation de tel motif de conception s'est avéré efficace. Les bénéfices qu'à un programmeur à suivre les motifs de conception sont multiples :

- pouvoir nommer les mécanismes utilisés dans ses programmes ;
- disposer d'une documentation sur chaque mécanisme, ainsi que d'une source d'aide pour les mettre en œuvre ;
- stéréotyper les parties de programme qui méritent de l'être pour une plus grande clarté.

Une fois acceptée par une communauté d'informaticiens, l'utilisation d'un motif de conception peut être facilitée par des structures syntaxiques d'un langage de programmation. C'est le cas pour le motif « Classe/Instance » qui est utilisé dans des bibliothèques C comme la Toolkit X et qui appartient au langage C++ [Stroustrup91].

Le livre *Design Patterns* décrit trois catégories de motifs : les motifs de création, structurels et de comportement. Ces motifs apparaissent souvent dans la conception des éditeurs graphiques ; connaître leur nom et leur fonction permet de simplifier la description des éditeurs.

Dans la suite de cette section, nous donnons une description succincte des motifs, en illustrant d'exemples ceux que l'on trouve le plus souvent dans le domaine des éditeurs graphiques.

5.5.1 Motifs de création

Cinq motifs décrivent plusieurs façons de créer de nouveaux objets :

Abstract Factory un objet est responsable de la création d'objets de nature semblable.

> Par exemple, un objet *widgetKit* définit des méthodes pour créer tous les objets de l'interaction définis par une boîte à outils. La création d'un label se fait par :
>
> ```
> label = widgetKit->label("hello");
> ```
>
> En utilisant ce motif, une application peut encapsuler la création d'objets de même nature et changer dynamiquement (et non à la compilation) la méthode de construction réelle.

Builder sépare la construction et la représentation d'un objet complexe. Un *Builder* est un objet abstrait qui définit l'interface permettant de construire fonctionnellement un objet. Son implantation varie en fonction de la structure concrète que l'on veut donner à l'objet.

> Par exemple, le décodage d'un format de fichier de description graphique comme celui produit par Illustrator [Adobe Systems Incorporated91] peut se faire en utilisant un *Builder*. Le décodage du format décrivant une entité comme un cercle provoquera l'appel à la méthode *cercle* du *Builder*. Chaque application peut alors utiliser le décodeur de format en lui passant un *Builder* spécialisé qui construira la structure de données au format requis.

Fatory Method définit une méthode abstraite destinée à créer des objets utiles à une classe. La méthode concrète est définie lors de la dérivation de la classe.

> Par exemple, MacApp définit une classe abstraite pour l'*Application* qui fait appel à une méthode abstraite *DoCreateDocument*, responsable de la création d'un nouveau

document. Cette méthode ne peut évidemment pas être définie génériquement mais le squelette d'application peut néanmoins l'utiliser dans la classe *Application*.

Prototype spécifie le type d'objet à créer en utilisant une instance prototype, et crée de nouveaux objets en copiant ce prototype. Ce motif est utilisé par Garnet et par Unidraw pour la création de nouveaux objets graphiques.

Singleton garantit qu'une classe n'a qu'une seule instance et définit son point d'accès.

Par exemple, InterViews comme MacApp spécifie qu'une seule instance de la classe **Session** doit exister. Cette instance est créée en début de programme et est détruite à la fin. (Elle peut être accédée à l'aide de la méthode *Session::instance()*).

5.5.2 Motifs de structures

Ils sont de sept types :

Adapter convertit l'interface d'une classe en une autre interface, généralement pour rendre compatible des classes issues de mondes différents.

Bridge découple une classe de son implantation afin que les deux puissent être modifiés indépendamment.

Composite compose des objets de façon arborescente, offrant une interface minimale pour parcourir l'arborescence de façon uniforme sur chaque nœud. Le schéma de collaboration des classes implantant le motif *composite* est donné en 5.6 car nous ferons référence à ce motif pour décrire notre architecture.

Par exemple, Fresco utilise ce motif pour définir les **Glyph**s, qui s'organisent sous forme de GDA. Une classe de composite définit des méthodes pour accéder au contenu des feuilles et des méthodes pour parcourir l'arborescence. Les méthodes d'accès au contenu des feuilles retournent une valeur particulière sur des nœuds non terminaux pour signifier que le contenu n'est pas défini, et les méthodes de parcours ne descendent pas au-delà des feuilles.

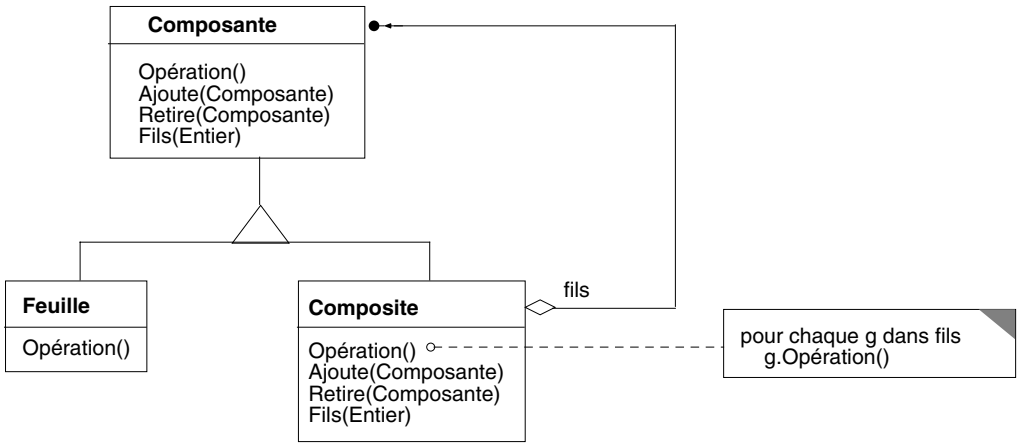


FIG. 5.6 - *Schéma OMT de collaboration des classes participant au motif* Composite

Decorator permet d'ajouter de nouvelles responsabilités dynamiquement à un objet. Les *Decorator*s sont des alternatives flexibles à la dérivation de type. Le schéma de collaboration des classes implantant le motif *Decorator* est donne en 5.7 car nous ferons référence à ce motif pour décrire notre architecture.

> Par exemple, dans Fresco et ET++, les objets graphiques qui décorent d'autres objets graphiques en leur rajoutant des ombres ou des barres de défilement, sont définis avec la même interface que l'objet qu'ils encapsulent. Il est possible de remplacer un objet simple par le même objet décoré, sans que rien d'autre dans l'interface ne soit modifié. Dans un autre registre, ET++ définit des **streams** avec le motif *Decorator* : un stream ramène des caractères avec la méthode *get*. Il existe des *streams* qui modifient le comportement d'un autre *stream* en gérant par exemple la décompression à la volée, ou le décryptage.

Facade définit une interface abstraite à une collection d'objets qui coopèrent pour une tâche. Le Macintosh définit énormément de façades pour gérer les *Managers*, responsable de la gestion d'objets de bas niveau.

> Par exemple, le texte peut être géré à plusieurs niveaux. Au niveau le plus bas, chaque caractère d'une police peut être affiché à un point précis de l'écran. À un niveau supérieur, des paragraphes peuvent être composés avec plusieurs styles typographiques et

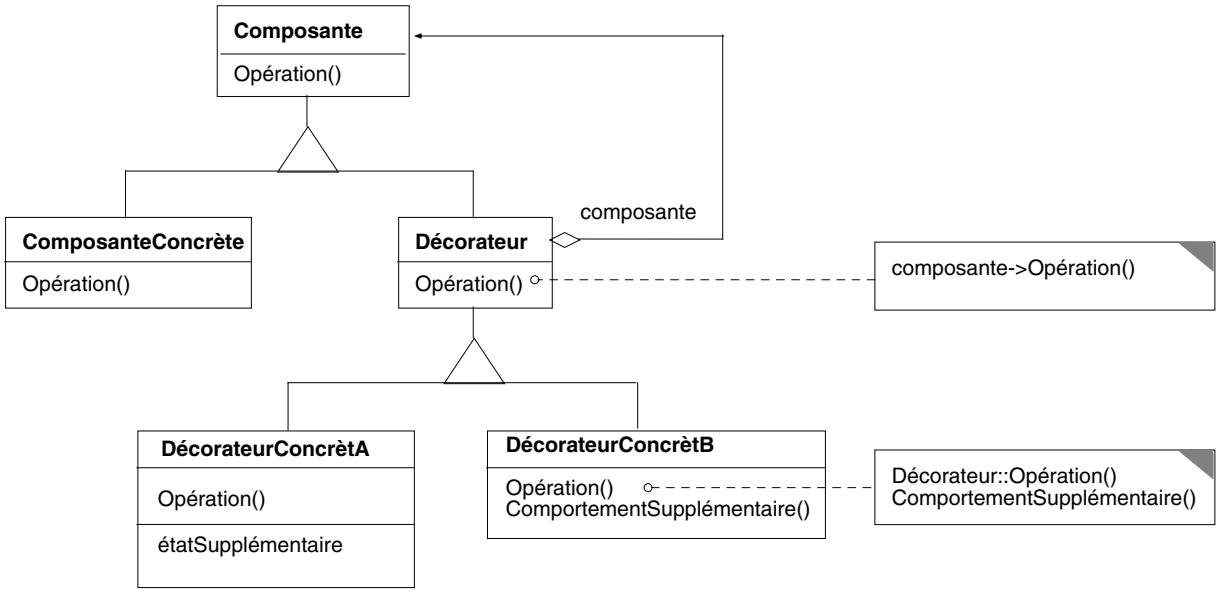


FIG. 5.7 - *Schéma OMT de collaboration des classes participant au motif* Decorator

plusieurs langues. Au niveau supérieur, du texte peut être édité, la gestion du curseur et de la sélection étant pris en charge par un *Manager* de haut niveau. Chacun de ces *Managers* se comporte comme une *Façade* pour les managers de niveaux inférieurs et cachent au programmeur beaucoup de complexité. Ils permettent aussi aux mécanismes de bas niveau d'évoluer tout en gardant l'interface de haut niveau plus stable.

Flyweight utilise le partage de structure pour permettre l'utilisation efficace d'un grand nombre d'objets très simples.

Par exemple, dans Fresco, chaque objet graphique est un **Glyph**, quel que soit son niveau de granularité. Chaque caractère est un **Glyph**, sans que les performances d'un éditeur de texte en soit affecté [Calder et al.90].

Proxy utilise un représentant à la place d'un objet pour contrôler son accès. Le contrôle peut porter sur le moment de création de l'objet, le droit d'accès ou la méthode d'accès dans le cas d'applications réparties.

Par exemple, un éditeur de texte pouvant incorporer des images utilisera un objet qui ne chargera pas l'image au moment ou celle-

ci est référencée, mais attendra le moment où l'image sera réellement affichée pour la charger. Ainsi, les utilisateurs qui chargent le document mais ne consultent pas la partie où l'image apparaît, n'auront pas à attendre le chargement de l'image. Dans d'autres cas, le *Proxy* est utilisé lorsque deux parties d'un programme n'ont pas les mêmes droits d'accès à l'objet. Un représentant permet de filtrer les opérations faites sur l'objet. Enfin, dans les applications réparties, un objet peut exister sur une machine particulière et être référencé d'une autre machine à travers un représentant qui gère la communication réseau. L'application ne sait alors pas si elle utilise un objet local ou distant.

5.5.3 Motifs comportementaux

Ils sont de onze types :

Chain of Responsability évite à un objet demandeur d'être couplé trop fortement avec un objet qui lui répond. Ce motif est utilisé lorsque la réponse peut avoir à être recherchée dans une chaîne d'objets.

Par exemple, certains objets de l'interaction sont capables de donner une aide contextuelle lorsqu'on appuie sur une touche pendant que le pointeur est placé dessus. Cependant, le niveau de l'aide dépend de plusieurs paramètres et l'implantation du mécanisme revient à demander à l'objet graphique placé sous le pointeur son aide. Celui-ci peut ne pas avoir d'aide spécifique et demander à l'objet qui le contient s'il a une aide, etc. Ainsi, la demande sera renvoyée jusqu'à ce qu'un objet réponde.

Command encapsule une requête dans un objet qui peut être manipulé, passé à différents objets, enregistré, et sert de base pour les opérations réversibles. MacApp et Unidraw utilisent ce motif pour offrir les fonctionnalités défaire et refaire.

Interpreter pour un langage donné, définit une représentation pour sa grammaire ainsi qu'une fonction pour interpréter cette représentation.

Iterator permet de parcourir séquentiellement tous les éléments d'une structure de données conteneur sans exposer son implantation. Ce motif est une des abstractions du langage CLU [Liskov et al.77].

Mediator définit un objet qui encapsule la communication entre plusieurs objets coopérants dans une tâche, afin d'éviter que chaque objet ait à connaître explicitement tous les autres qui sont concernés par ses modifications.

> Par exemple, une boîte de dialogue permettant de choisir une police de caractère peut avoir à afficher plusieurs objets de l'interaction, des labels représentant des textes d'exemple, des menus pour choisir un style, une liste pour choisir une police, etc. La modification d'un élément peut se répercuter sur beaucoup d'autres ; néanmoins, il est important d'éviter que chaque objet connaisse tous les objets qui dépendent de la modification de son état, sans quoi la modification de l'interface devient exponentiellement compliquée avec le nombre d'objets de l'interface utilisé. Utiliser un *Mediator* revient à définir un nouvel objet qui encapsule l'état de ce qui doit être utilisé dans la boîte de dialogue. Chaque objet de l'interaction se contente alors de se mettre à jour en fonction de la valeur de ce *Mediator* et, lorsqu'il est modifié, de mettre à jour le *Mediator*.

Memento permet de capturer ou de sauvegarder l'état d'un objet sans violer le principe d'encapsulation. Une classe abstraite **Memento** est définie et pour chaque classe qui doit sauvegarder son état, on définit une classe dérivée de **Memento** dans laquelle on rajoute les champs nécessaires à stocker cet état. Une méthode est nécessaire pour demander l'état et une autre est nécessaire pour remettre l'état à jour.

Observer définit une relation de dépendance de un à plusieurs, de manière à ce que lors de la modification d'un objet, tous les objets qui en dépendent soient prévenus et mis à jour automatiquement.

> Par exemple, dans la triade MVC, chaque vue est dépendant de son modèle, ainsi que les contrôleurs parfois. La modification du modèle entraîne automatiquement la notification des vues et des contrôleurs et leur remise à jour.

State permet à un objet de changer de comportement lorsque son état interne change. Il se comporte alors comme s'il avait changé de classe.

> Par exemple, les éditeurs définissent des « outils » qui sont des objets qui gèrent la façon dont l'application répondra aux événements. Suivant l'état de l'outil, l'éditeur changera de comportement.

Strategy définit une famille d'algorithmes et encapsule chacun dans un objet, de manière à pouvoir utiliser l'un ou l'autre en fonction des besoins.

Par exemple, un manipulateur de Unidraw se comporte comme une *Strategy*. Il est possible d'appliquer un manipulateur pour sélectionner, déplacer, agrandir. un objet graphique. Le choix du manipulateur est fonction de l'outil courant de l'application.

Template Method définit le squelette d'un algorithme dans une classe en faisant appel à des méthodes abstraites qui seront définies dans une dérivation concrète de la classe.

Par exemple, le squelette d'applications MacApp implante les méthodes *Document::save()* et *Document::load()* à partir de méthodes qui doivent être définies par dérivation de la classe **Document** (comme *AddDocument*, *OpenDocument*, *DoCreateDocument*, *CanOpenDocument*, *AboutToOpenDocument*).

Visitor représente une opération à appliquer aux éléments constituant la structure d'un objet.

Prenons par exemple un éditeur utilisant une structure graphique pouvant contenir quatre types d'objets graphiques : GroupeGraphique, BézierGraphique, ImageGraphique et TexteGraphique, suivant le motif *Composite*. Une classe suivant le motif *Visitor* peut être définie pour parcourir l'arbre en appliquant une opération sur chaque nœud en fonction de son type. Les classes abstraites VisiteurGraphique et Graphique auront alors la définition suivante :

```
class VisiteurGraphique {
        VisiteurGraphique();
        ~VisiteurGraphique();
        void visit_groupe(GroupeGraphique*) = 0;
        void visit_bézier(BézierGraphique*) = 0;
        void visit_image(ImageGraphique*) = 0;
        void visit_texte(TexteGraphique*) = 0;
};

class Graphique {
        Graphique();
        ~Graphique();
        void visit(VisiteurGraphique*) = 0;
```

```
};
```

> Chaque classe graphique doit définir la fonction *visit* en appelant la fonction du VisiteurGraphique correspondant à son type concret. Il est alors possible de créer un visiteur pour dessiner l'arbre des graphiques, pour l'imprimer, le sauvegarder, ou plus généralement parcourir l'arbre en appliquant un traitement spécifique à chaque type de nœud.

Les motifs décrits ici ne sont certainement pas exhaustifs, la liste augmentera au fur et à mesure que de nouvelles pratiques se répandront et qu'elles recevront l'aval des concepteurs d'applications à objets.

5.5.4 Analyse critique

L'idée de bâtir un squelette d'application qui puisse s'appliquer à tous les éditeurs est une utopie. Certaines évolutions des matériels ou des modes d'interaction (meilleure reconnaissance de l'écriture manuelle, ou du langage) peuvent avoir des implications structurelles importantes sur les éditeurs qui voudraient en tirer profit.

Plutôt que de persister dans la voie de l'architecture logicielle, les auteurs de *Design Patterns* ont préféré donner des règles empiriques d'agencements d'objets, ainsi que leurs responsabilités, afin de laisser une plus grande flexibilité architecturale en réutilisant des structures organisationnelles.

Nous pensons que, compte tenu de la complexité inhérente à la construction d'éditeurs, il serait illusoire d'attendre que des boîtes à outils offrent toutes les solutions. Si du temps doit être investi dans la spécification et la mise au point des structures de données et de contrôle spécialisées pour un éditeur, alors suivre des méta-structures est un bon moyen d'éviter de s'égarer et donne des bonnes pistes pour la compréhension et la maintenance des programmes.

À défaut d'offrir des outils, les motifs de conception donnent un vocabulaire pour décrire des architectures que nous utilisons dans le reste de cette thèse, et un guide de style structurel pour bâtir les parties spécifiques d'un éditeur.

Chapitre 6

Conclusion de la première partie

Dans cette partie, nous avons passé en revue les architectures logicielles et les modèles d'architectures qui permettent de bâtir des éditeurs.

Les architectures logicielles organisent une application graphique interactive en entités concrètes qui communiquent en suivant des schémas préétablis mais configurables. Pour arriver à simplifier la construction de ces applications, les architectures logicielles se sont axées sur la reconnaissance et la génération de stéréotypes, ce qui convient parfaitement au niveau *Widget* des éditeurs.

Pour aller au-delà des *Widget*s et décrire l'interaction graphique et la manipulation directe, les architectures logicielles ne proposent que des objets simples, affichant du graphique 2D schématique et manipulable à l'aide d'objets interacteurs qui sont difficilement configurables et dont l'extensibilité est très limitée. Ces limitations sont rédhibitoires pour beaucoup de types d'éditeurs graphiques, qui nécessitent un modèle graphique ou des dispositifs d'entrée particuliers.

Pourtant, l'unification de la visualisation des objets graphiques proposée par Fresco montre que différents modèles graphiques peuvent être factorisés par une architecture logicielle générique. Mais, pour être complets, les éditeurs doivent pouvoir aussi gérer des dispositifs d'entrées variés ainsi que la manipulation directe, aussi bien dans son aspect graphique (retour d'information) que dans son aspect comportemental, ce que ne résout pas Fresco.

Les modèles ont suivi une évolution semblable aux architectures logicielles. PAC est le pendant abstrait de MVC et les modèles généraux ont évolué conformément aux squelettes d'applications, en cherchant à faire apparaître les stéréotypes — qui varient d'un domaine à un autre — et en précisant le contenu de chacune des composantes abstraites des éditeurs. La subdivision des composantes à l'aide de PAC a permis d'arriver à un niveau important de finesse dans la modélisation, mais le choix des agents PAC utilisés par PAC-AMODEUS par exemple, reste encore empirique et n'a été utilisé, à présent, que pour modéliser la composante de dialogue des applications graphiques interactives. L'usage permettra certainement

d'ajouter un répertoire d'agents PAC pour modéliser plus finement le comportement de toutes les parties d'un éditeur, lorsque celles-ci seront clairement individualisées.

Plutôt que d'essayer d'encapsuler les stéréotypes apparaissant dans les éditeurs dans des « boîtes noires » destinées à cacher une complexité indéniable mais limitant leurs extensions, nous avons préféré chercher une architecture dans laquelle tous les niveaux de la présentation et de l'interaction sont décrits explicitement. Le problème principal est de rendre une telle architecture suffisamment générale pour implanter une grande variété d'éditeurs sans que les mécanismes à mettre en œuvre soient trop nombreux ou complexes.

Deuxième partie

Architecture multicouche

Chapitre 1

Présentation

Comme nous l'avons vu dans la première partie, tous les outils de construction d'interface utilisent le même modèle graphique pour gérer les objets visualisés et les objets graphiques de l'interaction.

Pour les événements, les boîtes à outils de bas niveau comme Motif ne donnent aucun mécanisme pour la manipulation directe tandis que les boîtes à outils de haut niveau sont basées sur les interacteurs de Garnet qui stéréotypent tellement la gestion des événements que de nouveaux modes d'interaction sont pratiquement impossibles à rajouter.

Dans les boîtes à outils qui supportent la gestion du graphique structuré, l'affichage des objets graphiques se fait sur une surface de dessin virtuelle rendue visible sur une fenêtre de l'écran à travers un point de vue. Les objets graphiques affichés peuvent être très différents : certains objets sont principalement inertes (le fond du graphique par exemple), d'autres changent souvent (vidéo affichée), d'autres encore changent parfois (objet sélectionné lorsqu'il est manipulé par la souris). Dans les modèles architecturaux actuels, tous les objets graphiques sont affichés sur la même surface graphique, sur un seul niveau. La seule exception est Display PostScript, qui préconise de séparer les objets graphiques en couches gérées explicitement par l'application à un niveau très bas.

Plutôt que de gérer les objets graphiques au sein d'un seul niveau graphique, nous proposons d'utiliser autant de niveaux que nécessaire pour séparer les éléments graphiques qui suivent un comportement spécifique. Une couche peut être créée pour :

- gérer les événements,
- gérer des objets graphiques,
- améliorer les performances.

Pour être visualisées, les couches sont placées dans une pile, qui les composent pour qu'elles s'affichent sur une surface virtuelle. Cette composition aussi peut être

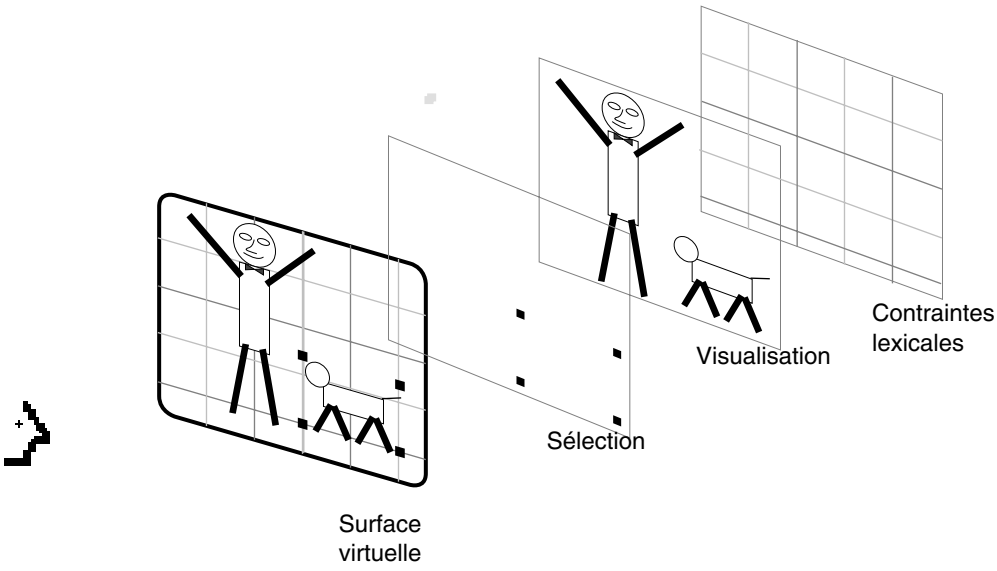


FIG. 1.1 - *Composition des couches graphiques dans notre architecture.*

spécialisée pour tirer profit de spécificités matérielles ou logicielles et de la sémantique graphique des couches.

La figure 1 montre l'ordre dans lequel les couches sont composées dans une pile pour obtenir une image finale. Les couches s'affichent du fond vers le premier plan tandis que les événements suivent le sens inverse.

Nous allons maintenant décrire notre architecture en quatre parties :

gestion du graphique de la couche : chaque couche de notre architecture peut afficher des objets en suivant le modèle graphique dont il a besoin ;

gestion des événements de la couche : suivant l'événement qui lui arrive, la couche peut décider de le traiter ou de le laisser passer à la couche suivante ;

gestion graphique de la superposition : lorsqu'une couche est modifiée dans une pile, l'image finale de la surface virtuelle doit être recalculée, généralement en réaffichant les couches du fond vers le premier plan ;

gestion concertée des événements entre les couches : la coopération de plusieurs couches dans la gestion des événements permet d'obtenir des fonctionnalités semblables aux interacteurs de Garnet mais non prédéfinies.

Dans les chapitres qui suivent, nous décrivons les couches qui peuvent être distinguées dans les éditeurs existants, et certains systèmes particuliers dans lesquelles des couches graphiques sont utilisées. Nous décrivons ensuite plus formellement les différents aspects de l'architecture : le modèle d'affichage (la sortie) et

le modèle de gestion des événements (l'entrée). Nous décrivons ensuite la sémantique graphique des couches introduites dans les premiers chapitres et montrons les mécanismes qui permettent d'optimiser le réaffichage et les mécanismes qui permettent de gérer les événements, pour chaque couche et pour une pile.

1.1 Décomposition des éditeurs en couches

Suivant leur complexité, on peut distinguer entre deux et un dizaine de couches graphiques dans les éditeurs. Les plus simples, ceux qui ne font que de la visualisation passive, distinguent déjà deux couches graphiques : le fond et la couche de visualisation des données.

Dans cette section, nous décrivons intuitivement l'intérêt architectural d'isoler certaines couches en montrant des exemples de couches utilisées dans divers éditeurs.

1.1.1 Fond

Le fond des applications graphiques a une fonction qui peut être neutre, décorative, support de métaphore ou de schéma. Dans les deux premiers cas, la nature du graphique de la couche est indépendante de la sémantique de l'application graphique. Le choix peut être motivé par :

des raisons ergonomiques : un fond gris neutre pour une application utilisée intensivement permet de ne pas fatiguer les yeux ;

des raisons décoratives : le logo de la société qui a créé le programme ;

pour appuyer une métaphore : dans une application iconique comme le *Finder* du Macintosh, le fond de l'écran est censé supporter la métaphore du bureau, et dans HyperCard de Apple [Goodman87], le graphique du fond est généralement utilisé pour appuyer une métaphore ;

ou pour représenter un fond de carte : dans les systèmes de contrôle aérien, le fond de l'écran affiche les balises et les pistes qui guident les avions.

Les systèmes de fenêtrage disposent de certaines primitives spéciales pour gérer le fond. Dans X par exemple, lorsqu'une fenêtre est créée, il est possible de lui associer un fond qui a une couleur, un motif ou une image. Le système de fenêtrage se charge d'afficher le fond lorsque la fenêtre est effacée.

1.1.2 Visualisation passive

La couche de visualisation passive des données (VP), comme son nom l'indique, se contente de visualiser une structure graphique. C'est cette couche qui affiche les objets graphiques principaux sur lesquel l'utilisateur veut agir. L'affichage peut nécessiter un modèle graphique particulier, implanté par une extension le cas échéant.

1.1.3 Gestion de la sélection

Lorsqu'un éditeur doit gérer la sélection, il a le choix entre l'utilisation d'attributs graphiques particuliers ou l'utilisation d'objets graphiques particuliers, comme décrit en première partie, § 4.2.4.6. Par exemple :

dans les éditeurs de texte, la portion de texte sélectionnée est généralement représentée avec une couleur de fond et de texte différente du texte non sélectionné ;

dans les éditeurs schématiques/iconiques, les objets sélectionnés sont généralement représentés avec des couleurs spécifiques ou grisés ;

dans les éditeurs graphiques, les objets sélectionnés sont « décorés » d'éléments graphiques qui permettent de les distinguer (les « poignées »). Ces éléments graphiques ont généralement une sémantique particulière, à la fois graphique — ils s'affichent rapidement et peuvent s'effacer rapidement — et interactive — ils sont réactifs au pointeur et la manipulation qu'ils déclenchent dépend de l'emplacement du pointeur lorsque la manipulation est commencée.

Dans les deux premiers cas, la visualisation de la sélection est généralement faite directement par l'objet graphique de la couche de visualisation passive, tandis que dans le dernier cas, il est intéressant de disposer d'une couche spécifique pour gérer la sélection. C'est la *couche de gestion de la sélection*. Le comportement de cette couche est décrit plus en détail en § 2.3.4.

La raison pour laquelle les deux premières catégories peuvent se passer d'utiliser une couche de gestion de la sélection tient au fait que leur visualisation n'utilise pas tous les attributs graphiques disponibles dans le modèle graphique des systèmes de fenêtrage. Le texte utilise deux couleurs, une pour le texte lui-même et une pour le fond, tandis que les éditeurs schématiques associent des couleurs conventionnelles à leurs éléments et peuvent donc décider d'utiliser une couleur/texture également conventionnelle pour représenter l'état sélectionné.

Les éditeurs graphiques généraux ne peuvent pas toujours utiliser un attribut graphique libre pour visualiser le fait qu'un objet graphique est sélectionné car ils permettent justement aux objets graphiques d'utiliser tous les attributs disponibles.

C'est la raison pour laquelle la sélection est visualisée par des objets graphiques particuliers, qui sont conçus pour se distinguer graphiquement des objets de la couche de visualisation passive. La méthode la plus efficace est certainement celle utilisée par PhotoShop [Adobe Systems Incorporated93a] qui trace la sélection en pointillés animés. Il est alors impossible de confondre la sélection avec le dessin.

1.1.4 Manipulation directe

Un des objectifs déclarés de certains concepteurs d'architectures permettant la manipulation directe est d'offrir un mode de manipulation conforme à la réalité : on prend un objet et on le déplace. Les *Pacers* en sont un exemple en 2D (décrits plus loin). En 3D, plusieurs boîtes à outils comme *Upfront* [Alias Research Inc.b] permettent une manipulation en perspective. Cependant, dans notre monde physique, les objets opaques cachent les objets placés derrière eux, tandis que les objets virtuels sur un écran n'ont pas nécessairement cet inconvénient. Il suffit qu'ils se dessinent au-dessus des autres objets pendant la manipulation directe. Leur allouer une couche permet ce résultat.

Lorsqu'un objet graphique est manipulé interactivement et que sa forme se modifie continuement à l'écran, les boîtes à outils InterViews et Fresco préconisent de modifier la structure graphique utilisée pour sa visualisation. Garnet permet de choisir entre la manipulation des objets graphiques et la création d'objets intérimaires avec lesquels sera visualisée la manipulation. Unidraw et ET++ créent des objets graphiques particuliers pour visualiser la manipulation directe.

Il existe plusieurs raisons pour utiliser une couche spéciale pour la manipulation directe (MD). En particulier lorsque la sémantique graphique doit être relâchée, schématique ou adaptée.

Sémantique graphique relâchée : Les éditeurs graphiques, comme Canvas et Idraw (fait avec Unidraw), utilisent une sémantique graphique relâchée pendant la manipulation directe. Les objets sélectionnés, lorsqu'ils sont manipulés, sont représentés par des rectangles englobant leur forme initiale, permettant ainsi à la manipulation de se faire rapidement dans la plupart des cas. Illustrator n'affiche pas que des rectangles, il affiche les courbes de construction du dessin, tracées de façon optimisées avec une épaisseur d'un pixel.

Sémantique graphique schématique : Dans les interfaces iconiques comme le *Finder* du Macintosh, les icones sont tracés en grisé lorsqu'ils sont déplacés. Ce mode de dessin n'accélère pas vraiment les performances mais permet de distinguer durant la manipulation les objets manipulés des objets en place.

Sémantique graphique adaptée : En dehors du modèle graphique, on peut vouloir modifier le comportement des objets durant leur manipulation directe. Par exemple, les *Pacers* décrits dans [Tang et al.93] sont des objets graphiques qui peuvent être manipulés directement et qui adaptent la qualité de leur affichage à la vitesse de la manipulation. S'ils n'arrivent pas à s'afficher suffisamment rapidement, ils dégradent leur qualité. Dans l'article original, ces objets sont directement les objets graphiques de la visualisation qui sont utilisés pendant la manipulation directe. En utilisant l'architecture multicouche, les *Pacers* pourraient devenir les objets de la manipulation directe et s'adapter plus facilement aux manipulations qu'on leur applique. Nous discutons de cet aspect en § 2.3.5.

1.1.5 Rectangle ou lasso de sélection

Dans la plupart des éditeurs, lorsque le mode de sélection est actif, la sélection peut se faire de deux manières : soit en cliquant directement sur un objet graphique, soit en cliquant en dehors de tout objet graphique et en tirant un rectangle de sélection qui suit le pointeur et, lorsque l'action est terminée, sélectionne tous les objets graphiques apparaissant à l'intérieur du rectangle. Le rôle de cette couche est d'afficher le rectangle et de gérer sa manipulation. Cette couche est décrite en § 2.3.6.

Remarque : Un des principes de notre architecture est d'allouer une couche pour chaque ensemble d'objets graphiques ayant sa propre sémantique. Bien qu'il soit possible d'utiliser une même couche, par exemple pour le lasso de sélection et la gestion de la sélection, la représentation graphique et le comportement des objets de ces deux couches sont différents nous utilisons donc deux couches.

1.1.6 Contraintes lexicales

Certains éditeurs graphiques comme Illustrator de Adobe, Canvas de Deneba ou MacDraw de Apple permettent de spécifier que les points de contrôle des objets graphiques doivent toujours être sur une grille dont la taille des mailles peut être définie. Pendant les phases de manipulation directe et de spécification de points à l'aide de la souris, ces applications modifient les coordonnées utilisées par le programme pour être le point de la grille le plus proche de la position du pointeur.

La grille elle-même peut être visualisée, elle appartient alors à une couche placée entre le fond et la couche de visualisation passive. Cette couche (CL) est décrite en § 2.3.2.

1.1.7 Retour d'information lexicale

Pour les dispositifs d'entrée positionnels comme les souris, les tablettes à numériser ou les boîtes à boutons, une information doit apparaître sur l'écran pour donner un retour visuel de leur position et de leur état. Ce retour visuel se caractérise généralement par une icone dont un point particulier désigne l'emplacement du dispositif (la pointe d'une flèche ou le centre d'une croix, etc.). Cette description longue peut sembler triviale, parce que tous les systèmes de fenêtrage s'occupent de gérer ce retour, mais en réalité, ce retour peut nécessiter une gestion plus compliquée qu'un icone qui suit un dispositif. Par exemple, les systèmes de fenêtrage ne savent pas gérer aujourd'hui plusieurs dispositifs d'entrée avec leur curseur. Une couche spécifique est donc utile pour gérer le retour lexical (RL) des dispositifs dans l'application graphique. Nous en discutons en § 2.3.7.

1.1.8 Autres utilisations des couches

L'usage des couches n'est pas limité. Nous avons décrit ici celles qui sont utilisées de façon semblable dans beaucoup d'applications graphiques interactives, mais d'autres utilisations intéressantes existent. Pour n'en citer que quelques unes :

- la séparation explicite d'un dessin en couches, comme le permet Canvas ou PhotoShop ;
- le placement d'un modèle numérisé sous la couche de visualisation pour permettre de décalquer virtuellement, comme sous Illustrator ;
- le placement d'une zone translucide au-dessus de la couche de visualisation pour délimiter la zone sur laquelle un outil est actif ; ce sont les *See Through Tools* [Bier et al.94] décrits dans la section suivante ;
- le retour des manipulations d'un autre utilisateur dans un éditeur collecticiel [Karsenty94].

1.2 La superposition dans les systèmes graphiques

Certains systèmes, que nous passons en revue, gèrent aussi des couches graphiques.

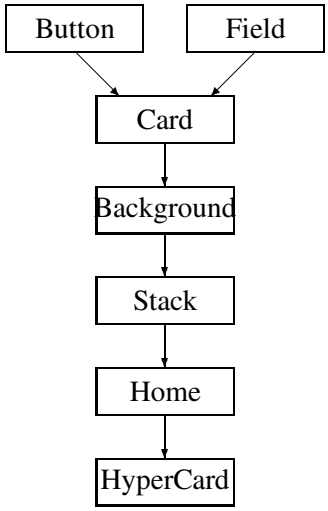


FIG. 1.2 - *Hiérarchie des objets HyperCard.*

1.2.1 Squelettes d'applications

Nous avons trouvé dans toutes les architectures d'applications graphiques hautement interactives, un système embryonnaire d'organisation du graphique en couches. Ainsi, Unidraw, au sommet de la hiérarchie des objets graphiques à éditer, crée un objet de type *Viewer*. Cet objet contient un sous-objet qui gère trois couches : une page, une grille et les objets graphiques eux-mêmes.

NextSTEP préconise l'utilisation de couches graphiques, comme nous l'avons décrit en deuxième partie, § 4.3.2. Ces couches ne sont utilisées que pendant la manipulation directe et restent du domaine de l'optimisation graphique.

1.2.2 HyperCard

HyperCard est un environnement de développement graphique accompagné d'un langage et qui est utilisé pour la construction d'applications graphiques interactives. Son modèle graphique est original car il propose deux niveaux pour l'affichage et six niveaux pour la gestion des événements, comme décrit en figure 1.2.

HyperCard utilise la métaphore de la carte pour visualiser des informations. Un document est une pile, composée de cartes. Chaque carte est composée d'un fond et d'un premier plan, chacun contenant des objets de trois types : texte, graphique et bouton. Plusieurs cartes peuvent avoir le même fond.

Bien que relativement rudimentaire, l'environnement d'HyperCard permet de réaliser des applications très rapidement, grâce à son langage interprété HyperTalk et à la facilité d'étendre le langage.

L'idée de factoriser le fond des cartes en leur allouant une couche graphique s'avère très commode pour la fabrication des documents de type Hypertexte. L'idée de faire parcourir un chemin hiérarchique aux événements permet aussi une bonne factorisation de code.

Sans vouloir départager quelle spécificité rend HyperCard aussi attractif, force est de constater que des projets sérieux de recherche en Interaction Homme-Machine l'utilisent comme environnement de base.

1.2.3 NeWS

NeWS [Gosling et al.89, Gosling86] a été un concurrent au système de fenêtrage X11, puis en a été une extension avant d'être abandonné par Sun en 1993. NeWS est un système de fenêtrage client-serveur asynchrone, comme X, qui utilise le langage PostScript comme langage graphique et comme langage de spécification du protocole entre l'application et le serveur. L'idée maîtresse est que, par rapport à X, chaque application peut charger dans le serveur le code PostScript qui permet ensuite une communication optimisée client-serveur et une bonne séparation entre la gestion du graphique, faite dans le serveur, et la définition des structures graphiques, envoyées par le client.

NeWS a été le précurseur de Display PostScript mais a fait des choix radicalement différents : les dispositifs sont gérés en PostScript (avec l'aide d'une extension multi-tâches) et la manipulation directe utilise un concept spécial : les *overlay canvas*.

Le problème principal du langage PostScript pour l'interaction est lié au modèle graphique du peintre. Rien n'est prévu dans ce modèle graphique pour gérer le réaffichage rapide d'un objet. Dans NeWS, chaque fenêtre contient une surface virtuelle (*canvas*) qui suit le modèle graphique PostScript. En plus de ce *canvas*, chaque fenêtre contient aussi un *overlay canvas* qui est une surface virtuelle implantant une *sémantique graphique relâchée*. Ainsi, NeWS garantit que tous les objets affichés sur ce *canvas* seront maintenus affichés, quitte à les effacer et les réafficher automatiquement si le *canvas* normal est modifié. Pour éviter que ces opérations soient trop coûteuses, seul le contour des objets est dessiné mais le programmeur d'application peut utiliser tous les opérateurs PostScript, les attributs graphiques trop coûteux étant simplement ignorés.

1.2.4 Les Sprites

Les Sprites [Foley et al.90, page 1065] sont des petites images pixellaires qui sont gérées directement par le contrôleur vidéo d'un ordinateur et qui sont surtout utilisées dans les jeux vidéos, généralement pour afficher des personnages qui se déplacent sur un décor.

Beaucoup de cartes vidéos permettent d'utiliser les sprites aujourd'hui. Les jeux électroniques les utilisent intensivement, pour afficher parfois plusieurs dizaines d'objets en même temps. Les stations de travail disposent aussi généralement de quelques sprites mais ils sont généralement utilisés par le système de fenêtrage pour afficher le curseur.

Avec les contrôleurs graphiques récents, la gestion des sprites peut être sortie du contrôleur vidéo et laissée au logiciel, qui dispose d'assez de temps entre deux trames pour afficher plusieurs dizaines d'objets graphiques dans une partie de la mémoire vidéo. Le sprite reste néanmoins un des objets graphiques destiné à gérer la superposition qui est le plus utilisé.

1.2.5 Les *See Through Tools*

Les *See Through Tools* [Bier et al.94] sont un paradigme de manipulation directe utilisant les deux mains. Tandis que la main dominante utilise une souris pour effectuer ses manipulations directes, la main non dominante déplace sur l'écran un ensemble de zones translucides, chacune des zones représentant un outil au sens où nous l'avons décrit en première partie, § 4.2.4.7.

La boîte d'outils, au lieu d'être fixe et modale comme dans les éditeurs classiques, peut être manipulée pour être placée au-dessus de la zone d'édition. Les rectangles représentant un mode d'interaction (un outil) sont translucides, permettant ainsi de voir les objets à manipuler. Une manipulation suit le mode d'interaction implanté par l'outil à travers laquelle elle a été initiée.

Indépendamment du paradigme d'interaction, les *See Through Tools* utilisent trois couches de façon très précise, ce qui les rend moins généraux que notre architecture multicouche. Par ailleurs, l'implantation a été difficile car aucun système de fenêtrage ne gère convenablement la transparence. En réalité, les *See Through Tools* sont au-dessus des fenêtres et non à l'intérieur, ce qui rend impossible l'utilisation de notre architecture, à moins de considérer le gestionnaire de fenêtres comme un éditeur particulier, ce que nous n'étudierons pas ici.

1.2.6 Synthèse

Tous les systèmes que nous avons décrits ont une architecture qui repose explicitement sur l'existence de couches graphiques. Cependant, elles n'en généralisent pas l'usage comme nous proposons de le faire.

Chapitre 2

Architecture multicouche

Notre architecture multicouche est composée de trois catégories d'objets : la pile, les couches et les outils. La pile gère une liste de couches superposées. Chaque couche gère une surface virtuelle, une structure de donnée adaptée à la visualisation et un ensemble de fonctions s'appliquant à cette structure de données. Un outil gère la manipulation directe ; à chaque couche est associé un outil qui reçoit les événements destinés à cette couche et les traite en faisant appel aux fonctions spécifiques de la couche, ou à des fonctions d'autres couches, de la pile, ou globales.

Nous commençons par présenter l'aspect graphique de l'architecture, qui ne concerne que les couches et la pile. Ensuite, nous décrivons la gestion de la manipulation, qui concerne les couches, la pile et les outils. Nous revenons ensuite avec plus de détails sur les couches introduites en § 1.1 et quelques outils.

Les définitions des types couche et pile sont données en figure 2.1. Les fonctions sont décrites dans la suite du chapitre. Le schéma OMT de la figure 2.2 décrit les relations entre les classes et le principe de dérivation de la classe couche.

Une *pile* est un conteneur de *couches*, elle implante les mécanismes de placement, affichage, réaffichage, et gestion des événements décrits en première partie, § 3.1. Les fonctions *requiert* et *alloue* servent à la négociation de taille pour le placement que nous avons déjà décrit en première partie, § 3.1.2.1.

2.1 Affichage et réaffichage

La gestion du graphique dans les couches peut se décomposer en deux parties :

– le graphique au sein d'une couche, et

– la composition des couches dans une pile.

```
type Pile = classe;
début
        couches: ListeDe(Couche);
        canvas: Canvas;
        région_visible: Région;

        fn requiert(): Dimension;
        proc alloue(in Canvas, in Région);
        proc dessine(inout Canvas, in Région);
        fn intercepte(Événement): Booléen;
        proc traite(Événement);
fin;
```

```
type Couche = classe;
début
        pile: Pile;
        attributs: EnsembleDe(AttributDeCouche);
        dommage: Région;

        fn requiert(): Dimension;
        proc alloue(in Canvas, in Région);
        proc dessine(inout Canvas, in Région);
        proc efface(inout Canvas, in Région);
        fn intercepte(Événement) : Booléen;
        proc traite(in Événement);
        proc endommage(in Région);
        proc installe_outil(in Entier);
        proc désinstalle_outil(in Entier);
        fnoutil_installé(Entier): Booléen;
fin;
```

FIG. 2.1 - *Les classes « Couche » et « Pile ».*

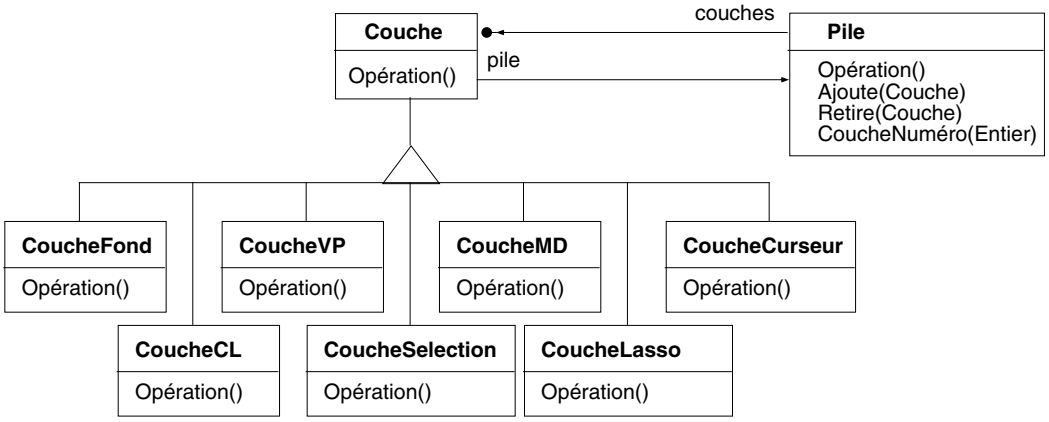


FIG. 2.2 - *Schéma OMT des relations entre les classes Pile, Couche et ses dérivées*

2.1.1 Modèle graphique au sein d'une couche

Sans l'aide de couches multiples, le modèle graphique de la surface de dessin est le même pour tous les éléments affichés. En utilisant plusieurs couches, chaque couche peut spécialiser son modèle graphique et la structure de données utilisée pour représenter les objets graphiques.

> Par exemple, une application de visualisation 3D a besoin d'une surface virtuelle qui suive un modèle graphique 3D, et des objets graphiques organisés pour s'afficher sur cette surface virtuelle. Si cette application désire gérer un pointeur 2D particulier et afficher sa position en permanence, elle n'aura certainement pas besoin du modèle graphique 3D, ni de la même structure de données.

Nous décrirons en § 2.5 les mécanismes permettant aux couches d'utiliser un modèle graphique différent de celui de la surface virtuelle passée à la pile. Nous préférons présenter complètement notre architecture et spécifier les couches standard avant d'y venir.

2.1.2 Composition des couches dans une pile

La pile est responsable de l'ordre de la composition des couches. Lors de l'affichage et du réaffichage, la fonction *dessine* de la pile est appelée avec en argument la surface virtuelle. Elle fait appel à son tour à la fonction *dessine* des couches, qui doit simplement dessiner *par-dessus* le contenu de la surface virtuelle, conformément au modèle du peintre.

La stratégie la plus simple que peut adopter la pile, est d'appeler la fonction *dessine* sur toutes les couches du fond jusqu'au premier plan. Lorsque la surface virtuelle doit être totalement rafraîchie, c'est même la seule option. Cependant, des optimisations peuvent parfois être utilisées pour le réaffichage.

2.1.3 Réaffichage

Lorsque le contenu d'une couche est modifié, le déclenchement du réaffichage se fait en deux temps, comme décrit en première partie, § 3.1.2.5 :

1. La région de la surface virtuelle à réafficher est déclarée endommagée par la fonction *endommage* de la couche. De plus, cette fonction stocke la région dans la couche.

2. La fonction *dessine* de la pile est appelée lorsque le réaffichage de la surface virtuelle est déclenché. La surface virtuelle contient alors une région endommagée, qui est au moins aussi grande, mais peut être plus grande que l'union des régions endommagées contenue dans les couches.

Comme pour la composition des couches dans la pile, la stratégie de réaffichage la plus simple consiste à appeler la fonction *dessine* de la couche de fond jusqu'à la couche de premier plan. Cette stratégie est suffisamment performante pour des éditeurs de texte ou certains éditeurs graphiques 2D schématiques qui utilisent des primitives graphiques affichées rapidement par le système de fenêtrage. Les autres éditeurs doivent utiliser des optimisations que nous préférons décrire en § 2.6 après avoir avoir présenté les couches standard, de manière à faire référence à des cas concrêts. La fonction *efface* est décrite avec les optimisations.

2.2 Gestion des événements

Le modèle de gestion des événements des boîtes à outils distingue deux phases : la désignation et le traitement de l'événement.

2.2.1 Désignation

Lorsqu'un événement est produit, il parcourt toutes les couches de la pile, en commençant par la plus proche de l'utilisateur jusqu'au fond. En principe, l'étape de désignation doit parcourir tous les objets graphiques afin de leur demander s'ils interceptent l'événement et, dans la positive, l'insèrent dans une liste des objets graphiques potentiellement concernés. À la fin du parcours, l'objet graphique de cette liste le plus proche de l'utilisateur est pris en compte.

Dans le cas de la gestion multicouche, toutes les couches n'ont pas besoin d'être traversées ; la première qui accepte de gérer l'événement interrompt le parcours des couches suivantes car c'est celle qui est la plus proche de l'utilisateur. La fonction *intercepte* de la couche retourne *vrai* si la couche accepte de gérer l'événement. La fonction *intercepte* de la pile se contente donc d'appeler la fonction *intercepte* de la première à la dernière couche et de s'arrêter à la première qui retourne *vrai*. Grâce à ce mécanisme, une couche placée au-dessus d'une autre peut intercepter tout événement, qui sera alors invisible aux couches suivantes. Cette propriété est particulièrement utile dans le cadre de la gestion de l'interaction.

> Par exemple, l'éditeur Canvas visualise la sélection d'un objet avec huit poignées noires placées au quatres coins du rectangle englobant l'objet et entre les quatre coins. Lorsqu'on clique sur une des poignées de coin, une manipulation est déclenchée qui retaille l'objet sélectionné. Ces poignées sont des objets graphiques qui ne sont pas dans la même structure graphique que le dessin sélectionné. Lors de l'étape de désignation, Canvas doit d'abord parcourir la structure graphique des poignées avant de parcourir la structure graphique normale. Ce type de

traitement particulier est typiquement ce que le modèle multicouche évite.

2.2.2 Traitement de l'événement dans une couche

Une fois qu'une couche a déclaré qu'elle interceptait un événement, elle le reçoit de nouveau pour le traiter (la fonction *traite* de la couche est appelée). Bien que la fonction *traite* des couches puisse gérer l'événement directement, nous utilisons un objet de type *outil* pour le faire.

2.2.2.1 Les Outils

Un outil est associé à chaque couche et responsable de la gestion des événements qui lui arrivent, à savoir la désignation (la fonction *intercepte*) comme le traitement (la fonction *traite*). Cette architecture offre une grande souplesse en séparant bien les responsabilités entre la gestion du graphique et le traitement des événement.

Plutôt que chaque couche reçoive les événements pour les renvoyer à un outil, nous préférons voir le type *Outil* comme dérivant du type *Couche* pour suivre le motif *decorator* (décrit page 76) comme le décrit le schéma OMT de la figure 2.3. Nous distinguons donc deux catégories de couches : les « outils » et les « couches terminales ». Un outil est vu structurellement comme une couche, mais, par défaut, il délègue son comportement à une couche terminale, sauf pour quelques opérations qu'il redéfinit (typiquement les fonctions *intercepte* et *traite*). Les couches terminales gèrent la visualisation d'une structure graphique et son réaffichage tandis que les modifications interactives de cette structure sont déclenchées par un outil.

Les interacteurs de Garnet et d'Unidraw utilisent le motif *strategy* (décrit page 80) qui les oblige à gérer une boucle d'événements dans l'interacteur afin de maintenir un état durant la manipulation. Notre architecture utilise la boucle de traitement des événements normale. L'état dans lequel se trouve la manipulation est géré par les couches.

2.2.2.2 Communication entre les Outils

Les outils liés aux couches d'une pile peuvent collaborer pour gérer une manipulation complexe. La figure 2.4 montre une pile, trois couches et trois outils qui collaborent pour implanter un mode de sélection à l'aide d'un rectangle de sélection. Les flèches pleines indiquent les liens entre les couches et la pile tandis que les pointillés indiquent les références que maintiennent les outils entre eux. Chaque couche terminale gère l'affichage de sa structure graphique et ne connaît

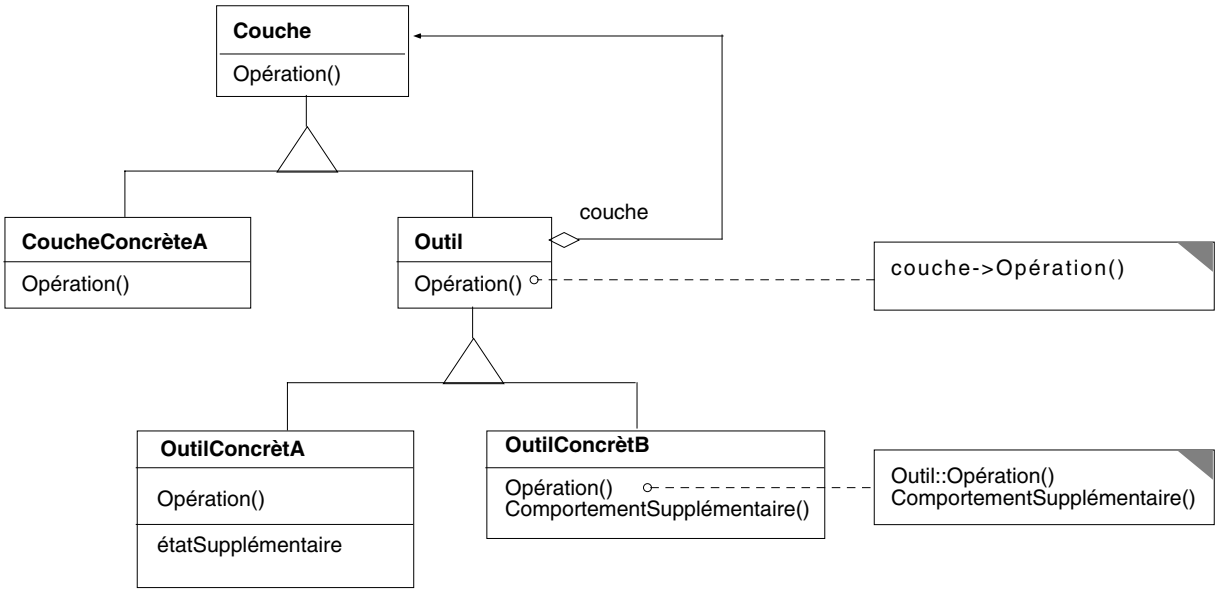


FIG. 2.3 - *Schéma OMT décrivant les relations entre la classe Couche et la classe Outil. Il s'agit d'une instance du motif decorator.*

pas les autres couches terminales, tandis que l'outil gérant la couche de fond référence l'outil gérant la couche du rectangle de sélection, qui référence l'outil de la couche de visualisation principale. Lorsque la couche de fond reçoit un événement de types « bouton de souris appuyé », elle peut en déduire qu'aucune autre couche de l'a intercepté et que l'utilisateur a cliqué sur le fond. L'outil associé à la couche de fond va alors initier la manipulation du rectangle de sélection, qui est géré par la couche du rectangle de sélection. C'est la raison pour laquelle l'outil de fond référence l'outil de gestion du rectangle.

Nous décrirons précisément la façon dont les outils gèrent la manipulation directe après avoir décrit les couches standards, en § 2.4.

2.2.2.3 Les couches terminales

Chaque couche terminale gère une forme généralisée de graphique à état. Elle implante une sémantique graphique sous la forme d'une structure de données d'objets graphiques et de fonctions de création, destruction, modification et désignation sur cette structure. Son rôle est de maintenir affichée une représentation fidèle de ses objets graphiques. La couche est aussi responsable de la gestion de la désignation et doit définir la fonction *intercepte*. Si cette désignation est coûteuse, la couche peut stocker le résultat de la dernière désignation dans une variable pour permettre son traitement par l'outil. Les outils qui veulent ignorer les événements

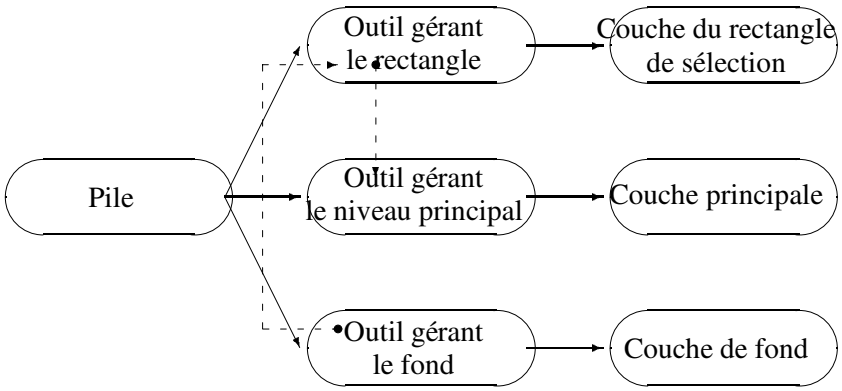


FIG. 2.4 - *Les outils dans l'architecture multicouche. Les flèches pleines relient la pile aux couches et les flêches en pointillés indiquent les références entre les outils.*

arrivant sur la couche redéfinissent la fonction *intercepte* pour retourner *faux* sans faire appel à la couche terminale.

Par exemple, si une couche est dédiée à la gestion d'un rectangle de sélection, elle peut avoir la structure suivante :

```
type CoucheRectangle = sous_classe de Couche;
début
    départ: Point;
    courant: Point;
    commencé: Booléen;

    proc commence(in Point);
    proc continue(in Point);
    proc termine(in Point);
    fn a_commencé() : Booléen;
fin;
```

L'objet graphique est simplement un rectangle tracé entre le point de départ et le point courant. Les fonctions *commence*, *continue* et *termine* permettent de modifier la structure graphique et déclenchent un réaffichage. La fonctions *a_commencé* permet de savoir si un rectangle est affiché ou non.

2.2.2.4 Changement d'outil

Un éditeur permet généralement d'utiliser plusieurs modes d'interaction. Changer de mode d'interaction consiste à changer les outils placés entre la pile et

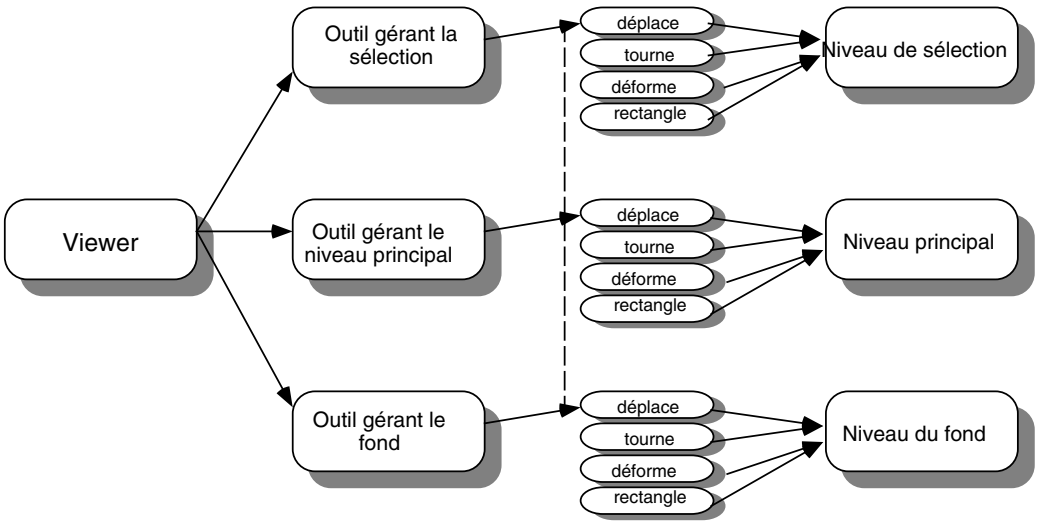


FIG. 2.5 - *Boîte d'outils permettant d'utiliser un outil parmi quatre.*

les couches terminales. Comme nous utilisons le motif *decorator*, nous pouvons dériver le type *Outil* pour définir un boîte d'outils qui gère plusieurs outils dont un seul est actif à la fois. La figure 2.5 décrit une configuration où un éditeur a besoin de quatre modes d'interaction nommés *déplace*, *tourne*, *déforme* et *rectangle*.

Pour gérer ces changements, les trois fonctions *installe_outil*, *désinstalle_outil* et *outil_installé* du type *Couche* sont nécessaires. Les deux premières sont utilisées lorsqu'un outil devient actif ou inactif et leur permet d'initialiser leur état et l'état de la couche terminale. La troisième fonction sert à savoir quel outil est actif.

2.3 Les couches standard

Les couches décrites au chapitre précédent se retrouvent dans beaucoup d'éditeurs avec une sémantique à peu près identique. Nous décrivons donc dans cette section la sémantique de chacune de ces couches, ainsi que leur degré de variabilité.

2.3.1 Le fond

La couche de fond n'a pas de sémantique particulière, elle se contente d'afficher une image qui ne varie pratiquement pas. Aucune opération supplémentaire n'est nécessaire pour le fond.

2.3.2 La visualisation des contraintes lexicales

Des contraintes lexicales s'appliquent généralement à des valeurs positionnelles produites par des dispositifs. La fonction décrivant le traitement de la couche de contraintes lexicales est :

fn *contraint*(in Événement): Événement

Les deux types de contraintes les plus fréquentes sont un alignement sur une grille et l'attraction de formes magnétiques.

2.3.2.1 Alignement sur une grille

Toutes les coordonnées des événements sont arrondies au point le plus proche d'une grille. La grille est définie par un point d'origine et un vecteur décrivant la taille de chaque maille de la grille. Notons que cette description fonctionne aussi bien pour une grille 2D que 3D et suppose seulement que les mailles soient rectangulaires (parallélépipédiques en 3D) avec les arêtes parallèles aux axes. L'interface de la couche est alors :

type CoucheCLGrille = sous_classe de Couche;
début
    origine: **Point**;
    maille: **Vecteur**;

    fn *contraint*(in Événement): Événement
fin;

La visualisation des contraintes se fait en dessinant les mailles d'une manière similaire à la réglure d'un cahier. Un problème qui se pose parfois est d'utiliser des attributs graphiques permettant de distinguer la réglure des autres objets graphiques. Suivant le modèle de dessin utilisé par les autres couches, la réglure pourra utiliser une couleur particulière ou une texture.

2.3.2.2 Attraction de trajectoires magnétiques

Une liste de trajectoires (points, segments, droites, courbes) définit des zones qui « attirent » le pointeur lorsqu'il passe à proximité. Le calcul de la contrainte n'est pas simple en général. Il consiste, pour une position initiale P_i, à chercher un point d'équilibre P telle que la somme des forces $\vec{F} = f_T(P)$ appliquées par toutes les trajectoires T en P s'annule :

$$\sum_{T \in \text{ trajectoires}} f_T(P) = \vec{0} \tag{2.1}$$

En général, la fonction $f_T(P)$ est nulle lorsque P est loin de la trajectoire ou bien sur la trajectoire. De plus, la position P ne doit pas être à une distance supérieure à $d_{\max}$ de P_i. Autrement dit, le pointeur applique une force semblable aux trajectoires et telle que $f_{\text{pointeur}}(P) = \infty$ quand $\|P - P_i\| > d_{\max}$.

La résolution de l'équation 2.1 est généralement complexe pour plus d'une trajectoire et peut avoir plusieurs solutions. Des détails peuvent être trouvés dans [Pier et al.88, Moloney et al.89], la vérification des contraintes en tant que tel sort de l'objet de cette thèse.

L'interface de la couche est :

```
type CoucheCLMagnétique = sous_classe de Couche;
début
    trajectoires: ListeDe(Trajectoire);

    fn contraint(in Événement): Événement
fin;
```

Comme pour la visualisation d'une grille, les trajectoires ne doivent pas être confondues avec les autres objets graphiques. Illustrator les affiche en pointillés, et uniquement quand le dessin est affiché en mode graphique relâché. Les autres objets sont alors affichés en traits pleins, ce qui évite toute confusion. Plusieurs éditeurs, dont Gargoyle [Pier et al.88] modifient la position du curseur lorsqu'il passe à proximité des trajectoires pour évoquer le magnétisme. Ce mode de retour peut être géré par la couche de retour d'information lexicale.

L'application des contraintes lexicales ne se fait pas pendant le parcours des événements mais doit être traitée explicitement par les outils qui le désirent, pendant l'exécution de la fonction *traite*. D'une manière générale, la gestion des contraintes lexicales ne peut pas être faite systématiquement par le mécanisme de distribution des événements.

> Par exemple, lors du déplacement d'un objet graphique à la souris, les contraintes doivent s'appliquer à l'objet graphique manipulé et pas aux positions de la souris. En effet, si le premier événement qui est produit au-dessus de l'objet graphique et qui déclenche son déplacement, était aligné sur une grille, il deviendrait impossible de sélectionner un objet plus petit que la grille et positionné au milieu d'une maille. Plusieurs autres problèmes interviennent pour le déplacement de l'objet si les positions des événements sont alignées. En réalité, seul le point

de référence de l'objet manipulé doit avoir sa position alignée sur la grille.

Pour la gestion des événements, la couche doit intercepter les événements qui sont proches de la grille ou des trajectoires afin de permettre à un outil de manipuler directement les mailles ou les trajectoires. L'éditeur de polices de caractères FontStudio [Letraset92] par exemple, permet la manipulation directe de lignes de contraintes.

2.3.3 La visualisation passive

Cette couche visualise la structure graphique principale qui devra être manipulée. La sémantique de cette couche dépend directement du domaine de l'éditeur et devra donc être spécialisée dans chaque application.

2.3.3.1 Graphique

Les modèles graphiques répandus sont :

1. 2D pixellaire,
2. 2D schématique,
3. 2D graphique,
4. 2D textuel,
5. 3D.

2D schématique, graphique et 3D : Pour le graphique structuré, la structure graphique (SG) est généralement définie en suivant le motif *Composite* (décrit page 75) avec l'interface suivante :

```
type CoucheVPStructuré = sous_classe de Couche;
début
    focus: CheminObjet;
    racine: SG;
    sélection: ListeDe(CheminObjet);

    fn objet(CheminObjet): SG;
    fn région_objet(CheminObjet): Région;
    proc endommage_objet(in CheminObjet);
fin;
```

La couche principale gère alors un arbre ou un GDA d'objets graphiques accessible par la variable *racine*. Les éditeurs ne permettent généralement que de manipuler les objets graphiques au premier niveau de la structure graphique, qui apparaît alors comme une liste d'objets graphiques dont certains sont terminaux et d'autres sont composites (on les appelle des *groupes* dans les éditeurs). Pour manipuler un objet à l'intérieur d'un groupe, un indice ne suffit pas, il faut avoir recours à une structure de données que nous avons appelée **CheminObjet** et qui décrit le cheminement à suivre pour aller de la racine à l'objet. C'est une liste d'indices, le premier est l'index du groupe dans le nœud racine, le suivant l'index du nœud suivant dans le groupe, etc. Si les objets du premier niveau du tableau sont les seuls à être manipulables, le type **CheminObjet** peut être un indice.

Nous avons omis toutes les fonctions de manipulation de la structure graphique qui sont définies par le type *SG*. Cependant, toute modification faite sur la structure graphique doit provoquer un réaffichage. La méthode la plus générale pour gérer le réaffichage est implantée par Fresco et décrite dans [Linton et al.94]. Elle consiste à pouvoir remonter d'un objet graphique à son ou ses parents pour propager toute modification. En réalité, seuls les objets mutables doivent gérer des pointeurs vers leur(s) parent(s) et les objets graphiques immutables peuvent rester très simples.

La sélection doit aussi être gérée par la couche de visualisation passive et est simplement une liste de *CheminObjet*. La fonction *région_objet* retourne la région où est placé un objet, et la fonction *endommage_objet* déclare cette région comme endommagée.

Le texte La gestion graphique du texte est semblable à la gestion du graphique structuré en 2D. Un caractère est un objet graphique, et le texte s'organise en groupes, ce qui nécessite un **CheminObjet** pour indiquer la position d'un caractère. Les éditeurs de texte non structurés décrivent leur contenu comme un tableau de caractères et peuvent utiliser un indice pour représenter chaque caractère édité.

À la différence des éditeurs graphiques, la sélection est généralement décrite comme un intervalle entre deux caractères et non comme une liste de **CheminObjet**. Le *focus* s'appelle généralement le point d'insertion ou la position du curseur de texte.

2D pixellaire L'interface de la couche de visualisation passive pour éditer du 2D pixellaire est la suivante :

```
type CoucheVPPixellaire = sous_classe de Couche;
début
    image: Image;
    sélection: ListeDe(Région);
fin;
```

La variable *image* contient la table de pixels. Un groupe de pixels est représenté par le type **Région**. Les fonctions de manipulation de l'image sont appliquées à la sélection ou à toute l'image. Pour déclencher le réaffichage, la région modifiée est endommagée.

2.3.3.2 Gestion de la désignation

Deux types d'événements doivent être gérés : positionnels et non positionnels.

Événements positionnels : La gestion des événements positionnels consiste généralement à trouver l'objet graphique placé à la position indiquée pour agir dessus, par exemple détruire l'objet, le couper, ou le sélectionner. Dans le cas du graphique pixellaire, il s'agit des coordonnées du pixel désigné dans l'image.

Événements non positionnels : La gestion d'événements non positionnels requiert un *focus*, c'est-à-dire un moyen de déterminer à quel(s) objet(s) graphique(s) l'événement s'adresse. Les éditeurs de texte utilisent un *curseur de texte* pour visualiser l'emplacement courant de l'édition pour les événements produits par les touches du clavier. Dans notre architecture, la visualisation du curseur est faite par la couche de retour d'information lexicale.

La boîte à outils Motif [Ferguson et al.93] visualise aussi le focus sur tous les objets graphiques.

2.3.4 La sélection

Lorsque la couche de sélection est utilisée pour visualiser la sélection, elle gère une liste d'objets graphiques particuliers, appelés « poignées » de sélection et dont l'interface est :

```
type Poignée = classe;
début
    chemin: CheminObjet;
    couche: Couche;

    proc dessine(inout Canvas,in Région);
    fn intercepte(Événement): Booléen;
    proc traite(in Événement);
fin;
```

Chaque poignée connaît la couche à laquelle appartient l'objet dont elle visualise la sélection, et le chemin vers l'objet. Elle a un rôle que nous décrivons plus loin.

L'interface de la couche est la suivante :

```
type CoucheSélection = sous_classe de Couche;
début
    poignées: ListeDe(Poignée);
    intercepteur: Poignée;
fin;
```

2.3.4.1 Graphique

Les poignées sont généralement carrées dans les éditeurs structurés. C'est le cas dans Illustrator, Canvas, Idraw, etc. Les éditeurs d'images pixellaires comme PhotoShop affichent la sélection sous forme de polygones tracés en pointillés et animés.

Dans certains cas, une forme différente est utilisée suivant le type d'objet graphique sélectionné. Par exemple, Unidraw différencie la sélection d'un objet terminal et la sélection d'un groupe d'objets. Dans le premier cas, la représentation interne de l'objet terminal sert de support à la représentation de sa sélection tandis que dans le second cas, seul le rectangle englobant sert de support et une poignée est affichée à chaque sommet.

Notons que dans tous les cas, les objets qui visualisent la sélection sont généralement moins complexes à dessiner que les objets sélectionnés.

2.3.4.2 Gestion de la désignation

Les poignées servent à déclencher la manipulation directe. Chaque poignée ne déclenche pas toujours le même type de manipulation.

> Par exemple, dans Canvas, lorsqu'un objet est sélectionné, huit poignées sont affichées par rapport aux rectangles englobant l'objet : quatre dans les coins du rectangle et quatre au milieu des arrêtes. Lorsque l'utilisateur « tire » sur un coin, l'objet est retaillé en gardant ses proportions. Lorsqu'il tire sur une poignée au milieu d'une arrête, l'objet est retaillé dans une dimension et pas dans l'autre. Lorsqu'il tire au milieu du rectangle, alors l'objet est déplacé.
>
> La création des poignées est faite au moment de la commande de sélection. La position de chaque poignée peut être calculée à partir de la région occupée par l'objet sélectionné dans sa couche (la fonction *région_objet* de la couche de visualisation passive).

La fonction *intercepte* de la couche retourne *vrai* si une des poignées intercepte l'événement. La poignée est placée dans la variable *intercepteur* qui pourra être consultée par l'outil qui gère la manipulation. Suivant le rôle de la poignée, l'outil pourra choisir un mode de manipulation ou un autre.

2.3.5 La manipulation directe

La gestion de la manipulation directe appartient à une couche particulière qui possède d'autres objets graphiques que la couche de visualisation passive et que la couche de sélection. En effet, le mode de représentation des objets pendant leur manipulation est généralement différent de la représentation des seules poignées de sélection et souvent moins complexe que la forme utilisée dans la couche de visualisation passive.

De même, pendant la manipulation, les objets graphiques ont une sémantique particulière qui peut être très sophistiquée et nécessiter une gestion spécifique, comme celle décrite en troisième partie, § 1.

Cette couche est aussi responsable, en général, de la gestion de la manipulation directe lors de la création d'un objet graphique. Dans ce cas, l'objet graphique manipulé n'a pas encore de représentant dans la couche de visualisation passive. L'action déclenchée à la fin de la manipulation directe est de créer un nouvel objet qui apparaît alors dans la couche de visualisation passive.

Les objets graphiques manipulés par cette couche sont des représentants d'objets existants ou allant exister dans la couche de visualisation passive que nous appelons « fantômes » et dont l'interface est décrite avec la couche de manipulation directe :

```
type Fantôme = classe;
début
        chemin: CheminObjet;
        couche: Couche;

        fn intercepte(Événement): Booléen;
        proc dessine(inout Canvas,in Région);
        proc commence(in Événement);
        proc continue(in Événement);
        proc termine(in Événement);
        fn a_commencé() : Booléen;
fin;

type CoucheMD = sous_classe de Couche;
début
        fantômes: ListeDe(Fantôme);
```

```
    intercepteur: Fantôme;

    proc dessine(inout Canvas,in Région);
    proc commence(in Événement);
    proc continue(in Événement);
    proc termine(in Événement);
    fn a_commencé() : Booléen;
fin;
```

Notons que les fantômes ont une durée de vie relativement courte.

2.3.5.1 Graphique

Pendant la manipulation, les objets graphiques affichés ont une forme qui est assez proche de l'objet graphique de la couche de visualisation passive. Certains éditeurs requièrent un affichage précis des objets graphiques tandis que d'autres se contentent d'une silhouette.

Tang décrit les *Pacers* [Tang et al.93], qui gèrent la manipulation directe et adaptent la qualité de leur affichage à la vitesse à laquelle on les manipule. Dans notre architecture, ces objets sont les fantômes de notre couche de manipulation directe et non des objets graphiques de la couche de visualisation passive.

En utilisant notre architecture, les fantômes peuvent même être spécialisés en fonction de la manipulation qui leur est appliquée. Tandis que les *Pacers* doivent déduire de leur vitesse de réaffichage des compromis de qualité, nous pouvons créer un fantôme particulier pour gérer la translation, la rotation, le changement d'échelle, etc. Le fantôme de la translation peut décider de créer une image de son contenu et de déplacer cette image (c'est alors un *sprite*) si son temps de rafraîchissement est trop long.

Dans des cas plus compliqués où un objet peut être transformé pendant sa manipulation directe, les *Pacers* sont utiles mais n'ont pas besoin, dans notre cas, d'être liés aux objets de la visualisation passive.

Notre modèle est aussi particulièrement intéressant lorsque la gestion de la manipulation directe est coûteuse. Une sémantique relâchée peut alors être choisie pendant le temps de la manipulation.

> Par exemple, en troisième partie, § 1, nous décrivons un système de représentation et de placement de graphe quelconque qui peut prendre beaucoup de temps pour calculer un placement de graphe qui soit stable. La couche de gestion de la manipulation directe peut utiliser tous les moyens pour optimiser l'affichage et le calcul interactif, en particulier afficher de façon simple et rapide tous les objets graphiques

et forcer un nœud du graphe à suivre le pointeur de façon synchrone mais calculer et mettre à jour le reste du graphe de façon asynchrone.

2.3.5.2 Gestion des événements

En général, lorsqu'un mode d'interaction utilise cette couche, il le fait en utilisant un outil qui intercepte tous les événements lorsque la couche n'est pas vide. La couche doit cependant implanter la fonction *intercepte* pour être conforme à ses attributions.

2.3.6 Le lasso de sélection

Cette couche permet de dessiner une forme géométrique qui, lorsqu'elle est terminée, déclenche la sélection des objets de la couche de visualisation passive placés en dessous. Sa version la plus simple est le rectangle de sélection, décrit en § 2.2.2.3. Cette définition est facile à généraliser à la gestion d'une région quelconque par tracé, souvent utilisée dans les éditeurs pixellaires (l'outil est parfois appelé « lasso »).

```
type CoucheLasso = sous_classe de Couche;
début
    régions: Région;

    proc dessine(inout Canvas,in Région);
    proc commence(in Point);
    proc continue(in Point);
    proc termine(in Point);
    fn a_commencé() : Booléen;
fin;
```

2.3.6.1 Graphique

Cette couche est utilisée pour afficher un rectangle ou une forme graphique pendant qu'elle se construit. En général, cette forme a une durée de vie courte et est simple à afficher. La forme à afficher ne requiert qu'un modèle graphique élémentaire, permettant d'afficher un polygone d'un pixel d'épaisseur.

2.3.6.2 Gestion des événements

Comme décrit en § 2.2.2, cette couche est généralement activée par une action extérieure. Elle intercepte alors les événements jusqu'à ce que l'action soit terminée.

2.3.7 Le retour d'information lexicale

Cette couche visualise des informations sans caractère sémantique qui se superposent aux autres objets graphiques. Son interface est la suivante :

```
type CoucheRIL = sous_classe de Couche;
début
    formes: ListeDe(FormeSimple);

    proc déplace(in Entier, in Point);
fin;
```

2.3.7.1 Graphique

Cette couche permet de visualiser :

- la position des dispositifs de pointage auxiliaires,
- la trace laissée par des dispositifs,
- l'écho d'actions se produisant dans d'autres couches ou dans d'autres éditeurs.

Écho des dispositifs Lorsqu'un dispositif auxiliaire n'est pas géré par le système de fenêtrage, cette couche doit en afficher le curseur, qui est défini par une image et son masque.

Certains éditeurs, en particulier en CAO, n'affichent pas de curseur mais deux lignes perpendiculaires qui se croisent à la position du curseur. La position est plus précise et peut être alignée avec d'autres positions visualisées. Il peut être utile de visualiser plusieurs dimensions avec le curseur.

> Par exemple, le curseur du stylet d'une tablette à numériser, sensible à la pression, peut être visualisé à l'aide d'un point central et d'un cercle dont le rayon varie avec la pression.

Enfin, la position des dispositifs est parfois écrite explicitement sur un bord de la couche afin d'en connaître la valeur numérique avec précision.

Trace Certaines applications graphiques utilisent un paradigme de « trace » ou d'« encre ». Un dispositif positionnel, lorsqu'il est dans un certain état, laisse alors une trace graphique pendant qu'il est manipulé. C'est le cas de certains systèmes de reconnaissance de gestes et d'éditeurs graphique permettant de dessiner à main levée. Ces applications peuvent gérer l'affichage de la trace dans cette couche (voir troisième partie, § 2.3.6).

Écho d'autres phénomènes Certains phénomènes déclenchés indirectement par manipulation ou déclenchés de façon extérieure peuvent aussi être représentés dans cette couche.

Deux exemples permettent de donner une idée de ces phénomènes :

l'animation d'un effet graphique lorsqu'on active une icone dans le *Finder* du Macintosh, une fenêtre doit apparaître. Cette apparition est précédée d'une animation évoquant l'agrandissement de l'icone ;

l'écho d'actions dans un collecticiel lors de la manipulation d'un collecticiel, les actions faites à distance par les autres utilisateurs sont parfois exprimées à l'aide d'animations sur cette couche. Ainsi, chaque utilisateur peut se rendre compte de l'activité des autres [Karsenty94].

2.3.7.2 Gestion des événements

Écho des dispositifs Pour l'écho des dispositifs, cette couche ne bloque pas les événements, elle se contente de répercuter leur état. La couche peut utiliser la fonction *intercepte* pour mettre à jour la position de ses objets graphiques.

Gestion de la trace Les applications qui gèrent une trace ne sont généralement pas intéressées par chaque événement ayant servi à créer cette trace, mais plutôt pas l'ensemble de la trace une fois celle-ci terminée.

Pendant la phase de création de la trace, tous les événements sont donc interceptés par la couche d'écho des dispositifs et la trace est dessinée. Une fois la trace terminée, la liste des points décrivant sa trajectoire peut être utilisée par l'outil actif pour interprétation. Cette trace peut ensuite servir à plusieurs fonctions : la reconnaissance de gestes, le détourage d'une région, la sélection d'objets, etc.

2.4 Les outils

Un mode d'interaction est décrit par un ensemble d'outils. À chaque couche est associé un outil qui dépend du mode et qui traite les événements arrivant à la couche. Par exemple, pour implanter le mode d'interaction qui permet, en cliquant avec un pointeur, de sélectionner un objet graphique affiché dans la couche de visualisation passive, seule la couche de visualisation passive nécessite un outil actif. Toutes les autres couches ignorent les événements, ce qui est fait en leur associant un outil qui définit sa fonction *intercepte* pour retourner toujours la valeur *faux*. L'outil lié à la couche de visualisation passive, quand à lui, définit la fonction

intercepte comme appelant la fonction *intercepte* de la couche. Lorsqu'un événement arrive sur la pile, il ne peut être intercepté que par la couche de visualisation principale à travers son outil. Lorsque la fonction *traite* de l'outil est appelée, cela signifie que la couche de visualisation principale a intercepté l'événement et la variable *focus* de la couche de visualisation passive référence l'objet graphique qui a intercepté l'événement et qui sera sélectionné lors du traitement de l'événement. Pour que seul l'événement « bouton appuyé » provoque la sélection, l'outil ne doit appeler la fonction intercepte de sa couche que lorsque l'événement est « bouton appuyé ».

Pour implanter à la fois la sélection par désignation directe et par un rectangle de sélection, plusieurs couches doivent collaborer, comme décrit en 2.2.2.2.

2.4.1 Notation UAN

Une notation souvent utilisée pour décrire l'interaction à un bas niveau est UAN [Hartson et al.90], décrit en annexe B. La gestion d'événements dans l'architecture multicouche s'exprime naturellement à partir d'UAN. Nous avons ajouté deux éléments à la notation : la couche et un contexte. Ce dernier aurait pu être exprimé à l'aide d'une précondition de UAN mais nous avons préféré l'exprimer opérationnellement dans notre architecture sous la forme du résultat d'un appel de fonction. Dans le schéma UAN II.1, le nom de la couche est placé en colonne de gauche, ensuite vient la précondition (qui est toujours vide dans cet exemple), puis vient l'événement UAN. Contrairement à UAN, nous ne distinguons pas les changements de l'interface et les calculs effectués. Nous préférons distinguer les actions locales à la couche et les actions transmises car nous ne nous plaçons pas au niveau de la description de la tâche mais de celle des actions des couches.

Comme les actions locales et transmises ne sont pas spécifiées mais juste décrites, en particulier à partir des opérations spécifiques des couches, nous ne prétendons pas aller au-delà d'une notation proche de l'implantation.

Un aspect particulier à notre notation est l'apparition des couches en colonne de gauche. Les couches sont décrites de la plus profonde dans la pile (en haut) vers la plus proche de l'utilisateur (en bas). Ainsi, le nom de la couche signifie : toutes les couches précédentes (placées au-dessus et décrites dans les lignes qui suivent) ont reçu l'événement et aucune ne l'a intercepté.

2.4.2 Outil de sélection

Dans sa version la plus simple, l'outil de sélection a le comportement décrit en figure II.1. Il s'interprète comme suit : lorsque l'événement « bouton de souris appuyé » arrive à la couche VP sur un objet *o*, la fonction locale à la couche *select* est appelée avec l'objet *o* en argument. Elle déclare l'objet *o* sélectionné ; c'est

l'action locale. La fonction *poignées* de la couche de sélection est ensuite appelée avec l'objet o en argument. Elle crée les poignées de visualisation de la sélection sur cette couche ; c'est l'action transmise.

Tâche II.1 Sélection simple

Couche	Contexte	Événement	Action locale	Action transmise
VP		$\sim [o]\mathrm{M} \downarrow$	select(o)	**Sélection** poignées(o)

Pour implanter un outil de sélection plus sophistiqué qui gère aussi un rectangle de sélection, d'autres couches doivent coopérer, comme le décrit la figure II.2.

Lorsque l'événement « bouton de souris appuyé » arrive sur la couche de fond, cela signifie qu'il n'a pas été intercepté par la couche de visualisation passive et qu'aucun objet n'est placé en dessous. La sélection de la couche de visualisation passive est alors annulée et la manipulation du rectangle de sélection est initiée.

La couche de gestion du rectangle n'intercepte les événements que lorsqu'elle a commencé la manipulation. C'est ce qu'exprime la colonne de contexte de cette couche. Elle gère alors le déplacement de la souris et l'événement « bouton de souris relâché », qui déclenche la sélection des objets sous le rectangle.

Tâche II.2 Sélection avec rectangle

Couche	Contexte	Événement	Action locale	Action transmise
Fond		$\sim [x, y]\mathrm{M} \downarrow$		**VP** désélectionne_tout() **Sélection** détruit_poignées() **Rectangle** commence(x, y)
VP		$\sim [o]\mathrm{M} \downarrow$	select(o)	**Sélection** poignées(o)
Rectangle	*a_commencé()*	$\sim [x', y']*$	continue(x', y')	
	a_commencé()	$\sim [x'', y'']\mathrm{M} \uparrow$	termine(x'', y'')	**VP** select(rectangle()) **Sélection** poignées(rectangle())

Les programmes graphiques comme Canvas ou Illustrator gèrent à la fois la sélection et le déplacement d'objets. La gestion de cette interaction est décrite en

figure II.3. La couche de sélection gère l'événement « bouton de souris appuyé » lorsqu'il arrive sur une poignée. Il détruit alors les poignées et initie la manipulation sur la couche de manipulation directe. Nous n'avons pas exprimé la création des fantômes qui est faite par la fonction *commence* de l'outil de déplacement de la couche de manipulation directe.

Tâche II.3 Sélection complète

Couche	Contexte	Événement	Action locale	Action transmise
Fond		$\sim [x, y]\mathrm{M} \downarrow$		**VP** désélectionne_tout() **Sélection** détruit_poignées() **Rectangle** commence(x, y)
VP		$\sim [o]\mathrm{M} \downarrow$	select(o)	**Sélection** poignées(o)
Sélection		$\sim [o]\mathrm{M} \downarrow$	détruit_poignées()	**MD** commence(x, y)
MD	*a_commencé()*	$\sim [x', y']*$	continue(x', y')	
	a_commencé()	$\sim [x'', y'']\mathrm{M} \uparrow$	termine(x'', y'')	**VP** déplace(dx, dy)
Rectangle	*a_commencé()*	$\sim [x', y']*$	continue(x', y')	
	a_commencé()	$\sim [x'', y'']\mathrm{M} \uparrow$	termine(x'', y'')	**VP** select(rectangle()) **Sélection** poignées(rectangle())

Comme on peut le voir dans cet exemple, l'organisation de l'interface en couches permet de raffiner les actions de la manipulation directe d'une façon simple et modulaire. L'ajout de fonctionnalités nouvelles s'est fait sans changer le comportement des couches déjà présentes. Cette caractéristique n'est pas particulière à cet exemple, il est très général et nous a donné beaucoup de flexibilité dans des applications réelles.

2.4.3 Analyse critique

Contrairement aux interacteurs de Garnet et Unidraw, notre architecture permet de décrire la manipulation directe de manière très déclarative et non uniquement procédurale. Le fait qu'un outil ne requiert la collaboration que de quelques

couches évite la centralisation de la description du contrôle qui nuit à sa compréhension, son extensibilité et sa robustesse.

Nous donnerons d'autres exemples de mode d'interaction dans la troisième partie.

2.5 Spécialisations du modèle graphique sur la couche

Après avoir décrit les couches standard et les outils, nous pouvons revenir sur les caractéristiques graphiques des couches et décrire les mécanismes de spécialisation du modèle graphique au niveau de la couche.

Comme nous l'avons vu, certaines couches peuvent nécessiter un modèle graphique différent de celui de la surface virtuelle du système de fenêtrage qui est passé en argument de la fonction *dessine* de la pile et des couches.

> Par exemple, la couche de visualisation passive peut gérer une structure d'objets graphiques 3D et vouloir la visualiser avec un modèle graphique 3D réaliste. La manipulation directe peut alors vouloir visualiser les fantômes des objets graphiques 3D avec un modèle graphique relâché.

Chaque couche doit utiliser un modèle dérivé de ceux que nous avons vus en première partie, § 2.5. Il est peu probable qu'un système de fenêtrage implante un modèle graphique unifié dans un avenir proche, l'approche étant plutôt à l'utilisation d'extensions pour chaque modèle graphique particulier. Par ailleurs, certains modèles comme GKS ou PHIGS continuent d'exister pour des raisons historiques et ne sont pas près de disparaître.

Cependant, le modèle graphique de la surface virtuelle que manipule la pile dépend du système de fenêtrage, il n'est pas modifiable. Si une couche désire un modèle graphique particulier, elle peut l'implanter de plusieurs manières mais elle doit, à la fin, se composer sur la surface virtuelle de la pile. Trois stratégies sont envisageables pour passer d'un modèle graphique à un autre :

- la traduction de modèle,
- l'utilisation d'extensions,
- le calcul par logiciel.

2.5.1 Traduction de modèle

Lorsque les primitives et les attributs du modèle graphique de la couche peuvent être exprimés avec les primitives et attributs du modèle graphique géré par

le système de fenêtrage, alors il est possible d'utiliser le mécanisme de dérivation des langages à objets pour spécialiser le modèle graphique.

> Par exemple, le modèle graphique du système X11 ne permet pas de tracer des objets graphiques décrits par des segments de courbes de Bézier [Bézier70]. Néanmoins, il est possible de convertir des segments de courbes de Bézier en segments de droites qui sont gérés par X11.

> Lorsque le système de fenêtrage considère l'attribut graphique « épaisseur de trait » comme décoratif, il est possible — par dérivation — de rendre cet attribut géométrique. Il suffit, lors de chaque opération de dessin, de multiplier l'épaisseur du trait par le facteur d'échelle appliqué au dessin [1].

La traduction de modèle est aussi utilisée pour implanter un modèle graphique relâché. Dans ce cas, certains attributs graphiques peuvent être ignorés ou remplacés par des attributs plus rapide à gérer.

> Par exemple, si des objets complexes doivent être redessinés pendant la manipulation directe, il est intéressant de définir une surface virtuelle offrant une interface identique à celle utilisée pour l'affichage sur la couche de visualisation passive, mais qui ne fasse que tracer les contours des formes des objets. La fonction de tracé des objets de la surface virtuelle peut alors être utilisée telle qu'elle.

2.5.2 Utilisation d'extensions

Il est formellement toujours possible de passer par la traduction de modèle graphique ; dans le pire des cas, il suffit de n'utiliser que la fonction de tracé de points. Il arrive cependant que le coût de cette traduction soit élevé. Pour résoudre ce problème, plusieurs systèmes de fenêtrage proposent des extensions qui donnent accès à d'autres modèles graphiques. Le problème qui se pose alors est de faire cohabiter le modèle graphique de l'extension et celui du système de fenêtrage.

Les extensions ont généralement trois modes de fonctionnement :

1. une (sous) fenêtre doit être allouée spécifiquement dans laquelle il est possible d'utiliser le modèle graphique de l'extension ;

1. Notons que cette façon de procéder ne fonctionne correctement que lorsque le facteur d'échelle est uniforme sur les axes.

2. l'extension permet que son modèle graphique et celui du système de fenêtrage soient utilisés sur la même fenêtre ;

3. l'extension permet que son modèle graphique soit utilisé sur une portion de mémoire accessible rapidement par le système de fenêtrage.

Avec l'architecture multicouche, il est possible d'utiliser le mode 2 à condition que la sémantique de composition soit bien définie par l'extension, et le mode 3 à condition que l'extension puisse gérer un plan alpha ou un masque pour décrire les zones de pixels qui ont été modifiés. Le mode 1 est incompatible avec notre architecture, mais nous ne connaissons pas d'extension graphique standard limitée au type 1. En revanche, certaines extensions de type 3 imposent que la mémoire partagée soit dans la carte vidéo, ce qui en limite la taille et le nombre.

2.5.2.1 Partage de la fenêtre

Dans le cas 2, nous appelons « sémantique bien définie » une sémantique de gestion du tracé qui garantit l'achèvement du tracé à un moment bien défini et qui ne modifie que les pixels nécessaires à l'affichage d'une structure de données, ce qui est généralement le cas. Les extensions qui s'appuient sur des accélérateurs matériels utilisent des appels de fonctions graphiques asynchrones, il est donc nécessaire de s'assurer de la fin de leur exécution pour terminer l'étape d'affichage ou de réaffichage de la couche. Dans certains cas, le délai de réaffichage peut être tel que seul le mode 3 peut être utilisé.

> L'extension de gestion d'image de X (XIE [Rogers94]) utilise un modèle à flot de données géré par le serveur X, dans lequel des sources sont connectées à des opérateurs qui sont à leur tour connectés à d'autres opérateurs ou à des sorties. Le traitement des images passe par le réseau d'opérateurs construit dans le serveur pour qu'une image résultante soit calculée. Le traitement est totalement asynchrone et XIE définit des procédures *Callback* qui permettent à l'application d'être notifiée lorsque des étapes sont effectuées. Le temps nécessaire pour l'accomplissement de certains traitements est tel qu'un programme qui gérerait l'affichage de façon synchrone serait inutilisable.

2.5.2.2 Utilisation d'image partagée

Dans la plupart des cas, les extensions peuvent calculer les images dans une portion de mémoire accessible aux primitives graphiques du système de fenêtrage. L'étape de calcul d'images sur la couche est donc faite en deux temps : l'image est

d'abord calculée en mémoire puis est recopiée sur la surface virtuelle gérée par le système de fenêtrage.

Lorsque le tracé d'images peut être fait de façon synchrone, alors le mécanisme est équivalent au mode 2 avec une image mémoire en plus. Cette image mémoire peut alors être aussi utilisée comme cache pour le réaffichage.

Lorsque le tracé d'images est asynchrone, l'image mémoire est mise à jour de façon asynchrone et la couche peut se redessiner de façon asynchrone. Pour garantir des performances acceptables, il est alors important de s'assurer que le redessin potentiellement fréquent de cette couche n'entraînera pas de réaffichage coûteux des couches placées au-dessus, en utilisant des techniques d'optimisation décrites en § 2.6.

2.5.3 Calcul par logiciel

Dans certains cas, le modèle graphique que l'on désire sur la couche est tellement spécifique qu'il doit être implanté en logiciel. Nous décrivons un tel cas en troisième partie, § 2.6. L'affichage passe alors par l'utilisation d'une image partagée entre l'application et le système de fenêtrage (des mécanismes existent toujours à cet effet). Le tracé est donc fait en mémoire puis reporté sur la surface virtuelle du système de fenêtrage, comme dans la section précédente.

2.6 Optimisations du réaffichage

Dans la section 2.1.3, nous avons indiqué que des optimisations de réaffichage étaient parfois indispensables pour qu'un éditeur réponde assez rapidement à la manipulation directe.

L'optimisation du réaffichage peut se faire à trois niveaux :

- au sein d'une couche,
- au niveau d'une couche dans la pile,
- entre plusieurs couches dans la pile.

Dans tous les cas, lorsqu'une image n'a jamais été affichée, elle doit être calculée au moins une fois. Nous décrivons ici des techniques d'optimisation qui consistent à éviter les recalculs d'images lorsque c'est possible.

2.6.1 Distinction entre affichage et réaffichage

Lorsque le modèle d'affichage et de réaffichage est unifié, la surface virtuelle gère une structure de donnée « région » décrivant les zones qui doivent être réaffi-

chées et qui sont invalides. L'algorithme de réaffichage le plus simple fonctionne comme suit :

```
proc redessine(s: Canvas, o: ObjetGraphique);
début
    s.limite_dessin(s.dommage());
    o.dessine(s, s.région_totale);
    s.finis_limite();
    s.pas_de_dommage();
fin;

proc Pile.dessine(s: Canvas, r: Région);
début
    si r.intersecte(s.limite()) alors
      pour c dans couches fait
          c.dessine(s, r);
fin;
```

La fonction *limite_dessin* empêche toute modification de la surface virtuelle en dehors de la région qui lui est passée en paramètre et la fonction *finis_limite* termine cette limitation. La fonction *redessine* commence donc par limiter la portée du réaffichage à la région endommagée, ceci pour éviter qu'un objet graphique devant se réafficher partiellement ne modifie le reste de la surface virtuelle en se redessinant.

> Par exemple, si la région à redessiner n'est recouverte que par le fond, celui-ci va se redessiner sur toute la surface virtuelle, effaçant les autres objets si la région de dessin n'est pas limitée.

Ensuite, l'objet graphique racine passé à la fonction de réaffichage est dessiné et la surface virtuelle n'a plus de région endommagée. La pile se trouve dans l'arbre des objets graphiques et sa fonction *dessine* est alors appelée. Elle commence par tester si sa région intersecte la région endommagée pour éviter du travail inutile à la surface virtuelle. Si c'est le cas, toutes les couches sont dessinées, en commençant par le fond.

Pour éviter de recalculer une image, il faut utiliser le fait que la surface graphique d'un système de fenêtrage garde généralement son contenu entre deux réaffichages. Si l'état de la surface virtuelle est connu à l'entrée de la fonction *dessine*, sa mise à jour peut parfois se faire sans tout redessiner. La mise à jour peut consister à dessiner par-dessus la surface virtuelle ou à effacer une partie de la surface virtuelle.

La surface virtuelle, lorsqu'elle est liée à une fenêtre, peut avoir une région invalide à cause d'autres fenêtres ou de phénomènes issus du système de fenêtrage. Pour gérer cette éventualité, il faut que la surface virtuelle gère deux régions : la région de dommage (accessible avec la fonction *dommage*) et la région invalide (accessible avec la fonction *invalide*). L'algorithme de réaffichage devient alors :

```
proc redessine(s: Canvas, o: ObjetGraphique);
début
    var total: Région;

    total := union(s.invalide(),s.dommage());
    s.limite_dessin(total);
    o.redessine(s, s.région_totale);
    s.recopie_buffer(total); s.finis_limite();
    s.pas_de_dommage();
fin;
```

2.6.2 *Double buffering*

Lorsque du *double buffering* introduit en première partie, § 3.1.2.5 est utilisé, la surface virtuelle a une région invalide presque toujours vide. Le *double buffering* utilisé dans les interfaces est un peu différent de celui utilisé par l'animation car toute l'image n'est pas recalculée à chaque affichage. Le mécanisme consiste à utiliser, en plus de la fenêtre, une image en mémoire de la taille de la fenêtre (appelée *back buffer*). Le réaffichage se fait toujours dans le *back buffer*, puis la fonction *recopie_buffer* copie la région sur la fenêtre. L'algorithme fonctionne aussi bien avec que sans le *double buffering*. Dans ce dernier cas, la fonction *recopie_buffer* ne fait rien. Un bénéfice supplémentaire du mécanisme apparaît lorsqu'une région de fenêtre devient visible à l'écran. Sa mise à jour se fait simplement en copiant la région du *back buffer* sur la fenêtre et ne déclenche pas de réaffichage de la structure graphique sur la surface virtuelle.

La région invalide n'est cependant pas toujours vide. Lors de l'affichage initial, il contient la région entière de la surface virtuelle, et lors du retaillage de la fenêtre, il décrit une région qui dépend de la sémantique du retaillage : lorsqu'on peut les déterminer, ce sont les régions nouvellement visibles, sinon, c'est la région de la fenêtre entière.

Nous allons maintenant voir comment les couches et la pile peuvent tirer profit de la distinction entre région endommagée et région invalide.

2.6.3 Optimisations au sein d'une couche

Les optimisations au sein d'une couche dépendent avant tout du modèle graphique utilisé. En-dessous d'une certaine complexité de modèle graphique ou de nombre d'objet graphique, toute optimisation est inutile. La puissance des machines actuelles rend la limite de plus en plus élevée. Il existe cependant quelques cas où les optimisations restent aujourd'hui indispensables : le graphique 3D en temps réel et le graphique 2D lourd, c'est-à-dire utilisant beaucoup d'objets, des objets de grande taille, ou des attributs graphiques coûteux au calcul.

Les optimisations de calcul débordent de notre sujet et sont l'un des sujets principal des conférences comme EUROGRAPHICS et SIGGRAPH. La plupart des optimisations de réaffichage du 2D pixellaire est décrite dans [Shantsis94]. Pour le 2D vectoriel, nous décrivons une méthode qui permet d'optimiser le réaffichage dans certains cas. [Gosling81] décrit des optimisations de réaffichage du texte basées sur la programmation dynamique ; la plupart de ces optimisations est plutôt rentable sur des terminaux alphanumériques. Pour le graphique 3D réaliste, seules des extensions matérielles permettent une vraie accélération au réaffichage. Le réaffichage 3D schématique peut être accéléré à partir d'arbres binaires de partitionnement de l'espace (*BSP trees*) qui autorisent des optimisations décrites dans [Chin95].

2.6.3.1 Optimisation du réaffichage en 2D vectoriel

Lorsqu'une scène 2D vectorielle est affichée sur une surface virtuelle, et que des objets graphiques sont ensuite rajoutés à la fin de la liste d'affichage, le réaffichage peut se contenter d'afficher les nouveaux objets à condition que la surface virtuelle n'ait pas été modifiée entre temps. Les trois conditions de l'optimisation sont :

1. la scène a déjà été affichée,
2. de nouveaux objets ont été ajoutés uniquement au-dessus des autres,
3. la scène n'a pas été modifiée lorsque le réaffichage est déclenché.

Cette optimisation est particulièrement intéressante lorsque la surface graphique garde son contenu dans un cache ; la condition (3) est alors toujours vérifiée. Cette optimisation est intéressante sur une couche contenant beaucoup d'objets graphiques 2D ou des objets coûteux à dessiner, et où de nouveaux objets sont fréquemment ajoutés, ce qui est le cas d'éditeurs permettant le dessin à main levée comme celui décrit en troisième partie, § 2. Lorsqu'elle n'utilise pas de cache, cette optimisation doit être utilisée en tenant compte de la région invalide de la surface virtuelle qui a toujours besoin d'être redessinée.

Le cache doit contenir non seulement l'image mais aussi le masque des pixels modifiés par la couche. Si le modèle graphique de la surface virtuelle le permet, le cache doit contenir une image avec un plan alpha [Porter et al.84] ou, plus simplement, un masque de bits décrivant les pixels modifiés par la couche. Le modèle graphique de la surface virtuelle doit pouvoir composer une image avec un plan alpha ou ne toucher qu'aux pixels indiqués par le masque, ce que tous les systèmes de fenêtrage actuels savent faire.

2.6.4 Optimisations du réaffichage d'une couche dans la pile

La pile peut parfois optimiser le réaffichage d'une couche si elle connaît des caractéristiques de cette couche. C'est la raison pour laquelle deux mécanismes sont gérés par les couches : les attributs de couche et la région endommagée, retournée par la fonction *dommage* de la couche.

Nous avons répertorié les attributs suivants :

- normal,
- *involutive*,
- *transitoire*,
- *cache*,
- *sauve-dessous*.

involutive : Une couche *involutive* garantit que tous les objets graphiques qui la composent, utilisent une fonction de composition involutive, c'est-à-dire que réafficher un objet deux fois produit son effacement. Dans les modèles graphiques courants, les opérateurs involutifs sont *xor* et *inverse*.

> Par exemple, si l'unique couche endommagée est involutive et placée au dessus de toutes les autres, la pile peut l'effacer avec la fonction *efface* et la redessiner avec la fonction *dessine* sans avoir besoin de dessiner les autres couches.

Cette optimisation nécessite que la fonction *efface* soit capable de réafficher (c'est-à-dire d'effacer) le contenu effectivement présent sur la surface virtuelle. Parmi les couches décrites en 2.3, la couche de sélection contient des objets qui ne bougent pas (les poignées) et peuvent donc être effacés avec la fonction *dessine*. Les autres couches — comme la gestion de la manipulation directe ou la gestion du rectangle de sélection — sont modifiées à chaque action. Pour que la fonction *efface* fonctionne, la couche doit garder l'état de la liste d'affichage du dernier dessin, avec généralement une pénalité en occupation mémoire.

Pour pouvoir utiliser convenablement une fonction de composition involutive, il est indispensable de maîtriser le contenu de la table des couleurs installée. Sinon, la couleur obtenue par application de la fonction involutive sur la valeur du pixel peut être impossible à distinguer de la couleur originelle du pixel, rendant le dessin invisible.

> Par exemple, si on utilise un écran où chaque pixel peut afficher 256 intensités lumineuses et la fonction involutive f, $f(i) = 255 - i$, qui est bien involutive :
>
> $$f(f(i)) = 255 - f(i) = 255 - (255 - i) = i$$
>
> Cette fonction permet un bon contraste pour les intensités extrêmes mais pas pour les objets gris moyen, dont la sélection sera indiscernable de l'objet. Le problème du choix de la table des couleurs et de la fonction involutive est traité en [Kopp et al.94].

transitoire : Une couche est *transitoire* lorsque son contenu n'apparaît que très peu de temps, de l'ordre de quelques secondes. La pile doit réagir rapidement aux modifications d'objets sur cette couche, quitte à ne pas prendre en compte des cas particuliers de réaffichage statistiquement peu probables mais coûteux.

> Par exemple, la couche de retour d'information lexicale gérant la trace a cet attribut. Lorsque la trace est initiée, elle doit s'afficher sans retard, sans quoi la manipulation donne une impression d'inertie et ne satisfait pas les utilisateurs.

La pile peut combiner plusieurs stratégies pour optimiser le réaffichage d'une couche transitoire :

- ignorer les dommages des autres couches si une couche transitoire est endommagée ;
- ignorer les régions invalides ;
- lorsque le système de fenêtrage et la carte graphique le permettent, utiliser un plan d'*overlay* (décrit en première partie, § 2.2) ;
- lorsque la surface virtuelle utilise du *double buffering*, dessine la couche transitoire sur la fenêtre directement plutôt que sur le *back buffer*.

Cache : Spécifie que cette couche utilise un cache et est très rapide à redessiner si elle n'est pas endommagée. Lorsqu'elle est endommagée, le cache

doit être recalculé avant de pouvoir se composer sur la surface virtuelle. Sur un système multiprocesseur, le réaffichage du cache peut être lancé en parallèle d'autres activités et diminuer ou annuler le temps d'attente lors du réaffichage.

Sauve-dessous : Spécifie que la couche, avant de se dessiner sur la surface virtuelle, fait une copie de la région sur laquelle elle s'affiche. Cette attribut n'existe que lorsque la surface virtuelle propose un mécanisme pour récupérer une région de l'image affichée, ce qui est le cas de tous les systèmes sauf Display PostScript. Nous discutons de la stratégie à adopter par la pile dans la section suivante.

2.6.5 Optimisation de l'affichage de plusieurs couches dans la pile

Lors de l'exécution de la fonction *dessine*, la pile peut combiner plusieurs mécanismes pour optimiser le réaffichage.

Par exemple, dans la configuration comportant un fond, une couche de visualisation passive, une couche de sélection involutive endommagée et une couche de retour d'information lexicale *sauve-dessous*, la fonction *dessine* de la pile peut effacer la couche de retour d'information lexicale et effacer la couche de sélection puis redessiner la couche de sélection et la couche de retour d'information lexicale sans redessiner le fond ni la couche de visualisation passive.

Prise en compte de l'effacement : Lorsqu'une couche qui sait s'effacer est endommagée et que les couches placées au-dessus savent aussi s'effacer, alors la pile a le choix entre redessiner du fond vers le premier plan ou effacer du premier plan à la couche endommagée puis la redessiner et redessiner les autres couches jusqu'au premier plan. Le choix dépend du coût de l'effacement et du coût du réaffichage et doit être fait pour chaque conception d'éditeur.

Prise en compte de l'attribut *sauve-dessous* : En général, l'attribut *sauve-dessous* est utilisé par la couche qui apparaît au-dessus de toutes les autres, en particulier la couche de retour d'information lexicale lorsqu'elle gère des curseurs. La pile doit alors collaborer avec la couche pour que l'image sauvée soit valide. Pour cela, au début de chaque réaffichage, elle demande à la couche de s'effacer (c'est-à-dire de réafficher le contenu de la surface virtuelle sauvée), puis réaffiche toutes

les couches avant d'appeler la fonction *dessine* sur la couche. Cette fonction commençe par sauver l'état de la surface virtuelle avant de dessiner le contenu de la couche.

Utilisation d'*overlay* : certaines architectures graphiques disposent d'un ou plusieurs plans d'*overlay*. Un éditeur peut décider d'utiliser ces *overlays* pour déléguer la composition des couches au système de fenêtrage. Le problème de cette stratégie est que les fonctions d'accès aux *overlays* ne sont pas portables, même lorsqu'elles utilisent le système X, car elles font appels à une extension non standardisée.

Ces trois stratégies de pile permettent d'éviter le réaffichage au détriment de la mémoire ou de ressources du système. Compte tenu des multiples configurations d'éditeurs, il n'est pas possible d'offrir une stratégie optimale, chaque application doit spécialiser sa pile en fonction des caractéristiques des couches à gérer.

2.6.6 Optimisations *ad-hoc*

En utilisant les attributs des couches, il est possible de définir des algorithmes de réaffichage *ad-hoc* qui optimisent la configuration exacte des couches utilisées par une application particulière. Bien qu'il ne soit pas possible de donner un algorithme général en fonction de la nature des couches, la mise au point de l'optimisation peut être repoussée jusqu'au dernier moment du développement d'un éditeur sans remettre en cause le reste de l'architecture. C'est à notre avis un argument méthodologique important en faveur de l'architecture multicouche, sur lequel nous revenons en § 3.

Les optimisations spécifiques peuvent être extrêmement importantes. Nous avons expérimenté les méthodes suivantes :

couche opaque : si une couche couvre toujours une région de la surface virtuelle (cas d'une image pixellaire ou vidéo par exemple), les couches placées derrière n'ont pas besoin d'être redessinées lorsque leur dommage est entièrement sous la région ;

masque de couches : Pour optimiser un réaffichage très fréquent d'une couche (cas de la vidéo), une technique consiste à calculer le masque de toutes les couches placées au-dessus et de protéger les pixels masqués avant d'appeler la fonction *dessine* de la couche.

2.7 Synthèse

Nous avons présenté dans ce chapitre notre architecture multicouche, qui repose sur la collaboration de trois types d'objets : la pile, les couches et les outils. La pile gère l'ordre d'affichage des couches et l'acheminement des événements. Chaque couche gère deux facettes :

- une surface virtuelle qui suit un modèle graphique, potentiellement différent de celui de la surface virtuelle du système de fenêtrage, et
- une structure d'objets graphiques destinées à s'afficher sur la surface virtuelle.

La couche implante les mécanismes d'affichage, de composition sur la surface virtuelle de la pile, et de désignation de ses objets graphiques. La gestion des événements est prise en charge par les outils, qui sont toujours associés aux couches. Plusieurs outils collaborent dans une pile pour implanter un mode d'interaction que nous décrivons de façon relativement simple et lisible à l'aide d'une variante de la notation UAN.

Cette architecture permet de décrire avec précision plusieurs types d'objets graphiques qui apparaissent dans les éditeurs, et leurs mécanismes de gestion des événements. Elle permet d'utiliser aussi bien les extensions graphiques disponibles dans les systèmes de fenêtrage que de multiples dispositifs d'entrée.

Nous ferons une comparaison entre l'architecture multicouche et les outils de construction d'interface en conclusion du chapitre. Avant cela, nous décrivons une méthode pour le développement d'éditeurs avec cette architecture.

Chapitre 3

Méthode

L'architecture multicouche permet de suivre une méthode de construction d'applications graphiques interactives où des étapes indépendantes peuvent être isolées et testées. Nous avons répertorié les phases suivantes :

- visualisation,
- visualisation paramétrée,
- interaction lexicale,
- interaction syntaxique,
- interaction sémantique.

Ces phases de développement correspondent généralement aussi à des phases de conception ou de raffinement des applications graphiques interactives.

3.1 Visualisation

La première étape du développement d'un éditeur doit permettre de visualiser les structures de données à éditer. Cette visualisation peut être décrite comme une projection du noyau sémantique vers la présentation, et ne requiert aucun retour de la présentation vers le noyau sémantique.

À ce stade, les éléments suivants doivent être spécifiés :

- le modèle graphique de la visualisation,
- la représentation structurelle des objets graphiques,
- l'algorithme de gestion du placement automatique des objets graphiques,
- la méthode d'instanciation de la structure graphique à partir du noyau sémantique.

Ces éléments correspondent, dans le modèle de l'Arche, à la spécification de la présentation et la définition de la partie du contrôleur de dialogue qui permet de passer du noyau vers la présentation.

Pour tester cette partie, il suffit de créer une couche de fond et une couche de visualisation passive spécialisée pour la visualisation des données de l'éditeur. L'algorithme de placement, ainsi que les structures graphiques et le modèle de visualisation peuvent être testés sur plusieurs types de données.

3.2 Visualisation paramétrée

Une fois la visualisation au point, plusieurs paramètres de visualisation doivent être fixés indépendamment du noyau sémantique. Le réglage de point de vue par exemple (déroulement horizontal et vertical, niveau de zoom, etc.) peut interagir avec la visualisation. Lorsque le modèle graphique est schématique et que plusieurs attributs graphiques sont décoratifs ou conventionnels, il est parfois possible de modifier interactivement ou automatiquement ces attributs, par l'intermédiaire de boîtes de dialogue par exemple.

> Par exemple, dans le *Finder* du Macintosh, il est possible de changer la couleur ou la taille des icones interactivement. En réalité, c'est uniquement un attribut de visualisation qui est modifié.

Cette phase ne rajoute généralement pas de couche mais, si la vitesse d'affichage n'est pas suffisante pour l'interaction, ou peut rajouter une commande pour optionnellement relâcher le modèle graphique de la surface virtuelle.

Par rapport au modèle de l'Arche, cette phase n'est qu'une amélioration de la composante de présentation. Les interventions sur la composante de dialogue sont minimes et principalement composées de liaisons entre *Widget*s standards et attributs de la présentation.

3.3 Interaction lexicale

La phase suivante consiste à spécifier les dispositifs d'interaction, leurs modalités d'utilisation et leurs modes de retour d'information pour toutes les interactions prévues.

À ce stade, une planification de l'interaction doit être faite, bien que les interactions ne soient définies qu'aux stades suivants. Cependant, la spécification peut se contenter de donner une liste des couches utiles à l'éditeur. Même si des modes de retour sont oubliés, ils pourront être ajoutés ou adaptés par la suite.

Les couches ajoutées par cette phase gèrent la visualisation des contraintes lexicales, le rectangle de sélection et le retour d'information lexicale. La fonction d'application des contraintes lexicales est définie s'il y a lieu et mise à la disposition des outils. La couche de gestion de la sélection multiple est choisie pour utiliser un lasso ou un rectangle. La couche de gestion du retour d'information doit être spécialisée pour pouvoir visualiser les dispositifs, le cas échéant la trace (pour la reconnaissance de geste ou pour la trace elle-même).

Pour tester cette phase, les outils doivent être partiellement implantés ; seules les actions lexicales doivent être connectées aux couches. L'éditeur peut alors être testé avec tous ses dispositifs activés et le retour lexical de tous les modes d'interaction.

Pour tester le retour d'information lexical comme les animations de zoom (comme le double clique du *Finder*), certaines fonctions des couches doivent être définies qui se contentent de déclencher les retours lexicaux, sans déclencher aucune commande.

À ce stade, aucun retour de la présentation vers le noyau sémantique n'est encore défini.

3.4 Interaction syntaxique

Ce n'est qu'à ce stade que le contrôleur de dialogue est défini dans le sens de la présentation vers le noyau sémantique. Les couches de sélection et de manipulation directe sont ajoutées.

Toutes les manipulations directes des structures graphiques pourront être testées, ainsi que les commandes qu'elles déclenchent. Si des manipulations ont des contraintes sémantiques ou nécessitent un contrôle sémantique, ceux-ci ne peuvent pas être vérifiées à ce stade et se traduisent par des manipulations erronées, provoquant des erreurs lors de l'exécution de la commande.

> Par exemple, en troisième partie, § 1, nous décrivons un éditeur de graphe générique. Une des manipulations permet de créer une arête entre deux sommets. Si le sommet de départ est le même que le sommet d'arrivée, aucune arête ne peut être créée et la commande est invalide.

Plusieurs éditeurs sont considérés complets à ce stade.

3.5 Interaction sémantique

Quelques éditeurs évaluent pendant la manipulation la validité des actions potentielles et interdisent les actions terminales qui déclencheraient une commande

érronée ; c'est ce que nous appelons l'interaction sémantique. Le fait que la manipulation connaisse les actions autorisées et celles qui ne le sont pas, lui donne la possibilité de visualiser de façon particulière les états dans lesquels une action terminale est valide.

> Par exemple, en CAO, certaines opérations n'ont pas de sens dans tous les contextes. La manipulation d'une « vis » ne peut se terminer correctement que si elle est placée dans un écrou qui lui correspond. Ainsi, la manipulation directe d'un objet « vis » est supervisée dans le contrôleur du dialogue et peut déclencher l'affichage en surbrillance des zones autorisées, ainsi que leur magnétisme pendant le déplacement, et le refus de la fin de l'action si le lieu où elle s'est terminée n'est pas valide.
>
> Une telle supervision ne peut pas se faire uniquement à partir des structures de données manipulées par la présentation et le contrôleur de dialogue, un retour au noyau sémantique doit être fait.

Le fait de ne pas appliquer de contraintes sémantiques pendant la manipulation directe rend possible l'expression d'actions erronées, qui déclencheront en retour un message d'erreur du noyau sémantique (visualisé par une boîte de dialogue ou un message sonore). Cette phase est par conséquent très importante pour l'utilisateur, mais très difficile à mettre en œuvre en pratique car nécessitant souvent la modification du noyau sémantique, comme nous le décrivons en troisième partie, § 1.7.

3.6 Analyse critique

Dans la première partie, nous avons décrit les méthodes disponibles pour concevoir et réaliser un éditeur. Nous comparons ici les méthode préconisées.

3.6.1 Boîtes à outils

Aucune méthode n'est préconisée dans les boîtes à outils pour la définition de l'interaction non stéréotypée. Seule la construction d'applications bâties à partir de *Widget*s est parfois décrite dans les manuels d'utilisation et les guides de style.

3.6.2 Architectures logicielles

En SmallTalk, aucune méthode n'est préconisée, en dehors du fait que la plupart des codes sources sont disponibles en ligne et sont une source d'inspiration intéressante.

Les squelettes d'applications fonctionnent par modifications successives pour spécialiser un schéma général, et ne permettent que rarement d'arriver à des étapes stables et testables. De plus, ils requièrent d'avoir compris l'ensemble du schéma avant de commencer à le modifier.

Dans Garnet et Unidraw, la manipulation directe est contrôlée par les interacteurs qui peuvent parfois être spécialisés mais contiennent l'ensemble du code du contrôleur et sont donc monolithiques et peu configurables.

3.6.3 Modèles architecturaux

La plupart des modèles architecturaux sont descriptifs. Ils ne préconisent aucune méthode de conception. PAC et surtout PAC-AMODEUS préconisent une méthode, mais qui est surtout adaptée aux interfaces de contrôle et pas à la manipulation directe. Bien que des règles heuristiques soient décrites dans [Nigay94], elles ne s'appliquent pas à la construction d'éditeurs dont le contrôleur de dialogue est complexe et gère finement les objets de l'interaction.

3.6.4 Architecture multicouche

La méthode que nous avons décrit permet d'envisager la construction d'éditeur de façon *incrémentale*. Cette propriété permet à la fois d'obtenir des éditeurs robustes, mais aussi de les faire évoluer.

Chapitre 4

Implantation du graphique multicouche

L'implantation de l'architecture multicouche a été faite dans la boîte à outils C++ InterViews. Nous utilisons une notation proche d'OMT [Rumbaugh et al.91] pour décrire graphiquement l'interface des classes et les relations entre les classes. La notation est décrite dans l'annexe A.

Nous avons choisi InterViews car, à l'époque, c'était la seule bibliothèque C++ qui offrait plus qu'une simple encapsulation du système de fenêtrage et offrait des mécanismes unificateurs pour la gestion des objets graphiques. Nous nous sommes inspirés de la boîte à outils Xtv [Beaudouin lafon et al.90, Beaudouin lafon et al.91] pour implanter les gestionnaires de dispositifs et les attributs graphiques. La première version de notre architecture multicouche a été développée avec Xtv [Fekete92] en utilisant un mécanisme interne permettant de visualiser plusieurs couches (appelées *scènes* dans Xtv) sur une même vue. Nous avons aussi tenté d'unifier notre architecture à Unidraw mais le mécanisme des interacteurs n'est pas compatible avec celui des outils des couches. Nous avons donc utilisé la bibliothèque qui comportait le plus d'éléments compatibles avec notre architecture et ajouté ceux qui manquaient.

4.1 Organisation

Notre implantation est séparée en trois parties :

– les modifications au noyau d'InterViews pour rajouter des mécanismes de base ;

– les extensions d'InterViews ainsi modifié ;

– la définition de couches, d'objets graphiques et de surfaces virtuelles permettant d'implanter quelques éditeurs.

4.2 Fonctionnement d'InterViews

Dans sa version 3.1 que nous avons utilisée, InterViews a les caractéristiques notables suivantes :

isole les application du système de fenêtrage : les objets vus par le système de fenêtrage sont encapsulés dans des objets C++ qui peuvent être portés ; de fait, InterViews fonctionne sous Unix avec X11, sous OS/2 et Windows. Les objets encapsulés sont la fenêtre (**Window**), la surface virtuelle (**Canvas**), l'événement (**Event**), la sélection (au sens du coupé/collé entre applications), les ressources (**Style**) ainsi que des attributs graphiques (image, police, bitmap, etc.).

unifie les objets graphiques : la racine de chaque objet graphique manipulé par InterViews est le **Glyph**. Les **Glyph**s forment un DAG en suivant le motif *Composite* (décrit page 75).

unifie structurellement les fonctionnalités : au lieu de dériver finement des objets semblables, InterViews prend le parti d'unifier toutes les fonctionnalités en une seule classe. L'arbre des classes d'InterViews est donc relativement peu profond et l'apprentissage des classes en est facilité.

utilise le comptage de référence : les objets susceptibles d'être partagés dérivent d'une classe qui compte les références et libère l'objet lorsque son compte devient nul. Contrairement à d'autres systèmes, le comptage de référence est explicite avec InterViews, le programmeur d'application doit appeler la méthode *ref* sur les objets qu'il référence et appeler la méthode *unref* lorsqu'il a terminé. Ce mécanisme est à l'origine de nombreux problèmes de fuites de mémoire, induit une lourdeur dans le code et pose des problèmes avec les références circulaires, qu'il faut éviter à tout prix.

La figure 4.1 décrit la hiérarchie des objets graphiques standards de InterViews.

4.2.1 Le Glyph

La classe la plus importante et dont tous les objets graphiques héritent est le **Glyph**, décrit en figure 4.2. Le type **Extension** est simplement un rectangle définit dans l'espace de sortie de la surface virtuelle.

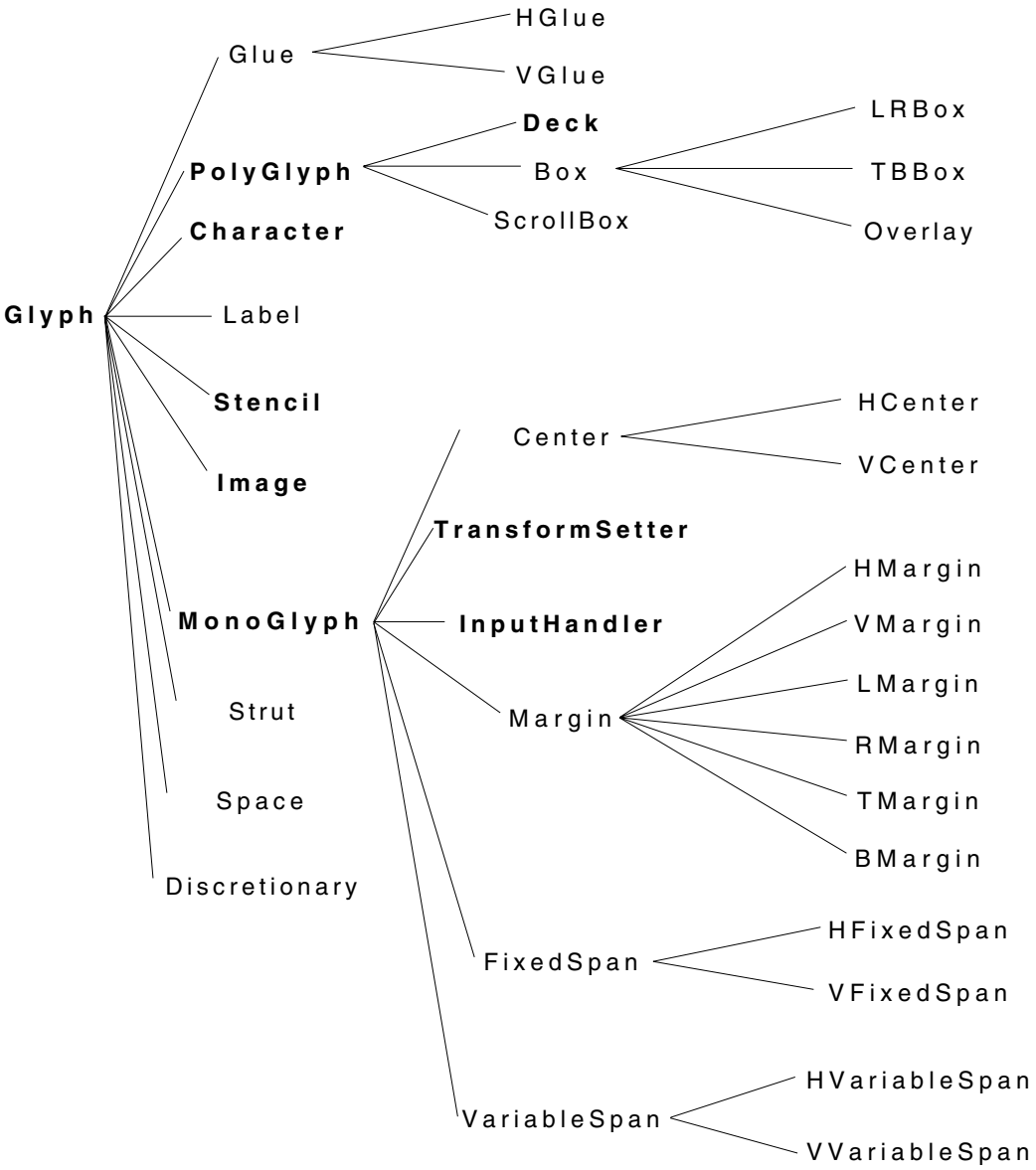


FIG. 4.1 - *Hiérarchie des classes d'objets graphiques d'InterViews. Les classes en gras font partie de la spécification tandis que les autres sont des implantations utilisées à travers des « kits ».*

```
type Glyph = classe;
début
    glyphs: ListeDe(Glyph);

    fn requiert(): Dimension;
    fn alloue(Canvas, Région): Région;
    proc dessine(in Canvas, in Région);
    proc oublie();
    proc intercepte(in Canvas, in Région, inout Hit);
    proc change();
fin;

type Axe = (X, Y);

type Dimension = classe;
début
    taille_naturelle: tableau[X..Y] de Réel;
    taille_max:       tableau[X..Y] de Réel;
    taille_min:       tableau[X..Y] de Réel;
    alignement:       tableau[X..Y] de Réel;
fin;

type Région = classe;
début
    largeur:          tableau[X..Y] de Réel;
    alignement:       tableau[X..Y] de Réel;
    origin:           tableau[X..Y] de Réel;
fin;

type Hit = classe; début
                    type CheminDeGlyph = ListeDe(Entier);

                    chemins: ListeDe(CheminsDeGlyph);
                    événement: Événement;
                    traiteur: Traiteur;
fin;
```

FIG. 4.2 - *Interface des **Glyph**s d'InterViews.*

Comme nous l'avons vu en première partie, § 3.1.2, la réquisition correspond à la dimension du **Glyph** exprimé dans un repère intrinsèque, tandis que l'allocation est la zone rigide qui est effectivement allouée au **Glyph** lorsqu'il doit s'afficher. Les **Glyph** suivent le motif *Flyweight* (décrit page 77) en passant l'allocation avec le **Canvas** lors de chaque affichage. Ainsi, un même objet peut apparaître à plusieurs endroits sur un même **Canvas** ou même sur plusieurs **Canvas**. Cette propriété est intéressante mais requiert beaucoup de soins pour être implantée correctement dans de nouveaux **Glyph**s.

4.2.2 Le réaffichage des Glyphs

Dans un éditeur, les objets graphiques sont modifiés et doivent se réafficher correctement. InterViews ne donne pas de mécanisme simple pour gérer la modification des **Glyph**s, c'est ce qui nous fait dire qu'il est plutôt adapté à la visualisation. Cependant, deux mécanismes peuvent être mis en œuvre pour réafficher une partie d'une surface virtuelle : le recalcul complet du GDA ou l'utilisation d'objets gérant une racine de GDA : les **Patch**s. Un **Patch** est un **Glyph** qui ne peut pas être partagé et qui retient le **Canvas** sur lequel il est affiché, ainsi que son allocation. Lorsqu'un objet dans la sous-hiérarchie du **Patch** est modifié, la fonction *redraw* du **Patch** peut être appelée pour invalider la zone sous le **Patch** et provoquer un réaffichage.

Notons qu'aujourd'hui, nous utiliserions Fresco, qui a été conçu avec un mécanisme de modification beaucoup plus simple à utiliser, chaque **Glyph** définissant deux méthodes : *need_redraw* et *need_resize* pour forcer le réaffichage sans imposer les contraintes du **Patch**.

4.2.3 Gestion des événements

La gestion des événements est faite de deux parties : désignation et action. Lorsqu'un événement arrive sur la fenêtre, il est placé dans un objet de classe **Hit** et la fonction *pick* est appelée sur la racine des **Glyph**s. À la fin du parcours, l'objet de type **Hit** contiendra la liste de tous les chemins vers les **Glyph**s qui reçoivent l'événement et, le cas échéant, l'objet **Handler** qui désire le gérer. La fonction *event* du **Handler** est appelée avec l'événement en paramètre. Ce n'est donc pas (nécessairement) l'objet graphique qui a accepté l'événement qui le gérera.

4.3 Les modifications du noyau

InterViews ne gère qu'une souris et un clavier, et n'implante qu'un modèle graphique. Nous l'avons étendu en ajoutant un objet **Device** pour décrire l'état des

dispositifs d'entrée. Nous avons aussi défini les mécanismes nécessaires à l'utilisation de plusieurs modèles graphiques, décrits en deuxième partie, § 2.5.

4.3.1 Événement et Dispositif

Pour gérer plusieurs dispositifs, nous avons défini une classe décrivant l'état d'un dispositif suivant le modèle de [Foley et al.90]. Nous avons ajouté à l'événement standard d'InterViews une fonction pour accéder au dispositif dont il est issu. Cette organisation permet de séparer clairement les rôles entre le dispositif qui a un état et l'événement qui décrit les modifications de l'état du dispositif à un certain moment.

4.3.2 Mécanismes pour étendre les modèles graphiques

InterViews définit une classe de surface virtuelle, le **Canvas**, qui définit un modèle semblable à PostScript. Le **Canvas** est dérivé en **Printer** qui traduit les opérations graphiques en langage PostScript dans un fichier, permettant ainsi d'imprimer les objets graphiques en leur passant un **Printer** à la place d'un **Canvas**.

Nous avons ajouté trois mécanismes, dont le schéma OMT est donné en figure 4.3, pour accéder à d'autres modèles graphiques et faire les optimisations décrites en § 2.6 :

- une classe implantant le motif *Decorator* (voir page 76) pour les **Canvas** ;
- une sous-classe de **Canvas** dessinant dans une image et gardant le masque des pixels modifiés ;
- une bibliothèque graphique complète permettant d'implanter dans l'application n'importe quel modèle graphique.

4.4 Couche et Pile

Nous avons aussi rajouté à InterViews la classe couche (**Layer**) et pile (**Viewer**) qui suivent fidèlement la définition donnée en figure 2.1 page 98.

La figure 4.4 décrit les classes **Layer** et **Viewer**. Étant dérivé de **InputHandler**, le **Viewer** peut s'utiliser comme n'importe quel objet graphique d'InterViews, sauf qu'il ne peut pas être affiché à deux places simultanément ; il est en général la racine d'un DAG. Structurellement, c'est un objet composite qui ne contient que des **Layer**s. La sémantique de placement des **Layer**s est simplement la superposition. Dans notre implantation, il est essentiel que le **Viewer** ne soit superposé avec aucun autre objet graphique car il doit maîtriser toute la surface virtuelle pendant le réaffichage.

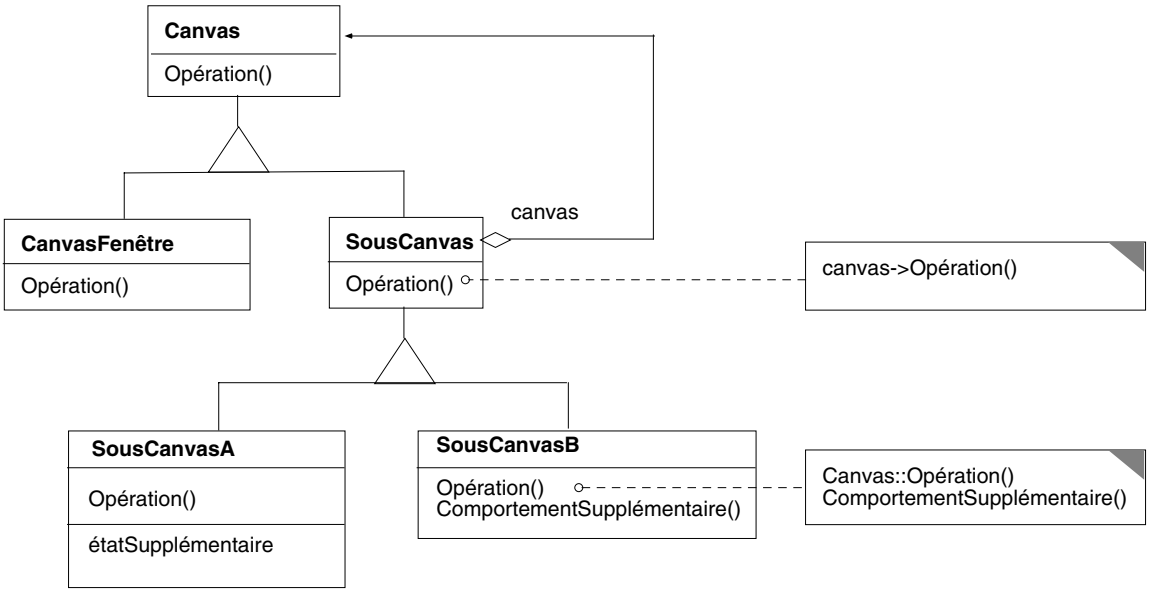


FIG. 4.3 - *Schéma OMT des classes dérivées de Canvas implantant le motif* Decorator

```
type Viewer = sous_classe de InputHandler;
début
        couches: ListeDe(Couche);
        canvas: Canvas;
        région_visible: Région;

        fn requiert(): Dimension;
        proc alloue(in Canvas, in Région);
        proc dessine(inout Canvas, in Région);
        fn intercepte(Événement): Booléen;
        proc traite(Événement);
fin;
```

```
type Layer = sous_classe de MonoGlyph;
début
        pile: Pile;
        attributs: EnsembleDe(AttributDeCouche);
        dommage: Région;

        fn requiert(): Dimension;
        proc alloue(in Canvas, in Région);
        proc dessine(inout Canvas, in Région);
        proc efface(inout Canvas, in Région);
        fn intercepte(Événement) : Booléen;
        proc traite(in Événement);
        proc endommage(in Région);
        proc installe_outil(in Entier);
        proc désinstalle_outil(in Entier);
        fnoutil_installé(Entier): Booléen;
fin;
```

FIG. 4.4 - *Les classes **Layer** et **Viewer**.*

```
type CoucheRectangle = sous_classe de Layer;
début
        proc commence(in Point);
        proc continue(in Point);
        proc termine(in Point);
        fn a_commencé() : Booléen;
fin;

type OutilRectangle = sous_classe de Tool;
début
        couche: CoucheRectangle;
fin;
```

FIG. 4.5 - *Exemple de dérivation de classes pour la couche Rectangle.*

4.4.1 Spécialisation des couches

Le schéma d'implantation des couches nécessite deux dérivations. Pour une couche nommée C, une classe abstraite dérivant de **Layer** est définie : CLayer. Elle définit les fonctions qui lui sont spécifiques. L'outil (CTool) est quant à lui dérivé de la classe outil et contient un pointeur vers un objet de la classe concrète CLayer.

Par exemple, la couche gérant le rectangle de sélection définit les deux classes décrites en figure 4.5.

L'implantation actuelle comporte les couches suivantes :

- CoucheFond,
- CoucheGrille (contraintes lexicales),
- CouchePrincipale (visualisation passive),
- CoucheSélection (sélection et manipulation directe),
- CoucheRectangle (gestion du rectangle de sélection),
- CoucheManipulation (gestion de la manipulation directe),
- CoucheCurseur (retour d'information lexicale).

Cette implantation est conforme à la description faite en deuxième partie, § 2.3.

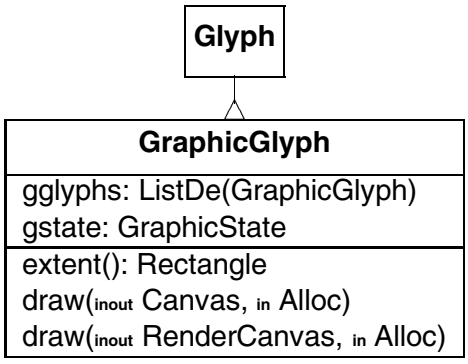


FIG. 4.6 - *Spécialisation d'un objet pour le graphique structuré.*

4.5 Graphique structuré

La couche de visualisation passive ne définit aucune sémantique particulière. Nous l'avons dérivée pour qu'elle gère du graphique structuré. Nous avons limité l'édition à une liste d'objets graphiques disposés sur la couche de visualisation. Chaque objet peut être composite mais nous n'avons pas implanté les fonctionnalités pour gérer l'intérieur des objets composites.

Les objets graphiques sont des **GraphicGlyph**s, ils héritent de **Glyph** et ont l'interface décrite en figure 4.6.

Pour rendre la classe **GraphicGlyph** la plus générique possible, nous l'avons définie ne contenant qu'un **GraphicState**. Le **GraphicState** est une table qui associe à un nom d'attribut graphique sa valeur. L'utilisation d'un état graphique au niveau des objets graphiques permet, comme dans PostScript, de représenter un objet graphique par une liste d'attributs dont certains peuvent être ignorés par un modèle graphique et pris en compte par un autre. C'est le cas des modèles graphiques relâchés, mais aussi lorsque des attributs sont dédiés à certains dispositifs de sortie et n'ont pas de sens sur d'autres (par exemple la trame qui n'a de sens que pour une imprimante).

Notons qu'une hiérarchie de **GraphicGlyph**s ne peut contenir que des **GraphicGlyph**s, qui sont des objets rigides. C'est pourquoi la méthode *extent* peut être utilisée à la place de la méthode *request*, qui fonctionne toujours mais implique une transformation vers une représentation moins naturelle et plus coûteuse qu'un rectangle englobant.

Nous avons défini les attributs suivants :

Transform la transformation homogène 3×3 appliquée au **GraphicGlyph** et à ses descendants ;

Path la description de la géométrie du **GraphicGlyph** ;

FillColor, StrokeColor, ... tous les attributs graphiques standards du modèle PostScript.

En décrivant les attributs graphiques et la géométrie sous forme d'attributs généralisés, nous avons rendu les **GraphicGlyph** génériques, ce qui permet à chaque éditeur de rajouter ses propres attributs ou descriptions de géométrie/topologie (nous en donnons un exemple en §2).

La classe **GraphicGlyph** est dérivée en plusieurs classes qui implantent les objets composites (**GraphicGroup**), les formes géométriques définies par des courbes de Bézier (**GraphicPath**), les images pixellaires (**GraphicImage**) ainsi que plusieurs autres classes utilitaires. Tous ces objets graphiques peuvent être composés en un GDA.

4.5.1 Gestion de modèle graphique modifié

Dans la définition d'un **Glyph**, la fonction *draw* implante le passage du graphique à état vers le graphique passif. Pour qu'une famille de **Glyph**s puisse se dessiner sur une surface virtuelle définissant un nouveau modèle graphique, il faut que chacune des classes de la famille hérite d'une classe de base définissant la fonction *draw* avec en premier paramètre le type concret définissant le nouveau modèle graphique.

Pour offrir la possibilité de rajouter un modèle graphique en C++, il faudrait pouvoir dériver toute une famille de classes dérivées d'une même classe de base, ce qui n'est pas très commode.

Une des solutions à ce problème est l'utilisation des informations de type pendant l'exécution. Chaque **Glyph** pourrait alors analyser le type réel de la surface virtuelle qui lui est passé au moment de s'afficher et choisir la méthode la plus spécialisée. Hélas, l'analyse de cas, en particulier pendant l'affichage, est onéreuse et doit être évitée.

Une autre solution, plus élégante, est de définir une famille de classes génériques, paramétrées par le type concret de la surface graphique sur laquelle ils s'affichent. Nous n'avons pas pu employer cette méthode à cause des limitations des compilateurs C++ concernant l'utilisation des classes génériques au moment de l'élaboration de notre système.

Nous avons opté pour une solution simple, consistant à déclarer, en plus du **Canvas**, l'existence d'un type **RenderCanvas** non spécifié dans la bibliothèque et qui peut être déclaré et défini par les applications qui le désirent. La bibliothèque contient aussi un ensemble de classes permettant de construire un **RenderCanvas** à partir des briques graphiques de base.

Chaque application peut donc définir ce type et implanter toutes les fonctions *draw* associées. Bien entendu, ce mécanisme ne permet pas de rajouter plus d'un

modèle graphique en dehors de celui du **Canvas**. Nous sommes sûre que des éditeurs nécessitant plusieurs modèles graphiques différents sont utiles et, avec l'évolution des compilateurs, la définition de classes génériques permettra de résoudre cette limitation.

4.6 Synthèse

Pour implanter des éditeurs utilisant notre architecture, nous avons modifié la boîte à outils C++ en lui rajoutant les mécanismes de base nécessaire à la gestion de dispositifs génériques, la définition de nouveaux modèles graphiques. Nous avons rajouté les classes **Layer** et **Viewer** pour gérer les couches et les piles. Enfin, nous avons défini des objets destinés à implanter le graphique structuré.

Pour donner une idée des modifications apportées à InterViews, voici une estimation quantitative :

InterViews, dans sa version native, définit environ 110 classes. Nous avons ajouté environ 70 classes pour définir notre bibliothèque. Les classes définies se répartissent dans les catégories listées ci-dessous.

Catégorie	Natif	Ajouté	Total
Noyau	29	1	30
Primitives	23	1	24
+ adjointes	20	3	23
Utilitaires	37	19	56
multicouche		15	15
Modèle graphique		13	13
Graphique structuré		20	20
Total	109	72	181

De notre point de vue, la quantité totale de classe n'est pas vraiment importante, c'est plutôt la quantité par catégorie qui doit être gardée la plus petite possible afin de faciliter la mémorisation et donc l'utilisation de la bibliothèque. Comme souvent dans les bibliothèques graphiques, la catégorie « utilitaire » est la plus grande car elle est constituée de tous les « *Widget*s». Les autres catégories contiennent entre 15 et 30 classes, ce qui nous paraît utilisable rapidement et mémorisable en un mois ou deux.

Chapitre 5

Synthèse

Notre architecture multicouche pour la construction d'éditeurs apporte les bénéfices suivants :

- séparation des objets graphiques de natures différentes dans des couches différentes ;
- gestion non centralisée de l'interaction ;
- possibilité d'optimisations graphiques et d'utilisation des spécificités matérielles de façon relativement simple ;
- description explicite et presque déclarative de la manipulation directe ;
- méthode de construction incrémentale.

Nous pouvons appliquer les mêmes critères d'évaluation que ceux utilisés dans la première partie sur les architectures logicielles.

Temps d'apprentissage : le schéma de base requiert moins de temps qu'un squelette d'application car il ne s'agit plus de modifier un squelette préconfiguré mais de construire une configuration à partir de briques de base. Les spécialisations peuvent être plus variées qu'avec les squelettes, il nous est donc difficile de faire une comparaison.

Temps de construction : Lorsque les briques de base suffisent, le temps de construction est comparable à celui requis par un squelette. Lorsque des modifications sont nécessaires, tout en restant conforme à l'architecture, la réalisation peut être plus précisément planifiée qu'avec les squelettes.

Méthodologie de construction : Nous préconisons une méthode constructive qui permet des tests au fur et à mesure de la construction d'un éditeur.

Modèle du graphique : Plusieurs modèles peuvent coexister, bien que l'implantation actuelle soit limitée à cause des limites des compilateurs.

Gestion de dispositifs : Nous pouvons gérer plusieurs dispositifs simultanément. Nous ne décrivons cependant aucun mécanisme pour les gérer de façon « synergique » (au sens décrit par [Nigay94]).

Modularité de l'architecture : Notre architecture décrit des schémas de collaboration entre des objets qui décrivent l'interaction de façon plus fine que les squelettes d'application. Les seuls problèmes de modularité apparaissent dans la définition des outils : ils doivent maîtriser parfaitement l'état de chaque couche pour être sûrs que les événements prévus peuvent bien leur arriver et que les préconditions seront vérifiées.

Extensibilité des applications : Notre architecture permettant la construction incrémentale, un éditeur peut généralement être étendu plus simplement qu'avec un squelette d'application. Nous n'avons pas rencontré d'exemple d'extension qui nous oblige à revoir globalement l'organisation d'un éditeur.

Compacité du code source : Comparée aux architectures dont le contrôle est centralisé, notre architecture pousse à définir plus de fonctions dont le code est généralement court. Chaque fonction participe pourtant à une étape précise de l'interaction et ne ressemble en rien aux *Callbacks* qui ne sont que des liants imposés par la structure de langages de programmation comme C.

Les propriétés de cette architecture permettent la construction effective d'éditeurs de qualité commerciale, c'est-à-dire optimisées, robustes et extensibles. Cependant, le niveau d'abstraction de l'architecture est élevé et requiert une bonne culture informatique qui est longue à acquérir.

Troisième partie

Utilisation de l'architecture

Dans cette partie, nous présentons deux applications graphiques interactives à manipulation directe construites à partir de notre architecture et de notre implantation sous InterViews.

La première application est un éditeur générique de graphes orientés. Elle permet de mettre en œuvre notre méthode sur un exemple concret. La seconde application est un éditeur graphique destiné au dessin animé qui fait partie du système professionnel TicTacToon [Fekete et al.95]. Il démontre une utilisation sophistiquée de notre architecture et quelques extensions qui en ont découlées sans modifications importantes de l'architecture initiale.

Tandis que le premier éditeur met l'accent sur le noyau sémantique et les contraintes qu'il impose à l'interaction, le second met l'accent sur l'utilisation intensive de plusieurs couches, à la fois pour le graphique et pour la gestion de l'interaction.

Nous décrivons chaque éditeur par :

- sa fonction et ses domaines d'applications,
- ses fonctionnalités, sous la forme d'une liste d'actions,
- les couches qui le composent,
- les outils qui gèrent ses modalités d'interaction,
- une conclusion avec quelques évaluations quantitatives.

Chapitre 1

Visualisation et édition d'un graphe

Nous allons décrire la construction d'un système d'édition de graphe générique en suivant notre méthode. Cet éditeur permet de manipuler une structure de données constituée de sommets et d'arêtes. Pour visualiser le graphe, le placement des sommets est calculé dynamiquement à l'aide d'un algorithme dérivé de [SK94].

Cette application illustre la situation où un noyau de programme existe (il s'agit des fonctions de gestion d'un graphe) et doit être rendu graphique.

1.1 Fonctionnalités de l'éditeur

Chaque sommet à un attribut pour le placement qui peut avoir la valeur « fixe » ou « déplaçable ». Seuls les sommets déplaçables sont placés automatiquement, les autres conservent leur place. Lorsqu'un sommet est déplacé interactivement, il prend le type « fixe » pour l'algorithme de placement afin de conserver la position spécifiée interactivement. La commande 6 permet de changer cet attribut interactivement.

Cet éditeur permet les commandes suivantes (voir figure 1.1) :

1. création de sommet,
2. création d'arête entre sommets,
3. destruction de sommet,
4. destruction d'arête,
5. déplacement de sommet,
6. modification de l'attribut (fixe,déplaçable) de sommet.

Nous décrirons les outils que nous avons choisis pour implanter ces actions en 1.5.

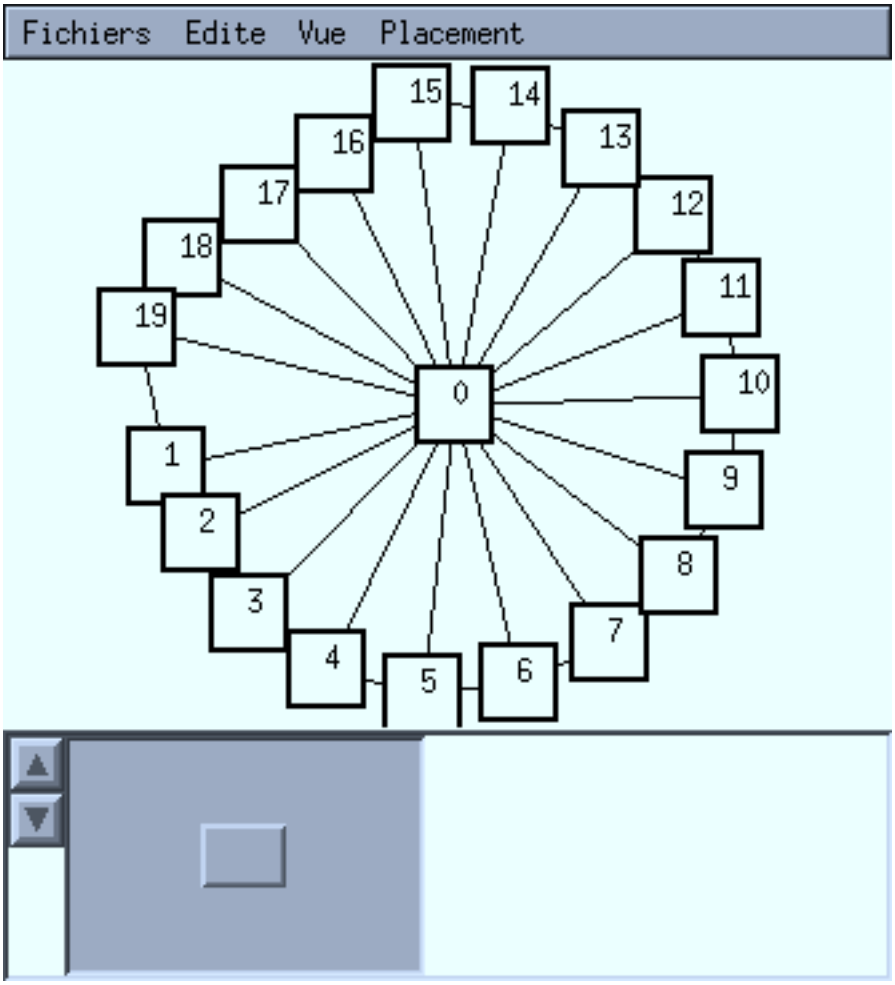


FIG. 1.1 - *Le programme d'édition de graphe*

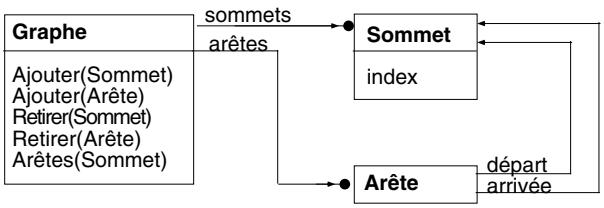


FIG. 1.2 - *Interface du noyau sémantique «Graphe général ».*

1.2 Méthode de conception

Nous suivons la méthode décrite en deuxième partie, § 3 qui décompose la conception et réalisation suivant les phases :

- visualisation,
- visualisation paramétrée,
- interaction lexicale,
- interaction syntaxique,
- interaction sémantique.

Notre éditeur ne nécessite pas d'interaction sémantique ni de dispositifs spécifiques, nous suivons donc l'ordre suivant :

1° mise au point de la visualisation passive du graphe ;

2° ajout de la visualisation paramétrée ;

3° ajout de l'interaction syntaxique : gestion de la manipulation directe de la sélection, sans retour du placement global du graphe ;

4° amélioration du retour d'information par l'affichage des étapes de convergence du placement du graphe pendant la manipulation directe.

Un graphe sera défini comme une liste de sommets et d'arêtes. Un sommet est identifié par son nom et contient une liste d'arêtes orientées. Une arête est identifiée par les sommets à ses deux extrémités (voir figure 1.2).

1.3 Visualisation

Pour visualiser le graphe, nous utilisons trois objets graphiques dérivés de **GraphicGlyph** pour représenter le sommet, l'arête et leur conteneur. Un algorithme de placement doit calculer la position des sommets dans le conteneur. Il requiert, en plus de la structure de graphe, les attributs décrits dans la figure 1.3.

La méthode la plus simple pour visualiser le graphe consiste à fixer arbitrairement la taille de chaque sommet. La présentation consiste alors à afficher le nom

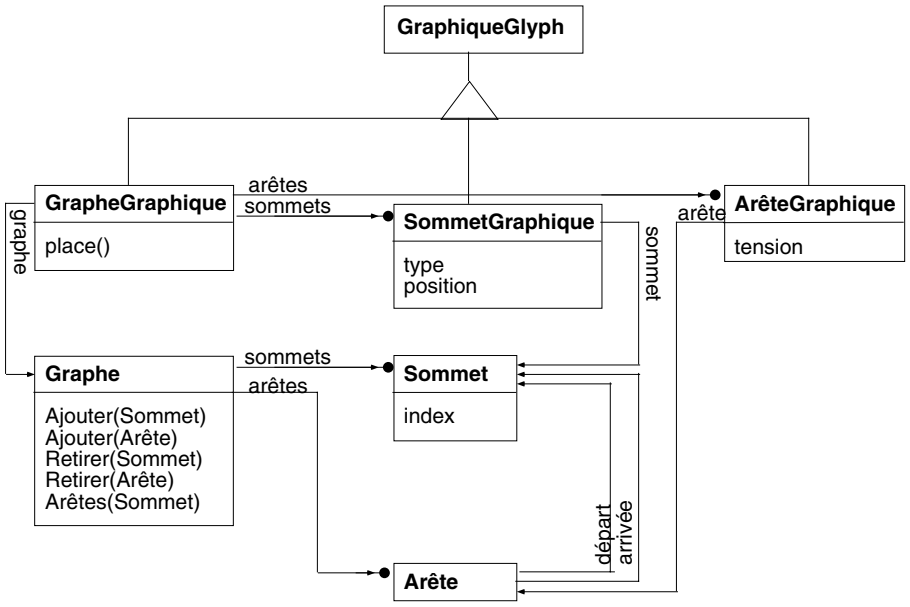


FIG. 1.3 - *Interface du Graphe adapté à la visualisation.*

du sommet encadré et à tracer les arêtes comme si elles partaient du centre des sommets.

La couche de visualisation n'a pas besoin d'être spécialisée pour visualiser le **GrapheGraphique**, celui-ci étant compatible avec un **Glyph** standard. Une fois les trois classes créées, le programme de visualisation affiche une image comme celle de la figure 1.4. Son implantation est la suivante :

```
fn main(argc: Entier, argv: tableau[0..argc] deChaine);
début
    var session: Session,
        graphe: Graphe,
        grapheg: GrapheGraphique;

    session := créer Session("Graphe", arguments);
    graphe := Graphe::charge(arguments[1]);
    grapheg := créer GrapheGraphique(graphe);

    session.window(
            créer ApplicationWindow(
                créer Viewer(
                    créer CoucheFond(),
                    créer CouchePrincipale(grapheg)
                )
```

```
                )
        );
        retourne session.run();
fin;
```

Le programme initialise la session, charge le graphe à partir d'un fichier dont le nom est passé en premier argument au programme et le passe au **GrapheGraphique** créé. Ensuite, l'arbre des **Glyph**s est créé et placé dans une fenêtre de type **ApplicationWindow**, qui est la fenêtre principale de l'application. L'arbre des **Glyph** est simplement composé d'une pile (**Viewer**) contenant la couche de fond et la couche de visualisation passive, celle-ci gérant le **GrapheGraphique**.

1.4 Visualisation paramétrée

Le programme de visualisation permet de tester l'algorithme de placement. Dans un éditeur complet, la visualisation doit se faire à partir d'un point de vue dont la taille est fixe tandis que le graphe peut prendre une dimension quelconque. La pile est donc placée dans un objet graphique qui gère un point de vue. Les *Widget*s de contrôle indirect sont placés et certaines fonctions peuvent être définies à ce stade, par exemple pour :

- charger un nouveau graphe,
- sauvegarder le graphe courant,
- placer aléatoirement les sommets du graphe,
- replacer les sommets du graphe.

Le point de vue est contrôlé par un facteur de zoom et un **Panner** qui remplace les barres de défilement. La boîte d'outils est positionnée mais n'est pas active. L'éditeur a alors son apparence finale.

1.5 Interaction syntaxique

Une fois tous les *Widget*s de contrôle indirect en place, les trois couches de contrôle de l'interaction peuvent être rajoutées : la sélection, le rectangle de sélection et la manipulation directe. Les outils sont ensuite définis et branchés à la boîte d'outils. La couche de visualisation passive est spécialisée et implante les fonctions décrites en figure 1.5, qui donnent aux outils toutes les fonctions permettant d'implanter les commandes décrites en § 1.1.

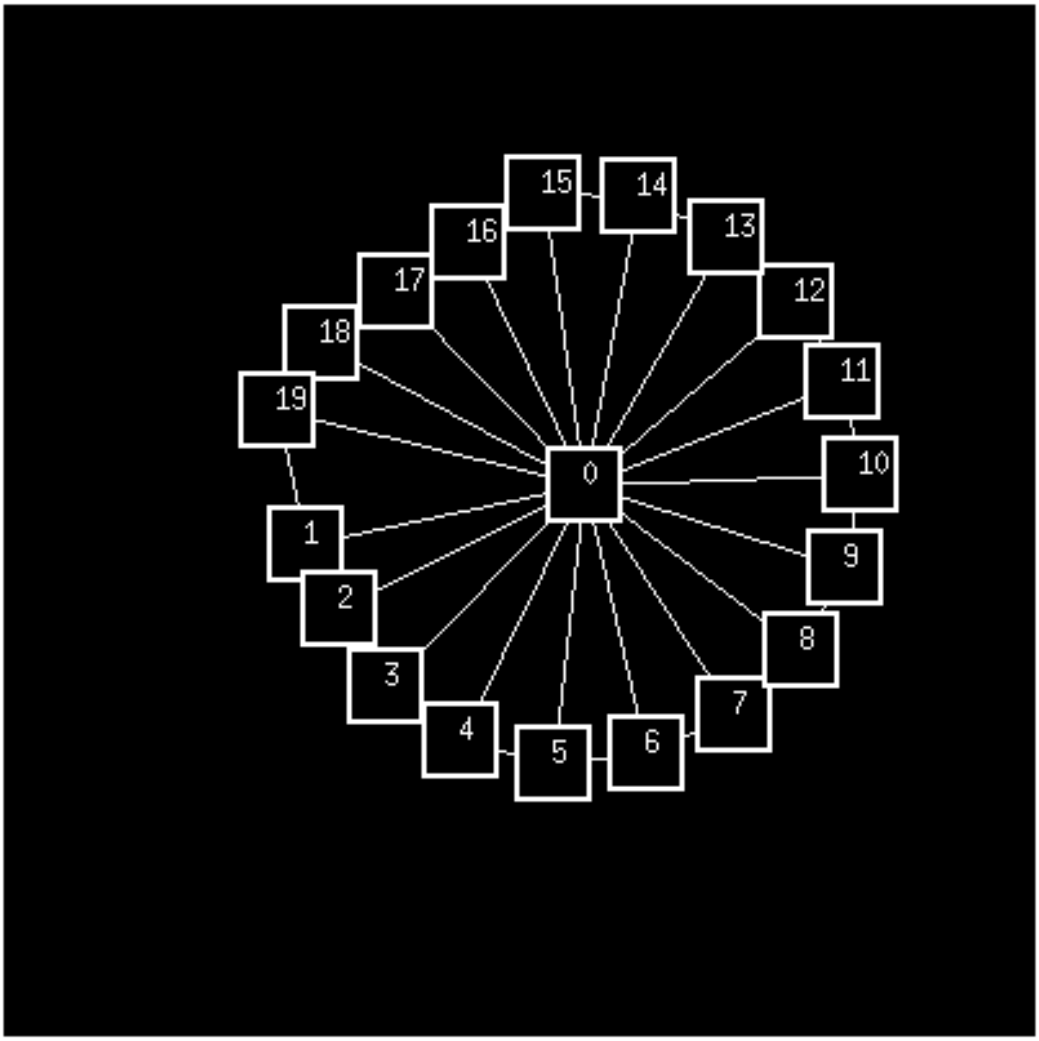


FIG. 1.4 - *Première étape de construction: La visualisation d'un graphe*

```
type CoucheVPGraphe = sous_classe de CouchePrincipale;
début
    graphe: GrapheGraphique;

    proc crée_sommet(in Point);
    proc détruit_sommet(in Sélection);
    proc crée_arête(in SommetGraphique,
                    in SommetGraphique);
    proc détruit_arête(in ArêteGraphique);
    proc replace(in Vecteur);
    proc change_type(in Sélection);
fin;
```

FIG. 1.5 - *Spécialisation de la couche de visualisation passive pour le graphe.*

La couche de manipulation directe spécialisée est décrite en figure 1.6. Les fonctions de la couche permettent de contrôler la manipulation du fantôme des sommets, elles seront décrites dans le contexte des outils qui les utilisent.

1.5.1 L'outil de sélection/déplacement

La sélection et le déplacement des sommets fonctionnent conformément à la description faite en deuxième partie, § II.3, décrite en III.1. C'est l'outil qui est activé initialement. L'unique différence est le nom de la commande appelée : *replace* au lieu de *déplace*. Elle est décrite dans la section qui suit.

1.5.1.1 Fonctions de la couche de visualisation passive

La fonction *replace* agit sur les sommets sélectionnés. Elle leur attribue le type « fixe », les déplace de $\Delta x, \Delta y$ et relance l'algorithme de placement sur le graphe ainsi modifié.

Les fonctions du menu *Edite* : couper, coller et détruire, utilisent la sélection et peuvent être implantées avec les fonctions de la couche de visualisation passive.

1.5.1.2 Fonctions de la couche de manipulation directe

La fonction *commence* crée des **Fantôme**s (décrits en deuxième partie, § 2.3.5) qui affichent un rectangle pour chaque sommet appartenant à la sélection et ont initialement la position des sommets de la couche de visualisation passive. Le pseudo-code de la fonction *commence* est donné en figure 1.7.

```
type CoucheMDGraphe = sous_classe de CoucheManipulation;
début
    départ: Point;
    dernier: Point;

    proc commence(in Point, in CoucheVPGraphe);
    proc commence_arête(in Point);
    proc continue(in Point);
    proc continue(in Point, in Placeur);
    proc termine(in Point);
    fn a_commencé(): Booléen;
fin;
```

FIG. 1.6 - *Spécialisation de la couche de manipulation directe pour le graphe.*

Tâche III.1 Sélection et déplacement dans le graphe (s représente un sommet).

Couche	Contexte	Événement	Action locale	Action transmise
Fond		$\sim [x, y]\mathrm{M}\downarrow$		**VP** désélectionne_tout() **Sélection** détruit_poignées() **Rectangle** commence(x, y)
VP		$\sim [s]\mathrm{M}\downarrow$	sélectionne(s)	**Sélection** poignées(s)
Sélection		$\sim [s]\mathrm{M}\downarrow$	détruit_poignées()	**MD** commence(x, y)
MD	*a_commencé()*	$\sim [x', y']*$	continue(x', y')	
	a_commencé()	$\sim [x'', y'']\mathrm{M}\uparrow$	termine(x'', y'')	**VP** replace($\Delta x, \Delta y$)
Rectangle	*a_commencé()*	$\sim [x', y']*$	continue(x', y')	
	a_commencé()	$\sim [x'', y'']\mathrm{M}\uparrow$	termine(x'', y'')	**VP** select(rectangle()) **Sélection** poignées(rectangle())

```
proc commence(p: Point, c: CoucheVPGraphe);
début
    var i: Entier
        f: Fantôme;

    départ := p;
    pour i := 0 jusqu'à c.count() fait
        si c.est_sélectionné(i) alors
        début
            f := créer FantômeSommet(i,c.région_objet(i));
            fantômes_ajouter(f);
        fin;
    endommage_tout(p);
fin;
```

FIG. 1.7 - *Création des fantômes des sommets au début de la manipulation*

1.5.2 L'outil de manipulation des arêtes

Nous avons choisi de ne pas rendre les arêtes sélectionnables. Leur manipulation requiert donc un outil particulier, décrit en III.2.

La création d'une arête se fait en commençant à cliquer sur un sommet et en relâchant le bouton sur un autre. Le fait de cliquer sur une arête la détruit. Toute autre action appelle la fonction *bip*. Notons que les actions sur la couche de manipulation directe sont pilotées par la couche de visualisation passive.

1.5.3 L'outil de création/destruction de sommet

Finalement, la création d'un nouveau sommet requiert un outil dont nous montrons le schéma UAN en III.3.

Lorsque le bouton du pointeur est pressé sur le fond, une boîte de dialogue apparaît pour saisir un nom et un sommet est créé à la place du pointeur. Lorsque le bouton est pressé sur un sommet existant, il est détruit.

1.6 Amélioration du retour d'information

L'outil de déplacement provoque le recalcul du placement global à la fin de la manipulation directe. Il est plus agréable de voir interactivement la déformation du graphe pendant que l'utilisateur déplace la sélection. Cette fonctionnalité n'est

Tâche III.2 Manipulation des arêtes dans le graphe (s représente un sommet et a une arête).

Couche	Contexte	Événement	Action locale	Action transmise
Fond		$\sim [x, y]\mathrm{M} \downarrow$	bip()	
VP		$\sim [a]\mathrm{M} \downarrow$	détruit(a)	
		$\sim [s]\mathrm{M} \downarrow$		**MD** commence_arête(s)
	MD.a_commencé()	$\sim [x', y']*$		**MD** continue(x', y')
	MD.a_commencé()	$\sim [s']\mathrm{M} \uparrow$	crée_arête(s,s')	**MD** termine(x'', y'')
	MD.a_commencé()	$\sim [x, y]\mathrm{M} \uparrow$	bip()	**MD** termine(x'', y'')
MD				

Tâche III.3 Création et destruction de sommet dans le graphe (s représente un sommet).

Couche	Contexte	Événement	Action locale	Action transmise
Fond		$\sim [x, y]\mathrm{M} \downarrow$		**VP** crée_sommet(x, y)
VP		$\sim [s]\mathrm{M} \downarrow$	détruit_sommet(s)	

généralement pas disponible à cause du temps de calcul de placement automatique qui peut être très grand et rendrait l'éditeur trop peu réactif.

Le calcul du placement étant itératif, la visualisation de chaque étape donne une idée intéressante de son évolution et permet de donner un retour à l'utilisateur. La table 1.1 donne un pseudo-code des fonctions. Pour l'implanter, nous avons spécialisé la couche de manipulation directe afin qu'elle gère une copie de la structure de graphe telle qu'elle est vue par l'algorithme de placement. Cette structure est passée lors de chaque mouvement de la souris pendant la manipulation, en plus des autres informations. Pour assurer la mise à jour incrémentale de l'affichage, nous utilisons la fonction *continue* de la couche MD pour répercuter la position du pointeur aux sommets sélectionnés et stocker l'objet **Placeur** dans la couche. Lors du réaffichage, la fonction *dessine* de la couche MD calcule la position des **Fantôme**s à l'itération suivante.

Cette implantation suppose que le coût d'une itération est acceptable pour l'interaction. Si ce n'est pas le cas, d'autres mécanismes sont bien sûr envisageables, comme l'utilisation d'un processus parallèle. Notre propos est de montrer comment l'interaction peut être améliorée incrémentalement en modifiant peu l'architecture. Cette méthode n'est qu'un exemple.

1.7 Interaction sémantique

Nous n'avons pas décrit de phase de spécialisation avec retour sémantique. Dans un éditeur dérivé de celui-ci, une contrainte sémantique pourrait empêcher de lier des sommets « incompatibles », ou pourrait limiter le nombre d'arêtes par sommet.

Un problème fréquemment rencontré avec les noyaux non prévus pour l'interaction sémantique est que le seul moyen de savoir si une opération est valide est d'appeler la fonction qui l'effectue et de tester si aucune erreur ne s'est produite. Ce mode de vérification des contraintes sémantiques ne peut pas être utilisé par l'interaction sémantique, elle est généralement trop coûteuse car elle oblige à annuler l'opération après qu'elle ait été faite (lorsque le noyau le permet).

La seule solution viable à ce problème est de renoncer au principe d'intégrité du noyau sémantique et d'essayer de le modifier pour lui ajouter des prédicats de validité sur les arguments des fonctions. Avec ces prédicats, un retour sémantique peut être fait.

> Par exemple, pour spécialiser notre éditeur en vérifiant des contraintes sémantiques, il faudrait ajouter une fonction à la couche de visualisation passive qui, pendant la phase de connexion d'une arête, provoque un changement d'aspect du sommet au-dessus duquel le poin-

```
proc commence(p: Point, c: CoucheVPGraphe, p: Pile);
début
    var i: Entier,
        f: Fantôme;

    départ := p;
    pour i := 0 jusqu'à c.count() fait
    début
        f := créer FantômeSommet(i,c.région_objet(i));
        fantômes_ajouter(f);
    fin;
    endommage_tout(p);
fin;

proc continue(p: Point, c: CoucheVPGraphe, g: Placeur);
début
    var i: Entier,
        v: Vecteur,
        f: Fantôme;

    dernier := p;
    placeur := g;
    v := dernier - premier;
    pour i := 0 jusqu'à c.count() fait
    début
        f := fantômes_accède(i);
        si c.est_sélectionné(i) alors
            f.déplace(v);
        sinon
            f.déplace(Vecteur(0,0));
        fin si
    fin;
    endommage_tout(p);
fin;

proc dessine(c: Canvas, r: Région);
début
    var i: Entier,
        v: Vecteur,
        f: Fantôme;

    pour f dans fantômes fait
        f.dessine(c, r);
    si placeur ≠ nil alors
    début
        placeur.iteration(fantômes);
        endommage_tout(p);
        si placeur.terminé() alors
            placeur := nil;
    fin;
fin;
```

TAB. 1.1 - *Fonctions de gestion de la manipulation directe avec retour du placement incrémental.*

teur passe pour indiquer qu'il peut être lié. Dans notre schéma UAN, une ligne serait rajoutée, comme l'indique le schéma UAN III.4

Tâche III.4 Manipulation des arêtes dans le graphe avec retour d'information sémantique. (s représente un sommet et a une arête).

Couche	Contexte	Événement	Action locale	Action transmise
Fond		$\sim [x, y]\text{M}\downarrow$	bip()	
VP		$\sim [a]\text{M}\downarrow$	détruit(a)	
		$\sim [s]\text{M}\downarrow$		**MD** commence_arête(s)
	MD.a_commencé()	$\sim [x', y']*$	éteint_tout()	**MD** continue(x', y')
	MD.a_commencé()	$\sim [s]*$	éclaire_départ(s)	
	MD.a_commencé()	$\sim [x, y]*$	éteint_tout()	
	MD.a_commencé()	$\sim [s]*$	éclaire_arrivée(s)	
	MD.a_commencé()	$\sim [s']\text{M}\uparrow$	crée_arête(s,s')	**MD** termine(x'', y'')
	MD.a_commencé()	$\sim [x, y]\text{M}\uparrow$	bip()	**MD** termine(x'', y'')
MD				

1.8 Synthèse

Dans cet exemple, nous avons mis en application notre architecture et notre méthode pour réaliser un éditeur de graphe. Cet éditeur est générique et demande à être spécialisé pour un domaine particulier ; en cela, il ressemble à un *Widget*. Cependant, contrairement à un *Widget*, son architecture interne est visible et modifiable, ce qui lui permet d'être spécialisé très finement.

Par exemple, nous avons appliqué cet éditeur à la sélection interactive de pages WWW pour leur impression et leur copie locale. Chaque page est un sommet et les arêtes représentent les références vers d'autres pages.

Il est possible de rajouter une couche gérant la trace et la reconnaissance de gestes, qui permet d'ajouter une modalité sans toucher à la structure de l'éditeur.

Catégorie	Classes
Couches	1
Objets graphiques	3
Outils	4
Commandes	6
Autres	4
Total	18

TAB. 1.2 - *Nombre de classes par catégorie utilisées par l'éditeur de graphe.*

La mise au point d'un programme semblable est au mieux difficile avec les boîtes à outils de bas niveau et impossible avec les squelettes d'applications actuels. Nous pensons qu'à l'aide du motif *façade* (décrit page 76), il est facile de transformer cet éditeur en un *Widget* tout en lui offrant la possibilité d'être étendu.

En cela, notre architecture permet de mettre en œuvre de manière opérationnelle le méta-modèle *slinky* associé au modèle de l'Arche [User Interface Developer's Workshop91] qui indique que suivant les besoins, un objet peut être placé dans la composante de présentation ou dans le contrôleur de dialogue. En encapsulant l'éditeur de graphe, le *Widget* appartient à la composante de présentation tandis qu'en ouvrant le *Widget*, ses fonctions de contrôle se déplacent dans le contrôleur de dialogue et peuvent être raffinées.

Une estimation quantitative est donnée en table 1.2.

Chapitre 2

Éditeur graphique interactif

Ce deuxième exemple décrit un éditeur permettant de dessiner interactivement en utilisant une souris ou une tablette graphique dotée d'un stylet délivrant des informations de position et de pression. Le noyau sémantique est réduit à la gestion d'une structure graphique arborescente.

Cet éditeur est actuellement utilisé dans le système commercial TicTacToon de dessin animé assisté par ordinateur [Fekete et al.95]. Il répond donc à des besoins particuliers de performance et de fonctionnalités qui nécessitent un usage élaboré du modèle.

Nous aurions pu nous contenter de présenter une version simplifiée de notre éditeur, mais nous pensons que la force d'un modèle architectural vient de sa capacité à répondre de façon propre à des problèmes réels plutôt que des exemples choisis.

2.1 L'animation sans papier dans TicTacToon

Les studios d'animation professionnels suivent une organisation rigoureuse pour la fabrication de dessins animés car la réalisation d'une série ou d'un film de long métrage nécessite des dizaines de milliers de dessins et des dizaines de personnes collaborant pendant plusieurs mois. La grande majorité des studios n'utilisent les ordinateurs que pour des tâches secondaires, la totalité du travail d'animation étant fait sur papier. Dans TicTacToon, la chaîne de fabrication est entièrement informatique, les tâches traditionnelles sont transposées sur un ordinateur.

Pour obtenir des gains de productivité sensibles, TicTacToon autorise la réutilisation de parties de séquences, gérées par une base de données. Pour tirer pleinement profit des réutilisations, les dessins sont gardés sous une forme indépendante de leur résolution. Ils peuvent donc être réutilisés à différentes échelles et dans différentes perspectives. Cette caractéristique a des implications importantes

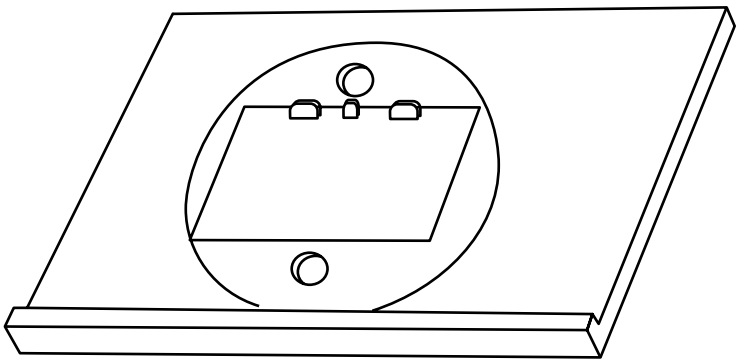


FIG. 2.1 - *Table lumineuse utilisée traditionnellement par les animateurs.*

sur l'éditeur d'animation, qui doit permettre la création interactive de dessins indépendants de la résolution.

Jusqu'à présent, les seuls éditeurs graphiques capables de répondre interactivement à un dessinateur professionnel étaient pixellaires et utilisaient un matériel dédié. La *PaintBox* de Quantel [Quantel inc.] est très utilisée en production vidéo et *StudioPaint 3D* de Alias [Alias Research Inc.a] est surtout utilisé en design. Le dessin pixellaire n'est pas indépendant de la résolution et ne permet donc pas de réutilisations partielles. TicTacToon utilise plusieurs algorithmes et structures de données graphiques originales, développées par le laboratoire de recherche parisien de Digital Equipment (PRL), pour autoriser le dessin vectoriel à main levée en temps réel [Pudet94], ainsi que son coloriage [Gangnet et al.89]. Tous les programmes spécifiques à l'exécution des tâches dans la chaîne de fabrication ont été réalisés par la société 2001 S.A. en utilisant notre bibliothèque graphique pour les éditeurs.

L'éditeur d'animation est une des composantes principales du système TicTacToon. Il permet à des animateurs professionnels de réaliser leurs dessins directement sur un ordinateur et reproduit les fonctionnalités de leur table lumineuse traditionnelle (voir figure 2.1).

Cette table est composée d'un disque de plastique translucide sur lequel est placée une barre possédant deux ou trois petites cales (barre à tenons) servant à maintenir une ou plusieurs feuilles de papier. Le disque translucide peut être tourné, à l'aide des trous placés en haut et en bas, pour amélioré le confort de l'animateur. Une lampe placée sous le disque de plastique peut être allumée afin que l'animateur voie par transparence tous les dessins placés dans la pile. Les animateurs vérifient la fluidité de leurs animations en parcourant rapidement les dessins de la pile (*flipper* dans le jargon technique).

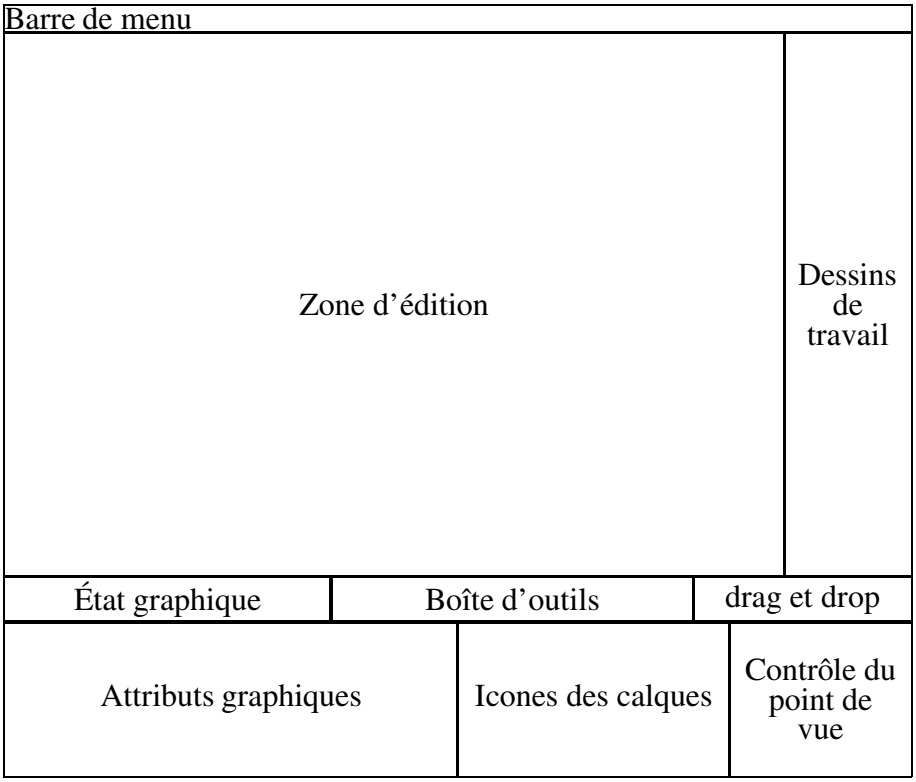
Une fois filmée, une animation est composée de 24 images par seconde, chaque image pouvant comporter plusieurs dessins (décor, éléments de décor et person-

nages). Pour paraître convenablement animée, l'action d'un personnage doit comporter une moyenne de 12 à 18 poses par seconde. Chaque pose du personnage doit être dessinée une fois, les systèmes d'interpolation automatique des poses ne permettent pas une qualité suffisante.

Ces caractéristiques imposent des contraintes sévères sur la productivité d'un animateur, et donc sur l'efficacité de l'éditeur et son confort.

2.2 Fonctionnalités de l'éditeur

L'éditeur de la figure 2.2 est composé des zones suivantes :



La barre de menu Comme dans la plupart des éditeurs, quatre menus sont disponibles permettant de manipuler les *fichiers*, d'appliquer les actions défaire, refaire, couper, coller, copier et détruire avec le menu *Edit*, de modifier la structure des objets graphiques sélectionnés en groupant, dégroupant, passant les objets devant ou derrière avec le menu *Structure*, et de modifier le point de vue.

Les deux menus *Edit* et *Structure* permettant d'accéder aux fonctions de manipulation indirecte.

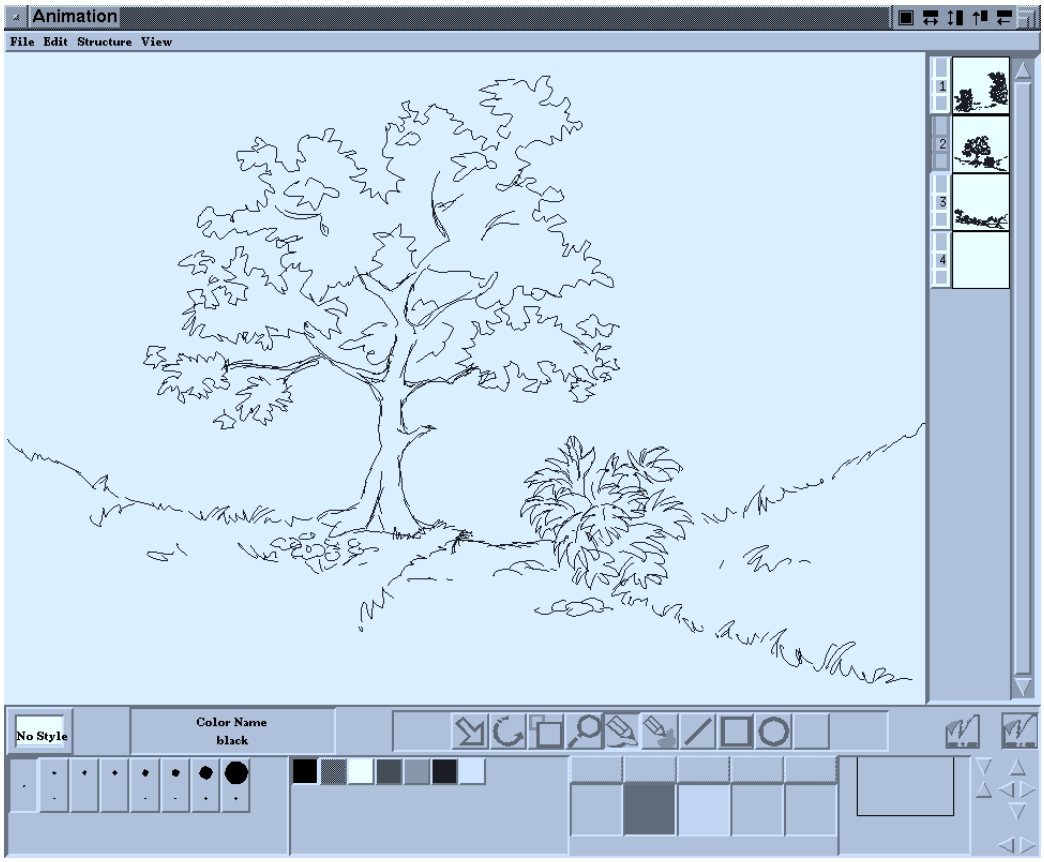


FIG. 2.2 - *Le programme d'édition d'animation.*

Zone de travail C'est la zone gérée par la pile et qui contient les couches décrites en § 2.3.

Dessins de travail La liste des icones représente tous les dessins avec lesquels un animateur peut vouloir travailler. Un seul dessin est édité à un moment donné. D'autres peuvent être placés en fond du dessin courant, comme sur la table lumineuse. Pour changer le dessin courant, il est possible de cliquer sur l'icone du dessin désiré ou d'utiliser les flèches bas et haut du clavier pour passer d'un dessin au suivant ou au précédent. Cette fonctionnalité sert aussi à *flipper*. La main dominante est utilisée pour tenir le stylet de la tablette et les touches du clavier sont utilisées par la main non dominante pour effectuer les actions les plus fréquentes : défaire/refaire, flipper, changer l'outil actif, tourner le point de vue ou le remettre droit. Un animateur peut donc regarder l'écran en permanence et rester concentré sur sa tâche.

État graphique Cette zone affiche l'état graphique courant ainsi que l'effet graphique utilisé le cas échéant. Un effet graphique peut être défini comme une liste nommée d'attributs graphiques qui peuvent être copiés dans l'état graphique.

La boîte d'outils Les outils, représentés sur la boîte à outils par des icones, sont de trois types (décrits du haut vers le bas) :

sélection/transformation géométrique : les trois premiers outils qui permettent de déplacer, tourner et redimensionner les objets graphiques ;

manipulation du point de vue : la fonction « zoom » ;

création d'objets graphiques : six formes différentes sont proposées : tracé à main levée, tracé de contour à main levée, création d'une ligne droite, d'un rectangle, d'une ellipse ou d'un texte.

Drag et drop Pour communiquer entre les éditeurs de TicTacToon, plusieurs sortes d'objets peuvent être déplacées par *drag et drop*. Cette zone permet de recevoir ou d'envoyer ces objets. Nous ne décrirons pas cette zone plus en détail ici.

Attributs graphiques La zone de contrôle des attributs graphiques affiche la brosse sélectionnée et la couleur de contour de l'état graphique courant. L'utilisateur peut modifier la brosse ou la couleur en cliquant sur un bouton dans la boîte.

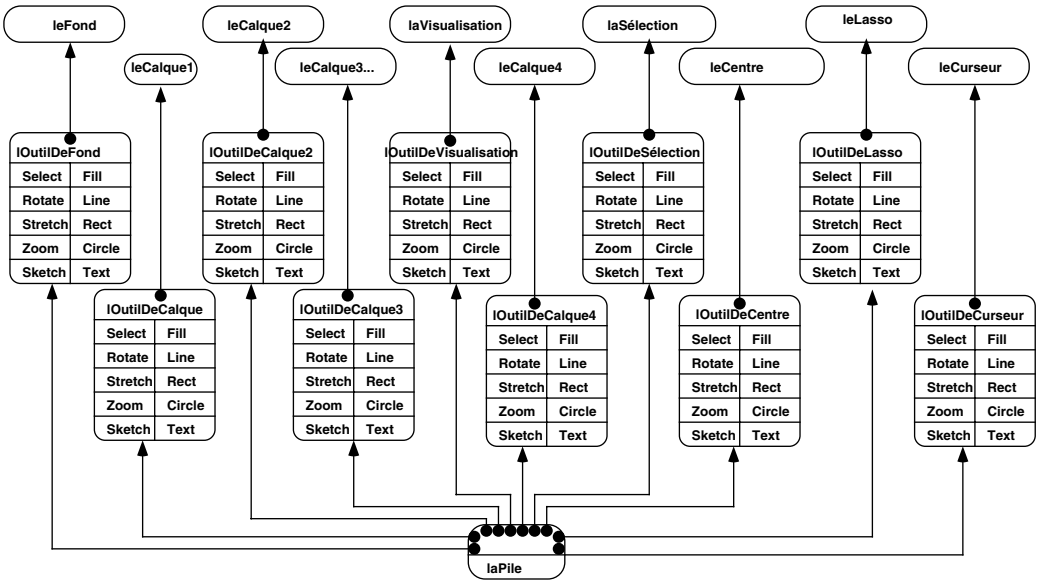


FIG. 2.3 - *Couches utilisées par l'éditeur d'animation*

Icones des calques Pour placer un dessin en calque, il suffit de le faire glisser sur un des icones. Cinq couches sont disponibles : quatre en dessous de la couche de visualisation passive et une au-dessus. L'icone le plus à gauche représente la couche de calque la plus profonde. La couleur de fond de ces icones représente la couleur dans laquelle le dessin est affiché dans sa couche.

Contrôle du point de vue Le contrôle du point de vue se fait à l'aide d'un *Panner* et de six flèches : quatre pour les directions du plan et deux pour grossir ou diminuer le zoom. Nous utilisons un *Panner* plutôt que des barres de défilement car le point de vue peut être tourné. Dans ce cas, le rectangle du *Panner* est tourné.

2.3 Les couches

Les couches de l'éditeur sont représentées en figure 2.3. Deux couches nouvelles apparaissent ici : **Calque** et **Centre**. Elles sont utilisées respectivement pour gérer les dessins placés sous le dessin courant (« sous le calque ») et pour afficher le centre de la rotation et du changement de taille lorsque leur outil est actif.

Nous allons maintenant décrire les particularités des couches et des outils.

```
type CoucheVPGraphique = sous_classe de CoucheVP;
début
    g_glyph: ListeDe(GraphicGlyph);
    historique: ListeDe(Command);
fin;
```

2.3.1 Le fond

Le fond a été choisi pour des raisons ergonomiques, un animateur professionnel devant dessiner pratiquement huit heures par jour. La couleur est unie et gris neutre.

Les dessins réalisés étant indépendant de la résolution, l'éditeur permet de travailler sur une surface de dessin beaucoup plus grande que les feuilles de papier traditionnelles. Certains animateurs étant perdus sans indication de taille, ils peuvent afficher sur le fond un graphique rappelant la limite de la feuille traditionnelle et la barre à tenons. Avec un peu d'habitude, ces repères deviennent inutiles et même gênants.

2.3.2 La couche de visualisation passive

Cette couche implante un modèle graphique plus riche en attributs que celui d'InterViews. La structure de données qu'elle gère est un GDA de **GraphicGlyph**s. Les fonctions de manipulation sont décrites en figure 2.3.2. Par convention, lorsqu'une fonction modifie la structure de données, son appel est fait à partir d'un objet de type **Command** (motif de conception décrit page 78) qui a auparavant sauvegardé un état permettant à la fonction d'être annulée.

2.3.2.1 Modèle graphique

Le modèle graphique utilisé par les dessins d'animation est 2D réaliste. La surface de dessin gère une transformation affine de visualisation en 2D. Les primitives graphiques sont de deux types : vectorielles et pixellaires. Elles utilisent des attributs graphiques spécifiques.

Les primitives vectorielles sont décrites par une courbe de Bézier de degré inférieur ou égal à 7 (InterViews et PostScript limitent le degré à 3). Dans la pratique, les courbes de Bézier sont généralement produites algorithmiquement par lissage d'échantillons issus d'une tablette ou d'une souris et sont composées de segments de degré 5 qui offrent un bon compromis entre le temps de lissage et le temps de réaffichage.

Les primitives pixellaires sont décrites sous la forme d'un tableau de pixels dont la résolution est connue. Chaque pixel a une couleur et un niveau de transparence.

Les attributs graphiques spécifiques sont les suivants :

Brosse : une brosse a une forme convexe décrite par une courbe de Bézier, et une taille minimale et maximale (utilisée avec le *profil de pression* décrit ci-dessous). Lorsqu'une courbe est *tracée* avec cette brosse, les pixels allumés sont ceux qui se trouvent à l'intérieur de la forme de la brosse lorsque celle-ci se déplace sur la courbe.

Profil de pression : lorsqu'une courbe est tracée à l'aide d'un stylet retournant des informations de pression, la trajectoire est lissée et convertie en liste de segments de courbes de Bézier tandis que la pression appliquée au stylet est transformée en une fonction qui associe une valeur de pression (entre 0 et 1) à une valeur d'abscisse curviligne normalisée (le point à mi-chemin sur la courbe aura une abscisse curviligne normalisée de 0,5). Lorsqu'un profil de pression est spécifié pour tracer une courbe avec une brosse, alors la taille de la brosse est modulée par la pression le long de la courbe. Cette modulation est linéaire entre la taille minimale de la brosse lorsque la pression est nulle et la taille maximale lorsque la pression est à 1 (voir figure 2.4).

Annotation longitudinale : fonction de variation de la couleur et de la transparence le long du tracé d'une courbe (voir figure 2.4) ;

Annotation transversale : fonction de variation de la couleur et de la transparence sur l'épaisseur d'une courbe (voir figure 2.4) ;

Transparence : lorsqu'une primitive est affichée, sa transparence intrinsèque est combinée avec l'attribut de transparence.

Flou : intensité du flou appliqué aux primitives graphiques lorsqu'elles sont calculées.

Ce modèle est utilisé pour le calcul final des images d'animation et par l'éditeur de décors. Dans l'éditeur d'animation, le modèle graphique est relâché pour obtenir les performances nécessaires aux animateurs. En particulier, la transparence, le flou et les annotations sont ignorés.

Le modèle graphique est implanté par traduction vers le modèle du système de fenêtrage. Cette traduction est coûteuse car le dessin d'un trait modulé par un profil de pression et dessiné avec une brosse est implanté par le remplissage du polygone décrivant le contour du trait. Ce polygone est parfois composé de plusieurs milliers de segments qui doivent être transmis au serveur X à travers un canal de communication sériel. De plus, l'algorithme de remplissage de polygone de X ne colorie que les pixels dont le centre est à l'intérieur du polygone, la connexité des traits n'est pas garantie lorsque l'épaisseur devient plus petite qu'un pixel et ne traverse pas un centre de pixel. La seule manière d'assurer le connexité d'un trait est de remplir le

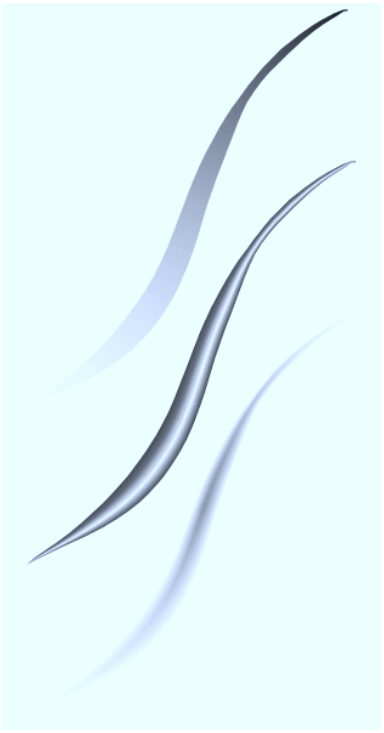


FIG. 2.4 - *Traits modulés en pression et avec une annotation de couleur longitudinale en haut, transversale au centre et les deux en bas.*

polygone puis de forcer les pixels sur les bords à être coloriés en les traçant, ce qui implique deux transferts au serveur X et des performances inacceptables lorsque le programme est utilisé à partir d'une machine distante.

Les dessins réalisés avec l'éditeur d'animation sont généralement faits pour être coloriés lors d'une étape suivante de la chaîne et n'utilisent pratiquement que des traits de couleur unis. Les attributs graphiques ignorés sont donc généralement inutilisés à ce stade. Nous avons donc dérivé le **Canvas** en un **RenderCanvas** qui implante le modèle graphique complet et le traduit en primitives graphiques appliquées au **Canvas** de la pile.

2.3.2.2 Fonctions spécifiques

La couche de visualisation passive gère l'affichage et le réaffichage d'un GDA de **GraphicGlyph**s. Pour optimiser le réaffichage, elle utilise le mécanisme décrit en deuxième partie, § 2.6.4. Elle offre toutes les fonctions de manipulation de la structure graphique en prenant soin de déclencher des réaffichages sur les zones modifiées.

2.3.3 Les calques

Les calques sont des couches qui affichent l'équivalent des dessins apparaissant par transparence sur une table lumineuse. Leur contenu graphique présente deux spécificités :

- il utilise un modèle graphique relâché ; chaque couche n'affiche que les traits de contour des objets graphiques, et cela avec une seule couleur pour que le dessin des couches ne soit pas confondu avec le dessin édité ;

- une transformation affine peut être appliquée par la couche au dessin pour qu'il paraisse déplacé, tourné ou déformé.

Pour gérer cet affichage, le **RenderCanvas** est dérivé en un **SousCanvas** qui ignore la fonction de remplissage de forme et applique la fonction de tracé en utilisant une couleur unique. C'est une utilisation à trois niveaux du motif *Decorator*, comme le décrit la figure 2.5.

La couche implantant le calque partage de nombreuses caractéristiques avec la couche de visualisation principale ; elle en hérite et, comme elle, gère une structure de **GraphicGlyph**s s'affichant sur un **RenderCanvas**.

Cet héritage sert à réutiliser des caractéristiques partagées par les deux couches, mais aussi à appliquer des outils de manipulation directe sur les objets des deux couches, comme nous le décrivons en 2.4.

```
type Canvas = classe;
début
    proc move_to(in Point);
    proc line_to(in Point);
    proc curve_to(in Point, in Point, in Point);
    proc close_path();
    proc fill(in Color,in FillRule);
    proc stroke(in Color,in Brush);
    proc clip(in FillRule);
    proc image(in Raster,in Point);
    proc character(in CharCode,in Font,in Point);
fin;

type RenderCanvas = sous_classe de Canvas;
début
    canvas: Canvas;

    proc curve_to(in ListeDe(Point));
    proc state(in GraphicState);
    proc stroke();
    proc fill();
    proc clip();
fin;

typeSousCanvas = sous_classe de RenderCanvas;
début
    canvas: RenderCanvas;
fin;
```

FIG. 2.5 - *La signature du modèle graphique du **Canvas**, du **RenderCanvas** et du **SousCanvas** dans l'éditeur d'animation.*

2.3.4 Gestion de la sélection

Comme nous l'avons décrit en deuxième partie, § 2.3.5, l'écho de la manipulation directe est fait par des objets de type **Fantôme**. Dans l'éditeur d'animation, ils sont spécialisés en **FantômeGraphique** pour utiliser la géométrie de leur **GraphicGlyph** de référence plutôt que d'utiliser leur boîte englobante.

Les fantômes s'affichent en utilisant un mode involutif. Le système TicTacToon met toujours en place une table des couleurs qui évite le problème, décrit en deuxième partie, § 2.6.5, de disparition des objets sélectionnés lorsqu'ils sont placés au-dessus de certaines couleurs de fond.

La forme de la sélection étant la même que celle de la manipulation directe, leurs fonctions (décrites en deuxième partie, § 2.3.4 et § 2.3.5) sont unifiées. Les *fantômes* sont utilisés aussi à la place des poignées. La structure graphique du *fantôme* est spécialisée comme suit :

```
type FantômeGraphique = classe;
début
    chemin: CheminObjet;
    couche: CoucheVPGraphique;
    transform: Transformer;

    fn intercepte(Événement): Booléen;
    proc dessine(inout Canvas,in Région);
    proc commence(in Transformer);
```

```
type CoucheSélectionGraphique = sous_classe de CoucheSélection;
début
    fantômes: ListeDe(Fantôme);

    proc ajoute_fantôme(in Couche, in Entier);
    proc détruit_fantôme(in Entier);
    proc commence_transform(in Transformer);
    proc continue_transform(in Transformer);
    proc termine_transform(in Transformer);
    proc oublie_transform();
fin;
```

FIG. 2.6 - *Fonctions spécifiques de la couche de gestion de la sélection du programme d'animation.*

```
    proc continue(in Transformer);
    proc termine(in Transformer);
    proc oublie();
    fn a_commencé() : Booléen;
fin;
```

Pour se dessiner, il utilise la géométrie du **GraphicGlyph** dont il visualise la sélection. Il gère aussi une transformation homogène 3×3 qu'il applique à la géométrie du **GraphicGlyph** pendant la manipulation directe. La figure 2.6 décrit les fonctions spécifiques de la couche.

2.3.5 La couche de gestion du centre des transformations

La couche de gestion du centre de rotation et de redimensionnement sert à visualiser un point qui est utilisé par l'outil de rotation et l'outil de redimensionnement comme centre. Lorsque ces outils ne sont pas actifs, le point n'apparaît pas et la couche est vide. Lorsqu'un de ces outils est actif, la couche affiche un carré de taille constante qui visualise le centre et est réactif aux événements positionnels émis dans sa région.

2.3.6 La couche de retour d'information lexicale

La couche de gestion du retour lexical est spécialisée pour gérer la trace. Les systèmes de fenêtrage classiques ne sont pas faits pour gérer cette trace, contrairement aux systèmes spécifiques des assistants personnels comme le Newton de

```
type CoucheTrace = sous_classe de CoucheRIL;
début
    trace_graphique: ListeDe(CourbeDeBézier);
    trace_à_dessiner: Entier;
    état_graphique: GraphicState;

    proc ajoute_courbe(in CourbeDeBézier);
    proc efface_trace();
```

FIG. 2.7 - *Fonctions spécifiques à la couche de retour d'information lexicale gérant la trace.*

Apple [Apple Computer Incorporated93]. L'implantation de la gestion de la trace dans la couche utilise les fonctions de la figure 2.7.

Le modèle graphique de cette couche est celui du système de fenêtrage. Elle gère une liste d'affichage composée de courbes de Bézier qui sont affichées avec les attributs de l'état graphique. Dans cette couche, l'état graphique ne contient qu'une couleur et parfois une épaisseur de trait. La couche se contente donc d'afficher une liste de courbes de Bézier remplies d'une couleur ou tracées avec une couleur et une épaisseur fixes. Elle ne gère pas l'écho de dispositifs ni d'un curseur de texte car le mécanisme de gestion de la trace est différent de celui de la gestion des curseurs pour deux raisons :

- tandis que la trace est modale, les curseurs sont persistants ;
- lorsqu'une trace existe, sa vitesse de mise à jour doit être la plus grande possible, parfois au détriment du réaffichage des autres couches.

C'est pourquoi nous avons créé l'attribut de couche *transitoire* qui autorise le mécanisme de composition à sacrifier l'intégrité d'affichage des autres couches pendant le réaffichage de celle-ci. Si nous avions eu besoin d'afficher un curseur, nous aurions créé une couche supplémentaire.

2.4 Les outils

Chaque outil peut se décomposer en deux parties : la gestion de la manipulation directe et, lorsqu'elle est terminée, le déclenchement de l'action qui en résulte. Les actions, lorsqu'elles modifient le noyau sémantique, sont encapsulées dans un objet de type **Command** (voir le motif de conception page 78) afin de les rendre réversible. Dans notre éditeur, seules les actions sur le point de vue (zoom ou pan) ne sont pas réversibles.

2.4.1 Les commandes

Certaines actions sémantiques sont déclenchées par manipulation indirecte (coupe, colle, etc.) ; il existe donc des objets **Command** qui ne sont pas utilisés par les outils. Plusieurs outils utilisent le même objet **Command**, en particulier le déplacement, la rotation et le redimensionnement sont implantés par une même commande qui applique une transformation affine 2D à la sélection. Une commande est décrite comme une fonction et ses arguments. Les commandes de l'éditeur d'animation sont :

- Insère(CoucheVPGraphique, ListeDe(GraphiqueGlyph),GlyphIndex) : insère une liste de **GraphicGlyph**s dans la couche de visualisation passive à un index donné ;
- Ajoute(CoucheVPGraphique, ListeDe(GraphiqueGlyph)) : ajoute une liste **GraphicGlyph**s à la fin de la couche de visualisation passive ;
- Retire(CoucheVPGraphique, ListeDe(GlyphIndex)) : retire les **GraphicGlyph**s dont l'index est donné de la couche de visualisation ;
- Transforme(CoucheVPGraphique, ListeDe(GlyphIndex), Transformer) : applique la transformation à tous les **GraphicGlyph**s de la couche de visualisation dont l'index est donné ;
- PasseEnHaut(CoucheVPGraphique, ListeDe(GlyphIndex)) : déplace à la fin de la liste d'affichage (en haut des objets graphiques affichés) tous les **GraphicGlyph** dont l'index est donné ;
- PasseEnBas(CoucheVPGraphique, ListeDe(GlyphIndex)) : déplace au début de la liste d'affichage (en bas des objets graphiques affichés) tous les **GraphicGlyph** dont l'index est donné ;
- ChangeAttribut(CoucheVPGraphique, ListeDe(GlyphIndex),AttributGraphique), remplace l'attribut graphique pour tous les **GraphicGlyph** dont l'index est donné ;
- Grouper(CoucheVPGraphique, ListeDe(GlyphIndex)) : crée un groupe, c'est-à-dire crée un **GraphicGlyph** composite, place tous les **GraphicGlyph** de la couche de visualisation dont l'index est donné dans ce **GraphicGlyph** composite en les retirant de la couche et ajoute le composite à la fin de la liste d'affichage de la couche de visualisation passive ;
- Dégrouper(CoucheVPGraphique, ListeDe(GlyphIndex)) : pour chaque **GraphicGlyph** dont l'index est donné, s'il s'agit d'un composite, le retirer et insérer à sa place tous les **GraphicGlyph** dont il est composé.

Ces commandes sont créées par des *Widget*s de manipulation indirecte ou lors de l'étape finale des manipulations et sont ensuite insérées dans l'historique des actions liées à une couche de visualisation passive.

2.4.2 Manipulation indirecte

Les menus permettent d'activer les commandes pour couper, coller, copier et détruire, qui sont aussi associées à des touches accélératrices destinées aux utilisateurs entraînés. Les commandes associées (*Ajoute*, *Retire*) agissent sur les objets graphiques sélectionnés de la couche de visualisation passive.

La partie inférieure de la fenêtre d'application représente l'état graphique courant et une palette de couleurs et de brosses sélectionnables. Un état graphique global est maintenu par l'application et est modifié par la sélection d'une couleur ou d'une brosse dans ces palettes. Cette modification se propage aussi sur les objets sélectionnés, activant la commande *ChangeAttribut*.

2.4.3 Outil de sélection/déplacement

Le schéma UAN 2.8 décrit l'outil de sélection et de déplacement. Il diffère des autres schémas de sélection/déplacement pour deux raisons : la couche de sélection est aussi la couche de manipulation directe et la couche de calque peut aussi être manipulée. Tandis que les objets graphiques de la couche de visualisation passive peuvent être sélectionnés sans être manipulés, l'objet (c'est un groupe) de la couche de calque ne peut pas être sélectionné sans être manipulé. Lorsque l'utilisateur clique sur une partie d'un calque, le dessin tout entier du calque sera déplacé. Le reste des manipulations est identique aux autres outils de sélection/déplacement.

2.4.4 Autres outils de transformations géométriques

Les autres outils de transformations (rotations et changement de taille) dérivent de l'outil de sélection/déplacement. Pour l'outil de gestion de la sélection et de la manipulation directe, seules les fonctions *commence* et *continue* sont redéfinies. Elles calculent la transformation adéquate à partir du point de départ et du point courant.

Ces transformations sont toutes les deux définies par rapport à un centre. La détermination du centre est généralement implicite dans les éditeurs schématiques. Par exemple, Canvas et Idraw prennent le centre du rectangle englobant les formes sélectionnées comme centre implicite à ces transformations. Illustrator utilise comme centre de rotation la position du pointeur lorsque la manipulation directe commence, obligeant l'utilisateur à éloigner rapidement son pointeur de sa position initiale pour obtenir une bonne précision dans le contrôle de l'angle. Nous aurions pu implanter une de ces deux méthodes de détermination du centre mais avons préféré gérer un centre de transformation explicite dans une couche particulière afin qu'il reste mémorisé et soit utilisable avec plusieurs objets, le cas échéant sur plusieurs dessins.

Couche	Contexte	Événement	Action locale	Action transmise
Fond		$\sim [x, y]\mathrm{M} \downarrow$		**VP** désélectionne_tout() **Sélection** détruit_poignées() **Rectangle** commence(x, y)
Calque		$\sim [o]\mathrm{M} \downarrow$		**VP** désélectionne_tout() **Sélection** fantôme(o) commence(x, y)
VP		$\sim [o]\mathrm{M} \downarrow$	sélectionne(o)	**Sélection** fantôme(o)
Sélection		$\sim [o]\mathrm{M} \downarrow$	nop()	
		$\sim [o]*$	commence(x, y)	
	a_commencé()	$\sim [x', y']*$	continue(x', y')	
	a_commencé()	$\sim [x'', y'']\mathrm{M} \uparrow$	termine(x'', y'')	**VP ou Calque** Transforme(dx, dy)
Rectangle	*a_commencé()*	$\sim [x', y']*$	continue(x', y')	
	a_commencé()	$\sim [x'', y'']\mathrm{M} \uparrow$	termine(x'', y'')	**VP** sélectionne(rectangle()) **Sélection** fantômes(rectangle())

FIG. 2.8 - *Sélection et déplacement dans l'éditeur d'animation.*

```
type OutilRIL = sous_classe de OutilRILTrace;
début
        positions: ListeDe(Point);
        pressions: ListeDe(Réel);
        dernière_date: Date;
        lisseur: LisseBézier;

        proc commence_trace(dans Événement);
        proc continue_trace(dans Événement);
        proc termine_trace(dans Événement);
        fn glyph(): GraphicGlyph;
fin;
```

FIG. 2.9 - *L'outil de gestion de la trace sur la couche de retour d'information lexical.*

Un objet graphique, représenté par un petit carré, localise le centre. Ce carré peut être déplacé avec le pointeur car la couche qui le gère est placée au dessus de la couche de sélection. Un accélérateur permet, en appuyant sur la touche **`Shift`** tout en cliquant, de placer le centre sous le pointeur.

2.4.5 Dessin à main levée

Une tablette graphique est utilisée pour dessiner. Pendant que l'utilisateur manipule le stylet sur la tablette, la couche de gestion du retour lexical affiche la trace en respectant le modèle graphique de l'application : l'épaisseur instantanée est donnée par la brosse et modulée par la pression. Une fois le trait terminé, la trace est effacée et un **GraphicGlyph** est créé, ajouté à la couche de visualisation passive et redessiné. L'état graphique du **GraphicGlyph** contient la courbe de Bézier calculée par lissage des positions du stylet de la tablette, son profil de pression et la copie des autres attributs graphiques stockés dans la couche de gestion du retour d'information lexical.

Cet outil utilise une bibliothèque graphique particulière pour traiter la trajectoire de la tablette, assurer le retour d'information et transformer la trace en une courbe de Bézier. Nous commençons donc par décrire cette bibliothèque en insistant sur l'adaptation que nous avons dû lui faire subir. Ensuite, nous décrivons la stratégie utilisée pour gérer interactivement la trace, puis nous donnons le schéma UAN de l'outil.

```
type TLisseBézier = classe;
début
    tolérance: Réel;

    proc lisse_échantillons(ListeDe(Point)): CourbeDeBézier;
    proc contour(CourbeDeBézier,Profil,Brosse): CourbeDeBézier;
    proc contour_rapide(Polygone,Profil,Brosse): Polygone;
    proc linéarise(CourbeDeBézier): Polygone;
fin;
```

FIG. 2.10 - *La signature de la bibliothèque de lissage des courbes.*

2.4.5.1 Bibliothèque de gestion des courbes de Bézier

L'outil de gestion de la trace doit afficher interactivement un trait modulé en épaisseur qui doit être identique — où du moins très semblable — à celui affiché par l'objet graphique qui sera construit après lissage de la courbe et redessiné. Nous avons utilisé la bibliothèque graphique (décrite en 2.10) qui a été conçue et réalisée lors d'un travail indépendant [Pudet94]. Cette bibliothèque définit les fonctions suivantes :

***lisse_échantillons* :** calcule, à partir d'une trajectoire décrite par une liste de points, une courbe de Bézier qui approxime la trajectoire, c'est-à-dire telle que la distance entre la courbe et la trajectoire soit toujours inférieure à la tolérance.

***contour* :** calcule la courbe de Bézier définissant le contour d'un trait à partir de sa trajectoire (décrite par une courbe de Bézier), de son profil de pression et d'une brosse.

***contour_rapide* :** calcule le polygone définissant le contour d'un trait à partir de sa trajectoire (décrite par une liste de points), son profil de pression et sa brosse. Cette fonction était interne à la bibliothèque originelle et servait à l'implantation de *contour*. Elle a été rajoutée pour gérer le retour d'information interactif durant la phase de tracé. Nous montrons en § 2.4.5.3 son utilisation.

***linéarise* :** transforme une courbe de Bézier en une suite de segments de droite qui approxime la courbe à la tolérance près.

2.4.5.2 Stratégie de gestion interactive de la trace

L'algorithme de suivi de la trace est différent du suivi d'un dispositif. Pour que la trajectoire de la trace n'apparaisse pas cassée, il est essentiel de tenir compte de toutes les positions parcourues par le dispositif. Le débit maximal d'une tablette est de l'ordre de 2000 événements par seconde lorsque le stylet se déplace rapidement. Même sur les machines les plus rapides, le serveur X ne peut pas acheminer des événements à un client à cette vitesse sans accumuler du retard. Pour éviter ce retard, nous utilisons les mécanismes de *motion hint* et de *motion history* disponibles dans X et décrits en première partie, § 2.3 page 16. Notons que les autres systèmes de fenêtrage obligent aussi à utiliser un mécanisme spécifique pour obtenir une trajectoire complète.

2.4.5.3 Gestion des événements

Le schéma UAN III.5 décrit le dessin à main levée. La couche de retour d'information lexicale intercepte tous les événements du pointeur pendant de dessin et ajoute le **GraphicGlyph** à la fin de la manipulation. Les fonctions *commence_trace*, *continue_trace*, *termine_trace* et *glyph* sont décrites par le pseudocode de la table 2.1.

Tâche III.5 Dessin à main levée dans l'éditeur d'animation.

Couche	Contexte	Événement	Action locale	Action transmise
VP				
RIL		$\sim [x, y, p]\mathrm{M}\downarrow$	commence_trace(x, y, p)	
		$\sim [x', y', p']*$	continue_trace(x', y', p')	
		$\sim [x'', y'', p'']\mathrm{M}\uparrow$	termine_trace(x'', y'', p'')	**VP** Ajoute(glyph())

2.4.6 Création de formes géométriques

Pour la création de formes géométriques, l'interaction est gérée à partir de la couche de retour d'information lexicale. Les objets créés sont ensuite ajoutés à la fin de la liste des objets de la couche de visualisation.

Les trois outils de création de forme géométrique sont très semblables et dérivent tous de l'outil de création de ligne (voir figure 2.4.6). Les fonctions *contraint* et *courbe* encapsulent les différences entre les outils. La première contraint l'angle de la ligne à être multiple de 45°si la touche `Shift` du clavier est enfoncée pendant la manipulation. Sa spécialisation consiste à contraindre le rectangle à être un

```
proc commence_trace(e: Événement);
début
    var p: Profil,
        b: Brosse;
        courbe: Polygone;

    positions.vider();
    pressions.vider();
    dernière_date := e.date;
    b := couche.état_graphique.brosse();
    p := créer Profil(e.position, e.pression);
    positions.ajoute(e.position);
    pressions.ajoute(e.pression);
    courbe := lisseur.contour_rapide(e.position,p,b);
    couche.ajoute_courbe(courbe);
    détruit p;
fin;

proc continue_trace(e: Événement);
début
    var p: Profil,
        b: Brosse,
        h: HistoriqueDeDispositif,
        i: Entier,
        courbe: Polygone;

    b := couche.état_graphique.brosse();
    i := longueur(positions);
    h := e.dispositif.historique(dernière_date,e.date);
    dernière_date := e.date;
    Ajouter les positions de l'historique à la liste positions
    Ajouter les pressions de l'historique à la liste pressions
    p := créer Profil(sous_liste(positions, i-1,…),
                      sous_liste(pressions, i-1,…));
    courbe := lisseur.contour_rapide(sous_liste(pressions,
                                                i-1,
                                                …),
                                     p,b);
    couche.ajoute_courbe(courbe);
    détruit p;
fin;

proc termine_trace(e: Événement);
début
    continue_trace(e);
    efface_tout();
fin;

fn glyph(): GraphicGlyph;
début
    var p: Profil,
        état: ÉtatGraphique,
        courbe: CourbeDeBézier;

    état := créer ÉtatGraphique(couche.état_graphique);
    p := créer Profil(positions,pressions);
    état.profil := p;
    courbe := lisseur.lisse_échantillons(pressions);
    état.courbe := courbe;
    retourne créer GraphicGlyph(état);
fin;
```

TAB. 2.1 - *Pseudo-code des fonctions de l'outil de gestion de la trace dans l'éditeur d'animation.*

```
type RILLigne = sous_classe de RIL;
début
        départ: Point;
        dernier: Point;

        proc commence(in Événement);
        proc continue(in Événement);
        proc termine(in Événement);
        fn contraint(Événement): Point;
        fn courbe(): CourbeDeBézier;
        fn glyph(): GraphicGlyph;
fin;
```

carré ou l'ellipse à être un cercle. La seconde crée la courbe de Bézier contenant une ligne à partir des premier et dernier points. Sa spécialisation crée un rectangle à partir des deux points ou un cercle dont le centre est le premier point et le rayon est spécifié par le second. Le schéma UAN de l'outil est donné en III.6 et le pseudo-code pour gérer la ligne est donné dans la table 2.2.

Tâche III.6 Création d'une ligne dans l'éditeur d'animation.

Couche	Contexte	Événement	Action locale	Action transmise
VP				
RIL		$\sim [x, y]\mathrm{M}\downarrow$	commence(x, y)	
		$\sim [x', y']*$	continue(x', y')	
		$\sim [x'', y'', p'']\mathrm{M}\uparrow$	termine(x'', y'')	**VP** Ajoute(glyph())

2.4.7 Création de texte

Dans cet éditeur, le texte est saisi dans une boîte de dialogue qui permet aussi de spécifier la police de caractères et sa taille. Lors de l'acquittement, une structure graphique est créée et ajoutée à la fin de la liste des objets de la couche de visualisation. La structure graphique est un groupe de structures graphiques, chacune étant un caractère défini par sa courbe de Bézier.

Cette gestion du texte est simpliste car l'objet créé perd sa sémantique spécifique et devient un objet graphique quelconque. Il est impossible de modifier une partie du texte une fois celui-ci créé.

```
proc commence(e: Événement);
début

    premier := e.position;
    continue(e);
fin;

proc continue(e: Événement);
début
    dernier := contraint(e);
    couche.efface_tout();
    couche.ajoute_courbe(courbe());
fin;

proc termine(e: Événement);
début
    continue_trace(e);
    couche.efface_tout();
fin;

fn contraint(e: Événement): Point;
début
    var p: Point,
        d: Dispositif,
        v: Vecteur;

    p := e.position; d := e.dispositif_clavier;
    si non d.touche_enfoncée(Shift) alors
        retourne p;
    fin siv := p- premier;
    si v == vecteur_nul alors
        retournep;
    fin si;
    contraint l'angle de ligne à être multiple de 45°.
    retourne p;
fin;

fn courbe(): CourbeDeBézier;
début
    var c: CourbeDeBézier;

    c := créerCourbeDeBézier();
    c.ligne(premier, dernier);
    retourne c;
fin;

fn glyph(): GraphicGlyph;
début
    var état: ÉtatGraphique;

    état := créerÉtatGraphique(couche.état_graphique);
    état.courbe := courbe();
    retourne créer GraphicGlyph(état);
fin;
```

TAB. 2.2 - *Pseudo-code des fonctions de l'outil de gestion de la trace dans l'éditeur d'animation.*

Une gestion plus sophistiquée nécessiterait la définition d'un outil gérant un point d'insertion et toutes les fonctions spécifiques au texte. Il est intéressant de se souvenir que l'éditeur Illustrator de Adobe, dans sa version initiale, offrait la même possibilité. Les versions plus récentes implantent des outils plus sophistiqués de gestion du texte. Notre architecture nous permettrait de réaliser un tel outil, mais la demande n'a pas encore été faite par les animateurs, qui utilisent le texte de façon très occasionnelle.

Nous ne donnons pas le schéma UAN de l'outil, qui se contente de réagir lors d'un clique, affiche la boîte de dialogue et ajoute un **GraphicGlyph** dans la couche de visualisation passive à la position du clique.

2.4.8 Outils de modification du point de vue

L'outil de modification du point de vue n'utilise que la couche de gestion du rectangle de façon triviale (nous omettons donc le schéma UAN). Lorsque le rectangle est nul, un zoom arrière est appliqué, d'une valeur arbitraire (1,2) et centré sur le rectangle. Lorsque le rectangle n'est pas nul, la portion de la surface virtuelle placée dans le rectangle est agrandie pour tenir au mieux sur la vue, c'est-à-dire que deux rapports de zoom sont calculés : le premier pour que la largeur du rectangle tienne sur la largeur de la fenêtre et le second pour que la hauteur du rectangle tienne sur la hauteur de la fenêtre. Le facteur de zoom permettant de voir toute la largeur du rectangle et toute sa hauteur est alors choisi et appliqué au point de vue.

2.5 Spécialisation de la composition

Nous utilisons les techniques d'optimisations décrites en deuxième partie, § 2.6 pour améliorer la réponse interactive de l'éditeur. La surface virtuelle d'affichage utilise le *double buffering* pour améliorer le confort visuel du réaffichage. La couche de visualisation passive n'utilise pas de cache mais optimise l'ajout d'objets graphiques en ne réaffichant pas les objets placés en-dessous lorsque les conditions décrites en deuxième partie, § 2.6.3.1 sont vérifiées.

La couche de gestion du rectangle et de la sélection sont involutives ; celle qui gère la trace est transitoire. Quand le choix se présente, l'effacement des couches involutives est toujours préféré au redessin de la couche de visualisation (qui n'utilise pas de cache). La trace est dessinée directement sur la fenêtre et non sur le *back buffer*, ce qui améliore sensiblement le temps de réponse pour deux raisons : une copie du *back buffer* vers la fenêtre est évitée à chaque événement, et les accélérateurs graphiques ne fonctionnent généralement que dans la mémoire vidéo, c'est-à-dire sur une fenêtre.

Toutes ces optimisations permettent à l'éditeur de répondre suffisamment rapidement pour ne pas gêner les animateurs.

2.6 Spécialisation du rendu pour la visualisation

La structure de l'éditeur d'animation est réutilisée dans l'éditeur de décors avec très peu de modifications. En dehors de quelques *Widget*s particuliers destinés à contrôler les valeurs des attributs graphiques utilisés pour le décor, la modification la plus importante est l'utilisation d'une surface virtuelle suivant le modèle graphique décrit en 2.3.2.1.

Dans l'éditeur de décors, la couche de gestion de la visualisation passive est créée avec une surface virtuelle qui calcule son image en logiciel. Le réaffichage de la couche de visualisation est fait par l'envoi de l'image calculée dans une zone de mémoire accessible par le serveur X. Notons que cette couche est portable car la traduction d'une image en mémoire — composée d'un tableau de **RenderPixel** — en une image composable par InterViews est encapsulée par un objet **FastRaster**.

L'implantation du modèle graphique en logiciel est intrinsèquement compliquée, du même ordre que l'implantation des primitives graphiques dans le serveur X ; cette complexité est complètement invisible du point de vue de l'architecture multicouche. Toutes les autres couches continuent de fonctionner, ainsi que les outils. Nous avons cependant modifié l'algorithme de réaffichage des couches pour tenir compte du fait que la couche de visualisation gère un cache, ce qui rend son réaffichage très peu cher.

2.7 Synthèse

Dans cet exemple, nous avons montré comment notre architecture permet de construire un éditeur graphique sophistiqué en spécialisant des objets de base. Contrairement au premier exemple, nous n'avons pas décrit le processus de création de l'éditeur mais son implantation complète et une de ses dérivations.

L'architecture multicouche utilisée dans cet éditeur rend sa construction systématique. Les fonctions qui gèrent les actions élémentaires de la manipulation directe sont très simples. Malgré cela, l'éditeur est complet et peut évoluer dans plusieurs dimensions sans nécessiter de modification architecturales majeures. Nous avons décrit l'évolution du modèle graphique de la couche de visualisation passive ; nous aurions pu décrire de manière similaire plusieurs autres types d'évolutions :

- reconnaissance de geste,
- manipulation du point de vue,

Catégorie	Classes
Couches	4
Objets graphiques	10
Outils	11
Commandes	9
Widgets	15
Autres	30
Total	79

TAB. 2.3 - *Nombre de classes par catégorie utilisé par l'éditeur d'animation.*

– ajout de contraintes lexicales,
– prise en compte de dispositifs supplémentaires.

Le système TicTacToon contient cinq modules qui utilisent des variantes de l'architecture de l'éditeur d'animation, ce qui démontre que l'architecture multi-couches permet une bonne modularité dans un éditeur, mais aussi une bonne réutilisation entre plusieurs éditeurs :

– l'éditeur d'animation,

– l'éditeur de décors,

– l'éditeur de retouche pixellaire et d'élimination de la couleur du fond (*chroma keying* avec plan alpha),

– l'éditeur de coloriage des personnages,

– l'éditeur de mise en place des scènes et du contrôle du tournage (*layout*).

La table 2.3 contient des éléments chiffrés permettant d'évaluer la complexité de l'éditeur.

Chapitre 3

Conclusion du chapitre

En conclusion à ce chapitre, nous voudrions aborder deux problèmes importants qui subsistent et auxquels notre architecture ne donne pas de solution ; puis nous réutilisons les critères d'analyse donnés en première partie, § 4 pour analyser les bénéfices de l'architecture multicouche visibles sur les exemples.

3.1 Problèmes

Nous avons éludé deux aspects importants de la construction d'éditeurs : la gestion de la mémoire et les sauvegardes et chargements des multiples formats de fichiers. Ces deux problèmes représentent une part importante de la conception et de la réalisation d'un éditeur et doivent être résolus de façon *ad-hoc*.

Gestion de la mémoire Les éditeurs graphiques complexes partagent toujours des structures de données complexes. Lorsque ces structures sont partagées, la mémoire qu'elles utilisent ne peut être libérée que lorsque plus personne ne les référence. Traditionnellement, les langages de programmation de prototypage gèrent la mémoire automatiquement avec des techniques de ramasse-miette. Exception faite des architectures spécialisées, ces techniques obligent l'application à s'interrompre pour nettoyer la mémoire. Ces interruptions pouvant survenir à un moment inopportun, les éditeurs professionnels sont généralement implantés dans des langages qui obligent à gérer la mémoire de façon explicite, comme C ou C++.

En C++, la stratégie la plus utilisée est le comptage de référence : chaque objet susceptible d'être partagé maintient un compte de ses références. Chaque structure qui garde un référent vers cet objet incrémente le compte et lorsque la structure va arrêter de référencer l'objet, elle décrémente son compte de référence. Lorsque le nombre de référence est nul, l'objet n'est plus référencé par personne et peut être libéré. Cette stratégie souffre de deux défauts majeurs : les objets non déréféren-

cés et les structures de données cycliques ne sont jamais libérés. Les premiers sont créés lorsqu'une fonction retourne un objet et que cet objet n'est pas utilisé. Les seconds sont créés lorsqu'un objet A référence un objet B qui, à son tour, référence A directement ou indirectement. Dans ce cas, le compte des références de A est toujours au moins égal à 1 et l'objet n'est jamais libéré.

Le premier problème est principalement lié à la sémantique du langage C++ qui autorise l'utilisation d'une fonction comme procédure. Le second est intrinsèque aux domaines complexes : les structures de données sont cycliques. Il suffit d'avoir besoin d'une liste doublement chaînée pour avoir un cycle.

Nous n'avons pas de solution générale à ce problème et pensons qu'il est responsable d'une partie importante de la complexité des éditeurs, et plus généralement des applications complexes.

Gestion du chargement et de la sauvegarde des objets Tous les éditeurs ont besoin de charger et de sauvegarder les objets qu'ils manipulent à partir de fichiers. L'implantation de ces fonctions représente généralement une part importante des éditeurs, les formats de fichiers graphiques répandus étant très nombreux. Comme pour le point précédent, nous n'avons pas de solution à ce problème mais constatons qu'il représente une part souvent importante et toujours fastidieuse de la réalisation des éditeurs.

3.2 Bénéfices de l'architecture

Pour terminer cette partie, nous reprenons les critères d'évaluation définis en première partie, § 4 pour les outils de construction d'interface en appuyant nos commentaires sur les deux exemples.

Méthode de construction : dans le premier exemple, nous avons mis en œuvre la méthode décrite en deuxième partie, § 3 pour réaliser un éditeur de graphe par étapes successives, testables et validables. Ces caractéristiques ne se retrouvent dans aucune autre architecture.

Modèle du graphique : dans le second exemple, nous avons montré deux modes de spécialisation du modèle graphique : par traduction et par implantation logicielle.

Gestion de dispositifs : la couche de gestion de la trace utilise un dispositif non standard : une tablette graphique avec un stylet renvoyant des informations de pression. Ce dispositif est traité de façon uniforme avec les autres dispositifs.

Extensibilité des applications : Nous avons vu dans le premier exemple une extension du retour d'information pendant la manipulation directe. Dans le second exemple, nous avons décrit l'éditeur d'animation qui utilise un modèle graphique relâché et l'éditeur de décors qui implante le modèle graphique complet. Nous avons aussi, durant le développement de TicTacToon, défini de nouveaux types de **GraphicGlyph**s sans que cela n'entraîne de modifications importantes à la majorité des éditeurs et de même, pour les attributs graphiques.

Modularité de l'architecture : les deux exemples ont montré des contextes de réutilisation partielle de couches et d'outils.

Compacité du code source : le pseudo-code décrivant les outils pages 188 et 190 donne une idée fidèle de la taille du code source nécessaire à implanter les outils. La couche de gestion de la trace, qui est la plus compliquée, utilise 300 lignes de C++.

Temps d'apprentissage : nous n'avons pas réellement évalué le temps d'apprentissage de notre architecture. Cependant, une dizaine de personnes ont travaillé sur des modules de TicTacToon, dont cinq ont développé des éditeurs à partir de notre architecture. L'éditeur nécessitant l'implantation du modèle graphique en logiciel a été conçu et développé avec mon concours direct, les autres modules ont été conçus et réalisés pratiquement sans intervention de ma part.

Temps de construction : nous ne disposons pas non plus de données précises sur ce point. Ayant mis au point le modèle pendant la conception des modules de TicTacToon, nous ne pouvons pas isoler le temps de conception de l'architecture et des éditeurs. Il a fallu une semaine pour concevoir et réaliser l'éditeur de graphe, dont la moitié pour adapter la bibliothèque de placement de graphe à InterViews, et une autre semaine à le spécialiser pour visualiser les dépendances entre pages d'hypertextes.

Conclusion

Nous avons décrit dans cette thèse une architecture basée sur la collaboration de trois types d'objets : la pile, les couches et les outils, et permettant de réaliser des éditeurs graphiques à l'intérieur d'un système de fenêtrage sans sacrifier ni la richesse graphique, ni les performances, ni la variété des dispositifs d'interaction. Cette architecture se place dans la lignée des outils de construction d'interface comme MacApp, Garnet et Unidraw dont elle affine les mécanismes de gestion des objets graphiques et de l'interaction.

Au lieu de limiter le modèle graphique à celui du système de fenêtrage, notre architecture permet d'utiliser plusieurs modèles graphiques, chaque couche pouvant utiliser le modèle le mieux adapté aux données dont elle assure la visualisation. Pour la manipulation directe, Garnet et Unidraw utilisent des objets de type *interacteur* qui implantent de façon monolithique un automate et déroutent la boucle standard de gestion des événements. Dans notre architecture, la manipulation directe est gérée par plusieurs outils qui peuvent être modifiés isolément et qui utilisent la boucle standard de gestion des événements. En outre, un mode d'interaction se décrit en terme d'outils et de couches à l'aide d'une notation dérivée de UAN qui est lisible.

Notre architecture a été principalement utilisée et validée dans des éditeurs d'objets graphiques 2D, qu'ils soient pixellaires ou vectoriels. Nous avons pu constater leur robustesse, leur extensibilité et leur modularité lors de leur utilisation en tant que produits commercialisés. En revanche, nous n'avons testé que des prototypes d'éditeur de texte et 3D. Pour vraiment valider l'architecture, la seule méthode serait de réaliser des éditeurs conséquents pour ces domaines.

Parmi les extensions possibles de notre architecture, nous pensons que des outils de constructions d'interface peuvent faciliter l'utilisation de l'architecture multicouche. Les schémas UAN en particulier, peuvent certainement servir de base à un formalisme opérationnel de description de l'interaction. Nous n'avons que peu exploré cette voie mais déjà plusieurs questions se posent : UAN est-il assez puissant pour décrire la plupart des manipulations interactives ? si non, faut-il l'étendre ou changer de formalisme ? Quel est le meilleur moyen d'exprimer les actions ? Un langage spécialisé de haut niveau ou le langage dans lequel le reste de l'application graphique est écrit ? Toutes ces questions nécessitent encore des investigations.

Une autre voie dans laquelle nous souhaiterions nous engager est l'utilisation de notre architecture comme base d'un système de *Widget*s. C'est une perspective que nous n'avons pas abordé auparavant mais que nous pouvons justifier de trois manières : expérimentalement, architecturalement et en nous plaçant dans le modèle de l'Arche (décrit en première partie, § 5.2).

FIG. 3.1 - *Le Widget Panner, qui remplace les barres de défilement horizontale et verticale.*

Expérimentalement, nous avons eu plusieurs fois besoin d'étendre des *Widget*s afin de changer leur apparence ou leur comportement.

> Un exemple intéressant vient du *Panner*, qui est un objet rectangulaire destiné à remplacer les barres de défilement horizontales et verticales d'une fenêtre (voir figure 3.1). Le rectangle extérieur représente la taille totale de la surface virtuelle sur laquelle s'affichent les données éditées. Le rectangle intérieur représente la taille du point de vue et sa position dans la surface virtuelle.
>
> Deux fois, nous avons voulu modifier ce *Widget* et avons dû le refaire. La première fois, nous voulions afficher dans le fond la silhouette du graphique affiché sur la surface virtuelle. La seconde fois, nous voulions pouvoir tourner le point de vue pour améliorer le confort de dessin à main levée, comme nous le décrivons en troisième partie, § 2.2.
>
> S'il avait été construit sur notre architecture, ce *Widget* aurait pu être modifié très simplement dans les deux cas au lieu d'être intégralement refait.

Architecturalement, il est contestable que les *Widget*s disponibles dans les boîtes à outils se comportent comme des boîtes noires. Les boîtes à outils implantent leurs *Widget*s à partir des bibliothèques de bas niveau qui sont accessibles au développeur. Cependant, la modification d'un *Widget* existant est, dans la pratique, très difficile, à moins de disposer de son code source, de le copier et de le modifier. Nous pensons que l'architecture multicouche offre une meilleur alternative.

Enfin, lorsqu'on veut utiliser le modèle de l'Arche pour concevoir et réaliser une application graphique interactive, on constate que la modélisation d'une application change lorsqu'un objet qui se trouvait dans la composante de l'interaction passe dans la composante de présentation pour être géré par le contrôleur du dialogue. Le contrôleur du dialogue devient soudain plus complexe car responsable

de tâches de contrôle de bas niveau. [Nigay94] subdivise le contrôleur du dialogue en agents PAC pour maîtriser cette complexité, qui subsiste malgré tout dans l'implantation.

Nous pensons que notre architecture permettrait de rendre la transition entre un *Widget* et un objet graphique spécialisé plus continue en définissant des mécanismes d'extension aux *Widget*s et en s'appuyant sur notre méthode pour la mise au point. Cette démarche nous paraît plus raisonnable que la réécriture exhaustive de *Widget*s pour tout besoin spécifique, qui est la méthode employée couramment aujourd'hui.

La gestion des couches à l'intérieur du système de fenêtrage améliorerait, à terme, l'architecture multicouche en autorisant l'interaction à l'extérieur des fenêtres. Cette amélioration autoriserait par exemple l'implantation des *See Through Tools*, et en définitive, permettrait d'enrichir les modes d'interaction graphique.

Annexes

Annexe A

Notation

Pour décrire les classes, nous utilisons la même notation que celle du livre *Design Patterns* [Gamma et al.94]. Cette notation est dérivée d'OMT [Rumbaugh et al.91] et sert à décrire graphiquement l'interface des classes et les relations entre les classes. La figure A.1 décrit le rôle des symboles dans la notation.

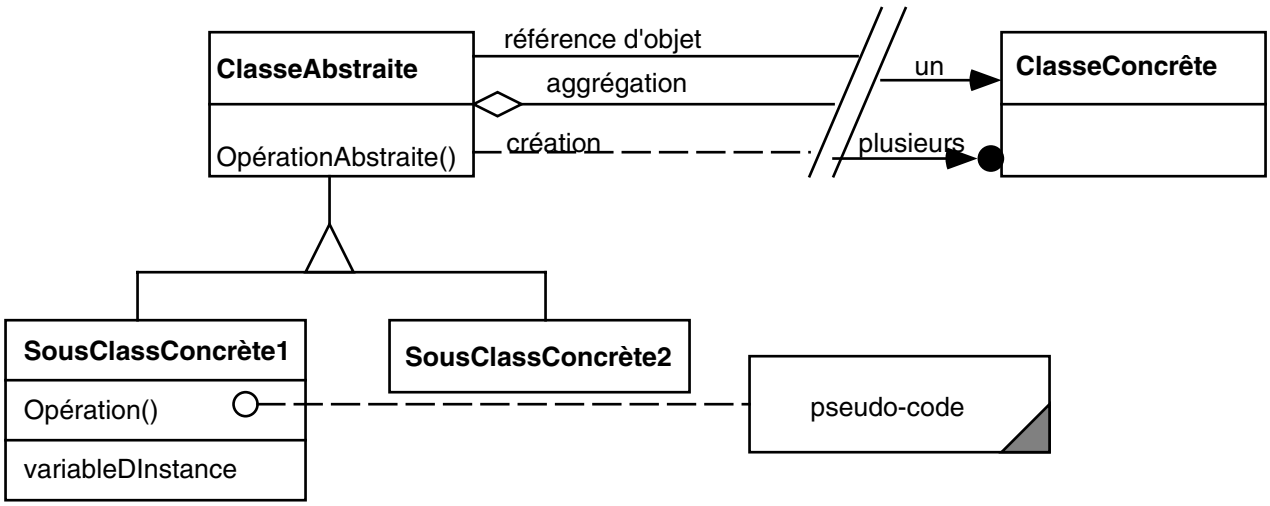
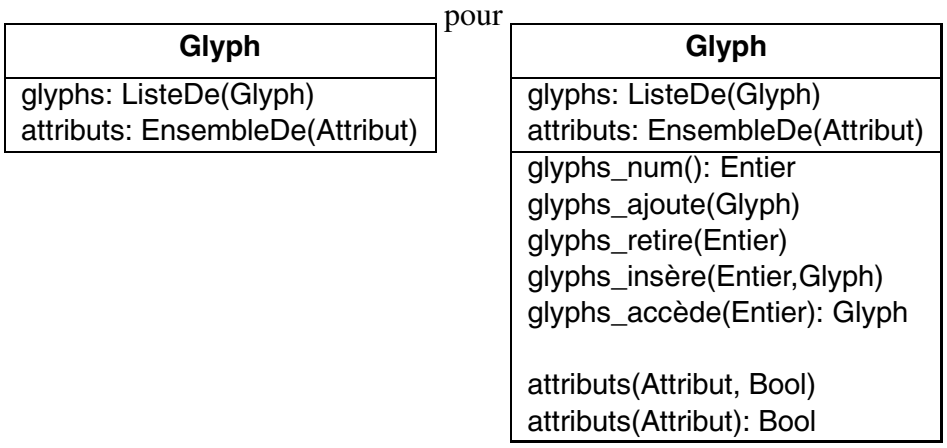


FIG. A.1 - *Notation de diagramme de classe*

De plus, nous utilisons les abréviations suivantes :



Annexe B

UAN

Pour décrire la gestion des événements dans les couches, nous utilisons une notation proche de UAN [Hartson et al.90]. UAN décrit un mode d'interaction (appelé tâche dans UAN) à l'aide d'une table dont chaque ligne indique les traitements à effectuer lorsqu' un événement particulier est produit par un dispositif d'entrée.

Nous utilisons une table à cinq colonnes pour décrire les actions associées aux événements :

Couche : la colonne de gauche indique le nom de la couche.

Contexte : la colonne suivante indique le contexte dans lequel la couche interceptera l'événement. Ce contexte est décrit pas une expression booléenne qui, lorsqu'elle est vraie, autorise la couche à intercepter l'événement. Nous laissons la colonne vide lorsque l'interception de l'événement n'est pas contextuelle.

Événement : la colonne suivante décrit l'événement en utilisant la syntaxe de UAN résumée dans la table B.1. Nous utilisons principalement la souris (notée M comme *Mouse*) qui désigne en réalité le pointeur actif, c'est-à-dire une souris ou une tablette.

Action locale : lorsque l'événement est traité par la couche, il peut agir directement sur la couche. Nous décrivons les actions sur la couche dans cette colonne.

Action transmise : lorsque l'événement est traité par la couche, il peut faire appel à des fonctions d'autres couches qui sont représentées dans la table UAN, ou même faire appel à des fonctions globales. Cette colonne décrit ces actions.

L'ordre des actions locales et transmises n'a généralement pas d'importance et n'est pas indiqué. En revanche, les couches sont décrites du fond (à la première ligne) au premier plan (à la dernière ligne). Ainsi, le fait qu'un événement soit traité

par une ligne signifie implicitement qu'il n'a pas été intercepté par les lignes suivantes. Par exemple, le mode d'interaction décrit en III.7 doit être lue comme suit :

Lorsque l'événement *événement1* arrive sur la couche **Couche1** — c'est-à-dire n'a pas été intercepté par toutes les couches **Couche2**, **Couche3**, etc. —, si la précondition *précondition1* est vraie, alors l'action *action1* est éxécutée dans le contexte de la couche **Couche1** et l'action *action2* est exécutée dans le contexte de la couche **Couche3**.

Lorsque l'événement *événement2* arrive sur la couche **Couche1** et que la précondition *précondition2* est vraie, l'action locale *action2* est exécutée et l'action *action3* est exécutée sur la couche **Couche4**. Les événements *événement1* et *événement2* peuvent être identiques si les préconditions ne le sont pas. En cas d'ambiguïté, c'est la première ligne qui est utilisée.

Tâche III.7 Exemple UAN

Couche	Contexte	Événement	Action locale	Action transmise
Couche1	précondition1	événement1	action1	**Couche3** action2
	précondition2	événement2	action2	**Couche4** action3
Couche2	précondition3	événement3	action3	**Couche4** action4
...	…	…	…	**Couche**n …

Action	Signification
˜	déplace le curseur
[X]	le contexte de l'objet graphique X
˜[X]	déplace le curseur dans le contexte de l'objet X
˜[x , y]	déplace le curseur au point x , y en dehors de tout contexte d'objet
˜[x , y in A]	déplace le curseur au point x , y dans le contexte de l'objet A
˜[X in Y]	déplace le curseur sur le contexte de l'objet X dans le contexte de l'objet Y
[X]˜	sort le curseur du contexte de l'objet X
$\downarrow$	appuyer
$\uparrow$	relâcher
X $\downarrow$	appuyer sur le bouton nommé X
X $\uparrow$	relâcher le bouton nommé X
X $\downarrow\uparrow$	cliquer sur le bouton nommé X
X"*abc*"	saisir les trois lettres « abc » à partir du dispositif X
X(xyz)	saisir la valeur de la variable xyz à partir du dispositif X
()	groupement
	répétition d'une action 0 fois ou plus
+	répétition d'une action 1 fois ou plus
{ }	l'action à l'intérieur des accolades est optionnelle
A B	séquence ; l'action A suivie de l'action B (A et B peuvent apparaître sur deux lignes)
OR	disjonction, choix d'une action
&	indépendance de l'ordre des actions
$\leftrightarrow$	entrelacement ; les actions peuvent être entrelacées dans le temps
$\|\|$	concurrence ; les actions peuvent être faites en même temps
;	interruption de séquence d'actions
$\forall$	pour tout

TAB. B.1 - *Résumé de la syntaxe des actions UAN et de leur signification*

Bibliographie

[Adobe Systems Incorporated90] Adobe Systems Incorporated. – *PostScript Language Reference Manual*. – Reading, MA, USA, Addison-Wesley, 1990, second édition, viii + 764p.

[Adobe Systems Incorporated91] Adobe Systems Incorporated, 1585 Charleston Road, P. O. Box 7900, Mountain View, CA 94039-7900, USA, Tel: (415) 961-4400. – *Adobe Illustrator 3.0 User Guide*, 1991.

[Adobe Systems Incorporated93a] Adobe Systems Incorporated, 1585 Charleston Road, P. O. Box 7900, Mountain View, CA 94039-7900, USA, Tel: (415) 961-4400. – *Adobe PhotoShop 2.5 User Guide*, 1993.

[Adobe Systems Incorporated93b] Adobe Systems Incorporated. – *Programming the Display PostScript System with NeXTstep*. – Reading, MA, USA, Addison-Wesley, 1993.

[Alias Research Inc.a] Alias Research Inc., Toronto, Canada. – *Alias StudioPaint 3D*.

[Alias Research Inc.b] Alias Research Inc., Toronto, Canada. – *Alias Upfront*.

[American national standards institute85] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. – *Information Systems—Computer Graphics—Graphical Kernel System (GKS). ANSI X3.124-1985*, 1985. Includes Fortran bindings to GKS.

[American national standards institute88] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. – *Information Systems—Computer Graphics—Programmer's Hierarchical Interactive Graphical System. Draft proposal X3.144.1988*, 1988.

[Apple Computer Incorporated93] Apple Computer Incorporated. – *Newton User's Manual*, 1993.

[Autodesk92] Autodesk. – *3D Studio Reference Manual Version 3*, 1992.

[Beaudouin lafon et al.90] Michel Beaudouin-Lafon, Yves Berteaud et Stéphane Chatty. – Creating direct manipulation applications with Xtv. *Proceedings of the European X Conference (EX'90)*. – novembre 1990.

[Beaudouin lafon et al.91] Michel Beaudouin-Lafon, Yves Berteaud, Stéphane Chatty, Jean-Daniel Fekete et Thomas Baudel. – *The X Television – C++ Library for Direct Manipulation Interfaces*. – Technical report, LRI, Université de Paris-Sud, France, février 1991. version 2.0.

[Beaudouin-Lafon91] Michel Beaudouin-Lafon. – *The Graph Widget – Implementation Manual*. – Technical report, LRI, Université de Paris-Sud, France, juin 1991. version 1.5.

[Bézier70] Pierre Bézier. – *Emploi des machines à commande numérique*. – Masson, 1970.

[Bier et al.94] Eric A. Bier, Maureen C. Stone, Ken Fishkin, William Buxton et Thomas Baudel. – A taxonomy of see-through tools. *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, pp. 358–364. – 1994.

[Borning79] Alan Borning. – *ThingLab — A Constraint-Oriented Simulation Laboratory.* – Rapport technique nSSL-79-3, Xerox Palo Alto Research Center, juillet 1979.

[Calder et al.90] Paul R. Calder et Mark A. Linton. – Glyphs: Flyweight Objects for User Interfaces. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 92–101. – 1990.

[Cardelli et al.85] Luca Cardelli et Peter Wegner. – On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, vol. 17, n4, décembre 1985, pp. 471–522.

[Chatty94] Stéphane Chatty. – Extending graphical toolkit for two-handed interaction. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '94)*. ACM, pp. 195–204. – New York, NY 10036, USA, novembre 1994.

[Chin95] Norman Chin. – *Graphic Gems V*, chap. A Walk through BSP Trees, pp. 121–138. – Reading, MA, USA, Addison-Wesley, 1995, *Graphic Gems*.

[Coutaz87] Joöelle Coutaz. – PAC, an implementation model for dialog design. *Proceedings of INTERACT'87*, pp. 431–436. – septembre 1987.

[Cox et al.91] Brad J. Cox et Andrew J. Novobilski. – *Object Oriented Programming: An Evolutionary Approach*. – Reading, MA, USA, Addison-Wesley, 1991.

[Custer93] Helen Custer. – *Inside Windows NT*. – One Microsoft Way, Redmond, WA 98052-6399, USA, Microsoft Corporation, 1993, xxiv + 385p.

[Deneba software91] Deneba Software, Miami, Florida 33122. – *Canvas 3.0*, juin 1991.

[Fekete et al.95] Jean Daniel Fekete, Erick Bizouarn, Eric Cournarie, Thierry Gallas et Frédéric Taillefer. – TicTacToon: A paperless system for professional 2d animation. *Proceedings of SIGGRAPH '95 (Los-Angeles, California, August 6–11, 1995)*, éd. par Stephen N. Spencer. ACM SIGGRAPH. – ACM Press, 1995.

[Fekete92] Jean Daniel Fekete. – A multi-layer graphic model for building interactive graphical applications. *Proceedings of Graphics Interface '92*, pp. 294–300. – mai 1992.

[Ferguson et al.93] Paula Ferguson et David Brennan. – *Motif Reference Manual*. – 981 Chestnut Street, Newton, MA 02164, USA, O'Reilly & Associates, Inc., juin 1993, volume 6B, 920p.

[Ferguson92] Paula Ferguson. – The X11 input extension: A tutorial. *The X Resource*, vol. 4, n1, décembre 1992, pp. 171–194.

[Flanagan92] David Flanagan. – *X Toolkit Intrinsics Reference Manual*. – 981 Chestnut Street, Newton, MA 02164, USA, O'Reilly & Associates, Inc., 1992, third édition, volume 5, xiii + 899p.

[Foley et al.90] James D. Foley, Andries van Dam, Steven K. Feiner et John F. Hughes. – *Fundamentals of Interactive Computer Graphics*. – Reading, MA, USA, Addison-Wesley, 1990, second édition, *The Systems Programming Series*, xxiii + 1174p.

[Gamma et al.94] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. – *Design Patterns*. – Reading, MA, USA, Addison-Wesley, 1994.

[Gangnet et al.89] Michel Gangnet, Jean-Claude Hervé, Thierry Pudet et Jean-Manuel Van Thong. – Incremental computation of planar maps. *Computer Graphics (SIGGRAPH '89 Proceedings)*, éd. par Jeffrey Lane, pp. 345–354. – juillet 1989.

[Gaskins93] Tom Gaskins. – The multi-buffering extension—smooth animation effortlessly. *The X Resource*, vol. 8, n1, octobre 1993, pp. 121–159.

[Goldberg et al.83] A. Goldberg et D. Robson. – *Smalltalk 80: The Language and its Implementation*. – Addison-Wesley, 1983, 736p.

[Goodman87] Danny Goodman. – *The Complete HyperCard Handbook*. – Bantam, 1987.

[Gosling et al.89] James Gosling, David S. H. Rosenthal et Michelle Arden. – *The NeWS Book*. – Berlin, Germany / Heidelberg, Germany / London, UK / etc., Springer-Verlag, 1989, vi + 235p.

[Gosling81] James Gosling. – *A redisplay algorithm*, pp. 123–129. – New York, NY 10036, USA, ACM Press, 1981. Published as ACM SIGPLAN Notices, v. 16, no. 6, and ACM SIGOA newsletter, vol. 2, nas 1/2, spring/summer 1981.

[Gosling86] James Gosling. – *Methodology of Window Management*, chap. SunDew - A Distributed and Extensible Window System. – Berlin, Springer Verlag, 1986, *Eurographics Seminars, Tutorials and Perspectives in Computer Graphics*.

[Hartson et al.90] H. Rex Hartson, Antonio C. Siochi et Deborah Hix. – The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, vol. 8, n3, 1990, pp. 181–203.

[Ilog94] ILOG, 2, Av. Galliéni, 94253 GENTILY Cedex, France. – *ILOG Views Reference Manual. Version 2.1*, 1994.

[Ingalls78] Dan H. Ingalls. – The smalltalk-76 programming system: Design and implementation. *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*. – 1978.

[Ingalls83] Dan H. Ingalls. – The evolution of the smalltalk virtual machine. *Smalltalk-80: Bits of History, Words of Advice*, éd. par G. Krasner. – Reading, MA, USA, Addison-Wesley, 1983.

[Karsenty94] Alain Karsenty. – *GroupDesign : un collecticiel synchrone pour l'edition partagee de documents*. – Thèse, LRI, Université Paris-Sud, Orsay, 1994.

[Knuth79] Donald E. Knuth. – *TEX and METAFONT—New Directions in Typesetting*. – 12 Crosby Drive, Bedford, MA 01730, USA, Digital Press, 1979, xi + 201 + 105p.

[Kopp et al.94] Manfred Kopp et Michael Gervautz. – *Graphic Gems IV*, chap. XOR-Drawing with Guaranteed Contrast, pp. 413–414. – Reading, MA, USA, Addison-Wesley, 1994, *Graphic Gems*.

[Krasner et al.88] Glenn E. Krasner et Stephen T. Pope. – A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, vol. 1, n3, août 1988, pp. 26–49.

[Laffra91] Chris Laffra. – Object oriented methods for graphics. *SIGGRAPH '91 course notes on Object and Constraint Paradigms for Graphics*. – New York, NY 10036, USA, ACM Press, juillet 1991.

[Leler88] Wm Leler. – *Constraint Programming Languages*. – Reading, MA, USA, Addison-Wesley, 1988.

[Letraset92] Letraset. – *FontStudio 2.1 User's Manual*, 1992.

[Lieberman85] Henry Lieberman. – There's more to menu systems than meets the screen. *Computer Graphics (SIGGRAPH '85 Proceedings)*, éd. par B. A. Barsky, pp. 181–189. – juillet 1985.

[Linton et al.89] Mark A. Linton, John M. Vissides et Paul R. Calder. – Composing user interfaces with interviews. *IEEE Computer*, vol. 22, n2, février 1989, pp. 8–22.

[Linton et al.93] Mark Linton et Chuck Price. – Building distributed user interfaces with Fresco. *The X Resource*, vol. 5, n1, janvier 1993, pp. 77–87.

[Linton et al.94] Mark Linton, Steven Tang et Steven Churchill. – Redisplay in Fresco. *The X Resource*, vol. 9, n1, janvier 1994, pp. 63–69.

[Liskov et al.77] B. Liskov, A. Snyder, R. Atkinson et C. Schaffert. – Abstraction mechanisms in CLU. *Communications of the ACM*, vol. 20, n8, août 1977. – Also published in/as: In "Readings in Object-Oriented Database Systems" edited by S.Zdonik and D.Maier, Morgan Kaufman, 1990.

[McCormack88] Joel McCormack. – An overview of the X toolkit. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, 1988, pp. 46–55.

[Microsoft Corporation91] Microsoft Corporation. – *Microsoft Windows 3.1 Guide to Programming*. – One Microsoft Way, Redmond, WA 98052-6399, USA, Microsoft Corporation, 1991.

[Microsoft Corporation92] Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA. – *Windows 3.1 Manual*, 1992.

[Moloney et al.89] J. Moloney, A. Borning et B. Freeman-Benson. – Constraint technology for user-interface construction in ThingLab II. *ACM SIGPLAN Notices*, vol. 24, n10, octobre 1989, pp. 381–388.

[Myers89] Brad A. Myers. – Encapsulating interactive behaviors. *Proceedings of ACM CHI'89 Conference on Human Factors in Computing*, pp. 319–324. – 1989.

[Myers90] Brad A. Myers. – A New Model for Handling Input. *ACM Transactions on Information Systems*, vol. 8, n3, 1990, pp. 289–320.

[Myers91a] Brad A. Myers. – The garnet user interface development environment. *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, p. 486. – 1991.

[Myers91b] Brad A. Myers. – Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 211–220. – 1991.

[Neider et al.93] Jackie Neider, Tom Davis et Mason Woo. – *OpenGL Programming Guide—The Official Guide to Learning OpenGL, Release 1*. – Reading, MA, USA, Addison-Wesley, 1993, xiii + 516p.

[Next Computer Inc.92] Next Computer Inc. – *NeXTSTEP Development Tools and Techniques*. – Redwood City, California, Addison-Wesley, 1992.

[Nigay94] Laurence Nigay. – *Conception et modelisation logicielles des systemes interactifs : application aux interfaces multimodales*. – Thèse, Universite Joseph Fourier, Grenoble, janvier 1994.

[Nye88] Adrian Nye. – *Xlib Programming Manual for Version 11*. – 981 Chestnut Street, Newton, MA 02164, USA, O'Reilly & Associates, Inc., 1988, volume 1, xxxiii + 615p.

[Open Inventor Architecture Group94] Open Inventor Architecture Group. – *Open Inventor C++ Reference Manual: The Official Reference Document for Open Systems*. – Reading, MA, USA, Addison-Wesley, 1994, vi + 767p.

[Ousterhout94] John K. Ousterhout. – *Tcl and the Tk Toolkit*. – Reading, MA, USA, Addison-Wesley, 1994, xx + 458p.

[Pfaff85] Günter E. Pfaff (édité par). – *User Interface Management Systems*. – Berlin, Springer Verlag, 1985, *Eurographics Seminars*.

[Pier et al.88] Ken Pier, Eric Bier et Maureen Stone. – An introduction to gargoyle: An interactive illustration tool. *Document Manipulation and Typography*. pp. 223–238. – Cambridge University Press, avril 1988.

[Porter et al.84] Thomas Porter et Tom Duff. – Compositing digital images. *Computer Graphics (SIGGRAPH '84 Proceedings)*, éd. par Hank Christiansen, pp. 253–259. – juillet 1984.

[Pudet94] Thierry Pudet. – Real Time Fitting of Hand-Sketched Pressure Brushstrokes. *Eurographics'94. Proceedings of the European Computer Graphics Conference and Exhibition*. – Amsterdam, Netherlands, 1994.

[Quantel inc.] Quantel Inc., England, somewhere. – *PaintBox User's Manual*.

[Rogers94] Gary Rogers. – *The X Image Extension*. – X Consortium, 1994.

[Rumbaugh et al.91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy et Wiliiam Lorenson. – *Object-Oriented Modeling and Design*. – Englewood Cliffs, NJ 07632, USA, Prentice-Hall, 1991.

[Saito et al.90] Takafumi Saito et Tokiichiro Takahashi. – Comprehensible rendering of 3-D shapes. *Computer Graphics (SIGGRAPH '90 Proceedings)*, éd. par Forest Baskett, pp. 197–206. – août 1990.

[Scheifler et al.86] Robert W. Scheifler et Jim Gettys. – The X window system. *ACM Transactions on Graphics*, vol. 5, n2, 1986, pp. 79–109.

[Schmucker87] Kurt J. Schmucker. – *Readings in Human-Computer Interaction: A Multidisciplinary Approach*, chap. MacApp: An Application Framework, pp. 591–594. – Los Altos, CA 94022, Morgan-Kaufmann Publishers Inc., 1987.

[Segal et al.93] Mark Segal et Kurt Akeley. – *The OpenGL Graphics Interface*. – Rapport technique, Mountain View, CA,USA, Silicon Graphics Computer Systems, 1993.

[Shantsis94] Michael A. Shantsis. – A model for efficient and flexible image computing. *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, éd. par Andrew Glassner. ACM SIGGRAPH, pp. 147–154. – ACM Press, juillet 1994. ISBN 0-89791-667-0.

[Shneiderman83] Ben Shneiderman. – Direct manipulation: a step beyond programming languages. *IEEE Computer*, août 1983, pp. 57–69.

[Shneiderman92] Ben Shneiderman. – *Designing The User Interface*. – Reading, MA, USA, Addison-Wesley, 1992, second édition.

[SK94] László Szirmay-Kalos. – Dynamic layout algorithm to display general graphs. *Graphics Gems IV*, éd. par Paul Heckbert, pp. 505–517. – Boston, Academic Press, 1994.

[Smith et al.82] D. Smith, R. Kimball, B. Verblank et E. Harslem. – Designing the star user interface. *Byte*, vol. 7, n4, avril 1982, pp. 242–282.

[Stallman87] Richard M. Stallman. – Emacs: The extensible, customizable, self-documenting display editor. *Interractive Programming Environments*, éd. par David R. Barstow, Howard E. Shrobe et Erik Sandewall, pp. 300–325. – Mc Graw-HILL, 1987.

[Steele Jr.84] Guy L. Steele Jr. – *Common Lisp—The Language*. – 12 Crosby Drive, Bedford, MA 01730, USA, Digital Press, 1984, xii + 465p. See also [Tatar87].

[Stroustrup91] Bjarne Stroustrup. – *The C++ Programming Language*. – Reading, MA, USA, Addison-Wesley, 1991, second édition, xi + 669p.

[Sun microsystems] Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA. – *Programmer's Reference Manual of the SUN Window System*.

[Tang et al.93] Steven H. Tang et Mark A. Linton. – Pacers: Time-elastic objects. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*. pp. 35–43. – New York, NY 10036, USA, 1993.

[Tang et al.94] Steven H. Tang et Mark A. Linton. – The effects of blending graphics and layout. *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM, pp. 167–174. – novembre 1994.

[Tatar87] Deborah G. Tatar. – *A Programmer's Guide to Common LISP*. – 12 Crosby Drive, Bedford, MA 01730, USA, Digital Press, 1987, x + 327p. See also [Steele Jr.84].

[Tesler81] Larry Tesler. – The smalltalk environment. *Byte*, vol. 6, n8, août 1981, pp. 90–147.

[User Interface Developer's Workshop91] User Interface Developer's Workshop. – The Arch model: Seeheim revisited. – avril 1991. Presented at ACM SIGCHI.

[Vlissides et al.89] John M. Vlissides et Mark A. Linton. – Unidraw: A Framework for Building Domain-Specific Graphical Editors. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 158–167. – 1989.

[Weinand et al.88] Andre Weinand, Erich Gamma et Rudolf Marty. – ET++—an object oriented application framework in C++. *ACM SIGPLAN Notices*, vol. 23, n11, novembre 1988, pp. 46–57.

[Wisskirchen91] Peter Wisskirchen. – *Object-Oriented Graphics*. – Berlin, Springer Verlag, 1991.

[Womack92] Paula Womack. – PEX protocol specification and encoding, version 5.1P. *The X Resource*, vol. Special issue A, mai 1992, pp. 1–179.

[Zanden et al.91] Brad Vander Zanden, Brad A. Myers, Dario Giuse et Pedro Szekely. – The Importance of Pointer Variables in Constraint Models. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 155–164. – 1991.

Résumé

Dans cette thèse, nous décrivons une architecture logicielle destinée à faciliter la construction d'applications graphiques interactives à manipulation directe que nous appelons *éditeurs*. Nous proposons d'utiliser plusieurs couches graphiques superposées, chaque couche gérant des objets graphiques de même nature. En utilisant cette architecture, nous pouvons décrire explicitement le comportement de tous les objets graphiques apparaissant dans les interfaces, en particulier les objets fugaces qui étaient jusqu'à présent mal définis par les modèles architecturaux et par les outils logiciels. Nous pouvons aussi mettre en œuvre des optimisations graphiques très poussées qui étaient jusqu'ici très difficile à concilier avec une architecture de haut niveau. Enfin, nous déduisons de notre architecture une méthode pour le développement incrémental des éditeurs.

En exemple, nous décrivons deux applications graphiques qui ont été bâties à l'aide de notre architecture. La première permet de comprendre notre méthode et la seconde est issue d'un éditeur faisant partie d'un produit commercialisé pour la fabrication de dessins animés assistée par ordinateur.

Abstract

In this thesis, we describe an architecture for building interactive graphical applications, called *editors*, based on direct manipulation. The architecture is based on superimposed graphical layers where each layer manages graphical objects of a similar nature. With this architecture, we can explicitely describe the behavior of all the graphical objects apprearing in editors, including transient objects that were incompletely supported by previous architectures and toolkits. We can also support optimisations that are usually considered incompatible with a high level architecture. Finally we describe a method for building editors incrementally.

Two examples demonstrate our architecture. The first example demonstrates the method whereas the second example demonstrates how the architecture was used to build an editor that is a part of a commercial system for computer aided animation.