FENIX : un système multifenêtres intégré à UNIX

Ivan Boule

THESE

présentée à

L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

pour obtenir le titre de
DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE
DE GRENOBLE
(arrêté ministériel du 5 Juillet 1984)
Spécialité : INFORMATIQUE

par

Ivan Boule

FENIX : un système multifenêtres intégré à Unix

Thèse soutenue le 13 avril 1987 devant la Commission d'Examen composée de :

MM.	Sacha	Krakowiak	Président
	Jean-Pierre	Banâtre	Rapporteurs
	Jean-François	Abramatic	
	Jacques	Mossière	
	Vania	Joloboff	

Thèse préparée au sein du Laboratoire de Génie Informatique

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Daniel BLOCH **Année 1987**

Vice - Présidents : René CARRE
Jean-Marie PIERRARD

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BESSON Jean	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
BUYLE-BODIN Maurice	ENSERG
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHERUY Arlette	ENSIEG
CHIAVERINA Jean	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Mar	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Pierre	ENSIEG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSEEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PAUTHENET René	ENSIEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SERMET PIERRE	ENSERG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
PLA Fernand
GHIBAUDO Gérard
HAMAR Sylvaine
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CAILLET Marcel
CARRE René
FRUCHART Robert
JORRAND Philippe
LANDAU Ioan
MARTIN

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BINDER Gilbert
BONNET Roland
BORNARD Guy
CALMET Jacques
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
JOUD Jean-Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MERMET Jean
MUNIER Jacques
SENATEUR Jean-Pierre
SUERY Michel
TEDOSIU
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

BERNARD Claude
CHATILLON Catherine
CHATILLON Christian
COULON Michel
DIARD Jean-Paul
FOSTER Panayotis
HAMMOU Abdelkader
MALMEJAC Yves
MARTIN GARIN Régina
SAINTFORT Paul
SARRAZIN Pierre
SIMON Jean-Paul
TOUZAIN Philippe
URBAIN Georges

E.N.S.E.R.G

BOREL Joseph
CHOVET Alain
DOLMAZON Jean-Marc
HERAULT Jeanny

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

BOIS Daniel
DARVE Félix
MICHEL Jean-Marie
ROWE Alain
VAUCLIN Michel

E.N.S.I.M.A.G

BERT Didier
COURTIN Jacques
COURTOIS Bernard
DELLA DORA Jean
FONLUPT Jean
SIFAKIS Joseph

E.F.P.G

CHARUEL Robert

C.E.N.G

CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T

DEMOULIN Eric
DEVINE
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : Monsieur M.MERMET
Directeur des Etudes et de la formation: Monsieur J. LEVASSEUR
Directeur des recherches : Monsieur J. LEVY
Secrétaire Général : Mademoiselle M. CLERGUE

PROFESSEURS DE 1ère CATEGORIE

COINDE Alexandre	Gestion
GOUX Claude	Métallurgie
LEVY Jacques	Métallurgie
LOWYS Jean-Pierre	Physique
MATHON Albert	Gestion
RIEU Jean	Mécanique-Résistance des matériaux
SOUSTELLE Michel	Chimie
FORMERY Philippe	Mathématiques Appliquées

PROFESSEURS DE 2ème CATEGORIE

HABIB Michel	Informatique
PERRIN Michel	Géologie
VERCHERY Georges	Matériaux
TOUCHARD Bernard	Physique Industrielle

DIRECTEUR DE RECHERCHE

LESBATS Pierre	Métallurgie

MAITRE DE RECHERCHE

BISCONDI Michel	Métallurgie
DAVOINE Philippe	Géologie
FOURDEUX Angeline	Métallurgie
KOBYLANSKI André	Métallurgie
LALAUZE René	Chimie
LANCELOT Francis	Chimie
LE COZE Jean	Métallurgie
THEVENOT François	Chimie
TRAN MINH Canh	Chimie

Personalités habilitées à diriger des travaux de recherche

DRIVER Julian	Métallurgie
GUILHOT Bernard	Chimie
THOMAS Gérard	Chimie

Professeurs à l'UER de Sciences de Saint-Etienne

VERGNAUD Jean-Maurice	Chimie des Matériaux et Chimie Industrielle

Je tiens à remercier

Sacha Krakowiak, Professeur à l'Université de Grenoble, de m'avoir fait l'honneur de présider le jury de cette thèse.

Jacques Mossière, Directeur du Laboratoire de Génie Informatique, qui a dirigé cette thèse. Qu'il me soit permis de lui témoigner ici toute ma reconnaissance pour les précieux conseils qu'il m'a donnés lors de la rédaction de ce document.

Jean-Pierre Banâtre, Professeur à l'Université de Rennes, et Jean-François Abramatic, Directeur du GIPSI-SM90 à l'INRIA, d'avoir bien voulu faire partie de ce jury.

Vania Joloboff, Ingénieur au Centre de Recherches Bull et responsable scientifique de ce projet, de m'avoir accueilli dans son équipe. Je tiens également à le remercier pour l'aide inappréciable qu'il m'a apportée durant l'élaboration de cette thèse.

Pieter van der Linden, Ingénieur au GIPSI-SM90, pour les idées échangées à chacune de nos rencontres.

Daniel Iglésias et son équipe de reprographie pour le soin apporté à la présentation matérielle de ce document.

Ivan Boule

Ce travail a été financé par le Centre de Recherches Bull de Grenoble dans le cadre du projet TIGRE.

TABLE DES MATIERES

TABLE DES MATIERES

Chapitre I

INTRODUCTION

La début des années 1980 représente une étape importante dans l'évolution des systèmes informatiques avec le développement des machines individuelles. Ce phénomène s'est traduit par une profonde mutation du marché de l'ordinateur qui, de machine [presque] exclusivement orientée vers le développement et l'exploitation, est devenu un outil de travail couvrant un nombre croissant d'activités telles que la bureautique, la conception et l'enseignement assistés par ordinateur, l'aide à la décision, etc.

Cette évolution est étroitement liée aux progrès continuels réalisés au niveau de l'intégration des composants (micro-processeurs, mémoires, etc.) et à la diminution simultanée de leur prix. La conjonction de ces deux facteurs permet de réaliser à des coûts de plus en plus faibles des machines individuelles très performantes et dotées pour la plupart d'un terminal graphique de grande définition et d'un mécanisme de désignation indépendant du clavier (souris, tablette graphique, etc.).

D'autre part, le développement des réseaux a permis la conception d'une nouvelle architecture de systèmes informatiques dits répartis qui tendent actuellement à se substituer aux systèmes centralisés en temps partagé [Brown 83] [Satyanarayanan 84]. Dans ce type de systèmes, chaque utilisateur dispose d'un ordinateur individuel suffisamment puissant pour satisfaire localement la majorité de ses besoins, le réseau assurant la communication entre utilisateurs et l'accès à des ressources communes plus importantes (bases de données, imprimantes puissantes, etc.) fournies par des centres serveurs.

De par les tâches qu'elles permettent d'accomplir, les machines individuelles appelées maintenant postes de travail individuels sont en général destinées à une clientèle (secrétaires, architectes, etc.) pour laquelle il importe que le système soit

facile à utiliser. Cette exigence a suscité de nombreux travaux de recherches dans un domaine - l'interface homme-machine - jusqu'alors relativement négligé.

Un premier axe de recherches a conduit au développement de systèmes multi-fenêtres, introduits à l'origine chez Xerox dans les environnements de programmation Smalltalk [Ingalls 78] et Interlisp [Teitelman 81]. L'objectif de tels systèmes est de permettre à l'utilisateur de mener plusieurs tâches en parallèle et de passer de l'une à l'autre rapidement, à l'image de ce qu'il est appelé à faire dans son environnement de travail habituel.

D'autres travaux se sont attachés à modéliser la notion d'interface utilisateur d'une application interactive. La plupart des modèles proposés sont basés sur la séparation de la partie fonctionnelle d'une application de la partie chargée de la communication avec l'utilisateur. Ce principe a conduit à l'élaboration de systèmes de gestion d'interface utilisateur (« User Interface Management System » ou UIMS) dont le rôle consiste à soulager le concepteur d'application de la gestion du dialogue avec l'utilisateur. Les systèmes les plus récents offrent ainsi un ensemble d'outils de haut niveau englobant la gestion des aspects lexicaux et syntaxiques des commandes d'une application et la gestion des informations affichées par celle-ci [Rosenthal 83] [Coutaz 85].

Notre travail s'inscrit dans le cadre de ces deux approches et a porté sur la réalisation d'un système multi-fenêtres appelé Fenix qui permet à l'utilisateur de mener plusieurs activités en parallèle et qui offre les mécanismes de base nécessaires à la conception d'outils de dialogue sophistiqués.

Fenix autorise le recouvrement des fenêtres sur l'écran. Le recouvrement des fenêtres est entièrement transparent aux applications et est géré par une méthode qui rend les opérations de manipulation [des fenêtres] particulièrement performantes et; sur le plan visuel, très confortables pour l'utilisateur.

Le système Fenix est conçu pour que chaque utilisateur puisse adapter l'interface du système au style de communication qui lui convient, dans les limites des contraintes imposées par l'environnement. L'architecture du système repose pour cela sur une séparation stricte entre, d'une part la partie opératoire regroupant les fonctions de base (manipulation des fenêtres, primitives graphiques, suivi de la souris sur l'écran, etc.) et, d'autre part, la partie contrôle qui gère les commandes mises à la disposition de l'utilisateur.

Plan de la thèse

L'approche que nous avons suivie dans cette thèse est plus spécialement orientée vers les aspects « système d'exploitation » de l'interface homme-machine. Nous nous intéressons notamment aux problèmes que posent la prise en compte du parallélisme et la gestion des ressources partagées dans la conception d'un système multi-fenêtres.

La première partie de ce document (chapitre II) est consacrée à l'étude des postes de travail individuels. Nous présentons les caractéristiques de ce type de systèmes en insistant tout particulièrement sur leur interface utilisateur. Nous exposons le principe de fonctionnement d'un écran à points et d'une « souris » et les mécanismes logiciels spécifiques à leur utilisation.

La seconde partie (chapitre III) traite exclusivement des systèmes multi-fenêtres. Nous présentons les fonctions de ces systèmes et les différentes méthodes et techniques qui peuvent être adoptées pour leur réalisation. Nous commençons cette présentation en nous plaçant du point de vue de l'utilisateur du système. Nous analysons sous cet angle les principales méthodes pouvant être adoptées pour partager l'écran entre les fenêtres puis nous étudions l'interface de commande des systèmes multi-fenêtres.

Nous nous intéressons ensuite à l'interface de programmation des systèmes multi-fenêtres. Nous distinguons à ce niveau trois composants fonctionnels en étudiant successivement les fonctions de gestion des fenêtres en tant qu'objets manipulables globalement, l'interface graphique qui regroupe les opérations d'affichage et l'interface d'accès aux informations d'entrée émises par l'utilisateur. Nous décrivons ensuite sur la base d'exemples concrets les principaux types d'architectures de systèmes multi-fenêtres en mettant en relief leurs avantages et inconvénients respectifs.

La troisième partie (chapitre IV) est consacrée à la description du système Fenix. Nous présentons les objectifs que nous nous sommes fixés et l'architecture que nous avons adoptée pour essayer de les atteindre.

Nous décrivons ensuite dans le détail les différents composants du système. Nous exposons la méthode et les algorithmes que nous avons développés pour gérer le recouvrement des fenêtres. Nous présentons ensuite le principe appliqué pour gérer les dispositifs d'entrée et la façon dont sont transmis aux activités les

événements générés par le système. Nous terminons par une description de l'interface utilisateur du système.

Le chapitre V conclut cette thèse par une évaluation du système vis à vis des objectifs visés. Nous présentons à cette occasion les améliorations que nous envisageons d'introduire dans une prochaine version du système Fenix. Nous terminons par quelques réflexions sur les aspects spécifiques de la gestion des ressources d'un poste de travail individuel dans un contexte multi-activités.

Nous présentons en annexe le système Unix, annexe à laquelle nous faisons implicitement référence à chaque évocation d'Unix dans cette thèse. Nous y décrivons les principaux concepts de base du système Unix et les caractéristiques de son interpréteur de commandes (le *Shell*).

Chapitre II

LES POSTES DE TRAVAIL INDIVIDUELS

---

1. Présentation générale

Les années 1980 représentent pour les constructeurs et les utilisateurs le début d'une ère nouvelle de l'informatique placée sous l'égide de l'ordinateur individuel. Son prix relativement faible permet en effet de l'utiliser dans une gamme de plus en plus large d'activités professionnelles jusqu'alors inadaptées à la taille des systèmes en temps partagé classiques. L'ordinateur est ainsi devenu un *poste de travail individuel*.

Le terme même de poste de travail individuel recouvre en fait deux notions différentes, *usage personnel* et *outil de travail*, que nous présentons séparément.

1.1. Machine individuelle

L'apparition des microprocesseurs au début des années 1970 est à l'origine du développement des machines individuelles. Dans un premier temps, celles-ci sont destinées à des activités de loisir (initiation à l'informatique, jeux, etc.) ou à couvrir les besoins de petites structures économiques (gestion, traitement de texte). Les contraintes de coût fondamentales pour le marché visé et les possibilités technologiques de l'époque dictent alors l'architecture matérielle de ce type de machines: processeurs 8 bits, lecteurs de disquettes, terminal alphanumérique classique.

En raison de leurs performances limitées, les ordinateurs individuels ne peuvent rivaliser avec les systèmes en temps partagé au niveau du choix et de la puissance des services qu'ils proposent : systèmes de développement souvent réduits à un interpréteur du langage « BASIC », machines à traitement de texte par exemple. Par contre, les avantages que ce type de systèmes procurent à leurs utilisateurs sont justement liés à leur caractère *autonome* :

- les performances de fonctionnement sont constantes ;
- l'utilisation n'est pas facturée.

Depuis, les progrès continuels réalisés au niveau de l'intégration des composants (processeurs, mémoires) et des disques permettent de construire à des coûts de plus en plus faibles des machines performantes offrant une grande puissance de calcul et des capacités de stockage importantes.

1.2. Poste de travail

Parallèlement aux progrès technologiques qu'elle a entraînés, mais aussi grâce à eux, la machine individuelle a évolué vers ce qu'on appelle maintenant le *poste de travail individuel*. Cette notion reflète une nouvelle conception de l'utilisation de l'ordinateur qui devient l'outil de travail privilégié dans des domaines d'activité de plus en plus variés comme la bureautique, l'enseignement ou la conception assistés par ordinateur où son rôle consiste à améliorer le confort et la productivité de ses utilisateurs.

La satisfaction de telles exigences repose avant tout sur la conception de l'interface utilisateur du poste de travail. Ceci est d'autant plus vrai que la plupart des tâches qu'il permet d'accomplir le destine à être souvent mis à la disposition de non spécialistes en informatique.

C'est au Centre de recherches Xerox de Palo Alto (Xerox PARC) que fut introduit puis développé le concept même de poste de travail individuel et ce dès 1973 avec le projet Alto. L'objectif était de fournir sur une petite machine un environnement pour le développement de programmes et des services de type bureautique comme la préparation et l'impression de documents [Thacker 79]. La définition et la réalisation de la machine, processeur y compris, étaient inclus dans le projet initial. Les besoins des application visées et en particulier ceux liés à la nature fortement interactive de ces applications purent donc être pris en compte lors

de la conception de son architecture. La configuration standard de la machine Alto comprend :

- un processeur microprogrammé supportant différents jeux d'instructions et chargé également de contrôler les périphériques d'entrées-sorties ;
- une mémoire centrale de 64K mots de 16 bits chacun, extensible à 256K mots ;
- un disque amovible de 2,5M octets ;
- une interface vers le réseau Ethernet ;
- un écran à points de 875 lignes sur 608 points de définition ;
- un clavier et un mécanisme de désignation sur l'écran appelé « souris ».

Outre l'environnement de programmation pour le langage « BCPL » qui est le langage de base du système, les applications d'origine sont principalement orientées vers la production de documents textuels et/ou graphiques. Le réseau Ethernet est exploité pour offrir divers services comme la messagerie, l'impression de documents ou le stockage de fichiers. La machine Alto servit également de support expérimental à d'autres projets internes au PARC, notamment pour implanter les environnements de programmation des langages Mesa [Mitchell 79] et Smalltalk [Ingalls 78] développés à la même époque.

L'ensemble de ces projets est caractérisé par la place importante, sinon essentielle, accordée à la conception de l'interface utilisateur basée principalement sur l'utilisation de la souris et sur les possibilités graphiques offertes par l'écran de la machine Alto. De nombreuses innovations furent introduites à cette occasion comme la notion de *menu de commandes* ou celle d'*icône* permettant de symboliser un objet ou un service que l'utilisateur peut alors sélectionner à l'aide de la souris.

Les concepteurs de Smalltalk furent les premiers à promouvoir le principe de l'*interaction sans mode* pour résoudre le « dilemme de la préemption » rencontré dans les systèmes classiques. Dans ces systèmes, les faibles capacités du terminal (nombre de touches limité, écran non partageable) ne permettent pas en effet aux applications interactives ni surtout au système lui-même d'offrir en permanence à l'utilisateur l'accès direct à l'ensemble des commandes ou services disponibles (mode « insertion » dans les éditeurs par exemple) [Tesler 81].

Le projet Alto fut à l'origine de travaux importants sur l'ergonomie de l'interface utilisateur des postes de travail tout d'abord au PARC même avec par exemple le projet Cedar [Teitelman 85] et surtout le projet Star [Smith 82].

Le Star, ou Station de travail 8010, est le premier système de bureautique intégré apparu sur le marché en 1981. L'objectif initial du projet est de réaliser un « bureau tout électronique » permettant d'accomplir l'ensemble des tâches de secrétariat sans aucun support papier.

La machine conçue selon une architecture similaire à celle de l'Alto est toutefois plus performante que celle-ci : processeur trois fois plus rapide, écran à points de 809 lignes de 1024 points chacune, mémoire centrale de 512K octets (dont 100K dédiés à l'écran), etc.

Le système permet avant tout de manipuler des documents contenant à la fois du texte, du graphique, des tables et des formules mathématiques. Comme sur l'Alto, le réseau Ethernet est utilisé pour la messagerie et l'accès à des serveurs d'impression ou de stockage de documents.

Le projet Star consacra des moyens importants à la conception de l'interface utilisateur du système (l'équivalent d'environ trente hommes-année). Il comporta une première phase d'analyse consacrée exclusivement aux aspects *convivialité* et *ergonomie* du poste de travail. Cette étape permit de dégager et de spécifier les différents principes de base qui devaient être respectés par l'interface utilisateur. Parmis ceux-ci, citons notamment :

- ***Restituer le plus fidèlement possible l'environnement habituel de l'utilisateur potentiel du système.***

  Le système Star est bâti selon le principe de la métaphore du bureau: l'interface globale du système appelée « desktop » utilise des icônes pour représenter sur l'écran les différents objets ou services associés au travail de bureau : boîtes aux lettres pour la messagerie, armoires et classeurs pour la conservation des documents, etc.

- *Présenter les informations sur l'écran selon l'apparence qu'elles auront à l'impression* (« What you see is what you get » ou WYSIWYG).

  L'éditeur de documents du Star intègre également les fonctions de mise en page qui dans les systèmes classiques sont réalisées non interactivement par un programme de formatage de texte.

• *Uniformiser l'interface de commandes* en introduisant un jeu d'opérateurs ayant chacun la même sémantique à tous les niveaux du système : c'est le principe des commandes génériques.

Le clavier du Star inclut un ensemble de touches (« move », « copy », « delete », etc) applicables à tous les types d'objets définis dans le système. La commande « move » permet ainsi d'effectuer toutes les opérations correspondant à un transfert d'informations : déplacement d'une partie d'un document édité, archivage ou impression d'un document, envoi de messages, etc.

• *Interaction sans mode.*

Ce principe revendiqué à l'origine par les concepteurs de Smalltalk est appliqué de façon similaire dans le Star (système multi-fenêtres, édition sans mode, etc.).

Le système Star aura beaucoup d'influence sur la conception d'autres systèmes individuels, en particulier sur celle des systèmes Lisa [Williams 83] et Macintosh [Williams 84] développés chez Apple avec la participation de « transfuges » de chez Xerox.

Par ailleurs, de nombreux postes de travail orientés vers le développement de logiciel sont apparus sur le marché comme les machines Apollo, Sun ou Perq fonctionnant sous le système d'exploitation Unix.

2. Caractéristiques particulières

Depuis la machine Alto, l'architecture matérielle des postes de travail individuels a en fait peu évolué. Les progrès les plus significatifs ont été réalisés au niveau des performances : processeur plus puissant, mémoires centrale et secondaire plus importantes, meilleure définition de l'écran, etc.

Le système de la plupart d'entre eux intègre un système multi-fenêtres qui représente une caractéristique essentielle de l'interface utilisateur des postes de travail. Le chapitre suivant est consacré à l'étude des systèmes multi-fenêtres et aux problèmes que pose leur conception. Toutefois, la réalisation de tels systèmes s'appuie en général sur les possibilités offertes par l'*écran à points* et la *souris* qui sont intégrés maintenant à la plupart des postes de travail individuels. C'est

pourquoi nous exposons auparavant les principes de fonctionnement de ces deux éléments ainsi que les techniques logicielles spécifiques à leur utilisation.

2.1. Principe de fonctionnement d'un écran à points

Dans un écran à points, la surface d'affichage est organisée sous forme d'une matrice rectangulaire de points appelée trame ou *raster* en anglais pour « rectangular array of intensity samples ». A cette matrice est associée une zone mémoire appelée mémoire-écran qui contient la représentation binaire de l'intensité ou de la couleur de chaque point ou *pixel* : 1 bit par pixel pour les écrans monochromes, plusieurs bits par pixel pour les écrans polychromes (d'où le nom anglais de « bitmap »).

Un mécanisme matériel indépendant du processeur est utilisé pour rafraîchir la surface d'affichage : son rôle consiste à balayer entièrement la mémoire-écran à intervalles réguliers (30 à 60 fois par seconde) et à interpréter la « couleur » de chaque pixel pour l'émettre sous forme de signaux analogiques vers un moniteur vidéo.

Remarque : la suite de cette présentation est uniquement consacrée au cas des écrans monochromes, les plus répandus actuellement car plus simples à réaliser et moins gourmands en mémoire (100 Koctets pour une définition de 800 lignes de 1024 pixels chacune par exemple).

La représentation d'une matrice dans une mémoire par définition linéaire implique une organisation particulière de cette dernière imposée par le mécanisme de rafraîchissement. La solution couramment adoptée consiste à faire correspondre les N bits d'un mot de la mémoire à des pixels successifs d'une même ligne sur l'écran. Chaque ligne est ainsi représentée dans des mots consécutifs, le dernier mot d'une ligne précédant le premier mot de la ligne suivante.

L'origine étant supposée être le coin supérieur gauche de l'écran, ce qui est souvent le cas, on obtient la représentation décrite dans la figure 2.1 ci-dessous. Par rapport aux terminaux classiques, les écrans à points présentent de nombreux avantages :

- la possibilité de visualiser des images et de tracer des figures géométriques ;

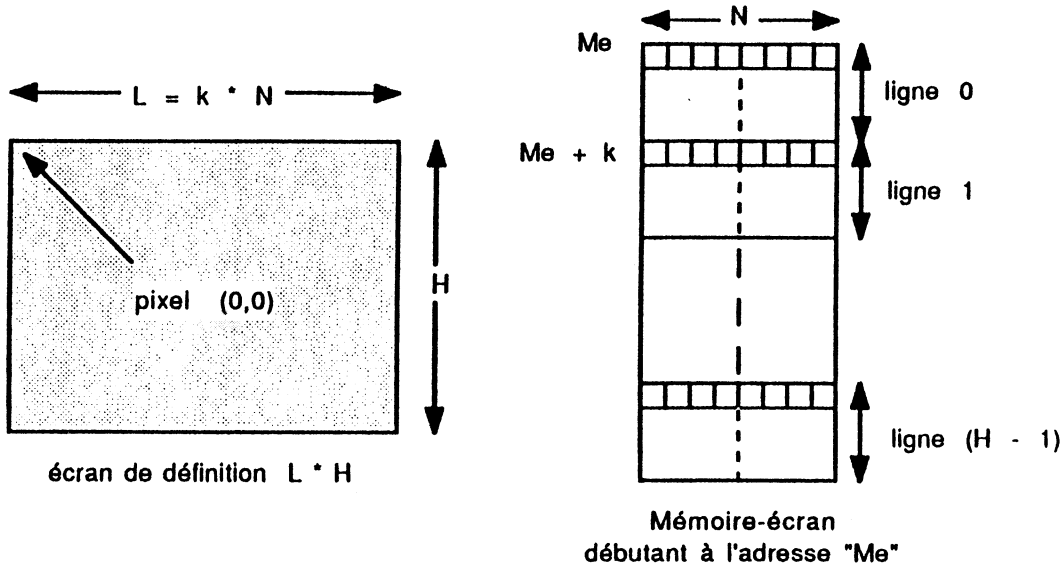


Figure 2.1 Principe d'un écran à points

• le jeu de caractères affichables n'est pas unique : l'écran à point autorise la représentation de caractères de différents alphabets, avec des tailles et des styles différents ;

• l'affichage est aléatoire : il n'y a pas de position courante d'écriture imposée par le matériel ;

• le contenu de l'écran est accessible en permanence et peut donc être utilisé comme source d'information en cas de besoin.

Sur un écran à points, toute opération d'affichage d'informations revient finalement à modifier le contenu de la mémoire-écran. Mais quelle que soit l'organisation adoptée pour celle-ci, l'accès à un pixel désigné par ses coordonnées (x, y) sur l'écran doit être réalisé en deux étapes : calcul de l'adresse du mot-mémoire contenant le bit correspondant puis accès au bit lui-même dans ce mot. En particulier, ce dernier accès n'est pas immédiat et nécessite l'exécution de plusieurs instructions-machine sur la plupart des processeurs.

Un ensemble de primitives de base appelées *RasterOp* est donc fourni aux applications pour masquer ce type de calculs [Newman 79]. En fait, les « RasterOp » offrent essentiellement des fonctions de manipulation de rectangles de points qui permettent de réaliser une grande partie des opérations d'affichage nécessaires aux applications.

Les rectangles manipulés pouvant être représentés en dehors de la mémoire-écran, les « RasterOp » définissent le type *matrice de points* qui décrit l'organisation utilisée pour représenter un rectangle de points en mémoire. En général, une *matrice de points* comprend au moins l'adresse de la zone mémoire contenant les points de la matrice, la largeur (en bits ou en octets) d'une ligne de la matrice et la hauteur de celle-ci. Leur interface d'utilisation spécifie également d'autres types de base, comme par exemple les types *point* et *rectangle*.

En général, les manipulations offertes par les « RasterOp » permettent d'assurer les opérations suivantes :

L'affichage de caractères

Les caractères d'un même alphabet sont regroupés par *police*. Une police est une structure de données en mémoire qui contient :

- l'aspect visuel de chaque caractère sous forme d'une matrice de points ;
- des informations décrivant la hauteur et la largeur de chaque caractère, la position relative de la matrice par rapport au point d'affichage, etc.

Dans son principe de base, l'affichage d'un caractère en un point (x, y) de l'écran est donc une *copie* d'un rectangle d'une matrice de points source vers un autre rectangle d'une matrice de points destination. Pour obtenir différents effets visuels, il est souvent utile de pouvoir également réaliser une fonction logique (OU, ET, NON, OU exclusif, etc) entre chaque point des deux rectangles.

Les informations de description contenues dans les polices sont utilisées par les primitives d'affichage de caractères pour piloter cette opération de copie. Les programmes d'application comme par exemple les éditeurs ou les formateurs doivent également pouvoir accèder à ces informations pour adapter la présentation des informations qu'ils manipulent à la surface d'affichage disponible.

La largeur des caractères d'une même police constitue un paramètre important pour ceux-ci et on distingue souvent les polices dites à *châsse fixe* de celles dites à *châsse variable* selon que tous leurs caractères ont ou non la même largeur. Les polices à châsse fixe permettent en effet de précalculer le nombre de caractères affichables dans une ligne, ce qui est par définition impossible avec les polices à châsse variable où ce calcul doit être à priori fait au fur et à mesure.

Remarque : bien qu'elles soient à l'origine dédiées aux caractères d'imprimerie, les polices peuvent être également utilisées pour décrire des icônes ou les différents motifs du curseur associé aux déplacements de la souris.

Défilement, mise en évidence d'information

Le défilement consiste à déplacer verticalement ou horizontalement un rectangle sur l'écran. Ce déplacement peut lui aussi être réalisé par une opération de *copie* d'un rectangle source vers un rectangle destination. Les rectangles appartenant à la même matrice de points et pouvant se recouvrir, un certain ordre doit être respecté pour le transfert en mémoire afin de garantir la correction de l'opération. La partie de l'écran libérée par le rectangle déplacé doit le plus souvent être effacée, ce qui nécessite une opération de *remplissage* affectant une même valeur à tous les points d'un rectangle.

L'information sélectionnée par l'utilisateur est souvent mise en évidence en *inversant* tous les points du rectangle l'englobant sur l'écran.

Affichage d'objets graphiques et d'images

Outre les diverses fonctions de manipulation de rectangles, les « RasterOp » peuvent inclure le tracé de différentes figures géométriques de base telles que les segments de droite, les cercles, etc. Leur réalisation fait appel à un certain nombre de techniques et d'algorithmes spécifiques [Foley 82] dont l'étude sort du cadre de cette thèse. Notons toutefois que le tracé de segments de droite est le minimum requis par les gestionnaires de fenêtres qui l'utilisent pour réaliser le bord des fenêtres. Du point de vue du système, une *image* est composée d'un ensemble non structuré de points et l'opération de copie entre matrices de points permet d'assurer l'affichage d'images.

Quelque soit le type d'information qu'elles manipulent, il est souhaitable que toutes ces primitives de base intègrent la fonction de *coupage* appelée « clipping » en anglais et qui consiste à déterminer et à n'afficher que la partie visible d'une information par rapport à un rectangle de base. Cette fonction joue tout d'abord le rôle d'un mécanisme de protection vis à vis des programmes d'application, mais évite également à ces derniers d'avoir à gérer ce type de calculs lorsque la visualisation partielle d'une information est justement l'effet désiré.

De par leur rôle, les « RasterOp » sont utilisées fréquemment ; leur efficacité conditionne donc les performances du système et notamment ses capacités de réaction visuelle vis à vis de l'utilisateur du poste de travail. Pour offrir un temps d'exécution optimal, les versions purement logicielles des « RasterOp » sont généralement codées en langage d'assemblage. Ce soucis d'optimisation peut également conduire à la réalisation microprogrammée des « RasterOp » comme par exemple sur les machines Alto ou Perq ou même à la conception de processeurs spécialisés [Bennett 85] [Stock 86]. Une autre approche consiste à rechercher une organisation optimale de la mémoire-écran permettant de manipuler directement des rectangles de points [Gupta 81].

2.2. Principe de fonctionnement d'une souris

La plupart des postes de travail offrent un mécanisme de désignation appelé *souris* qui est un petit boîtier indépendant que l'utilisateur peut déplacer sur une surface plane. Ces déplacements sont captés à l'aide d'un dispositif particulier placé sous le boîtier et traduits électroniquement en valeurs numériques. En général, le processeur central accède à ces valeurs dans des registres de données associés au contrôleur de la souris.

Ces valeurs peuvent représenter une position absolue si le dispositif le permet: c'est le cas sur le Perq qui utilise une tablette magnétique sur laquelle doit évoluer le boîtier, les valeurs délivrées représentant la position du boîtier sur la tablette. Toutefois, les déplacements sont le plus souvent délivrés sous forme de valeurs (dx, dy) relatives comme sur les souris des machines Star, Lisa et Macintosh où le capteur est une boule ou sur celle du Sun qui utilise un dispositif optique. Notons que dans ce cas, le fait de soulever le boîtier et de le poser à un autre endroit ne provoque aucun déplacement effectif.

Un ou plusieurs boutons, au plus trois en général, sont placés sur le dessus du boîtier. A chaque bouton sont associés deux états possibles, « enfoncé » et « relâché » ; l'état courant des boutons de la souris est également accessible en permanence dans un registre d'état du contrôleur de la souris.

La position de la souris est visualisée sur l'écran à l'aide d'un motif que nous appellerons *réticule* par la suite. D'un point de vue interface utilisateur, il est intéressant de pouvoir changer le réticule en fonction de la position de la souris pour distinguer visuellement des contextes de travail différents. Un contexte est en

général défini par une zone rectangulaire et peut par exemple correspondre à l'opération associée au bouton de la souris dans cette zone. Par ailleurs, il est souvent utile de pouvoir associer différentes figures géométriques aux déplacements de la souris :

- la *ligne élastique* (« rubberband ») est un segment de droite reliant un point fixe de l'écran à la position courante de la souris. La longueur et l'orientation de ce segment varient donc en fonction des déplacements de la souris ;

- le *cadre élastique* (« rubberbox ») est un rectangle dont l'un des sommets est fixe et le sommet diagonalement opposé associé à la position courante de la souris. La taille de ce rectangle varie donc avec la position de la souris ;

- le *cadre mobile* est un rectangle de taille fixe, mais programmable, dont l'un des sommets est aussi associé à la position de la souris.

Les gestionnaires de fenêtres utilisent notamment le cadre élastique et le cadre mobile pour permettre à l'utilisateur de spécifier à l'aide de la souris la taille ou la position d'une fenêtre. Nous distinguerons par la suite deux types d'actions que peut effectuer l'utilisateur avec la souris :

- *désigner* en déplaçant la souris à un point particulier de l'écran ;

- *sélectionner* en agissant sur un (au moins) des boutons de la souris.

Ces deux actions sont en effet interprétées différemment dans la majorité des systèmes. *Désigner* ne modifie pas le contexte de travail courant de l'utilisateur, c'est une action immédiate dont l'effet est uniquement visuel - changement du réticule, mise en évidence de l'objet désigné, etc. - et dépend du contexte d'interprétation de la position de la souris. *Sélectionner* au contraire implique systématiquement un changement d'état au niveau du contexte où cette action est interprétée, comme par exemple la modification du point d'insertion des caractères.

Le réticule doit refléter les déplacements de la souris en évoluant « au dessus » de l'information affichée sur l'écran par les applications. Pour cela, le contenu de l'écran et le réticule sont considérés logiquement comme deux plans-mémoire distincts affichés par superposition à la surface de l'écran.

Lorsque l'affichage du réticule est assurée par logiciel, ce qui est encore souvent le cas, le programme affecté à cette tâche doit réaliser cette superposition dans la mémoire-écran. Pour garantir la cohérence du contenu de l'écran, ce

programme que nous appellerons *Suivre_Souris* par la suite doit exécuter l'algorithme suivant :

- *restituer* la partie modifiée par l'affichage du réticule à l'ancienne position de la souris ;

- *sauvegarder* la partie modifiée à la nouvelle position de la souris ;

- *afficher* le réticule à cette nouvelle position ;

Deux solutions peuvent être adoptées pour implanter cet algorithme :

1) affichage du réticule par application de la fonction logique « OU exclusif » entre sa représentation en mémoire hors-écran et les points du rectangle correspondant dans la mémoire-écran. la restitution consiste à appliquer une deuxième fois la même fonction avec les mêmes paramètres ;

2) copie complète en mémoire hors-écran de la partie modifiée puis affichage direct du réticule dans la mémoire-écran. La restitution est alors également une opération de copie ;

Comparons ces deux solutions. Elles ont tout d'abord des effets visuels différents : la solution par copie assure la restitution exacte sur l'écran de la forme du réticule qui dans l'autre cas est mélangée avec le reste des informations affichées. Au niveau de la réalisation, la solution par copie est par contre plus coûteuse en temps puisqu'elle nécessite l'exécution de trois opérations et en mémoire pour la sauvegarde des parties modifiées.

Quelle que soit la solution adoptée, le programme *Suivre_Souris* doit être exécuté sur interruption pour éviter l'attente active. Cette interruption peut être déclenchée par le contrôleur de la souris à la suite d'un déplacement du boîtier ; si ce type de mécanisme n'est pas disponible, une autre solution consiste à exécuter *Suivre_Souris* à intervalles de temps réguliers fournis par une horloge.

Le programme *Suivre_Souris* pouvant a priori interrompre une primitive d'affichage exécutée pour le compte d'une application, un mécanisme d'exclusion mutuelle doit alors être utilisé pour garantir la cohérence de la mémoire-écran. L'exclusion mutuelle peut porter systématiquement sur tout l'écran ou être restreinte à la partie affectée par la primitive d'affichage courante.

Pour éliminer tous ces problèmes, un mécanisme matériel indépendant peut être chargé d'afficher le réticule. La solution la plus courante consiste à ranger le réticule en dehors de la mémoire-écran dans une zone spéciale associée au mécanisme de rafraîchissement de l'écran qui peut alors effectuer lui-même la superposition de ces deux plans avant d'en transmettre le résultat sous forme de signaux vidéo. En général, les constructeurs adoptent l'équivalent visuel de l'une des deux solutions utilisées pour la réalisation logicielle de la superposition.

Le rôle du programme *Suivre_Souris* se résume dans ce cas à gérer l'adresse de la souris sur l'écran et à la fournir au mécanisme de rafraîchissement dans des registres de contrôle prévus à cet effet.

Chapitre III

SYSTEMES MULTI-FENETRES

---

1. Introduction

Actuellement, de nombreux postes de travail individuels sont munis d'un écran à points et les systèmes qui les gèrent intègrent de plus en plus souvent un gestionnaire de fenêtres. Si l'utilité d'un tel outil semble ainsi faire l'unanimité, il n'en va pas de même de sa conception qui varie considérablement d'un système à l'autre. Celle-ci s'appuie néanmoins sur un principe de base commun :

> s'affranchir des limites imposées par le terminal en offrant une surface d'affichage plus importante que celle de l'écran physique. Une fenêtre représente donc un écran virtuel et la fonction du gestionnaire de fenêtres est d'assurer le partage de l'écran entre les fenêtres.

Ce principe étant acquis, c'est dans son application qu'apparaissent les différences, sinon les divergences, entre les systèmes développés jusqu'à maintenant.

L'objet de ce chapitre est de dégager et d'analyser les principes généraux des systèmes multi-fenêtres et les différentes approches qui peuvent être envisagées pour leur mise en œuvre. Nous considèrerons également au cours de cette étude le problème posé par l'intégration d'un système multi-fenêtres dans un système d'exploitation existant.

Il s'agit en effet d'introduire des concepts nouveaux tout en restant compatible avec la philosophie initiale du système. En d'autres termes, il faut enrichir le système sans perdre aucune des facilités offertes à l'origine et notamment permettre l'exploitation immédiate de toutes les applications développées auparavant. Nous avons choisi de traiter cet aspect en prenant comme exemple le système Unix qui a servi de support à la réalisation de nombreux systèmes multi-fenêtres et en particulier à celle du système Fenix.

Les environnements de programmation Smalltalk [Ingalls 78] et Interlisp [Teitelman 81] furent parmi les premiers à intégrer un système multi-fenêtres. L'objectif du système est de permettre à l'utilisateur d'accomplir plusieurs tâches à la fois et de passer de l'une à l'autre facilement. Celui-ci doit pouvoir démarrer plusieurs activités en parallèle et choisir à tout instant celle avec laquelle il désire communiquer, l'*activité courante*, sans pour autant perdre le contexte des autres activités et notamment les informations affichées par celles-ci sur l'écran. Chaque fenêtre représente alors le contexte visuel d'une activité et la *fenêtre courante* est la fenêtre à travers laquelle l'utilisateur communique à un instant donné.

Cette démarche vise en fait à appliquer au niveau même du système le principe d'« interaction sans mode » cher aux concepteurs de Smalltalk : le système multi-fenêtrés permet en effet de donner à l'utilisateur l'accès permanent à tous les services offerts par le système [Tesler 81].

Un certain nombre de systèmes multi-fenêtres développés depuis ont été conçus dans un but identique. C'est le cas notamment pour les systèmes Bruwin [Meyrowitz 81], Wm [Jacob 82] ou le Maryland Window System [Weiser 83] adaptés au système Unix et conçus pour des terminaux alphanumériques classiques. Dans ces systèmes, chaque fenêtre représente alors un terminal virtuel permettant l'exécution de plusieurs *Shell* en parallèle.

Il ne faut pas pour autant assimiler systématiquement multi-fenêtres et multi-activités. Certains systèmes mono-activité intègrent néanmoins un système multi-fenêtres comme celui du Macintosh qui constitue un des exemples les plus connus. Dans ce cas, les applications ou le système peuvent utiliser plusieurs fenêtres pour séparer sur l'écran des informations de natures différentes. Par exemple, un outil d'aide à la mise au point de programmes peut présenter le programme en langage source dans une fenêtre, les sorties d'informations effectuées par ce programme dans une autre fenêtre, etc.

Nous pouvons donc distinguer deux objectifs à la mise en œuvre d'un système multi-fenêtres [Joloboff 84]:

- permettre à l'utilisateur de mener plusieurs activités indépendantes en parallèle ;

- permettre le développement d'applications interactives disposant de plusieurs surfaces d'affichage indépendantes pour dialoguer avec l'utilisateur.

Ces deux objectifs ne sont pas incompatibles et de nombreux systèmes multi-fenêtres sont conçus de façon à offrir les deux avantages précédents. C'est le cas du système Vitrail développé dans le cadre du projet Kayak à l'INRIA [Wegmann 83], de Sapphire [Myers 84] et de nombreux systèmes multi-fenêtres adaptés à Unix comme par exemple Oriel [Sweetman 86], SunWindows [Sun 85], Apôtre [Cany 85] et W [Van Der Linden 85].

Le système multi-fenêtres joue le rôle de médium de communication entre l'utilisateur connecté au système et les activités avec lesquelles l'utilisateur dialogue. Nous pouvons donc distinguer deux types d'utilisateurs du système :

- l'*utilisateur* proprement dit, c'est à dire la personne qui communique depuis son terminal avec le système et les activités ;

- le *programmeur* d'applications qui utilise les services offerts par le système pour gérer la communication avec l'utilisateur.

Nous avons choisi de suivre cette distinction pour présenter les principaux aspects des systèmes multi-fenêtres. La première partie de ce chapitre est consacrée à l'étude de *l'interface utilisateur* des systèmes multi-fenêtres. Dans la seconde partie, nous nous intéressons à l'*interface de programmation* de ces systèmes, c'est à dire à l'ensemble des fonctions qu'ils offrent au programmeur d'applications pour gérer la communication avec l'utilisateur. Enfin, nous étudions dans la troisième partie l'aspect mise en œuvre des systèmes multi-fenêtres. Nous décrivons les principales architectures qui peuvent être choisies et leurs avantages et inconvénients respectifs.

2. L'interface utilisateur

C'est l'aspect le plus important dans la conception d'un gestionnaire de fenêtres qui constitue par définition le composant de base de l'interface du système. Nous étudions l'interface utilisateur d'un système multi-fenêtres selon l'approche introduite précédemment où nous avons distingué le principe du partage de l'écran de l'objectif de sa mise en œuvre dans un système multi-fenêtres.

Le partage de l'écran

Nous considérons à ce niveau le gestionnaire de fenêtres comme une entité isolée du reste du système pour nous intéresser uniquement à la manière dont les fenêtres peuvent être disposées les unes par rapport aux autres sur l'écran. Cet aspect concerne essentiellement l'utilisateur car c'est à lui que doit revenir finalement le contrôle sur la gestion de l'écran.

L'interface de commandes

Le rôle du gestionnaire de fenêtres ne se limite pas à la gestion de l'écran : c'est en effet le médium de communication privilégié entre l'utilisateur et le reste du système et c'est par son intermédiaire que l'utilisateur accède aux différentes applications et communique avec elles. Le gestionnaire de fenêtres doit dans ce cas être envisagé dans le contexte général du système. Nous étudions cet aspect à travers l'interface de commandes puisque c'est à ce niveau que l'utilisateur perçoit le fonctionnement de l'ensemble du système et implicitement ses objectifs de conception.

2.1. Partage de l'écran

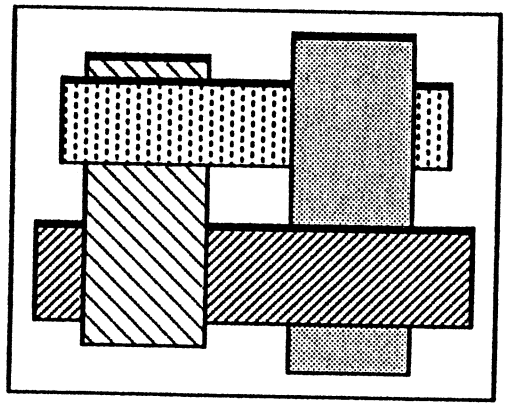
Nous présentons successivement les principales méthodes qui peuvent être adoptées pour gérer le partage de l'écran entre les fenêtres en distinguant celles qui autorisent leur recouvrement sur l'écran de celles qui ne le permettent pas.

2.1.1. Partage avec recouvrement

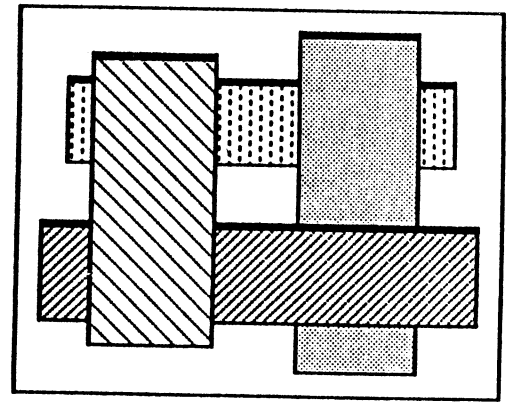
Le système permet aux fenêtres de se recouvrir entièrement ou partiellement sur l'écran : l'utilisateur peut ainsi choisir la position ou la taille de chaque fenêtre et l'ordre selon lequel les fenêtres se recouvrent sur l'écran.

Il nous faut préciser la manière dont cet ordre est envisagé au niveau du système et donc la manière dont il est perçu par l'utilisateur. Nous supposerons par la suite que toutes les fenêtres appartiennent à des plans différents « parallèles » à la surface de l'écran et que le système gère l'ensemble des fenêtres comme une « pile ».

Les fenêtres ne peuvent donc pas être disposées de façon à former un cycle comparable dans son principe à ceux obtenus par Escher dans certaines de ses lithographies. La figure 3.1 ci-dessous explicite les deux principes.



cycle de fenêtres



pile de fenêtres

Figure 3.1 Cycle de fenêtres et pile de fenêtres

La position respective des fenêtres dans la pile n'est significative que pour celles qui se recouvrent effectivement. Nous pouvons alors définir les opérations suivantes permettant de changer l'ordre de superposition des fenêtres :

- *faire monter* une fenêtre : placer la fenêtre au sommet de la pile des fenêtres et la rendre ainsi complètement visible ;

• *faire descendre* une fenêtre : placer la fenêtre en bas de la pile et n'en visualiser que les parties non recouvertes par toutes les autres ;

• *changer le rang* d'une fenêtre : insérer la fenêtre à une place quelconque dans la pile. La visibilité de la fenêtre dépend alors des fenêtres situées au dessus. Les deux opérations précédentes sont des cas particuliers de cette dernière.

La majorité des systèmes développés jusqu'à maintenant autorisent le recouvrement des fenêtres sur l'écran. Outre les exemples de Smalltalk, de Vitrail ou du Macintosh évoqués précédemment, citons également le système WHIM réalisé dans le cadre du projet QuickSilver au centre de recherches IBM de San Jose [Goodfellow 85] et la plupart des systèmes multi-fenêtres adaptés aux différentes versions du système Unix tels que PNX-WMS [Perq 84], le terminal graphique Blit [Pike 84b], Apollo [Apollo 85], SunWindows [Sun 85], Apôtre [Cany 85], W [Van Der Linden 85], X [Rosenthal 86], etc.

Bien qu'adoptant le même principe, tous ces systèmes présentent des différences de mise en œuvre interne importantes. Notre propos n'est pas d'étudier ici les diverses solutions qui peuvent être choisies pour implanter la gestion du recouvrement, mais de montrer que leurs différences de conception ne peuvent pas toujours être masquées à l'utilisateur lorsqu'il manipule les fenêtres sur l'écran.

Ces différences ne portent pas sur la logique des opérations qui reste la même dans tous les systèmes mais apparaissent à travers les effets visuels que leur déroulement provoque sur l'écran. Pour illustrer cet aspect ergonomique de la mise en œuvre du partage de l'écran avec recouvrement, nous avons choisi de comparer l'opération de destruction d'une fenêtre du terminal Blit et du système WHIM. Nous nous intéressons donc dans ce cas à la manière dont la fenêtre détruite est « effacée » de l'écran.

Sur le terminal Blit, la destruction d'une fenêtre se déroule ainsi :

- faire monter la fenêtre ;
- repeindre le rectangle qu'elle occupe sur l'écran avec la couleur de fond associée à l'écran ;
- faire descendre la fenêtre ;

Cette dernière opération est réalisée en faisant monter successivement chacune des autres fenêtres à partir de la plus basse. Cette séquence est particulièrement « lourde » et ses effets visuels pour le moins déconcertants pour l'utilisateur qui ne demande en fait qu'à voir disparaître les parties visibles de la fenêtre détruite et à ne voir s'afficher de nouveau que les parties des autres fenêtres recouvertes précédemment.

C'est ce que fait le système WHIM qui contient une opération spécifique pour faire descendre une fenêtre, opération qui ne modifie sur l'écran que le strict nécessaire. Cette opération est utilisée pour détruire une fenêtre d'une manière plus « naturelle » pour l'utilisateur selon la démarche suivante :

- faire descendre la fenêtre ;
- repeindre ses parties visibles avec la couleur de fond de l'écran ;

Dans cet exemple, la différence essentielle provient de la solution adoptée pour faire descendre une fenêtre. Cette opération est une primitive de base du système WHIM, c'est à dire qu'elle est implantée au plus bas niveau de façon à exploiter toutes les possibilités d'optimisation qu'offre par définition un tel niveau. Sur le Blit au contraire, elle est construite à partir des opérations de l'interface de base et doit donc manipuler globalement d'autres fenêtres pour arriver au même résultat.

Cette remarque est valable pour toutes les opérations de manipulation des fenêtres. Par exemple, l'interface de base de Sapphire inclut l'opération de changement de rang d'une fenêtre (celle qui permet d'insérer une fenêtre à n'importe quelle position dans la pile). La plupart des systèmes permettent uniquement de faire monter ou descendre une fenêtre et l'opération de changement de rang doit alors être réalisée en faisant monter ou descendre d'autres fenêtres. Les mêmes motivations peuvent conduire à inclure également au niveau de l'interface de base une opération permettant de modifier simultanément la taille, la position et le rang d'une fenêtre.

2.1.2. Partage sans recouvrement

Le principe de cette méthode consiste à disposer toutes les fenêtres dans un même plan - le plan de l'écran - en partageant la surface de l'écran de façon à ce que les fenêtres soient toujours complètement visibles. Avec cette approche, le choix de la taille et de la position des fenêtres et éventuellement de leur présence ou non sur

l'écran ne peut plus être uniquement du ressort de l'utilisateur qui doit abandonner au système une partie de ses prérogatives en la matière. Différentes stratégies peuvent être adoptées par ce dernier pour répartir les fenêtres sur l'écran :

Organisation en colonnes

L'écran est divisé en plusieurs colonnes et les fenêtres sont disposées verticalement dans chacune d'elles. C'est la méthode utilisée par exemple dans le système Microsoft Windows [Lemmons 83a] et dans le système Cedar développé chez Xerox [Teitelman 85]. Ce dernier divise l'écran en deux colonnes au plus et permet à l'utilisateur de choisir la répartition de la surface de l'écran entre les colonnes (voir figure 3.2 ci-dessous).

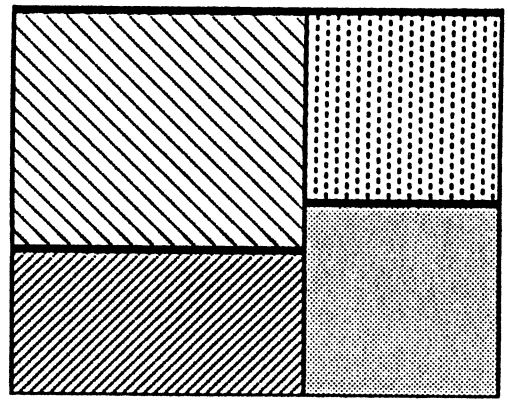


Figure 3.2 Répartition des fenêtres en deux colonnes

Le système Cedar se réserve le bas de l'écran pour y afficher des icônes symbolisant les différents services mis à la disposition de l'utilisateur. Lorsque celui-ci active un service, une fenêtre est créée et l'icône correspondante disparaît du bas de l'écran. Par défaut, le système partage automatiquement la surface d'une colonne entre toutes les fenêtres qu'elle contient : la création et la destruction d'une fenêtre s'accompagne donc d'une modification de la taille de ses voisines.

L'utilisateur peut déplacer une fenêtre en haut ou en bas de la colonne où elle se trouve. Il peut également déplacer une fenêtre dans le bas de l'autre colonne, auquel cas la largeur de la fenêtre est adaptée à celle de la colonne. L'utilisateur ou l'application peuvent modifier la hauteur d'une fenêtre pour lui attribuer par exemple la surface totale d'une colonne. Les autres fenêtres de cette colonne sont

alors automatiquement détruites et représentées par une icône dans le bas de l'écran.

Une stratégie similaire est utilisée dans la deuxième version du gestionnaire de fenêtres du système réparti Andrew développé dans le cadre du projet ITC [Morris 86]. Toutefois, une politique différente est appliquée pour la répartition d'une colonne entre les fenêtres : la surface totale n'est plus systématiquement partagée mais attribuée selon les besoins nécessaires, l'espace inutilisé étant visualisé en bas de chaque colonne. Cet espace libre permet d'éviter le plus souvent que la manipulation d'une fenêtre n'affecte d'autres fenêtres de l'écran.

Organisation hiérarchique

L'écran est divisé horizontalement ou verticalement en partitions, chaque partition pouvant être récursivement divisée selon le même principe. Cette décomposition peut être représentée sous forme d'un arbre binaire dont les feuilles correspondent aux fenêtres (voir figure 3.3 ci-dessous).

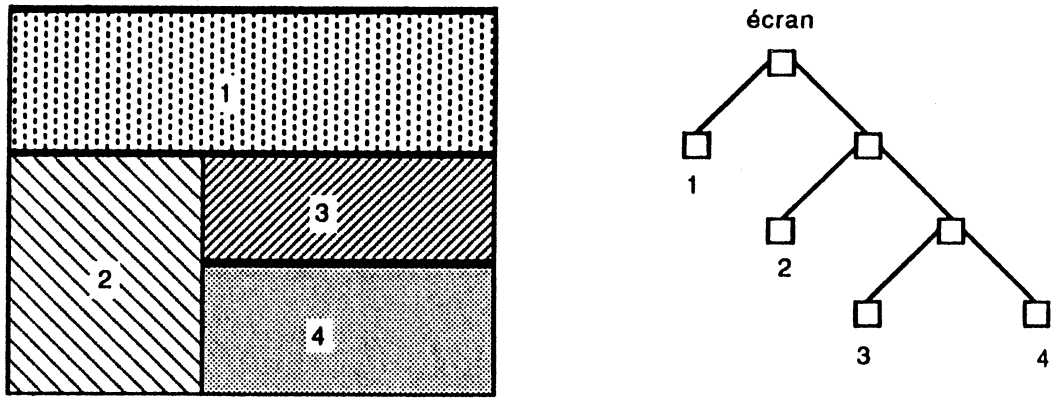


Figure 3.3 Répartition hiérarchique des fenêtres

Cette méthode fut choisie dans la première version du système Andrew évoqué précédemment [Gosling 84]. Dans cette version, la surface de l'écran est toujours complètement allouée. Une taille *minimum* et une taille *souhaitée* sont associées à une fenêtre lors de sa création. Chaque fenêtre est créée aux dépens d'une autre choisie de façon à respecter au maximum les contraintes de taille des fenêtres déjà présentes sur l'écran.

L'utilisateur peut modifier simultanément la position et la taille d'une fenêtre en spécifiant avec la souris deux points sur l'écran. Cette information ne sert toutefois que d'indication pour le système qui seul décide de la taille et de la position finales de la fenêtre afin de minimiser les modifications qu'il doit effectuer dans la hiérarchie et donc le nombre de fenêtres affectées par l'opération demandée.

Organisation non hiérarchique

La décomposition de l'écran peut être arbitraire et n'est plus imposée comme dans les organisations précédentes. Cette méthode est utilisée dans le système RTL/CRTL développé chez Siemens [Cohen 85].

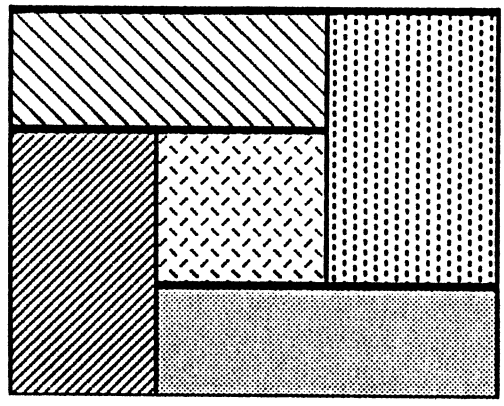


Figure 3.4 Répartition non hiérarchique des fenêtres

Le partage de l'écran est assuré automatiquement par le système en fonction de contraintes spécifiées par les applications ou l'utilisateur. Ces contraintes concernent essentiellement la taille minimum, l'emplacement et la présence obligatoire d'une fenêtre sur l'écran. Les applications et l'utilisateur peuvent également assigner une priorité aux contraintes associées à chaque fenêtre pour aider le système à résoudre les incompatibilités éventuelles. Cette méthode permet de réduire les relations de dépendance entre les fenêtres et laisse donc à l'utilisateur une plus grande liberté pour la manipulation des fenêtres sur l'écran.

2.1.3. Comparaison des deux approches

Le partage de l'écran avec recouvrement est souvent considéré comme le mieux adapté a priori aux besoins de l'utilisateur. En fait, deux critères essentiels sont utilisés pour évaluer les avantages et inconvénients respectifs des deux approches [Bly 86] :

1) la contribution apportée par le système pour aider l'utilisateur à gérer la disposition des fenêtres sur l'écran ;

2) la capacité d'adapter la taille des fenêtres à la nature et au volume des informations qu'elles contiennent.

Les promoteurs du partage de l'écran sans recouvrement invoquent le premier critère pour justifier leur choix. Cette méthode conduit en effet le système à choisir le plus souvent la taille et la position des fenêtres, ce qui réduit d'autant les interventions de l'utilisateur lors de la manipulation des fenêtres sur l'écran. Ce principe est en général apprécié par les utilisateurs peu expérimentés à qui la gestion de l'écran apparaît surtout comme une charge et une source d'erreurs supplémentaires qui ne peuvent alors que gêner leur apprentissage du fonctionnement du système. Il peut également être préféré des utilisateurs confirmés qui ne souhaitent pas être perturbés par la gestion de l'écran au cours de leur travail.

La *non-indépendance des fenêtres* les unes par rapport aux autres est le principal inconvénient du partage sans recouvrement : toute manipulation d'une fenêtre sur l'écran modifie souvent la taille d'autres fenêtres ou entraîne éventuellement la destruction de certaines d'entre elles. Cette solution ne satisfait donc pas aux exigences du deuxième critère énoncé précédemment. La prise en compte de contraintes permet d'assouplir le caractère arbitraire du système à ce niveau mais leur gestion peut s'avérer compliquée pour l'utilisateur et augmente ses interventions, ce qui va à l'encontre de l'objectif visé initialement.

Le principe du partage avec recouvrement résout par définition ce problème puisque les fenêtres sont dans ce cas complètement indépendantes les unes des autres. Inversement, les détracteurs de cette solution lui reprochent essentiellement d'obliger l'utilisateur à assumer lui-même la gestion de l'écran. Vis à vis de l'utilisateur, cette contrainte constituerait en quelque sorte la contrepartie à l'entière liberté dont il dispose pour manipuler les fenêtres.

En fait, aucune caractéristique inhérente au principe du partage avec recouvrement ne permet de justifier le bien-fondé de cette affirmation qui s'appuie uniquement sur la faiblesse de l'interface utilisateur de certains systèmes à ce niveau. Il est en effet possible de permettre le recouvrement des fenêtres sur l'écran tout en soulageant l'utilisateur de la plupart des décisions et des manipulations correspondantes en matière de gestion d'écran.

C'est par exemple l'approche suivie sur le Macintosh : le système comme les principales applications (en particulier les différents éditeurs tels que MacWrite, MacPaint, MacDraw, etc.) choisissent ainsi une taille et une position par défaut pour les fenêtres qu'ils utilisent. Qui plus est, les modifications éventuelles effectuées par l'utilisateur sur la taille ou la position des fenêtres du système sont conservées par celui-ci entre deux sessions dans un « fichier de ressources » qui sert ainsi de contexte d'initialisation.

Outre les deux critères précédents, d'autres de moindre importance sont évoqués pour comparer les deux méthodes de partage de l'écran.

Tout d'abord, les systèmes interdisant le recouvrement évitent à l'utilisateur de « perdre » une fenêtre en la masquant complètement par une autre, ce qui peut se produire éventuellement avec l'autre méthode. Cet inconvénient reste toutefois limité dans la mesure où le système offre en permanence à l'utilisateur un accès indirect aux fenêtres recouvertes involontairement. Une solution consiste par exemple à généraliser à ce type de fenêtres le principe de représentation et d'accès adopté pour les fenêtres explicitement cachées par l'utilisateur, c'est à dire celles dont il a demandé la disparition temporaire de l'écran.

Les différentes stratégies de partage sans recouvrement optimisent le taux d'occupation de la surface de l'écran, le plus souvent en répartissant systématiquement toute cette surface entre les fenêtres. Cette technique a par l'inconvénient d'attribuer éventuellement à certaines fenêtres une taille supérieure à celle effectivement utile. Elle peut également ne pas convenir aux utilisateurs qui préfèrent laisser un espace entre les fenêtres pour mieux les distinguer sur l'écran.

Un dernier argument nous permet de conclure ici cette comparaison qui fait ressortir la supériorité des systèmes autorisant le recouvrement des fenêtres : il est toujours possible d'adapter au dessus de ces systèmes une interface qui assure si besoin est le partage sans recouvrement (« qui peut le plus peut le moins »).

2.2. L'interface de commandes

Nous avons choisi d'étudier l'interface de commandes des gestionnaires de fenêtres en décrivant séparément les fonctions offertes par le système et les moyens mis à la disposition de l'utilisateur pour exprimer les commandes correspondantes. Cette démarche reflète le principe de conception des applications interactives (application au sens large, c'est à dire y compris le système) qui s'impose actuellement [Hayes 85].

2.2.1. Description des fonctions

L'interface de commandes d'un système multi-fenêtres comprend en fait trois sous-ensembles fonctionnels que nous présentons séparément :

- la gestion des activités. C'est à ce niveau que nous aborderons la création et la destruction des fenêtres car ces deux opérations sont étroitement liées au démarrage et à l'arrêt des activités ;
- la manipulation des fenêtres sur l'écran ;
- la gestion de la communication entre l'utilisateur et les activités.

2.2.1.1. Gestion des activités

La notion d'activité est naturellement liée au principe d'accès interactif introduit par les systèmes en temps partagé. Précisons tout d'abord ce que recouvre cette notion pour l'utilisateur :

> une *activité* est une *application interactive* avec laquelle l'utilisateur dialogue pour accomplir une tâche donnée comme l'édition d'un document ou la mise au point d'un programme.

Les activités et les applications [interactives] sont des entités distinctes. Une application est un service potentiel du système auquel tout utilisateur peut accéder lorsqu'il démarre une activité ; une activité est une application en cours d'exécution pour le compte d'un utilisateur particulier.

Nous avons distingué deux types de systèmes multi-fenêtres en introduction de ce chapitre : les systèmes mono-activité et les systèmes multi-activités. Nous n'avons pas jugé utile de décrire séparément ces deux types de système, si ce n'est implicitement en étudiant l'opération de changement d'activité courante qui ne concerne que les systèmes multi-activités. Les avantages de ces systèmes par rapport aux systèmes mono-activité sont en effet évidents et ne méritent aucun commentaire supplémentaire.

Démarrage des activités

Démarrer une activité consiste à désigner une application et éventuellement divers paramètres, par exemple l'application « éditeur » et le fichier à éditer. Dans les systèmes Cedar et Star dévelopés chez Xerox ou Lisa et Macintosh de chez Apple, l'utilisateur démarre une activité en désignant une des applications disponibles dans son environnement global ou un des objets qu'il a créés par leur intermédiaire.

Lorsque l'utilisateur désigne une application, l'activité correspondante peut démarrer « à vide » comme dans le cas des éditeurs ou donner accès à un objet prédéfini tel que le courrier électronique avec l'application « messagerie » dans le système Cedar. Lorsque l'utilisateur désigne un objet, il désigne implicitement l'application permettant de le manipuler car le système associe à chaque objet le nom de l'application qui a permis de le créer.

La commande de démarrage d'une activité n'a donc qu'un seul paramètre dans de tels systèmes et c'est aux applications d'offrir dans leur propre interface les commandes permettant à l'utilisateur de désigner d'autres paramètres éventuels (nom de la personne à qui envoyer un courrier dans le cas de la messagerie par exemple).

Le même principe ne peut être appliqué dans un environnement comme celui d'Unix, tout du moins de manière systématique comme dans les exemples cités ci-dessus où sa mise en œuvre a été prévue dès la conception du système. Il est en effet difficile d'inclure dans une interface de ce type certaines des possibilités (passage de paramètres, redirection des entrées-sorties standard, principe du pipe-line, etc) offertes par le *Shell* [Gittins 84]. Tous les systèmes multi-fenêtres adaptés à Unix doivent donc permettre à l'utilisateur de démarrer une activité *Shell* pour accéder à l'environnement standard du système.

Dans certains d'entre eux comme Bruwin ou PNX-WMS, le *Shell* est d'ailleurs la seule activité que l'utilisateur peut démarrer depuis l'interface du système. Cette méthode est la plus facile à implanter mais elle oblige l'utilisateur à démarrer un *Shell* pour pouvoir démarrer une nouvelle activité : arrêter pour cela l'une de celles déjà démarrées depuis un autre *Shell* irait en effet à l'encontre de l'objectif initial du système. Ce principe de démarrage par indirection systématique peut devenir fastidieux, surtout pour les applications utilisées fréquemment comme les éditeurs par exemple.

Le système SunWindows élimine cet inconvénient en proposant dans son interface plusieurs applications prédéfinies que l'utilisateur peut sélectionner lorsqu'il veut démarrer une activité. Le choix offert en standard comprend entre autres un éditeur de police de caractères, un outil de mise au point de programmes et le *Shell* pour accéder aux autres applications du système.

Création des fenêtres

La création d'une fenêtre est une opération particulière qui ne se réduit pas à l'allocation d'une surface d'affichage : une fenêtre est toujours créée pour répondre à un besoin de communication particulier entre l'utilisateur et une activité donnée et permettre à l'utilisateur de créer une fenêtre n'a aucun sens en soi au niveau de l'interface du système.

Les fenêtres sont donc toujours créées à la suite du démarrage d'une activité, soit par l'activité elle-même comme c'est en général le cas, soit par le système comme dans Cedar par exemple ou lorsque l'utilisateur démarre une activité *Shell* dans les systèmes adaptés à Unix.

Remarque : les rares systèmes comme Bruwin ou PNX-WMS qui permettent à l'utilisateur de créer explicitement une fenêtre ne font d'ailleurs pas vraiment exception au principe général puisque ils démarrent automatiquement une activité *Shell* dans la fenêtre créée. Le nom de la commande ne reflète donc pas le but réel de l'opération, et il serait plus cohérent de l'appeler « Démarrer-Shell » comme le fait Sapphire par exemple.

La création d'une fenêtre n'est pas toujours associée au démarrage d'une activité. L'interface utilisateur d'une application peut également inclure des commandes dont l'exécution implique la création d'une fenêtre par l'activité correspondante. L'éditeur de documents structurés Grif développé sur le système PNX-WMS offre ainsi la commande « voir » permettant à l'utilisateur de visualiser des parties

différentes du document en cours d'édition dans des fenêtres séparées créées au fur et à mesure par Grif [Quint 86].

Une activité peut éventuellement permettre à l'utilisateur de choisir la taille de la fenêtre qu'elle est en train de créer. L'activité peut alors laisser l'utilisateur décider librement des dimensions de la fenêtre ou lui imposer un minimum et/ou un maximum sur chacune d'elles. Le même principe s'applique pour le choix de la position de la fenêtre sur l'écran.

Changement d'activité courante

Cette opération permet à l'utilisateur de rediriger les dispositifs d'entrée vers l'activité avec laquelle il veut communiquer, l'*activité courante*.

Nous supposons en effet qu'il n'y a qu'une seule activité courante à un instant donné, c'est à dire que tous les dispositifs d'entrée (le clavier et la souris) sont attribués globalement à une seule l'activité. Ce principe est peu contraignant sur les postes de travail actuels où l'utilisateur ne dispose en fait que d'un clavier et d'une souris.

L'apparition de nouveaux supports de communication (microphones, etc.) peut remettre en question un tel choix : le problème sera alors de décider si tous les dispositifs d'entrée sont indépendants et de refléter en permanence à l'utilisateur l'attribution respective de chacun d'eux, c'est à dire l'activité courante pour chaque périphérique.

Que les dispositifs d'entrée soient attribués globalement ou pas, leur attribution doit résulter exclusivement d'une action de l'utilisateur qui est le seul à pouvoir choisir l'activité avec laquelle il veut dialoguer à un instant donné. Ce principe évite d'autre part qu'une information ne soit reçue par une activité différente de celle à laquelle elle était destinée : une telle éventualité pourrait en effet provoquer des erreurs d'interprétation aux conséquences désastreuses pour l'utilisateur.

Certains systèmes comme Bruwin, Apollo ou Sapphire sélectionnent automatiquement l'activité démarrée comme activité courante afin d'anticiper ce qu'ils considèrent comme une requête implicite de l'utilisateur, ce qui est a priori réaliste.

Ces deux opérations restent dissociées sur tous les autres systèmes (PNX-WMS, SunWindows, etc.) où l'utilisateur doit explicitement désigner la nouvelle activité démarrée pour la faire devenir activité courante

Cette dernière approche est préférable car plus souple, d'autant plus que le démarrage effectif d'une activité (chargement du programme, création des fenêtres, etc.) prend beaucoup plus de temps que le changement d'activité courante qui est une opération presque « immédiate » pour l'utilisateur.

Remarque : nous supposerons à partir de maintenant que les dispositifs d'entrée sont attribués globalement à l'activité courante.

Dans la majorité des systèmes, l'utilisateur sélectionne l'activité courante en sélectionnant avec la souris une des fenêtres de cette activité. Cette méthode est la plus naturelle pour l'utilisateur, surtout dans le cas d'une activité multi-fenêtres où elle lui permet de choisir directement la fenêtre à travers laquelle il veut commencer à dialoguer avec l'activité.

Certains systèmes comme Apollo ou Vitrail font automatiquement monter en sommet de pile la fenêtre sélectionnée par l'utilisateur. Les systèmes SunWindows et W ne modifient pas la position dans la pile de la fenêtre désignée par l'utilisateur mais choisissent au contraire de sélectionner comme activité courante celle associée à la fenêtre montée explicitement par l'utilisateur.

Cette approche est préférable car elle évite de bouleverser une configuration de l'écran convenant à l'utilisateur lorsque celui-ci désigne une fenêtre. L'utilisateur peut en effet vouloir uniquement interrompre son activité principale pour lancer une tâche de fond (une compilation par exemple) dans une fenêtre dédiée à cet usage, fenêtre dont il n'a justement conservé qu'une partie visible suffisante pour l'écho de la commande qu'il entre.

L'utilisateur doit avoir le moyen de distinguer en permanence l'activité courante. La solution la plus couramment employée consiste à modifier l'apparence de la dernière fenêtre désignée par l'utilisateur. Cette solution est bien adaptée aux activités mono-fenêtre où il ne peut y avoir d'ambiguïté sur la *fenêtre courante*, c'est à dire celle dans laquelle apparaîtra l'écho des informations transmises depuis tous les dispositifs d'entrée. La fenêtre courante est dans ce cas identique à la dernière fenêtre sélectionnée par l'utilisateur.

Le même principe ne peut plus être appliqué aux activités multi-fenêtres où la notion de fenêtre courante doit être alors considérée pour chaque dispositif d'entrée comme le prouve l'exemple suivant :

soit un éditeur utilisant deux fenêtres pour communiquer avec l'utilisateur, une fenêtre pour lui permettre de choisir les paramètres d'édition (police courante pour les caractères, etc.) que nous appelerons le menu d'édition et l'autre fenêtre pour afficher le document édité, la fenêtre d'édition. L'utilisateur peut alors sélectionner cette activité en désignant le menu d'édition avec la souris pour y effectuer un choix quelconque et entrer immédiatement après des caractères au clavier qui sont affichés dans la fenêtre d'édition.

Il y a alors deux fenêtres courantes, le menu sélectionné explicitement par l'utilisateur et la fenêtre d'édition sélectionnée implicitement. Distinguer la dernière fenêtre désignée par l'utilisateur n'est donc plus suffisant pour représenter l'activité courante. Le système doit alors appliquer ce principe à toutes les fenêtres de l'activité courante. Une solution moins systématique consiste à permettre aux activités de choisir elles-mêmes les fenêtres devant être mises en évidence lorsqu'elles deviennent l'activité courante.

Arrêt des activités

L'utilisateur a en général deux possibilités pour arrêter une activité : à l'aide de la commande du système ou par une commande propre à chaque activité. Lorsque l'utilisateur arrête l'activité courante, c'est à lui de sélectionner ensuite une autre activité courante. Le système doit entre temps placer tous les dispositifs d'entrée dans un mode « non attribué » pour respecter le postulat qui veut que l'utilisateur soit le seul à choisir l'activité courante.

L'arrêt d'une activité et la destruction des fenêtres suivent la démarche symétrique de celle adoptée pour le démarrage d'une activité et la création des fenêtres. L'utilisateur arrête une activité et celle-ci se termine en détruisant automatiquement toutes les fenêtres qu'elle a créées.

En général, l'arrêt d'une activité à la suite d'une erreur d'exécution du programme provoque la destruction automatique de la ou des fenêtre(s) créée(s) par cette activité. Le système PNX-WMS offre une option de création d'une fenêtre qui évite sa destruction lorsque l'activité est arrêtée. Une telle fenêtre ne peut plus alors qu'être déplacée sur l'écran ou détruite par l'utilisateur. Cette option permet de conserver une trace des informations affichées par une activité et peut être exploitée pour la mise au point de l'application correspondante par exemple.

2.2.1.2. Manipulation des fenêtres

Cet aspect de l'interface de commandes des gestionnaires de fenêtres a déjà été abordé au cours de la présentation des diverses méthodes de partage de l'écran. Nous nous sommes alors contentés d'évoquer l'exemple de certaines opérations pour comparer les possibilités que chacune de ces méthodes offrait à l'utilisateur pour disposer les fenêtres sur l'écran.

Nous allons maintenant décrire la nature des différentes manipulations que l'utilisateur peut effectuer sur une fenêtre. Nous nous intéressons ici uniquement à la logique des opérations : les aspects particuliers à chaque méthode de partage d'écran ne seront donc présentés que dans la mesure où cela est nécessaire. L'ensemble des fonctions étudiées ci-dessous ne prétend pas être complet. Il regroupe néanmoins les principales commandes offertes à l'utilisateur par la plupart des systèmes actuels.

Cacher et montrer une fenêtre

Cacher une fenêtre consiste à la faire disparaître entièrement de l'écran et *montrer* une fenêtre à la ramener sur l'écran. Le système doit fournir à l'utilisateur un moyen de désigner les fenêtres cachées pour pouvoir les montrer à nouveau.

Ces deux opérations sont nécessaires dans les systèmes adoptant une méthode de partage de l'écran sans recouvrement. Ce type de gestion impose en effet le plus souvent qu'une ou plusieurs fenêtres soient ainsi retirées de l'écran pour permettre la création d'une fenêtre ou l'agrandissement de sa taille. Les fenêtres sont alors cachées par effet de bord d'une autre opération à l'initiative du système qui choisit lui-même la ou les fenêtres concernées comme dans Cedar par exemple. Le système Andrew applique le même principe mais permet également à l'utilisateur de cacher explicitement une fenêtre. Celui-ci peut ainsi alléger l'écran en cachant les fenêtres qu'il n'utilise pas momentanément.

C'est d'ailleurs dans un but identique que cette possibilité est offerte par les systèmes autorisant le recouvrement des fenêtres. Il est en effet plus facile pour l'utilisateur de cacher une fenêtre que de chercher à la recouvrir entièrement par d'autres. D'autre part, les deux mécanismes ne sont pas équivalents même si la fenêtre devient invisible dans les deux cas : une fenêtre cachée disparaît effectivement de l'écran et reste invisible tant que l'utilisateur le désire alors qu'une fenêtre

recouverte peut apparaître de nouveau lorsque l'utilisateur manipule l'une des fenêtres qui la recouvrent.

Déplacer une fenêtre

Nous envisagerons uniquement le cas des systèmes autorisant le recouvrement des fenêtres qui sont les seuls à n'imposer aucune restriction à l'utilisateur pour le choix de la position des fenêtres sur l'écran. Le système doit fournir un point de référence pour cette position : la plupart d'entre eux choisissent la coin supérieur gauche de la fenêtre.

Certains systèmes comme PNX-WMS ou Sapphire autorisent le déplacement d'une fenêtre en partie hors de l'écran. Ce procédé permet de ne visualiser que l'information momentanément utile d'une fenêtre et d'alléger ainsi l'écran sans avoir à réduire pour autant la taille de la fenêtre ou à la recouvrir partiellement par une ou plusieurs autres (la possibilité de cacher la fenêtre ne conviendrait pas dans ce cas à l'utilisateur qui veut en conserver une partie sur l'écran). Le système Sapphire permet également de déplacer entièrement une fenêtre hors de l'écran, ce qui équivaut alors à cacher la fenêtre.

Changer la taille d'une fenêtre

Cette opération est particulière dans la mesure où elle concerne à la fois la gestion du contenu de la fenêtre et la gestion de l'écran. Il est souvent utile pour l'utilisateur de pouvoir choisir une taille prédéfinie par le système comme par exemple la taille de l'écran afin de disposer rapidement d'une grande surface de travail dans une fenêtre. Ce peut être aussi une taille adaptée à la nature de l'information visualisée comme dans le cas des fenêtres simulant un terminal classique dont la taille privilégiée est celle qui permet d'afficher un tableau de 80 * 24 caractères.

Le contenu d'une fenêtre ne représente souvent qu'un sous-ensemble de l'information globale accessible. De nombreuses applications représentent cette information globale sous la forme d'un espace à deux dimensions sur lequel l'utilisateur peut faire « évoluer » la fenêtre par l'intermédiaire des commandes de défilement pour accéder à différentes parties de l'information. La fenêtre a alors une position par rapport à l'espace sur lequel elle évolue qui est indépendante de sa position sur l'écran.

Il est alors intéressant pour l'utilisateur de pouvoir choisir la manière dont il fait évoluer la fenêtre par rapport à cet espace lorsqu'il change la taille de la fenêtre. En d'autres termes, il s'agit pour l'utilisateur de pouvoir augmenter et/ou diminuer la taille de la fenêtre dans toutes les directions et de choisir ainsi les parties de l'information qu'il veut voir apparaître et/ou disparaître. L'utilisateur fournit alors deux types d'informations lorsqu'il change la taille d'une fenêtre :

- les nouvelles dimensions de la fenêtre ;
- sa nouvelle position par rapport à l'espace sur lequel elle évolue.

Notons que le système peut ou non changer la position de la fenêtre sur l'écran lorsque l'utilisateur change sa taille. La figure 3.5 ci-dessous explicite ce principe.

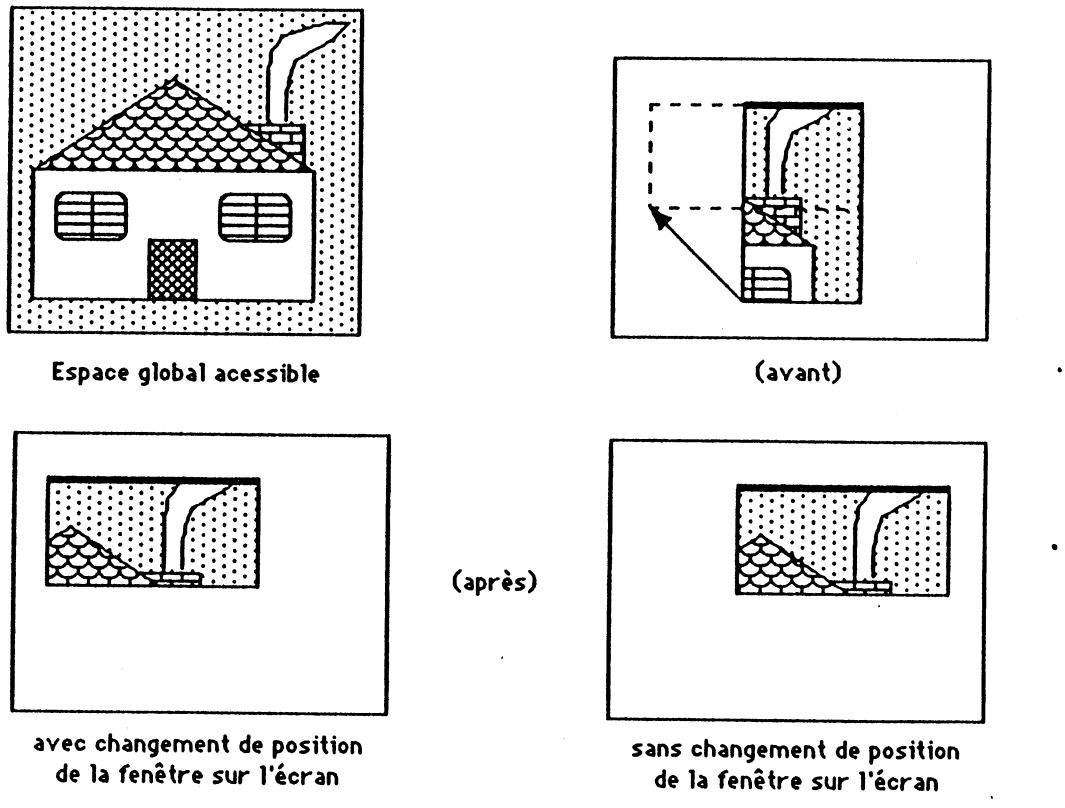


Figure 3.5 Changement de taille dans une direction particulière

Remarque : le changement de taille d'une fenêtre peut également avoir un effet de « zoom » sur l'information visualisée. Il s'agit dans ce cas de conserver le

même nombre d'objets affichés en adaptant leur représentation à la taille de la fenêtre.

Modifier l'ordre de superposition des fenêtres

Ce type de manipulation ne concerne évidemment que les systèmes autorisant le recouvrement des fenêtres sur l'écran. Rappelons la supposition émise lors de la présentation de cette méthode : la superposition des fenêtres est gérée par le système à l'aide d'une pile sur laquelle sont définies les opérations suivantes :

- *faire monter* une fenêtre : placer la fenêtre au dessus de toutes les autres sur l'écran ;
- *faire descendre* une fenêtre : placer la fenêtre en dessous de toutes les autres sur l'écran ;
- *changer le rang* d'une fenêtre : mettre la fenêtre à une position quelconque dans la pile.

La plupart des systèmes offrent uniquement les deux premières opérations à l'utilisateur et nous avons vu que le système du terminal Blit se contente de simuler la descente d'une fenêtre en faisant monter successivement toutes les autres. Un tel choix est basé sur le fait que les utilisateurs font beaucoup plus souvent monter que descendre une fenêtre [Gaylin 86]. Cette deuxième opération représente néanmoins un élément de confort supplémentaire pour l'utilisateur car elle lui permet de découvrir et d'accéder rapidement à toutes les fenêtres entièrement recouvertes par une autre.

L'opération de changement de rang d'une fenêtre est rarement implantée (si ce n'est dans Sapphire par exemple) et a priori n'est jamais mise à la disposition de l'utilisateur. Elle n'a en effet aucun sens pour ce dernier puisque la position d'une fenêtre dans la pile n'est pas significative de sa visibilité sur l'écran.

L'intérêt d'une telle opération est en fait indirect : elle peut ainsi servir à réaliser la commande *défaire* qui annule la dernière manipulation effectuée par l'utilisateur sur l'écran. Cette commande doit alors replacer éventuellement la fenêtre manipulée à son ancienne position dans la pile pour restaurer exactement la configuration précédente de l'écran. C'est entre autres le cas lorsque la commande à annuler a justement consisté à fait monter ou descendre une fenêtre qui n'était pas en dessous ou au dessus de toutes les autres.

2.2.1.3. Communication avec les activités

La communication entre l'utilisateur et une activité comprend deux aspects :

- l'envoi d'informations à l'activité ;
- la visualisation d'informations dans les fenêtres.

Ces deux aspects sont en général gérés par l'activité qui interprète les informations entrées par l'utilisateur et qui reflète le résultat de cette interprétation dans la ou les fenêtres qu'elle utilise. Les fonctions d'édition, de désignation et de défilement d'informations font donc partie de l'interface de commande des activités, le système se contentant uniquement d'assurer et de contrôler le transfert des informations échangées.

Cette répartition des rôles n'est pas toujours aussi stricte dans les systèmes. Par exemple, le noyau du système Unix assure par défaut l'écho des caractères entrés au clavier et permet à l'utilisateur d'effacer le dernier caractère entré ou même toute la « ligne courante ». Ces fonctions sont implantées dans le noyau dans un souci d'efficacité mais ce choix est en fait transparent à l'utilisateur. Notons que les fonctions d'édition du noyau ne portent que sur la ligne courante, c'est à dire sur les caractères entrés au clavier et pas encore transmis à l'application. Le noyau d'Unix ne permet pas à l'utilisateur d'accéder aux caractères affichés sur l'écran afin de les réutiliser en entrée, principe connu sous le terme de « couper-coller » (« cut and paste »).

Certains systèmes multi-fenêtres offrent des commandes de « couper-coller » permettant à l'utilisateur de copier ou de déplacer une partie du contenu d'une fenêtre dans cette fenêtre ou dans une autre. La démarche sous-jacente est la suivante : l'utilisateur désigne l'information à copier ou à déplacer (il « coupe ») dans la fenêtre source puis désigne la position à laquelle il veut la voir apparaître dans la fenêtre cible (il « colle »). Le système peut permettre à l'utilisateur de transférer l'information de manière *asynchrone* ou *synchrone* :

- transfert *asynchrone*. Le système offre deux commandes séparées pour « couper » et « coller » dans une fenêtre ; la commande *couper* range dans un tampon global l'information désignée dans la fenêtre source par l'utilisateur et la commande *coller* copie le contenu de ce tampon dans la fenêtre cible ;.

• transfert *synchrone*. Le système regroupe les deux opérations en une seule commande et l'information est alors directement transférée de la fenêtre source à la fenêtre cible ;

• Les systèmes comme Lisa, Macintosh ou SunWindows qui offrent ce service choisissent la solution du transfert asynchrone qui est plus avantageuse puisque le tampon représente alors un espace de travail temporaire pour l'utilisateur qui peut ainsi « coller » successivement son contenu dans plusieurs fenêtres après l'avoir éventuellement modifié. Le tampon permet également de conserver une information avant de la modifier dans la fenêtre où elle est affichée ou avant d'arrêter l'activité correspondante.

Transférer de l'information suppose un langage commun entre l'émetteur et le récepteur : ce problème ne se pose pas a priori lorsque l'utilisateur « coupe » et « colle » à l'intérieur d'une même activité ou entre des activités de même nature, c'est à dire exécutant le même programme d'application. Dans le cas d'activités de natures différentes, il importe que l'activité réceptrice soit capable d'accepter le format employé pour décrire l'information transférée et qu'elle puisse alors interpréter correctement cette information dans son propre contexte [Lemmons 83b].

Ce problème peut être résolu de manière globale dans un environnement intégré comme Lisa offrant un nombre limité d'applications étroitement liées. Notons qu'il ne l'est pas sur le Macintosh qui n'offre pas un éditeur général mais plusieurs éditeurs spécialisés (MacWrite, MacDraw etc.) ayant leur propre langage. MacWrite permet à l'utilisateur d'insérer dans un document textuel un dessin réalisé avec MacDraw ; le dessin est dans ce cas transféré sous forme d'une matrice de points et non comme un objet structuré et ne peut donc plus être modifié par la suite dans le document.

La solution adoptée pour le Macintosh est inenvisageable dans un environnement comme celui d'Unix où les applications standard manipulent uniquement des chaînes de caractères : le *Shell* ne peut pas recevoir la description d'une figure géométrique ou une matrice de points. C'est pourquoi le système SunWindows autorise uniquement le transfert de chaînes de caractères entre fenêtres.

2.2.2. Expression des commandes

Nous avons vu en II.1.2 que cet aspect de l'interface utilisateur a évolué considérablement avec l'apparition des postes de travail. Cette évolution est due essentiellement à l'intégration dans l'architecture des postes de travail d'un écran à points et d'un mécanisme indépendant de désignation sur l'écran tel que la souris. L'utilisation conjointe de ces deux éléments a permis d'élaborer de nouveaux schémas de communication dont le principe de base consiste à désigner sur l'écran et non plus à mémoriser et à épeler au clavier. La représentation d'une entité par une *icône* et la sélection dans un *menu* font maintenant partie des techniques de base pour la réalisation de l'interface des systèmes multi-fenêtres. Nous présentons donc séparément chacune de ces techniques avant de voir comment elles sont exploitées dans les systèmes multi-fenêres.

Les icônes

Une *icône* est un petit dessin conçu de façon à fournir une description significative de la fonction ou des caractéristiques (du type) de l'entité du système qu'elle représente. Résumer un concept dans une surface réduite de manière non ambiguë pour l'utilisateur n'est pas évident et un tel objectif ne peut être atteint qu'en faisant appel à certaines techniques de communication graphique élaborées dans d'autres disciplines comme par exemple la réalisation de « logos » pour les entreprises [Marcus 84].

Notons que le principe de représentation par icônes ne peut être appliqué systématiquement pour distiguer toutes les entités d'un système. Une icône peut être associée aux différents services offerts par le système (éditeur, messagerie, etc) mais le principe de nomination par chaînes de caractères est irremplaçable pour les objets de même type comme les fichiers par exemple. Par contre, ces deux moyens de désignation ne sont pas exclusifs : sur le Macintosh, le nom de chaque fichier est affiché sur l'écran en dessous d'une icône désignant l'application (MacWrite, MacPaint, etc.) avec laquelle il a été créé. Cette icône constitue la « signature » de l'application dans la terminologie Apple.

Les menus

Le principe du *menu* consiste à regrouper des entités de même nature (commandes, objets, etc) et à visualiser cet ensemble sur l'écran afin que l'utilisateur puisse sélectionner un des éléments avec la souris. Les menus peuvent être

statiques ou *dynamiques* selon qu'ils restent constamment visibles ou qu'ils apparaissent uniquement à la demande de l'utilisateur et ne restent visibles que pour la durée de la sélection.

Les menus statiques sont surtout utilisés pour offrir un accès rapide aux commandes les plus couramment employées ou pour visualiser en permanence les choix effectués par l'utilisateur. Les menus dynamiques offrent l'avantage de ne pas consommer de place sur l'écran tout en conservant le principe de la désignation directe qui se fait alors en deux temps.

Que les menus soient statiques ou dynamiques, les éléments qu'ils contiennent peuvent être représentés sous forme icônique ou textuelle. Le système Star regroupe ainsi dans des menus dynamiques appelés « feuilles de propriétés » (« property sheet ») les caractéristiques de tous les objets manipulables dans le système, principe qui sera repris par la suite pour la conception des systèmes Lisa et Macintosh. La figure 3.6 ci-dessous est une reproduction de l'exemple utilisé dans [Lipkie 82] pour décrire le principe des feuilles de propriétés ; il s'agit ici de celle associée aux objets de type « segment de droite » par l'éditeur du Star.

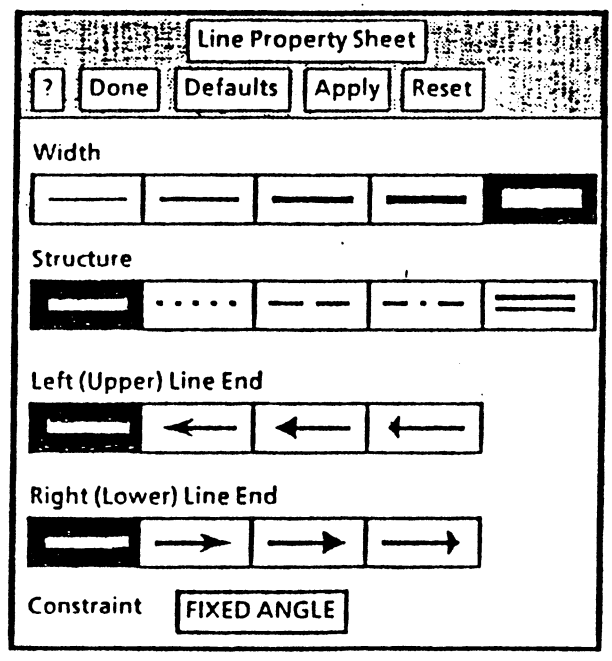


Figure 3.6 Feuille de propriétés des segments de droite

Le contenu de cette feuille de propriétés signifie que toute ligne tracée par l'utilisateur sera continue et d'épaisseur maximum, sans flèche aux extrémités et de direction fixe durant le tracé.

2.2.2.1. Gestion des activités

Les systèmes Cedar, Star, Lisa et Macintosh représentent par des icônes l'environnement de travail de l'utilisateur, c'est à dire les différentes applications disponibles et les fichiers créés à l'aide de ces applications. Les fichiers possèdent également un nom permettant de les distinguer. Ces icônes sont affichées en bas de l'écran dans Cedar, sur toute la surface de l'écran dans le Star où ils peuvent être éventuellement recouverts par une fenêtre et dans des fenêtres « système » sur le Macintosh, à raison d'une fenêtre par répertoire. Dans tous ces systèmes, l'utilisateur démarre une activité en sélectionnant une icône avec la souris.

Nous avons vu que ce principe de représentation uniforme ne pouvait être envisagé dans les systèmes multi-fenêtres développés sur le système Unix (cf III.2.2.1.1). Dans la majorité d'entre eux (PNX-WMS, Apôtre, W, SunWindows, etc.), l'utilisateur démarre une activité en sélectionnant l'application dans un menu dynamique qui apparaît lorsqu'il sélectionne avec la souris un point quelconque de l'écran situé en dehors de toute fenêtre.

Changement d'activité courante

Dans les systèmes SunWindows ou Apollo, l'utilisateur sélectionne l'activité courante en *désignant* une de ses fenêtres, c'est à dire en déplaçant la souris dans une fenêtre (cf II.2.2). Cette méthode a l'avantage d'être rapide pour l'utilisateur mais induit de nombreuses erreurs lorsque la souris est bougée par inadvertance. C'est pourquoi la majorité des systèmes multi-fenêtres exigent que l'utilisateur *sélectionne* une fenêtre en appuyant sur un des boutons de la souris pour changer d'activité courante.

Une touche du clavier peut également être dédiée à cette fonction : le système doit alors fournir un moyen implicite ou explicite pour désigner la fenêtre. Le système Sapphire propose une solution basée sur l'ordonnancement dans le temps des fenêtres désignées par l'utilisateur et un mécanisme de bascule associé à une touche particulière du clavier. L'utilisateur peut de cette manière alterner rapidement entre les deux dernières fenêtres qu'il a sélectionnées. Il doit par contre utiliser

la souris pour désigner une autre fenêtre, ce qui nuit à la cohérence d'un tel mécanisme.

2.2.2.2. Manipulation des fenêtres

La plupart des systèmes choisissent d'implanter les commandes de manipulation dans le *bord* de chaque fenêtre, ce qui permet à l'utilisateur de désigner simultanément la fenêtre qu'il veut manipuler lorsqu'il sélectionne une commande quelconque.

Un emplacement différent peut être attribué dans le bord à chaque commande de manipulation selon le principe des menus statiques comme sur le Macintosh par exemple. Une variante consiste à regrouper les commandes dans un menu dynamique qui apparaît lorsque l'utilisateur désigne avec la souris n'importe qu'elle partie du bord de la fenêtre : PNX-WMS et SunWindows adoptent ce principe. Les deux méthodes peuvent être utilisées conjointement comme dans Sapphire par exemple.

Le choix de l'emplacement des commandes dans le bord n'est pas anodin. Tout d'abord, l'utilisateur doit toujours pouvoir accéder aux commandes (ou du moins à certaines d'entre elles comme *faire monter* la fenêtre) lorsque la fenêtre est en partie recouverte. Ce problème est résolu en attribuant par exemple chaque coin du bord à une opération particulière ou à menu dynamique les regroupant toutes.

D'autre part, la taille d'une fenêtre doit pouvoir être changée dans toutes les directions (cf III.2.2.1.2). Un moyen pratique consiste à implanter cette commande dans les quatre coins (ou les quatre côtés) du bord de chaque fenêtre. Le coin (ou le côté) sélectionné par l'utilisateur fournit alors implicitement cette direction. Le même principe peut être appliqué pour la commande de déplacement.

Lorsque la souris comporte plusieurs boutons, le même emplacement peut être attribué à plusieurs commandes: c'est ainsi que Sapphire permet à l'utilisateur de déplacer ou de changer la taille d'une fenêtre depuis chaque côté de son bord.

Les commandes sont souvent représentées par des icônes qui peuvent être visibles en permanence ou apparaître uniquement lorsque l'utilisateur désigne l'emplacement correspondant dans le bord. Dans Sapphire, l'icône n'apparaît que lorsque l'utilisateur sélectionne un point de cet emplacement.

Le système doit également prévoir un moyen d'accès aux fenêtres cachées. Les systèmes Cedar et SunWindows choisissent de représenter ces fenêtres sous forme d'icônes. Dans Cedar, cette icône représente en fait une application potentielle (l'horloge, la messagerie, etc) ou une activité démarrée par l'utilisateur mais dont la fenêtre a été momentanément retirée de l'écran par le système.

Apôtre choisit la solution du menu dynamique pour donner accès aux fenêtres cachées par l'utilisateur : le menu contient alors le titre affiché dans la partie supérieure du bord de chaque fenêtre.

2.2.2.3. Communication avec les activités

Cet aspect est en général géré par les activités qui ont donc entière liberté pour réaliser leur propre interface de commande. Le système peut néanmoins offrir aux activités une représentation standard des commandes les plus courantes. C'est le cas par exemple des commandes de défilement horizontal et vertical qui permettent à l'utilisateur de faire apparaître des parties différentes de l'information globalement accessible dans une fenêtre.

La méthode employée sur le Macintosh consiste à symboliser la position de la fenêtre par rapport à cet espace global à l'aide de deux « curseurs » évoluant dans deux barres disposées dans le bord de la fenêtre parallèlement à chacun des deux axes du repère. L'utilisateur déplace chaque curseur dans la barre pour faire évoluer la fenêtre selon l'axe correspondant de l'espace global.

Les opérations associées au mécanisme de transfert d'informations entre fenêtres sont en général offertes dans un menu. le Macintosh offre ainsi trois commandes : *couper* (« cut »), *copier* (« copy ») et *coller* (« paste »). Les deux premières rangent dans le tampon global l'information sélectionnée dans la fenêtre courante, *couper* détruisant cette information et *copier* ne faisant que la dupliquer. *Coller* duplique toujours le contenu du tampon dans la fenêtre courante à la position désignée par l'utilisateur avec la souris.

2.3. Conclusion

L'utilisateur doit pouvoir *personnaliser* l'interface du système afin de l'adapter à ses goûts. Il doit ainsi pouvoir choisir la manière dont il exprime les commandes de manipulation des fenêtres, changer le choix offert par le menu à partir duquel il sélectionne une application pour démarrer une activité.

Dans les systèmes qui gèrent le recouvrement des fenêtres sur l'écran, l'utilisateur doit pouvoir imposer que les fenêtres soient disposées automatiquement par le système de façon à se recouvrir le moins possible ou selon une heuristique particulière.

Il doit également pouvoir choisir la manière dont le système combine certaines opérations : changement de taille avec changement de position ou non, faire monter une fenêtre en la désignant, etc. Ce choix peut éventuellement être sélectif et ne concerner que certaines fenêtres. Le système peut aussi offrir différentes commandes permettant de choisir entre la combinaison ou non des opérations.

L'utilisateur doit pouvoir faire évoluer l'interface du système en fonction de son degré de maîtrise du fonctionnement des commandes. Il peut ainsi préférer exprimer certaines commandes depuis le clavier que de passer systématiquement par un menu qui a l'inconvénient de freiner la sélection de la commande : faire apparaître le menu, déplacer la souris pour choisir la commande. Ce principe des « accélérateurs » est offert par défaut pour de nombreuses commandes sur le Macintosh par exemple.

Le système doit enfin permettre également l'expérimentation de nouvelles idées pour réaliser l'interface de commandes.

3. L'interface de programmation

3.1. Introduction

L'interface de programmation d'un système multi-fenêtres définit les services mis à la disposition du programmeur pour gérer la communication avec l'utilisateur dans une application interactive. Par définition, la *fenêtre* représente le support de base pour la mise en œuvre de l'interface utilisateur des applications. Celles-ci suivent le schéma général qui consiste à afficher dans une (ou plusieurs) fenêtre(s) le résultat de l'interprétation des commandes émises par l'utilisateur à l'aide des dispositifs d'entrée gérés par le système.

Dans la majorité des systèmes, l'interface de programmation est structurée en [au minimum] deux niveaux :

- l'*interface de base* qui donne accès aux fonctions gérées par le système ;
- un ensemble d'*outils standard* construits au dessus de l'interface de base et mis à la disposition du programmeur dans des bibliothèques appelées « boîtes à outils ».

Définir l'interface de base consiste à déterminer l'ensemble des fonctions implantées dans le système et le niveau d'abstraction que ces fonctions offrent aux applications et au concepteur des bibliothèques standard. C'est pourquoi nous nous intéresserons uniquement à l'interface de base en étudiant les différents critères (efficacité à l'exécution, etc.) qui peuvent intervenir dans le choix d'implanter ou non dans le système une fonction donnée. Nous avons choisi de présenter l'interface de base des systèmes multi-fenêtres en distinguant les trois aspects suivants :

1) la *gestion des fenêtres* en tant qu'objets manipulables globalement. Nous étudierons séparément à ce niveau la gestion de l'espace d'affichage et son partage entre les fenêtres, la création des fenêtres et enfin la manipulation des fenêtres sur l'écran ;

2) l'*interface graphique* qui regroupe les opérations d'affichage d'informations dans une fenêtre et par extension toutes celles permettant de manipuler le contenu d'une fenêtre ;

3) l'*interface d'accès* aux informations émises par l'utilisateur depuis les dispositifs d'entrée et d'une façon plus générale à l'ensemble des événements transmis aux activités par le système.

Dans un contexte multi-activités, le système doit également partager les différentes ressources qu'il gère entre les activités démarrées par l'utilisateur. Nous nous intéressons ici aux ressources directement liées à la gestion des fenêtres proprement dite. Le problème du partage du processeur et de la mémoire centrale entre les activités est uniquement abordé à travers les contraintes particulières que la gestion de ces ressources peut imposer au gestionnaire de fenêtres. Pour chaque type de ressource gérée par le système (mémoire-écran, dispositif d'entrée, police de caractères, etc), nous présentons les différentes stratégies qui peuvent être mises en œuvre pour son partage et les conséquences de chacune d'elles pour le système comme pour les applications.

3.2. Gestion des fenêtres

3.2.1. Gestion de l'espace d'affichage des fenêtres

Cet aspect a été abordé en III.2.1 où nous avons étudié les différentes méthodes de répartition de la surface de l'écran en distinguant les systèmes qui autorisent le recouvrement des fenêtres et ceux qui ne le permettent pas. Quelle que soit la méthode choisie pour gérer la surface de l'écran, le système peut adopter vis à vis des applications l'une des deux techniques suivantes :

1) *gérer deux niveaux d'espace d'affichage* en distinguant pour chaque fenêtre l'espace réel occupé sur l'écran et l'espace virtuel alloué en mémoire hors-écran et dont la taille peut être supérieure à celle de l'espace réel. C'est l'approche suivie dans W par exemple ;

2) *gérer l'espace d'affichage à un seul niveau*, c'est à dire ne gérer que l'espace réel des fenêtres.

3.2.1.1. Espace d'affichage à deux niveaux

La distinction entre espace d'affichage virtuel et espace d'affichage réel a déjà été évoquée en III.2.2.1.2 à travers la notion d'espace global représentant l'ensemble des informations manipulées par une application, espace sur lequel l'utilisateur peut faire « évoluer » la fenêtre en changeant sa taille ou par l'intermédiaire des commandes de défilement. Il s'agit donc ici d'intégrer ce principe dans le système et de permettre aux applications de représenter dans l'espace virtuel les informations accessibles à l'utilisateur.

Le système peut ainsi gérer de façon transparente aux activités le changement de taille d'une fenêtre (c'est à dire de la taille de l'espace réel) et en particulier son agrandissement. Ce principe permet également d'assurer plus efficacement le défilement des informations dans la fenêtre puisque la mise à jour du contenu de l'espace réel est réalisée dans le système par une opération de copie dans la mémoire-écran d'une partie de l'espace virtuel. Les commandes de défilement pourraient être entièrement prises en compte par le système mais il est préférable que ce soient les applications qui « pilotent » le défilement pour les raisons suivantes :

• le « pas » de défilement peut être fonction de la nature des informations manipulées et doit alors être calculé par l'application. C'est le cas par exemple lorsque l'utilisateur fait défiler ligne par ligne un document textuel et en particulier si plusieurs polices de caractères de hauteurs différentes sont utilisées pour présenter le document ;

• l'application doit de toute façon intervenir si la taille allouée à l'espace virtuel de la fenêtre ne lui permet pas de réprésenter l'ensemble des informations auxquelles l'utilisateur peut accèder, c'est à dire si la taille de l'espace virtuel est inférieure à celle de l'espace global. L'application doit alors mettre à jour le contenu de l'espace virtuel lorsque l'utilisateur demande à accèder à une partie de l'espace global qui n'y est pas représentée.

Le second intérêt de cette technique provient du fait que l'espace virtuel joue le rôle de mémoire de sauvegarde pour le contenu des fenêtres cachées ou en partie recouvertes sur l'écran. Quelle que soit la méthode adoptée pour gérer la surface de l'écran (recouvrement ou non des fenêtres), le système peut assurer de façon transparente aux activités le réaffichage sur l'écran [d'une partie] d'une fenêtre cachée découverte à la suite de la manipulation de cette fenêtre ou d'une autre fenêtre.

Cette technique a par contre l'inconvénient d'être coûteuse en ressource-mémoire et complexe à mettre en œuvre. Le système doit en effet gérer la correspondance entre les deux espaces associés à chaque fenêtre et inclure des fonctions supplémentaires, entre autres les fonctions de déplacement de l'espace réel par rapport à l'espace virtuel permettant aux applications de piloter le défilement des informations.

3.2.1.2. Espace d'affichage à un seul niveau

Cette technique est plus simple à mettre en œuvre que la précédente et moins gourmande en mémoire puisque la gestion de l'espace virtuel est à la charge des applications. Gérer l'espace virtuel au niveau des applications offre également les avantages suivants :

• cela évite d'en faire supporter le coût aux applications pour lesquelles ce principe n'est d'aucune utilité ;

• les applications peuvent adapter le mode de représentation et de structuration du contenu de l'espace virtuel à la nature des informations qu'elles manipulent ;

• le même espace virtuel peut être associé à plusieurs fenêtres par une application. C'est ce que fait l'éditeur de documents Grif pour visualiser dans plusieurs fenêtres différentes parties d'un même document, par exemple la table des matières dans une fenêtre et le contenu d'un paragraphe particulier dans une autre fenêtre.

C'est pour toutes ces raisons que la majorité des systèmes actuels adoptent cette technique et c'est uniquement ces systèmes que nous considérerons dans la suite de cette thèse. Nous utiliserons donc à partir de maintenant le terme de *fenêtre* pour désigner l'espace d'affichage alloué aux applications par le système. Dans ce type de systèmes, la gestion de l'écran ne peut plus être complètement transparente aux applications.

Dans les systèmes qui interdisent le recouvrement des fenêtres, les fenêtres ne sont pas indépendantes les unes des autres : toute manipulation d'une fenêtre (y compris sa création et sa destruction) peut éventuellement nécessiter la modification de la taille d'autres fenêtres ou même leur retrait de l'écran (cf III.2.1.2). La taille des fenêtres est imposée par le système aux applications qui doivent être programmées de façon à s'adapter à la taille allouée par le système, que ce soit à la création d'une fenêtre ou à la suite d'une manipulation de l'utilisateur.

Dans les systèmes qui gèrent le recouvrement des fenêtres sur l'écran, les fenêtres peuvent être manipulées indépendamment les unes des autres. Les applications peuvent donc choisir la taille des fenêtres qu'elles créent et éventuellement interdire à l'utilisateur de la modifier.

Les systèmes adoptant la méthode sans recouvrement ne gèrent pas le contenu des fenêtres cachées qui en fait sont des fenêtres de taille nulle et envoient une requête de réaffichage aux applications lorsqu'une fenêtre cachée devient à nouveau visible sur l'écran. Les applications doivent donc maintenir en permanence le contenu de leurs fenêtres.

Certains systèmes (Vitrail, Macintosh, SunWindows, Apollo, X) qui autorisent le recouvrement des fenêtres sur l'écran adoptent le même principe pour les fenêtres [en partie] cachées. Les systèmes PNX-WMS, Blit, Oriel, Apôtre et WHIM gèrent par contre le recouvrement des fenêtres de façon transparente aux applica-

tions en conservant en mémoire hors-écran le contenu des parties cachées des fenêtres.

Nous pouvons donc distinguer deux approches selon que le réaffichage du contenu des fenêtres sur l'écran est assuré par le système ou au contraire laissé à la charge des applications. Analysons leurs avantages et inconvénients respectifs.

Lorsque le réaffichage est à la charge des applications, le système n'a pas à gérer d'espace-mémoire supplémentaire ni à le partager entre les activités démarrées par l'utilisateur, ce qui garantit l'indépendance des activités qui ne sont plus alors limitées que par la taille de leur propre espace d'adressage. Cette méthode évite également toute redondance au niveau de la conservation des informations contenues dans les fenêtres. De nombreuses applications sont en effet obligées de par leur fonction de gérer une structure de données représentant les informations affichées ou susceptibles de l'être. C'est le cas des éditeurs par exemple.

Pour le système comme pour les applications, il n'y a pas de différence fondamentale entre l'opération de changement de taille d'une fenêtre et les autres opérations (montrer ou faire monter une fenêtre) qui nécessitent toutes l'intervention des applications pour la mise à jour de l'écran.

Cette méthode complique par contre la programmation des applications qui doivent systématiquement conserver une copie du contenu de leurs fenêtres. La gestion de cette copie peut être difficile à mettre en œuvre, en particulier lorsqu'une requête de réaffichage est reçue par une activité durant une opération d'affichage.

Les systèmes qui gèrent le réaffichage simplifient la conception des applications et éliminent les erreurs qu'une telle gestion peut entraîner. Cette simplification se traduit également en gain de mémoire, le code chargé du réaffichage étant inclus dans le système et donc partagé par toutes les applications.

Dans un contexte multi-activités, les systèmes qui n'assurent pas le réaffichage sont obligés de « réveiller » les activités dont les fenêtres ont été [en partie] découvertes à la suite d'une manipulation sur l'écran. Ces changements de contexte artificiellement provoqués diminuent l'efficacité globale du système, surtout lorsque le contexte des activités réveillées a été rangé temporairement en mémoire secondaire et doit être préalablement ramené en mémoire centrale. Cette charge supplémentaire supportée par le système, donc finalement par l'utilisateur, est d'autant plus coûteuse que la surface réelle à mettre à jour sur l'écran est petite.

C'est le cas notamment dans les systèmes autorisant le recouvrement où cette surface peut à la limite se réduire à un seul point de l'écran.

Dans ce cas, le problème de l'efficacité se pose également au niveau du réaffichage proprement dit. Il est souvent impossible pour les applications de réafficher uniquement le contenu des parties cachées car la décomposition d'une fenêtre en parties cachées et en parties visibles n'a aucune raison (si ce n'est le hasard) de correspondre à la structure des informations qui y sont affichées. Les applications sont alors obligées de réafficher le contenu entier de la fenêtre et qui plus est de le faire par appel de primitives graphiques interprétées dans le système.

Lorsque la gestion des parties cachées est à la charge du système, celui-ci peut optimiser la mise à jour de l'écran en ne réaffichant que le strict nécessaire et en utilisant une opération de recopie (un « RasterOp ») de la mémoire de sauvegarde dans la mémoire-écran. Cette optimisation permet d'éliminer tout effet visuel superflu et diminue le temps nécessaire à la réalisation des manipulations sur l'écran, donc améliore le confort de l'utilisateur (cf III.2.1.1).

3.2.2. Création d'une fenêtre

Rappelons le principe général énoncé précédemment : l'utilisateur démarre une activité qui crée elle-même les fenêtres dont elle a besoin pour communiquer avec lui (cf III.2.2.1.1). Du point de vue des applications, créer une fenêtre consiste essentiellement à obtenir une *surface d'affichage* et un *nom* permettant de désigner la fenêtre dans les autres primitives de l'interface.

Gestion des noms

Les systèmes SunWindows, PNX-WMS et Oriel développés sur Unix associent deux noms à chaque fenêtre : un *nom global* représentant le nom d'un fichier spécial et un *nom local* au processus créant la fenêtre. Le nom local est un numéro de flot et sert à désigner la fenêtre dans les autres primitives de l'interface du système. Plusieurs processus non apparentés peuvent ainsi accèder à une même fenêtre par l'intermédiaire de son nom de fichier ; le nombre de fenêtres qu'une application peut utiliser simultanément est par contre limité par le nombre de flots qui peuvent être alloués à un processus dans Unix (20 en général).

Les systèmes W et X gèrent l'espace de noms des fenêtres à un seul niveau : c'est le nom global de la fenêtre (un entier) qui est transmis aux activités lorsqu'elles créent une fenêtre. Cette technique offre l'avantage de ne pas restreindre les applications sur le nombre de fenêtres.

Le système Oriel distingue pour chaque fenêtre créée le processus créateur et le processus propriétaire, tous deux étant les seuls à pouvoir manipuler la fenêtre sur l'écran et à la détruire. Ces deux processus sont initialement les mêmes mais le processus propriétaire peut transmettre la « propriété » d'une fenêtre à un autre processus.

Allocation de la surface d'affichage

Nous avons vu que les systèmes interdisant le recouvrement des fenêtres sur l'écran ne permettent pas aux applications de choisir la taille des fenêtres qu'elles créent (cf III.1.1.2). Dans le système Andrew, la taille indiquée par les applications ne sert que d'indication pour le système qui seul décide de la taille réelle de la fenêtre. Les applications doivent donc systématiquement contrôler la taille effectivement allouée par le système à la fenêtre qu'elles viennent de créer. L'avantage avec le principe de partage de l'écran sans recouvrement est que la création d'une fenêtre n'échoue jamais du point de vue des applications puisque le système fixe lui-même la taille de la fenêtre créée, en diminuant éventuellement la taille d'autres fenêtres.

Remarque : nous négligeons le problème de l'allocation du nom qui est une ressource « peu coûteuse » dont la probabilité de saturation peut être considérée comme nulle.

Dans les systèmes qui gèrent le recouvrement des fenêtres, les applications peuvent choisir la taille des fenêtres qu'elles créent. Les systèmes qui ne sauvegardent pas le contenu des fenêtres [en partie] cachées garantissent également que la création d'une fenêtre ne peut échouer (en considérant toujours la probabilité de saturer l'espace des noms comme négligeable).

Ce problème se pose par contre dans les systèmes qui gèrent le contenu des parties cachées des fenêtres. Dans ce cas, le système doit en effet partager entre les fenêtres un espace mémoire réservé à la sauvegarde des parties cachées. Lorsque qu'une fenêtre ne peut être créée pour cause de manque de mémoire, le système peut adopter l'une des deux politiques suivantes :

1) bloquer l'activité dans une file d'attente jusqu'à ce que la fenêtre puisse être créée, et garantir ainsi aux activités la réussite de l'opération ;

2) refuser la création de la fenêtre et retourner un code d'erreur à l'activité.

La première méthode facilite la programmation des applications mais est par contre plus complexe à mettre en œuvre que la seconde car le système doit gérer la file d'attente et choisir une politique d'allocation (selon la technique du « premier arrivé - premier servi » ou selon la taille des fenêtres par exemple). Pour ne pas troubler l'utilisateur qui s'attend (normalement) à voir apparaître une fenêtre sur l'écran, le système doit l'avertir qu'une activité est bloquée sur une opération de création de fenêtre. L'utilisateur peut alors éventuellement intervenir en détruisant une fenêtre ou en arrêtant une activité pour libérer de l'espace mémoire.

La seconde solution simplifie la tâche du système en repoussant le problème au niveau des programmes d'application. Le système doit néanmoins avertir l'utilisateur lorsqu'il refuse la création d'une fenêtre, l'activité pouvant ne pas en avoir le moyen si cette fenêtre est la première qu'elle crée.

Nous avons vu que l'utilisateur peut éventuellement être invité à spécifier interactivement la taille et la position de la fenêtre créée par une activité (cf III.2.2.1.1). Dans Oriel, cette possibilité est une option de la primitive de création d'une fenêtre : l'application fournit en paramètre une taille maximum et c'est le système qui se charge du dialogue avec l'utilisateur pour obtenir la position et la taille effectives de la fenêtre. Ce principe offre les avantages suivants :

- il soulage le programmeur d'applications de la conception et de la mise au point du dialogue ;
- il garantit l'homogénéité du dialogue entre toutes les applications.

Vis à vis de l'utilisateur, nous avons vu que la surface d'une fenêtre est décomposée en deux parties : l'*intérieur* de la fenêtre qui contient les informations affichées par l'application et le *bord* qui délimite la surface globale de la fenêtre et où sont généralement implantées les commandes de manipulation des fenêtres (cf III.2.2.2.2). La démarche sous-jacente à cette décomposition consiste à réserver l'intérieur de la fenêtre aux fonctions spécifiques à chaque application et à regrouper dans le bord les commandes associées aux fonctions générales du système.

La majorité des systèmes (Oriel, Vitrail, X, Apôtre, W, etc.) reflètent cette décomposition en gérant le bord de façon transparente aux applications qui ne « voient » que l'intérieur de la fenêtre au niveau de l'interface de programmation. Ce sont les dimensions de l'intérieur de la fenêtre que les applications fournissent lorsqu'elles créent une fenêtre, le système ajoutant automatiquement l'espace occupé par le bord pour déterminer la surface réelle de la fenêtre. Ce sont également les dimensions de l'intérieur de la fenêtre qui sont transmises aux applications par le système lorsque l'utilisateur choisi la taille de la fenêtre à sa création (comme dans Oriel par exemple) et lorsqu'il la modifie par la suite.

Remarque : le système a dans ce cas le choix entre interpréter les dimensions fournies par l'utilisateur comme celles de l'intérieur de la fenêtre ou comme celles de la surface globale, c'est à dire bord y compris. La majorité des systèmes adoptent la seconde interprétation qui permet à l'utilisateur de s'exprimer en valeurs réelles par rapport à l'écran, ce qui est à priori plus naturel.

Les coordonnées transmises dans les primitives d'entrée-sortie sont d'autre part exprimées relativement à l'origine de l'intérieur des fenêtres : en entrée pour la position de la souris et en sortie pour la position à laquelle sont affichées les informations.

L'existence et le rôle même du bord ne sont pas pour autant complètement transparents aux applications dans les systèmes cités ci-dessus. PNX-WMS offre une option pour créer des fenêtres avec ou sans bord. Dans Vitrail, les applications peuvent demander au système d'implanter les commandes de défilement dans le bord de la fenêtre. Le système prend alors en charge l'aspect interface utilisateur de ces commandes et transmet aux applications des requêtes de défilement pour qu'elles mettent à jour le contenu de la partie intérieure.

Hiérarchie de fenêtres

Les systèmes W, X, Oriel, SunWindows et Vitrail autorisent les applications à constituer des hiérarchies de fenêtres. Ces systèmes gèrent l'ensemble des fenêtres selon une structure arborescente dont la racine est la fenêtre associée à l'écran. Une fenêtre est toujours créée dans une autre fenêtre - son ancêtre dans l'arbre - et hérite de certains de ses attributs : une fenêtre est toujours incluse dans son ancêtre, est déplacée avec elle et ne peut être visible sur l'écran que si son ancêtre l'est.

Le système doit répercuter la diminution de la taille d'une fenêtre sur la taille de ses filles, soit par troncature, soit par réajustement proportionnel. Ce mécanisme peut être également appliqué en cas d'agrandissement de la taille d'une fenêtre ou laissé à la charge des applications qui peuvent ou non vouloir agrandir la taille de ses filles.

Créer des « sous-fenêtres » dans une fenêtre permet par exemple de décomposer l'espace d'affichage d'une fenêtre en plusieurs régions et d'associer à chacune d'elle une fonction prédéfinie selon le principe des menus statiques. Dans Oriel et W, cette technique est exploitée par le système lui-même pour constituer le bord des fenêtres.

3.2.3. Manipulation des fenêtres

Les opérations de manipulation - cacher, montrer, déplacer une fenêtre, etc. - ont été présentées en tant que commandes de l'interface utilisateur (cf III.2.2.1.2). Nous avons vu en comparant les méthodes de partage de l'écran que celles qui interdisent le recouvrement sont très contraignantes par rapport aux systèmes avec recouvrement qui au contraire laissent toute liberté à l'utilisateur pour manipuler les fenêtres. Cet aspect est également reflèté au niveau de l'interface de programmation.

Les systèmes sans recouvrement ne permettent pas aux applications de manipuler librement les fenêtres sur l'écran et leur interface de programmation n'inclut aucune primitive pour cela. Ce principe a comme avantage d'empêcher les applications de modifier de manière intempestive la configuration de l'écran et de laisser la gestion de ce dernier sous le contrôle exclusif de l'utilisateur.

La majorité des systèmes gérant le recouvrement des fenêtres laissent au contraire toute liberté aux applications pour manipuler les fenêtres qu'elles ont créées. Un tel principe ne permet pas au système de contrôler les manipulations effectuées par les applications, ce qui peut être gênant pour l'utilisateur en cas de mauvais fonctionnement d'une application. Il a par contre l'avantage de laisser toute liberté pour la conception de l'interface utilisateur des applications. Il facilite en particulier la conception des outils de communication basés sur le principe consistant à faire apparaître *temporairement* sur l'écran des objets « dynamiques » tels que les menus, les formulaires, etc.

Dans les systèmes autorisant le recouvrement, les fenêtres servent de support à l'implantation des objets dynamiques puisque les applications peuvent choisir la taille des fenêtres qu'elles créent et les manipuler ensuite librement sur l'écran. Les objets dynamiques sont gérés par les applications ou dans une bibliothèque standard selon le principe général suivant :

- créer un objet dynamique consiste à créer une fenêtre d'une taille adaptée à la taille de cet objet ;

- faire apparaître (respectivement disparaître) un objet dynamique consiste à montrer (respectivement cacher) la fenêtre correspondante.

L'application ayant le choix de la position d'une fenêtre sur l'écran, elle peut par exemple faire apparaître un menu dynamique à la position courante de la souris de façon à minimiser la distance à parcourir pour sélectionner un élément du menu.

Cette technique est par définition inapplicable dans les systèmes interdisant le recouvrement des fenêtres sur l'écran puisque les applications ne peuvent ni choisir la taille des fenêtres qu'elles créent ni les manipuler librement sur l'écran. La seule possibilité consiste à implanter ce type de service dans le système et à inclure dans son interface de base les primitives permettant aux applications d'y accèder : le système peut en effet sauvegarder temporairement la partie de l'écran recouverte par un objet dynamique et la restaurer lorsqu'il fait disparaître cet objet. C'est la méthode adoptée dans le système Andrew pour gérer les menus qui sont les seuls objets dynamiques mis à la disposition des applications. Le système Cedar ne gère aucun type d'objets dynamiques et c'est la raison pour laquelle ces outils ne sont pas utilisés dans l'environnement du système [Teitelman 86].

Implanter la gestion des objets dynamiques au niveau des applications et non dans le système laisse toute liberté pour leur réalisation et notamment pour la conception de leur propre interface utilisateur. Il est ainsi possible par exemple de choisir et de modifier facilement l'aspect général des menus, le type des éléments (chaînes de caractères, icônes, etc.) et leur mode sélection (par un bouton de la souris ou par une touche du clavier, avec confirmation ou non, etc.).

Remarque : le système PNX-WMS qui gère le recouvrement des fenêtres interdit aux applications de manipuler leurs fenêtres. C'est pour cette raison que la gestion des menus dynamiques est intégrée dans le système. Les applications peuvent néanmoins contourner cette interdiction et changer la taille d'une fenêtre ou la déplacer en la détruisant puis en la recréant

immédiatement après avec de nouveaux paramètres. Le principe inverse (créer puis détruire une fenêtre) permet de simuler l'apparition temporaire d'une fenêtre. Cette technique est exploitée pour la mise en œuvre des menus et des formulaires dynamiques dans l'éditeur de documents Grif.

Cet exemple prouve qu'il est illusoire de vouloir interdire aux applications de manipuler les fenêtres dès lors qu'elles peuvent les créer et les détruire dynamiquement, ce qui est le minimum nécessaire pour la conception d'applications multi-fenêtres telles que Grif.

3.3. L'interface graphique

L'interface graphique occupe une place prépondérante dans un système multi-fenêtres car l'affichage d'informations représente la partie essentielle de la mise en œuvre de la communication avec l'utilisateur dans une application interactive. Le premier problème qui se pose pour le concepteur de cette interface est de fixer les besoins à couvrir car il n'y a a priori aucune limite au niveau des besoins des applications. La définition de l'interface graphique résulte donc d'un compromis entre d'une part le souci de satisfaire un maximum de besoins, d'autre part la nécessité de limiter la taille et la charge du système. Ce compromis est basé sur différents critères tels que l'efficacité des opérations et leur fréquence d'emploi par exemple.

L'interface graphique peut être décomposée en trois parties selon le type d'objets manipulés :

1) l'affichage de caractères ;

2) l'affichage d'objets graphiques ;

3) l'affichage d'images.

Un *objet graphique* est un ensemble de points satisfaisant une propriété géométrique particulière, ensemble qui peut donc être décrit par une fonction mathématique (cercle, rectangle, etc.). Inversement, une *image* est un ensemble non structuré de points décrit uniquement par son contenu, c'est à dire en « extension » selon l'expression mathématique consacrée.

3.3.1. L'affichage de caractères

Nous avons vu que les caractères sont regroupés en polices et que le principe de fonctionnement des écrans à points autorise l'emploi simultané de différentes polices (cf II.2.1). La majorité des systèmes gèrent l'ensemble des polices à deux niveaux :

1) l'ensemble des polices potentiellement utilisables sont stockées dans des fichiers. Chaque police est désignée dans cet espace par un *identificateur* ;

2) le sous-ensemble des polices qui sont chargées en mémoire centrale : l'affichage de caractères dans une police donnée nécessite son chargement préalable en mémoire centrale. Chaque police en mémoire centrale y est désignée par un *nom* attribué dynamiquement lors de son chargement.

L'espace des polices chargées est partagé par le système entre toutes les activités et le *nom* d'une police chargée est un nom global au niveau du système. L'interface de base inclut une primitive d'accès à une police désignée par son *identificateur*. Cette primitive a pour rôle de charger si nécessaire la police et de retourner son *nom* pour permettre aux applications de désigner la police dans les autres primitives de l'interface.

L'accès à une police est donc essentiellement un opération d'allocation de ressources dans le système : un *espace mémoire* pour ranger la description des différents caractères de la police et un *nom*. La primitive d'accès à une police peut donc échouer si l'une de ces ressources ne peut être allouée ou si l'application fournit un identificateur erroné, ce qui peut se produire dans la plupart des systèmes (Vitrail par exemple) qui interprètent l'identificateur comme un nom de fichier.

Pour éviter ce problème et faire en sorte que la primitive d'accès à une police n'échoue jamais, le système Andrew interprète l'*identificateur* d'une police comme la description des caractéristiques typographiques des caractères qu'elle contient, par exemple *TimesRoman12* où *12* indique la taille des caractères. L'identificateur fournit à la primitive d'accès sert alors d'indication au système qui recherche parmi les polices disponibles celle dont les caractéristiques se rapprochent le plus des caractéristiques décrites dans l'identificateur.

Cette technique garantit l'indépendance des applications vis à vis de la localisation effective des polices (qui peuvent éventuellement être stockées sur une machine distante) et résout le problème de la saturation des ressources, le système se

contentant dans ce cas de rechercher parmi les polices chargées celle qui satisfait au mieux la comparaison avec les caractéristiques demandées.

La plupart des systèmes offrent des primitives pour accèder aux caractéristiques d'une police (hauteur maximum des caractères, etc.) ou aux caractéristiques d'un caractère d'une police (largeur, hauteur, position par rapport à la ligne de base, etc.). Ces informations sont nécessaires pour gérer la disposition des caractères selon un schéma de présentation précis et pour retrouver le caractère désigné par un point (x, y) de la fenêtre.

Les caractères sont affichés dans une police donnée et à une position donnée dans la fenêtre. Le principe couramment adopté consiste à associer à chaque fenêtre différents attributs contenant ce type d'informations et à inclure dans l'interface du système les primitives permettant de modifier chaque attribut et d'accèder à sa valeur courante. Les attributs d'une fenêtre sont généralement les suivants :

- le nom de la police courante ;
- la position courante d'affichage ;
- l'aspect du curseur visualisant cette position : le curseur est défini par un caractère appartenant obligatoirement à la police courante ou à une police indépendante de celle-ci comme dans Andrew par exemple ;
- l'état allumé ou éteint du curseur.

Ce principe permet d'inclure dans l'interface du système une primitive d'affichage de caractères ayant comme seuls paramètres d'appel un nom de fenêtre et une chaîne de caractères (une suite d'octets). Cette primitive doit gérer le déplacement du curseur, c'est à dire effacer le curseur à la position courante, afficher les caractères puis mettre à jour la position courante et y afficher de nouveau le curseur. L'interface du système peut inclure également une primitive plus élaborée permettant l'affichage d'une chaîne de caractères dans une police et/ou à une position différentes de celles associées à la fenêtre. Cette primitive évite ainsi à l'application d'avoir à modifier et à restaurer les attributs correspondants pour afficher uniquement une chaîne particulière.

Tous les systèmes multi-fenêtres adaptés à Unix doivent simuler le fonctionnement d'un terminal alphanumérique classique dans les fenêtres servant de support à l'exécution du *Shell* et de l'environnement standard d'Unix. La majorité de ces systèmes choisissent d'émuler les caractéristiques d'un terminal existant sur le

marché (un VT100 dans Oriel par exemple). Nombre d'entre eux en profitent pour enrichir les fonctions mises à la disposition de l'utilisateur dans ce type de fenêtres. Le système Apollo associe ainsi à chacune de ces fenêtres un fichier d'historique (le « pad ») que l'utilisateur peut éditer par « couper-coller » de façon à fournir en entrée des informations qui y sont contenues.

3.3.2. L'affichage d'objets graphiques

Rappelons qu'un *objet graphique* est un ensemble structuré de points pouvant être décrit par une propriété particulière. Un objet graphique est donc affiché par l'exécution d'une primitive dont l'algorithme consiste à produire l'ensemble des points satisfaisant à la propriété correspondante. De nombreux systèmes gèrent l'affichage dans une fenêtre des objets graphiques *élémentaires* tels que les segments de droite, les polygones, les cercles ou les arcs de cercle, les ellipses, etc. Il s'agit alors uniquement de tracer le contour de ces objets ; il est également intéressant pour les applications de pouvoir remplir ces objets à l'aide d'un motif quelconque. Par exemple, l'interface graphique du système Andrew inclut la primitive

FillTrapezoid(x1, y1, l1, x2, y2, l2, f, c)

qui remplit le trapèze décrit par deux segments horizontaux *((x1,y1),l1)* et *((x2,y2),l2)* avec le caractère *c* de la police *f*.

Remarque : dans Andrew, toutes les primitives graphiques opèrent dans une fenêtre « courante » spécifiée par l'application à l'aide d'une primitive particulière de l'interface du système.

L'ensemble des fonctions d'affichage d'objets graphiques peut être considéré comme le jeu d'instructions d'un langage très simple à l'aide duquel les applications programment l'affichage d'objets graphiques plus élaborés. Ces programmes sont exécutés par les applications, le système se contentant d'interpréter les primitives d'affichage qu'ils contiennent. Considérons par exemple une application qui veut afficher dans une fenêtre une grille telle que celle dessinée dans la figure 3.7 ci-dessous.

En supposant que l'interface graphique du système inclut la fonction

Tracer_Seg(x1, y1, x2, y2)

qui affiche un segment de droite entre deux points *(x1,y1)* et *(x2,y2)*, le programme exécuté par l'application pour tracer la grille de la figure 3.7 appelle *18* fois la fonction *Tracer_Seg*.

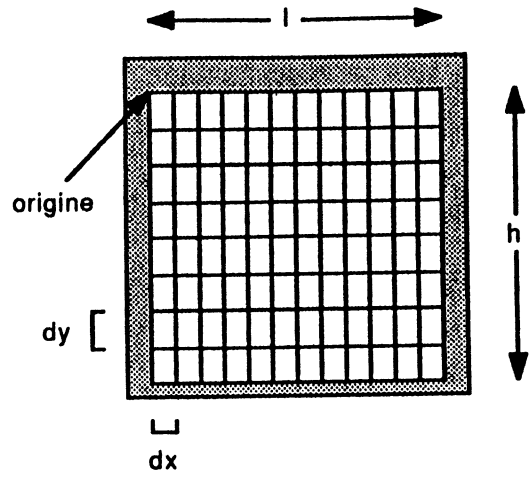


Figure 3.7 Exemple de tracé d'une grille dans une fenêtre

Cette succession d'appels diminue l'efficacité du programme car l'invocation d'une fonction du système est en général une opération coûteuse qui peut éventuellement nécessiter l'envoi d'un message à un processus serveur (cf III.4.2.1 suivant). Ce problème provient du fait que l'interface graphique du système est basé sur un langage simple qui exclut toute possibilité de « programmation » au sens classique du terme.

Une solution consiste alors à étendre le langage pour en faire un véritable langage de programmation et à implanter dans le système un interpréteur de ce langage. Dans ce cas, les applications n'invoquent plus des fonctions de base du système mais transmettent des programmes qui sont exécutés par le système. C'est le principe adopté pour la nouvelle version du système multifenêtres SunDew développée pour le Sun [Gosling 86].

L'interface graphique de SunDew est basée sur le langage PostScript [Adobe 85] conçu à l'origine pour communiquer avec des imprimantes de haute définition selon la démarche suivante : l'ordinateur envoie à l'imprimante des programmes PostScript qui sont interprétés sur un processeur local, interprétation qui provoque finalement l'impression d'un dessin sur le papier. Le langage PostScript a été retenu pour l'imprimante à laser « LaserWriter » du Macintosh par exemple.

PostScript est un langage très puissant qui intègre les capacités d'un langage algorithmique classique - manipulation de variables, structures de contrôle, appel de sous-programmes, etc. - dans un formalisme de description d'images basé sur le remplissage de *patrons* (« stencils » en anglais) par une *peinture*.

Un *patron* est un contour composé de courbes splines non obligatoirement convexes décrites dans un espace de coordonnées réelles (c'est à dire non obligatoirement entières). Une *peinture* est une couleur ou un motif - une autre image par exemple - qui est appliquée sur la surface d'affichage à travers un patron selon un principe analogue à celui de la sérigraphie.

Ce principe constitue le modèle général d'affichage dans PostScript : les caractères et les lignes peuvent être définis par des patrons, même si en réalité ces objets sont élaborés différemment par le système qui exploite autant que possible les caractéristiques particulières de certains objets de base.

L'interface graphique de SunDew comprend en fait deux couches. La premiere cache entièrement l'existence de PostScript dans un ensemble de procédures qui construisent et transmettent des programmes PostScript. Ces procédures sont regroupées dans des bibliothèques standard et assurent l'affichage des objets graphiques les plus fréquemment manipulés par les applications. Le second niveau expose le langage PostScript au programmeur d'applications, ce qui est inévitable au delà du niveau de fonctionnalités offertes en standard.

L'interprétation d'un langage tel que PostScript représente pour le système SunDew une charge beaucoup plus importante que celle supportée par les systèmes qui se contentent de mettre en œuvre un ensemble de primitives graphiques prédéfinies. Le principe consistant à transmettre des programmes et à les interpréter « localement » dans le système offre par contre de nombreux avantages dans le contexte d'un système multi-fenêtres.

Il permet tout d'abord de rendre le système beaucoup plus souple : la possibilité de définir des fonctions augmente « à l'infini » les capacités du système. Il contribue d'autre part à accroître les performances en réduisant le nombre d'appels aux primitives système et le volume des informations transmises par les applications pour afficher un objet complexe. L'exemple du programme de tracé d'une grille introduit précédemment illustre tout à fait les avantages de cette démarche. Dans SunDew, ce programme a uniquement à transmettre au système la procédure PostScript suivante (en notation postfixée qui est celle de PostScript) pour tracer la grille de la figure 3.7 :

```
newpath
dx 0 moveto
11 (0 H rlineto dx -H rmoveto) repeat
0 dy moveto
7 (L 0 rlineto -L dy rmoveto) repeat
```

La première ligne du programme initialise un nouveau chemin (un patron). Rappelons qu'un chemin est formé d'un ensemble de courbes appelées segment dans la terminologie PostScript.

Les instructions des deux lignes suivantes tracent toutes les verticales de la grille. L'opérateur *moveto* commence un nouveau segment du chemin en *(dx, 0)*. Ce point est le point courant du chemin et correspond ici à l'origine de la première verticale de la grille. L'opérateur *rlineto* termine le segment courant en traçant une droite verticale de hauteur H depuis le point courant et déplace celui-ci relativement à son ancienne position. Le point courant est alors en bas de la verticale tracée. L'opérateur *rmoveto* commence un nouveau segment en déplaçant le point courant de *(dx, -H)* relativement à son ancienne position, ce qui le place à l'origine de la verticale suivante de la grille. Cette séquence est répétée *11* fois. Les instructions des deux dernières lignes tracent toutes les horizontales de la grille en suivant la même démarche.

3.3.3. La manipulation d'images

Rappellons qu'une *image* est un ensemble non structuré de points qui ne peut être décrit que par son contenu. Une image est une matrice de points telle que nous l'avons définie en II.2.1. où nous avons présenté les opérations pouvant être appliquées sur des matrices de points. Ces opérations appelées « RasterOp » consistent essentiellement à appliquer une fonction logique (OU, ET, etc.) entre deux matrices de points et à ranger le résultat dans l'une d'entre elles ou à copier une matrice-source dans une matrice-cible.

L'interface de base de la majorité des systèmes (Andrew, Vitrail, etc.) inclut des primitives équivalentes permettant de manipuler le contenu d'une fenêtre comme une matrice de points. Ces opérations sont exploitées par les applications pour implanter des fonctions plus élaborées telles que la copie d'informations dans une même fenêtre - pour faire défiler l'information par exemple - ou la mise en évidence d'un objet par passage en inverse vidéo du rectangle l'englobant dans la

fenêtre. Insistons sur le fait que toutes ces opérations portent exclusivement sur le contenu des fenêtres.

Certains systèmes tels que X autorisent également le *transfert* d'images entre les applications et les fenêtres. Dans X, le transfert de matrices de points s'effectue en deux temps : les applications demandent l'allocation dans le système d'un espace-mémoire - un tampon - et disposent de primitives pour transférer une image entre ce tampon et leur propre espace d'adressage ou entre le tampon et l'espace d'affichage d'une fenêtre.

Le principe du transfert d'images entre une application et une fenêtre est adopté dans Oriel pour implanter l'ensemble des opérations graphiques du système dont l'interface de base inclut uniquement l'opération de *copie* dans une fenêtre d'une matrice de points contenue dans l'espace d'adressage de l'application. Le principe sous-jacent est le suivant :

- les applications associent à une fenêtre une matrice de points d'une taille en général égale à celle de [l'intérieur de] la fenêtre ;

- toutes les primitives de la bibliothèque graphique opèrent sur une matrice de points passée en paramètre et demandent ensuite la copie de la partie modifiée de cette matrice dans la fenêtre à laquelle elle est associée.

Permettre le transfert d'images aux applications a l'avantage de ne pas limiter en théorie les capacités graphiques du système mais a par contre l'inconvénient d'être peu efficace. Son emploi pose également le problème de l'indépendance des applications par rapport à l'organisation imposée à la mémoire-écran par son mécanisme de rafraîchissement (cf II.2.1). Pour assurer cette indépendance, le système doit intégrer au niveau de son interface de base la notion de matrice de points « virtuelle » et adapter la copie de telles matrices dans la mémoire-écran aux caractéristiques particulière du matériel. Cette technique, adoptée dans Oriel, peut s'avèrer complexe à mettre en œuvre et être éventuellement incompatible avec l'emploi de processeurs spécialisés dans la manipulation de matrices de points. Notons également que l'exploitation d'un écran couleur est difficile (et encore moins efficace) ou même impossible comme dans Oriel ou les matrices « virtuelles » sont monochromes.

3.4. L'interface d'entrée

Nous avons vu que la majorité des systèmes actuels allouent l'ensemble des dispositifs d'entrée à l'activité courante sélectionnée par l'utilisateur, c'est à dire qu'il y a à un instant donné au plus une seule activité ayant accès à l'ensemble des dispositifs d'entrée gérés par le système (cf III.2.2.1.1). C'est sur cette base que nous allons étudier maintenant l'interface d'entrée des systèmes multi-fenêtres. Du point de vue des applications, la définition de l'interface d'entrée comprend les deux aspects suivants :

1) les événements transmis par le système ;

2) la manière dont ces événements sont transmis aux applications.

Nous avons choisi de décrire séparément ces deux aspects bien qu'ils ne soient pas toujours complètement indépendants : nous verrons par exemple que le choix du mode de transmission peut être fonction du type de l'événement à transmettre. Nous aborderons ensuite la gestion du réticule de la souris, c'est à dire la manière dont le système permet aux applications de choisir le réticule de la souris en fonction de sa position sur l'écran.

3.4.1. Evénements transmis par le système

Les événements transmis peuvent être répartis en deux classes :

- les *événements physiques* reflétant toute action de l'utilisateur sur un des dispositifs d'entrée : touches du clavier, boutons de la souris et déplacement de la souris ;

- les *événements synthétisés* par le système et qui correspondent à des actions implicites de l'utilisateur.

3.4.1.1. Evénements physiques

Les événements physiques sont générés à la suite d'un changement d'état des dispositifs d'entrée : frappe d'une touche du clavier, enfoncement ou relâchement des boutons de la souris, déplacement de la souris. Il est préférable que le système transmette la valeur courante de chaque dispositif - l'état de tous les boutons de la

souris par exemple - de façon à simplifier la tâche des applications et leur éviter de perdre l'état global de chaque dispositif.

Il est souhaitable que le système transmette la position de la souris relativement à l'origine de la plus petite unité de structuration de la surface de l'écran : X et Oriel fournissent la position de la souris par rapport aux fenêtres « feuille » de l'arbre de fenêtres qu'ils gèrent. Le système PNX-WMS permet aux applications de spécifier le pas minimum de déplacement de la souris qui doit provoquer la transmission d'un événement correspondant : cette technique évite au système de transmettre des événements inutiles pour l'application et à celle-ci d'avoir à les reconnaître.

Les systèmes comme Oriel, PNX-WMS, SunWindows, Macintosh ou W datent chaque événement transmis. Ce procédé est très utile pour réaliser des commandes basées sur l'intervalle de temps séparant deux actions de l'utilisateur. C'est le cas de la commande de sélection d'une icône par « double-appui » du bouton de la souris sur le Macintosh.

Le système peut regrouper dans un même événement « logique » l'état de tout ou partie des dispositifs d'entrée de façon à accroître les capacités de base de chacun d'eux. SunWindows transmet simultanément l'état de certaines touches du clavier (« control », « shift », etc.) et celui des boutons de la souris. Cette technique permet de multiplier le nombre de fonctions différentes qu'une application peut associer simultanément aux boutons de la souris. SunWindows applique en fait ce principe de manière globale en transmettant systématiquement la position de la souris dans chaque événement logique.

3.4.1.2. Evénements synthétisés

Le principe sous-jacent consiste à fournir des informations contextuelles qu'il serait difficile voire impossible d'élaborer au niveau des applications, informations dont elles peuvent avoir besoin pour se placer dans un état cohérent vis à vis de l'utilisateur. C'est le cas par exemple lorsque la souris franchit le bord d'une fenêtre ou lors du changement d'activité courante.

Les systèmes X, Andrew et SunWindows génèrent un événement lorsque la souris entre dans une fenêtre ou en sort. Ces informations sont utiles lorsque le traitement effectué par une activité sur le contenu d'une fenêtre est fonction de la position de la souris et donc de sa présence ou non dans la fenêtre. Elles deviennent

même nécessaires si le système ne transmet pas la position de la souris lorsque celle-ci est en dehors des fenêtres de l'activité courante, ce qui est le cas dans SunWindows où la position de la souris détermine l'activité courante (cf III.2.2.2.1). L'utilité de ce type d'informations peut être montrée à travers l'exemple des menus dynamiques.

Dans SunWindows, les menus dynamiques sont implantés dans des fenêtres et gérés selon l'approche classique qui consiste à mettre en évidence l'élément désigné en faisant passer en inverse vidéo le rectangle l'englobant dans le menu. La même opération doit être appliquée sur ce rectangle lorsque la souris ne désigne plus cet élément :

- soit parce que l'utilisateur l'a déplacée pour désigner un autre élément, ce qui peut être déterminé au niveau de l'activité puisque le souris ne quitte pas la fenêtre du menu ;
- soit parce qu'il l'a déplacée en dehors de la fenêtre, ce qui ne peut plus être détecté par l'activité qui a alors besoin de l'événement correspondant transmis par le système.

Dans SunWindows, ces événements sont également utilisés pour informer une activité qu'elle est devenue ou respectivement qu'elle n'est plus activité courante. Ces changements d'état sont transmis à l'aide de deux événements spécifiques dans les systèmes Vitrail, Andrew et X. Ce principe est intéressant à plus d'un titre. Il permet tout d'abord de faire gérer la mise en évidence de l'activité courante par les activités elles-mêmes (cf III.2.2.1.1). Cette technique est adoptée dans SunWindows.

L'activité qui quitte l'état « activité courante » peut d'autre part avoir besoin de cette information pour terminer ou annuler une commande en cours d'élaboration en faisant disparaître un menu dynamique par exemple. Elle peut également profiter du fait qu'elle n'est plus activité courante pour accomplir diverses tâches coûteuses en temps d'exécution et pouvant être différées jusqu'à ce moment : un éditeur peut mettre à jour en mémoire secondaire le document en cours d'édition.

Lorsque l'utilisateur sélectionne l'activité courante en sélectionnant un point d'une de ses fenêtres, l'événement décrivant le point désigné peut être transmis à l'activité - ce que fait Vitrail - ou au contraire être « consommé » par le système comme dans PNX-WMS. La première méthode permet à l'utilisateur d'effectuer deux opérations en une seule commande : changer d'activité courante et désigner un

objet à cette activité, un élément d'un menu statique affiché dans la fenêtre par exemple. La seconde méthode oblige par contre l'utilisateur à sélectionner deux fois de suite le même point sur l'écran, la première fois pour changer d'activité courante et la seconde pour sélectionner l'objet en question. Inversement, elle évite par exemple à l'utilisateur de changer contre son gré la position courante d'affichage dans la fenêtre qu'il a désignée pour sélectionner une nouvelle activité. Ceci est notamment intéressant lorsque cette fenêtre est en partie recouverte.

Les systèmes Vitrail, X et Andrew qui ne gèrent pas le contenu des fenêtres [en partie] cachées utilisent le principe des événements synthétisés pour transmettre aux applications des requêtes de « repeinture » lorsque l'utilisateur a changé la taille d'une fenêtre ou l'a fait [en partie] réapparaître sur l'écran. Les systèmes Oriel, W, et PNX-WMS qui gèrent le recouvrement des fenêtres de façon transparente aux applications transmettent un tel événement uniquement lorsque l'utilisateur a changé la taille d'une fenêtre.

3.4.2. Transmission des événements

Le principe généralemnt appliqué pour transmettre les événements consiste à mémoriser les événements dans des files d'attente auxquelles les applications accèdent par l'intermédire de primitives système. La transmission des événements aux applications comprend deux aspects :

1) le *mode de transmission*, c'est à dire la manière dont une activité est avertie par le système de l'arrivée ou de la présence d'événements qui lui sont destinés ;

2) le *support de transmission* des événements, c'est à dire le type d'entité que doit désigner l'application pour accèder aux événements.

3.4.2.1. Mode de transmission

Le mode de transmission peut être *synchrone* ou *asynchrone*.

• *transmission synchrone* : les événements sont uniquement transmis aux activités lorsqu'elles le demandent explicitement par appel d'une primitive de lecture. Lorsqu'aucun événement n'est disponible pour l'activité, cette primitive peut être bloquante - éventuellement durant un délai maximum fixé au

moment de l'appel - ou au contraire être non bloquante et retourner par exemple l'état courant des dispositifs d'entrée (état des boutons, position de la souris, etc.) ;

• *transmission asynchrone* : le système avertit l'activité de l'arrivée d'un événement par un mécanisme d'interruption logicielle analogue dans son principe au mécanisme d'envoi d'un « signal » à un processus dans Unix. L'activité accède ensuite à l'événement par appel d'une opération de lecture comme dans le cas précédent.

La transmission asynchrone évite de bloquer en attente d'un événement utilisateur les applications qui doivent également communiquer avec d'autres entités du système. Elle complique par contre la programmation des applications qui ont à prendre en compte les problèmes classiques des mécanismes d'interruption tels que la sauvegarde du contexte courant d'exécution et l'accès en exclusion mutuelle aux données partagées. C'est pourquoi cette méthode n'est pas employée, si ce n'est dans le système W par exemple, ou réservée à la transmission d'événements particuliers.

Le système Andrew développé sur Unix transmet de manière asynchrone les événements de « repeinture » par l'intermédiaire d'un « signal » particulier. Les systèmes PNX-WMS et Oriel adoptent le même principe pour transmettre les événements de changement de taille d'une fenêtre. Dans les deux cas, il s'agit d'interrompre les activités afin qu'elles prennent immédiatement en compte les modifications effectuées par l'utilisateur sur la disposition des fenêtres sur l'écran.

Le système X transmet au contraire de manière synchrone tous les événements, y compris les événements de « repeinture » et de changement de taille d'une fenêtre. Les activités ne sont donc informées du changement de taille d'une fenêtre qu'au moment où elles demandent à accéder aux événements qui leur ont été envoyés. Il s'ensuit par exemple qu'une activité peut être amenée à poursuivre l'affichage d'informations dans une fenêtre dont la taille a été agrandie entre temps par l'utilisateur.

Ce principe simplifie par contre la conception du système qui n'a à gérer qu'un seul mode de transmission des événements. Il simplifie également la programmation des applications en leur évitant les problèmes de gestion des interruptions cités précédemment.

3.4.2.2. Support de transmission

Quel que soit le mode de transmission adopté, les applications doivent en général appeler explicitement une primitive de lecture pour accèder aux événements qui leur ont été envoyés. Nous pouvons considérer trois méthodes d'accès selon le type d'entité que doit désigner l'activité pour récupérer les événements :

1) le système gère une file d'attente par dispositif d'entrée. Ce principe est appliqué sur le Macintosh qui offre deux primitives, l'une pour obtenir la position courante de la souris et l'autre pour lire les événements associés aux touches du clavier et aux boutons de la souris, boutons qui sont considérés comme une extension du clavier ;

2) le système gère une file d'événements par fenêtre et les applications désignent là ou les fenêtres dans la primitive de lecture. Cette technique est utilisée dans Andrew, PNX-WMS et Oriel ;

3) le système alloue à chaque activité une seule file d'entrée dans laquelle il range tous les événements qui lui sont destinés. C'est l'approche suivie dans les systèmes W et X.

La première technique est simple à implanter dans un système mono-activité comme le Macintosh. Ce système ne gère pas de file d'événements pour les déplacements de la souris mais se contente de retourner la position courante de la souris dans la primitive correspondante. Gérer une file d'événements par dispositif d'entrée complique par contre le partage de ces dispositifs dans un système multi-activités et ne permet pas de transmettre les événements dans l'ordre où ils ont été émis.

Ce principe complique également la programmation des applications car le système ne peut pas leur transmettre dans un même événement « logique » l'état de tous les dispositifs d'entrée (cf III.3.4.1.1). Les applications développées sur le Macintosh doivent systématiquement appeler la primitive d'accès à la position courante de la souris après avoir reçu un événement associé au changement d'état du bouton de la souris.

La seconde technique consiste à assimiler une fenêtre à un *terminal virtuel* et non plus uniquement à un *écran virtuel* tel que nous l'avons défini en introduction de ce chapitre, à savoir une surface d'affichage et un système de coordonnées. Cette technique n'a de sens que dans le cas d'activités mono-fenêtre où la notion d'activité

courante et celle de fenêtre courante sont équivalentes (cf III.2.2.1.1). La fenêtre joue alors effectivement le rôle de terminal virtuel aussi bien pour le système que pour les applications. Ce principe n'est par contre plus adapté aux activités multi-fenêtres pour les raisons suivantes :

1) la notion de fenêtre courante n'a dans ce cas plus de sens au niveau des entrées, notamment en ce qui concerne les touches du clavier : il est impossible pour le système de choisir à priori la fenêtre qui doit « recevoir » les caractères frappés par l'utilisateur.

Seule l'application est à même d'effectuer ce choix, comme le prouve l'exemple de l'éditeur introduit en III.2.2.1.1 où la fenêtre sélectionnée par l'utilisateur pour sélectionner l'activité courante peut être différente de celle où sont affichés les caractères entrés au clavier. Ce problème se pose également pour la transmission des événements synthétisés lors du changement d'activité courante.

2) l'utilisateur ayant le choix de la fenêtre à travers laquelle il peut communiquer avec une activité, les activités doivent potentiellement attendre un événement depuis toutes leurs fenêtres. Cela suppose alors que la primitive de lecture du système autorise l'attente simultanée sur plusieurs flots d'entrée, ce qui est impossible dans Unix version 7 par exemple.

3) certaines applications peuvent dédier une (ou plusieurs) de leurs fenêtres à l'affichage d'informations et uniquement à cela. C'est le cas par exemple de l'application *talk* du système Unix qui permet à deux utilisateurs connectés au système de dialoguer depuis leur terminal respectif.

Le programme *talk* partage horizontalement l'écran de chaque terminal en deux « fenêtres » pour afficher séparément les informations transmises de celles reçues au cours du dialogue. Les caractères entrés par chaque utilisateur sont affichés dans la fenêtre du haut de son terminal et dans la fenêtre du bas de celui de son interlocuteur. Sur chaque terminal, la fenêtre du bas ne sert qu'à l'affichage et n'est donc pas un terminal virtuel.

La troisième méthode consistant à associer une seule file d'entrée par activité évite tous ces problèmes. Du point de vue du système, une activité est alors représentée par sa file d'entrée dans laquelle le système range tous les événements qui lui sont destinés. Le système associe à chaque fenêtre créée par une activité le « nom » de cette activité, c'est à dire l'identificateur de la file d'événements qui

représente l'activité dans le système. Le système doit inclure l'dentificateur de la fenêtre désignée dans les événements décrivant la position de la souris puisque cet identificateur n'est plus fourni implicitement par la primitive de lecture comme dans la seconde solution.

Gérer une seule file d'entrée par activité simplifie d'autre part la conception du système et garantit par définition la transmission des événements dans l'ordre où ils ont été émis, ce qui est fondamental pour la cohérence du dialogue entre l'utilisateur et les activités.

3.4.3. Gestion du réticule de la souris

La plupart des systèmes permettent aux activités de choisir l'aspect du réticule de la souris en fonction de sa position sur l'écran. Le Macintosh et le système PNX-WMS se contentent d'offrir une primitive pour changer dynamiquement le réticule, ce qui oblige les applications à effectuer elles-mêmes le suivi de la souris dans leurs fenêtres. Nous avons vu que l'interface du Macintosh offre une primitive particulière pour accèder à la position courante de la souris. Le système PNX-WMS, qui est multi-activités, permet aux activités d'avoir en permanence accès à la position courante de la souris, que les activités soient ou non l'activité courante du point de vue de l'utilisateur.

Dans les systèmes SunWindows, W, X et Andrew, les applications associent un réticule à chaque fenêtre et le système gère ensuite automatiquement le changement du réticule en fonction de la fenêtre désignée par la souris. Le système Oriel adopte le même principe en le raffinant : les applications disposent d'une primitive pour découper une fenêtre en régions rectangulaires et leur associer un réticule particulier.

Cette seconde technique est préférable à la première pour les raisons suivantes :

1) *elle simplifie la programmation des applications*, notamment celles dont l'interface utilisateur est conçue selon l'approche classique qui consiste à prendre uniquement en compte la position de la souris au moment où l'utilisateur effectue une sélection en appuyant ou en relâchant un des boutons de la souris.

Sur le Macintosh ou sur PNX-WMS, de telles applications doivent néanmoins gérer systématiquement tous les déplacements de la souris dans leurs fenêtres pour assurer l'affichage de réticules différents.

2) *dans un contexte multi-activités tel que celui d'Unix, elle améliore l'efficacité globale du système.* Le système ne transmet la position de la souris qu'aux activités pour lesquelles cette information est nécessaire, par exemple pour mettre en évidence l'élément courant d'un menu dynamique. De plus, la position de la souris est uniquement transmise à l'activité courante.

Dans PNX-WMS, la position de la souris doit être systématiquement transmise aux activités qui gèrent l'affichage de réticules différents dans leurs fenêtres. Le système est donc obligé de « réveiller » l'activité propriétaire de la fenêtre dans laquelle est déplacée la souris. Ces changements de contexte diminuent l'efficacité globale du système, en particulier si le contexte de l'activité réveillée doit être ramené en mémoire centrale.

3) *elle garantit en toutes circonstances un temps de réponse immédiat pour le suivi de la souris sur l'écran.* Ce principe est fondamental car tout retard dans le suivi de la souris est perçu de façon très pénible par l'utilisateur.

Un tel retard est inévitable sur le Macintosh ou sur PNX-WMS lorsque l'activité qui doit gérer l'affichage du réticule à la position courante de la souris est en train d'accomplir une tâche quelconque et ne peut être « interrompue ». C'est le cas lorsqu'elle effectue une opération d'entrée/sortie en mémoire secondaire, opération qui est synchrone sur le Macintosh et sur PNX-WMS. C'est également le cas dans PNX-WMS si le contexte de cette activité doit être préalablement ramené en mémoire centrale.

4. Architecture des systèmes multi-fenêtres

4.1. Introduction

Nous avons présenté les fonctions des systèmes multi-fenêtres dans les paragraphes précédents en considérant le système comme une entité unique - une « boite noire » - ayant pour rôle de gérer la communication entre l'utilisateur et les programmes d'application. Nous allons maintenant étudier la manière dont ces fonctions peuvent être implantées et envisager les différents aspects qui interviennent dans la conception d'un système multi-fenêtres.

Nous ne considérerons ici que les systèmes multi-activités en raison des difficultés et des contraintes qu'impose la prise en compte du parallélisme dans la réalisation d'un système multi-fenêtres. Nous avons choisi de limiter notre étude aux systèmes développés sur Unix qui constituent la majorité des systèmes existant actuellement et parce qu'Unix tend à devenir de fait le système d'exploitation standard des postes de travail. Ce choix nous permet également d'introduire les problèmes que nous avons rencontrés lors de la mise en œuvre du système Fenix.

Dans un contexte multi-activités, un des rôles essentiels du système multi-fenêtres consiste à gérer un ensemble de ressources partagées et à synchroniser les activités qui accèdent en parallèle à ces ressources, tout en garantissant des performances acceptables pour l'utilisateur. L'ensemble des ressources partagées inclut les ressources matérielles du poste de travail (mémoire-écran, dispositifs d'entrée, etc.) et les structures de données associées à leur exploitation, par exemple les structures de données décrivant chaque fenêtre (taille, position sur l'écran, etc.) et les polices de caractères.

Comme exemple de problème de synchronisation, considérons le cas du déplacement d'une fenêtre sur l'écran provoquée par une commande de l'utilisateur. Cette opération est effectuée en parallèle avec les activités. Elle peut donc éventuellement interrompre une activité au milieu d'une opération de tracé d'un segment de droite dans une autre fenêtre. Si cette fenêtre devient [en partie] recouverte par la fenêtre déplacée, l'activité interrompue va reprendre l'opération dans un contexte erroné et tracer le reste du segment dans la fenêtre déplacée. La manipulation des fenêtres et les opérations graphiques doivent être synchronisées pour éviter ce type d'erreurs.

Dans un système multi-activités tel qu'Unix, les fonctions d'un système multi-fenêtres peuvent être a priori implantées à trois endroits différents :

1) dans un programme « serveur » ;

2) dans chaque programme d'application ;

3) dans le noyau du système.

Le choix entre l'une ou l'autre de ces possibilités est basé sur différents critères parmi lesquels nous retiendrons :

- la *performance*. Les aspects communication et synchronisation pour l'accès aux ressources partagées interviennent pour beaucoup sur la performance du système ;
- la *souplesse*, c'est à dire la capacité d'évolution et d'adaptation du système. Cet aspect concerne notamment les opérateurs graphiques mis à la disposition des programmes d'application ;
- la *portabilité* des programmes d'application et du système multi-fenêtres lui-même ;
- la *facilité de développement* et de mise au point du système.

Le paragraphe suivant est consacré à l'étude des principales solutions qui peuvent être adoptées pour répartir les fonctions d'un système multi-fenêtres développé sur Unix. Nous étudions successivement chaque solution en nous appuyant à chaque fois sur un exemple de système existant et nous montrons les avantages et inconvénients de chacune d'elles vis à vis des critères introduits ci-dessus.

4.2. Qui affiche sur l'écran ?

La démarche que nous avons choisi de suivre pour aborder la conception d'un système multi-fenêtres consiste à envisager les différentes solutions qui peuvent être adoptées pour implanter les opérations graphiques. Plusieurs raisons justifient cette démarche :

• les opérations graphiques ont un rôle prépondérant dans le système car ce sont les opérations les plus fréquemment exécutées par les activités ;

• les opérations graphiques constituent un point de référence privilégié pour aborder le problème de la synchronisation des activités dans la mesure où ces opérations accèdent à la plupart des ressources partagées du système (mémoire-écran, structures de données décrivant les fenêtres, etc.).

Nous allons donc décrire successivement les trois approches qui peuvent être adoptées pour implanter les opérations graphiques d'un système multi-fenêtres développé sur Unix, ce que nous résumons par la question « Qui affiche sur l'écran ? »

4.2.1. Un programme serveur

Dans les systèmes tels que Unix 4.2BSD qui offrent un mécanisme de communication par messages entre processus, le code des opérations graphiques peut être placé dans un programme indépendant qui joue le rôle de serveur d'affichage vis à vis des programmes d'application. Le serveur est le seul à accèder à la mémoire-écran et interprète les commandes d'affichage transmises dans des messages par les programmes d'application.

Dans ce schéma, le serveur contrôle l'ensemble des ressources partagées et regroupe toutes les fonctions de base du système, à savoir :

• les opérations de création, de destruction et de manipulation des fenêtres ;

• les opérations graphiques ;

• la transmission des événements aux activités.

Les seules fonctions du système implantées dans le noyau sont les pilotes des dispositifs d'entrée. Le serveur est le seul « client » de ces pilotes. Il reçoit tous les événements physiques émis par l'utilisateur, y compris les déplacements de la souris, et les transmet aux activités par des messages. Le serveur assure également le suivi de la souris sur l'écran si cette fonction n'est pas effectuée automatiquement par le contrôleur de la souris.

Cette solution est adoptée dans les systèmes SunDew, Andrew et X développés sur Unix 4.2BSD. Dans Andrew et SunDew, l'interface utilisateur du système est également implantée dans le serveur. Dans le système X, la gestion de l'interface

utilisateur est par contre assurée par une activité indépendante se comportant comme un client privilégié du serveur. Les figures 3.8 et 3.9 illustrent l'architecture des systèmes Andrew et X respectivement.

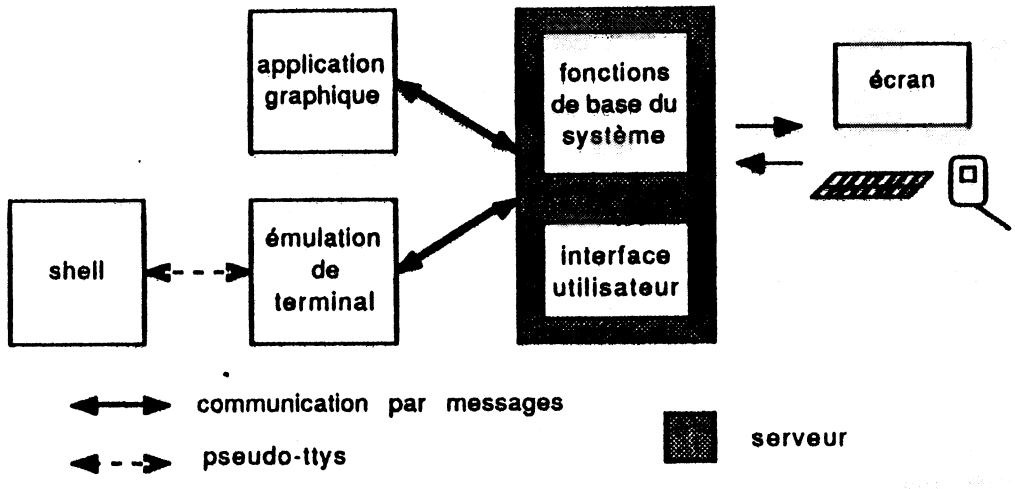


Figure 3.8 Architecture du système Andrew

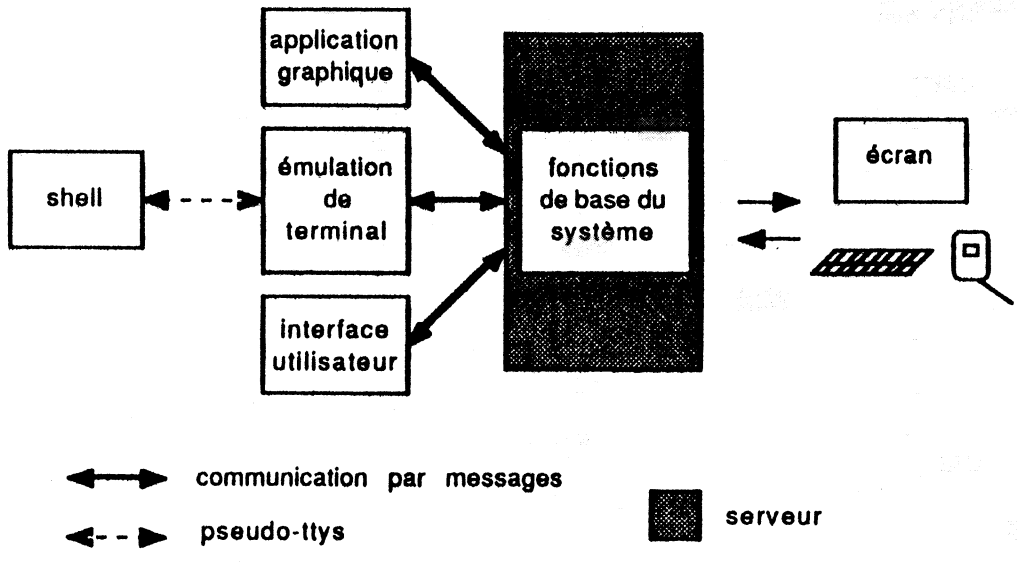


Figure 3.9 Architecture du système X

Les systèmes Andrew, X et SunDew implantent la gestion des terminaux virtuels dans des programmes exécutés par des processus séparés. Chaque *Shell* (et tous ses descendants) est interconnecté au niveau de ses flots d'entrée-sortie

standard avec un processus qui exécute un programme d'émulation de terminal. Ce programme affiche dans la fenêtre les caractères reçus du *Shell* et inversement envoie au *Shell* les caractères émis par l'utilisateur. Dans tous ces systèmes, le programme d'émulation de terminaux gère également les commandes de « couper-coller » permettant à l'utilisateur de transférer des chaînes de caractères entre plusieurs terminaux virtuels. La connection entre le *Shell* et l'émulateur de terminal est assurée par l'intermédiaire du mécanisme de communication appelé « pseudo-tty » dans la terminologie d'Unix 4.2BSD. Décrivons brièvement le principe de fonctionnement de ce mécanisme illustré dans la figure 3.10.

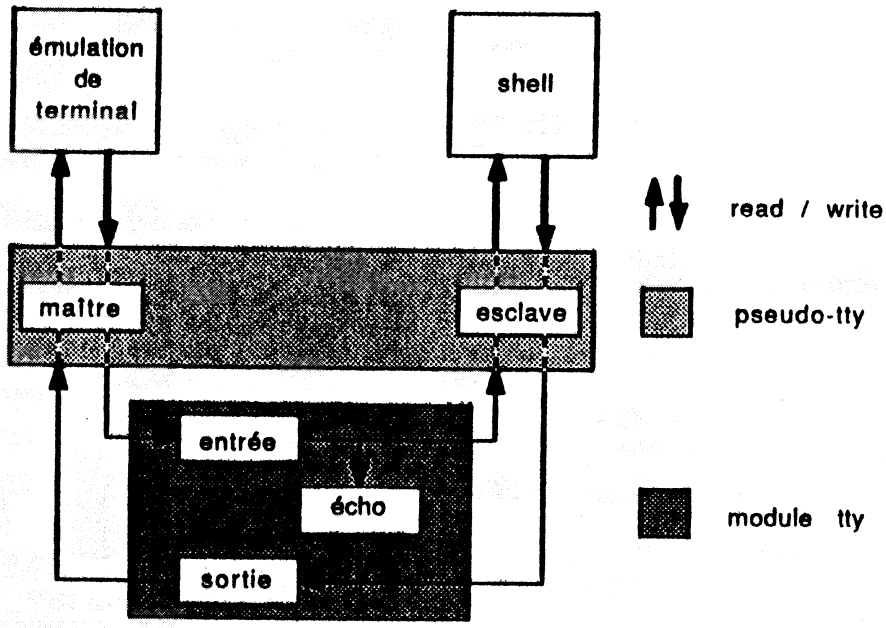


Figure 3.10 Principe de fonctionnement des pseudo-ttys

Les « pseudo-tty » sont des fichiers spéciaux regroupés par paire comportant chacune un élément esclave et un élément maître. Ces deux éléments sont reliés à travers le module *tty* du noyau comme le sont d'une part les fonctions d'entrée-sortie d'un pilote de terminal classique et, d'autre part, le contrôleur de ce terminal.

Chaque programme d'émulation de terminal est connecté à l'élément maître d'une paire, élément qui joue le rôle du contrôleur de terminal vis à vis du module *tty*. Dans ce schéma, l'appel de la primitive système *read* sur l'élément maître correspond à l'*envoi* de caractères sur l'écran et, inversement, l'appel de la primitive

write sur cet élément correspond à la *réception* de caractères en provenance du clavier.

Les flots d'entrée-sortie standard du *Shell* sont connectés à l'élément esclave de la paire. Sur cet élément, les primitives système *read*, *write* et *ioctl* donnent accès à l'ensemble des fonctions du module *tty* comme pour un terminal classique. Notons que la solution consistant à connecter les deux processus par des tubes est exclue du fait que ce mécanisme n'offre justement aucune compatibilité avec le module *tty* du noyau.

Implanter le gestionnaire de fenêtres dans un serveur offre plusieurs avantages :

- le contrôle de l'accès concurrent aux ressources partagées est réalisé au niveau du serveur qui traite séquentiellement les messages qu'il reçoit des activités et les événements émis par l'utilisateur depuis les dispositifs d'entrée. La synchronisation est en fait assurée implicitement à travers le mécanisme de communication ;

- le développement et la mise au point sont plus faciles puisque [presque] toutes les fonctions du système multi-fenêtres sont implantées en dehors du noyau. Les erreurs éventuelles ne menacent pas l'intégrité du système d'exploitation hôte et l'intégration du système multi-fenêtres peut être menée en parallèle avec d'autres tâches ;

- les applications sont indépendantes des caractéristiques du matériel. Le serveur est le seul programme qui doit être modifié pour adapter le système à un terminal particulier ou exploiter les capacités d'un coprocesseur d'affichage par exemple.

Le problème fondamental avec cette méthode est celui de la performance. Chaque opération graphique effectuée par une activité implique l'envoi d'un message au serveur. Il en est de même pour la transmission des événements. Le serveur est le seul à accéder aux dispositifs d'entrée et chaque caractère et éventuellement chaque déplacement de la souris transitent par le serveur avant d'être reçus par l'activité à laquelle ils sont destinés.

Il s'en suit que les interactions entre l'utilisateur et l'activité courante provoquent de nombreuses commutations de processus qui diminuent l'efficacité

globale du système. Plusieurs techniques permettent de réduire le coût de la surcharge induite par la communication de messages :

- regrouper plusieurs requêtes dans un même message. Cette technique est employée dans Andrew pour les requêtes ne nécessitant aucune réponse du serveur, ce qui est le cas de toutes les opérations d'affichage par exemple ;

- élever le niveau d'abstraction de l'interface du système. Le système SunDew applique ce principe en intégrant un interpréteur du langage PostScript dans le serveur (cf III 3.3.2).

Ces techniques ne sont toutefois applicables que pour la transmission des informations depuis les activités vers le serveur. Les événements émis par l'utilisateur depuis les dispositifs d'entrée doivent par contre être transmis immédiatement à l'activité courante qui est la seule capable de les interpréter. C'est le cas par exemple des menus dynamiques où l'activité doit recevoir tous les déplacements de la souris pour gérer la mise en évidence de l'élément désigné par l'utilisateur. Une solution à ce problème consiste à permettre aux applications de télécharger dans le serveur des programmes interactifs qui se chargent de tout ou partie du dialogue avec l'utilisateur.

Cette technique est utilisée dans le terminal Blit développé chez AT&T [Pike 84b]. Le Blit est un terminal graphique programmable conçu pour fonctionner avec le système Unix. Le système du Blit inclut un noyau de gestion de processus et un gestionnaire de fenêtres. Le Blit est connecté à la machine hôte par une liaison RS-232 et se comporte comme un serveur vis à vis des activités exécutées sur le système Unix de la machine hôte.

Le système Blit associe à chaque fenêtre un processus qui exécute localement un programme téléchargé depuis la machine hôte par l'activité propriétaire de la fenêtre. Le gestionnaire de fenêtres du Blit multiplexe le clavier et le souris entre les programmes téléchargés qui disposent de primitives spéciales pour accéder aux événements d'entrée, afficher dans une fenêtre et envoyer des informations aux activités exécutées sur la machine hôte.

Par exemple, l'éditeur de texte *jim* conçu pour exploiter les capacités du terminal Blit est décomposé en deux parties : un programme exécuté sur Unix qui maintient une copie complète du fichier édité et un programme téléchargé sur le Blit qui gère les commandes d'édition et la mise à jour de la fenêtre.

Le système SunDew applique un principe équivalent à celui du Blit. Le serveur inclut un noyau de gestion de processus - appelés « lightweight processes » - qui exécutent des programmes PostScript envoyés par les activités dans des messages. Le langage PostScript a été étendu pour permettre à ces programmes d'accèder aux événements émis depuis les dispositifs d'entrée. (cet aspect n'est pas pris en compte dans le langage PostScript standard qui, rappelons-le, a été conçu à l'origine comme support de communication avec des imprimantes). Un processus PostScript peut spécifier le type d'événements qu'il attend dans une région décrite par un *patron*, ce qui permet de filtrer les déplacements de la souris à l'intérieur d'un cercle par exemple. Le système SunDew exploite cette technique pour mettre en œuvre les commandes de manipulation de fenêtres. Ces commandes sont gérées par des programmes PostScript exécutés par un « leightweight process » du serveur.

4.2.2. Les programmes d'application

Le code des opérations graphiques peut être placé dans les programmes d'application lorsque le système inclut un mécanisme de mémoire partagée entre processus. C'est le cas par exemple de la version System V d'Unix. Dans ce schéma, la mémoire-écran est partagée entre tous les processus et les opérations graphiques sont exécutées directement dans l'environnement utilisateur des processus. Ce principe est appliqué dans le système W développé sur une machine SM90 fonctionnant sous Unix System V.

Dans le système W, le code des opérations graphiques est dupliqué dans chaque programme d'application et les structures de données décrivant les fenêtres sont également implantées dans une mémoire partagée. Les fonctions de création, de destruction et de manipulation des fenêtres sont par contre centralisées dans un programme serveur indépendant appelé *Wmgr* qui gère la mémoire partagée entre les programmes d'application. Les programmes d'application appellent les fonctions de *Wmgr* par envoi de messages. Le système W s'appuie pour cela sur le mécanisme de communication par messages d'Unix System V. Lors de la création d'une fenêtre, le programme *Wmgr* alloue dans la mémoire partagée :

- un espace de sauvegarde d'une taille égale à celle de la fenêtre ;
- une structure de données qui contient la taille de la fenêtre , sa position sur l'écran et une liste de rectangles décrivant les parties visibles de la fenêtre.

Les programmes d'application ont accès en lecture seulement aux informations contenues dans les structures de données que seul le programme *Wmgr* peut modifier. Par contre, les programmes d'application et le programme *Wmgr* ont tous accès en écriture à la mémoire-écran et aux mémoires de sauvegarde des fenêtres. Le programme *Wmgr* assure également la gestion des polices de caractères qui sont partagées en lecture seulement entre les programmme d'application. Enfin, c'est le programme *Wmgr* qui gère l'interface utilisateur du système en ce qui concerne la gestion des activités et la manipulation des fenêtres.

La gestion des terminaux virtuels est effectuée de la même manière que dans X ou Andrew (cf III.4.2.1). Un pilote de « pseudo-tty » a été pour cela ajouté dans le noyau car ce mécanisme n'était pas inclus dans Unix System V à l'époque où W a été développé. Les programmes d'émulation de terminaux autorisent le transfert de chaînes de caractères selon le principe du « couper-coller ». Ils assurent également de cette façon la gestion de l'« historique » des informations affichées dans les fenêtres. L'historique est un fichier regroupant les commandes entrées par l'utilisateur depuis le lancement du *Shell* et le résultat de ces commandes sur l'écran.

Outre la gestion des « pseudo-tty », le noyau inclut un pilote de contrôle des ressources partagées dans le système. Les fonctions de ce pilote sont les suivantes :

1) gérer le clavier et la souris et transmettre les événements à l'activité courante ;

2) synchroniser les activités et le programme *Wmgr* lorsqu'ils accèdent à la mémoire-écran et aux structures de données décrivant les fenêtres.

Dans W, l'utilisateur sélectionne l'activité courante en sélectionnant une de ses fenêtres avec la souris. Cette opération est effectuée de la manière suivante. A chaque instant, le pilote de contrôle possède une copie de la liste des parties visibles de la dernière fenêtre sélectionnée par l'utilisateur. Lorsque l'utilisateur sélectionne un point de l'écran en dehors de cette fenêtre, le pilote de contrôle transmet cet événement au programme *Wmgr* qui devient automatiquement l'activité courante. *Wmgr* détermine la nouvelle fenêtre sélectionnée et transmet sa liste de parties visibles au pilote de contrôle. L'activité propriétaire de cette fenêtre devient alors l'activité courante.

Contrairement à la solution de type « serveur » précédente, les opérations graphiques et les opérations de manipulation des fenêtres sont exécutées dans des processus différents qui doivent explicitement se synchroniser lorsqu'ils accèdent

aux données partagées. La solution adoptée dans W consiste à faire en sorte qu'aucune commutation de processus ne puisse intervenir durant l'exécution d'une opération graphique ou d'une opération de manipulation de fenêtres. Il s'agit donc de rendre chacune de ces opérations atomique du point de vue du système. Précisons que dans Unix le processus courant ne peut perdre le processeur que dans les deux situations suivantes :

- lorsqu'il exécute dans le noyau une primitive système qui le bloque explicitement, dans le cas d'une opération d'entrée-sortie par exemple ;
- lorsqu'il s'exécute dans son environnement utilisateur et qu'un processus plus prioritaire est réveillé par une routine d'interruption, par exemple sur interruption horloge au bout d'une seconde d'allocation de l'unité centrale.

Les opérations graphiques et les opérations de manipulation de fenêtres n'appelant par définition aucune primitive système, il suffit d'éviter qu'une routine d'interruption ne provoque la commutation du processeur. Le pilote de contrôle introduit pour cela un verrou qui conditionne la commutation de processus et offre dans son interface deux primitives spéciales pour poser et respectivement enlever le verrou de commutation. Chaque opération graphique et chaque opération de manipulation de fenêtre suivent alors le schéma fonctionnel suivant :

- appeler le pilote de contrôle pour poser le verrou de commutation ;
- effectuer l'opération proprement dite ;
- appeler le pilote de contrôle pour enlever le verrou et autoriser à nouveau la commutation de processus ;

Ces deux primitives spéciales servent également à interdire (respectivement autoriser) le suivi de la souris si le réticule de la souris est affiché dans la mémoire-écran. Le pilote de contrôle rompt le verrouillage de la commutation de processus au bout d'un intervalle de temps donné pour éviter qu'une application erronée ne bloque le système. L'architecture du système W est illustrée dans la figure 3.11.

Implanter les opérations graphiques dans les programmes d'application a essentiellement deux avantages :

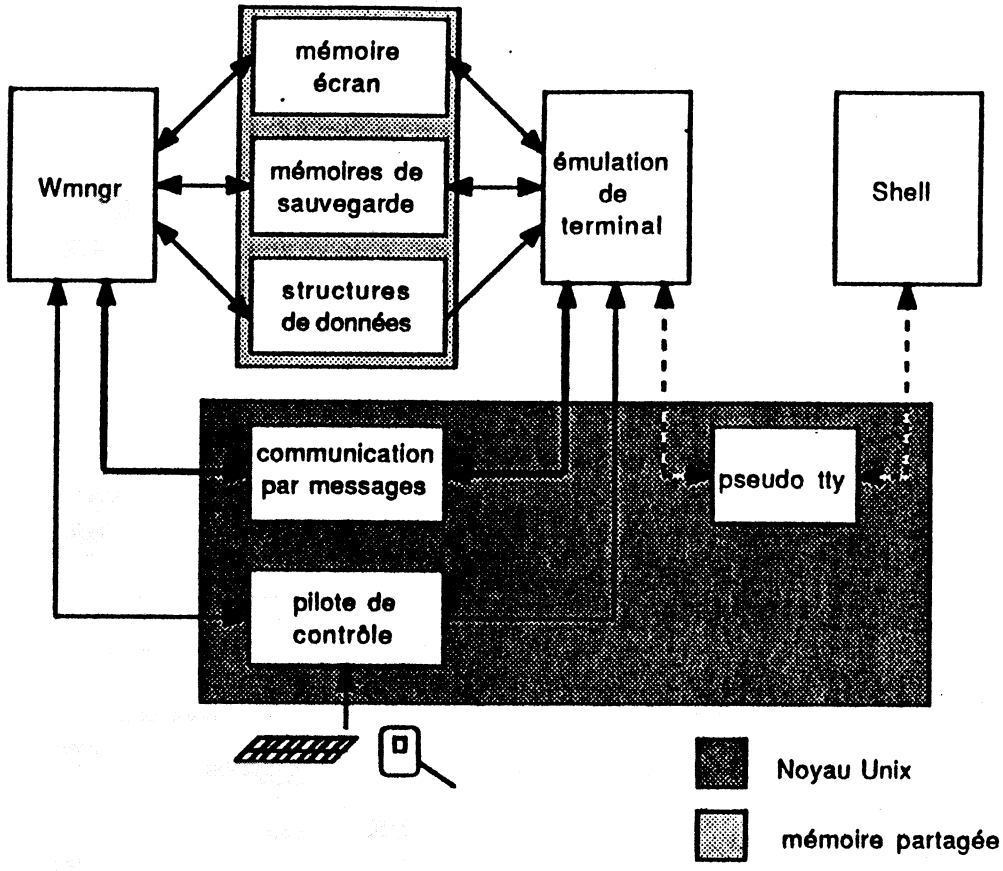


Figure 3.11 Architecture du système W

1) elle permet de ne pas limiter a priori les capacités graphiques du système puisque les applications ont directement accès au contenu de leurs fenêtres. Chaque application peut ainsi créer ses propres opérations graphiques lorsque cela est nécessaire ;

2) elle optimise les performances des opérations graphiques qui sont directement exécutées dans l'environnement utilisateur des processus.

Ces deux avantages ont en contre-partie de nombreux inconvénients. Les applications ne sont plus indépendantes du matériel. Adapter le système à un nouveau terminal graphique nécessite la liaison de chaque application avec une nouvelle bibliothèque graphique. Le principe de l'édition de liens dynamique serait très appréciable dans ce cas.

Cette solution implique également que le code des opérations graphiques est dupliqué dans chaque programme d'application. Une solution à ce problème consiste à modifier le système (et l'éditeur de liens) pour placer la (les) bibliothèque(s) à une adresse prédéfinie en mémoire et à partager leur code entre tous les programmes d'application.

Toutes les primitives graphiques doivent être encadrées par deux appels au système pour assurer l'accès en exclusion mutuelle aux données partagées, ce qui diminue le gain d'efficacité escompté. Notons également qu'une telle approche repose sur le « bon comportement » des activités. Le système n'a en effet aucun moyen d'imposer aux applications de respecter le schéma fonctionnel précédent lorsqu'elles accèdent aux données partagées. Il s'ensuit qu'un programme d'application erroné peut éventuellement détruire l'ensemble de la mémoire-écran et non uniquement le contenu de ses propres fenêtres.

4.2.3. Le noyau du système

Certaines versions d'Unix ne permettent pas d'adopter l'une des deux solutions décrites précédemment. C'est le cas par exemple d'Unix version 7 qui ne permet pas le partage de mémoire entre processus et n'offre aucun mécanisme de communication par messages entre processus. Le noyau est alors le seul endroit où peuvent être gérées les ressources partagées.

Cette contrainte conduit à intégrer dans le noyau l'ensemble des fonctions de base du système multi-fenêtres, à savoir les fonctions de création et de manipulation des fenêtres, les opérations graphiques et la gestion des dispositifs d'entrée. Dans un tel schéma, l'interface utilisateur est la seule fonction du système a être gérée au dessus du noyau dans un programme exécuté par un processus indépendant. Ce principe est appliqué dans le système PNX-WMS développé sur la version 7 d'Unix adaptée au Perq.

Dans PNX-WMS, chaque fenêtre est représentée par un fichier spécial dans le répertoire */dev* du système de fichiers. Ces fichiers donnent accès par l'intermédiaire des primitives d'entrées-sorties standard d'Unix à un pilote du noyau regroupant l'ensemble des fonctions du système de gestion de fenêtres, c'est à dire :

- la création, le destruction et la manipulation des fenêtres. Le système PNX-WMS autorise le recouvrement des fenêtres sur l'écran et gère le contenu des parties cachées de manière transparente aux programmes d'appli-

cation. Il maintient pour cela une copie complète du contenu des fenêtres dans une mémoire de sauvegarde associée à chacune d'elles ;

- l'ensemble des opérations graphiques supportées par le système ;

- les fonctions d'accès aux événements émis par l'utilisateur depuis les dispositifs d'entrée du terminal graphique du Perq

La gestion des terminaux virtuels est également intégrée dans le noyau. Chaque fenêtre est en fait gérée par défaut comme un terminal virtuel et supporte la compatibilité avec le module *tty* du noyau au niveau des primitives système *read*, *write* et *ioctl*.

L'interface utilisateur du système est implantée dans un programme appelé *winit* exécuté par un processus indépendant. Le programme *winit* permet à l'utilisateur de créer de nouvelles fenêtres dans lesquelles sont automatiquement démarrées un *Shell* et de manipuler les fenêtres sur l'écran. Les commandes de manipulation ne sont pas implantées dans le bord des fenêtres comme c'est généralement le cas dans la majorité des systèmes multi-fenêtres. Le bord des fenêtres est en fait entièrement géré par le pilote de gestion des fenêtres et sert uniquement à distinguer les fenêtres les unes des autres et à mettre en évidence la fenêtre courante. L'aspect du bord des fenêtres est donc figé dans le noyau d'Unix et ne peut être modifié pour être adapté aux goûts de chaque utilisateur du système. Le programme *winit* offre à l'utilisateur le choix entre les deux possibilités suivantes pour exprimer une commande :

- sélectionner la fenêtre « système » créée par *winit* et épeler la commande au clavier ;

- sélectionner la commande dans un menu dynamique qui apparaît lorsque l'utilisateur sélectionne un point de l'écran situé en dehors de toute fenêtre.

L'utilisateur doit ensuite exprimer avec la souris les paramètres éventuels de la commande, par exemple la fenêtre concernée dans le cas d'une commande de manipulation. Rappelons que les programmes d'application ne peuvent pas manipuler les fenêtres sur l'écran (cf III.3.2.3). Le programme *winit* est le seul à pouvoir appeler les fonctions de manipulation de fenêtres implantées dans le pilote du noyau. La figure 3.12 ci-dessous illustre l'architecture du système PNX-WMS.

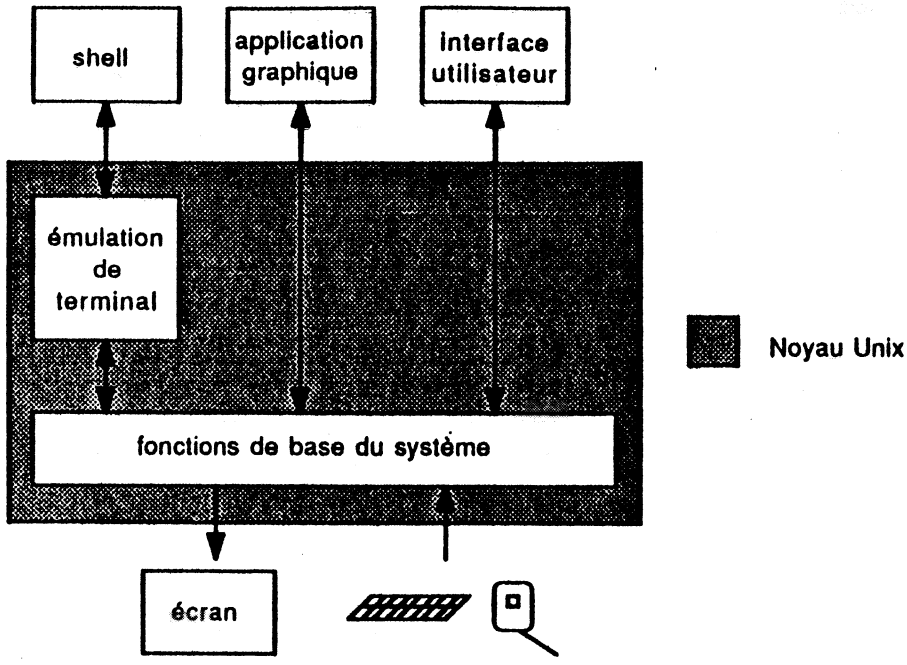


Figure 3.12 Architecture du système PNX-WMS

Analysons les avantages et inconvénients d'une telle approche. Le noyau d'Unix est tout d'abord un endroit privilégié pour synchroniser les activités lorsqu'elles accèdent aux ressources partagées du système. Les opérations graphiques et les opérations de manipulation de fenêtres s'exécutent par définition en exclusion mutuelle puisqu'un processus n'est jamais commuté tant qu'il exécute une primitive système non bloquante (cf III.4.2.2). Le pilote de gestion des fenêtres doit uniquement inclure un mécanisme d'exclusion mutuelle entre ces opérations et les routines d'interruption associées aux dispositifs d'entrée, notamment avec la routine *Suivre_Souris* si le réticule de la souris est affiché dans la mémoire-écran.

Le noyau garantit également l'indépendance des programmes d'application vis à vis des caractéristiques du matériel. Le pilote du noyau est le seul programme qui doit être modifié pour adapter le système à un terminal graphique particulier ou pour exploiter les capacités graphiques d'un co-processeur d'affichage.

Soulignons ici que ces deux points - synchronisation implicite des opérations et indépendance des applications par rapport au matériel - font également partie des avantages de la solution de type « serveur » présentée auparavant. Cette analogie est due au fait que dans les deux cas la gestion de l'ensemble des ressources

partagées est centralisée dans un seul programme indépendant - en termes d'espace d'adressage et d'espace d'exécution - des programmes d'application.

Implanter le gestionnaire de fenêtres dans le noyau évite par contre les problèmes de performances propres à la solution de type « serveur ». Chaque opération graphique effectuée par une activité est directement exécutée dans le noyau et n'implique ni transmission de messages ni commutation de processus. Cette remarque est également valable pour les événements qui sont directement transmis aux activités depuis le pilote de gestion des dispositifs d'entrée.

Cette solution présente en contre-partie plusieurs inconvénients. Le développement du gestionnaire de fenêtres est tout d'abord plus difficile. La mise au point des programmes intégrés au noyau ne peut être effectuée à l'aide des outils interactifs classiques et chaque erreur non détectée menace potentiellement l'intégrité de l'ensemble du système. Il s'ensuit que la mise au point du système multi-fenêtres ne peut être menée en parallèle avec d'autres tâches.

Ce type de solution limite d'autre part les capacités graphiques du système. Il n'est en effet pas réaliste d'intégrer dans le noyau un interpréteur du langage PostScript (ou de tout autre langage graphique de ce niveau) et les applications doivent en général se contenter d'un ensemble relativement pauvre de primitives graphiques. Une solution à ce problème consiste à permettre le transfert dans une fenêtre de matrices de points situées dans l'environnement utilisateur des processus. Cette technique, adoptée dans PNX-WMS, diminue les performances du système et rend les programmes d'application dépendants du matériel (à moins que le système ne définisse un protocole de transfert de matrices de points virtuelles, aspect qui n'est pas abordé dans le manuel de description du système PNX-WMS).

Le système PNX-WMS ne permet pas le transfert de chaînes de caractères selon le principe du « couper-coller ». Ceci est dû au fait que la gestion des terminaux virtuels est directement intégrée dans le noyau et qu'il ne peut être envisagé de placer dans le noyau des aspects relevant de l'interface utilisateur.

Chapitre IV

LE SYSTEME FENIX

1. Introduction

Le système Fenix a été développé dans le cadre du projet TIGRE (Traitement d'Informations Généralisées REparti) mené en coopération par le Laboratoire de Génie Informatique et le Centre de Recherches Bull de Grenoble depuis 1982. Ce projet comprend deux aspects [Tigre 83] :

1) un serveur de Base de Données de Documents. La notion de Document est générale et dépend des applications utilisant la Base de Données (bureautique, développement de logiciel, conception assistée par ordinateur, etc.) ;

2) la réalisation d'outils pour l'acquisition et la manipulation interactive des Documents sur des postes de travail individuels connectés au serveur à travers un réseau local.

Les postes de travail utilisés pour le prototype sont des machines SPS7 de Bull fonctionnant sous le système d'exploitation SMX4.1 développé à l'INRIA. Ce système est une adaptation de la version 7 d'Unix à l'architecture multi-processeurs (Motorola 68000) du SPS7. Chaque poste de travail comprend un terminal graphique Numelec doté d'un écran à points monochrome (1024 * 780 de définition) et d'une souris.

Notre participation dans ce projet a consisté à développer pour les postes de travail individuels un système multi-fenêtres qui offre les mécanismes de base

nécessaires à la conception d'outils de communication évolués. Les principaux objectifs que nous nous sommes fixés sont les suivants.

Nous voulons tout d'abord que le système permette à l'utilisateur de mener plusieurs activités en parallèle et que chaque activité puisse disposer d'un nombre quelconque de fenêtres pour communiquer avec l'utilisateur. Le système doit autoriser le recouvrement des fenêtres et le gérer de façon transparente aux programmes d'application (cf III.3.2.1.2).

Nous désirons par ailleurs ne pas figer l'interface de commandes du système et laisser toute liberté aux utilisateurs d'adapter cette interface au style de communication qui leur convient. Chaque utilisateur doit par exemple pouvoir choisir la manière dont sont disposées les commandes de manipulation dans le bord des fenêtres ou redéfinir les fonctions associées aux boutons de la souris par l'interface du système.

Enfin, nous voulons que le système soit portable sur les différentes versions d'Unix commercialisées actuellement et qu'il puisse être facilement adapté à n'importe quel type de terminal.

Nous présentons dans le paragraphe suivant l'architecture du système et les choix que nous avons faits pour satisfaire dans la mesure du possible aux différents objectifs énumérés ci-dessus. Nous décrivons ensuite de façon détaillée les principaux composants du système.

2. Architecture du système

L'architecture du système Fenix résulte d'un compromis entre, d'une part la recherche de performances satisfaisantes au niveau des opérations d'entrée-sortie et, d'autre part, le souci de faciliter l'adaptation et l'évolution de l'interface utilisateur du système.

L'interface utilisateur du système est gérée par un programme nommé *Fenix_Int* qui est exécuté au dessus du noyau par un processus ordinaire. L'ensemble des fonctions de base du système (manipulation des fenêtres, primitives graphiques, transmission des entrées, etc.) est par contre intégré dans le noyau d'Unix. Ce schéma suit en fait dans son principe la démarche adoptée dans le

système Unix standard où l'interpréteur de commandes (le *Shell*) est exécuté au dessus du noyau et peut être adapté à chaque utilisateur.

Les programmes d'application opèrent à travers une bibliothèque pour communiquer avec l'utilisateur. Selon la nature de l'opération, l'appel d'une fonction de la bibliothèque se traduit, soit par l'envoi d'une requête à *Fenix_Int*, soit directement par l'appel d'une fonction de base. Le choix entre les deux techniques résulte là encore du même compromis entre performances et souplesse.

Les fonctions de base sont directement appelées lorsqu'il s'agit d'opérations d'entrée-sortie, c'est à dire pour afficher dans une fenêtre et pour accéder aux événements émis par l'utilisateur depuis les dispositifs d'entrée.

Une requête est par contre envoyée à *Fenix_Int* par toutes les fonctions de manipulation de fenêtres (création et destruction comprises). C'est *Fenix_Int* qui réalise l'opération demandée en appelant la fonction de base correspondante, après avoir éventuellement modifié les paramètres transmis par l'activité.

Cette technique permet à *Fenix_Int* de contrôler et de connaître en permanence l'état des fenêtres et leur disposition relative sur l'écran. Il peut ainsi associer à chaque opération de manipulation un ensemble d'actions appropriées, par exemple afficher une icône sur l'écran lorsqu'une fenêtre est cachée. C'est d'autre part *Fenix_Int* qui se charge du dialogue avec l'utilisateur lorsqu'une activité demande à ce que celui-ci choisisse la position et/ou la taille d'une fenêtre au moment de sa création par exemple (cf III.3.2.2).

L'ensemble des fonctions de base est décomposé en trois modules, le *gestionnaire de fenêtres*, le *gestionnaire d'entrées* et le *gestionnaire du terminal graphique*.

Le *gestionnaire de fenêtres* gère l'espace d'affichage des fenêtres et met en œuvre les fonctions de manipulation de fenêtres ainsi que l'ensemble des primitives graphiques. Nous avons également implanté dans ce module la primitive appelée dans la bibliothèque standard de Fenix pour transmettre les requêtes de manipulation de fenêtres à *Fenix_Int*. Cela évite aux activités d'avoir à établir une voie de communication particulière avec *Fenix_Int* et permet de rendre le protocole de communication entièrement transparent aux programmes d'application.

Le *gestionnaire d'entrées* multiplexe les dispositifs d'entrée entre les activités et leur transmet les événements d'entrée. Nous avons retenu la méthode décrite en III.3.4.2.1 qui consiste à associer à chaque activité une file d'entrée unique dans laquelle lui sont transmis tous les événements qui lui sont destinés.

Le *gestionnaire du terminal graphique* regroupe les fonctions dépendantes du terminal, par exemple la fonction de suivi de la souris sur l'écran et les routines d'interruption associées aux dispositifs d'entrée. Il est également chargé de transmettre à *Fenix_Int* les requêtes envoyées par les activités et, inversement, de transmettre aux activités les messages d'acquittement émis par *Fenix_Int*.

Nous avons d'autre part été obligés d'intégrer la gestion des terminaux virtuels dans le noyau car le système SMX4.1 n'inclut pas de mécanisme de communication approprié, les « pseudo-ttys » d'Unix BSD4.2, permettant d'implanter cette fonction au dessus du noyau (cf III.4.2.1).

L'architecture du système Fenix est résumée dans la figure 4.1 ci-dessous.

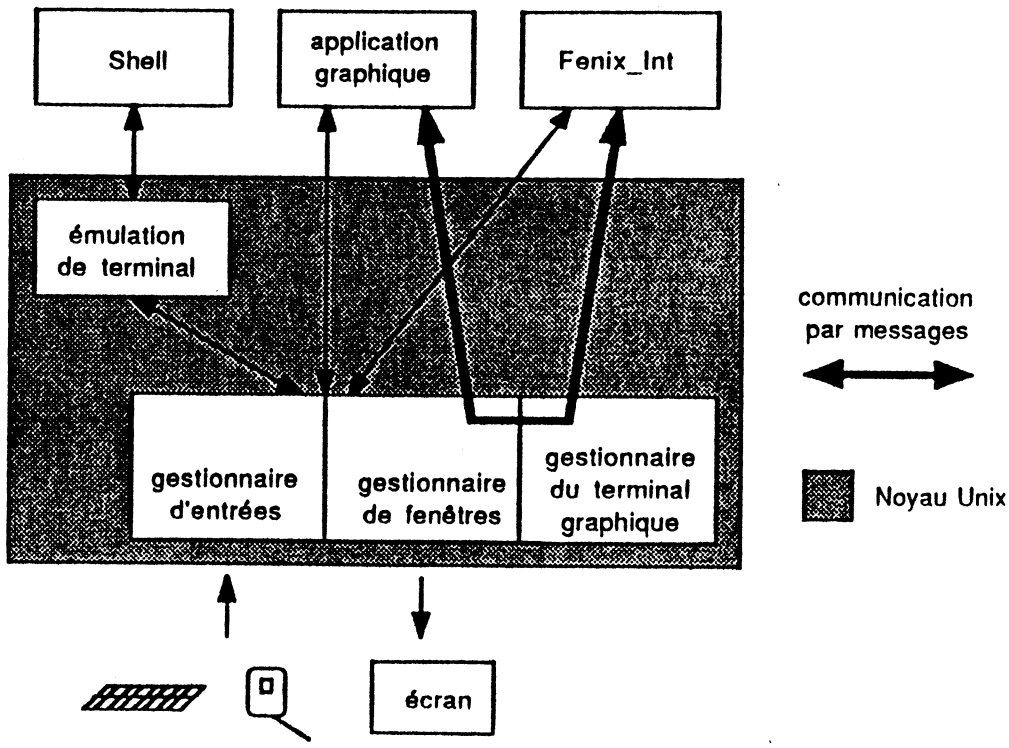


Figure 4.1 Architecture du système Fenix

Toutes les fonctions de base sont implantées dans des pilotes (« drivers ») et l'accès à ces fonctions est réalisé à l'aide des primitives d'entrée-sortie standard d'Unix. Nous n'avons introduit aucun appel système supplémentaire dans le noyau afin que le système Fenix puisse être intégré sans aucun changement à toutes les versions d'Unix.

Le fonctionnement des différents modules intégrés dans le noyau et du programme *Fenix_Int* sont décrits de façon détaillée dans les paragraphes suivants.

3. Le gestionnaire de fenêtres

Le gestionnaire de fenêtres gère un seul niveau d'espace d'affichage par fenêtre (cf III.3.2.1.1) et réalise les fonctions de base opérant sur les fenêtres, c'est à dire les primitives de création, de destruction et de manipulation de fenêtres et l'ensemble des primitives graphiques de base.

Pour gérer le recouvrement des fenêtres de façon transparente aux programmes d'application, il doit conserver en mémoire le contenu des parties cachées de toutes les fenêtres [en partie] recouvertes. Il existe deux solutions à ce problème :

1) ne conserver effectivement en mémoire hors-écran que le contenu des parties cachées des fenêtres. Cette solution est adoptée dans les systèmes Sapphire [Myers 85], WHIM [Goodfellow 85] et dans le terminal Blit [Pike 83] ;

2) associer à chaque fenêtre une mémoire de sauvegarde complète. C'est l'approche suivie dans le système multi-fenêtres de la LISP Machine [Weinreb 81].

La première solution présente a priori deux avantages : elle minimise l'espace mémoire nécessaire à la sauvegarde des parties cachées et elle optimise le temps d'exécution des primitives graphiques dans les fenêtres entièrement visibles sur l'écran.

En fait le gain de mémoire est au plus égal à la taille de la mémoire-écran et est relativement négligeable dans le cas d'écrans monochromes (96 Koctets pour le terminal Numelec). Cette solution complique par contre la mise en œuvre des opérations de manipulation qui toutes - exceptées *monter* et *descendre* une fenêtre -

impliquent en général l'allocation et/ou la libération de mémoire dans l'espace de sauvegarde [Pike 83].

D'autre part, elle pénalise les performances de ces opérations qui doivent systématiquement effectuer deux transferts d'informations : le premier pour sauvegarder les parties visibles des fenêtres qui vont être recouvertes et le second pour copier dans la mémoire-écran ce qui doit y être affiché.

C'est pour éviter ces inconvénients que nous avons choisi la seconde méthode, c'est à dire allouer à chaque fenêtre une mémoire de sauvegarde d'une taille équivalente à celle du rectangle la délimitant sur l'écran. Les seules opérations à soumettre des requêtes d'allocation et/ou de libération de mémoire dans l'espace global sont alors *créer*, *détruire* et *changer la taille* d'une fenêtre. En règle générale, ces trois opérations sont exécutées beaucoup moins souvent que les autres opérations de manipulation. Par exemple, *montrer* et *cacher* une fenêtre sont exécutées fréquemment si l'interface utilisateur du système et des programmes d'application est basée sur l'emploi d'objets dynamiques tels que les menus et les formulaires (cf III.3.2.3).

Enfin, les opérations de manipulation sont plus efficaces à l'exécution car elles n'ont pas à sauvegarder préalablement le contenu des parties visibles des fenêtres qui vont être recouvertes.

Chaque point d'une mémoire de sauvegarde est représenté par une valeur qui dépend du type de terminal géré par le système. Sur un terminal graphique, chaque point est représenté par un ou plusieurs bits selon que l'écran du terminal est monochrome ou polychrome. Sur un terminal alphanumérique classique, chaque point est représenté par un octet contenant le code du caractère.

Nous avons développé le gestionnaire de fenêtres en cherchant à isoler au maximum cette partie dépendante du matériel. Ce principe s'applique notamment à la gestion du recouvrement qui est mise en œuvre dans un module interne appelé *gestionnaire de cadres* conçu de façon à pouvoir fonctionner sur n'importe quel type de terminal.

3.1. Le gestionnaire de cadres

La gestionnaire de cadres n'accède pas directement aux mémoires de sauvegarde des fenêtres ni à la mémoire-écran mais opère sur des entités appelées *cadres*. Un *cadre* représente le rectangle délimitant sur l'écran la fenêtre à laquelle il est associé et l'ensemble des cadres reflète la disposition respective des fenêtres sur l'écran selon leur ordre de recouvrement.

Le gestionnaire de cadres est implanté comme une couche intermédiaire munie d'une interface de haut niveau à destination des primitives du gestionnaire de fenêtres et d'une interface de bas niveau spécifique au terminal. L'interface de haut niveau inclut les opérations suivantes :

- *Init_Ecran* : initialiser le cadre associé à l'écran. Cette procédure est appelée à l'initialisation du système ;
- *Init_Cadre* : initialiser le cadre d'une fenêtre. Cette procédure est appelée à la création d'une fenêtre ;
- *Changer_Cadre* : changer simultanément l'origine, la taille et le rang d'un cadre ;
- *Cacher_Cadre* : cacher un cadre ;
- *Montrer_Cadre* : montrer un cadre caché ;
- *Monter_Cadre* : monter un cadre au sommet de la « pile » de cadres ;
- *Descendre_Cadre* : descendre un cadre au bas de la « pile » de cadres ;
- *Maj_Cadre* : mettre à jour sur l'écran les parties visibles d'un rectangle quelconque inclus dans un cadre ;
- *Désigner_Cadre* : retourner le cadre visible à un point (x, y) de l'écran.

L'appel de l'une quelconque de ces opérations (exceptées *Init_Ecran*, *Init_Cadre* et *Désigner_Cadre*) résulte en une séquence d'appels de l'une et/ou l'autre des deux opérations suivantes de l'interface de bas niveau :

- *Rafraîchir_Ecran(c, r)* qui rafraîchit le rectangle *r* de l'écran à partir de la mémoire de sauvegarde de la fenêtre délimitée par le cadre *c* ;

• *Effacer_Ecran(r)* qui efface le rectangle *r* de l'écran.

Les opérations de manipulation d'un cadre sont appelées par les primitives correspondantes du gestionnaire de fenêtres. Par exemple, la primitive *Changer_Taille* chargée de changer la taille d'une fenêtre alloue et initialise la mémoire de sauvegarde puis appelle ensuite *Changer_Cadre* pour mettre à jour son contenu sur l'écran (cf IV.3.2 ci-dessous). Toutes les primitives graphiques du gestionnaire de fenêtres suivent la démarche suivante pour afficher une information dans une fenêtre :

1) afficher l'information dans la mémoire de sauvegarde de la fenêtre et calculer le rectangle englobant cette information. Le rectangle englobant est le plus petit rectangle à l'extérieur duquel aucun point n'a été modifié par l'opération graphique ;

2) appeler la procédure *Maj_Cadre* du gestionnaire de cadres pour mettre à jour sur l'écran le contenu du rectangle englobant.

Le fonctionnement du gestionnaire de cadres est illustré dans la figure 4.2 ci-dessous.

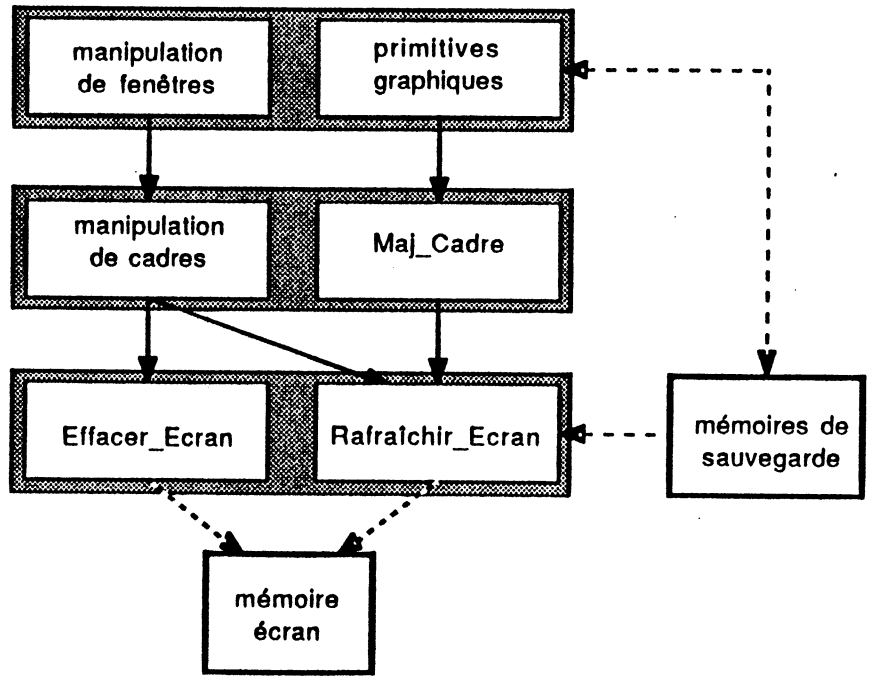


Figure 4.2 Fonctionnement du gestionnaire de cadres

Le rang d'un cadre désigne la position du cadre dans la « pile ». Le système dans sa version actuelle ne supporte que deux valeurs pour le rang correpondant respectivement au *bas* et au *sommet* de la pile. Le rang d'un cadre caché n'est pas mémorisé et doit être systématiquement passé en paramètre à la procédure *Montrer_Cadre*.

L'ensemble des cadres est géré sous forme d'une liste doublement chaînée. Le cadre *c_écran* de la fenêtre associée à l'écran représente l'élément initial de la liste de cadres (voir figure 4.3 ci-dessous). Seuls les cadres *montrés* sont placés dans cette liste selon l'ordre de recouvrement des cadres depuis le bas jusqu'au sommet de la « pile » de cadres. Il n'est pas nécessaire en effet de gérer une liste des cadres *cachés* car ceux-ci n'interviennent aucunement au niveau de la gestion du recouvrement.

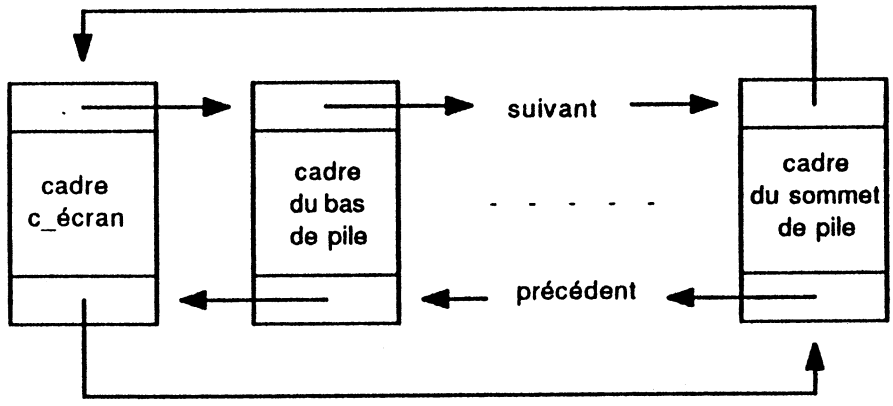


Figure 4.3 Liste des cadres

Chaque procédure de manipulation du gestionnaire de cadres doit réévaluer la liste de parties visibles de tous les cadres concernés par l'opération. La liste de parties visibles d'un cadre est exploitée par la procédure *Maj_Cadre* pour déterminer les sous-ensembles du rectangle englobant qui doivent être mis à jour sur l'écran.

Dans un premier temps, nous avons choisi de conserver en mémoire les listes de parties visibles de tous les cadres afin de réduire le temps d'exécution des opérations d'affichage dans une fenêtre [Boule 85]. Le gain est obtenu au niveau de la procédure *Maj_Cadre* appelée par chacune de ces opérations.

Cette technique présente par contre deux inconvénients majeurs. Elle complique tout d'abord la mise en œuvre des opérations de manipulation qui doivent systématiquement réévaluer les listes de parties visibles de tous les cadres concernés par une manipulation. Elle oblige d'autre part le système à gérer un « tas » de parties visibles partagé entre tous les cadres. Cette gestion est coûteuse en mémoire, d'autant plus qu'il est difficile de déterminer a priori la taille maximum du tas permettant de représenter à un instant donné toutes les parties visibles de l'ensemble des cadres. Ce nombre est en effet fonction du nombre de fenêtres visibles et de leur disposition respective sur l'écran.

Pour éliminer ces deux inconvénients, nous avons développé une seconde version du gestionnaire de cadres dans laquelle la procédure *Maj_Cadre* calcule dynamiquement la liste des parties visibles du cadre sur lequel elle doit opérer. Avec ce principe, les listes de parties visibles de l'ensemble des cadres ne sont plus conservées statiquement et seule la liste du cadre sur lequel opère la procédure *Maj_Cadre* est représentée en mémoire à un instant donné.

Qui plus est, le coût du calcul de cette liste est très faible pour la ou les fenêtres de l'activité courante car ces fenêtres sont la plupart du temps complètement visibles sur l'écran. Leur liste de parties visibles ne contient alors qu'un seul élément. Enfin, des mesures ont montré que le temps de calcul de ces listes est relativement négligeable par rapport au temps d'exécution des « Raster_Op ».

Principe de calcul des parties visibles d'un cadre

Le calcul des parties visibles d'un cadre est basé sur la propriété géométrique suivante :

> étant donnés deux rectangles R et R' disjoints ou non, la différence $(R - R')$ entre ces deux rectangles peut être représentée par un ensemble noté $PV(R, R')$ d'au plus 4 rectangles.

La différence $(R - R')$ est interprétée dans notre contexte comme l'ensemble éventuellement vide des parties visibles du rectangle R recouvert par le rectangle R'. La figure 4.4 ci-dessous résume tous les cas possibles de recouvrement entre deux rectangles. Les rectangles hachurés désignent les parties du rectangle R non recouvertes par le rectangle R'.

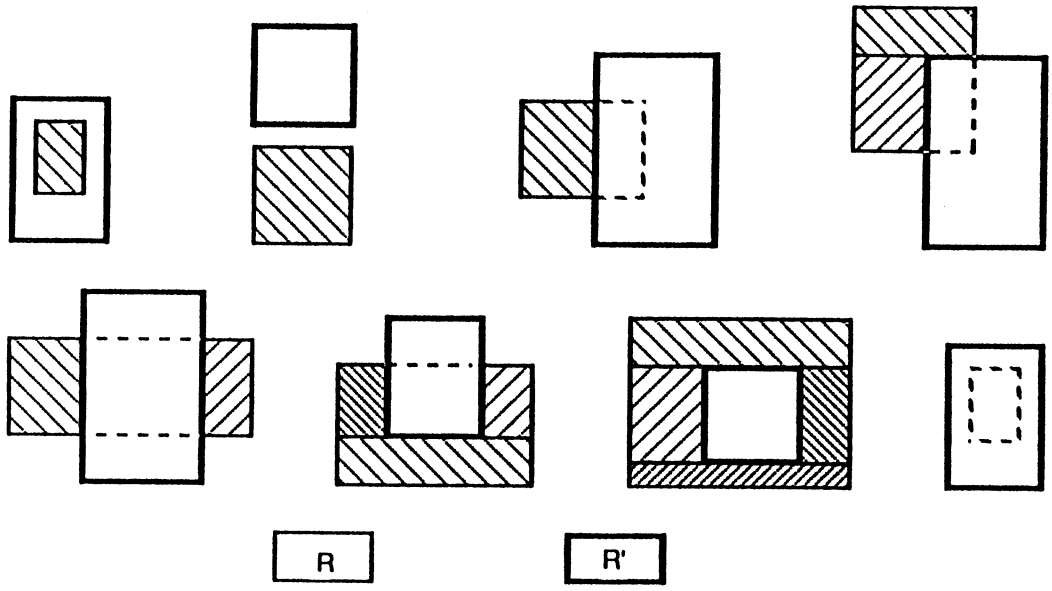


Figure 4.4 Différence de deux rectangles

Remarque : la différence de deux rectangles peut être représentée de deux manières selon que l'on adopte une méthode de décomposition « horizontale » ou « verticale ». Nous avons choisi dans la figure 4.4 la méthode « horizontale » qui est celle adoptée dans Fenix.

Le résultat suivant découle immédiatement de la propriété précédente :

l'ensemble noté *PV(R, R', R")* des parties visibles d'un rectangle *R* recouvert par deux rectangles *R'* et *R"* est l'union de tous les ensembles *PV(r, R")* pour chaque *r* appartenant à *PV(R, R')*.

En d'autres termes, l'ensemble *PV(R, R', R")* est l'ensemble des feuilles d'un arbre « quaternaire » de rectangles, arbre dont *R* est la racine et *PV(R, R')* l'ensemble des nœuds intermédiaires. Cette règle de décomposition est par construction récursive et sa généralisation au cas de *n* rectangles s'exprime alors comme suit :

l'ensemble des parties visibles d'un rectangle *R* recouvert par une suite de *n* rectangles est l'ensemble des feuilles d'un arbre de racine *R*, arbre dont l'ensemble des nœuds de niveau *i* représente les parties de *R* non recouvertes par les *i* premiers rectangles de la suite.

L'exemple de la figure 4.5 illustre le principe de décomposition récursive d'un rectangle. Dans cet exemple, le rectangle R est recouvert par 3 rectangles notés respectivement A, B et C.

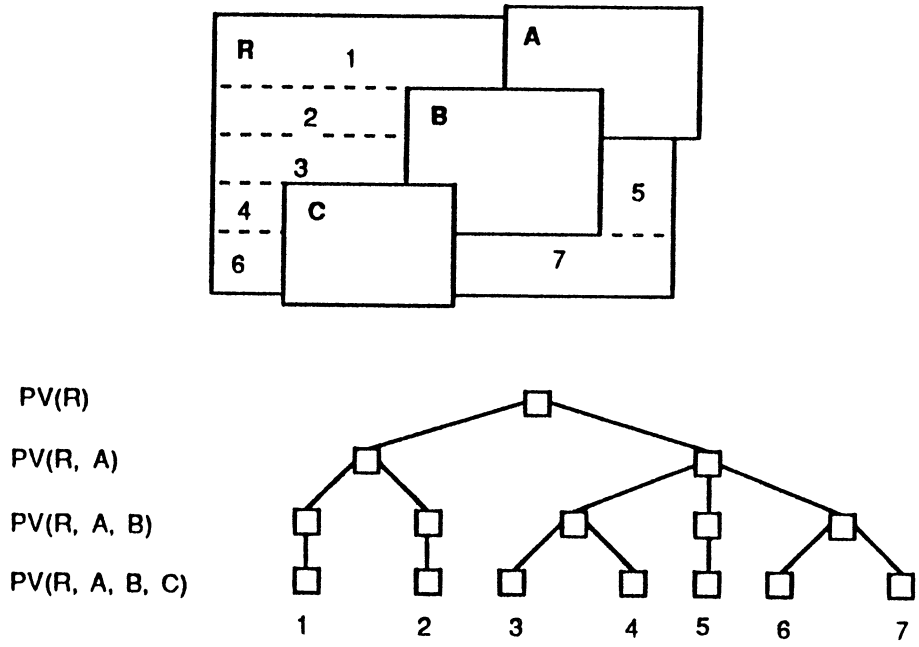


Figure 4.5 Arbre des parties visibles d'un rectangle

Déterminer les parties visibles d'un cadre en fonction de ceux qui le recouvrent sur l'écran consiste alors à parcourir l'arbre de décomposition correspondant et à « produire » les feuilles de cet arbre. Ce parcours est effectué par la procédure récursive *Parties_Visibles* qui, étant donné un rectangle r inclus dans un cadre c, produit l'ensemble des parties de r non recouvertes par le cadre c' et ses suivants dans la liste de cadres :

```
procédure Parties_Visibles(c, r, c')
début
    si r non vide alors
        si c' = c_écran alors
            /* fin de la liste de cadres : r est complètement visible sur l'écran */
            Produire(c, r) ;
        sinon
            /* continuer la décomposition de r en fonction de c' et de ses suivants dans la
            liste de cadres */
            Dr ← Différence(r, R(c'))
            /* la procédure Différence calcule l'ensemble Dr éventuellement vide des parties
            de r non recouvertes par le rectangle R(c') délimitant le cadre c' */
            pour chaque rectangle r' de Dr faire
                Parties_Visibles(c, r', Suivant(c'))
            fin
        finsi
    finsi
fin
```

La procédure *Produire* désigne l'une ou l'autre des deux procédures *Effacer_Ecran* et *Rafraîchir_Ecran* de l'interface de bas niveau selon que le rectangle visible est inclus dans le cadre associé à l'écran ou dans un cadre délimitant une fenêtre. Cette distinction est transparente à la procédure *Maj_Cadre* qui appelle indirectement une procédure dont l'adresse est mémorisée dans chaque *cadre*. Cette adresse est un paramètre des procédures *Init_Cadre* et *Init_Ecran*.

L'algorithme de la procédure *Maj_Cadre* s'écrit comme suit :

```
procédure Maj_Cadre(c, r)
début
    /* L'origine du rectangle r passé en paramètre doit être translatée car le gestionnaire de
    cadres travaille en coordonnées absolues alors que les primitives graphiques du gestionnaire
    de fenêtres travaillent en coordonnées relatives à l'origine du coin supérieur gauche des
    fenêtres */
    r' ← r + Origine(c) ;
    Parties_Visibles(c, r', Suivant(c)) ;
fin
```

La procédure *Maj_Cadre* sert d'autre part à mettre en œuvre dans le gestionnaire du terminal graphique la primitive *Peindre_Ecran* appelée par *Fenix_Int* pour changer la couleur de fond de l'écran (cf IV.7 ci-dessous). La réalisation de *Peindre_Ecran* est triviale : modifier l'attribut *couleur_fond* de la fenêtre de l'écran puis appeler *Maj_Cadre* avec comme paramètres d'appel le cadre *c_écran* et le rectangle *R(c_écran)* délimitant la surface de l'écran.

Réalisation des opérations de manipulation

Rappelons tout d'abord que ces opérations n'ont pas à construire les listes de parties visibles des cadres puisque ces listes sont calculées dynamiquement par la procédure *Maj_Cadre*. Leur rôle consiste uniquement à déterminer les cadres concernés par la manipulation effectuée et à calculer les parties de ces cadres qui doivent être mises à jour dans la mémoire-écran.

Chacune de ces opérations est conçue de façon à respecter la contrainte suivante : ne mettre à jour dans la mémoire-écran que les points effectivement concernés par l'opération et ne modifier chacun d'eux qu'une seule fois. Le but recherché est double :

- optimiser les performances de ces opérations en minimisant le nombre de points modifiés ;
- éviter à l'utilisateur des effets visuels inutiles et désagréables (cf III.2.1.1).

Nous avons choisi de présenter uniquement les opérations les plus significatives de la démarche suivie pour gérer le recouvrement. Cet ensemble comprend, par ordre de complexité croissante, les procédures suivantes :

1) *Descendre_Cadre* ;

2) *Cacher_Cadre* ;

3) *Changer_Cadre*.

1) Réalisation de *Descendre_Cadre*

Le traitement à effectuer sur l'écran pour *descendre* un cadre c en bas de « pile » consiste à rafraîchir sur l'écran les parties des cadres recouvertes uniquement par le cadre c. Les cadres concernés sont ceux qui précèdent c dans la liste (*c_écran* non compris) et qui intersectent avec c. L'exemple de la figure 4.6 ci-dessous illustre le traitement effectué sur l'écran lors de la *descente* d'un cadre.

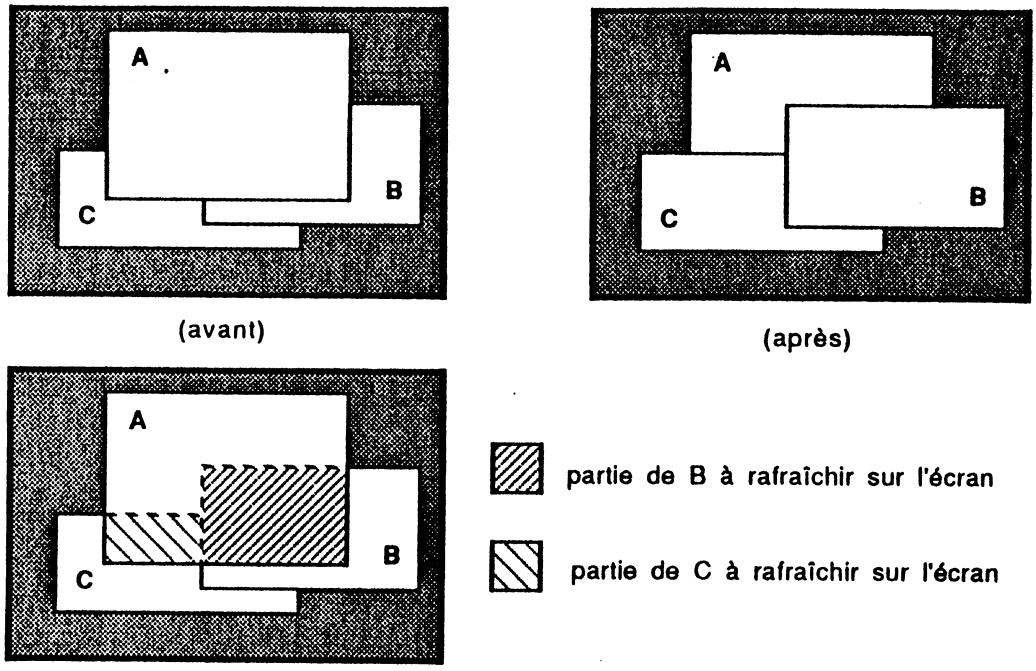


Figure 4.6 Descente d'un cadre sur l'écran

La procédure *Descendre_Cadre* exécute l'algorithme suivant :

```
procédure Descendre_Cadre(c)
début
    c' ← Précédent(c) ;
    /* c' est le premier cadre éventuellement concerné par l'opération */
    Retirer(c) ;
    tantque c' ≠ c_écran faire
        r ← Intersection(R(c), R(c')) ;
        Parties_Visibles(c', r, Suivant(c')) ;
        /* Si r est vide, alors c' n'était pas recouvert par c et n'est pas concerné par l'opération.
        Ce cas est implicitement traité par le test au début de Parties_Visibles. */
        c' ← Précédent(c') ;
    fin
    Insérer(c, BAS_PILE) ;
    /* le cadre c est inséré après le cadre c_écran */
fin
```

2) Réalisation de *Cacher_Cadre*

Le traitement à effectuer sur l'écran pour *cacher* un cadre *c* consiste à *descendre* le cadre *c* puis à « effacer » les parties visibles de *c* en les repeignant à la couleur de fond de l'écran. Cette dernière opération est réalisée en demandant la mise à jour des « parties visibles » du cadre *c_écran* incluses dans le rectangle *R(c)*, une fois le cadre *c* retiré de la liste. La figure 4.7 décrit la démarche suivie pour *cacher* un cadre.

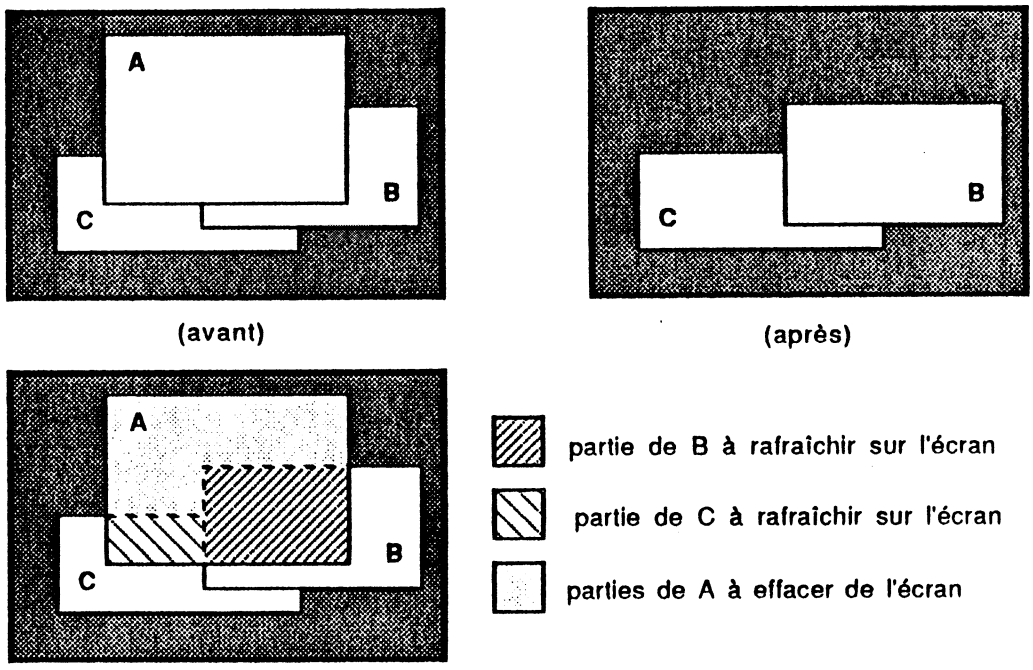


Figure 4.7 Effacement d'un cadre de l'écran

L'algorithme de la procédure *Cacher_Cadre* est le suivant :

```
procédure Cacher_Cadre(c)
début
    Descendre_Cadre(c) ;
    Retirer(c) ;
    /* effacer de l'écran les parties visibles de c */
    Parties_Visibles(c_écran, R(c), Suivant(c_écran)) ;
fin
```

3) Réalisation de *Changer_Cadre*

La procédure *Changer_Cadre* est la plus importante du module de gestion du recouvrement. Elle permet de changer simultanément la position, la taille et le rang d'un cadre tout en respectant la contrainte de base, à savoir ne modifier sur l'écran que le strict nécessaire.

La démarche adoptée pour déplacer et/ou changer la taille d'un cadre *c* consiste à distinguer temporairement deux cadres *c_ancien* et *c_nouveau* représentant le cadre *c* avant et respectivement après modification de sa position et/ou de sa taille. Le traitement à effectuer sur l'écran est alors le suivant :

1) rafraîchir sur l'écran les parties des cadres qui étaient recouvertes uniquement par *c_ancien* et qui ne seront pas recouvertes par *c_nouveau*. La seconde condition intervient uniquement lorsque le cadre *c* doit être placé au sommet de la « pile » de cadres.

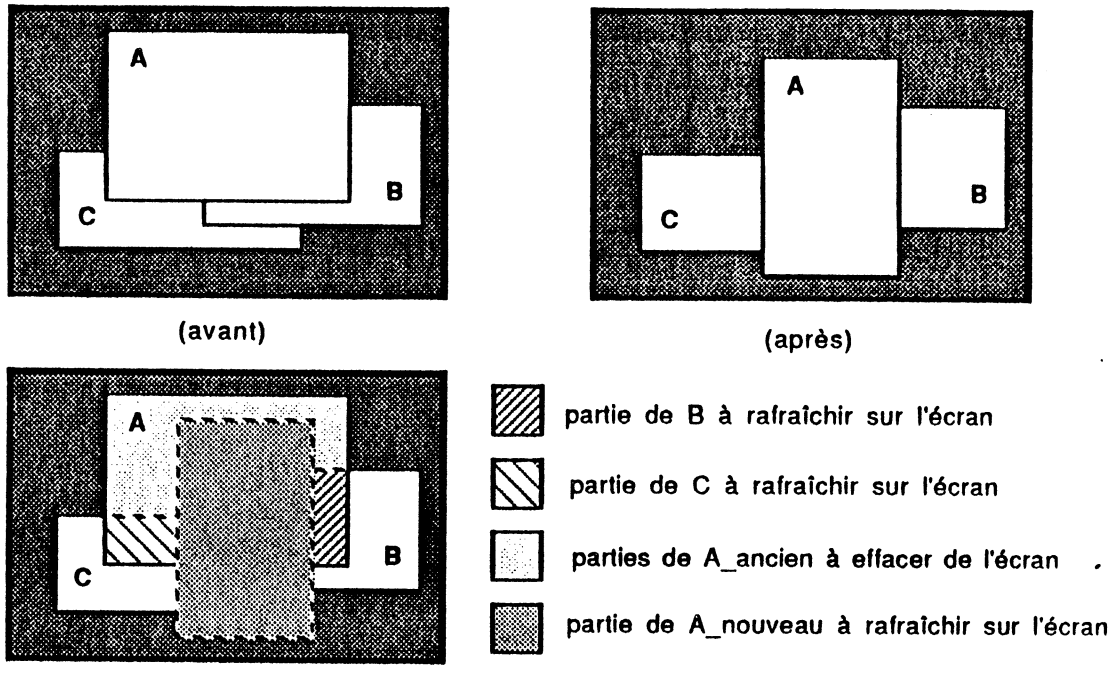
2) effacer de l'écran les parties visibles de *c_ancien* qui ne seront pas recouvertes par *c_nouveau*, c'est à dire effacer la différence entre *c_ancien* et *c_nouveau*. Cette opération consiste à mettre à jour les « parties visibles » du cadre *c_écran* incluses dans le rectangle délimitant *c_ancien*.

3) rafraîchir sur l'écran les parties visibles de *c_nouveau* à partir de la mémoire de sauvegarde de la fenêtre ayant *c* pour cadre.

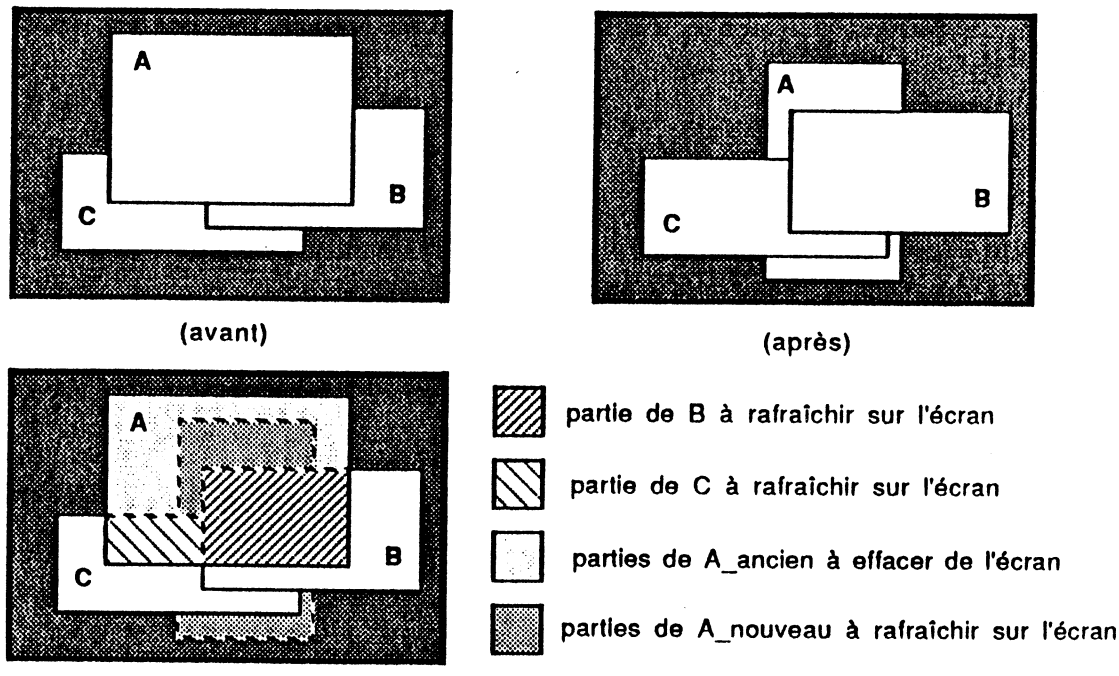
Remarque : la primitive *Changer_Taille* du gestionnaire de fenêtres appelée par les programmes d'application pour changer la taille d'une fenêtre alloue et initialise une mémoire de sauvegarde à la taille demandée avant d'appeler la procédure *Changer_Cadre* (voir IV.3.2).

La démarche suivie par la procédure *Changer_Cadre* est illustrée par deux exemples dans la figure 4.8 ci-dessous. Dans les deux cas, la position et la taille du cadre *A* sont modifiées et les cadres *A_ancien* et *A_nouveau* intersectent. Dans le premier exemple, le cadre *A* est monté au sommet de la « pile » de cadres alors qu'il est descendu en bas de la « pile » dans le second exemple.

Le cadre *c_ancien* sert uniquement à déterminer les parties des autres cadres qui doivent être mises à jour sur l'écran dans les deux premières étapes du traitement. Il n'est donc pas nécessaire de représenter effectivement le cadre *c_ancien* et seul le rectangle délimitant ce cadre suffit pour effectuer les opérations de mises à jour. La procédure *Changer_Cadre* exécute l'algorithme suivant :



Exemple 1 : A est monté au sommet de la pile de cadres



Exemple 2 : A est descendu en bas de la pile de cadres

Figure 4.8 Changement de taille et de position d'un cadre

```
procédure Changer_Cadre(c, r_nouveau, rang)
/* r_nouveau est le nouveau rectangle délimitant le cadre c sur l'écran. Le rang est soit
SOMMET_PILE soit BAS_PILE */
début
    si c caché alors
        /* changer uniquement le rectangle R(c) délimitant le cadre c */
        Changer_Rect(c, r_nouveau) ;
    sinon
        r_ancien ← R(c) ; /* le rectangle r_ancien délimite le cadre c_ancien */
        Changer_Rect(c, r_nouveau) ; /* c représente le cadre c_nouveau */
        c' ← Précédent(c) ;
        /* c' est le premier cadre éventuellement concerné par l'opération */
        Retirer(c) ;
        si rang = SOMMET_PILE alors
            /* c_nouveau doit être pris en compte lors du calcul des parties [des cadres] qui
            doivent être mises à jour sur l'écran dans la première étape du traitement */
            Insérer(c, rang) ;
        finsi
        /*  étape 1)  */
        tantque c' ≠ c_écran faire
            r' ← Intersection(r_ancien, R(c')) ;
            Parties_Visibles(c', r', Suivant(c')) ;
            c' ← Précédent(c') ;
        fin
        si rang = BAS_PILE alors
            Insérer(c, rang) ;
        finsi
        /*  étape 2)  */
        Parties_Visibles(c_écran, r_ancien, Suivant(c_écran)) ;
        /*  étape 3)  */
        Parties_Visibles(c, R(c), Suivant(c)) ;
    finsi
fin
```

La procédure *Changer_Cadre* se contente de modifier le rectangle délimitant un cadre caché, état dans lequel est placé un cadre lors de son initialisation par la procédure *Init_Cadre*. Ce principe permet de créer une fenêtre hors-écran et de ne la montrer qu'après avoir élaboré son bord par exemple.

Les opérations *Monter_Cadre* et *Montrer_Cadre* sont très simples à réaliser : insérer le cadre dans la file puis appeler la procédure *Parties_Visible*. De même, *Désigner_Cadre* se contente de parcourir la liste de cadres depuis la fin de la liste jusqu'à détection du cadre contenant le point *(x, y)* passé en paramètre.

3.2. Création d'une fenêtre

Créer une fenêtre consiste à allouer un *nom* et une *surface d'affichage* (cf III.3.2.2). Ces deux ressources sont gérées par le gestionnaire de fenêtres mais seul le nom d'une fenêtre est directement alloué à l'activité. Celle-ci doit ensuite envoyer une requête à *Fenix_Int* qui se charge d'allouer la surface d'affichage en appelant la primitive correspondante du gestionnaire de fenêtres. Ce protocole est transparent aux programmes d'application qui appellent la procédure *Créer_fenêtre* de la bibliothèque standard de Fenix pour créer une fenêtre.

Créer_Fenêtre commence par demander l'allocation d'un nom de fenêtres. Elle élabore ensuite une requête d'allocation d'espace d'affichage qu'elle envoie à *Fenix_Int*. Elle appelle pour cela une primitive spéciale du gestionnaire de fenêtres qui transmet la requête par l'intermédiaire de la procédure *Envoyer_Message* du gestionnaire du terminal graphique (voir IV.6). Sur réception de la requête, *Fenix_Int* contrôle sa validité puis effectue le traitement approprié à son type, par exemple afficher un message dans une fenêtre « système » pour inviter l'utilisateur à choisir interactivement la taille de la fenêtre. Il appelle ensuite la primitive *Changer_Taille* du gestionnaire de fenêtres pour allouer l'espace d'affichage de la fenêtre. Enfin, il envoie à l'activité un message d'acquittement qui contient le résultat de l'opération, par exemple la taille de la fenêtre si celle-ci a été choisie par l'utilisateur.

Allocation du nom

Chaque fenêtre possède deux noms globaux :

- un *numéro* correspondant à l'indice dans une table interne d'une structure de données de type *fenêtre* ;

- un nom de fichier spécial dans le répertoire */dev* du système de fichiers. Ce nom est de la forme */dev/windowX* où *X* représente le *numéro* de la fenêtre.

Un fichier spécial de nom */dev/window* sert uniquement à mettre en œuvre l'allocation d'un nom de fenêtre. Chaque ouverture de ce fichier par un processus est interprétée par le gestionnaire de fenêtres comme une demande d'allocation d'une [structure de données de type] *fenêtre*. Le gestionnaire de fenêtres alloue et initialise la première *fenêtre* libre - soit *I* son numéro - puis appelle la procédure *Init_Cadre* pour initialiser le *cadre* associé à la *fenêtre*. Il envoie ensuite un message à *Fenix_Int* pour l'avertir que la fenêtre numéro *I* a été allouée. Le gestionnaire de fenêtres simule ensuite dans le noyau d'Unix l'ouverture du fichier */dev/windowI* par le processus de façon à ce que le « file descriptor » retourné au processus désigne ce fichier et non le fichier */dev/window*.

Le « file descriptor » constitue le nom local de la fenêtre pour le processus. C'est ce nom qui désigne la fenêtre dans toutes les primitives de l'interface de base du gestionnaire de fenêtres.

Remarque 1 : aucun message n'est envoyé à *Fenix_Int* lorsque celui-ci est à l'origine de la demande d'allocation.

Remarque 2 : le gestionnaire de fenêtres retourne un code d'erreur lorsqu'un programme d'application essaie d'accèder directement à une fenêtre libre en ouvrant le fichier correspondant. Par défaut, toute fenêtre allouée est non partageable et seul *Fenix_Int* est autorisé à ouvrir le fichier associé à une fenêtre allouée.

Allocation de l'espace d'affichage

La surface d'affichage est allouée par la primitive *Changer_Taille* de l'interface de base du gestionnaire de fenêtres. Comme son nom l'indique, cette primitive sert également à changer la taille d'une fenêtre déjà créée. L'algorithme général de *Changer_Taille* est le suivant :

- allouer à partir d'un tas global un espace de la taille demandée ;

- appeler la procédure *Init_Surface* pour initialiser la mémoire de sauvegarde allouée à la fenêtre ;

- libérer le cas échéant l'ancienne mémoire de sauvegarde de la fenêtre ;

- appeler la procédure *Changer_Cadre* pour mettre à jour la mémoire-écran. Lorsque la fenêtre est cachée, ce qui est le cas lors de sa création, *Changer_Cadre* se contente de modifier le rectangle délimitant le *cadre* associé à la fenêtre.

La manière dont est initialisée la mémoire de sauvegarde allouée à une fenêtre est fonction du « type » de la fenêtre. Pour les fenêtres associées à un terminal virtuel (cf IV.5), cette opération consiste à recopier [une partie] de l'ancienne mémoire de sauvegarde dont l'adresse est passée en paramètre à la procédure *Init_Surface*. La mémoire de sauvegarde des autres fenêtres est initialisée à la couleur de fond de la fenêtre.

Le « tas » global réservé pour l'allocation des mémoires de sauvegarde est géré par le module *Gestion_Mem* qui recompacte à partir de l'origine du tas les parties déjà allouées lorsqu'il ne peut satisfaire immédiatement une demande d'allocation. Le module *Gestion_Mem* est conçu de façon à pouvoir opérer sur différents espaces-mémoire. La primitive *Init_Mem* initialise l'espace dont l'adresse et la taille lui sont passées en paramètre. Elle initialise au début de l'espace deux listes doublement chaînées qui représentent respectivement l'ensemble des blocs libres et l'ensemble des blocs alloués de cet espace. Ces listes sont ordonnées par adresses croissantes.

Chaque élément d'une liste est décomposé en deux parties contiguës : un descriptif de taille fixe placé au début de l'élément et le bloc proprement dit. Le descriptif contient les liens de chaînage, la taille du bloc et un « pointeur-arrière » dans le cas d'un bloc alloué. Le pointeur-arrière désigne le mot-mémoire contenant l'adresse du bloc dans la structure de données associée à l'objet auquel ce bloc est alloué.

L'adresse de l'espace, la taille demandée et le « pointeur-arrière » sont passés en paramètre à la procédure *Allouer_Mem* qui exécute l'algorithme suivant :

- rechercher le premier bloc libre d'une taille supérieure ou égale à celle demandée ;

- si aucun bloc libre ne convient, recompacter à partir du début de l'espace tous les blocs déjà alloués jusqu'à obtenir un bloc libre d'une taille suffisante pour satisfaire la demande. Cette opération utilise le « pointeur-arrière » du descriptif de chaque bloc alloué qu'elle déplace pour mettre à jour l'adresse de ce bloc dans l'objet auquel il est alloué.

- allouer un bloc d'une taille égale à celle demandée, initialiser son descriptif et retourner l'adresse du bloc.

La procédure *Allouer_Mem* retourne la valeur « NIL » lorsque le bloc libre obtenu après avoir recompacté tous les blocs alloués ne suffit pas à satisfaire la demande. La procédure *Libérer_Mem* a deux paramètres : l'adresse du bloc libéré et l'adresse de l'espace auquel appartient ce bloc. Cette procédure insère le bloc dans la liste des blocs libres et le fusionne avec son ou ses bloc(s) voisins si il y a lieu.

3.3. Manipulation des fenêtres

L'ensemble des primitives de manipulation supportées par le gestionnaire de fenêtres correspond à celui offert par le gestionnaire de cadres. Chacune de ces primitives a uniquement pour rôle de contrôler la validité des paramètres fournis par l'activité avant d'appeler la procédure appropriée du gestionnaire de cadres. La primitive *Changer_Taille* chargée d'allouer et de libérer la mémoire de sauvegarde des fenêtres est la seule à faire exception à cete règle.

Les primitives *Déplacer_Fenêtre* et *Montrer_Fenêtre* offrent une option pour déplacer et respectivement montrer une fenêtre à la position courante de la souris. La primitive *Déplacer_Fenêtre* demande alors cette position au gestionnaire du terminal graphique avant d'appeler la procédure *Changer_Cadre*. La primitive *Montrer_Fenêtre* se contente d'appeler dans ce cas *Déplacer_Fenêtre* avant d'appeler *Montrer_Cadre*.

Suivant le principe énoncé précédemment, chaque procédure de manipulation de la bibliothèque standard de Fenix élabore et envoie une requête à *Fenix_Int* qui se charge d'appeler la primitives correspondante du gestionnaire de fenêtres.

Pour des raisons d'efficacité, le gestionnaire de menus dynamiques implanté dans la bibliothèque standard de Fenix manipule directement les fenêtres dans lesquelles il crée les menus à la demande des programmes d'application. La procédure *Créer_Menu* appelle ainsi directement la primitive *Changer_Taille* du gestionnaire de fenêtres pour allouer l'espace d'affichage [de la fenêtre] du menu après avoir obtenu un nom de fenêtre selon la démarche classique. La procédure *Proposer_Menu* appelle de même les primitives *Montrer_Fenêtre* et *Cacher_Fenêtre* pour faire apparaître un menu puis le faire disparaître avant de retourner la commande sélectionnée par l'utilisateur. *Proposer_Menu* offre également une option pour faire apparaître le menu à la position courante de la souris.

3.4. L'interface graphique

Le système Fenix n'ayant pas pour objectif initial d'être un système graphique élaboré, nous avons choisi d'offrir aux applications toutes les opérations graphiques disponibles au niveau du terminal graphique utilisé. Selon les capacités du terminal, ces opérations peuvent éventuellement être [en partie] assurées par un co-processeur graphique. Nous avons isolé cette partie dépendante du matériel dans un module implanté dans le gestionnaire du terminal graphique. C'est au niveau de ce module que sont définis les objets de base tels que le *point*, le *rectangle* et la *matrice de points*.

Sur le terminal Numelec, les opérations de ce module doivent être entièrement réalisées par logiciel. La conception d'un tel logiciel est une tâche importante et c'est pourquoi nous avons jugé préférable d'intégrer dans Fenix la bibliothèque de « Raster_Op » développée à l'INRIA. Cette bibliothèque codée en langage d'assemblage Motorola 68000 inclut les opérations suivantes :

- *Vid_Cnt* qui trace un segment entre deux points (x1, y1) et (x2, y2), ce dernier non compris ;

- *Rast_Op* qui copie dans une matrice de points « cible » un rectangle d'une matrice de points « source ». Cette opération permet de réaliser différentes fonctions logiques (ET, OU, etc.) entre deux points correspondants des deux matrices ;

• *Fill* qui initialise à une couleur quelconque un rectangle inclus dans une matrice de points ;

• *Vid_Chr* qui affiche un caractère d'une police dans une matrice de points.

L'interface de base du gestionnaire de fenêtres fournit un ensemble équivalent de primitives graphiques opérant dans une fenêtre. Ces primitives sont d'autre part utilisées dans la bibliothèque standard pour implanter des fonctions de plus haut niveau telles que le défilement d'informations dans une fenêtre par exemple.

Remarque : *Rast_Op* et *Fill* servent également à réaliser les deux opérations *Rafraîchir_Ecran* et *Effacer_Ecran* de l'interface de bas niveau du gestionnaire de cadres.

Le système Fenix adopte dans son principe la démarche décrite en III.3.3.1 pour gérer l'affichage de caractères. Chaque [structure de données de type] *fenêtre* contient un ensemble d'attributs utilisés pour gérer l'affichage de caractères dans une fenêtre. Cet ensemble comprend le nom de la police courante, la position courante du curseur, l'aspect du curseur et son état allumé ou éteint. L'aspect du curseur est défini par un caractère quelconque d'une police quelconque éventuellement différente de la police courante. Le gestionnaire de fenêtres fournit un ensemble de primitives pour changer la valeur de chacun des attributs d'une fenêtre et deux primitives pour afficher une chaîne de caractères dans une fenêtre :

• la primitive *write* de l'interface standard d'Unix qui affiche la chaîne de caractères à la position courante et avec la police courante associées à la fenêtre ;

• la primitive *Afficher* qui affiche une chaîne de caractères avec une police et à une position quelconques passées en paramètre.

La primitive *write* gère le déplacement du curseur ; elle efface le curseur à la position courante, affiche les caractères à partir de cette position qu'elle met à jour avant d'afficher à nouveau le curseur. La primitive *Afficher* implantée sous forme d'une commande *ioctl* ne change ni la police courante ni la position courante du curseur. Elle évite ainsi aux programmes d'application d'avoir à modifier et à restaurer ces valeurs pour afficher une chaîne de caractères avec une police et/ou à une position particulières.

Un module indépendant contrôlé par un pilote associé au fichier spécial */dev/police* est chargé de gérer dans le noyau un espace global contenant les polices de caractères chargées en mémoire à la demande des activités. Ce module alloue à chaque police chargée un *nom* qui sert à désigner la police dans toutes les autres primitives de l'interface du système. Le *nom* d'une police est l'indice dans une table interne d'un descriptif regroupant les informations suivantes :

- le nom du fichier à partir duquel la police a été chargée ;
- un compteur du nombre courant d'utilisateurs de la police ;
- les caractéristiques de la police telles que la hauteur et la largeur maximum des caractères par exemple ;
- l'adresse dans l'espace global à partir de laquelle sont rangées les matrices de points décrivant l'aspect des caractères de la police.

Les programmes d'application appellent la procédure *Charger_Police* de la bibliothèque standard de Fenix pour demander le chargement d'une police désignée par le nom du fichier qui la contient. Cette procédure exécute la séquence suivante :

- vérifier que le fichier existe et que son contenu est conforme au format de description d'une police ;
- ouvrir le fichier */dev/police* pour avoir accès au module de gestion de polices ;
- appeler la fonction *Allouer_Police* de ce module avec le nom du fichier comme paramètre d'appel. Cette fonction retourne deux valeurs :
  - un *booléen* indiquant si oui ou non la police est déjà chargée en mémoire ;
  - le *nom* alloué à la police ;
- si la police n'est pas déjà chargée en mémoire, lire dans le fichier les matrices de points décrivant l'aspect des caractères de la police et transférer ces matrices au gestionnaire de polices par l'intermédiaire de la primitive *Copier_Police*.
- fermer le fichier */dev/police* ;
- retourner le *nom* de la police.

L'espace mémoire partagé entre toutes les polices chargées est également géré par le module *Gestion_Mem* décrit précédemment. Le compteur d'une police est incrémenté lorsqu'une activité demande à charger la police. Il est décrémenté à chaque fois qu'une activité cesse d'utiliser la police, soit implicitement lors de son arrêt, soit explicitement par appel de la primitive *Libérer_Police*. Lorsque la valeur du compteur après décrémentation devient nulle, l'espace occupé par la police est libéré.

3.5. Gestion des zones

Le système Fenix permet de découper une fenêtre en régions rectangulaires appelées *zones*. A chaque *zone* sont associés un réticule souris et la file d'entrée de l'activité qui a créé la zone, l'activité propriétaire de la zone. La souris se trouve en permanence dans une zone d'une fenêtre, la *zone courante*, qui détermine le réticule affiché sur l'écran par le module de contrôle du terminal graphique.

Remarque : le gestionnaire de fenêtres associe par défaut à chaque fenêtre une zone d'une taille égale à celle de la fenêtre.

Lorsque l'utilisateur appuie sur un des boutons de la souris, l'activité propriétaire de la zone courante est sélectionnée comme l'activité courante par le gestionnaire d'entrées. Il peut être alors intéressant de répartir l'ensemble des fonctions d'une même application entre plusieurs programmes exécutés par des processus séparés se partageant une même fenêtre pour dialoguer avec l'utilisateur. La démarche sous-jacente consiste à allouer une file d'entrée à chaque processus qui se crée ensuite ses propres zones dans la fenêtre.

Par exemple, un outil de mise au point de programmes peut être décomposé en deux processus qui utilisent une même fenêtre partagée horizontalement en deux zones, un processus qui visualise le programme source dans la zone inférieure de la fenêtre et un processus qui implante les commandes de mise au point dans la zone supérieure. Cette décomposition est transparente à l'utilisateur pour lequel la fenêtre représente l'entité logique de dialogue avec l'application de mise au point de programmes.

Les zones sont utilisées par le gestionnaire de menus dynamiques intégré dans la bibliothèque standard de Fenix. Chaque menu est implanté dans une fenêtre dans laquelle sont créées autant de zones qu'il y a de commandes dans le menu. Lorsque

l'utilisateur sélectionne un point quelconque du menu, le nom de la zone inclus dans l'événement transmis à l'activité désigne directement la commande choisie.

La primitive *Créer_Zone* de l'interface du gestionnaire de fenêtres retourne le *nom* de la zone créée, nom qui sert à désigner une zone dans les primitives suivantes de l'interface du système :

- les primitives permettant de modifier la position, la taille et le réticule d'une zone ;
- la primitive *Afficher* de l'interface graphique ;
- la primitive *Lire_Event* qui retourne le premier événement présent dans la file d'entrée de l'activité. Le gestionnaire d'entrées inclut systématiquement dans chaque événement la position courante de la souris sous la forme « nom de fenêtre, nom de zone, position par rapport à l'origine de la zone » ;
- la primitive *Détruire_Zone* qui détruit la zone désignée.

Les zones possèdent une propriété supplémentaire : la taille et la position d'une zone sont gérées par défaut proportionnellement à la taille de la fenêtre dans laquelle la zone a été créée. Cette propriété évite à l'activité propriétaire de la zone d'avoir à spécifier une nouvelle position et/ou taille de la zone lorsque la taille de la fenêtre est modifiée.

4. Le gestionnaire d'entrées

Le gestionnaire d'entrées a pour rôle de transmettre aux activités les événements émis par l'utilisateur depuis les dispositifs d'entrée du terminal graphique et de multiplexer ces dispositifs entre les activités. Rappelons que le principe adopté dans Fenix consiste à allouer à chaque activité une seule file d'entrée dans laquelle sont rangés tous les événements qui lui sont destinés. Cette file est associée au flot d'entrée standard du processus créé par *Fenix_Init* lorsqu'il démarre une activité à la demande de l'utilisateur (cf IV.7 ci-dessous).

L'allocation d'une file d'entrée est effectuée selon une démarche analogue à celle suivie pour l'allocation d'un nom de fenêtre (cf IV.3.2). L'espace occupé par les événements d'une file n'a pas à être réservé par l'activité : cet espace est alloué dynamiquement par le gestionnaire d'entrées à partir d'un tas global partagé entre

toutes les files. Le gestionnaire d'entrées refuse d'allouer plus d'une file à un processus mais autorise par contre plusieurs processus - un processus et tous ses fils par exemple - à se partager une même file.

•Une file d'entrée est libérée lorsque le dernier processus y ayant accès ferme cet accès, soit implicitement en se terminant, soit explicitement en fermant son flot d'entrée standard à l'aide de la primitive *close*. Le gestionnaire d'entrées vide alors la file, libère l'espace occupé par les événements qu'elle contenait et envoie un message à *Fenix_Int* pour l'avertir que la file a été libérée.

Le gestionnaire d'entrées multiplexe les dispositifs d'entrée selon le principe décrit en III.2.2.1.1. L'ensemble des dispositifs d'entrée est attribué à chaque instant à une seule file - la *file courante* - qui peut être sélectionnée de trois manières :

- par *Fenix_Int* qui désigne la nouvelle file courante à l'aide d'une primitive spéciale du gestionnaire du terminal graphique. Cette primitive appelle la procédure *Select_File* du gestionnaire d'entrée avec le nom de la file désignée ;

- lorsque la file courante est libérée. La file d'entrée de *Fenix_Int* devient alors automatiquement la file courante ;

- par l'utilisateur lorsqu'il appuie sur un bouton de la souris. La file associée à la zone contenant la souris à cet instant - la *zone courante* - est sélectionnée comme la file courante par le gestionnaire d'entrée.

Les événements sont transmis aux programmes d'application sous la forme d'une structure de données de type *event* regroupant les informations suivantes :

- le type de l'événement ;

- la valeur de l'événement ;

- la position de la souris ;

- la date d'émission de l'événement.

La position de la souris est décrite sous la forme d'une structure de données qui contient la position relative de la souris par rapport à l'origine de la *zone courante*, le nom de la *zone courante* et le nom de la *fenêtre courante* (la fenêtre contenant la *zone courante*). Le gestionnaire d'entrées conserve en permanence ces

trois informations dans la variable *position_courante* qu'il met à jour à chaque déplacement de la souris. Il appelle pour cela la fonction *Pos_Courante* du gestionnaire de fenêtres qui détermine chacune de ces informations à partir de la position absolue [de la souris] passée en paramètre.

Le gestionnaire d'entrées distingue deux classes d'événements : les événements *physiques* reflétant un changement d'état d'un dispositif d'entrée à la suite d'une action de l'utilisateur et les événements *synthétisés* par le système (cf III.3.4.1.2).

La classe des événements physiques comprend trois types d'événements - *caractère*, *déplacement_souris* et *click_souris* - qui correspondent respectivement à la frappe d'une touche du clavier, au déplacement de la souris et à l'enfoncement ou au relâchement d'un bouton de la souris. Les événements synthétisés sont générés par le gestionnaire d'entrées dans les cas suivants :

- une nouvelle file courante est sélectionnée par l'une des trois manières décrites ci-dessus ;

- la souris entre dans une zone ou en sort, c'est à dire en cas de changement de *zone courante* puisque la souris se trouve en permanence dans une zone d'une fenêtre (cf IV.3.5).

La sélection d'une nouvelle file courante est réalisée par la procédure *Select_File* du gestionnaire d'entrées qui envoie un événement de type *fin_sélection* dans la file courante, change la file courante et envoie dans celle-ci un événement de type *début_sélection*. En cas de changement de zone courante, le gestionnaire d'entrées envoie un événement de type *sortie_zone* relatif à l'ancienne zone courante et un événement de type *entrée_zone* relatif à la nouvelle zone courante.

Le gestionnaire d'entrées ne change pas la file courante lorsque l'utilisateur déplace la souris sur l'écran : seule la zone courante est éventuellement changée et la file associée à la zone courante ne devient la file courante qu'au moment où l'utilisateur enfonce un bouton de la souris. Pour être cohérent avec ce choix, le gestionnaire d'entrées n'envoie aucun événement de type *déplacement_souris*, *sortie_zone* et *entrée_zone* si la file associée à la zone concernée n'est pas la file courante. L'utilisateur peut ainsi déplacer la souris en dehors des fenêtres de l'activité courante sans provoquer l'envoi d'événements à d'autres activités et sans changer l'activité courante à laquelle sont toujours envoyés les caractères du clavier.

Les programmes d'application peuvent spécifier le type des événements qui doivent uniquement leur être transmis par le gestionnaire d'entrées. Un filtre d'événements de type *déplacement_souris*, *click_souris*, *entrée_zone* et *sortie_zone* est attaché à chaque zone créée par une activité. Un programme d'application peut ainsi créer deux zones dans une fenêtre et demander par exemple à recevoir uniquement les événements de type *déplacement_souris* dans une zone et les événements de type *entrée_zone* et *sortie_zone* dans l'autre zone. Le filtre des événements de type *caractère*, *fin_sélection* et *début_sélection* est par contre attaché à la file de l'activité.

Les événements sont transmis de manière synchrone. Les programmes d'application appellent la procédure *Lire_Event* de la bibliothèque standard de Fenix pour accéder aux événements. Cette procédure appelle la primitive *read* sur le flot d'entrée standard du processus pour récupérer le premier événement de la file associée et retourne cet événement. La primitive *read* du gestionnaire d'entrées déroule la séquence suivante :

- bloquer le processus si la file est vide ;
- retirer le premier événement de la file et recopier son contenu dans l'environnement utilisateur du processus ;
- libérer [l'espace occupé par] l'événement ;

Les événements sont générés à partir des événements « bruts » émis par les dispositifs d'entrée à la suite d'une action de l'utilisateur. Le gestionnaire d'entrées distingue trois types d'événements « bruts » - caractère clavier, déplacement souris et click souris - et associe à chacun d'eux une procédure de traitement. Ces trois procédures sont appelées par la ou les routine(s) d'interruption du gestionnaire du terminal graphique chargée(s) de contrôler les dispositifs d'entrée.

La procédure *Traiter_Car* se contente de générer et d'envoyer dans la file courante un événement de type *caractère*. La procédure *Traiter_Click* sélectionne comme file courante la file associée à la zone courante (si les deux files sont différentes à cet instant) puis génère et envoie dans la file courante un événement de type *click_souris*. La procédure *Traiter_Dep* effectue le traitement suivant :

- appeler la fonction *Pos_Courante* du gestionnaire de fenêtres pour mettre à jour la variable *position_courante* après avoir sauvegardé son ancienne valeur ;

si changement de zone courante **alors**

- demander au gestionnaire du terminal graphique d'afficher le réticule associé à la nouvelle zone courante ;
- générer et envoyer dans la file courante un événement de type *sortie_zone* et un événement de type *entrée_zone* ;

sinon

- générer et envoyer dans la file courante un événement de type *déplacement_souris* ;

finsi

Le gestionnaire d'entrées ne génère aucun événement de type *déplacement_souris* lorsque la souris change de zone courante, les événements de type *sortie_zone* et *entrée_zone* ayant justement pour but de rendre compte de ce cas particulier. Les procédures de traitement et la procédure *Changer_File* déroulent la séquence suivante pour chaque événement qu'elles ont à générer et à envoyer dans la file courante :

si les conditions d'envoi de l'événement sont satisfaites **alors**

- allouer [l'espace occupé par] un événement ;
- élaborer l'événement ;
- appeler la procédure *Envoyer_Event* qui insère l'événement en queue de la file courante et réveille si nécessaire le premier processus en attente sur la file ;

finsi

Les conditions d'envoi d'un événement dépendent du type de l'événement à envoyer. Par exemple, la procédure *Traiter_Dep* appelée à chaque déplacement de la souris génère et envoie dans la file courante un événement de type *sortie_zone* si la file associée à l'ancienne zone courante (la zone quittée par la souris) est la file courante et si l'activité propriétaire de la zone en question a demandé à recevoir ce type d'événement.

5. Le gestionnaire de terminaux virtuels

Un terminal virtuel est une entité logique composée d'une fenêtre et d'une file d'entrée sur lesquelles sont émulées les fonctions d'un terminal classique. Le gestionnaire de terminaux virtuels s'interface entre d'une part le gestionnaire de fenêtres et le gestionnaire d'entrées et, d'autre part, le module « tty » d'Unix vis à vis duquel il joue le rôle d'un contrôleur de terminal classique. Pour être indépendant des caractéristiques de chaque type de terminal, le module *tty* communique avec les contrôleurs de terminaux à travers une couche spécifique à chacun d'eux. C'est au niveau de cette couche que sont implantées les fonctions d'émulation d'un terminal classique.

La fenêtre et la file d'entrée d'un terminal virtuel sont regroupées dans une structure de données de type *term_virt*. Chaque terminal virtuel possède deux noms globaux, un numéro et un nom de fichier spécial de la forme */dev/vtermX* où *X* est le numéro du terminal virtuel. Un terminal virtuel est créé lors du démarrage d'une activité *Shell* par *Fenix_Init* (cf IV.7 ci-dessous). Le nom d'un terminal virtuel est alloué selon la technique utilisée pour allouer un nom de fenêtre (cf IV.3.2), le « file descriptor » retourné au processus désignant dans ce cas la fenêtre et la file d'entrée du terminal virtuel alloué. La surface d'affichage de la fenêtre est allouée comme pour une fenêtre normale par la primitive *Changer_Taille* du gestionnaire de fenêtres.

L'espace d'affichage de la fenêtre de chaque terminal virtuel est géré sous forme d'un tableau de caractères à deux dimensions sur lequel sont émulées les fonctions standard assurées par les terminaux alphanumériques (défilement vertical de la dernière ligne, etc.). Pour simplifier la mise en œuvre de ces fonctions, le gestionnaire de terminaux virtuels impose que la police courante de la fenêtre soit à châsse fixe, c'est à dire que tous les caractères de la police soient de même largeur (cf II.2.1).

Enfin, les filtres d'événements respectivement attachés à la file d'entrée et à la zone intérieure de la fenêtre d'un terminal virtuel sont initialisés de façon à n'autoriser que la transmission de caractères dans la file (cf IV.4).

6. Le gestionnaire du terminal graphique

Le gestionnaire du terminal graphique a principalement pour rôle de gérer les ressources matérielles du terminal graphique et de transmettre à *Fenix_Int* les messages envoyés par les activités.

Gestion des ressources matérielles

Cette partie regroupe l'ensemble des fonctions dépendantes du terminal graphique. Cet ensemble comprend les primitives graphiques de base (les « Raster-Op »), les fonctions de contrôle des dispositifs d'entrée et la primitive *Suivre_Souris* chargée d'afficher sur l'écran le réticule de la souris.

Sur le terminal Numelec, les primitives graphiques sont entièrement réalisées par logiciel (cf IV.3.4) ainsi que le suivi de la souris sur l'écran. La primitive *Suivre_Souris* affiche le réticule dans la mémoire-écran en appliquant la fonction logique *ou exclusif* entre la matrice de points du réticule et le rectangle correpondant de la mémoire-écran (cf II.2.2). Le mécanisme d'accès en exclusion mutuelle à la mémoire-écran est implanté à l'aide d'un verrou qui conditionne l'affichage du réticule dans la primitive *Suivre_Souris* de la manière suivante :

si réticule_affiché = *vrai* **alors**
- effacer le réticule à la position courante de la souris ;
- réticule_affiché ← *faux* ;

finsi
- mettre à jour la position courante ;

si verrou_affichage = *faux* **alors**
- afficher le réticule à la position courante de la souris ;
- réticule_affiché ← *vrai* ;

finsi

Le verrou d'affichage du réticule est géré par les deux primitives *Arrêter_Suivi* et *Autoriser_Suivi* respectivement appelées au début et à la fin de chaque procédure du gestionnaire de cadres. La primitive *Arrêter_Suivi* met le verrou à *vrai* puis appelle *Suivre_Souris* qui efface le réticule si celui-ci est affiché. La primitive *Autoriser_Suivi* met le verrou à *faux* puis appelle *Suivre_Suivre* qui affiche à nouveau le réticule.

Les fonctions de contrôle des dispositifs d'entrée récupèrent les événements « bruts » émis par le clavier et la souris et envoient la valeur de ces événements au gestionnaire d'entrées par l'intermédiaire des trois primitives *Traiter_Car*, *Traiter_Click* et *Traiter_Dep* (cf IV.4).

Gestion de la communication entre les activités et *Fenix_Int*

Les messages transmis à *Fenix_Int* sont de deux types :

- les messages envoyés par le gestionnaire de fenêtres (respectivement d'entrées) à chaque fois qu'une fenêtre (respectivement une file d'entrée) est allouée ou libérée par une activité ;

- les requêtes de manipulation de fenêtres envoyées par les activités. Le type et la sémantique de ces messages sont entièrement définis par *Fenix_Int*, le mécanisme de communication imposant uniquement une taille maximum aux messages échangés.

Tous ces messages sont transmis à *Fenix_Int* par l'intermédiaire de la file d'entrée associée au terminal graphique, file dans laquelle lui sont également transmis les événements physiques envoyés par l'utilisateur. Ce choix est dû au fait que la primitive de lecture d'Unix version 7 ne permet pas l'attente simultanée sur plusieurs flots d'entrée.

Par défaut, seuls les événements physiques sont transmis et c'est à *Fenix_Int* d'autoriser explicitement la transmission d'un message dans la file d'entrée associée au terminal graphique. La procédure *Envoyer_Message* du gestionnaire du terminal graphique n'envoie dans la file d'entrée qu'un seul message à la fois et bloque les autres activités émettrices jusqu'à ce que *Fenix_Int* autorise la transmission du message suivant. *Fenix_Int* est ainsi assuré de ne recevoir que des événements physiques dans sa file d'entrée pendant qu'il dialogue avec l'utilisateur. Enfin, l'activité à l'origine du message transmis reste bloquée jusqu'à ce que sa requête soit acquittée par une réponse de *Fenix_Int*.

Le gestionnaire du terminal graphique est associé à un fichier spécial et *Fenix_Int* communique avec lui par l'intermédiaire des primitives d'entrées/sorties standard d'Unix. La primitive *read* retourne le premier message ou le premier événement de la file associée au terminal graphique. La primitive *write* affiche directement la chaîne de caractères dans la mémoire-écran et est uniquement

utilisée pour la mise au point du programme *Fenix_Int* lui-même. La primitive *ioctl* sert de support aux différentes commandes permettant à *Fenix_Int* de :

- changer la couleur de fond de l'écran ;
- changer la file courante (cf IV.4) ;
- contrôler la transmission de messages dans la file d'entrée et retourner les messages d'acquittement des requêtes envoyées par les activités ;
- sélectionner la police de caractères et la police de réticules standard du système ;
- placer le terminal graphique dans le mode « multi-fenêtres ». Cette commande autorise le suivi de la souris sur l'écran et l'allocation des ressources contrôlées par les gestionnaires d'entrée, de fenêtres et de terminaux virtuels ;
- placer le terminal graphique dans le mode « terminal virtuel ».

Lors de l'ouverture du fichier spécial associé au terminal graphique, le terminal est automatiquement placé dans le mode « terminal virtuel » et géré comme tel à l'aide des fonctions d'émulation du gestionnaire de terminaux virtuels (cf IV.5). Au démarrage du système Unix, le programme *init* associe au terminal graphique un processus qui ouvre le fichier spécial correspondant puis exécute le programme *login* pour permettre à un utilisateur de se connecter au système.

Le fichier */etc/passwd* contient pour chaque utilisateur connu du système le nom du programme à exécuter lorsque l'utilisateur se connecte. Par défaut, ce programme est l'interpréteur de commandes *Shell*. Chaque utilisateur peut ainsi utiliser le terminal graphique comme un terminal alphanumérique classique. Il peut alors démarrer *Fenix_Int* depuis le *Shell* pour exploiter les possibilités du système multi-fenêtres ...

7. L'interface utilisateur

L'interface utilisateur est gérée par le programme *Fenix_Int* qui est un client privilégié du système multi-fenêtres intégré dans le noyau. Il est en particulier le seul à pouvoir dialoguer avec le pilote du terminal graphique qu'il contrôle.

Fenix_Int commence par initialiser le terminal graphique de façon à permettre son exploitation dans le mode « multi-fenêtres ». Il charge en mémoire les polices standard de caractères et de réticules souris. Il appelle ensuite le pilote de gestion du terminal graphique pour autoriser le mode « multi-fenêtres » et le suivi de la souris sur l'écran. A partir de ce moment, les déplacements de la souris sont pris en compte et la position de la souris est visualisée sur l'écran par le système. L'activité *Fenix_Int* devient l'activité courante et tous les événements physiques émis par l'utilisateur sont placés dans la file d'entrée associée au terminal graphique.

Fenix_Int met en place l'interface utilisateur en créant une fenêtre « système » pour l'affichage de messages et un menu de commandes qui apparaît lorsque l'utilisateur sélectionne n'importe quel point de l'écran situé en dehors de toute fenêtre. Ce menu offre par défaut les commandes suivantes :

- démarrer un *Shell* ;
- démarrer une activité d'édition de texte. Un éditeur « plein écran » a été adapté pour fonctionner dans une fenêtre et exploiter la souris ;
- arrêter une activité ;
- arrêter *Fenix_Int* et sortir du mode « multi-fenêtres ». Cette commande provoque l'arrêt de toutes les activités et fait revenir le poste de travail dans le mode « terminal virtuel ». L'utilisateur se retrouve alors sous le *Shell* à partir duquel il avait lancé *Fenix_Int* ;
- changer la couleur de fond de l'écran. La sélection de cette commande fait apparaître un second menu dynamique offrant un choix de couleurs prédéfinies ;

Démarrer une activité consiste à créer un processus par appel de la primitive système *fork()* et à lui allouer un flot d'entrée ou un terminal virtuel si l'application sélectionnée fait partie de l'environnement standard d'Unix, ce qui est le cas du *Shell*. Le démarrage d'une activité est effectué selon la séquence suivante :

```
début
    pid ← fork()
    si pid = 0 alors  /* c'est le processus fils */
        close(0) ;      /* fermer le flot d'entrée standard */
        si l'application fait partie de l'environnement standard d'Unix alors
            open(/dev/vterm) ;   /* allouer un terminal virtuel */
        sinon
            open(/dev/input) ;   /* allouer une file d'entrée */
        finsi
        exécuter le programme d'application par appel de la primitive
        système exec() ;
    finsi
fin
```

Les applications de l'environnement standard d'Unix telles que le *Shell* sont en fait démarrées indirectement. Dans ce cas, le processus créé exécute tout d'abord un programme spécial - le « lanceur » - qui déroule la séquence suivante :

- associer le flot de sortie standard du processus à la fenêtre du terminal virtuel alloué ;

- envoyer à *Fenix_Int* une requête d'allocation d'espace d'affichage pour créer la fenêtre ;

- exécuter enfin le programme d'application par appel de la primitive système *exec()*.

Le menu de commandes de *Fenix_Int* offre deux commandes de démarrage d'un *Shell*. Lorsque l'utilisateur sélectionne la commande *User_Shell*, le « lanceur » du *Shell* transmet une requête d'allocation d'espace en demandant à ce que l'utilisateur choisisse interactivement la taille de la fenêtre. Lorsque l'utilisateur sélectionne la commande *System_Shell*, la taille de la fenêtre est par contre automatiquement calculée par le « lanceur » du *Shell* et transmise dans la requête d'allocation. Cette taille est calculée de façon à permettre l'affichage d'un tableau de 80 * 24 caractères avec la police standard du système.

Une fois terminée sa phase d'initialisation, *Fenix_Int* se met en attente sur son flot d'entrée standard pour recevoir les événements physiques en provenance de l'utilisateur et les messages envoyés par les activités. Comme au départ il est la seule

activité démarrée, il ne peut recevoir que des événements physiques sur son flot d'entrée. Il va notamment recevoir un tel événement lorsque l'utilisateur va sélectionner un point de l'écran pour faire apparaître le menu de commandes et démarrer une activité.

Remarque : avant de se mettre en attente sur son flot d'entrée standard, *Fenix_Int* peut éventuellement démarrer un certain nombre d'activités initiales (un *Shell* par exemple) définies dans un fichier de « profile » associé à l'utilisateur.

Après avoir été démarrée, une activité commence (normalement) par créer une ou plusieurs fenêtres. *Fenix_Int* va alors recevoir des requêtes de création de fenêtres. Ces requêtes peuvent contenir la taille demandée par l'activité ou au contraire spécifier que cette taille doit être choisie par l'utilisateur. Dans ce cas, *Fenix_Int* affiche un message dans la fenêtre système pour inviter l'utilisateur à choisir la taille de la fenêtre à l'aide de la souris.

Fenix_Int appelle ensuite la primitive *Changer_Taille* du gestionnaire de fenêtres pour allouer l'espace d'affichage de la fenêtre. *Fenix_Int* peut alors construire le bord de la fenêtre à l'aide des opérations graphiques offertes par le gestionnaire de fenêtres. Comme toute activité, il peut également créer dans la fenêtre ses propres zones correspondant à la surface occupée par le bord. Comme la file d'entrée de l'activité est associée aux zones qu'elle crée, *Fenix_Int* deviendra automatiquement l'activité courante lorsque l'utilisateur sélectionnera un point quelconque du bord de la fenêtre et recevra l'événement correspondant. *Fenix_Int* peut ainsi implanter les commandes de manipulation (déplacer, cacher, etc.) dans le bord des fenêtres de façon totalement transparente aux activités.

Après avoir créé la fenêtre, *Fenix_Int* envoie un compte rendu d'acquittement à l'activité. Cette activité est alors réveillée et peut commencer à afficher dans la fenêtre. Elle ne peut toutefois modifier la surface occupée par les zones éventuellement créées par *Fenix_Int*. Seule la zone intérieure de la fenêtre est visible au niveau de l'activité. La taille de cette zone est égale à la taille demandée dans la requête de création, *Fenix_Int* créant la fenêtre avec une taille réelle égale à la « somme » de la taille demandée et de la surface occupée par le bord.

La file d'entrée de l'activité est associée à la zone intérieure de la fenêtre. L'activité peut également se créer ses propres zones à l'intérieur de la fenêtre avant de se mettre en attente sur son flot d'entrée standard pour recevoir les événements en provenance de l'utilisateur. Dans ce cas, elle reste bloquée jusqu'à ce que

l'utilisateur la fasse devenir « activité courante » en sélectionnant un point quelconque à l'intérieur de la fenêtre.

Chapitre V

CONCLUSION

Nous avons dégagé et analysé dans cette thèse les principes généraux des systèmes multi-fenêtres puis nous avons présenté le système Fenix développé sur Unix.

Fenix est un système multi-fenêtres qui permet à l'utilisateur de mener en parallèle plusieurs activités possédant chacune une ou plusieurs fenêtres pour communiquer avec l'utilisateur. Il offre également un ensemble de mécanismes destinés à faciliter la conception d'outils de dialogue évolués.

1. Evaluation du système Fenix

Le prototype actuel fonctionne sur des machines SPS7 de Bull et répond aux principaux objectifs que nous nous étions fixés. Le système est composé de deux parties. La partie opératoire qui regroupe les fonctions de base (suivi de la souris, gestion des fenêtres, primitives graphiques, etc.) et qui est intégrée dans le noyau d'Unix. La partie contrôle qui gère l'interface utilisateur du système et qui est implantée dans un programme exécuté au dessus du noyau par un processus ordinaire. Cette architecture offre un compromis satisfaisant entre, d'une part la recherche de performances acceptables pour l'utilisateur et, d'autre part, le souci de faciliter l'adaptation et l'évolution de l'interface de commandes du système.

L'interface de manipulation des fenêtres inclut la plupart des commandes - déplacer, monter, descendre, cacher, montrer et changer la taille d'une fenêtre - étudiées en III.2.2.1.2. Les commandes de manipulation sont implantées dans le bord des fenêtres. Nous avons choisi d'attribuer un emplacement différent à chaque

commande et le reste du bord à un menu qui les regroupe toutes, solution qui s'avère tout à fait satisfaisante à l'usage.

Nous avons développé dans Fenix une méthode de gestion du recouvrement efficace qui apporte un élément de confort indéniable au niveau de la manipulation des fenêtres. Toutes les opérations de manipulation sont conçues pour ne modifier sur l'écran que les points réellement concernés par la manipulation et pour modifier une seule fois chacun de ces points. Le respect de cette contrainte permet d'éliminer les effets visuels inutiles et désagréables (les « trous noirs » sur l'écran par exemple) que nous avons pu observer sur de nombreux systèmes.

Le système Fenix ne permet pas à l'utilisateur de transférer des informations d'une fenêtre à l'autre selon le principe du « couper-coller ». Cette lacune importante sur le plan de l'interface utilisateur ne peut pas pour autant être imputée aux choix de conception de Fenix mais à l'absence dans Unix version 7 d'un mécanisme de communication interprocessus approprié. Nous considérons que la mise en œuvre du « couper-coller » n'entre pas dans les attributions d'un système multi-fenêtres mais doit être implantée à un niveau supérieur. C'est uniquement à ce niveau que peuvent être conservées sous une forme adaptée à leur transmission ultérieure les informations affichées dans les fenêtres. Le système ne manipule en effet que des matrices de points qui, à de rares exceptions près, seraient inexploitables par l'application réceptrice.

L'interface utilisateur de Fenix est pour l'instant gérée « statiquement » et toute adaptation de l'interface nécessite de modifier le programme correspondant. Un effort important reste à faire sur ce plan pour permettre à l'utilisateur de choisir le style de dialogue qui lui convient. Peu de systèmes offrent actuellement cette facilité, le système X paraissant le plus avancé dans ce domaine [Gancarz 86]. La solution à ce problème dépasse en fait le cadre du système multi-fenêtres et doit être envisagée pour l'ensemble des applications interactives dont l'interface de commandes du système ne constitue qu'un cas particulier.

Examinons maintenant l'interface de programmation de Fenix au niveau de la bibliothèque standard mise a à la disposition des programmeurs d'applications.

Cette bibliothèque propose un ensemble de procédures de manipulation de fenêtres (création et destruction comprises) relativement puissant. Ces procédures se contentent d'élaborer, à partir des paramètres d'appel, des requêtes qu'elles transmettent au programme gérant l'interface utilisateur par l'intermédiaire de la fonction d'envoi de messages intégrée dans le gestionnaire de fenêtres. L'interface

utilisateur du système se charge ainsi de la gestion du dialogue nécessaire à la réalisation de l'opération lorsque celle-ci requiet l'intervention de l'utilisateur, pour choisir par exemple la taille d'une fenêtre au moment de sa création.

Le système gère le recouvrement des fenêtres de façon complètement transparente aux programmes d'application. Il conserve pour cela en mémoire hors-écran une copie complète du contenu des fenêtres. Cette méthode est toutefois très controversée en raison de l'espace mémoire relativement important que sa mise en œuvre nécessite. Ses détracteurs invoquent en particulier l'apparition de terminaux graphiques munis d'écrans polychromes pour appuyer leur argumentation.

La solution intermédiaire adoptée dans Sapphire [Myers 85] et permettant aux applications de demander la création de fenêtres sans mémoire de sauvegarde nous semble être un bon compromis. La mémoire de sauvegarde est par exemple inutile pour les fenêtres des menus qui ne restent visibles sur l'écran que pour la durée d'une sélection. De plus, le contenu d'un menu reste inchangé entre temps et peut donc être facilement regénéré à chaque fois qu'il doit être visualisé sur l'écran. Ce principe s'applique en fait à toutes les fenêtres utilisées dans un but identique, pour les formulaires par exemple.

L'interface graphique est sans nul doute possible le point faible de Fenix, notamment au niveau de l'affichage d'objets graphiques puisque la version actuelle du gestionnaire de fenêtres ne supporte que le tracé de segments de droites. La bibliothèque exploite cette fonction pour offrir des procédures de tracé d'objets graphiques plus élaborés (rectangles, etc.) mais reste très inférieure à celle d'un système comme SunDew basée sur le langage PostScript et même à celle du Macintosh.

Le fait d'avoir implanté le gestionnaire de fenêtres dans le noyau constitue un handicap difficile à surmonter puique toute extension graphique, aussi minime soit elle, nécessite de regénérer le noyau. Il n'est pas non plus possible d'intégrer dans le noyau d'Unix un module graphique couvrant une large gamme de besoins en raison du volume de code important que cela représente.

Nous envisageons de pallier cette insuffisance en ajoutant une primitive de transfert de matrices de points entre l'espace d'adressage d'un processus et une fenêtre. Cette solution permet d'étendre à volonté les capacités graphiques du système en reportant cette extension au niveau des applications mais elle se traduit par une diminution importante des performances. Elle a de plus l'inconvénient de

rendre les applications dépendantes du matériel, à moins d'introduire la notion de matrice de points logique qui ne fait que compliquer sa mise en œuvre. En fait, ce problème n'a pas vraiment de solution satisfaisante.

Chaque activité accède aux informations d'entrée qui lui sont destinées par l'intermédiaire d'une file unique associée à son flot d'entrée standard. Le système range dans cette file tous les événements émis par l'utilisateur depuis les dispositifs d'entrée ainsi que les événements synthétisés tels que ceux qui indiquent à l'activité qu'elle est devenue (respectivement qu'elle n'est plus) l'activité courante.

Cette méthode tend à s'imposer dans les systèmes récents tels que X [Rosenthal 86] et nous semble supérieure à celle plus classique qui consiste à associer une file d'entrée par fenêtre. Elle garantit la transmision des événements d'entrée dans l'ordre chronologique de leur émission, ce qui est primordial pour la cohérence du dialogue entre l'utilisateur et les activités. Elle respecte d'autre part la philosophie initiale d'Unix en séparant de façon nette les entrées et les sorties d'un programme, principe sur lequel s'appuient les notions de « pipe-line » et de « filtres » qui constituent un des apports fondamentaux d'Unix. Nous pensons qu'un tel schéma reste valable dans un contexte multi-fenêtres où les entrées d'une application doivent pouvoir être filtrées par un processus intermédiaire. Ce processus peut par exemple gérer une partie du dialogue avec l'utilisateur ou bien simuler le fonctionnement d'un dispositif d'entrée inexistant ou défectueux.

2. Perspectives

Le système Fenix comme d'ailleurs la majorité des systèmes multi-fenêtres intégrés à Unix gèrent uniquement les ressources directement liées aux fonctions d'entrées-sorties (dispositifs d'entrée, mémoires de sauvegarde des fenêtres, polices de caractères, etc.). En particulier, aucune modification n'est introduite au niveau de la stratégie d'allocation du processeur et de la mémoire centrale. Ces deux ressources continuent à être gérées selon le principe classique des systèmes en temps partagé qui consiste à les répartir équitablement dans le temps entre des programmes s'exécutant pour le compte de plusieurs utilisateurs.

Un tel principe n'est pas adapté aux postes de travail individuels où par définition toutes les activités s'exécutent pour le compte d'un même utilisateur qui communique à un instant donné avec une seule d'entre elles, l'*activité courante*.

Dans un tel contexte, le système doit gérer le processeur et la mémoire centrale selon une stratégie qui privilégie l'activité courante pour garantir un meilleur temps de réponse à l'utilisateur. Cette stratégie doit distinguer l'activité courante des autres activités, même lorsque l'activité courante est bloquée en attente d'une entrée de l'utilisateur comme le sont (normalement) toutes les autres. Ce dernier point concerne plus particulièrement la gestion de la mémoire centrale : le système doit éviter autant que possible de transférer en mémoire secondaire le contexte de l'activité courante lorsque celle-ci est bloquée sur une opération d'entrée.

Privilégier l'activité courante consiste en fait à tenir compte de la destination *potentielle* des entrées de l'utilisateur. En d'autres termes, il faut que la stratégie appliquée à chaque ressource prenne en compte les *décisions* de l'utilisateur (l'attribution des dispositifs d'entrée) et ne se contente plus de réagir uniquement à ses *actions* (l'envoi d'événements).

La mise au point d'une stratégie d'allocation de ressources satisfaisant aux objectifs visés est une tâche complexe, en raison notamment du comportement non déterministe des activités contrôlées par le système. Les postes de travail individuels sont de conception relativement récente et nous pensons que la gestion de leurs ressources est un domaine encore très ouvert.

ANNEXE

Le système Unix

1. Introduction

Le système Unix [Thompson 74] a été conçu aux Laboratoires Bell à partir de 1969 par Ken Thompson et Dennis Ritchie pour fournir un environnement de programmation interactif sur des petites machines du type PDP-11 de DEC.

Ce qui fait l'originalité d'Unix, c'est la démarche suivie par ses concepteurs dont l'objectif initial était d'exploiter leur expérience d'utilisateur pour se doter d'un outil adapté à leurs besoins de programmeurs. Les premières versions du système écrites en assembleur ne furent effectivement exploitées qu'en usage interne aux Laboratoires Bell.

En 1975, l'ensemble du système réécrit par Thompson et Ritchie en langage « C » [Kernighan 78] commence à être diffusé à l'extérieur de la Bell sous le nom d'*Unix version 6* puis à être porté sur des machines différentes de celles de la gamme PDP-11.

Pour résoudre le problème du transport d'Unix, ses concepteurs introduirent d'importantes modifications au niveau du langage et du compilateur « C » et restructurent le système pour en isoler les parties dépendantes du matériel. La *version 7* d'Unix est le résultat de ces travaux et représente actuellement la version de référence du système qui comprend :

- le système d'exploitation proprement dit (le noyau Unix) ;

• l'interpréteur de commandes (le *Shell*) ;

• de nombreux utilitaires tels que éditeurs et formateurs de textes, compilateurs (C, Fortran 77, Pascal), outils de mise au point de programmes.

2. Le noyau Unix

Le noyau peut être décomposé en trois sous-ensembles fonctionnels assurant respectivement [le contrôle de] l'exécution des programmes, la réalisation des entrées-sorties et la gestion de l'espace d'archivage.

2.1. Exécution des programmes

Dans Unix, chaque programme est exécuté par un processus séquentiel évoluant parmi deux *environnements* strictement exclusifs et qualifiés respectivement d'*environnement utilisateur* et d'*environnement système*.

Ces deux environnements constituent l'*image* du processus et représentent l'état de ce qu'on peut appeller la machine abstraite « Unix » offrant comme opérateurs le jeu d'instructions du processeur réel et l'ensemble des primitives systèmes.

Chaque primitive système est exécutée dans l'environnement système du processus, les changements d'environnement s'effectuant durant l'appel et le retour de la primitive.

L'environnement utilisateur

L'environnement utilisateur est un espace-mémoire virtuel linéaire divisé logiquement en trois segments (voir figure 1 ci-dessous) :

• le *segment de code* du programme commençant à l'adresse 0 de cet espace ;

• le *segment de données* situé après le code et dont la taille peut être modifiée par appel de la primitive système *break* ;

• la *pile d'exécution* implantée dans la partie « haute » de l'espace virtuel et qui est agrandie automatiquement par le système au fur et à mesure des besoins.

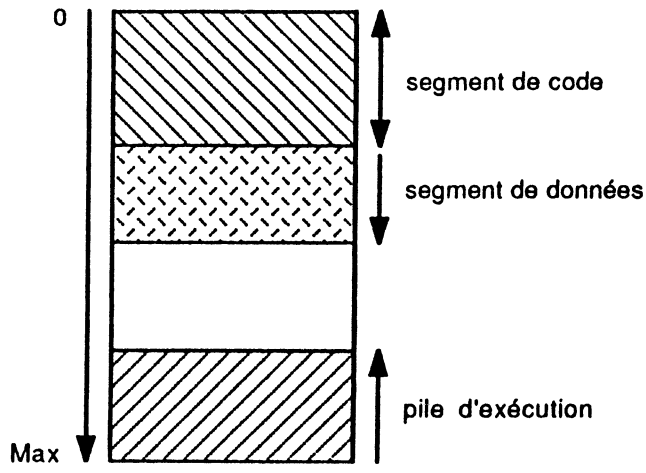


Figure 1 Environnement utilisateur d'un processus

Un processus peut exécuter un nouveau programme par appel de la primitive système

exec(prog, parm1, parm2, ..., parmn)

qui remplace son environnement utilisateur (code et données) par celui contenu dans le fichier de nom *prog* et transfère le contrôle d'exécution au point d'entrée du nouveau programme, sans possibilité de retour à l'appelant. Le segment de pile du processus est réinitialisé et contient selon un format prédéfini les chaînes de caractères $parm_i$ constituant les paramètres d'appel de *prog*.

Le noyau permet le partage en lecture d'une seule copie en mémoire du segment de code d'un programme exécuté par plusieurs processus. Cette possibilité est optionnelle et doit être explicitement demandée par l'utilisateur au translateur-éditeur de liens chargé de construire un programme exécutable selon le format imposé par le noyau.

L'environnement système

- L'environnement système regroupe les informations (état des fichiers ouverts, répertoire courant, etc.) qui, pour des raisons de protection, ne peuvent être manipulées que par appel de primitives systèmes.

Il contient aussi une zone de taille fixe utilisée comme pile d'exécution des différentes procédures du noyau impliquées dans la réalisation d'une primitive système pour le compte du processus.

La primitive système *exec* ne modifie pas l'environnement système du processus exécutant un nouveau programme qui conserve ainsi l'accès aux fichiers déjà ouverts par exemple.

Création de processus

Excepté durant la phase d'initialisation du noyau, un processus ne peut être creé que par un autre processus appellant la primitive système *fork*. L'image du processus créé - le fils - est initialisée par recopie de celle du processus créateur - le père - dans une nouvelle zone de mémoire allouée au fils.

Il n'y a donc aucun partage de mémoire entre les deux processus (si ce n'est éventuellement le segment de code du programme) mais tous deux exécutent en séquence du *fork* le même programme dans des environnements identiques. Le fils hérite ainsi du répertoire courant de son père et de l'accès aux fichiers ouverts par celui-ci.

Seule la valeur entière retournée par *fork* à chacun des processus leur permet de s'identifier : nulle pour le fils et strictement positive pour le père (numéro unique forgé par le noyau et identifiant le fils).

Synchronisation des processus

Unix n'offre aucun mécanisme de synchronisation (sémaphores ou autres) permettant à plusieurs processus de coopérer de façon cohérente pour accéder à un même fichier par exemple.

En fait, le seul type de synchronisation disponible consiste pour un processus père à attendre la terminaison de l'un de ses fils grâce à la primitive système *wait* qui retourne l'identificateur du processus fils terminé et la cause de cette terminaison. Cette primitive n'autorise pas l'attente sélective qui doit donc être gérée par le processus père.

La terminaison d'un processus peut être provoquée :

- par le noyau à la suite d'un mauvais fonctionnement du programme (instruction illégale, violation mémoire, etc.) ;
- par l'utilisateur émettant depuis son terminal un caractère spécial interprété par le noyau ;

• par un autre processus à l'aide de la primitive système *kill* ;

• par le processus lui-même exécutant la primitive système *exit* avec comme paramètre un code de terminaison destiné au processus père.

Communication entre processus

Nous avons vu précédemment que seul le segment de code d'un programme peut être partagé en lecture uniquement par plusieurs processus. Pour assurer la communication inter-processus, Unix offre la notion de *tube* (« pipe ») qui est un flux de données géré par le noyau selon le modèle des producteurs-consommateurs et accédé comme un fichier au niveau des opérations de lecture et d'écriture.

Un processus crée un tube par appel de la primitive système *pipe* qui ajoute à son environnement système deux accès (en lecture et en écriture respectivement) à ce fichier particulier et retourne un identificateur local associé à chacun d'eux.

Le seul moyen qu'a un processus de transmettre l'accès à un tube est de créer un processus fils qui en hérite par copie d'environnements. Cette contrainte limite l'emploi des tubes à des processus apparentés ou ayant un ancêtre commun et coopérant à la réalisation d'une même tâche.

Dans Unix version 7, les fichiers représentent le seul moyen d'assurer d'autres schémas d'échange d'informations entre processus, par exemple :

• le partage de données communes entre plusieurs processus coopérants ;

• la communication entre processus utilisateurs et processus serveurs (serveur d'imprimante, d'une station de transport avec des systèmes distants, etc.).

Traitement d'événements exceptionnels

Un certain nombre d'événements matériels ou logiciels appelés *signaux* peuvent être envoyés de manière asynchrone à un processus :

• par le système détectant une anomalie quelconque :
  - coupure d'une ligne de communication ;
  - violation mémoire ;
  - écriture dans un tube sans processus consommateur ;

• par l'utilisateur depuis son terminal à l'aide de caractères spéciaux (« del », « quit ») filtrés par le système ;

• par un autre processus appellant la primitive système *kill* avec comme paramètres le numéro de l'événement signalé et l'identité du processus destinataire.

Selon l'option demandée par le processus destinataire à l'aide de la primitive système *signal*, le noyau peut déclencher une des trois actions suivantes lors de l'envoi d'un signal :

• forcer la terminaison du processus (action par défaut) ;

• ignorer le signal qui est alors définitivement perdu ;

• provoquer l'appel de la procédure désignée par le processus dans la primitive *signal* pour traiter ce signal.

Cette procédure est exécutée dans l'environnement utilisateur du processus comme une séquence d'interruption avec sauvegarde et restitution automatiques par le système du contexte courant du processus, sauf si celui-ci est en attente sur certaines primitives systèmes qui se terminent alors prématurément en retournant un code d'erreur.

Les signaux ne sont pas mémorisés par le système. Pour un processus donné, seul le dernier signal envoyé est conservé jusqu'à sa prise en compte éventuelle par le processus. La principale fonction des signaux est en fait de permettre la prise en compte de situations anormales ne permettant plus le déroulement correct des processus concernés et le contrôle par l'utilisateur de programmes défaillants (boucle infinie, etc).

Un autre cas d'utilisation est celui du signal « horloge » déclenché en fin d'un délai de temps armé à l'aide de la primitive système *alarm* et qui permet par exemple d'assurer une fonction de chien de garde sur des opérations d'entrée-sortie.

2.2. Le système de fichiers

Du point de vue de l'utilisateur, il existe trois types de fichiers dans Unix :

• les *répertoires* (« directories ») ;

• les *fichiers ordinaires* ;

• les *fichiers spéciaux*.

Les répertoires

Le système de fichiers est organisé selon une structure arborescente dont les nœuds intermédiaires sont des *répertoires*. Chaque répertoire est un fichier assurant la correspondance entre les noms externes (14 caractères maximum) d'autres fichiers et leur nom interne (un entier) donnant accès à leur descripteur.

La désignation d'un fichier est une suite d'identificateurs séparés par le caractère « / » représentant le *chemin d'accès* (« pathname ») à ce fichier. Le caractère « / » en tête de la suite désigne le répertoire *racine* (unique au système) de l'arbre et offre ainsi un moyen d'accès absolu à tout fichier du système.

Par ailleurs, le système associe à chaque processus un répertoire courant permettant de désigner un fichier par un chemin d'accès relatif à celui-ci. Ce type d'accès est évidemment plus simple pour l'utilisateur et plus efficace au niveau de la gestion des ressources internes du noyau.

Unix permet d'établir plusieurs chemins d'accès à un même fichier par une opération de création de *lien* (appel système *link*) qui consiste à référencer ce fichier dans des répertoires différents de l'arborescence. La création d'un lien n'est possible que vers un fichier déjà existant, mais tous les chemins d'accès à un fichier sont équivalents. Il n'y a pas de commande « directe » de destruction d'un fichier : l'appel système *unlink* détruit une référence à un fichier qui n'est effectivement détruit que s'il s'agit du dernier lien.

Enfin, un lien ne peut être établi vers un répertoire, ce qui garantit l'absence de cycle dans l'arborescence.

Les fichiers ordinaires

Les fichiers ordinaires contiennent les informations stockées par les utilisateurs, par exemple des documents imprimables ou des programmes en binaire.

Aucune méthode de structuration particulière n'est gérée ni donc imposée par le système d'entrées-sorties : un fichier ordinaire est une suite d'octets de taille variable, mais *toujours croissante*.

Remarque : les fichiers contenant un programme exécutable ont une structure particulière imposée par la primitive *exec* jouant le rôle de chargeur dans le noyau.

Les fichiers spéciaux

Les fichiers spéciaux sont la seule voie d'accès aux différents périphériques gérés par le système. Les opérations d'entrée-sortie sur un fichier spécial se traduisent par l'exécution des fonctions correspondantes associées au pilote (« driver ») du périphérique concerné.

Les périphériques gérés par le système sont traités comme tous les autres fichiers d'Unix et sont regroupés par convention dans le répertoire « /dev » de l'arborescence. Cette approche offre plusieurs avantages. Les fichiers et les périphériques sont désignés suivant les mêmes règles syntaxiques, leur accès est contrôlé par le même mécanisme de protection et les opérations d'entrée-sortie sont identiques.

Gestion des volumes

L'ensemble des fichiers de l'arborescence qui est unique peut résider sur plusieurs volumes installés sur des unités périphériques différentes et éventuellement amovibles. Un volume doit avoir la structure arborescente d'un système de fichiers indépendant et est appelé dans la terminologie Unix un *système de fichiers* (« file system »).

Le système de fichiers d'un volume peut être intégré dans l'arborescence globale par appel de la primitive système *mount* avec comme paramètres le nom du fichier spécial donnant accès au volume monté et celui d'un répertoire qui représente ensuite la racine du nouveau sous-arbre.

La primitive système *umount* permet de démonter un volume si aucun des fichiers du sous-arbre correspondant n'est ouvert pour un des processus existants.

La racine (unique) du système de fichiers global est celle du volume à partir duquel a été chargé le code du noyau lors du démarrage du système (le volume de « bootstrap »), volume qu'il n'est pas possible de démonter.

Au niveau utilisateur, il n'y a aucune distinction entre les fichiers appartenant à un volume monté et ceux du volume permanent, la seule restriction concerne les liens qu'il n'est pas possible d'établir entre deux volumes différents.

2.3. Le système d'entrées-sorties

Un des objectifs d'Unix est de permettre l'écriture de programmes d'application totalement indépendants de la nature des supports (fichier ordinaire, terminal, tube, etc.) utilisés pour leurs échanges d'informations avec l'extérieur.

Pour cela, le système d'entrées-sorties offre une interface basée sur la notion classique de *flot* [Krakowiak 85]. Dans Unix, un flot est une suite d'octets désignée dans toutes les primitives d'accès par un entier appelé le « file descriptor ». Chaque processus dispose dans son environnement système d'un ensemble de flots (20 en général) qui peuvent être associés indifféremment à des fichiers ordinaires ou spéciaux, des répertoires ou des tubes.

Le système gère *implicitement* toutes les ressources nécessaires à la réalisation des entrées-sorties. En particulier, la taille d'un fichier ordinaire est automatiquement augmentée au fur et à mesure des besoins et n'est limitée qu'en fonction des ressources physiques disponibles.

Les primitives système *creat(nom, mode)* et *open(nom, mode)* permettent de créer un fichier ordinaire ou d'ouvrir un fichier quelconque (ordinaire, spécial, répertoire) avec le mode d'accès - lecture et/ou écriture - indiqué. Toutes deux initialisent un descripteur d'accès au fichier et l'associent à un nouveau flot alloué dans l'environnement système du procesus et retournent le numéro de ce flot qui représente pour le processus le *nom local* du fichier.

Le système gère dans chaque descripteur d'accès un pointeur courant sur le prochain octet à lire ou à écrire. Ce pointeur est automatiquement incrémenté par les primitives systèmes d'entrée-sortie *read* et *write* qui sont donc implicitement séquentielles.

La primitive *read(fd, tampon, n)* permet de lire dans la zone mémoire *tampon* de l'environnement utilisateur du procesus *n* octets à partir de l'octet courant du flot *fd* et retourne le nombre d'octets effectivement lus ou un code d'erreur éventuelle. De même, la primitive *write(fd, tampon, n)* permet d'écrire dans un flot.

Les entrées-sorties sont synchrones. La primitive *read* bloque ainsi le processus tant que l'opération de lecture n'est pas terminée et un même processus ne peut pas se mettre simultanément en attente d'informations sur plusieurs flots.

Le système autorise l'accès aléatoire dans un fichier lorsque le support physique le permet. La primitive système *lseek(fd, dep, org)* déplace le pointeur courant dans le fichier *fd* de *dep* octets à partir de la position *org* indiquée comme suit :

- 0 pour le début du fichier ;
- 1 pour la position courante du pointeur ;
- 2 pour la fin du fichier.

puis retourne la nouvelle position du pointeur ou un code d'erreur. L'opération *lseek* est par définition interdite sur un tube et sans effet sur certains fichiers spéciaux comme les terminaux par exemple.

La primitive système *close(fd)* rompt l'association entre le flot *fd* et le fichier ou le tube correspondant et libère le flot dans l'environnement système du processus.

Cette interface inclue également la primitive système *ioctl* qui peut uniquement être invoquée sur des fichiers spéciaux et qui est utilisée pour contrôler le fonctionnement des périphériques correspondants. C'est notamment à l'aide de cette fonction qu'un programme peut contrôler le traitement effectué par le module *tty* du noyau sur les caractères reçus du clavier. Ce module s'interface entre les routines d'interruption associées au(x) contrôleur(s) des lignes de connection et les fonctions d'entrée-sortie du ou des pilote(s) correspondant(s). Le module *tty* assure par défaut le traitement suivant sur les caractères entrés par l'utilisateur au clavier :

- l'écho des caractères ;
- la mémorisation temporaire des caractères jusqu'à réception du caractère « fin de ligne », avec interprétation de certains caractères particuliers permettant par exemple à l'utilisateur de commander l'effacement du caractère précédent ou de toute la ligne courante.

3. L'interpréteur de commandes

Dans Unix, un processus est associé à chaque ligne de connection d'un terminal pour gérer sur cette ligne l'interface entre un utilisateur et le système. Aucun privilège spécial n'est attaché à ces processus. En particulier, leur image peut aussi être rangée sur disque par le module du noyau assurant le partage de la mémoire centrale entre tous les processus.

Le fait d'exécuter ainsi le programme d'interface au système *en dehors du noyau* présente plusieurs avantages :

- La taille de ce programme ne constitue pas un facteur limitatif de sa puissance qui peut donc être augmentée à moindre coût sans pénaliser le système ;

- le développement et l'intégration d'une nouvelle version de ce programme ne nécessitent aucune intervention au niveau du noyau et peuvent donc être réalisées facilement et rapidement.

Cette interface est mise en place de la manière suivante dans Unix. Après avoir été chargé en mémoire au moment du démarrage du système, le noyau termine sa phase d'initialisation par la création d'un processus dont le segment de code contient uniquement l'équivalent en binaire de l'appel de *exec(/etc/init)*. Le rôle du programme *init* est de créer un processus indépendant pour chaque ligne de connection disponible dans le système.

Après avoir été créé, chacun de ces processus ouvre en lecture puis en écriture le fichier spécial correspondant à la ligne qu'il gère et exécute le programme *login* pour permettre à un utilisateur de se connecter. Chaque utilisateur connu du système est répertorié par un identificateur associé dans le fichier */etc/passwd* à une entrée contenant un mot de passe optionnel et les noms respectifs d'un répertoire et d'un programme exécutable représentant le contexte de début de *session* de l'utilisateur.

A partir de ce fichier, le programme *login* contrôle l'identificateur et le mot de passe éventuel fournis par un utilisateur puis installe celui-ci dans son contexte initial en changeant le répertoire courant du processus et en exécutant le programme indiqué dans ce contexte.

En général, ce programme est l'interpréteur de commandes *Shell* qui constitue l'interface privilégié du système Unix. Sa fonction principale est de donner accès à toutes les ressources offertes par le système en permettant à l'utilisateur de lancer interactivement l'exécution d'autres programmes.

Syntaxe des commandes du Shell

Dans sa forme la plus simple, une commande est une ligne de caractères contenant le nom de la commande suivi de ses paramètres séparés par un ou plusieurs blancs.

Le nom de la commande est le nom externe d'un fichier contenant le programme à exécuter. Ce nom peut être absolu ou relatif, auquel cas le Shell applique un certain nombre de règles de recherches programmables pour trouver le fichier si celui-ci n'est pas dans son répertoire courant. Par exemple,

ls /usr

exécute le programme de nom global */bin/ls* qui affiche sur l'écran le nom des fichiers contenus dans le répertoire */usr*.

Le mode d'interaction est du type « exécution séquentielle de commandes ». Chaque commande est exécutée par un processus créé par le Shell qui attend sa fin avant d'en accepter une autre selon le schéma fonctionnel suivant :

faire toujours
afficher un message d'invite (le « prompt ») sur l'écran ;
acquérir une ligne de commande ;
analyser la ligne et copier dans les variables *cmd* et $parm_i$ le nom de la commande et les différents paramètres ;
pid ← *fork()* ; /* crée un processus */
si *pid* = 0 **alors**
/* c'est le processus-fils : lui faire exécuter la commande ;
$(parm_i)$ = liste des paramètres d'appel de la commande */
exec(cmd, $(parm_i)$*)* ;
/* pas de retour... sauf si erreur (commande inexistante par exemple) */
finsi
/* séquence exécutée par le père, c'est à dire le Shell */
wait() ; /* attend la terminaison du processus fils */
fin

Au retour de *wait*, le Shell affiche de nouveau son message d'invite et informe ainsi l'utilisateur :

- que la commande courante est terminée ;
- qu'il est prêt à en acquérir une nouvelle.

Entrées-Sorties standard

Lorsqu'un processus est créé, il hérite de l'environnement système de son père et donc de tous les flots qui y sont déjà associés et sur lesquels il peut effectuer directement des opérations d'entrée-sortie. C'est ainsi que le Shell et tous les programmes lancés par celui-ci commencent leur exécution avec leurs flots 0 et 1 respectivement ouverts en lecture et en écriture.

Ces deux flots appelés *flots d'entrée-sortie standard* sont par défaut associés au clavier et à l'écran du terminal de l'utilisateur.

Le Shell permet de rediriger les entrées-sorties standard d'un programme avant de l'exécuter. Lorsque l'un des paramètres d'une ligne de commande est préfixé par le caractère « > », le Shell interprète directement ce paramètre comme le nom d'un fichier à créer et à associer au flot de *sortie standard* du processus exécutant la commande. De même, un paramètre préfixé par le caractère « < » est interprété comme le nom d'un fichier à ouvrir et à associer au flot d'*entrée standard*. Par exemple, l'exécution de la commande

ls > liste

a pour effets de copier dans le fichier *liste* le nom de tous les fichiers contenus dans le répertoire courant, y compris *liste* lui-même.

Le Shell assure la redirection des flots d'entrée-sortie standard de façon complètement transparente aux programmes qui les utilisent. Si un élément $parm_j$ de la liste de paramètres $(parm_i)$ est de la forme « <nom », le processus créé par le Shell pour exécuter la commande déroule la séquence suivante :

retirer l'élément $parm_j$ de la liste ;

close(0) ;

open(nom) ;

avant d'appeller la commande par *exec(cmd, (parm$_i$))*.

La même séquence avec *close(1)* puis *creat(nom)* est éventuellement utilisée si un argument de la liste est de la forme « >nom ».

Les pipe-line

Le Shell permet l'exécution « simultanée » de plusieurs commandes selon un schéma de fonctionnement global de type *pipe-line* où la sortie standard de chaque commande du pipe-line est redirigée par l'intermédiaire d'un *tube* sur l'entrée standard de la commande suivante.

Syntaxiquement, un pipe-line est composé d'une séquence de commandes séparées par le caractère « | ». Par exemple, la ligne de commande

ls /bin | lpr

a pour effets de faire imprimer par la commande *lpr* les noms des fichiers du répertoire */bin* fournis par la commande *ls*.

Ce type de redirection est lui aussi géré de façon transparente aux différentes commandes constituant un pipe-line ; le Shell crée un processus pour exécuter chacune d'elles et un tube pour relier, selon une méthode analogue à celle décrite précédemment, la sortie et l'entrée standard de deux commandes successives du pipe-line.

Pris dans son ensemble, un pipe-line peut être perçu par l'utilisateur comme une commande unique. Il possède entre autres un flot d'entrée (respectivement de sortie) standard représenté par celui de la première (respectivement la dernière) commande de la séquence correspondante, chacun de ces flots pouvant être redirigé vers un fichier comme pour une commande simple.

Exécution de commandes en parallèle

Le Shell créant un nouveau processus pour exécuter une commande, tous deux évoluent dans des environnements (utilisateur et système) différents et peuvent donc à priori s'exécuter « simultanément » de façon indépendante. En attendant la fin d'une commande immédiatement après l'avoir lancée, le Shell garantit en fait à celle-ci l'accès exclusif au terminal de l'utilisateur qui constitue la seule ressource potentiellement partagée par les deux processus via leurs flots d'entrée-sortie standard respectifs.

Le Shell est conçu pour conserver durant l'exécution d'une commande l' environnement système dont il a hérité initialement. Par exemple, le Shell ne redirige jamais ses propres flots standard pour rediriger (à l'aide du mécanisme d'héritage) les flots standard d'une commande qui sont effectivement redirigés par le processus chargé d'exécuter la commande.

Cette caractéristique est exploitée pour permettre à l'utilisateur de faire exécuter simultanément plusieurs commandes en *arrière-plan*. Lorsque le caractère « & » apparaît à la suite d'une commande, le Shell n'attend pas la fin de celle-ci mais retourne immédiatement après l'avoir lancée pour acquérir et exécuter une nouvelle commande fournie par l'utilisateur. Dans l'exemple suivant :

ls /dev > dev.liste &
date

ls copie dans le fichier *dev.liste* le contenu du répertoire */dev* « pendant que » *date* affiche sur l'écran l'heure et la date courantes.

Il est préférable que les tâches lancées ainsi en parallèle avec le Shell ne soient pas elles-mêmes interactives car le partage du même terminal entre tous ces processus est alors assuré par l'ordonnanceur (le « scheduler ») du noyau et échappe au contrôle de l'utilisateur. Les caractères entrés au clavier sont dans ce cas « distribués » de façon aléatoire à chacun des processus et les informations qu'ils affichent sont mélangées sur l'écran. Pour pallier ces inconvénients, certains programmes interactifs autres que le Shell permettent à l'utilisateur d'exécuter temporairement un autre programme interactif ou non et de revenir ensuite dans le contexte initial. Cette facilité est notamment offerte par la plupart des éditeurs sous la forme de l'opérateur « ! » préfixant le nom de la commande. Toutefois, un tel mode d'exécution est analogue à celui d'un appel de procédure avec empilement et dépilement de contexte.

Les fichiers de commandes

Le Shell est lui-même une commande que l'utilisateur peut faire exécuter récursivement. Celui-ci peut donc lancer une deuxième instance du Shell en redirigeant son flot d'entrée standard sur un fichier ordinaire pour lui faire exécuter ainsi les différentes commandes que ce fichier contient.

Outre l'invocation de commandes, le langage offert par le Shell inclut la notion de *paramètre*, la manipulation de *variables* et des structures de contrôle

analogues à celles fournies par les langages algorithmiques classiques telles que *while*, *if-then-else*, *case*, etc.

L'utilisateur peut donc élaborer de véritables programmes exprimés en langage « Shell » et les mettre au point facilement puisque ce langage est interprété. Soit par exemple le fichier *print* suivant :

```
case $1 in
    /) FILENAME=$1 ;;
    *) FILENAME=`pwd`/$1 ;;
esac
echo $FILENAME > /dev/lp
tabtospace < $1 > /dev/lp
```

L'interprétation de la commande

sh print < print

provoque le lancement d'une nouvelle instance de */bin/sh* qui par redirection de son flot d'entrée standard « exécute » en fait le programme *print* dont le rôle est d'imprimer le nom global puis le contenu du fichier passé en paramètre, en l'occurence ici *print* lui-même.

/dev/lp étant le nom du fichier spécial associé à l'imprimante et *$1* référençant le [premier] paramètre d'appel du programme */bin/sh*, celui-ci interprète le contenu du fichier *print* comme suit :

1) affecter à la variable *FILENAME* le nom global du fichier passé en paramètre:
   - soit en recopiant la valeur du paramètre dans le cas où celui-ci commence par le caractère « / ».
   - soit en concaténant le nom global du répertoire courant, le caractère « / » et la valeur du paramètre dans tous les autres cas; le nom global du répertoire courant est alors obtenu en exécutant la commande *pwd*.

2) exécuter la commande *echo* qui copie sur son flot de sortie standard la valeur de chacun de ses paramètres d'appel pour faire imprimer la valeur de la variable *FILENAME*.

3) exécuter la commande *tabtospace* qui produit en sortie les données d'entrée en remplaçant chaque caractère de tabulation par un nombre fixe d'espaces pour enfin imprimer le contenu du fichier.

Cet aspect purement algorithmique contribue ainsi pour beaucoup à la puissance d'expression du langage offert par le Shell. Mais c'est à travers la construction de pipe-line que le Shell met en valeur un des apports fondamentaux d'Unix qu'est la notion de *tube*.

En permettant à l'utilisateur de connecter *dynamiquement* plusieurs programmes par l'intermédiaire de leurs flots d'entrée-sortie standard, le Shell induit en fait une méthode de développement de logiciel spécifique à Unix [Lucas 84] où la conception d'un programme s'appuie sur le schéma fonctionnel suivant :

- lecture des données sur le flot d'entrée standard ;
- traitement des données lues ;
- écriture des résultats sur le flot de sortie standard.

Dans la terminologie Unix, de tels programmes sont appelés des *filtres*.

Cette approche privilégie la conception de programmes qui tout en étant indépendants peuvent être associés ultérieurement à d'autres pour la réalisation d'une tâche quelconque [Pike 84a]. Elle est particulièrement adaptée aux applications manipulant des données textuelles. Dans ce cas, le traitement peut souvent être réalisé selon un schéma d'exécution de type pipe-line construit à partir de programmes existant, du moins pour une partie d'entre eux.

4. Evolution du système Unix

Depuis la commercialisation d'Unix version 7, le noyau du système a subi plusieurs modifications destinées à pallier certaines de ses insuffisances [Ritchie 78] [Broze 82], la plus importante concernant les possibilités de communication entre processus ou *IPC* (Inter Process Communication). Deux principales versions, System V d'AT&T et 4.2 BSD (Berkeley System Distribution), reflètent actuellement cette évolution.

System V offre trois mécanismes d'IPC : la mémoire partagée, les sémaphores pour la synchronisation et les files d'attente de messages qui, contrairement aux tubes, sont manipulées par de nouvelles primitives système.

La version 4.2 BSD développée à l'Université de Berkeley pour le Département de la Défense américain propose un nouveau concept d'IPC, le *socket*. Après avoir été créé, un *socket* peut être nommé dans le système de fichiers et donc être accessible à tout processus. Cette version offre également une primitive de lecture qui permet à un programme de se mettre en attente sur plusieurs flots d'entrée.

C'est également dans cette version qu'a été introduite sur la gamme des machines Vax de DEC la gestion de l'espace virtuel des processus par le mécanisme de pagination.

Plus récemment, A&TT propose dans la version 8 d'Unix un nouveau type d'objet de connection appelé *I/O stream* qui permet de faire communiquer en « full-duplex » un processus avec un périphérique ou un autre processus à l'aide des primitives classiques *read* et *write* [Ritchie 84].

Bien que l'existence de plusieurs versions d'un même système constitue un handicap à sa diffusion, Unix connait depuis plusieurs années un réel succès commercial que l'on peut attribuer aux facteurs suivants :

- la grande portabilité du système [Bodenstab 84] : la majeure partie du noyau est écrite en « C », ainsi que le Shell et l'ensemble des utilitaires standards ;

- sa simplicité qui fait qu'Unix est adapté à la puissance des microprocesseurs 16/32 bits actuels comme ceux de la famille Motorola ou Intel: la taille du noyau est relativement modeste (de l'ordre de 100 K octets) et son fonctionnement nécessite uniquement la présence matérielle d'une horloge et d'un mécanisme de réimplantation dynamique d'adresses ;

- la puissance et la souplesse de son langage de commande ;

- la présence de nombreux logiciels d'application qui permet aux constructeurs choisissant Unix d'offrir rapidement et à moindre coût un environnement particulièrement riche à leurs clients.

La récente adoption d'Unix par IBM pour sa nouvelle machine PC/RT constitue d'ailleurs une sorte de consécration en la matière.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[Adobe 85]

PostScript Language Manual
Adobe Systems Palo Alto (1985)

[Apollo 85]

DOMAIN System User's Guide
Apollo Computer Inc. (July 1985)

[Bennet 85]

Bennet J.
Raster Operations
BYTE, Vol. 10, No. 12 (November 1985), pp. 187-203

[Bly 86]

Bly S. A., Rosenberg J. K.
A Comparison of Tiled and Overlapping Windows
Proceedings of the ACM/SIGCHI conference (April 1986), pp. 101-106

[Bodenstab 84]

Bodenstab D. E., Houghton T. F., Kelleman K. A., Ronkin G., Schan E. P.
UNIX Operating System Porting Experiences
AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2
(October 1984), pp. 1769-1790

[Boule 85]

Boule I., Joloboff V.
Un gestionnaire de fenêtres sous le système SMX
Actes des journées SM90, Eyrolles (Décembre 1985), pp. 397-411

[Brown 83]

Brown M., Meyrowitz N., van Dam A.
Personnal Computer Networks and Graphical Animation: Rationale and Practice for Education
ACM-SIGCSE, Vol. 15, No. 1 (February 1983), pp. 296-307

[Broze 82]

Broze H., Bruffaerts A., Declerfayt M. F., Henin E., Jamart P., Lobelle M., Milgrom E., Verbaeten P., Willems Y. D.
Evaluation critique du système UNIX
Technique et Science Informatique, Vol. 1, No. 4 (1982), pp. 315-324

[Cany 85]

Cany G.
APOTRE : Un système d'accès à un poste de travail sous Unix
Actes des journées SM90, Eyrolles (Décembre 1986), pp. 412-419

[Cohen 85]

Cohen E. S.
Constraint-Based Tiled Windows
Proceedings of the 1st IEEE Computer Society Conference on Computer Workstations (November 1985), pp. 2-11

[Coutaz 85]

Coutaz J.
Abstraction for User Interface Design
IEEE Computer, Vol 18. No 9 (September 1985), pp. 21-34

[Foley 82]

Foley J. D., van Dam A.
Fundamentals of Interactive Computer Graphics
Addison Wesley (1982)

[Gancarz 86]

Gancarz M.
Uwm: A User Interface for X Windows
Proceedings of the Usenix Summer Conference
Atlanta (June 1986), pp. 429-440

[Gaylin 86]

Gaylin K. B.
How are Windows Used? Some Notes on Creating an Empirically-Based Windowing Benchmark Task
Proceedings of the ACM/SIGCHI conference (April 1986), pp. 96-100

[Gittins 84]

Gittins D. T., Winder R. L., Bez H. E.
An Icon-Driven End-User Interface to Unix
International Journal of Man-Machine Studies, Vol. 21, No. 5
(November 1984), pp. 451-461

[Goodfellow 85]

Goodfellow M. J.
WHIM, The Window Handler and Input Manager
Proceedings of the 1st IEEE Computer Society Conference on Computer Workstations (November 1985), pp. 12-21

[Gosling 84]

Gosling J. A., Rosenthal D. S. H.
A Network Window-Manager
Technical report, Information Technical Center
Carnegie-Mellon University (1984)

[Gosling 86]

Gosling J. A.
SunDew - A Distributed and Extensible Window System
Methodology of Window Management, Springer-Verlag (1986), pp. 47-57

[Gupta 81]

Gupta S.
Architectures and Algorithms for Parallel Updates of Raster Scan Displays
PhD thesis, Computer Science Department,
Carnegie-Mellon University (December 1981)

[Hayes 85]

Hayes P. J., Szekely P. A., Lerner R. A.
Design Alternatives for User Interface Management Systems Based on Experience with Cousin
Proceedings of ACM/CHI conference on Human Factors in Computing Systems, San Francisco (April 1985), pp. 169-175

[Ingalls 78]

Ingalls D. H.
The Smalltalk-76 Programming System Design and Implementation
Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages, (January 1978) pp. 9-16

[Jacob 82]

Jacob R. J. K.
A Window Manager for UNIX
Computer Science and Systems Branch
Naval Research Laboratory, Washington (September 82)

[Joloboff 84]

Joloboff V.
Aspects logiciels de la communication homme-machine sur les postes de travail individuels
Rapport de Recherche TIGRE, No. 17 (Juin 1984)
Centre de Recherches Bull Grenoble

[Kernighan 78]

Kernighan B. W., Ritchie D. M.
The C Programming Language
Prentice-Hall (1978)

[Krakowiak 85]

Krakowiak S.
principes des systèmes d'exploitation des ordinateurs
Dunod informatique (1985)

[Lemmons 83a]

Lemmons P.
Microsoft Windows
BYTE, Vol. 8, No. 12 (December 1983), pp. 48-54

[Lemmons 83b]

Lemmons P.
A Guided Tour of Visi On
BYTE, Vol. 8, No. 6 (June 1983), pp. 256-277

[Lipkie 82]

Lipkie D. E., Evans S. R., Newlin J. K., Weissman R. L.
Star Graphics: An Object-Oriented Implementation
ACM Computer Graphics, Vol. 4, No. 3 (1982), pp. 115-124

[Lucas 84]

Lucas H., Martin B., de Sablet G.
UNIX : mécanismes de base, langage de commande, utilisation
Collection Informatique et Entreprise, Eyrolles (1984)

[Marcus 84]

Marcus A.
Corporate Identity for Iconic Interface Design: The Graphic Design Perspective
IEEE Computer Graphics and Applications, Vol. 4, No. 12
(December 1984), pp. 24-32

[Mitchell 79]

Mitchell J. G., Maybury W., Sweet R.
Mesa Language Manual
Xerox Palo Alto Research Center, Report CSL-79-3 (1979)

[Meyrowitz 81]

Meyrowitz N., Moser M.
BRUWIN: An adaptable design strategie for Window Manager / virtual terminal systems
Proceedings of the 8th Symposium on Operating Systems Principles
ACM Operating Systems Rewiew, Vol. 15, No. 5
(December 1981), pp. 180-189

[Morris 86]

Morris J. H., Satyanarayanan M., Conner H., Howard J. H.,
Rosenthal D. S. H., Smith F. D.
Andrew: A Distributed Personal Computing Environment
Communications of the ACM, Vol. 29, No. 3 (March 1986), pp. 184-201

[Myers 84]

Myers B. A
The User Interface for Sapphire
IEEE Computer Graphics and Applications, Vol. 4, No. 12
(December 1984), pp. 13-23

[Myers 85]

Myers B. A.
A complete and Efficient Implementation of Covered Windows
IEEE Computer, Vol. 19, No. 9 (September 1986), pp. 57-67

[Newman 79]

Newman W. M., Sproull R. F.
Principles of Interactive Computer Graphics
McGraw-Hill (1979)

[Perq 84]

Perq : Guide to PNX
PERQ Systems Corporation, Pittsburg (1984)

[Pike 83]

Pike R.

Graphics in Overlapping Bitmap Layers

ACM Transactions on Graphics, Vol. 2, No. 2 (April 1983), pp. 135-160

[Pike 84a]

Pike R., Kernighan B. W.

Program Design in the UNIX System Environment

AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2

(October 1984), pp. 1595-1605

[Pike 84b]

Pike R.

The Blit: A Multiplexed Graphics Terminal

AT&T Bell Laboratories Technical Journal Vol. 63, No. 8, Part 2

(October 1984), pp. 1607-1631

[Quint 86]

Quint V., Vatton I.

Grif: An Interactive System for Structured Document Manipulation

Proceedings of the International Conference on Text Processing and Document Manipulation

Cambridge University Press (April 1986), pp. 200-213

[Ritchie 78]

Ritchie D. M.

UNIX Time-Sharing System : A Retrospective

The Bell System Technical Journal, Vol. 57, No. 6, Part 2

(July-August 1978), pp. 1947-1969

[Ritchie 84]

Ritchie D. M.

A Stream Input-Output System

AT&T Bell Laboratories Technical Journal, Vol. 63 No. 8, Part 2

(October 1984), pp. 1897-1910

[Rosenthal 83]

Rosenthal D. S. H., Yen A.
User Interface Models Summary
in Graphical Input Interaction Technique Workshop Summary
ACM/SIGGRAPH Computer Graphics, Vol. 17, No. 1
(January 1983), pp. 5-29

[Rosenthal 86]

Rosenthal D. S. H.
Window System Implementations
Tutorial Materials, Usenix Technical Conference, Denver (January 1986), pp. 24-27

[Satyanarayanan 84]

Satyanarayanan M.
The ITC Project: An Experiment in Large-Scale Distributed Personal Computing
Carnegie-Mellon University (October 1984)

[Smith 82]

Smith D. C., Irby C., Kimball R., Verplank B., Harslem E.
Designing the Star User Interface
BYTE, Vol. 7, No. 4 (April 1982), pp. 242-282

[Stock 86]

Stock R., Robertson B.
New Chips Unleash Super Graphics
Computer Graphics World, Vol. 9, No. 6 (June 1986), pp. 24-32

[Sun 85]

Programmer's Reference Manual for the SUN Window System
SUN Microsystems (1985)

[Sweetman 86]

Sweetman D.
A Modular Window System for Unix
Methodology of Window Management, Springer-Verlag (1986), pp. 73-79

[Teitelman 81]

Teitelman W., Masinter L.
The Interlisp programming Environment
IEEE Computer, Vol. 14, No. 4 (April 1981), pp. 25-33

[Teitelman 85]

Teitelman W.
A Tour Through Cedar
IEEE Transactions on Software Engineering, Vol. SE-11, No. 3
(March 1985), pp 285-302

[Teitelman 86]

Teitelman W.
Ten Years of Window Systems - A Retrospective View
Methodology of Window Management, Springer-Verlag (1986), pp. 41-44

[Tesler 81]

The Smalltalk Environment
BYTE, Vol. 6, No. 8 (August 1981), pp. 90-147

[Thacker 79]

Thacker C. P., McCreight E. M., Lampson B. W., Sproull R. F., Boggs D.
Alto: A Personal Computer
Xerox Palo Alto Research Center, Report CSL-79-11 (August 79)

[Thompson 74]

Ritchie D. M , Thompson K.
The UNIX Time-Sharing System
Communications of the ACM, Vol. 17, No. 7 (July 1974), pp. 365-375

[Tigre 83]

Présentation générale du projet TIGRE
Rapport de Recherche TIGRE, No. 1 (Janvier 1983)
Centre de Recherches Bull Grenoble

[Van Der Linden 85]

Van Der Linden P.
Un système de fenêtres sur la SM90
Actes des journées SM90, Eyrolles (Décembre 1986), pp. 386-396

[Wegmann 83]

Wegmann A.
VITRAIL : Conception et réalisation d'un noyau de communication homme-machine pour un système bureautique intégré
These de Docteur-Ingénieur Université, Paris VI (Septembre 1983)

[Weinreb 81]

Weinreb D., Moon D.
Introduction to using the window system
Symbolics Inc. (1981)

[Weiser 83]

Weiser M., Torek C., Trigg R., Wood R.
The Maryland Window System
Technical Report 1271, Computer Science Department
University of Maryland (1983)

[Williams 83]

Williams G.
The Lisa Computer System
BYTE, Vol. 8, No. 2 (February 1983), pp. 33-50

[Williams 84]

Williams G.
The Apple Macintosh Computer
BYTE, Vol. 9, No. 2 (February 1984), pp. 30-54

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de

. J.P BANATRE, Professeur
. J.F ABRAMATIC, Directeur de recherche

Monsieur BOULE Yvan

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité Informatique.

Fait à Grenoble, le 10 avril 1987

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,

RESUME

Nous dégageons et analysons dans cette thèse les principaux concepts des systèmes multi-fenêtres. Une place importante est consacrée aux aspects « système d'exploitation » des gestionnaires de fenêtres. Nous nous intéressons en particulier aux problèmes que posent la prise en compte du parallélisme et la gestion des ressources partagées dans la conception d'un système multi-fenêtres.

Nous présentons ensuite le système multi-fenêtres Fenix développé sur Unix dans le cadre du projet TIGRE. Le système Fenix permet l'exécution parallèle de plusieurs applications interactives disposant chacune d'un nombre quelconque de fenêtres pour communiquer avec l'utilisateur. Il offre également un ensemble de mécanismes destinés à faciliter la conception d'outils de dialogue évolués.

Fenix gère le recouvrement des fenêtres sur l'écran de façon totalement transparente aux applications. Le recouvrement des fenêtres est géré à l'aide d'une méthode originale qui rend les opérations de manipulation [de fenêtres] particulièrement performantes et, sur le plan visuel, très confortables pour l'utilisateur.

Le système Fenix est conçu pour que chaque utilisateur puisse adapter l'interface du système au style de communication qui lui convient. L'architecture de Fenix repose pour cela sur une séparation stricte entre, d'une part la partie opératoire regroupant les fonctions de base du système (manipulation des fenêtres, primitives graphiques, suivi de la souris sur l'écran, etc.) et, d'autre part, la partie contrôle qui gère les commandes mises à la disposition de l'utilisateur.

MOTS CLES

Systèmes multi-fenêtres, interface utilisateur, poste de travail individuel, Unix, systèmes d'exploitation.